Konstruktion von LR(1)-Automaten

Anfangszustand, Übergänge, und Zustände konstruieren bis keine neuen mehr entstehen.

Dabei auf jeden Zustand *Hüllenbildung* anwenden:
Wenn [A ::= u . B v R] im Zustand q ist, dann ist auch für jede Produktion B ::= w
[B ::= . w Anf (v R)] im Zustand q.

Anfangszustand:
Hülle von [S ::= . u {#}]
S ::= u ist eindeutige Startproduktion,
 # ist künstliches Schlußzeichen (eof)

Übergänge in Nachfolgezustände mit Symbolen x , die in Situationen auf die Analyseposition folgen, x Terminal oder Nichtterminal:

... A ::= u . x v R1 ... B ::= w . x z R2 ...	—x→	A ::= u x . v R1 B ::= w x . z R2 *Hülle bilden*

Konflikte: Automat nicht determ., Grammatik nicht LR(1):

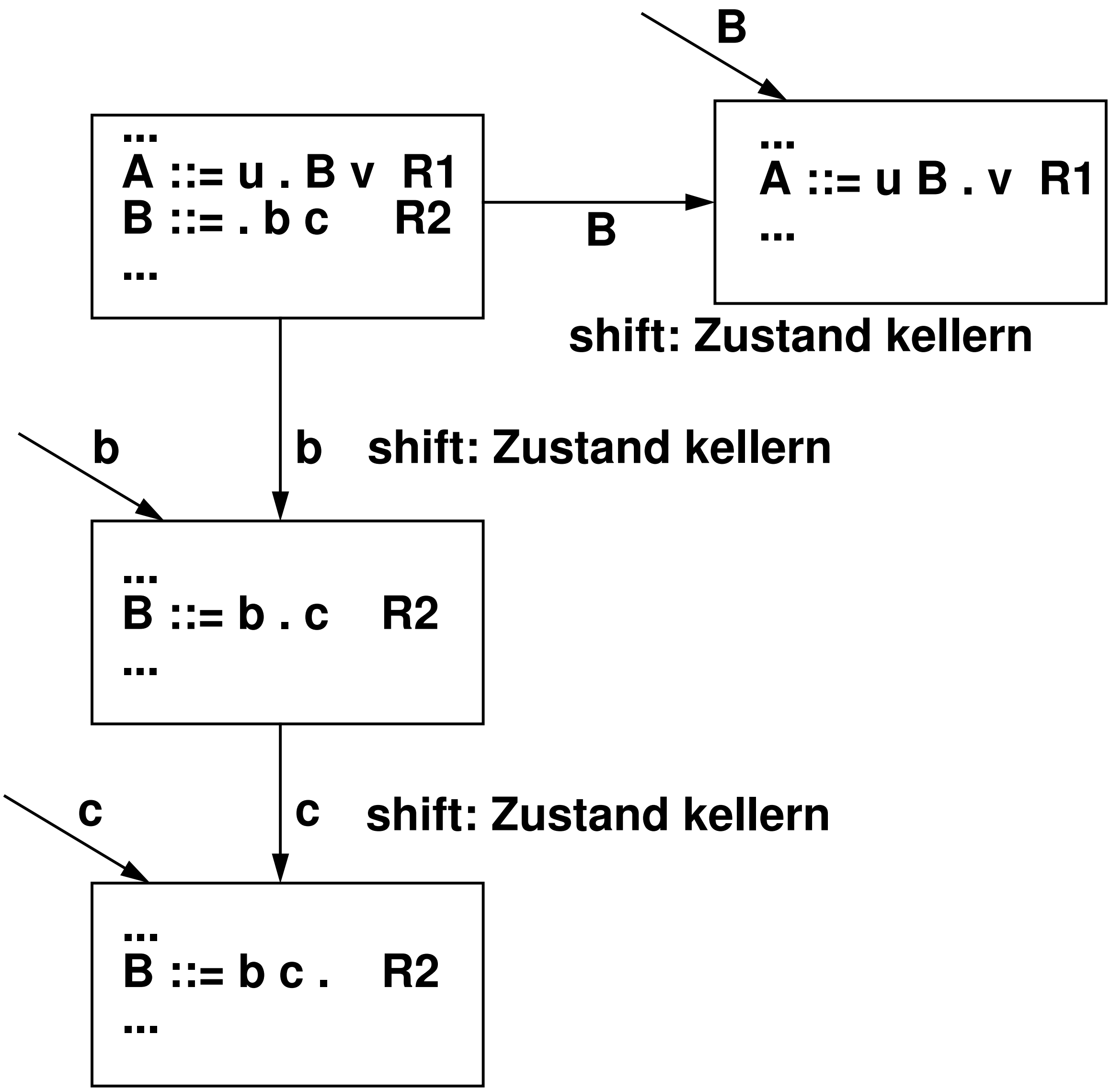
reduce / reduce

... A ::= u . R1 B ::= v . R2 ...	R1, R2 nicht disjunkt

shift / reduce

... A ::= u .t v R1 B ::= w . R2 ...	t $\in$ R2

Operationen des LR(1)-Automaten schematisch



reduziere B ::= b c , falls Eingabe in R2
entkellere 2 Zustände (Länge rechte Seite)
Übergang mit B (linke Seite)

