8. Bottomup-Syntaxanalyse

Hanspeter Mössenböck, Institut für Systemsoftware, JKU

Arbeitsweise eines Bottomup-Parsers

Beispiel

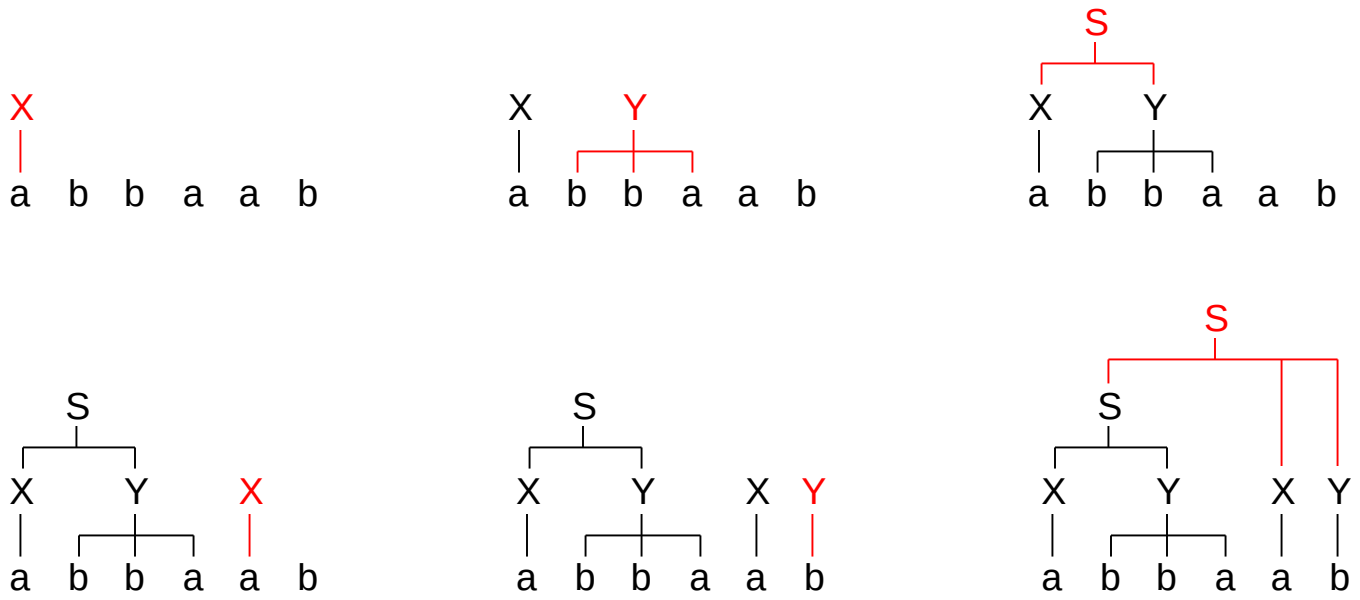
```
S = X Y | S X Y.
X = a | a a b.
Y = b | b b a.
```

nicht LL(1)!

- für Topdown-Analyse ungeeignet
- kein Problem für Bottomup-Analyse

Eingabesatz: a b b a a b

Syntaxbaum wird bottomup aufgebaut



Erkennung mit Hilfe eines Kellers

Grammatik

```
S = X Y | S X Y.
X = a | a a b.
Y = b | b b a.
```

Eingabesatz

a b b a a b # (# ... eof-Symbol)

Keller	*Eingabe*	*Aktion*
	a b b a a b #	lies
a	b b a a b #	reduziere *a* zu *X*
X	b b a a b #	lies
X b	b a a b #	lies (nicht reduzieren, weil sonst Sackgasse!)
X b b	a a b #	lies
X b b a	a b #	reduziere *b b a* zu *Y*
X Y	a b #	reduziere *X Y* zu *S*
S	a b #	lies
S a	b #	reduziere *a* zu *X*
S X	b #	lies
S X b	#	reduziere *b* zu *Y*
S X Y	#	reduziere *S X Y* zu *S*
S	#	Satz erkannt (Keller enthält Startsymbol, Eingabe leer)

Erkennung mit Hilfe eines Kellers

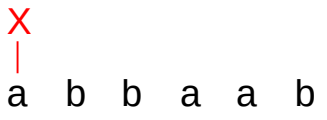
Vier Analyseaktionen

Shift	Lies und kellere nächstes Eingabesymbol (TS)
Reduce	Reduziere Kellerende zu einem NTS
Accept	Satz erkannt (nur bei *S* . #)
Error	Analyse kann nicht weiter ($\Rightarrow$ Fehlerbehandlung)

Bottomup-Parser heißen daher

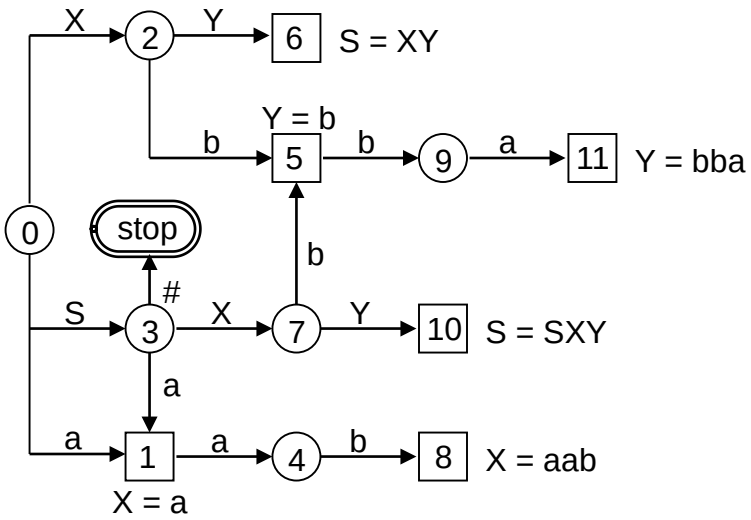
- **Shift-Reduce-Parser**
- **LR-Parser**
  Erkennung von **L**inks nach rechts mit **R**echtskanonischen Ableitungen

  Rechtskanonische Ableitung = linkskanonische Reduktion
  d.h. die linkeste einfache Phrase (der *Ansatz*) wird reduziert



Parser als Kellerautomat (PDA)

Kellerautomat



Grammatik

1 S = X Y.
2 S = S X Y.
3 X = a.
4 X = a a b.
5 Y = b.
6 Y = b b a.

nur reine BNF möglich!

Zustandsübergangstabelle (Parsertabelle)

	TS			NTS		
	a	b	#	S	X	Y
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

s1 ... Shift in Zustand 1
r3 ... reduziere nach Produktion 3
- ... Fehler

Erkennung mit Shift-Reduce-Aktionen

	a	b	#	S	X	Y
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

1	S = X Y.
2	S = S X Y.
3	X = a.
4	X = a a b
5	Y = b
6	Y = b b a

Erkennung von a b b a a b #

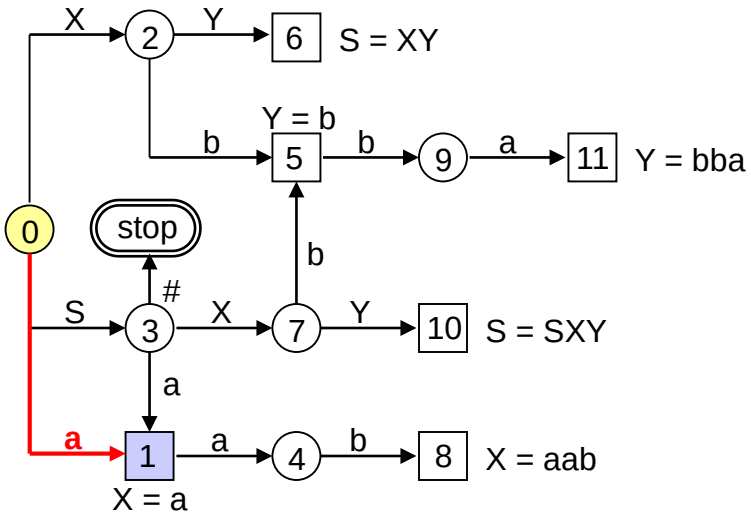
Keller		*Eingabe*	*Aktion*
0		a b b a a b #	s1
0 1		b b a a b #	r3 (X=a)
0	X	b b a a b #	s2 (shift mit X!)
0 2		b b a a b #	s5
0 2 5		b a a b #	s9
0 2 5 9		a a b #	s11
0 2 5 9 11		a b #	r6 (Y=bba)
0 2	Y	a b #	s6
0 2 6		a b #	r1 (S=XY)
0	S	a b #	s3
0 3		a b #	s1
0 3 1		b #	r3 (X=a)
0 3	X	b #	s7
0 3 7		b #	s5
0 3 7 5		#	r5 (Y=b)
0 3 7	Y	#	s10
0 3 7 10		#	r2 (S=SXY)
0	S	#	s3
0 3		#	acc

r3 ... reduziere nach Produktion 3 (X = a)

