Kapitel 3

Syntaktische Analyse

Kapitel 3: Syntaktische Analyse

Syntaktische Analyse

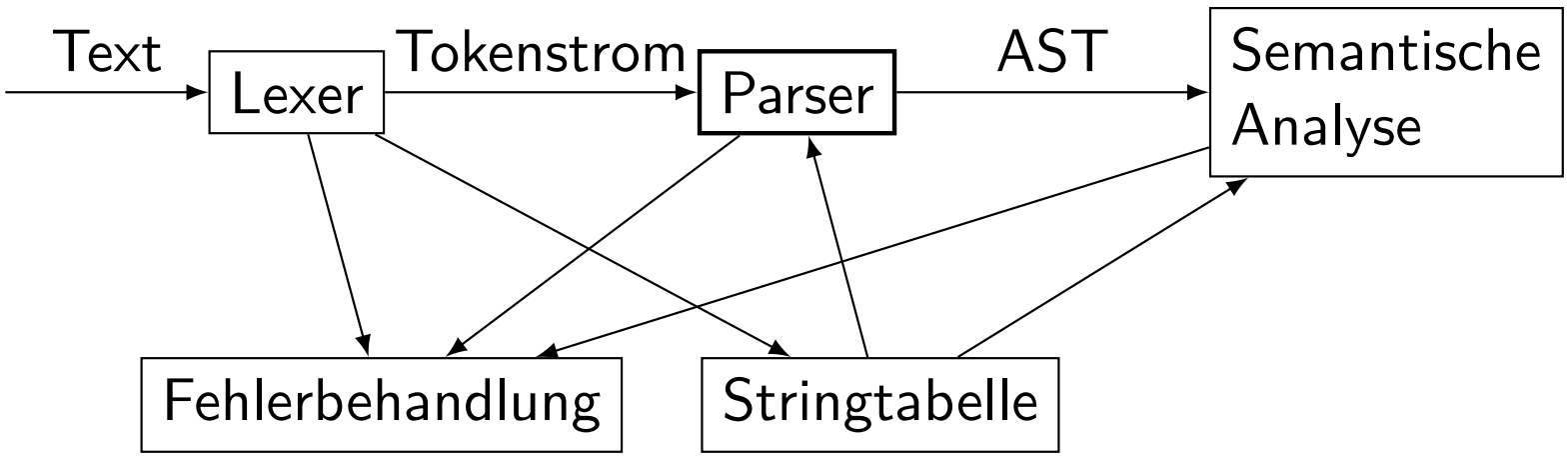
vorgegeben:

- Tokenstrom
- kontextfreie Grammatik (deterministisch?)

Aufgaben

- syntaktische Struktur bestimmen
- syntaktische Fehler melden, korrigieren (?)
- Ausgabe (immer): abstrakte Syntax (Rechts-/Linksableitung), Symbole (Bezeichner, Konstanten, usw.)

Eingliederung in den Übersetzer



ADT Parser

gelieferte Operationen:

- `parse() : AST`

benötigte Operationen:
Lexer/Tokenstrom:

- `next_token() : Token`

Fehlerbehandlung:

- `add_error(nr, pos)`

für Aufbau Strukturbaum:

- `production(nr)`, `symbol(value)`

Aufgabe des Parsers, formal

Gegeben: Grammatik $G = (T, N, P, Z)$ mit
T Alphabet, N Nichtterminale, P Produktionen, Z Zielsymbol

Gesucht: Entscheidung gehört Tokenstrom s zur Sprache $L(G)$,

- wenn ja, Produktionenfolge für Links-/Rechtsableitung
- wenn nein, Fehlerbehandlung zur Korrektur des Tokenstroms

Unterscheide konkrete Syntax G_k und abstrakte Syntax G_a:

Gesucht: Entscheidung, gehört Tokenstrom s zu $L(G_k)$,
wenn ja, Produktionenfolge für Links-/Rechtsableitung für G_a?
Beziehung zwischen G_k und G_a?

Annahmen für das Parsen

Syntax ist kontextfrei

- eigentlich ist sie kontext-sensitiv
- aber kontext-sensitive Grammatiken nicht in linearer Zeit parsbar (Kontextfreiheit ist selbsterfüllende Prophezeiung)
- der über die kontextfreie Grammatik hinausgehende Teil der Syntax heißt im Übersetzerbau statische Semantik

Syntax ist deterministisch kontextfrei

- keine wesentliche Einschränkung, da auch vom menschlichen Leser erwünscht

keine Rückkopplung zur lexikalischen Analyse

- sonst gäbe es mehrere Grundzustände des Lexers, gesteuert vom Parser

keine Rückkopplung semantische Analyse – syntaktische Analyse

- typunabhängige Syntaktische Analyse: Zustände des Parsers unabhängig von der Namens- und Typanalyse

Fragen

- Wie wird Sprache erkannt?
- Wie wird abstrakter Strukturbaum aufgebaut?
- Was geschieht bei Fehlern?

Historie, kf Grammatiken + Verarbeitung

1955	Definition und Klassifikation (Chomsky und Bar Hillel)
1957–1959	Kellerautomaten (Bauer&Samelson, sequentielle Formelübersetzung, 1959)
1961	formaler Zusammenhang kfG-Kellerautomat (Öttinger)
1958–1966	kfGs und BNF setzen sich für die Syntax von Programmiersprachen durch (Algol 58, Algol 60, ...)
1960–1972	Verfahren des rekursiven Abstiegs (Glennie) und dessen theoretische Fundierung als LL-Grammatiken (auch heute noch oft neu erfunden!)
1963–1969	deterministische kfGs: beschränkte Operatorpräzedenz, LR, SLR, LALR, ...
seit 1972	nichts wesentlich Neues außer Optimierung, Fehlerbehandlung

Kapitel 3: Syntaktische Analyse

Grundbegriffe

Kontextfreie Grammatik

Eine Grammatik $G = (T, N, P, Z)$ heißt kontextfrei, wenn für jede Produktion $A \rightarrow \alpha$ gilt: $A \in N$.

Sprache $L(G)$ einer Grammatik G

$L(G) = \{\omega \in T^* \mid Z \Rightarrow_G^* \omega\}$ ist die Menge aller in der Grammatik ableitbaren Wörter.

Linksableitung ($\Rightarrow_L$)

Es wird stets das linkeste Nichtterminal ersetzt.

Rechtsableitung ($\Rightarrow_R$)

Es wird stets das rechteste Nichtterminal ersetzt.

Beispiel

$G = (N, T, P, Z)$ mit

$$\begin{aligned} T &= \{\text{bez}, +, *, (,)\} \\ N &= \{Z, A, T, F\} \\ P &= \{Z \to A, A \to T, T \to F, F \to \text{bez}, \\ &\qquad A \to A + T, T \to T * F, F \to (A)\} \end{aligned}$$

ist eine kontextfreie Grammatik.

Linksableitung für a+(b+c):
A $\Rightarrow$ A+T $\Rightarrow$ T+T $\Rightarrow$ F+T $\Rightarrow$ a+T $\Rightarrow$ a+(A) $\Rightarrow$
a+(A+T) $\Rightarrow$ a+(T+T) $\Rightarrow$ a+(F+T) $\Rightarrow$ a+(b+T) $\Rightarrow$
a+(b+F) $\Rightarrow$ a+(b+c)

Rechtsableitung für a+(b+c):
A $\Rightarrow$ A+T $\Rightarrow$ A+F $\Rightarrow$ A+(A) $\Rightarrow$ A+(A+T) $\Rightarrow$
A+(A+F) $\Rightarrow$ A+(A+c) $\Rightarrow$ A+(T+c) $\Rightarrow$ A+(F+c) $\Rightarrow$
A+(b+c) $\Rightarrow$ T+(b+c) $\Rightarrow$ F+(b+c) $\Rightarrow$ a+(b+c)

Schreibweise der Produktionen

in der Theorie: $A \to x|y|\ldots$, $A \in N$, x, $y \in V^*$, $V = T \cup N$
praktisch: Backus-Naur-Form (BNF)

- Nichtterminale in spitzen Klammern,
- Terminale als Symbole oder wie Nichtterminale
- ::= statt $\to$

<Ausdruck> ::= <Term> | <Ausdruck> + <Term>

Rechnereingabe: Erweiterte Backus-Naur-Form (EBNF)

- wie BNF, aber Bezeichner oft ohne spitze Klammern
- | (oder), . (Abschluß), () (Gruppierung), [] (optional), $*$ (Wiederholung, auch 0-mal), $+$ (Wiederholung, mindestens einmal) als Beschreibungssymbole
- Terminale durch Apostrophs o. ä. ausgezeichnet

Ausdruck ::= Term ('+' Term)$*$.

Fortran-, Cobol-, Java-Beschreibung: Abarten von EBNF

Grammar Engineering (1/3)

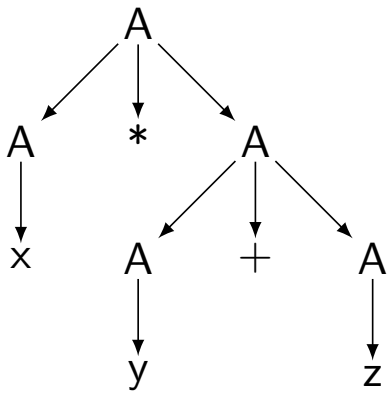
Forderungen:

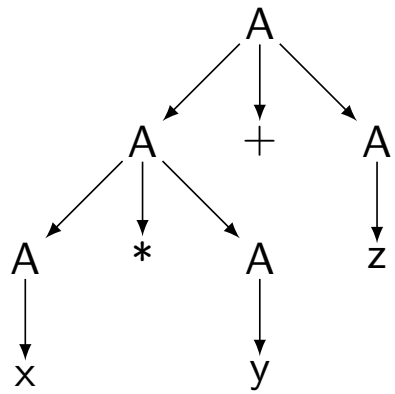
- deterministische Grammatik: zu einer Eingabe existiert höchstens ein Syntaxbaum
- Operatorprioritäten: Grammatik erzeugt Syntaxbaum gemäß Prioritäten

Gegenbeispiel:

$$A \rightarrow A + A \mid A * A \mid \textbf{bez} \mid (A)$$

x*y+z hat 2 Syntaxbäume - sogar einen, der „Punkt vor Strich" ignoriert.



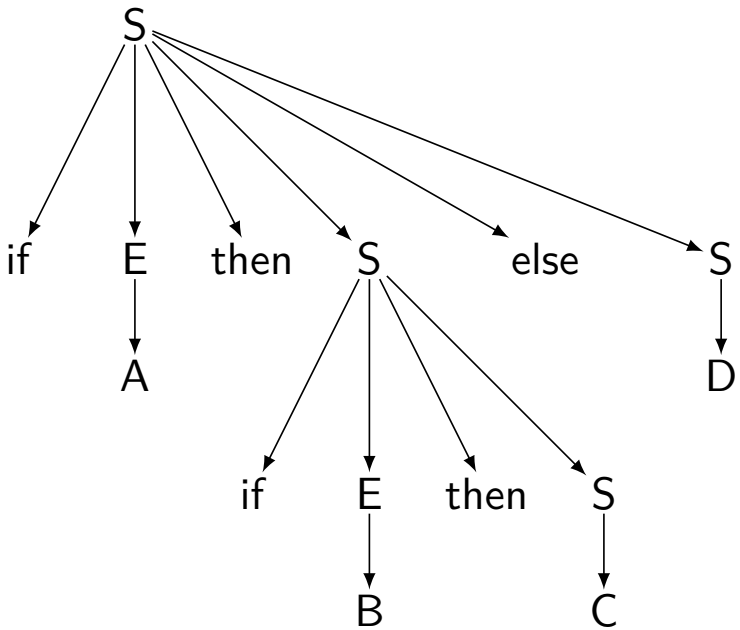


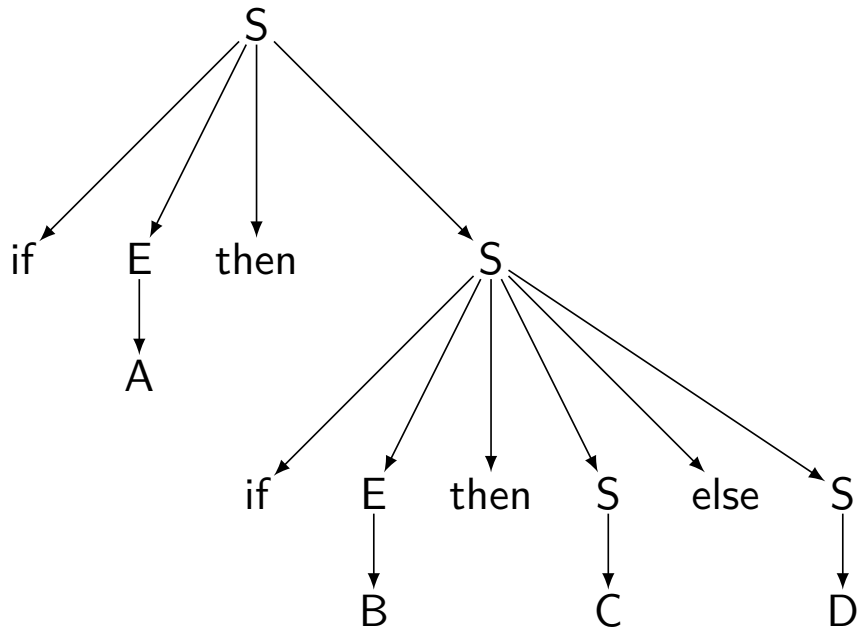
Grammar Engineering (2/3)

Weiteres Gegenbeispiel: „Dangling Else"

$$
\begin{aligned}
S &\rightarrow \textbf{if } E \textbf{ then } S \\
&\mid \textbf{if } E \textbf{ then } S \textbf{ else } S
\end{aligned}
$$

if A then if B then C else D
hat 2 verschiedene Syntaxbäume.

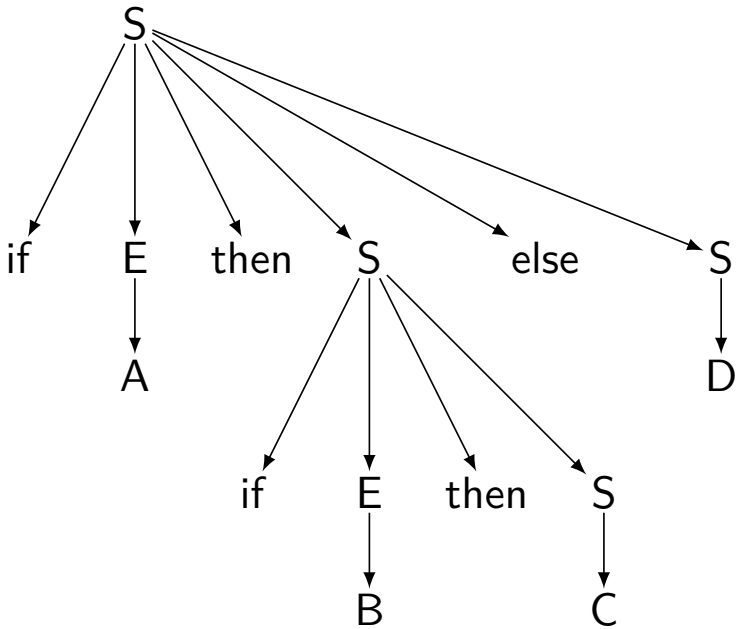


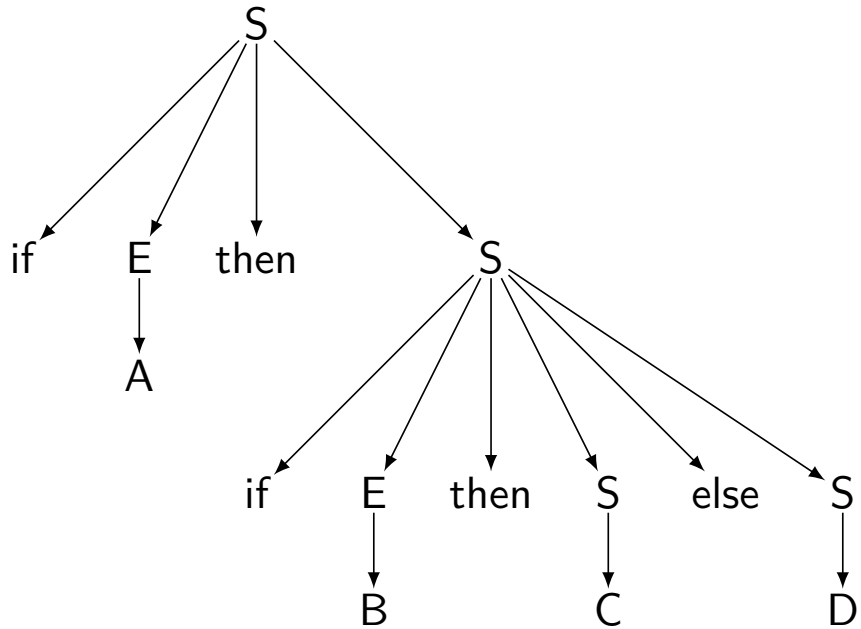


Grammar Engineering (3/3)

if A then if B then C else D

hat 2 verschiedene Syntaxbäume.