Fehler: kein Übergang möglich

Halt: reduziere Startproduktion bei #

Vereinfachte LR-Klassen

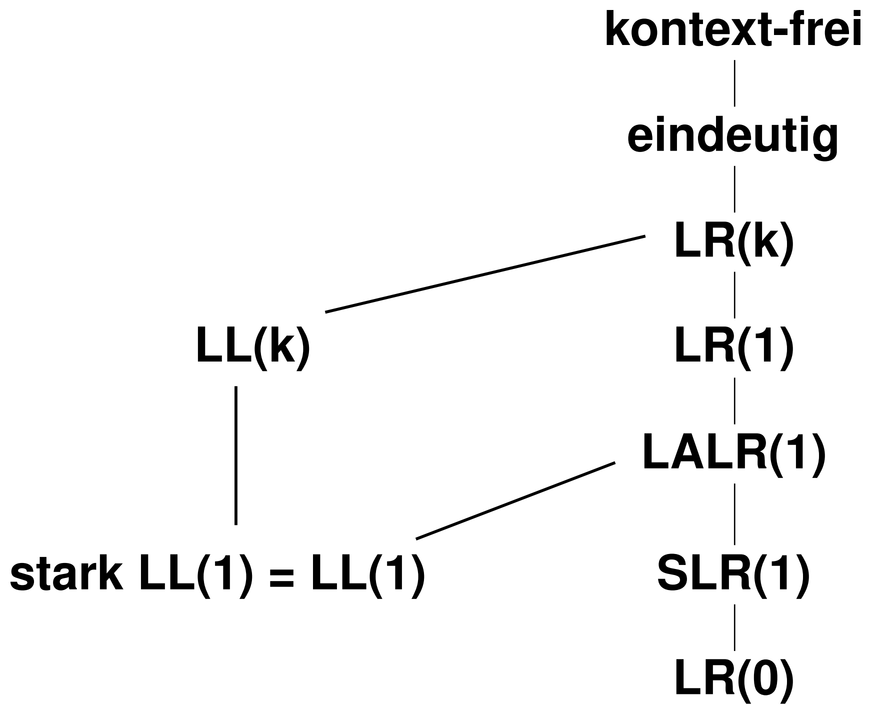
LR(1): zuviele Zustände für praktischen Einsatz
Ursache: Rechtskontexte differenzieren stark
Abhilfe: Rechtskontexte vereinfachen, weniger Zustände, Grammatik weniger mächtig

LR(0): Alle Situationen **ohne** Rechtskontext
Konsequenz: Reduktionssituationen einzeln im Zustand

SLR(1): **LR(0)-Zustände**; bei Reduktionssituationen vergröberten Rechtskontext zur Entscheidung:
[A ::= u . Folge (A)]

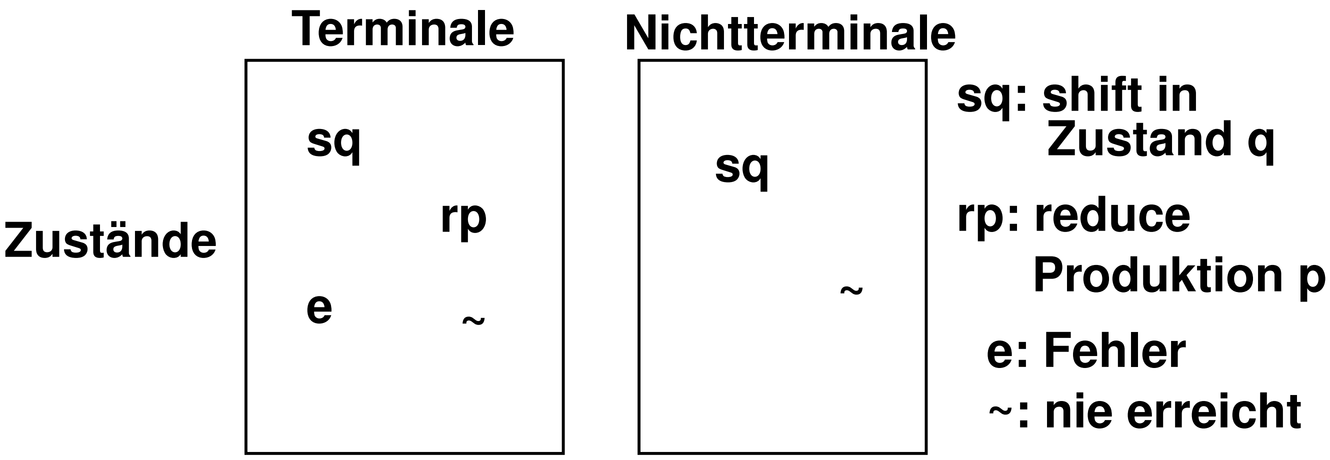
LALR(1): LR(1)-Zustände zusammenlegen, wenn sich ihre Situationen nur im Rechtskontext unterscheiden; liefert **LR(0)-Zustände** mit "exaktem" Rechtskontext; **wichtigste Parser-Generatoren**