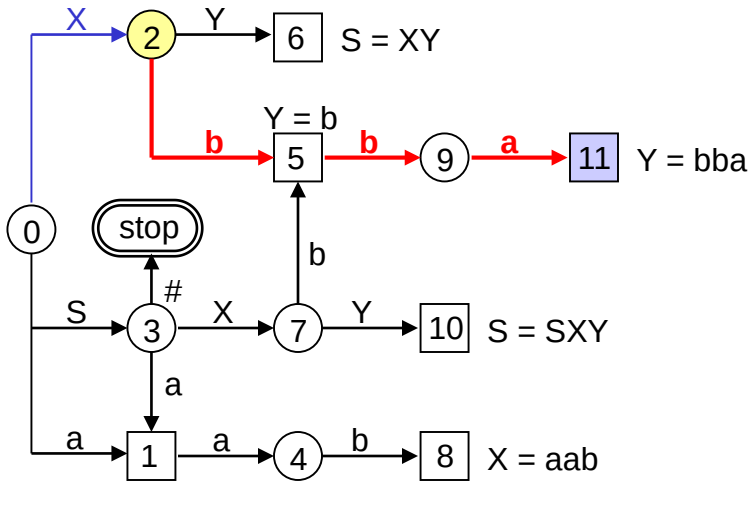
- 1 Zustand auskellern (weil rechte Seite 1 Symbol lang)
- linke Seite (*X*) in die Eingabe einfügen
- Nach reduce erfolgt immer ein shift mit einem NTS