In der Praxis gehört ein **else** aber immer zum letzten **if**.

- Parsergeneratoren erkennen Mehrdeutigkeiten
- aber in Grammatiken mit hunderten Produktionen sind Mehrdeutigkeiten schwer zu beheben

Faustregeln

- ein Nichtterminal pro Prioritätsebene
- nicht zweimal dasselbe Nichtterminal auf der rechten Seite
- Links- oder Rechtsassoziaitivität von Operatoren wird durch links- bzw. rechtsrekursive Regeln ausgedrückt

Beispielgrammatik

Ausdrücke:

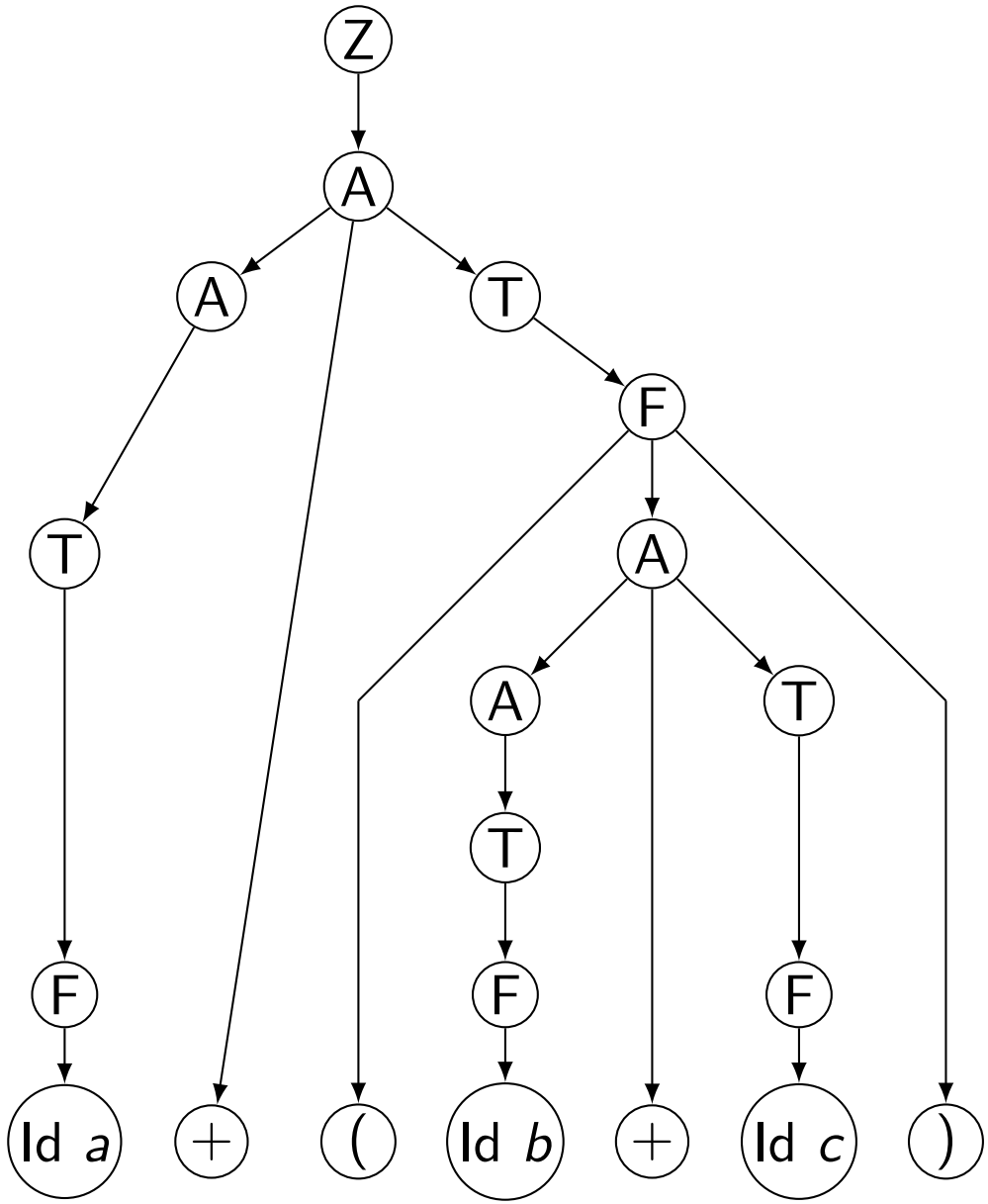
(0) $Z \rightarrow A$
(1) $A \rightarrow T$
(2) $T \rightarrow F$
(3) $F \rightarrow \text{bez}$
(4) $A \rightarrow A + T$
(5) $T \rightarrow T * F$
(6) $F \rightarrow (A)$

EBNF:

(0) Z ::= A.
(1) A ::= T ('+' T)*.
(2) T ::= F ('*' F)*.
(3) F ::= **bez** | '(' A ')'.

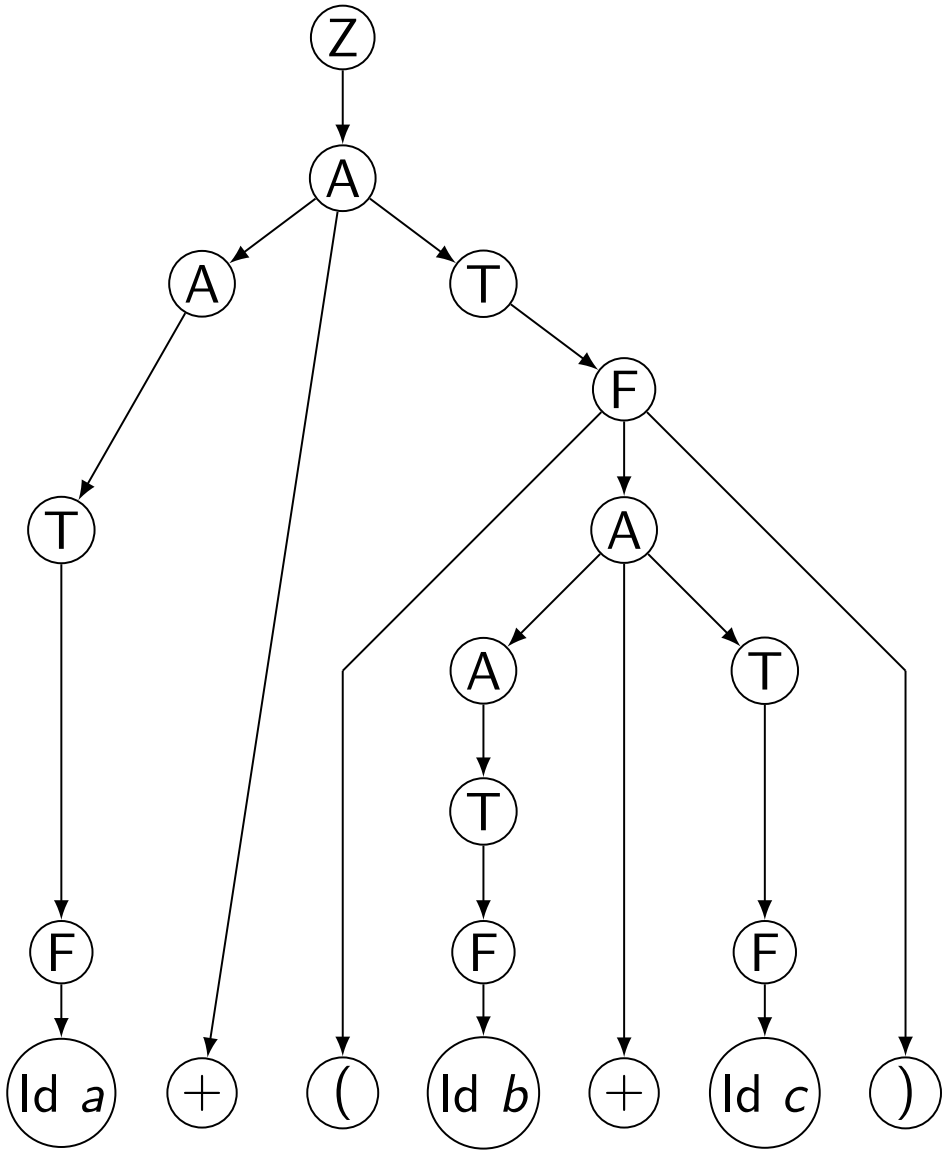
Ausdruck $a + (b + c)$

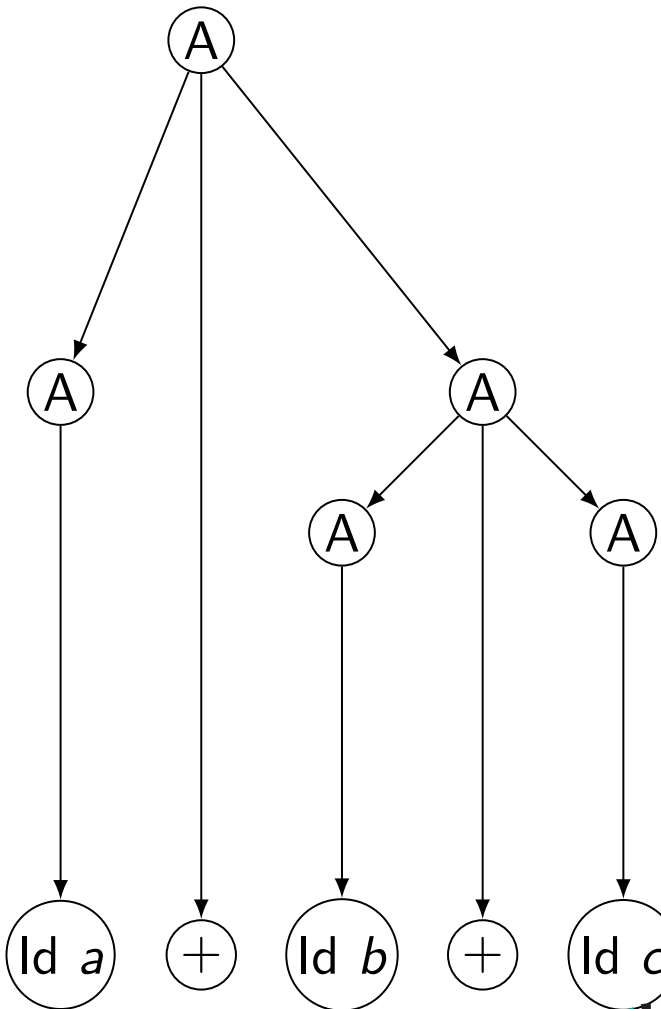
Konkreter Strukturbaum:



Konkrete und abstrakte Syntax

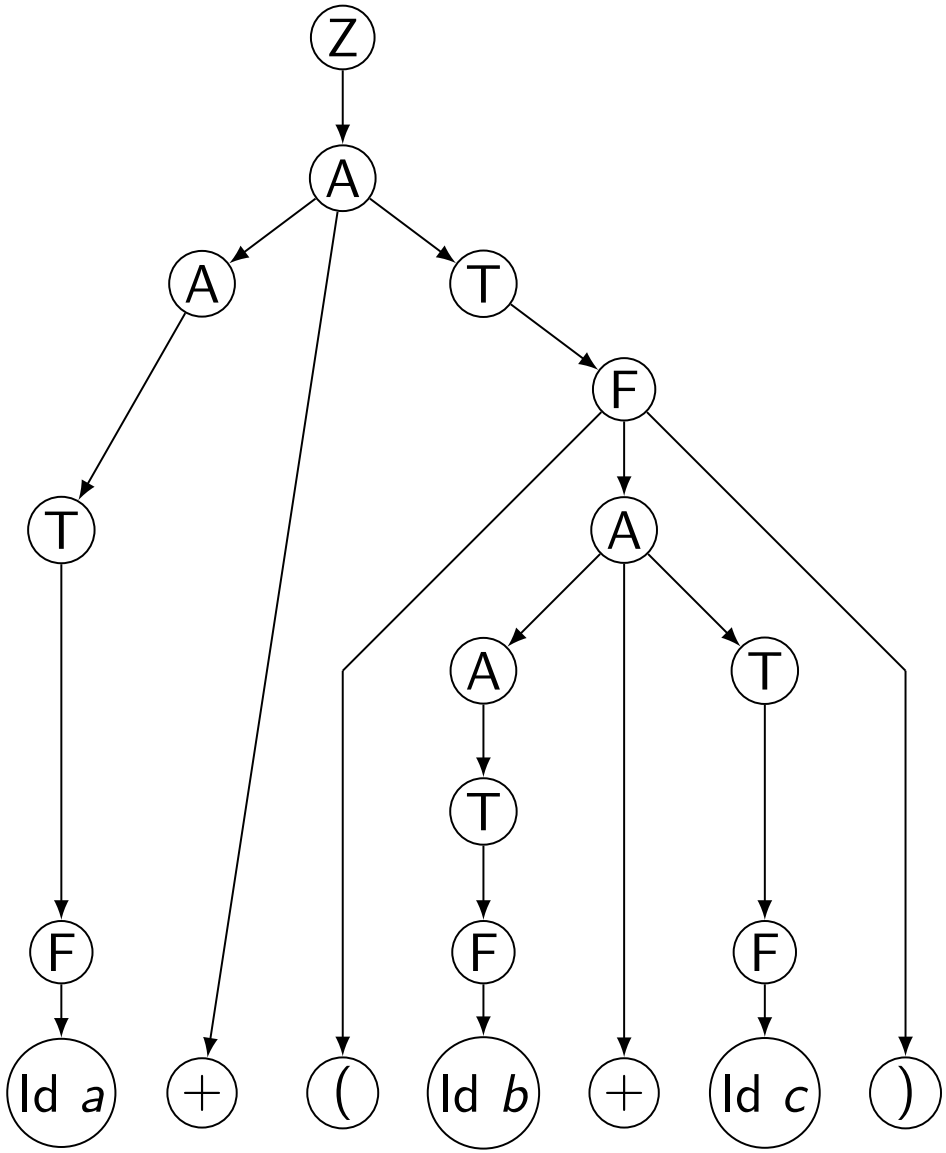
Prinzip der abstrakten Syntax: nur die für die Semantik wichtige Struktur behalten: $A \rightarrow A + A | A * A | \mathrm{bez}$

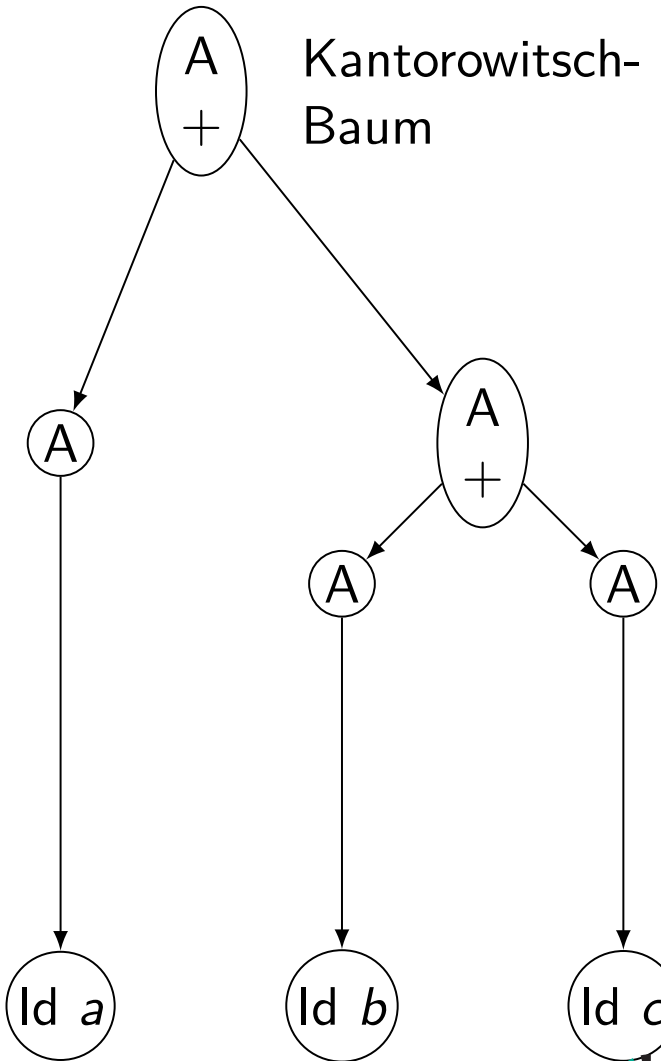




Konkrete und abstrakte Syntax

Prinzip der abstrakten Syntax: nur die für die Semantik wichtige Struktur behalten: $A \rightarrow A + A | A * A | \mathrm{bez}$





Übergang: Konkrete und abstrakte Syntax (1/2)

Konkrete Syntax G_k der zu übersetzenden Sprache (Datenstruktur: Tokenstrom)

1. explizite Strukturinformation ( ), begin end, etc.
2. Ketten- und Verteilerproduktionen $A \longrightarrow B$ bzw. $A \longrightarrow B \mid C \mid \ldots$
3. Schlüsselwörter

Abstrakte Syntax G_a des Strukturbaums (Datenstruktur: Baum, AST)

1. Klammerung durch AST bereits eindeutig
2. Kettenproduktionen überflüssig, wenn keine semantische Bedeutung
3. Schlüsselwörter dienten dem eindeutigen Parsen, jetzt immer überflüssig, werden weggelassen.