Grammatik-Hierarchie:



Implementierung von LR-Automaten

Tabellen-gesteuert:
(direkt programmmiert möglich, aber zu groß)



Tabellen komprimieren:

- **verträgliche Zeilen / Spalten zusammenlegen**
  Graphfärbung
- **separate Fehlermatrix für Terminaltabelle (Bit-Matrix)**
  erhöht die Zahl verträglicher Zeilen / Spalten
- **kleinsten Wert von Zeilen / Spalten subtrahieren**
  erhöht die Zahl verträglicher Zeilen / Spalten
  verkleinert Wertebereich der Einträge
- **LR(0)-Reduktionszustände streichen, neue Operation im Vorgängerzustand: shift - reduce**
  eliminiert ca. 30% in der Praxis

Insgesamt Tabellenreduktion auf ca. 10-20% möglich.

Behandlung syntaktischer Fehler

Allgemeine Kriterien:

- **Fehler so früh wie möglich erkennen**
- **Symptom quellbezogen melden**
- **Parsing möglichst kurz nach Fehlerstelle fortsetzen**
- **Folgefehler vermeiden**
- **strukturell korrekter Baum trotz Fehler**
- **kein Zurücksetzen, Zurücknehmen von Aktionen**
- **möglichst kein zusätzlicher Aufwand für korrekte Programme**

LL- und LR-Verfahren erkennen Fehler frühest möglich.

Begriffe: **Fehlerstelle t, korrekter Präfix w:**

Eingabe	$w\ t\ x \in T^*$	
korrekter Präfix	w	d.h. es gibt ein $u \in T^*$ mit $w\,u \in L(G)$
	$w\ t$	ist kein korrekter Päfix

Parser akzeptiert t nicht.

Aufsetzpunkt d für die Fortsetzung des Parsing:

$$w\ t\ x \;=\; w\ y\ d\ z$$

$$\downarrow \quad \text{ersetzen (konzeptionell)}$$

$w\ v\ d\ z$ mit $w\ v\ d$ ist korrekter Präfix

Methode: simulierte Fortsetzung

anwendbar bei LL- und LR-Verfahren (mit kopierbarem Keller)

Idee: Aus Fehlerstelle und Keller Menge von Symbolen als mögliche Aufsetzpunkte bestimmen.

Verfahren:

1. **bei Fehler Keller speichern**
   Fehlersymbol überlesen
2. **Menge D möglicher Aufsetzpunkte bestimmen**
   Automat zuende laufen lassen (Modifikation siehe unten),
   Symbole nicht aus der Eingabe lesen, sondern
   in D einfügen
3. **einen Aufsetzpunkt finden**
   Eingabe überlesen bis Symbol in D
4. **Aufsetzpunkt erreichen**
   gespeicherter Keller wird wieder Keller des Automaten,
   Übergänge bis der Aufsetzpunkt akzeptabel ist,
   dabei Symbole einfügen statt aus Eingabe lesen.
   Damit ist ein korrekter Präfix konstruiert.
5. **Automat normal fortsetzen**

Automatenkonstruktion für Schritt 2 erweitern:

Automat muß Keller leeren und terminieren,
dazu in jedem Zustand ein zulässiges Symbol für
Übergang auswählen
(automatisch generierbar)

Parser-Generatoren

PGS	Univ. Karlsruhe; in Eli	**LALR(1), Tabellen-gest.**
Cola	Univ. Paderborn; in Eli	**LALR(1), Tab. o. dir. progr.**
Lalr	Univ. / GMD Karlsruhe	**LALR(1), Tabellen-gest.**
Yacc	Unix-Werkzeug	**LALR(1), Tabellen-gest.**
Bison	Gnu	**LALR(1), Tabellen-gest.**
Llgen	Amsterdam Comp. Kit	**LL(1), recursive descent**
Deer	Univ. Col. Bouder	**LL(1), recursive descent**

Spezifikationsform:

EBNF: Cola, PGS, Lalr
BNF: Yacc, Bison

Fehlerbehandlung:

simulierte Fortsetzung (automatisch): Cola, PGS, Lalr
Fehlerproduktionen (manuell): Yacc, Bison

Aktionen:

Anweisungen in Implementierungssprache
am Ende von Produktionen: Yacc, Bison
auch innerhalb von Produktionen: Cola, PGS, Lalr

Konfliktlösung:

Einfluß auf Zustände (reduziere bei ...)Cola, PGS, Lalr
Reihenfolge von Produktionen: Yacc, Bison
Präzedenz und Assoziativität: Yacc, Bison

Implementierungssprache:

C Cola, Yacc, Bison
C, Pascal, Modula-2, Ada PGS, Lalr