Simulation am Kellerautomaten

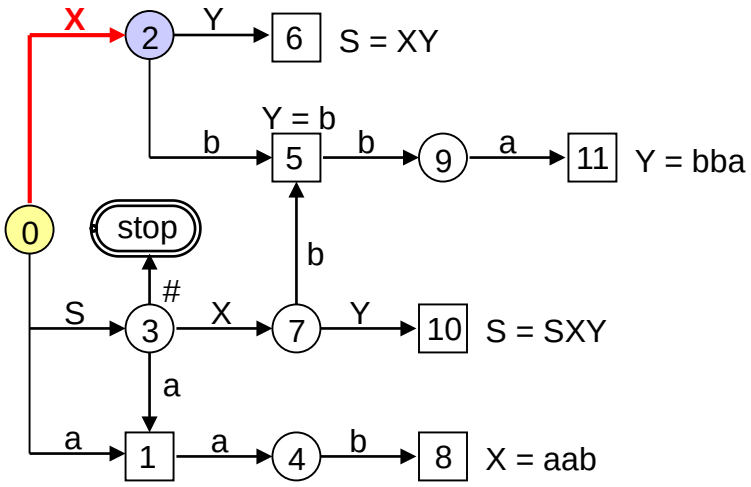
a b b a a b #



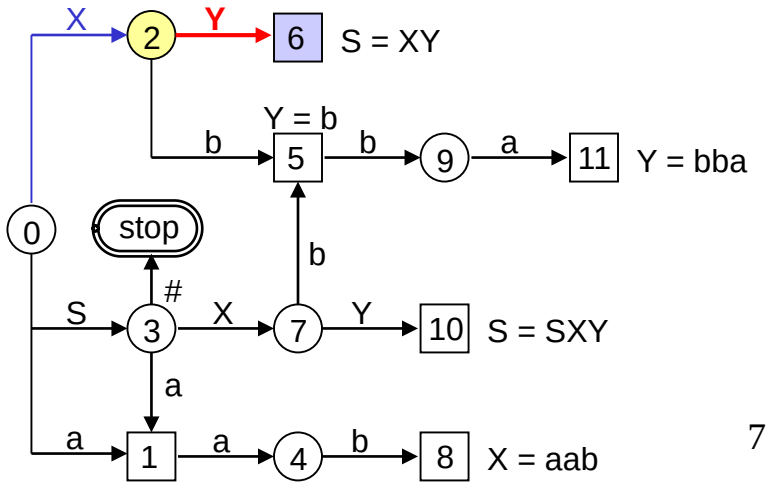
b b a a b #



X b b a a b #

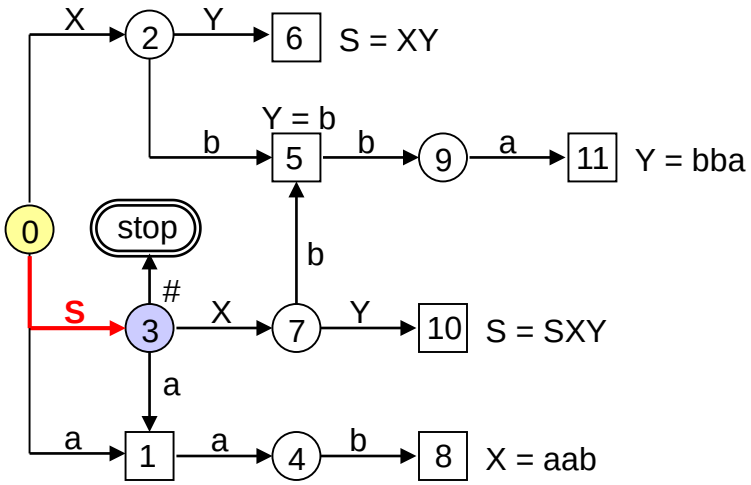


Y a b #

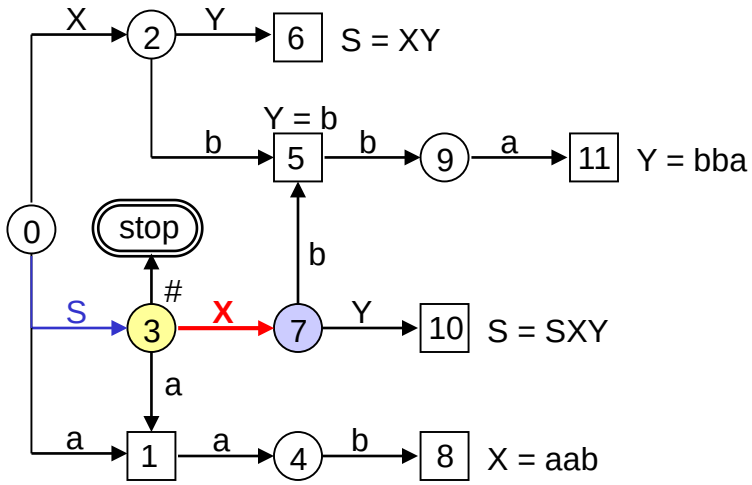


Simulation am Kellerautomaten

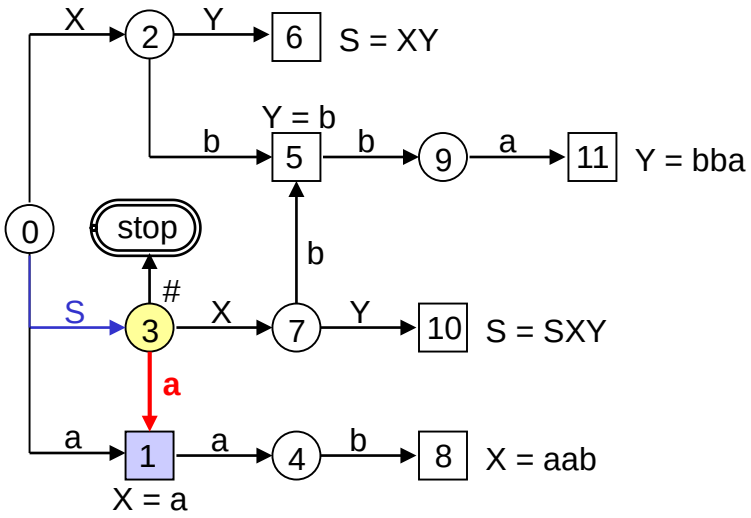
S a b #



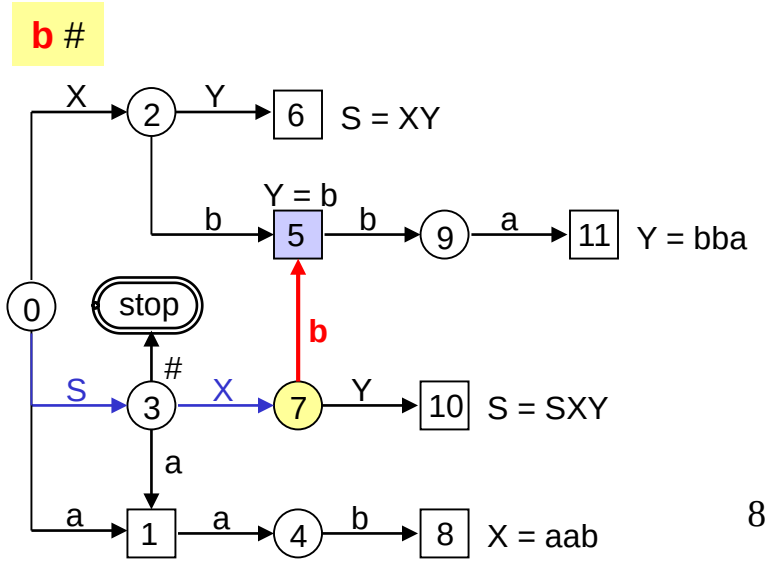
X b #



a b #

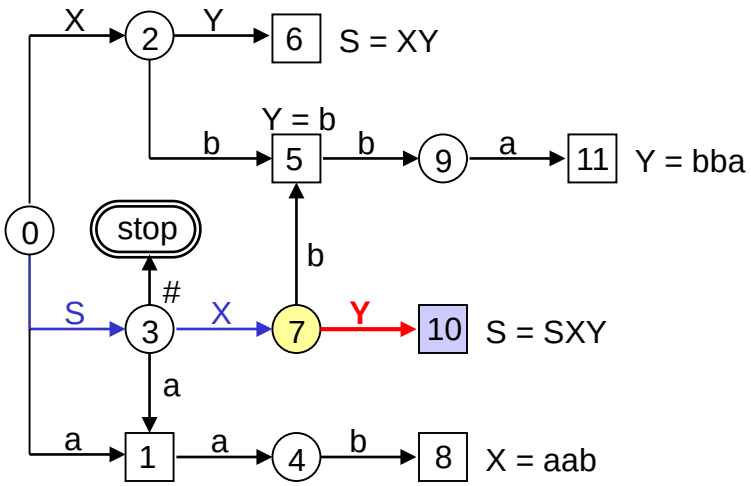


b #

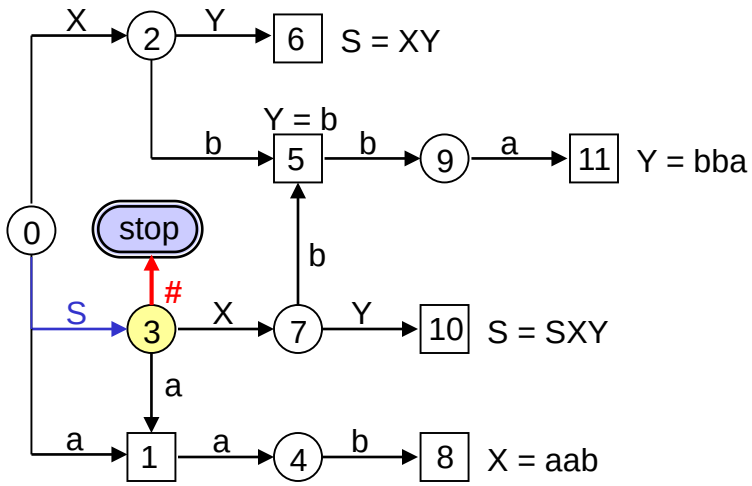


Simulation am Kellerautomaten

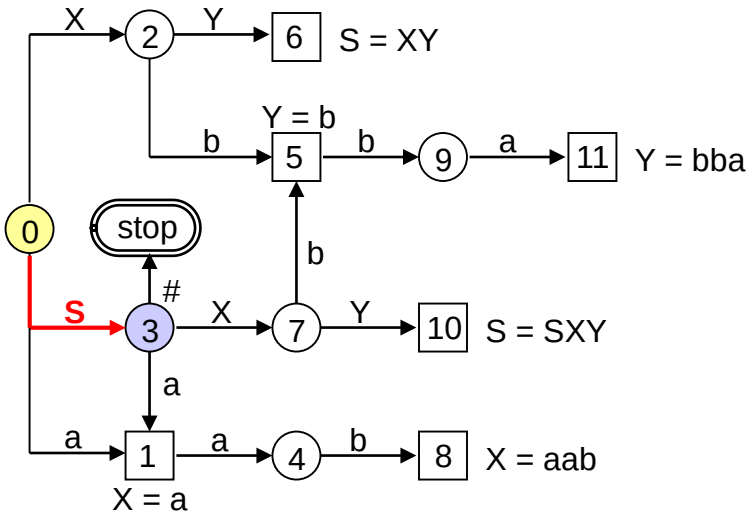
Y #



#



S #



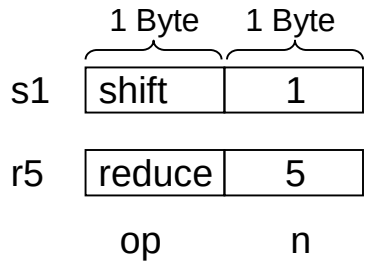
Implementierung des LR-Parsers

```
void parse() {
   short[][]   action = { {...}, {...}, ...};   // state transition table
   byte[]      length = { ... };                // production lengths
   byte[]      leftSide = { ... };              // left side NTS of every production
   int         state;                           // current state
   int         sym;                             // next input symbol
   int         op, n, a;

   clearStack();
   state = 0; sym = next();
   for (;;) {
      push(state);
      a = action[state][sym]; op = a / 256; n = a % 256;
      switch (op) {
         case shift:       // shift n
            state = n; sym = next(); break;
         case reduce:      // reduce n
            for (int i = 0; i < length[n]; i++) pop();
            a = action[top()][leftSide[n]]; n = a % 256;  // shift n
            state = n; break;
         case accept:
            return;
         case error:
            System.exit(1);  // error handling is still missing
      }
   }
}
```

Tabellengesteuertes Programm für beliebige Grammatiken

action-Eintrag (2 Bytes)




```
          ...
0 3 7 10        #   r2 (S=SXY)
      0   S     #   s3
```

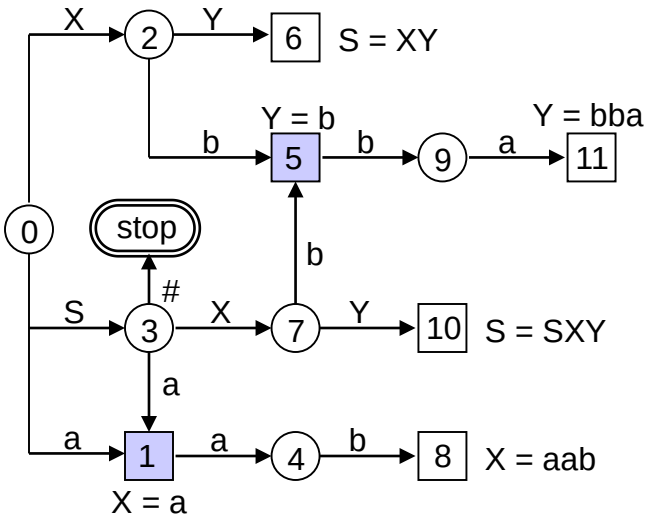
8. Bottomup-Syntaxanalyse

LR(0)- und LR(1)-Grammatiken

LR(0) Erkennbar von **L**inks nach rechts
mit **R**echtskanonischen Ableitungen
und **0** Vorgriffssymbolen

Eine Grammatik ist LR(0)

- wenn es keinen Reduce-Zustand gibt, aus dem auch ein Shift weggeht
- wenn in jedem Reduce-Zustand nur nach einer einzigen Produktion reduziert werden kann



Diese Grammatik ist nicht LR(0)!