Übergang: Konkrete und abstrakte Syntax (2/2)

- Abbildung konkrete auf abstrakte Syntax durch Parser (Verarbeitung von semantischen Aktionen), ggf. weitere Transformation während semantischer Analyse
- Produktionsnummer wird Knotentyp
- Operatoren als Attribute des Knotens für den Ausdruck

Abstrakte Syntax als abstrakte Algebra

Heute fasst man eine abstrakte Syntax als Signatur einer ordnungssortierten Termalgebra auf, und einen AST als Term gemäß dieser Signatur.

- Klassen der abstrakten Syntax entsprechen Sorten der Algebra
- innere Baumknoten entsprechen Operatoren (Funktionssymbolen) der Algebra inkl. Signatur.

Beispiel: abstrakte Syntax für Expressions und Statements

$$
\begin{array}{rcl}
\mathit{Stmt} & = & \mathit{IfStmt} \mid \mathit{IfElseStmt} \mid \mathit{WhileStmt} \mid \mathit{Assignment} \mid \mathit{Block} \mid \ldots \\
\mathit{IfStmt} & :: & \mathit{Expr}\ \mathit{Stmt} \\
\mathit{IfElseStmt} & :: & \mathit{Expr}\ \mathit{Stmt}\ \mathit{Stmt} \\
\mathit{WhileStmt} & :: & \mathit{Expr}\ \mathit{Stmt} \\
\mathit{Block} & :: & \mathit{Decls}\ \mathit{StmtList} \\
\mathit{StmtList} & :: & \mathit{Stmt}\ + \\
\mathit{Assignment} & :: & \mathit{Var}\ \mathit{Expr} \\
\mathit{Var} & = & \mathbf{Bez} \mid \ldots
\end{array}
\qquad
\begin{array}{rcl}
\mathit{Expr} & = & \mathit{Addop} \mid \mathit{MultOp} \mid \mathit{Var} \mid \ldots \\
\mathit{Addop} & :: & \mathit{Expr}\ \mathit{Expr} \\
\mathit{Multop} & :: & \mathit{Expr}\ \mathit{Expr}
\end{array}
$$

Abstrakte Syntax als abstrakte Algebra

Entsprechende Bäume können auch als Terme dargestellt werden.
Schreibweise zur Konstruktion von Termen z.B.

Beispiele

$Addop(Bez(\textbf{hinz}), Bez(\textbf{kunz}))$
$IfStmt(Bez(\textbf{test}), Assignment(Bez(\textbf{x}), Addop(Bez(\textbf{x}), Bez(\textbf{y}))))$
$Block(Decls(\ldots), StmtList(s_1, s_2, \ldots, s_{42}))$

Assoziativitäten/Präzedenzen werden durch Termstruktur dargestellt
Achtung: Die abstrakte Syntax enthält keine semantischen Bedingungen z.B. „Typ einer **If**-Expression muss boolesch sein"

Semantische Aktionen

%Ausgabe

- Nach Erkennen des vorgehenden (Nicht-)Terminals ausgeführt
- Für AST: Konstruktor des entsprechenden Knotens im Ableitungsbaum für G_a aufrufen

&Ausgabe

- Wird ausgeführt, wenn Symbol erkannt aber noch nicht fortgeschaltet wurde
- Für AST: Konstruktoren werden gegebenenfalls Merkmale von Symbolen übergeben

Beachte:

- Semantische Aktionen basieren auf Seiteneffekt beim Parsen
- Symbole werden in der Reihenfolge abgenommen, in der sie in der Symbolfolge erscheinen

Ausgabe von Postfix- oder Präfixform

Ausgaberoutinen:

addop	gib aus '+'
mulop	gib aus '*'
bezeichner	gib aus bez
merke	merke bez
bez_aus	gib gemerkten bez aus, falls vorhanden

Postfixform, d.h. abstrakter Syntaxbaum als Rechtsableitung:

1. $A \rightarrow T\ (\ '+'\ \ T\ \%\text{addop}\ )^*$
2. $T \rightarrow F\ (\ '*'\ \ F\ \%\text{mulop}\ )^*$
3. $F \rightarrow \text{bez}\ \&\text{bezeichner} \mid '('\ A\ ')'$

Präfixform, d.h. abstrakter Syntaxbaum als Linksableitung:

1. $A \rightarrow T\ (\ '+'\ \ \%\text{addop}\ \%\text{bez\_aus}\ \ T\ )^*$
2. $T \rightarrow F\ (\ '*'\ \ \%\text{mulop}\ \%\text{bez\_aus}\ F\ )^*\ \%\text{bez\_aus}$
3. $F \rightarrow \text{bez}\ \&\text{merke} \mid '('\ A\ ')'$

Beispiel Postfixform

1. $A \rightarrow T\ (\ '+'\ \ T\ \%\text{addop}\ )^*$
2. $T \rightarrow F\ (\ '*'\ \ F\ \%\text{mulop}\ )^*$
3. $F \rightarrow \text{bez}\ \&\text{bezeichner} \mid '('\ A\ ')'$

Ableitung für x∗y+z	Ausgabe
A	
$\Rightarrow$ T + T	
$\Rightarrow$ F ∗ F + T	
$\Rightarrow$ x ∗ F + T	x
$\Rightarrow$ x ∗ y + T	xy∗
$\Rightarrow$ x ∗ y + F	xy∗
$\Rightarrow$ x ∗ y + z	xy∗z+

Beispiel Präfixform

1. $A \rightarrow T\ (\ '+'\ \%\text{addop}\ \%\text{bez\_aus}\ T\ )^*$
2. $T \rightarrow F\ (\ '*'\ \%\text{mulop}\ \%\text{bez\_aus}\ F\ )^*\ \%\text{bez\_aus}$
3. $F \rightarrow \text{bez}\ \&\text{merke} \mid '('\ A\ ')'$

Ableitung für x∗y+z	Ausgabe
A	
$\Rightarrow$ T + T	
$\Rightarrow$ F ∗ F + T	
$\Rightarrow$ x ∗ F + T	∗x
$\Rightarrow$ x ∗ y + T	∗x+y
$\Rightarrow$ x ∗ y + F	∗x+y
$\Rightarrow$ x ∗ y + z	∗x+yz

Sonderfälle in abstrakter Syntax

- Bezeichner:
  - $A \rightarrow \mathrm{bez}$ ist Kettenproduktion, soll aber wegen semantischer Analyse erhalten bleiben
- Klammern in Fortran:
  - Information eigentlich bereits in der Baumstruktur
  - aber Klammern sind bindend (kein Umordnen erlaubt)
  - sonst gilt eventuell Assoziativgesetz (Umordnen möglicherweise erlaubt)
  - müssen als Operator gespeichert werden
- Anweisungslisten in C:
  - sind Verteilersymbole
  - aber Strichpunkt-Operator legt Auswertungsfolge fest (auch ohne Datenabhängigkeiten), Code-Verschiebung verboten?

Abstrakte Syntax II

- abstrakte Syntax quellsprachenunabhängig?
  - Programmstruktur in semantischer Analyse aufgearbeitet, danach nur noch Prozeduren interessant
  - Prozeduraufrufe nur bezüglich Parameterübergabe unterschiedlich
  - Ablaufsteuerung identisch, eventuelle Ausnahme: Zählschleifen
  - Ausnahmebehandlung in allen modernen Sprachen identisch
  - Zuweisung, Ausdrucksoperatoren, usw.: identisch, manchmal vielleicht Ergänzungen erforderlich
- Konsequenz: weitere Verarbeitung (Transformation, Optimierung, Codegenerierung) weitgehend unabhängig von der Quellsprache
  - Systeme: UNCOL, ANDF, Dotnet
  - Dotnet kann als Postfixcodierung von UNCOL angesehen werden

Beseitigung von ε-Produktionen

Satz:

Für jede kfG G mit ε-Produktionen gibt es eine kfG G' ohne ε-Produktionen mit $L(G) \setminus \{\varepsilon\} = L(G')$ und umgekehrt.

Technik dazu: ε-Abschluß

Einsetzen von Ableitungen der Form $A \longrightarrow \varepsilon$ in alle rechten Seiten der Form $X \longrightarrow \alpha A \beta$, mit $\alpha, \beta \in (T \cup N)^*$

Beispiel zur Beseitigung von ε-Produktionen

(1) $Z \rightarrow aS$
(2) $S \rightarrow aS \mid \varepsilon$

Einsetzen von $S \rightarrow \varepsilon$ auf den rechten Seiten führt zu

(1) $Z \rightarrow aS \mid a$
(2) $S \rightarrow aS \mid a$

ohne ε-Produktionen.

Linksfaktorisierung

Produktionen $X \to Yb \mid Yc$ mit gleicher LS und gemeinsamem Anfang Y kann man nicht mit rekursivem Abstieg verarbeiten, wenn Länge $|y|$, $Y \Rightarrow^* y$, unbeschränkt, $|y| \geq 0$.

Lösung: den gemeinsamen Anfang ausklammern
Ersetze $X \to Yb \mid Yc$ durch $X \to YX'$, $X' \to b \mid c$
Analog kann man bei LR-Analyse rechtsfaktorisieren (seltener benötigt).

Beispiel zur Linksfaktorisierung

Die Produktionen

$S \rightarrow \text{if } E \text{ then } S \text{ endif}$
$S \rightarrow \text{if } E \text{ then } S \text{ else } S \text{ endif}$

haben gemeinsamen Anfang if E then S.
Linksfaktorisierung ergibt:

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

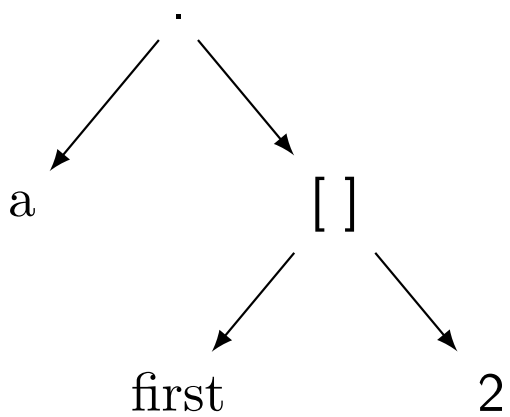
Typunabhängiges Parsen

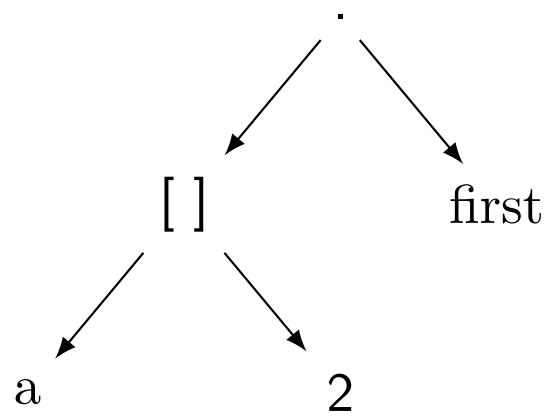
Typunabhängiges Parsen

- Parsen ohne Kenntnis über Typen von Symbolen
- ist üblich, aber nicht immer ausreichend

Typabhängiges Parsen

- Bsp: ADA a.first(2)





Typabhängiges Parsen

Beispiel: Formate in FORTRAN

- print(r 20, real_const)
- r 20 ist Format und muss anders behandelt werden, sonst r Bezeichner und 20 ganze Zahl

Parser umschaltbar, um Formate zu bearbeiten

- D.h., es gibt zwei verschiedene Parser
- Erst semantische Analyse erkennt Bezeichner print

Umschaltung also semantik- (oder typ-) gesteuert

Ähnliche Probleme in ABAP/4

Kellerautomaten

- Kellerautomat $A = (T, Q, R, q_0, F, S, s_0)$
  - T Eingabealphabet (Tokens)
  - Q Zustandsmenge
  - R Menge von Regeln $sqx \rightarrow s'q'x'$, $s, s' \in S^*$, $q, q' \in Q$, $x, x', x'' \in T^*$, $x = x''x'$
  - q_0 Anfangszustand
  - $F \subseteq Q$ Menge von Endzuständen
  - S Kelleralphabet
  - $s_0 \in S$ Anfangszeichen im Keller
- Konfiguration: $\underline{s}q\underline{x}$, $\underline{s}$ vollständiger Kellerinhalt, $\underline{x}$ restliche Eingabe
- Anfangskonfiguration: $s_0 q_0 y$, y vollständige Eingabe
- Regel $sqx \rightarrow s'q'x'$ anwendbar, wenn $\underline{s} = \underline{s'}s, \underline{x} = x\underline{x'}$
- Folgekonfiguration: $\underline{s'}s'q'x'\underline{x'}$
- Halt bei Konfiguration sq, $q \in F$, Eingabe vollständig gelesen
- praktisch Endezeichen # erforderlich, Halt bei $sq\#$

Beispiel: Kellerautomat für Palindrome

Kellerautomat $A = (T, Q, R, q_0, F, S, s_0)$ mit

- $Q = \{q_0, q_1, q_2\}$
- $R = \{q_0 t \to t q_0 \mid t \in T\} \cup \{q_0 t \to q_1 \mid t \in T \cup \{\varepsilon\}\} \cup \{t q_1 t \to q_1 \mid t \in T\} \cup \{s_0 q_1 \# \to q_2 \#\}$
- $F = \{q_2\}$
- $S = T \cup \{s_0\}$

Abarbeitung von *otto*:

Keller	Zustand	Eingabe
s_0	q_0	$otto\#$
$s_0 o$	q_0	$tto\#$
$s_0 ot$	q_0	$to\#$
$s_0 ot$	q_1	$to\#$
$s_0 o$	q_1	$o\#$
s_0	q_1	$\#$
	q_2	$\#$

Kontextfreie Grammatik und Kellerautomaten

Satz:
Für jede kontextfreie Grammatik G gibt es einen (nichtdeterministischen) Kellerautomaten A mit $L(A) = L(G)$.

$\Rightarrow$ das Akzeptionsproblem für kontextfreie Sprachen ist entscheidbar

Aber: Aufwand i.a. $\mathcal{O}(n^3)$
$\Rightarrow$ praktisch nur Teilklassen mit linearem Aufwand brauchbar, dazu Grammatik-Umformungen erforderlich

Aber: Sprachinklusion und Gleichheit nicht entscheidbar
$\Rightarrow$ keine eindeutige Normalform

Systematische Parserkonstruktion

- Es gibt weit mehr als 25 verschiedene Techniken zur Parserkonstruktion, vgl. Aho&Ullman, The Theory of Parsing and Compiling, 2 Bde, 1972
- Nur zwei Techniken , LL und LR, haben die Eigenschaften:
  - Der Parser liest die Quelle einmal von links nach rechts und baut dabei die Links- bzw. Rechtsableitung auf (daher die 2 Buchstaben).
  - Der Parser erkennt einen Fehler beim ersten Zeichen t, das nicht zu einem Satz der Sprache gehören kann. t heißt parserdefinierte Fehlerstelle (parser defined error): Wenn $x \notin L(G)$ und der Parser erkennt den Fehler beim Zeichen t, $x = x'tx''$, so gibt es einen Satz $y \in L(G)$ mit $y = x'y'$.
  - Alternative: Erkennen des Fehlers einige Zeichen später, keine syntaktische Fehlerlokalisierung möglich.

Kapitel 3: Syntaktische Analyse

Herleitung der LL- und LR-Parser

- gegeben Grammatik $G = (T, N, P, Z)$, $V = T \cup N$, konstruiere indeterministischen Kellerautomat mit genau einem Zustand q, angesetzt auf Eingabe x

Für LL: (prädiktiv)	Für LR: (reduzierend)
$tqt \rightarrow q,\ t \in T$	$qt \rightarrow tq,\ t \in T$
$Xq \rightarrow x_n \ldots x_1 q,$	$x_1 \ldots x_n q \rightarrow Xq,$
$X \rightarrow x_1 \ldots x_n \in P$	$X \rightarrow x_1 \ldots x_n \in P$

- mache Kellerautomat deterministisch durch Hinzunahme Rechtskontext, also Vorhersage $Xqx' \rightarrow x_n \ldots x_1 qx'$ bzw. Reduktion $x_1 \ldots x_n qx' \rightarrow Xqx'$
  x' Anfang des unverarbeiteten Eingaberests
- deterministisch machen geht nur für eingeschränkte Grammatikklassen

Nichtdeterministische LL- und LR-Parser

Für LL: (prädiktiv)	Für LR: (reduzierend)
Vergleich (compare):	Schift (shift):
$tqt \longrightarrow q,\ t \in T$	$qt \longrightarrow tq,\ t \in T$
Vorhersage (produce):	Reduktion (reduce):
$Xq \longrightarrow x_n \ldots x_1 q,$	$x_1 \ldots x_n q \longrightarrow Xq,$
$X \longrightarrow x_1 \ldots x_n \in P$	$X \longrightarrow x_1 \ldots x_n \in P$

top-down Parser
vom Startsymbol zum Wort

bottom-up Parser
vom Wort zum Startsymbol

Anmerkung: Der Zustand q ist noch bedeutungslos, er wird später beim deterministisch Machen benötigt.