LR(1) Erkennbar von **L**inks nach rechts
mit **R**echtskanonischen Ableitungen
und **1** Vorgriffssymbol

Eine Grammatik ist LR(1), wenn in jedem Zustand mit 1 Vorgriffssymbol entscheidbar ist

- ob Shift oder Reduce ausgeführt werden soll
- zu welchem NTS reduziert werden soll

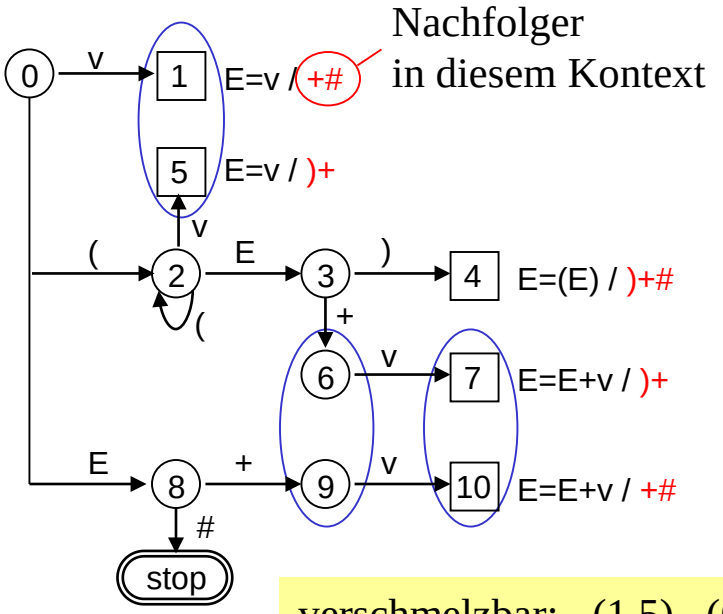
LALR(1)-Grammatiken

Lookahead-LR(1)

- Teilmenge der LR(1)-Grammatiken
- Haben kleinere Tabellen als LR(1)-Grammatiken, weil Zustände mit gleichen Aktionen aber unterschiedlichen Vorgriffssymbolen verschmolzen werden können.

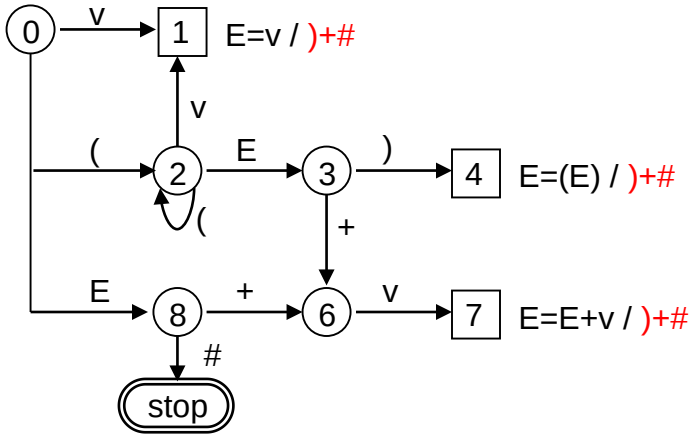
Beispiel E = v | E "+" v | "(" E ")"

LR(1)-Tabellen



verschmelzbar: (1,5) (6,9) (7,10)

LALR(1)-Tabellen



Grammatik ist LALR(1),
wenn durch diese Verschmelzung keine
LR(1)-Konflikte entstehen

LR-Analyse versus rekursiver Abstieg

Vorteile

- LALR(1) ist mächtiger als LL(1)
  - linksrekursive Produktionen erlaubt
  - Alternativen mit gleichen terminalen Anfängen erlaubt
- LR-Parser sind kompakter als Parser im rekursiven Abstieg
  (Tabellen benötigen allerdings auch viel Speicher)
- Tabellengesteuerter Parser ist universeller Algorithmus, der mit Tabellen parametrisiert wird
- Tabellengesteuerter Parser erlaubt bessere Fehlerbehandlung

Nachteile

- LALR(1)-Tabellen sind manuell schwer zu konstruieren (man braucht Werkzeuge)
- LR-Parser sind etwas langsamer als RD-Parser
- Semantikanschluss ist beim LR-Parser komplizierter
- Die Analyse ist beim LR-Parser schwer verfolgbar
- LR-Parser benötigen BNF statt EBNF

LR-Parser lohnen sich nur für komplexe Sprachen.
Für kleinere Sprachen (z.B. Kommandosprachen) ist rekursiver Abstieg besser.

8. Bottomup-Syntaxanalyse

LR-Items

Beispiel: Analyse von

```
S = a X b
X = c
```

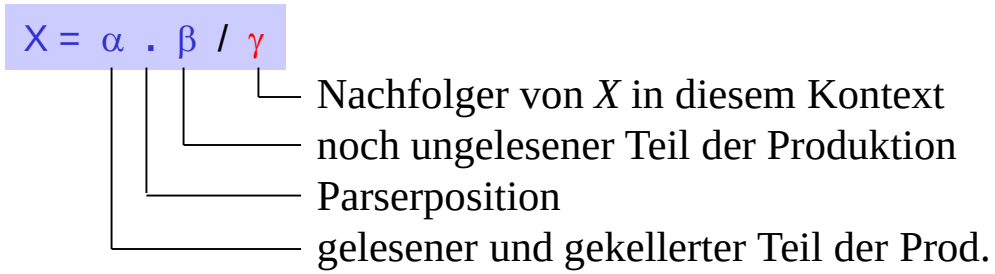
Parser (gekennzeichnet durch Punkt) **bewegt sich durch die Produktion**

```
S = . a X b
S = a . X b   ←
X = . c
X = c .
S = a X . b
S = a X b .
```

wenn der Parser vor *X* steht, steht er gleichzeitig vor der rechten Seite der *X*-Produktion

LR-Item

Schnappschuss der Syntaxanalyse



Steuersymbol

Symbol, das auf den Punkt folgt.
z.B.:

X = a . a b / c	Steuersymbol = a
X = a a b . / c	Steuersymbol = c

Shift-Item	X = α . β / γ	Punkt steht nicht am Regelende
Reduce-Item	X = α . / γ	Punkt steht am Regelende

Parserzustand als Itemmenge

Zustand des Parsers

Repräsentiert durch die Menge der Items, an denen der Parser gerade arbeitet

```
S = X . Y c     / #
Y = . b         / c
Y = . b b a     / c
```

Topdown-Parser arbeitet immer nur an *einer* Produktion.
Bottomup-Parser kann an *mehreren* Produktionen *gleichzeitig* arbeiten.

```
X = . a a b
X = . a a c
```
lies *a* →
```
X = a . a b
X = a . a c
```
lies *a* →
```
X = a a . b
X = a a . c
```

erst jetzt werden die beiden Produktionen an Hand des Vorgriffs-symbols unterschieden

Bottomup-Parser sind daher mächtiger als Topdown-Parser.

Kern und Hülle eines Zustands

Kern

Alle Items eines Zustands, die nicht mit Punkt beginnen (außer in Produktion des Startsymbols)

```
S = X . Y c / #
```

Aus dem Kern lassen sich alle anderen Items des Zustands ableiten

Hülle

```
foreach (item in state) {
   if (item ist von der Art X = α . Y β / γ)
      füge alle Produktionen Y = . ω / First(βγ) als Items zum Zustand hinzu;
}
```

Beispiel

```
S = X . Y c    / #
Y = . b        / c
Y = . b b a    / c
```

Kern

Hülle

```
S = X Y c
X = a
X = a a b
Y = b
Y = b b a
```

Beispiel: Berechnung der Hülle

Grammatik

```
0   S' = S #
1   S = X Y
2   S = S X Y
3   X = a
4   X = a a b
5   Y = b
6   Y = b b a
```

Kern

```
S' = . S #
```

Füge alle *S*-Produktionen hinzu

```
S' = . S #
S = . X Y     / #
S = . S X Y   / #
```

Füge alle *X*-Produktionen hinzu

```
S' = . S #
S = . X Y     / #
S = . S X Y   / #
X = . a       / b
X = . a a b   / b
```

Füge alle *S*-Produktionen hinzu (wegen S = . S X Y / #)
Nachfolger von *S* ist hier *a*

```
S' = . S #
S = . X Y     / #a
S = . S X Y   / #a
X = . a       / b
X = . a a b   / b
```

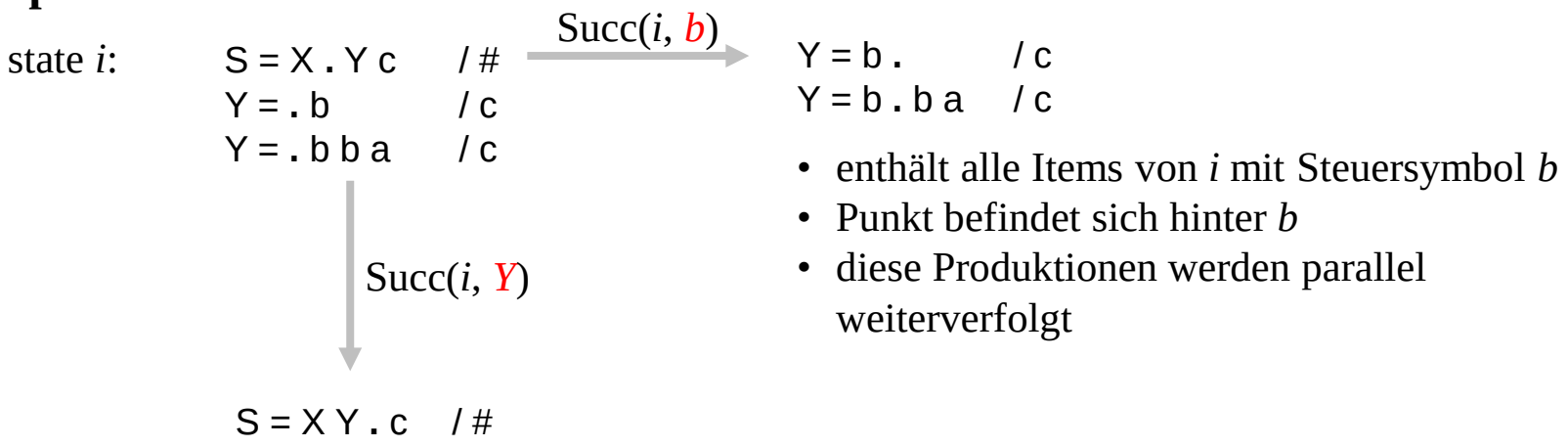
} Hülle

Folgezustand

Succ(state, sym)

Itemmenge des Zustands, in dem man aus *state* mit *shift sym* kommt (*sym* = Steuersymbol)

Beispiel



- enthält alle Items von *i* mit Steuersymbol *b*
- Punkt befindet sich hinter *b*
- diese Produktionen werden parallel weiterverfolgt

Succ(state, sym) ist nur der Kern des neuen Zustands, der erst zur Hülle erweitert werden muss

LALR(1)-Tabellenerzeugung

Erweitere Grammatik um Pseudoproduktion S' = S #
Erzeuge Zustand 0 mit Kern S' = **.** S # // Ausnahme: einziger Kern mit Punkt am Anfang

```
while (noch nicht alle Zustände betrachtet) {
   s = nächster noch nicht betrachteter Zustand;
   Bilde Hülle von s;
   for (all Items of s) {
      switch (Item-Art) {
         case S' = S . # :        erzeuge accept #; break;
         case X = α . y β / γ :   erzeuge Hilfszustand s1 = Succ(s, y);
                                  if (∃ s2: Kern(s1) == Kern(s2)) {
                                     mische Vorgriffssymbole von s1 zu s2 hinzu;
                                     s1 = s2;
                                  } else nimm s1 als neuen Zustand auf;
                                  erzeuge shift y, s1; break;
         case X = α . / γ :       erzeuge reduce γ, (X = α); break;
      }
   }
   // alle nicht definierten Übergänge sind error-Aktionen
}
```

Die Kerne werden ohne Nachfolgersymbole verglichen, z.B.:

s2		s1		s2'
E = v . / #+	+	E = v . /)+	⇒	E = v . /)#+

LALR(1)-Tabellenerzeugung

Grammatik

0	S' = S #
1	S = X Y
2	S = S X Y
3	X = a
4	X = a a b
5	Y = b
6	Y = b b a

0	S' = . S #		shift	a	1
	S = . X Y	/ #a	shift	X	2
	S = . S X Y	/ #a	shift	S	3
	X = . a	/ b			
	X = . a a b	/ b			
1	X = a .	/ b	red	b	3
	X = a . a b	/ b	shift	a	4
2	S = X . Y	/ #a	shift	b	5
	Y = . b	/ #a	shift	Y	6
	Y = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . X Y	/ #a	shift	a	1 (!)
	X = . a	/ b	shift	X	7
	X = . a a b	/ b			
4	X = a a . b	/ b	shift	b	8
5	Y = b .	/ #a	red	#,a	5
	Y = b . b a	/ #a	shift	b	9
6	S = X Y .	/ #a	red	#,a	1
7	S = S X . Y	/ #a	shift	b	5
	Y = . b	/ #a	shift	Y	10
	Y = . b b a	/ #a			
8	X = a a b .	/ b	red	b	4
9	Y = b b . a	/ #a	shift	a	11
10	S = S X Y .	/ #a	red	#,a	2
11	Y = b b a .	/ #a	red	#,a	6

Parser-Tabelle

0	S' = . S #		shift	a	1
	S = . X Y	/ #a	shift	X	2
	S = . S X Y	/ #a	shift	S	3
	X = . a	/ b			
	X = . a a b	/ b			
1	X = a .	/ b	red	b	3
	X = a . a b	/ b	shift	a	4
2	S = X . Y	/ #a	shift	b	5
	Y = . b	/ #a	shift	Y	6
	Y = . b b a	/ #a			
3	S' = S . #		acc	#	
	S = S . X Y	/ #a	shift	a	1
	X = . a	/ b	shift	X	7
	X = . a a b	/ b			
4	X = a a . b	/ b	shift	b	8
5	Y = b .	/ #a	red	#,a	5
	Y = b . b a	/ #a	shift	b	9
6	S = X Y .	/ #a	red	#,a	1
7	S = S X . Y	/ #a	shift	b	5
	Y = . b	/ #a	shift	Y	10
	Y = . b b a	/ #a			
8	X = a a b .	/ b	red	b	4
9	Y = b b . a	/ #a	shift	a	11
10	S = S X Y .	/ #a	red	#,a	2
11	Y = b b a .	/ #a	red	#,a	6

Zustandsübergangstabelle

	a	b	#	S	X	Y
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Parser-Tabelle in Listenform

	a	b	#	S	X	Y
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Fehlereinträge bei NT-Aktionen können eigentlich nie vorkommen

	TS-Aktionen		*NTS-Aktionen*	
0	a	s1	S	s3
	*	error	X	s2
1	a	s4		
	*	r3		
2	b	s5	Y	s6
	*	error		
3	a	s1	X	s7
	#	acc		
	*	error		
4	b	s8		
	*	error		
5	b	s9		
	*	r5		
6	*	r1		
7	b	s5	Y	s10
	*	error		
8	*	r4		
9	a	s11		
	*	error		
10	*	r2		
11	*	r6		

- Aktionen in jedem Zustand werden seq. geprüft
- letzte T-Aktion jedes Zustands ist * *error* oder * *r* (falls fälschlicherweise reduziert wird, wird der Fehler beim nächsten shift bemerkt)
- kompakter aber langsamer als mit einer Tabelle

LALR(1)- versus LR(1)-Tabellen

LR(1) ist etwas mächtiger als LALR(1), hat aber etwa 10 x größere Tabellen

LR(1)-Tabellenerzeugung

- Zustände werden nie verschmolzen
- Grammatik ist LR(1), wenn in keinem Zustand einer der folgenden Konflikte auftritt:

shift-reduce-Konflikt	shift a ... red a ...	Parser kann sich mit 1 Symbol Vorgriff nicht entscheiden, ob er lesen oder reduzieren soll
reduce-reduce-Konflikt	red a, n red a, m	Parser kann sich mit 1 Symbol Vorgriff nicht entscheiden, zu welchem NTS er reduzieren soll

LALR(1)-Tabellenerzeugung

- Zustände dürfen verschmolzen werden, wenn Kerne bis auf Vorgriffssymbole gleich sind

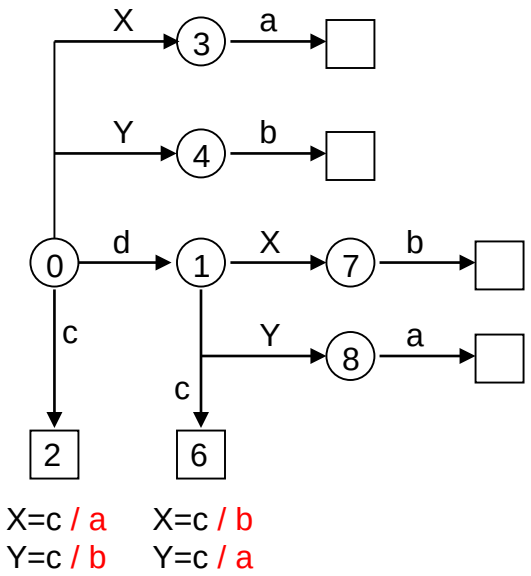
`E = v . / #+` + `E = v . / )+` ⇒ `E = v . / )#+`

- Grammatik ist LALR(1), wenn sich nach dem Verschmelzen kein Parse-Konflikt ergibt (kann höchstens ein reduce-reduce-Konflikt sein; warum?)