Textmengen

$k : x$

$$k : x = x\# \quad \text{falls } x = x_1 \dots x_m \wedge m < k$$
$$k : x = x_1 \dots x_k \quad \text{falls } x = x_1 \dots x_m \wedge m \geq k$$

$\mathrm{Anf}_k(x) = \{u \mid \exists y \in T^* : x \Rightarrow^* y \wedge u = k : y\}$
in der Literatur auch $\mathrm{First}_k(x)$ genannt

$A \Rightarrow_{R'} \alpha$ gdw. $A \Rightarrow_R \alpha \wedge \nexists B \in N : A \Rightarrow_R B\alpha \Rightarrow \alpha$

$\mathrm{Anf}'_k(x) = \{u \in \mathrm{Anf}_k(x) \mid \exists y \in T^* : x \Rightarrow_{R'} uy\}$
in der Literatur auch $\mathrm{EFF}_k(x)$ genannt (EFfective First)

$\mathrm{Folge}_k(x) = \{u \mid \exists m, y \in V^* : Z \Rightarrow^* mxy \wedge u \in \mathrm{Anf}_k(y)\}$
in der Literatur auch $\mathrm{Follow}_k(x)$ genannt

LL(k)-Grammatiken

Für $k \geq 1$ heißt eine kfG $G = (T, N, P, Z)$ eine LL(k)-Grammatik, wenn für alle Paare von Ableitungen

$$\begin{array}{ll} Z \Rightarrow_L \mu A \chi \Rightarrow \mu\nu\chi \Rightarrow^* \mu\gamma & \mu, \gamma \in T^*; \nu, \chi \in V^*, A \in N \\ Z \Rightarrow_L \mu A \chi' \Rightarrow \mu\omega\chi' \Rightarrow^* \mu\gamma' & \gamma' \in T^*; \omega, \chi' \in V^* \end{array}$$

gilt:

$$(k : \gamma = k : \gamma') \Rightarrow \nu = \omega$$

Also: Aus den nächsten k Zeichen kann unter Berücksichtigung des Kellerinhalts die nächste anzuwendende Produktion eindeutig vorhergesagt werden.

Die k Zeichen können aus der Produktion resultieren oder ganz oder teilweise dem Folgetext angehören, z.B. bei ε-Produktionen.

Beispiele von LL-Grammatiken

- A $\rightarrow$ TA', A' $\rightarrow$ ε | '+' TA', T $\rightarrow$ FT', T' $\rightarrow$ ε | '$*$' FT', F $\rightarrow$ **bez** | '(' A ')' ist $\mathrm{LL}(1)$.
- Z $\rightarrow$ aAab | bAbb, A $\rightarrow$ ε | a ist $\mathrm{LL}(2)$, nicht $\mathrm{LL}(1)$: Vorschau *aa*, *ab*, *bb* entscheidet unter Berücksichtigung der Produktion für Z über die Produktion für A.
- Z $\rightarrow$ X, X $\rightarrow$ Y | bYa, Y $\rightarrow$ c | ca ist $\mathrm{LL}(3)$.
- Z $\rightarrow$ X, X $\rightarrow$ Yc | Yd, Y $\rightarrow$ a | bY ist für kein k $\mathrm{LL}(k)$; aber Linksfaktorisieren macht daraus $\mathrm{LL}(1)$.
- Anweisungen, die mit Schlüsselwort while, if, case, usw. beginnen, sind mit LL(1)-Technik vorhersagbar. Bei Beginn mit Bezeichner sind Linksfaktorisierungen nötig.

Satz über Linksrekursion

Satz:

Eine linksrekursive kfG ist für kein k $\mathrm{LL}(k)$.

Beweisidee:

Seien A $\rightarrow$ Ax und A $\rightarrow$ y linksrekursive bzw. terminierende Regeln. Jeder k-Anfang der terminierenden Regel ist auch k-Anfang der linksrekursiven Regel.

Elimination von Linksrekursion (1/2)

Satz:

Für jede kfG G mit linksrekursiven Produktionen gibt es eine kfG G' ohne Linksrekursion mit $L(G) = L(G')$.

Elimination von Linksrekursion (2/2)

Konstruktion:

- Nummeriere Nichtterminale beliebig $X_1, \ldots, X_n$
- Für $i = 1, \ldots, n$
  - Für $j = 1, \ldots, i-1$ ersetze $X_i \to X_j x$ durch $\{X_i \to y_j x \mid X_j \to y_j \in P\}$ (danach $i \leq j$, wenn $X_i \to X_j x \in P$)
  - Ersetze die Produktionenmengen $\{X_i \to X_i x\} \cup \{X_i \to z \mid z \neq X_i z'\}$ durch $\{Y_i \to x Y_i \mid X_i \to X_i x \in P\} \cup \{Y_i \to \varepsilon\} \cup \{X_i \to z Y_i \mid X_i \to z \in P \wedge z \neq X_i z'\}$ mit einem neuen Nichtterminal Y_i. (Nummerierung der Y_i mit $n+1, n+2, \ldots$)
- Ergebnis: $i < j$, wenn $X_i \to X_j x \in P$

Beachte: in Schritt 2 Ersetzung durch $\{Y_i \to x, Y_i \to x Y_i \mid X_i \to X_i x \in P\} \cup \{X_i \to z, X_i \to z Y_i \mid X_i \to z \in P \wedge z \neq X_i z'\}$ ohne ε-Produktionen möglich, wenn x nicht mit X_j, $j \leq i$, beginnt.

Beispiel

- A $\rightarrow$ T | A + T , T $\rightarrow$ F | T $*$ F, F $\rightarrow$ **bez** | (A) ist linksrekursiv
- Ersetzung: Schritt 1 leer, Schritt 2: A $\rightarrow$ T | A + T durch A $\rightarrow$ T A', A' $\rightarrow$ ε | + TA' ersetzen; T $\rightarrow$ F | T $*$ F analog. Dies entspricht der EBNF A $\rightarrow$ T ('+' T)*, T $\rightarrow$ F ('$*$' F)*, F $\rightarrow$ **bez** | '(' A ')'.
- Andere mögliche Ersetzung A $\rightarrow$ T | T A', A' $\rightarrow$ + T | TA'
- Vorsicht: Die Ersetzung durch A $\rightarrow$ T | T + A ist semantisch unzulässig!
- Sie transformiert Links- in Rechtsassoziativität, verändert also die semantisch bedeutungsvolle Struktur.
- Beseitigung von Linksrekursion bei $\mathrm{LL}(k)$-Analyse nötig für alle Anweisungen, die mit <Bezeichner> <Operator> anfangen können (Zuweisungen, Ausdrücke)

SLL(k)-Grammatiken

Für $k \geq 1$ heißt eine kfG $G = (T, N, P, Z)$ eine SLL(k)-Grammatik (starke LL-Grammatik), wenn für alle Paare von Ableitungen

$$\begin{array}{ll} Z \Rightarrow_L \mu A \chi \Rightarrow \mu\nu\chi \Rightarrow^* \mu\gamma & \mu, \gamma \in T^*; \nu, \chi \in V^*, A \in N \\ Z \Rightarrow_L \mu' A \chi' \Rightarrow \mu'\omega\chi' \Rightarrow^* \mu'\gamma' & \mu', \gamma' \in T^*; \omega, \chi' \in V^* \end{array}$$

gilt:

$$(k : \gamma = k : \gamma') \Rightarrow \nu = \omega$$

Also: Aus den nächsten k Zeichen kann ohne Berücksichtigung des Kellerinhalts die nächste anzuwendende Produktion eindeutig vorhergesagt werden.

SLL-Bedingung

Satz:

Eine Grammatik ist genau dann eine $\mathrm{SLL}(k)$-Grammatik, wenn für alle Paare von Produktionen A $\rightarrow$ x | x', $x \neq x'$, die $\mathrm{SLL}(k)$-Bedingung gilt:

$$\mathrm{Anf}_k(x\mathrm{Folge}_k(A)) \cap \mathrm{Anf}_k(x'\mathrm{Folge}_k(A)) = \emptyset$$

Beweis: trivial

- Also: SLL(k)-Eigenschaft durch Berechnung von Anf_k- und Folge_k- Mengen einfach nachzuprüfen.
- Wenn aus x, x' nur terminale Zeichenreihen mit mindestens k Zeichen ableitbar sind, trägt Folge_k(A) nichts zum Ergebnis bei und kann entfallen.
- wichtiger Spezialfall: $k = 1$, $x \not\Rightarrow^* \varepsilon, x' \not\Rightarrow^* \varepsilon$. Dann muss

$$\mathrm{Anf}_k(x) \cap \mathrm{Anf}_k(x') = \emptyset$$

gelten. Falls $x \Rightarrow^* \varepsilon$, so muss außerdem gelten:

$$\mathrm{Folge}_k(A) \cap \mathrm{Anf}_k(x') = \emptyset$$

LL(1) und SLL(1)

Satz: Jede $\mathrm{SLL}(k)$-Grammatik ist auch eine $\mathrm{LL}(k)$-Grammatik.

Satz: Jede $\mathrm{LL}(1)$-Grammatik ist eine $\mathrm{SLL}(1)$-Grammatik.

Beweis (indirekt):

Angenommen, G ist $\mathrm{LL}(1)$, aber die $\mathrm{SLL}(1)$-Bedingung ist nicht erfüllt. Dann gibt es Produktionen $A \to x \mid x'$, $x \neq x'$, und ein Zeichen

$$t \in \mathrm{Anf}_1(x\mathrm{Folge}_1(A)) \cap \mathrm{Anf}_1(x'\mathrm{Folge}_1(A)).$$

Fall $t \in \mathrm{Anf}_1(x)$, $t \in \mathrm{Anf}_1(x')$ verstößt gegen die LL(1)-Definition, da wegen $x = \nu, t \in \mathrm{Anf}_1(\nu\chi)$ und $x' = \omega, t \in \mathrm{Anf}_1(\omega\chi')$ gilt:
$1 : \gamma = 1 : \gamma'$, jedoch $\nu \neq \omega$. Widerspruch.
Andere Fälle analog.

Satz nicht auf k $>$ 1 verallgemeinerbar:
Z $\to$ aAab | bAbb, A $\to$ ε | a ist $\mathrm{LL}(2)$, aber nicht $\mathrm{SLL}(2)$.

Konstruktion der $\mathrm{LL}(1)$-Tabelle

$$LL[X, \mathbf{a}] = \{X \to X_1 \dots X_n \in P \mid \mathbf{a} \in \mathrm{Anf}_1(X_1 \dots X_n \mathrm{Folge}_1(X))\}$$

Es muss gelten $|LL[X, \mathbf{a}]| = 1$ für alle X, $\mathbf{a}$, sonst ist die Grammatik nicht $\mathrm{LL}(1)$.

Parsertabelle

Grammatik:

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \varepsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

Nichtterminal	Anf_1	Folge_1	**Eingabesymbol**					
			id	**+**	*****	**(**	**)**	**#**
E	$(, \mathbf{id}$	$), \#$	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'	$+, \#$	$), \#$		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$(, \mathbf{id}$	$+, \#$	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'	$*, \#$	$+, \#$		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$(, \mathbf{id}$	$+, *, \#$	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

Abbildung: Parsertabelle

Parsertabelle

Grammatik:

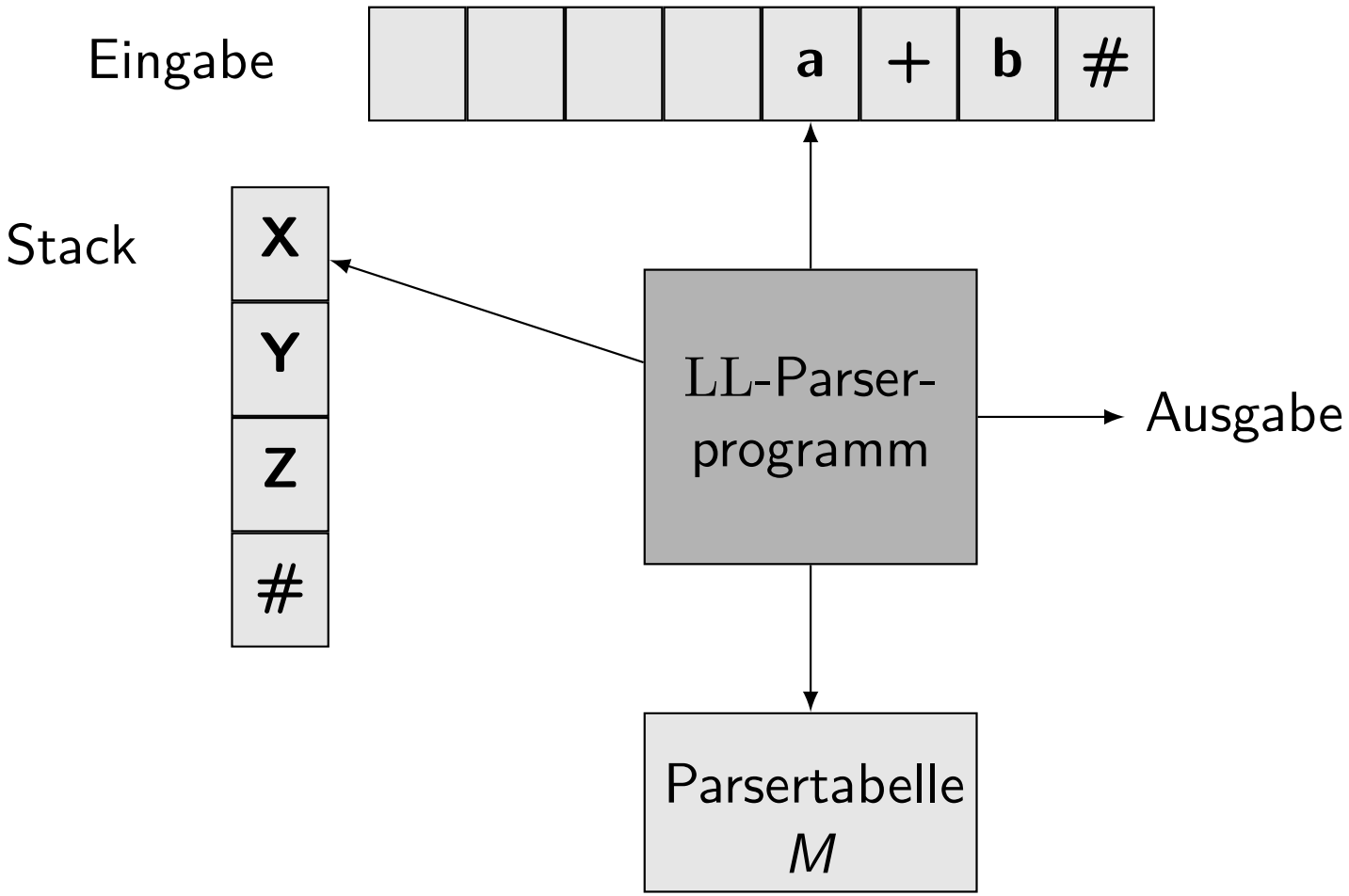
$$
\begin{aligned}
S &\rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ S' \mid \mathbf{a} \\
S' &\rightarrow \mathbf{e}\ S \mid \varepsilon \\
E &\rightarrow \mathbf{b}
\end{aligned}
$$

Parsertabelle:

Nichtterminal	Anf_1	Folge_1	**Eingabesymbol**					
			a	**b**	**e**	**i**	**t**	**#**
S	**i**, **a**	**e**, #	$S \rightarrow \mathbf{a}$			$S \rightarrow \mathbf{i}E\mathbf{t}SS'$		
S'	**e**, #	**e**, #			$S' \rightarrow \mathbf{e}S$			$S' \rightarrow \varepsilon$
E	**b**	**t**		$E' \rightarrow \mathbf{b}$				

Nach Definition ist $LL[S', \mathbf{e}] = \{S' \rightarrow eS, S' \rightarrow \varepsilon\}$ und somit die Grammatik nicht $\mathrm{LL}(1)$. Zur Auflösung des Konfliktes wird die zweite Produktion manuell aus dem Tabelleneintrag entfernt. Dadurch gehört ein **e** immer zum letzten **i**.

Modell eines tabellengesteuerten LL-Parsers



Verhalten eines LL-Parsers

Übereinstimmung	Stack	Eingabe	Aktion
	$E\#$	**id+id*id#**	
	$TE'\#$	**id+id*id#**	Ausgabe von $E \rightarrow TE'$
	$FT'E'\#$	**id+id*id#**	Ausgabe von $T \rightarrow FT'$
	id $T'E'\#$	**id+id*id#**	Ausgabe von $F \rightarrow$ **id**
id	$T'E'\#$	**+id*id#**	Übereinstimmung mit **id**
id	$E'\#$	**+id*id#**	Ausgabe von $T' \rightarrow \varepsilon$
id	$+TE'\#$	**+id*id#**	Ausgabe von $E' \rightarrow +TE'$
id+	$TE'\#$	**id*id#**	Übereinstimmung mit **+**
id+	$FT'E'\#$	**id*id#**	Ausgabe von $T \rightarrow FT'$
id+	**id** $T'E'\#$	**id*id#**	Ausgabe von $F \rightarrow$ **id**
id+id	$T'E'\#$	***id#**	Übereinstimmung mit **id**
id+id	$*FT'E'\#$	***id#**	Ausgabe von $T' \rightarrow *FT'$
id+id*	$FT'E'\#$	**id#**	Übereinstimmung mit *****
id+id*	**id** $T'E'\#$	**id#**	Ausgabe von $F \rightarrow$ **id**
id+id*id	$T'E'\#$	#	Übereinstimmung mit **id**
id+id*id	$E'\#$	#	Ausgabe von $T' \rightarrow \varepsilon$
id+id*id	#	#	Ausgabe von $E' \rightarrow \varepsilon$

LL(1)-Parseralgorithmus

```
push('#'); push(Z); t = next_token();
while (t != '#') {
    if (stackEmpty()) { error("end of input expected"); }
    else if (top() ∈ T) {
        if (top() == t) {
            pop(); t = next_token();
        } else { error(top() + " expected"); pop(); }
    } else if (LL[top(), t] == ⊥) {
        error("illegal Symbol " + t); t = next_token();
    } else {
        (X → X_1 ... X_n) = LL[top(), t];
        pop();
        for(i = n; i >= 1; --i)
            push(X_i);
    }
}
if (top() != '#')
    error("unexpected end of input");
```

LL(1)-Parser mit rekursivem Abstieg

1. Definiere Prozedur X für alle Nichtterminale X
2. Für alternative Produktionen $X \rightarrow X_1 \mid \ldots \mid X_n$ sei Rumpf von X

   switch t $\{$
   case $\mathrm{Anf}_1(X_1\mathrm{Folge}_1(X))$: Code für X_1;
   $\ldots$
   case $\mathrm{Anf}_1(X_n\mathrm{Folge}_1(X))$: Code für X_n;
   default : Fehler(...);
   $\}$
3. Für rechte Seite $X_i = Y_1 \ldots Y_m$ erzeuge:

   $C_1; \ldots; C_m$; **return**;

   Es gilt $C_i =$

   1. **if** (t == Y_i) t = nächstesSymbol() **else** Fehler(...);

      wenn $Y_i \in T$
   2. $Y_i()$;

      wenn $Y_i \in N$

Parser aus Grammatik in EBNF

Nichtterminal	X	$X()$;
Terminal	t	**if** (symbol $==$ t) nextSymbol(); **else** Fehler();
Option	$[X]$	**if** (symbol $\in \mathrm{Anf}_1(X)$) $X()$;
Iteration	X^+	**do** X(); **while** (symbol $\in \mathrm{Anf}_1(X)$);
	X^*	**while** (symbol $\in \mathrm{Anf}_1(X)$) $X()$;
Liste	$X\|\|d$	$X()$; **while** (symbol $\in \mathrm{Anf}_1(d)$) { d(); X(); }
semantische Aktion	t&Y	**if** (symbol $==$ t) { Y(); nextSymbol(); } **else** Fehler();
	$\%Z$	$Z()$;

Beispielgrammatik

Beispielgrammatik in EBNF-Notation zum Parser auf der nächsten Folie:

1. $Z ::= A$
2. $A ::= T\ \{\text{'+'}\ T\}^*$
3. $T ::= F\ \{\text{'*'}\ F\}^*$
4. $F ::= id \mid \text{'('}A\text{')'}$

Parser für Beispielgrammatik (1/2)

```
AST parse() { t = nextSymbol(); return Z(); }
AST Z() { return A(); }
AST A() {
    AST res = T(); // merke 1. Operand
    while (t == '+') {
        t = nextSymbol();
        AST res1 = new AST(plus);
        res1.left = res;
        res1.right = T();
        res = res1;
    }
    return res;
}
T() // analog A
```

Hinweis: Additionen/Multiplikationen werden im AST linksassoziativ interpretiert.

Parser für Beispielgrammatik (2/2)

```
AST F() {
    AST res = null;
    if (t == bez) {
        res = new AST(t); t=nextSymbol();
    }
    else if (t == '(') {
        t = nextSymbol(); res = A();
        if (t == ')')
            t = nextSymbol();
        else
            Fehlerbehandlung.Fehlereintrag(KlammerZuFehlt, t.pos);
    }
    else
        Fehlerbehandlung.Fehlereintrag(unzulässigesSymbol, t.pos);
    return res;
}
```

Praxis des rekursiven Abstiegs

- Einfügung von semantischen Aktionen:
  Semantische Aktion formal wie Produktion $A \longrightarrow \varepsilon$
  behandeln, statt der Prozedur für ein Nichtterminal A die Ausgabeprozedur aufrufen.
- Rekursiver Abstieg baut Linksableitung auf.
  Vorteil: beim Aufbau bereits erste Berechnungen von semantischen Attributen möglich (s. Kapitel "Semantische Analyse").
- Problem: Durch die Handprogrammierung können leicht während der Wartung syntaktische Eigenschaften eingeschleust werden, die die Systematik der Syntax und die Unabhängigkeit Syntax-Semantik zerstören. Negativbeispiel: ABAP 4
- Rekursiver Abstieg kann auch tabellengesteuert implementiert werden! Parser wird Interpretierer der Tabelle.
  Vorteile: Vermeidung von Prozeduraufrufen, einfachere Fehlerbehandlung. Nachteil: nicht von Hand programmierbar.

Situationen (Items)

Ziel: bei Prüfung der Anwendbarkeit von Regeln sqx $\rightarrow$ s'q'x' Kellerinhalt und Zustand sq mit einem Zustandssymbol codieren (Prüfung mehrerer Einträge im Keller vermeiden)

Lösungsidee:

- bei LL und LR ist s rechte bzw. linke Seite einer Produktion $X \rightarrow x_1 \dots x_n$
- Übergänge $tqt \rightarrow q$ (bei LL) bzw. $qt \rightarrow tq$ (bei LR) sind nur zulässig, wenn in der Produktion ein Terminalzeichen t ansteht, $x_1 \dots x_n = x' t x'''$, wobei $x'' := tx'''$
- also: ersetze sqx durch Situation $[X \rightarrow x' \cdot x''; x]$, die durch den Punkt anzeigt, wie weit die Produktion abgearbeitet ist.
- Situationen $[X \rightarrow \cdot\ x''; x]$ oder $[X \rightarrow x' \cdot; x]$ sind erlaubt und notwendig.
- Verwende Situationen als Zustände und als Kellersymbole.
- Situationen heißen englisch *items.*

Formale Konstruktion $\mathrm{LL}(k)$-Automat

1. Initial $Q = \{q_0\}$ und $R = \emptyset$, mit $q_0 = [Z \to \cdot S; \#]$.
   Anfangszustand und erster Kellerzustand q_0.
   Hinweis: $\mathrm{Folge}_k(Z) = \{\#\}$.
2. Sei $q = [X \to \mu \cdot \nu; \Omega] \in Q$ und noch nicht betrachtet.
3. Wenn $\nu = \varepsilon$ setze $R := R \cup \{q\varepsilon \to \varepsilon\}$
   Auskellern $q'q \to q'$ mit beliebigem q'.
4. Wenn $\nu = t\gamma$ mit $t \in T$ und $\gamma \in V^*$, setze
   $q' = [X \to \gamma t \cdot \gamma; \Omega]$. Setze $Q := Q \cup \{q'\}$ und
   $R := R \cup \{qt \to q'\}$.
5. Wenn $\nu = B\gamma$ mit $B \in N$ und $\gamma \in V^*$, setze
   $q' = [X \to \gamma B \cdot \gamma; \Omega]$ und
   $H = \{[B \to \cdot\beta_i; \mathrm{Anf}_k(\gamma\Omega)] \mid B \to \beta_i \in P\}$.
   Hinweis: $1 \leq i \leq m$, wenn es m Produktionen mit linker Seite
   B gibt. Setze $Q := Q \cup \{q'\} \cup H$ und
   $R := R \cup \{q\tau_i \to q'h_i\tau_i \mid h_i \in H, \tau_i \in \mathrm{Anf}_k(\beta_i\gamma\Omega)\}$.
6. Wenn alle $q \in Q$ betrachtet wurden, stop. Sonst, gehe zu 2.

Formale Konstruktion $\mathrm{SLL}(k)$-Automat

1. Initial $Q = \{q_0\}$ und $R = \emptyset$, mit $q_0 = [Z \to \cdot S; \#]$.
   Anfangszustand und erster Kellerzustand q_0.
   Hinweis: $\mathrm{Folge}_k(Z) = \{\#\}$.
2. Sei $q = [X \to \mu \cdot \nu; \Omega] \in Q$ und noch nicht betrachtet.
3. Wenn $\nu = \varepsilon$ setze $R := R \cup \{q\varepsilon \to \varepsilon\}$
   Auskellern $q'q \to q'$ mit beliebigem q'.
4. Wenn $\nu = t\gamma$ mit $t \in T$ und $\gamma \in V^*$, setze
   $q' = [X \to \gamma t \cdot \gamma; \Omega]$. Setze $Q := Q \cup \{q'\}$ und
   $R := R \cup \{qt \to q'\}$.
5. Wenn $\nu = B\gamma$ mit $B \in N$ und $\gamma \in V^*$, setze
   $q' = [X \to \gamma B \cdot \gamma; \Omega]$ und
   $H = \{[B \to \cdot\beta_i; \mathrm{Folge}_k(B)] \mid B \to \beta_i \in P\}$.
   Hinweis: $1 \leq i \leq m$, wenn es m Produktionen mit linker Seite B gibt. Setze $Q := Q \cup \{q'\} \cup H$ und
   $R := R \cup \{q\tau_i \to q'h_i\tau_i \mid h_i \in H, \tau_i \in \mathrm{Anf}_k(\beta_i\mathrm{Folge}_k(B))\}$.
6. Wenn alle $q \in Q$ betrachtet wurden, stop. Sonst, gehe zu 2.

Formale Konstruktion $\mathrm{SLL}(k)$-Automat

Einzig Regel 5 der $\mathrm{LL}(k)$ Konstruktion ändert sich:

5 Wenn $\nu = B\gamma$ mit $B \in N$ und $\gamma \in V^*$, setze
$q' = [X \to \gamma B \cdot \gamma; \Omega]$ und
$H = \{[B \to \cdot\beta_i; \mathrm{Folge}_k(B)] \mid B \to \beta_i \in P\}$.
Hinweis: $1 \leq i \leq m$, wenn es m Produktionen mit linker Seite B gibt. Setze $Q := Q \cup \{q'\} \cup H$ und
$R := R \cup \{q\tau_i \to q' h_i \tau_i \mid h_i \in H, \tau_i \in \mathrm{Anf}_k(\beta_i \mathrm{Folge}_k(B))\}$.

Kapitel 3: Syntaktische Analyse

Nichtdeterministische LL- und LR-Parser

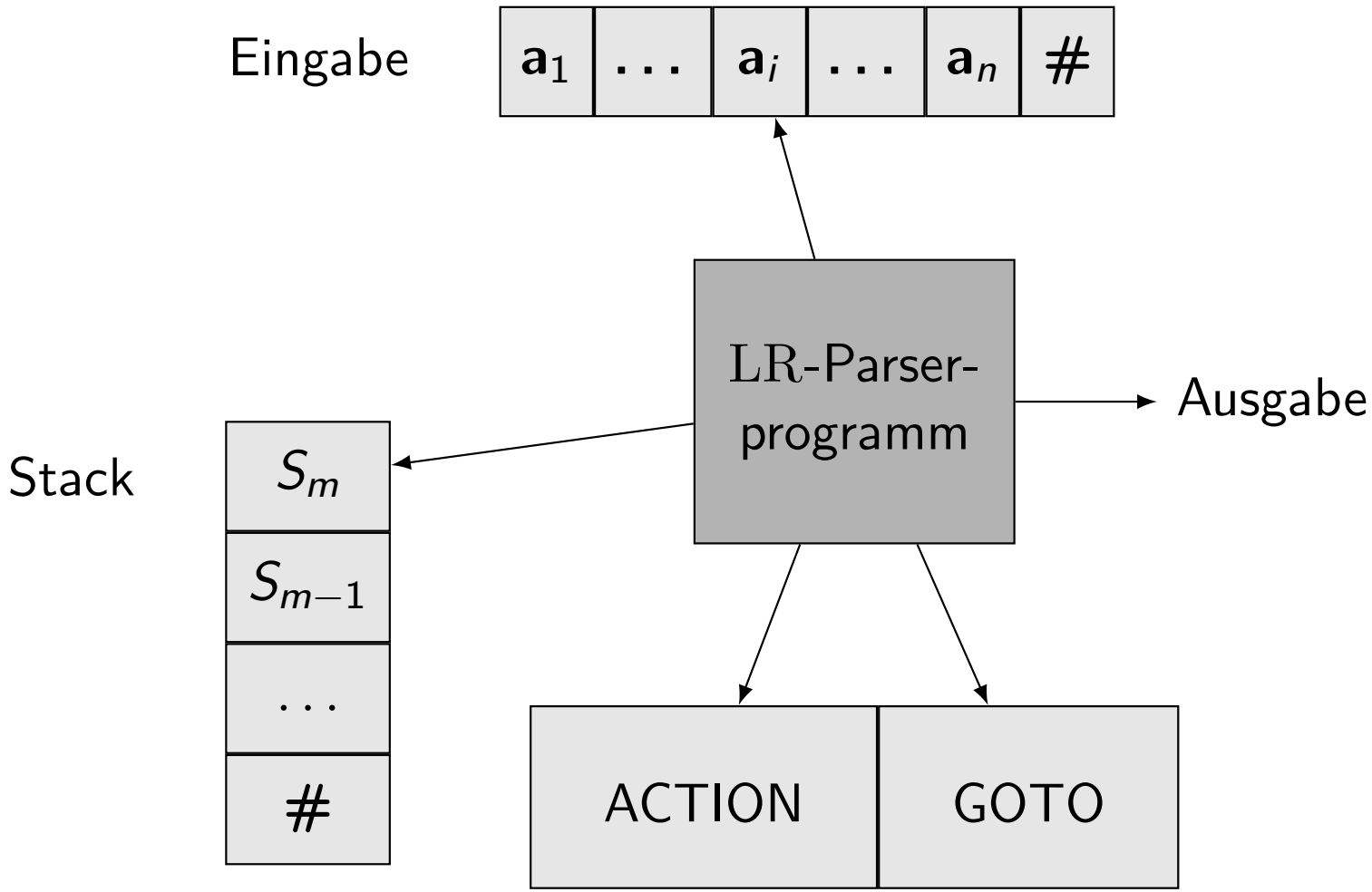
Für LL: (prädiktiv)	Für LR: (reduzierend)
Vergleich (compare):	Schift (shift):
$tqt \longrightarrow q,\ t \in T$	$qt \longrightarrow tq,\ t \in T$
Vorhersage (produce):	Reduktion (reduce):
$Xq \longrightarrow x_n \ldots x_1 q,$	$x_1 \ldots x_n q \longrightarrow Xq,$
$X \longrightarrow x_1 \ldots x_n \in P$	$X \longrightarrow x_1 \ldots x_n \in P$

top-down Parser
vom Startsymbol zum Wort

bottom-up Parser
vom Wort zum Startsymbol

Anmerkung: Der Zustand q ist noch bedeutungslos, er wird später beim deterministisch Machen benötigt.