Beispiel: LR(1)-Grammatik, die nicht LALR(1) ist

Grammatik

S' = S #
S = d X b
S = d Y a
S = X a
S = Y b
X = c
Y = c



0	S' = . S #		shift	d	1
	S = . d X b	/ #	shift	c	2
	S = . d Y a	/ #	shift	X	3
	S = . X a	/ #	shift	Y	4
	S = . Y b	/ #	shift	S	5
	X = . c	/ a			
	Y = . c	/ b			
1	S = d . X b	/ #	shift	c	6
	S = d . Y a	/ #	shift	X	7
	X = . c	/ b	shift	Y	8
	Y = . c	/ a			
2	X = c .	/ a	red	a	(X = c)
	Y = c .	/ b	red	b	(Y = c)
...	...				
6	X = c .	/ b	red	b	(X = c)
	Y = c .	/ a	red	a	(Y = c)
...	...				

Verschmelzen von 2 und 6 würde zu folgendem Zustand führen

2	X = c .	/ ab	red	a,b	(X = c)
	Y = c .	/ ab	red	a,b	(Y = c)

reduce-reduce-Konflikt!

SLR(1)-Tabellen (Simple LR(1))

- SLR(1) ist weniger mächtig als LALR(1)
+ SLR(1)-Tabellen sind einfacher zu erzeugen als LALR(1)-Tabellen

SLR(1)-Items

Shift-Items	$X = \alpha \,.\, \beta$	⟵	es werden keine Nachfolger gespeichert
Reduce-Items	$X = \alpha \,.\, / \gamma$	⟵	γ = Follow(*X*), d.h. alle Nachfolger von *X* in jedem beliebigen Kontext

Beispiel: LALR(1)-Grammatik, die nicht SLR(1) ist

Grammatik

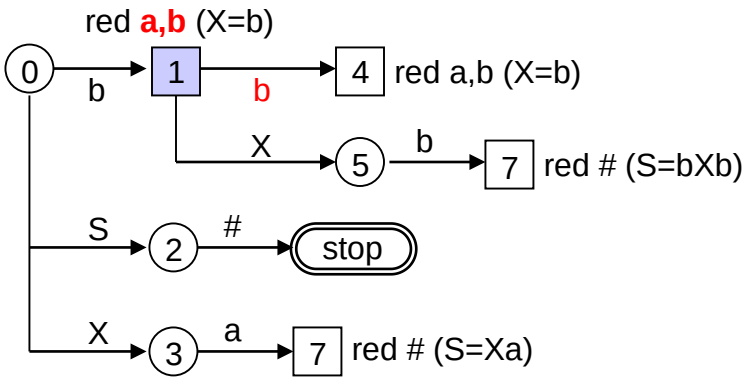
S' = S #
S = b X b
S = X a
X = b

0	S' = . S #	shift	b	1
	S = . b X b	shift	S	2
	S = . X a	shift	X	3
	X = . b			
1	S = b . X b	red	a,b	(X = b)
	X = b .	shift	b	4
	X = . b	shift	X	5

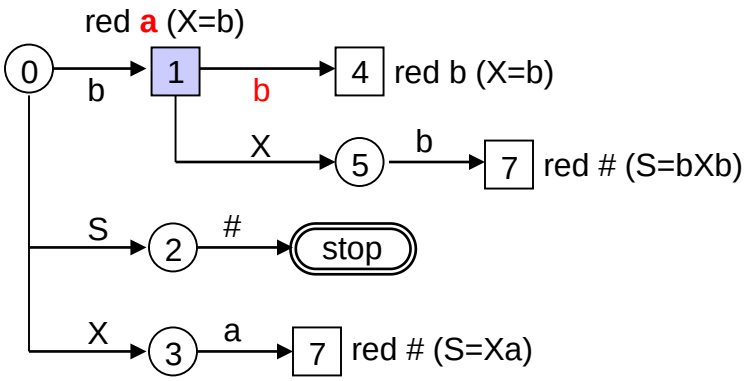
shift-reduce-Konflikt, der in LALR(1)-Tabellen nicht aufgetreten wäre

Follow(S) = {#}
Follow(X) = {a, b}

SLR(1)



LALR(1)



8. Bottomup-Syntaxanalyse

Größenabschätzung

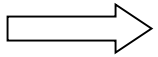
Annahme (z.B. C#)

80	Terminalsymbole		
200	Nonterminalsymbole		
2500	Zustände	⇒	280 x 2.500 = 700.000 Tabellen-Elemente
4	Byte pro Element	⇒	700.000 x 4 = 2.800.000 Byte = 2,8 MByte

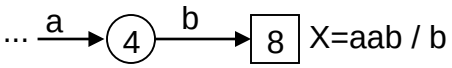
Tabellengröße kann um ca. 90% verkleinert werden

Zusammenfassen von shift und reduce

	a	b	#
...	...	...	...
4	-	s8	-
...	...	...	...
8	-	r4	-
...	...	...	-



	a	b	#
...	...	...	...
4	-	sr4	-
...	...	...	...



- Wenn ein *shift* in einen Zustand *s* führt, in dem nur *reduce*-Aktionen mit der gleichen Produktion vorkommen, kann dieses *shift* durch *shift-reduce* ersetzt werden.
- Zustand *s* kann dann entfallen

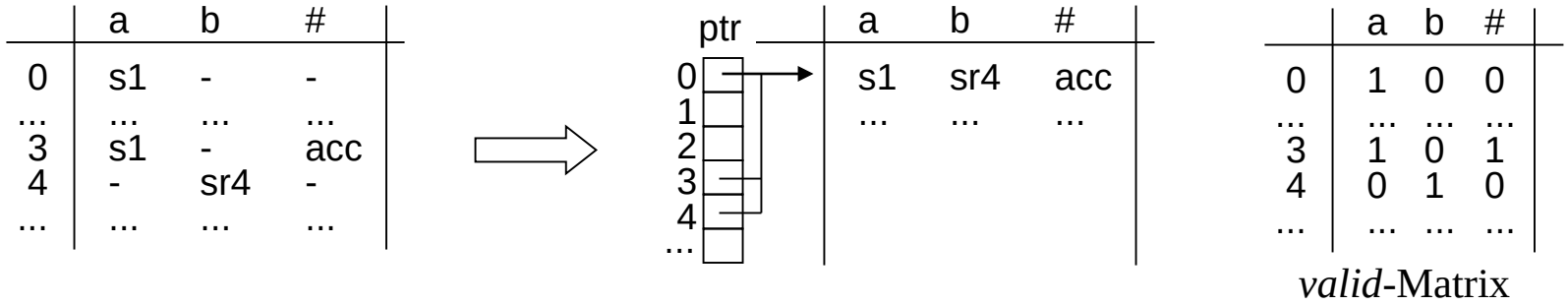
Änderung im Parser

```
case shiftred:  // shiftred n
    sym = next();
    do {
        for (int i = 0; i < length[n]-1; i++) pop();
        a = action[top()][leftSide[n]];
        op = a / 256; n = a % 256;
    } while (op == shiftred);
    state = n; break; // op == shift
```

reduce muss auch angepasst werden

```
case reduce:  // reduce n
    for (int i = 0; i < length[n]; i++) pop();
    a = action[top()][leftSide[n]];
    op = a / 256; n = a % 256;
    while (op == shiftred) {
        for (int i = 0; i < length[n]-1; i++) pop();
        a = action[top()][leftSide[n]];
        op = a / 256; n = a % 256;
    }
    state = n; break; // op == shift
```

Zeilenverschmelzung



valid-Matrix

- Zeilen, deren Aktionen beim Verschmelzen nicht in Konflikt stehen, können verschmolzen werden.
- Benötigt Zeigerarray, um indirekt auf die richtige Zeile zuzugreifen.
- Benötigt Bitmatrix, die sagt, welche Aktionen in den jeweiligen Zuständen gültig sind.

Änderung im Parser

```
BitSet[] valid;
short[] ptr;
...
a = action[ptr[state]][sym];
op = a / 256; n = a % 256;
if (!valid[state].get(sym)) op = error;
switch (op) {
    ...
```

- T- und NT-Aktionen sollten unabhängig voneinander verschmolzen werden.
- Gleiche Technik kann auch für Spaltenverschmelzung angewendet werden.
- Weitere Kompression möglich, wird aber immer teurer.