Modell eines LR-Parsers



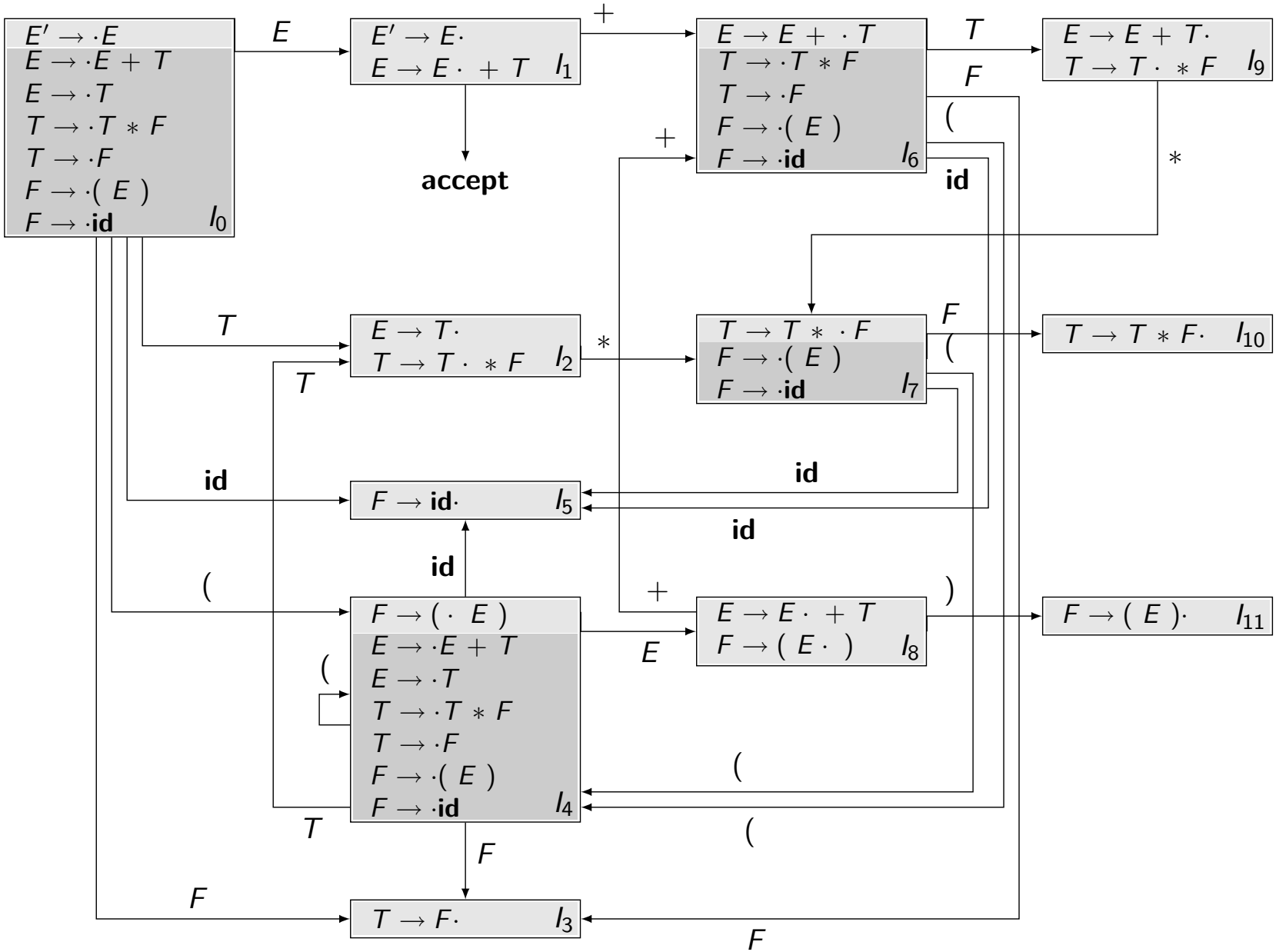
Beispiel: Grammatik G

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Verhalten eines Shift-Reduce-Parsers

Stack	**Eingabe**	**Aktion**
$\#$	$\mathbf{id}_1 * \mathbf{id}_2 \#$	Verschieben (shift)
$\# \mathbf{id}_1$	$* \mathbf{id}_2 \#$	Reduzieren durch $F \rightarrow \mathbf{id}$
$\# F$	$* \mathbf{id}_2 \#$	Reduzieren durch $T \rightarrow F$
$\# T$	$* \mathbf{id}_2 \#$	Verschieben
$\# T *$	$\mathbf{id}_2 \#$	Verschieben
$\# T * \mathbf{id}_2$	$\#$	Reduzieren durch $F \rightarrow \mathbf{id}$
$\# T * F$	$\#$	Reduzieren durch $T \rightarrow T * F$
$\# T$	$\#$	Reduzieren durch $E \rightarrow T$
$\# E$	$\#$	Akzeptieren

LR(0)-Automat für Grammatik G



Parsertabelle für Ausdrucksgrammatik

Zustand	ACTION						GOTO		
	id	+	*	(	)	#	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r3	s7		r3	r3			
3		r5	r5		r5	r5			
4	s5			s4			8	2	3
5		r7	r7		r7	r7			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r2	s7		r2	r2			
10		r4	r4		r4	r4			
11		r6	r6		r6	r6			

Syntaxanalyse von **id * id**

Stack	**Symbole**	**Eingabe**	**Aktion**
0	#	**id** $*$ **id** #	Verschieben (shift)
0 5	# **id**	$*$ **id** #	Reduzieren durch $F \rightarrow$ **id**
0 3	# F	$*$ **id** #	Reduzieren durch $T \rightarrow F$
0 2	# T	$*$ **id** #	Verschieben
0 2 7	# T $*$	**id** #	Verschieben
0 2 7 5	# T $*$ **id**	#	Reduzieren durch $F \rightarrow$ **id**
0 2 7 10	# T $*$ F	#	Reduzieren durch $T \rightarrow T * F$
0 2	# T	#	Reduzieren durch $E \rightarrow T$
0 1	# E	#	Akzeptieren

Verhalten eines LR-Parsers

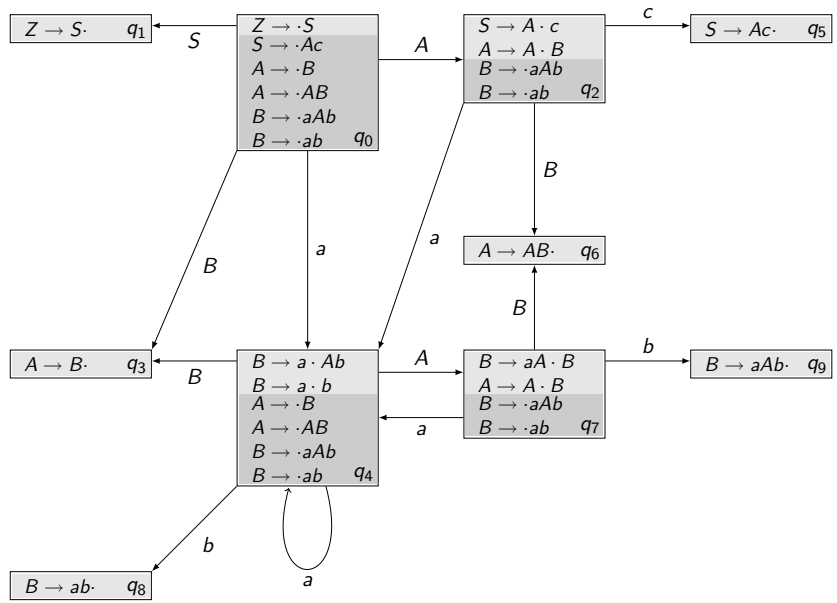
Stack	**Symbole**	**Eingabe**	**Aktion**
0		**id** $*$ **id** $+$ **id** $\#$	Verschieben
0 5	**id**	$*$ **id** $+$ **id** $\#$	Reduzieren durch $F \rightarrow$ **id**
0 3	F	$*$ **id** $+$ **id** $\#$	Reduzieren durch $T \rightarrow F$
0 2	T	$*$ **id** $+$ **id** $\#$	Verschieben
0 2 7	$T *$	**id** $+$ **id** $\#$	Verschieben
0 2 7 5	$T *$ **id**	$+$ **id** $\#$	Reduzieren durch $F \rightarrow$ **id**
0 2 7 10	$T * F$	$+$ **id** $\#$	Reduzieren durch $T \rightarrow T * F$
0 2	T	$+$ **id** $\#$	Reduzieren durch $E \rightarrow T$
0 1	E	$+$ **id** $\#$	Verschieben
0 1 6	$E +$	**id** $\#$	Verschieben
0 1 6 5	$E +$ **id**	$\#$	Reduzieren durch $F \rightarrow$ **id**
0 1 6 3	$E + F$	$\#$	Reduzieren durch $T \rightarrow F$
0 1 6 9	$E + T$	$\#$	Reduzieren durch $E \rightarrow E + T$
0 1	E	$\#$	Akzeptieren

Beispiel: Übergangstabelle eines $\mathrm{LR}(0)$-Automaten

	a	b	c	#	A	B	S
0	4	-	-	-	2	3	1
1	-	-	-	#	-	-	-
2	4	-	5	-	-	6	-
3	+4	+4	+4	-	+4	+4	+4
4	4	8	-	-	7	3	-
5	+2	+2	+2	-	+2	+2	+2
6	+3	+3	+3	-	+3	+3	+3
7	4	9	-	-	-	6	-
8	+6	+6	+6	-	+6	+6	+6
9	+5	+5	+5	-	+5	+5	+5

-	Fehler
+r	Reduziere mit Regel r
q	Schifte, neuer Zustand: q
#	HALT

1: $Z \rightarrow S$
2: $S \rightarrow Ac$
3: $A \rightarrow AB$
4: $A \rightarrow B$
5: $B \rightarrow aAb$
6: $B \rightarrow ab$



Beispiel: Ablauf eines $\mathrm{LR}(0)$-Automaten

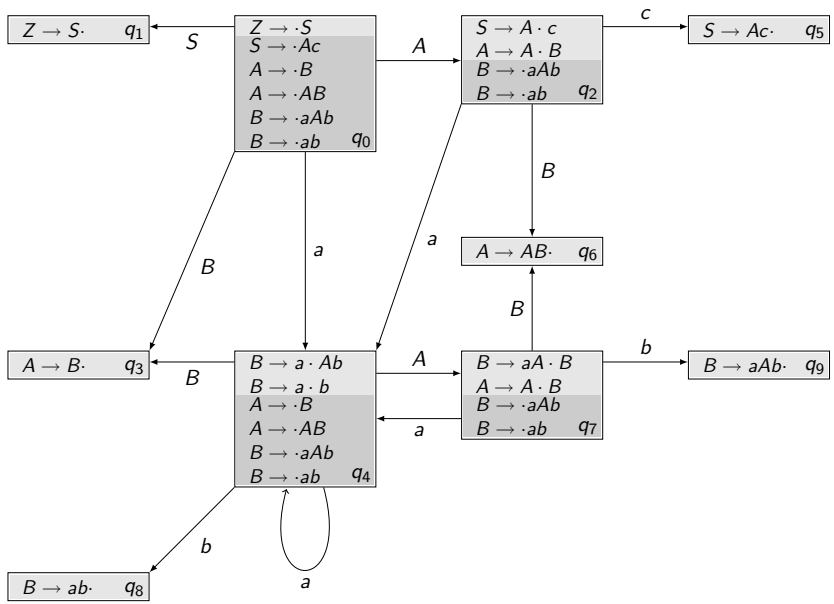
Keller	Eingabe	Bemerkungen
0	aabbc#	schifte(a)
04	abbc#	schifte(a)
044	bbc#	schifte(b)
0448	bc#	reduziere($B \to ab$)
04	bc#	schifte(B)
043	bc#	reduziere($A \to B$)
04	bc#	schifte(A)
047	bc#	schifte(b)
0479	c#	reduziere($B \to aAb$)
0	c#	schifte(B)
03	c#	reduziere($A \to B$)
0	c#	schifte(A)
02	c#	schifte(c)
025	#	reduziere($S \to Ac$)
0	#	schifte(S)
01	#	HALT

1: $Z \to S$ 2: $S \to Ac$

3: $A \to AB$ 4: $A \to B$

6: $B \to ab$

5: $B \to aAb$



rot: aktuelle Symbole bzw. zu reduzierender Kellerteil

blau: Reduktionssymbol mit Schift im nächsten Schritt

Situationen (Items)

Ziel: bei Prüfung der Anwendbarkeit von Regeln sqx $\rightarrow$ s'q'x' Kellerinhalt und Zustand sq mit einem Zustandssymbol codieren (Prüfung mehrerer Einträge im Keller vermeiden)

Lösungsidee:

- bei LL und LR ist s rechte bzw. linke Seite einer Produktion $X \rightarrow x_1 \dots x_n$
- Übergänge $tqt \rightarrow q$ (bei LL) bzw. $qt \rightarrow tq$ (bei LR) sind nur zulässig, wenn in der Produktion ein Terminalzeichen t ansteht, $x_1 \dots x_n = x' t x'''$, wobei $x'' := tx'''$
- also: ersetze sqx durch Situation $[X \rightarrow x' \cdot x''; x]$, die durch den Punkt anzeigt, wie weit die Produktion abgearbeitet ist.
- Situationen $[X \rightarrow \cdot\ x''; x]$ oder $[X \rightarrow x' \cdot; x]$ sind erlaubt und notwendig.
- Verwende Situationen als Zustände und als Kellersymbole.
- Situationen heißen englisch *items.*

Berechnung von CLOSURE

```
set<item> closure(item I)
{
    J = {I};
    do {
        changed = false;
        foreach((A → α · Bβ) ∈ J) {
            foreach((B → γ) ∈ G) {
                if((B → ·γ) ∉ J) {
                    J = J ∪ {B → ·γ};
                    changed = true;
                }
            }
        }
    } while (changed);

    return J;
}
```

Beispiel: closure

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

Berechnung von $\mathit{closure}(E' \rightarrow \cdot E)$ ergibt:

$$
\begin{aligned}
E' &\rightarrow \cdot E \\
E &\rightarrow \cdot E + T \\
E &\rightarrow \cdot T \\
T &\rightarrow \cdot T * F \\
T &\rightarrow \cdot F \\
F &\rightarrow \cdot(\ E \ ) \\
F &\rightarrow \cdot \mathbf{id}
\end{aligned}
$$

GOTO

$$\mathrm{GOTO}(I, X) = \mathit{closure}(\{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in I\})$$

Beispiel: Ist $I = \{[E' \to E\cdot], [E \to E \cdot + T]\}$, dann enthält $\mathrm{GOTO}(I, +)$ folgende Situationen:

$$\begin{aligned} E &\to E + \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot(E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

Berechnung der kanonischen $\mathrm{LR}(0)$-Situationsmengen

```
void items(grammar G) {
    C = { closure( {[S' → ·S]} ) };
    do {
        changed = false;
        foreach(set<item> I ∈ C) {
            foreach(grammar_symbol X) {
                if (GOTO(I, X) ≠ ∅ && GOTO(I, X) ∉ C) {
                    C = C ∪ { closure(GOTO(I, X)) };
                    changed = true;
                }
            }
        }
    } while(changed);
}
```

LR(1)-Parseralgorithmus

```
push(0);
a = next_token();
while (true) {
    if (ACTION[top(), a] == shift t) {
        push(t);
        a = next_token();
    } else if(ACTION[top(), a] == reduce A → β) {
        for (i = 0; i < |β|; ++i)
            pop();
        push(GOTO[top(), A]);
    } else if(ACTION[top(), a] == accept) {
        break;
    } else {
        report_error(); break;
    }
}
```

$\mathrm{SLR}(k)$-Grammatiken

$\mathrm{SLR}(k)$-Grammatik (simple LR(k)):
Eine Grammatik heißt $\mathrm{SLR}(k)$, wenn sie $\mathrm{LR}(0)$ ist oder die $\mathrm{SLR}(k)$-Übergangsfunktion bezüglich des charakteristischen $\mathrm{LR}(0)$-Automaten C, $q \in Q$, $t \in V$ und $v \in T^{\leq k-1}$:

$$f(q, tv) = \begin{cases} \text{SHIFT} & \text{wenn} & [X \to \mu \cdot t\gamma] \in q \wedge \\ & & (v \in \mathrm{Anf}'_{k-1}(\gamma \mathrm{Folge}_{k-1}(X)) \vee k < 2) \\ \text{RED}(X \to x) & \text{wenn} & [X \to x\cdot] \in q \wedge \\ & & k : tv \in \mathrm{Folge}_k(X) \\ \text{HALT} & \text{wenn} & t = \# \wedge [Z \to S\cdot] \in q \\ \text{FEHLER} & \text{sonst} & \end{cases}$$

eindeutig ist.
rot: Unterschied $\mathrm{LR}(k)$ und $\mathrm{SLR}(k)$

SLR(1)-Übergangsfunktion am Beispiel

$$
\begin{aligned}
E' &\rightarrow E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \textbf{id}
\end{aligned}
$$

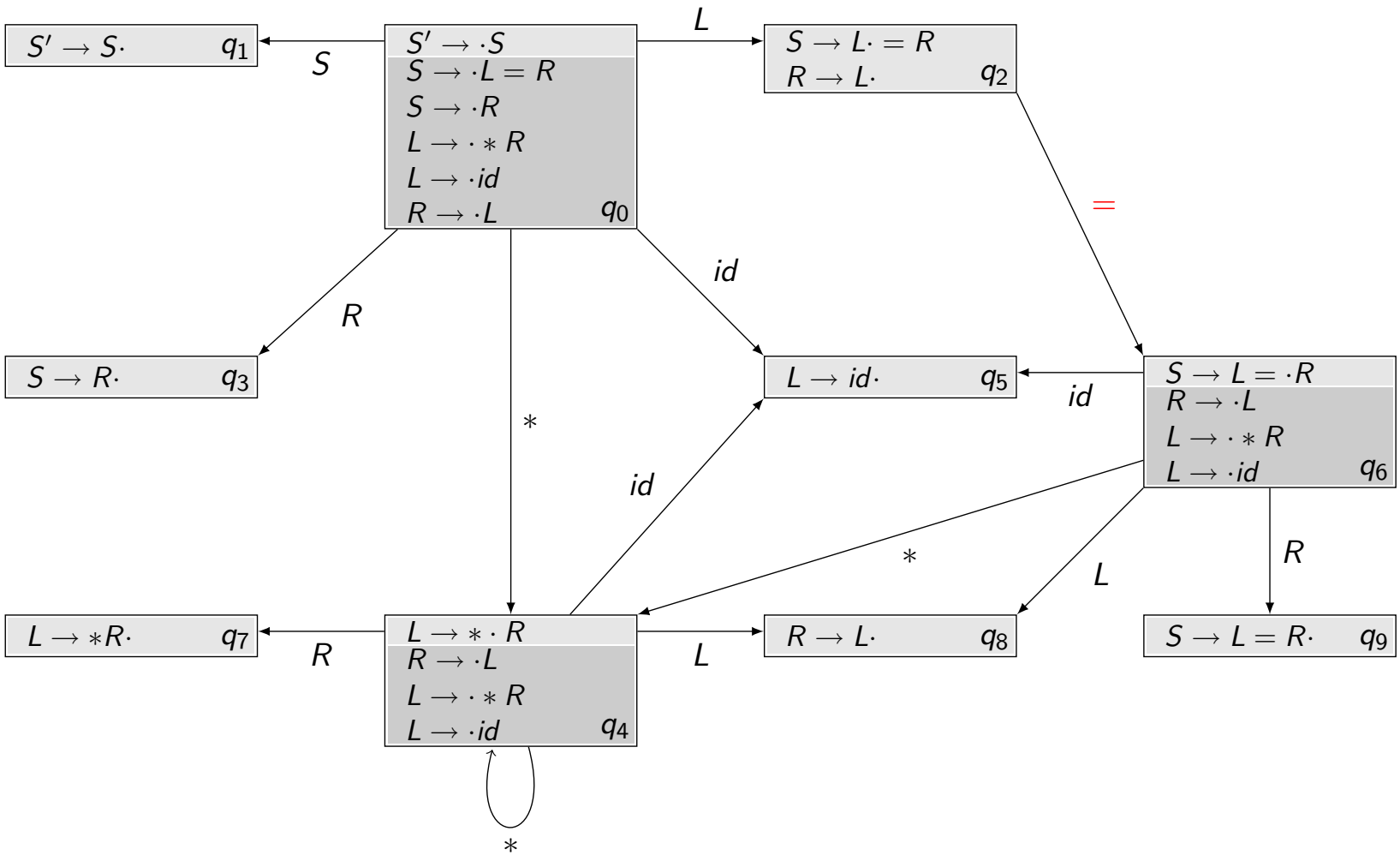
$$
\begin{aligned}
I_1 : E' &\rightarrow E \cdot \\
E &\rightarrow E \cdot + T \mid T
\end{aligned}
$$

Es gilt $\mathrm{Folge}_1(E') = \{\#\}$. SLR(1)-Übergangsfunktion für I_1 ergibt:

$$
\begin{aligned}
f(I_1, +) &= \text{SHIFT} \\
f(I_1, \#) &= \text{HALT}
\end{aligned}
$$

Beispiel für Konflikt bei SLR(1)

$S \to L = R \mid R$ $\quad$ $L \to * R \mid \mathbf{id}$ $\quad$ $R \to L$



$\text{Folge}_1(S) = \{\#\}$ $\quad$ $\text{Folge}_1(L) = \{=, \#\}$ $\quad$ $\text{Folge}_1(R) = \{=, \#\}$

Motivation für LR/LALR (1/4)

LR-Parser erfassen alle deterministischen kf Grammatiken. Sie sind noch einigermaßen schnell berechenbar und es gibt ausgereifte Generatoren. Für den yacc-Generator produziert die Grammatik

```
%token IF ELSE THEN IDENT
%%
Statement: IfStatement | Expr ;
Expr: IDENT;
IfStatement: IF Expr THEN Statement |
                IF Expr THEN Statement ELSE Statement ;
```

allerdings folgende Fehlermeldung

state 7 contains 1 shift /reduce conflict .

Motivation für LR/LALR (2/4)

Grammatik:

```
%token IF ELSE THEN IDENT
%%
Statement: IfStatement | Expr ;
Expr: IDENT;
IfStatement: IF Expr THEN Statement |
                IF Expr THEN Statement ELSE Statement ;
```

Fehlermeldung:

```
state 7 contains 1 shift/reduce conflict.
...
state 7
IfStatement -> IF Expr THEN Statement . (rule 4)
IfStatement ->
  IF Expr THEN Statement . ELSE Statement (rule 5)
ELSE shift, and go to state 8
ELSE [reduce using rule 4 (IfStatement)]
$default reduce using rule 4 (IfStatement)
```

Motivation für LR/LALR (3/4)

Frage: Wie kann man die vorige Fehlermeldung verstehen?

- Zweck aller nachfolgenden Ausführungen ist nur darauf zu beziehen.
- Detailliertes Wissen über automatische Konstruktion vernachlässigbar
- Aber: Intuitive „Hand“konstruktion muss beherrscht werden.

Motivation für LR/LALR (4/4)

Lösungen:

- Sprache ändern: abschließendes „end“ einführen (zulässig?)
- Ausfaktorisieren:

```
%token IF ELSE THEN IDENT
%%
Statement: IfStatement | Expr ;
Expr: IDENT;
IfStatement:
  IF Expr THEN Statement |
  IF Expr THEN IfThenElseStat ELSE Statement ;
IfThenElseStat:
  IF Expr THEN IfThenElseStat ELSE IfThenElseStat |
  Expr ;
```

LR-Grammatiken

Ziel:

- alle deterministisch parsbaren kfG charakterisieren ($\mathrm{LL}(k)$ ist stark eingeschränkt)
- Rechtsableitung konstruieren

Endgültige Definition von D.E. Knuth 1966:
Eine kf-Grammatik heißt eine $\mathrm{LR}(k)$-Grammatik, wenn für alle Paare von Ableitungen

$$\begin{aligned} Z &\Rightarrow_R^* \mu A\omega \Rightarrow \mu\chi\omega & \mu &\in V^*, \omega \in T^*, A \to \chi \in P \\ Z &\Rightarrow_R^* \mu' B\omega' \Rightarrow \mu'\gamma\omega' & \mu' &\in V^*, \omega' \in T^*, B \to \gamma \in P \end{aligned}$$

gilt:

$$(|\mu\chi| + k) : \mu\chi\omega = (|\mu\chi| + k) : \mu'\gamma\omega') \Rightarrow \mu = \mu', A = B, \chi = \gamma$$

Idee für deterministisch Machen des LR-Parsers

- Nimm alle verfügbare Information (Reduktionsklassen):
  - unendliche Vorschau auf die noch zu verarbeitende Eingabe
  - den gesamten Kellerinhalt
- Allerdings sind die Reduktionsklassen nicht effektiv berechenbar

  Deshalb: Beschränkung auf k-Zeichen Vorschau und Erkenntnis, dass der zur jeweiligen Entscheidung nötige Kellerinhalt in einem bzw. beschränkt vielen Zuständen subsumierbar ist.

  Dies führt zu *k*-Kellerklassen
- Mit Hilfe der *k*-Kellerklassen wird der charakteristische Automat definiert
- Aus dem charakteristischen Automaten ist die Übergangsfunktion des $\mathrm{LR}(k)$-Parsers ablesbar
- Aber: Deterministisch machen geht nur für eingeschränkte Grammatikklassen

Textmengen

$k : x$

$$k : x = x\# \quad \text{falls } x = x_1 \dots x_m \wedge m < k$$
$$k : x = x_1 \dots x_k \quad \text{falls } x = x_1 \dots x_m \wedge m \geq k$$

$\mathrm{Anf}_k(x) = \{u \mid \exists y \in T^* : x \Rightarrow^* y \wedge u = k : y\}$
in der Literatur auch $\mathrm{First}_k(x)$ genannt

$A \Rightarrow_{R'} \alpha$ gdw. $A \Rightarrow_R \alpha \wedge \nexists B \in N : A \Rightarrow_R B\alpha \Rightarrow \alpha$

$\mathrm{Anf}'_k(x) = \{u \in \mathrm{Anf}_k(x) \mid \exists y \in T^* : x \Rightarrow_{R'} uy\}$
in der Literatur auch $\mathrm{EFF}_k(x)$ genannt (EFfective First)

$\mathrm{Folge}_k(x) = \{u \mid \exists m, y \in V^* : Z \Rightarrow^* mxy \wedge u \in \mathrm{Anf}_k(y)\}$
in der Literatur auch $\mathrm{Follow}_k(x)$ genannt

Reduktionsklassen

Unter welchen Bedingungen soll der LR-Automat schiften / reduzieren?
maximal verfügbare Information:
Reduktionsklasse $= \{(\text{Kellerinhalt}, \text{Eingaberest})\}$

R_0 für Schift und R_p für Produktionen $A_p \rightarrow y_p$, $p = 1, \ldots, n$:

$$\begin{aligned} R_0 &= \{(r'r, ss') \mid Z \Rightarrow_R^* r'As' \wedge A \Rightarrow_{R'} rs \wedge s \neq \varepsilon\} \\ R_p &= \{(r'y_p, s) \mid Z \Rightarrow_R^* r'A_p s \wedge A_p \rightarrow y_p \in P\} \end{aligned}$$

$R_i \cap R_j = \emptyset$ für $i \neq j$ bedeutet: Schiften bzw. Reduzieren mit Produktion p kann stets eindeutig entschieden werden. Jeder Satz besitzt eindeutige Rechtsableitung und eindeutigen Strukturbaum.
Grammatik ist eindeutig.
Leider: Eindeutigkeit von kf Grammatiken nicht entscheidbar, d.h. $R_i \cap R_j = \emptyset$ nicht algorithmisch entscheidbar.
Problem ist der unbeschränkt lange Eingaberest.

Warum ist $\Rightarrow_{R'}$ nötig: Beispiel

$L(G) = \{cb, b\}$ und
$P = \{1\colon\ Z \rightarrow S,\ 2\colon\ S \rightarrow Ab,\ 3\ :\ A \rightarrow c,\ 4\colon\ A \rightarrow \varepsilon\ \}$
Es gilt $b \in \mathrm{Anf}_1(S)$, aber $b \notin \mathrm{Anf}'_1(S)$.

$R_0 = \{(\varepsilon, cb\#), (A, b\#), \cancel{(\varepsilon, b\#)}\},$
$R_1 = \{(S, \#)\},$
$R_2 = \{(Ab, \#)\},$
$R_3 = \{(c, b\#)\},$
$R_4 = \{(\varepsilon, b\#)\}$

Beispiel: Rechtsableitung $Z \Rightarrow_R S \Rightarrow_R Ab \Rightarrow_R b$:
ε auf A reduzieren, bevor das Zeichen b geschiftet wird.
Ohne Unterscheidung zwischen $\Rightarrow_{R'}$ und $\Rightarrow_R$: $(\varepsilon, b\#)$ gehört zu R_0, also wird geschiftet aber Konfiguration $bq\#$ kann keiner Reduktionsklasse zugeordnet werden, d.h. Sackgasse

k-Kellerklassen

Idee: Beschränke den betrachteten Eingaberest auf $k \geq 0$ Zeichen:
k-Kellerklasse K_p^k, $p = 0, \ldots, n$ definiert durch

$$K_p^k = \{(r, k : s) \mid \exists (r, s) \in R_p\}$$

Vergleich mit LR-Definition: Mit

$$Z \Rightarrow_R^* \mu A \omega \Rightarrow \mu \chi \omega \qquad \mu \in V^*, \omega \in T^*, A \to \chi \in P$$
$$Z \Rightarrow_R^* \mu' B \omega' \Rightarrow \mu' \gamma \omega' \qquad \mu' \in V^*, \omega' \in T^*, B \to \gamma \in P$$

gilt:

$$(|\mu\chi| + k) : \mu\chi\omega = (|\mu\chi| + k) : \mu'\gamma\omega') \Rightarrow \mu = \mu', A = B, \chi = \gamma$$

Satz: Eine Grammatik ist genau dann $\mathrm{LR}(k)$, wenn die k-Kellerklassen paarweise disjunkt sind. Eine Grammatik G ist genau dann deterministisch parsbar, wenn es ein $k \geq 0$ gibt, so dass G $\mathrm{LR}(k)$ ist.

Reguläre Erkenner für k-Kellerklassen

Satz(Büchi): Alle Kellerklassen K_p^k für $p = 0, \ldots, n$ sind regulär.

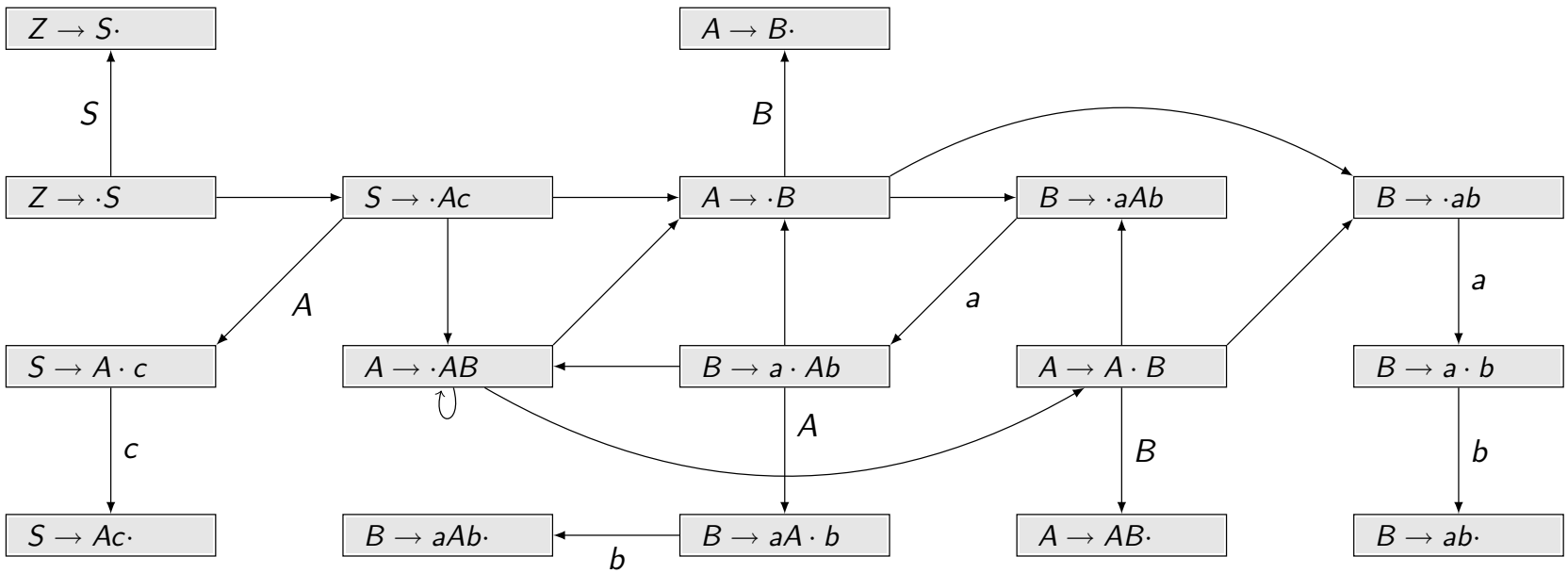
Nachweis durch Angabe einer rechtslinearen Grammatik G_p^k für jede Kellerklasse.