Beispiel

Originaltabelle (6 Spalten x 12 Zeilen x 2 Bytes = **144** Bytes)

	a	b	#	S	X	Y
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	s6
3	s1	-	acc	-	s7	-
4	-	s8	-	-	-	-
5	r5	s9	r5	-	-	-
6	r1	-	r1	-	-	-
7	-	s5	-	-	-	s10
8	-	r4	-	-	-	-
9	s11	-	-	-	-	-
10	r2	-	r2	-	-	-
11	r6	-	r6	-	-	-

Zusammenfassen von shift und reduce (6 Spalten x 8 Zeilen x 2 Bytes = **96** Bytes)

	a	b	#	S	X	Y
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Beispiel (Fortsetzung)

Nach Zusammenfassen von shift und reduce (**96** Bytes)

	a	b	#	S	X	Y
0	s1	-	-	s3	s2	-
1	s4	r3	-	-	-	-
2	-	s5	-	-	-	sr1
3	s1	-	acc	-	s7	-
4	-	sr4	-	-	-	-
5	r5	s9	r5	-	-	-
7	-	s5	-	-	-	sr2
9	sr6	-	-	-	-	-

Zeilenverschmelzung (3 Spalten * 6 Zeilen * 2 Bytes
+ 2 * 8 Zeilen * 2 Bytes + 8 * 1 Byte = 36 + 32 + 8 = **76** Bytes)

T-Aktionen

	a	b	#
0,3,4	s1	sr4	acc
1	s4	r3	-
2,7,9	sr6	s5	-
5	r5	s9	r5

NT-Aktionen

	S	X	Y
0,2	s3	s2	sr1
3,7	-	s7	sr2

valid

	a	b	#
0	1	0	0
1	1	1	0
2	0	1	0
3	1	0	1
4	0	1	0
5	1	1	1
7	0	1	0
9	1	0	0

8. Bottomup-Syntaxanalyse

Semantische Aktionen

Sind nur am Ende von Produktionen möglich (d.h. beim Reduzieren)

Würde man semantische Aktionen inmitten von Produktionen einfügen

```
X = a (. ... .) b.
```

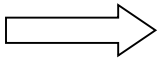
müssten folgendermaßen implementiert werden:

```
X = a Y b.
Y = (. ... .).
```
← leere Produktion, bei deren Reduktion die sem. Aktion ausgeführt wird

Problem: das kann die LALR(1)-Eigenschaft zerstören

```
X = a b c.
X = a (. ... .) b d.
```

KFG wäre LALR(1)



```
X = a b c.
X = a Y b d.
Y = (. ... .).
```

Tabellenerzeugung

i	X = a . b c / #	shift	b	i+1
	X = a . Y b d / #	red	b	(Y=)
	Y = . / b	shift	Y	i+2

shift-reduce-Konflikt

Grund: Parser kann nicht mehr beide Produktionen parallel verfolgen, sondern muss sich entscheiden, in welcher Produktion er ist (d.h. ob er die sem. Aktion ausführen soll oder nicht).

Attribute

- Jedes Symbol hat *genau ein* Ausgangsattribut (u.U. ein Objekt mit mehreren Feldern)
- Nach Erkennen eines Symbols wird sein Attribut auf einen *Attributkeller* gelegt

hinterlassen ihre Attribute am Attributkeller

$X_{\uparrow x}$ = $A_{\uparrow a}\ B_{\uparrow b}\ C_{\uparrow c}$ (. c = pop(); b = pop(); a = pop();
x = ... f(a, b, c) ...;
push(x); .)

Konkretes Beispiel (Auswertung arithmetischer Ausdrücke)

sem. Aktionen werden durchnummeriert

$Expr_{\uparrow x}$ = $Term_{\uparrow x}$.	~~(. push(pop()); .)~~	
$Expr_{\uparrow x}$ = $Expr_{\uparrow x}$ "+" $Term_{\uparrow y}$	(. push(pop() + pop()); .). // x = x + y;	← 1
$Term_{\uparrow x}$ = $Factor_{\uparrow x}$.		
$Term_{\uparrow x}$ = $Term_{\uparrow x}$ "*" $Factor_{\uparrow y}$	(. push(pop() * pop()); .). // x = x * y;	← 2
$Factor_{\uparrow x}$ = const	(. push(t.numVal); .).	← 3
$Factor_{\uparrow x}$ = "(" $Expr_{\uparrow x}$ ")".		

Änderungen im Parser

Produktionen

```
Expr    = Term.
Expr    = Expr "+" Term    (. 1 .).
Term    = Factor.
Term    = Term "*" Factor  (. 2 .).
Factor  = const            (. 3 .).
Factor  = "(" Expr ")".
```

Produktionentabelle

	leftSide	length	sem
0	Expr	1	0
1	Expr	3	1
2	Term	1	0
3	Term	3	2
4	Factor	1	3
5	Factor	3	0

Semantikauswerter

```
void semAction (int n) {
   switch (n) {
      case 1: push(pop() + pop()); break;
      case 2: push(pop() * pop()); break;
      case 3: push(t.numVal); break;
   }
}
```

Variablen, die in mehreren semantischen Aktionen vorkommen, müssen global sein (Probleme bei rekursiven NTS).

Parser

```
...
switch(op) {
   ...
   case reduce:  // red n
      if (sem[n] != 0) semAction(sem[n]);
      for (int i = 0; i < length[n]; i++) pop();
      ...
}
```

Eingangsattribute

Im Prinzip implementierbar als semantische Aktionen mit Attributzuweisungen:

X = a $Y_{\downarrow v}$ b.
$Y_{\downarrow w}$ = ...

kann dargestellt werden als

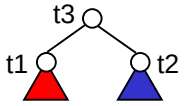
X = a (. w = v; .) Y b.
Y = ...

kann aber wieder die LALR(1)-Eigenschaft zerstören

Daher

- In der Praxis bauen LALR(1)-Parser einfach einen Syntaxbaum auf
  (kann mit Ausgangsattributen und sem. Aktionen am Regelende implementiert werden)

$A_{\uparrow t3} = B_{\uparrow t1} C_{\uparrow t2}$ (. t2 = pop(); t1 = pop();
t3 = new Tree(NodeKind.A, t1, t2);
push(t3); .)



- Semantikauswertung findet dann am Syntaxbaum statt

8. Bottomup-Syntaxanalyse

Idee

Situation beim Auftreten eines Fehlers

$s_0 \ldots s_n \;.\; t_0 \ldots t_m \#$

Stack Input

t_0 ist in s_n nicht erlaubt

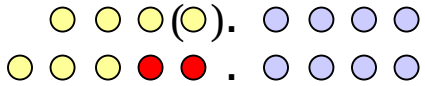
Ziel: Stack und Input so synchronisieren, dass gilt:

$s_0 \ldots s_i \;.\; t_j \ldots t_m \#$

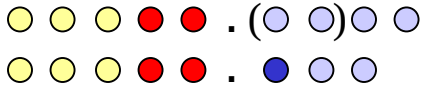
und t_j in s_i akzeptiert wird

Vorgehensweise

- Zustände ein- und/oder auskellern



- Token einfügen und/oder löschen



Algorithmus

Röhrich J.: *Methods for the Automatic Construction of Error-correcting Parsers*. Acta Informatica 13, 115-139, 1980

1. Suche Fluchtweg

- Ersetze t_0 .. t_m durch eine virtuelle (fiktive) Eingabe v_0 .. v_k, die den Parser vom Fehlerzustand s_n möglichst schnell in den Endzustand steuert.
- Sammle bei der simulierten Analyse von v_0 .. v_k alle Token, die in durchlaufenen Zuständen gültig sind $\Rightarrow$ *Anker*

$s_0 \ldots s_n \; . \; \sout{t_0 \ldots t_m \; \#}$

$v_0 \ldots v_k \; \#$

2. Lösche fehlerhafte Token

- Überlies Token aus der Eingabe t_0 .. t_m, bis ein Token t_j auftritt, das ein Anker ist.

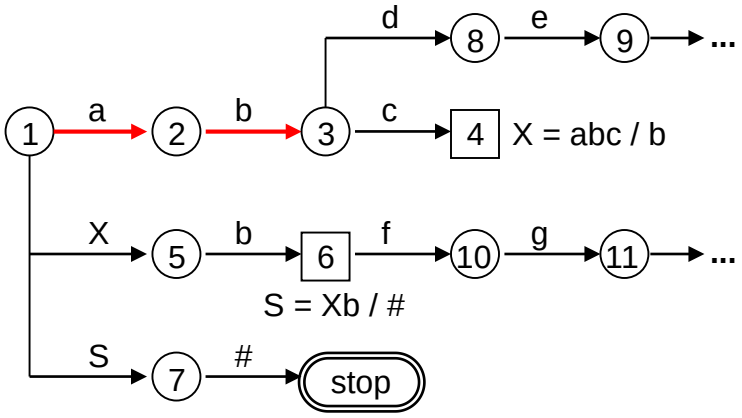
$s_0 \ldots s_n \; . \; \sout{t_0 \ldots t_{j-1}} \; t_j \ldots t_m \; \#$

3. Füge fehlende Token ein

- Steuere die Analyse von s_n weg mit v_0 .. v_k, bis ein Zustand s_i erreicht wird, in dem t_j gültig ist.
- Füge alle "gelesenen" virtuellen Token von v_0 .. v_k in den Quelltext vor t_j ein.

$s_0 \ldots s_i \; . \; t_j \ldots t_m \; \#$

Beispiel



Eingabe: a b b #

Suche Fluchtweg

fiktive Eingabe: c b #

Anker:

3	c, d
4	b
5	b
6	f, #
7	#

Füge fehlende Token ein

c einfügen, um nach 4 zu kommen.

In 4 geht es mit *b* wieder weiter.

Lösche fehlerhafte Token

Es wird nichts überlesen, weil *b* bereits ein Anker ist

Korrigierte Eingabe

a b **c** b #

Fehlermeldungen

Geben an, was eingefügt oder gelöscht wurde

- Wenn Token *a*, *b*, *c* aus der restlichen Eingabe gelöscht wurden

  line ... col ... : "a b c" deleted

- Wenn Token *x*, *y* vor der restlichen Eingabe eingefügt wurden

  line ... col ... : "x y" inserted

- Wenn Token *a*, *b*, *c* aus der restlichen Eingabe gelöscht und *x*, *y* eingefügt wurden

  line ... col ... : "a b c" replaced by "x y"

Nochmals Beispiel (mit Simulation des Parsers)

Grammatik

0	S' = S #
1	S = X Y
2	S = S X Y
3	X = a
4	X = a a b
5	Y = b
6	Y = b b a

	a	b	#	S	X	Y	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	#
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	#
6	r1	-	r1	-	-	-	#
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	#
11	r6	-	r6	-	-	-	#

Fehlerhafte Eingabe

a a a b #

Jeder Zustand hat ein "Wegweisersymbol", das den Fluchtweg weist. Wie man dazu kommt, siehe später.

Beginn der Analyse

Stack	*Input*	*Aktion*
0	a a a b #	s1
0 1	a a b #	s4
0 1 4	a b #	-- error!

Fluchtweg suchen und Anker sammeln

Stack	*guide*	*Aktion*	*Anker*
0 1 4	b	s8	b
0 1 4 8	b	r4, s2	b
0 2	b	s5	b
0 2 5	#	r5, s6	a, b, #
0 2 6	#	r1, s3	a, #
0 3	#	acc	a, #

Anker = {*a*, *b*, #}

Beispiel (Fortsetzung)

Grammatik

0	S' = S #
1	S = X Y
2	S = S X Y
3	X = a
4	X = a a b
5	Y = b
6	Y = b b a

	a	b	#	S	X	Y	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	**#**
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	**#**
6	r1	-	r1	-	-	-	**#**
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	**#**
11	r6	-	r6	-	-	-	**#**

restliche Eingabe

a b #

Anker

{a, b, #}

Input überlesen

Man braucht nichts zu überlesen, weil nächstes Token *a* bereits in der Ankermenge {*a*, *b*, #} ist

Fehlende Token einfügen

Stack	*guide*	*Aktion*	*eingefügt*
0 1 4	b	s8	b
0 1 4 8	b	r4, s2	
0 2	b	s5	b
0 2 5			

← Nur *shift* mit TS führt zu eingefügten Token, nicht *reduce*

Hier kann mit *a* fortgesetzt werden

Beispiel (Fortsetzung)

Grammatik

0	S' = S #
1	S = X Y
2	S = S X Y
3	X = a
4	X = a a b
5	Y = b
6	Y = b b a

	a	b	#	S	X	Y	guide
0	s1	-	-	s3	s2	-	**a**
1	s4	r3	-	-	-	-	**b**
2	-	s5	-	-	-	s6	**b**
3	s1	-	acc	-	s7	-	#
4	-	s8	-	-	-	-	**b**
5	r5	s9	r5	-	-	-	#
6	r1	-	r1	-	-	-	#
7	-	s5	-	-	-	s10	**b**
8	-	r4	-	-	-	-	**b**
9	s11	-	-	-	-	-	**a**
10	r2	-	r2	-	-	-	#
11	r6	-	r6	-	-	-	#

restliche Eingabe

a b #

Fortsetzen der Analyse

Stack	*Input*	*Aktion*
0 2 5	a b #	r5, s6
0 2 6	a b #	r1, s3
0 3	a b #	s1
0 3 1	b #	r3, s7
0 3 7	b #	s5
0 3 7 5	#	r5, s10
0 3 7 10	#	r2, s3
0 3	#	acc

Korrigierte Eingabe

a a b b a b #

Fehlermeldung

line ... col ...: "b b" inserted

Wie findet man die Wegweiser?

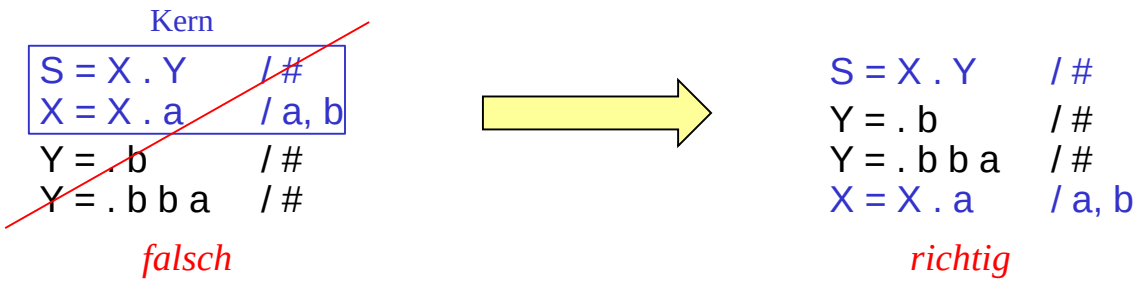
1. Produktionen so ordnen, dass die erste Produktion jedes NTS jene ist, die am schnellsten terminiert (=> Fluchtgrammatik)

```
S = X Y.
S = S X Y.
X = a.
X = X a.
Y = b.
Y = b b a.
```

rot: Produktionen der Fluchtgrammatik

rekursive Produktionen vermeiden
linksrekursive Produktionen gehören nie zur Fluchtgrammatik

2. LALR(1)-Tabellenerzeugung, aber beim Expandieren des Kerns neue Items sofort hinter Item einfügen, das die Expansion verursacht hat (Reihenfolge der Items ist wichtig!).



3. Erste erzeugte T-Aktion gibt in jedem Zustand den Wegweiser an

```
shift b ...
shift a ...
shift Y ...
```

Wegweiser in diesem Zustand ist b

Beispiel für Wegweiserberechnung

0	S' = S #
1	S = X Y
2	X = a
3	X = X a
4	Y = a
5	Y = b

Grammatik mit richtig geordneten Produktionen

						Wegweiser
0	S' = . S #		shift	a	1	a
	S = . X Y	/ #	shift	S	2	
	X = . a	/ a, b	shift	X	3	
	X = . X a	/ a, b				
1	X = a .	/ a, b	red	a, b	X = a	a
2	S' = S . #		acc	#		#
3	S = X . Y	/ #	shift	a	4	a
	Y = . a	/ #	shift	b	5	
	Y = . b	/ #	shift	Y	6	
	X = X . a	/ a, b				
4	Y = a .	/ #	red	#	Y = a	#
	X = X a .	/ a, b	red	a, b	X = X a	
5	Y = b .	/ #	red	#	Y = b	#
6	S = X Y .	/ #	red	#	S = X Y	#

Wenn in Zustand 3 die Items nicht so wie hier geordnet worden wären,
wären die Items in Zustand 4 in anderer Reihenfolge
und es hätte sich in Zustand 4 ein anderer Wegweiser ergeben, der nicht entlang des Fluchtwegs führt.

Beurteilung dieser Fehlerbehandlungstechnik

- Behindert fehlerfreie Analyse nicht
- Terminiert immer (# ist immer ein Anker)
- Meldet die Fehler nicht nur, sondern "korrigiert" sie auch.
  Die Korrektur ist allerdings nicht immer die, die man will.
- Ermöglicht gute Fehlermeldungen