Man kann daraus einen nichtdeterministischen endlichen Automaten C ableiten, der die regulär erkennbaren Wortanfänge der durch die (original) Grammatik G gegebenen Sprache akzeptiert. Dieser hat:

- Situationen als Zustände $\in Q$
- $[Z \rightarrow \cdot\ S; \#]$ als Startzustand (Z nichtrekursives Start-NT von G)
- die Übergangsfunktion f
  - $f([X \rightarrow x \cdot vy; \omega], v) = [X \rightarrow xv \cdot y; \omega]$ mit $v \in V$
  - $f([X \rightarrow x \cdot By; \omega], \varepsilon) = [B \rightarrow \cdot\ b; \tau]$ mit $B \rightarrow b \in P$, $\tau \in \mathrm{Anf}_k(\gamma\omega)$
- Endzustände sind belanglos für das weitere Vorgehen

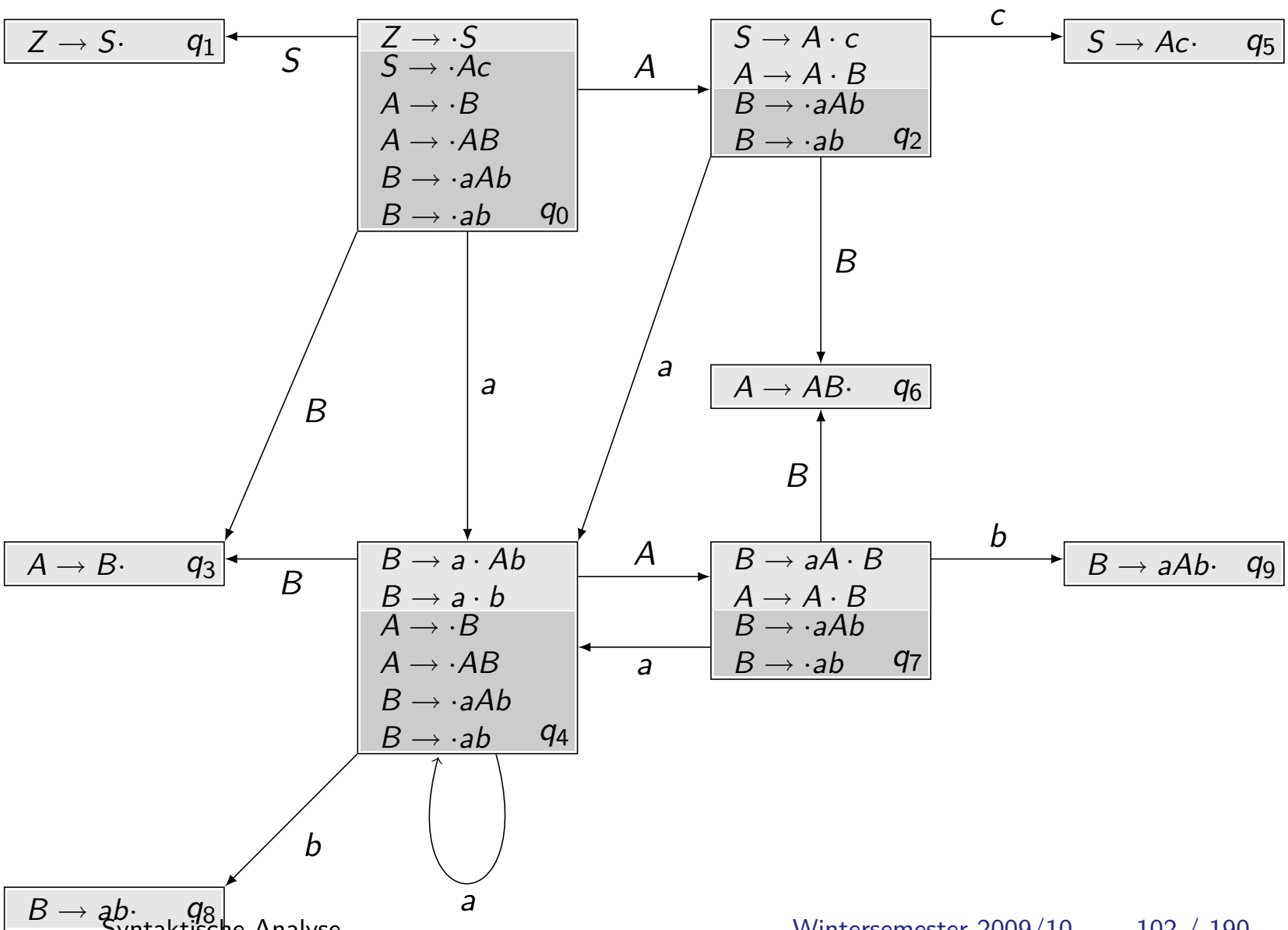
Nichtdeterministischer, endlicher Automat C

$$Z \to S,\ S \to Ac,\ A \to AB,\ A \to B,\ B \to aAb,\ B \to ab$$



Charakteristischer Automat für $k = 0$

$$Z \to S,\ S \to Ac,\ A \to AB,\ A \to B,\ B \to aAb,\ B \to ab$$



Hilfskonstruktionen mit dem charakteristischen Automaten

Der nichtdeterministische endliche Automat C kann mit den bekannten Verfahren deterministisch gemacht werden. Ergebnis: charakteristischer Automat

- Definiere basis(q, t, ω) als Menge von Situationen die mit Einlesen des (Nicht-)Terminals t aus Situationen von q ausgehend erreichbar sind:

$$\text{basis}(q, t, \omega) = \{[X \to \mu t \cdot \gamma; \omega] \mid [X \to \mu \cdot t\gamma; \omega] \in q\}$$

- Definiere next(q, t, ω) als die transitive Hülle von basis(q, t, ω)

$$\text{next}(q, t, \omega) = H(\text{basis}(q, t, \omega)), \text{ wenn } \text{basis}(q, t, \omega) \neq \emptyset$$

mit $H(M) = M \cup \{[B \to \cdot\beta; \tau] \mid \exists [X \to \mu \cdot B\gamma; \xi] \in H(M) : B \to \beta \in P \wedge \tau \in \text{Anf}_k(\gamma\xi)\}$

Hinweis: $H(M)$ erweitert Situationen (oder Situtationsmengen) auf die Zustände des charakteristischen Automaten.

Übergangsfunktion des $\mathrm{LR}(k)$-Automaten

Für einen charakteristischen Automaten C, $q \in Q$, $t \in V$ und $v \in T^{\leq k-1}$:

$$f(q, tv) = \begin{cases} \text{next}(q, t, \omega) & \text{wenn} & [X \to \mu \cdot t\gamma; \omega] \in q \wedge \\ & & (v \in \mathrm{Anf}'_{k-1}(\gamma\omega) \vee k < 2) \\ \text{RED}(X \to x) & \text{wenn} & [X \to x\cdot; k : tv] \in q \\ \text{HALT} & \text{wenn} & t = \# \wedge [Z \to S\cdot; \#] \in q \\ \text{FEHLER} & \text{sonst} & \end{cases}$$

- Zustand q heißt inadäquat, wenn Übergangsfunktion $f(q, tv)$ für irgendein tv nicht eindeutig die Fälle eins, zwei und drei unterscheiden kann. Nur zwei Möglichkeiten existieren:
  - Schift-Reduktionskonflikt: schiften und reduzieren möglich
  - Reduktions-Reduktionskonflikt: Reduktion mit zwei verschiedenen Produktionen möglich
- Eine kfG G ist genau dann $\mathrm{LR}(k)$-Grammatik, wenn der charakteristische Automat keine inadäquaten Zustände besitzt.

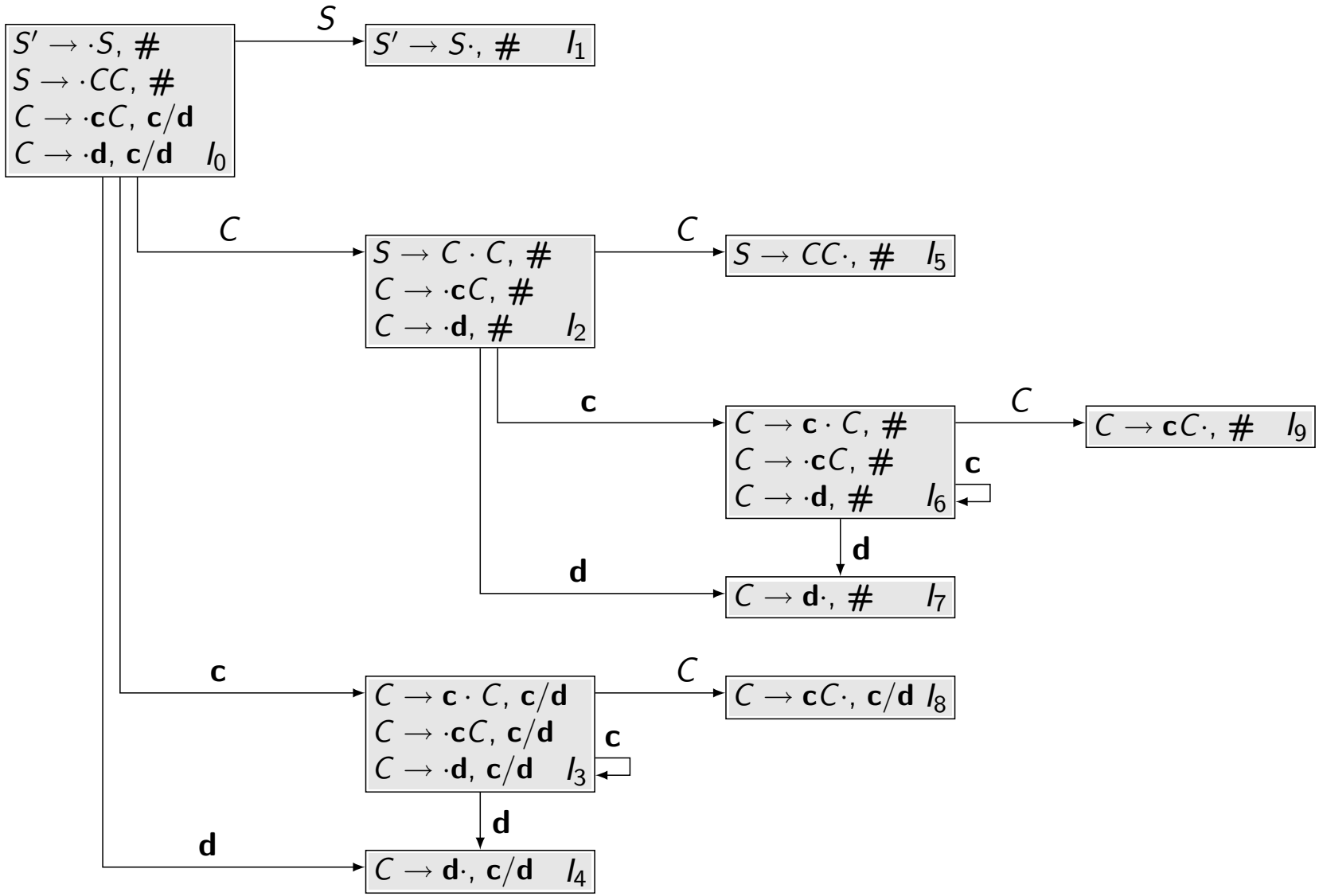
Ablauf des $\mathrm{LR}(k)$-Automaten

- Startzustand des Automaten ist: $H(\{[Z \to \cdot S; \#]\})$
- In jedem Schritt wird die Übergangsfunktion f angewandt:
  - Beim Schiften wird statt des konkreten Symbols ein Zustand des charakteristischen Automaten auf den Keller gelegt. Nur durch Schiften von Terminalen wird die Eingabe verkürzt; Schiften von Nichtterminalen läßt die Eingabe unverändert.
  - RED$(X \to x)$ bedeutet Reduzieren mit Produktion $X \to x$, d.h.:
    - Löschen von $|x|$ Kellereinträgen.
    - Oberster Kellereintrag wird aktueller Zustand.
    - Die Übergangsfunktion $f(q, Xtv)$ wird angewandt. Dies führt sofort zu einem Schift des Nichtterminals X und kann auch in einem Schritt mit dem Kellerlöschen behandelt werden.
  - Bei HALT wird die Eingabe akzeptiert.
  - Bei FEHLER ist die Eingabe nicht akzeptabel.

Beispiel: Grammatik G'

$$
\begin{aligned}
S' &\rightarrow S \\
S &\rightarrow C\ C \\
C &\rightarrow \mathbf{c}\ C \mid \mathbf{d}
\end{aligned}
$$

GOTO-Graph für die Grammatik G'



Beispiel: Ein $\mathrm{LR}(2)$-Kellerautomat

1: $Z \rightarrow X$, 2: $X \rightarrow Y$, 3: $X \rightarrow bYa$, 4: $Y \rightarrow c$, 5: $Y \rightarrow ca$

Keller	Eingabe	Bemerkungen
0	bcaa#	schifte(b)
03	caa#	schifte(c)
031	aa#	schifte(a)
0312	a#	reduziere($Y \rightarrow ca$)
03	a#	schifte(Y)
035	a#	schifte(a)
0358	#	reduziere($X \rightarrow bYa$)
0	#	schifte(X)
01	#	HALT

	bc	c#	ca	a#	aa	#	X#	Y#	Ya
0	3	7	7	-	-	-	9	8	-
1	-	-	-	+4	2	-	-	-	-
2	-	-	-	+5	-	-	-	-	-
3	-	-	1	-	-	-	-	-	4
4	-	-	-	5	-	-	-	-	-
5	-	-	-	-	-	+3	-	-	-
6	-	-	-	-	-	+5	-	-	-
7	-	-	-	6	-	+4	-	-	-
8	-	-	-	-	-	+2	-	-	-
9	-	-	-	-	-	#	-	-	-

-	Fehler
+r	Reduziere mit Regel r
q	Schifte, neuer Zustand: q
#	HALT

rot: aktuelle Symbole bzw. zu reduzierender Kellerteil

grün: Vorschau

blau: Reduktionssymbol mit Schift im nächsten Schritt

LR-Grammatiken: Probleme

Probleme:

- Wegen Unterscheidung verschiedener Rechtskontexte hat der $\mathrm{LR}(k)$-Automat bereits für $k = 1$ sehr viele Zustände (verglichen mit $\mathrm{LR}(0)$)
  - Beispiel: für Ausdrucksgrammatik mit „+“ 17 statt 9 für $k = 0$.
- $\mathrm{LR}(1)$-Automat nur automatisch generierbar
- sehr große Tabellen
- Test der Eigenschaft nur durch Konstruktion möglich

In der Praxis Beschränkung auf Unterklassen von LR(1) mit LR(0)-Zustandsmenge.

$\mathrm{SLR}(k)$-Grammatiken (1/2)

$\mathrm{SLR}(k)$-Grammatik (simple LR(k)):
Eine Grammatik heißt $\mathrm{SLR}(k)$, wenn sie $\mathrm{LR}(0)$ ist oder die modifizierte Übergangsfunktion bezüglich des charakteristischen $\mathrm{LR}(0)$-Automaten C, $q \in Q$, $t \in V$ und $v \in T^{\leq k-1}$:

$$f(q, tv) = \begin{cases} \text{next}(q, t, \omega) & \text{wenn} & [X \to \mu \cdot t\gamma] \in q \wedge \\ & & (v \in \mathrm{Anf}'_{k-1}(\gamma \mathrm{Folge}_{k-1}(X)) \vee k < 2) \\ \text{RED}(X \to x) & \text{wenn} & [X \to x\cdot] \in q \wedge \\ & & k : tv \in \mathrm{Folge}_k(X) \\ \text{HALT} & \text{wenn} & t = \# \wedge [Z \to S\cdot] \in q \\ \text{FEHLER} & \text{sonst} & \end{cases}$$

keine inadäquaten Zustände liefert.
rot: Unterschied $\mathrm{LR}(k)$ und $\mathrm{SLR}(k)$

$\mathrm{SLR}(k)$-Grammatiken (2/2)

- Achtung: next(q, t, ε) operiert auf einem charakteristischen $\mathrm{LR}(0)$-Automaten, also ohne Rechtskontexte.
- Problem: $\mathrm{SLR}(k)$ genügt zwar für Ausdrucksgrammatiken, berücksichtigt aber Linkskontext zu wenig (da die Vorausschau ohne Kenntnis der schon verarbeiteten mithin links stehenden Zeichen bestimmt wird).

$\mathrm{LALR}(k)$-Grammatiken

Sei $\mathrm{kern}(q) = \{[X \to \mu \cdot \gamma] \mid [X \to \mu \cdot \gamma; \Omega_i] \in q\}$.

Eine Grammatik heißt $\mathrm{LALR}(k)$ (look ahead $\mathrm{LR}(k)$), wenn es keine inadäquaten Zustände gibt, falls man im $\mathrm{LR}(k)$-Automaten alle Zustände q, q' mit $\mathrm{kern}(q) = \mathrm{kern}(q')$ zusammenlegt.

- Satz: Jeder $\mathrm{SLR}(k)$- oder $\mathrm{LALR}(k)$-Automat hat die gleiche Anzahl von Zuständen wie der $\mathrm{LR}(0)$-Automat zur gleichen Grammatik.
- Der Unterschied der parsbaren Sprachen zwischen $\mathrm{LALR}(k)$ und $\mathrm{LR}(k)$ ist praktisch unerheblich.
- Alle verbreiteten LR-Parsergeneratoren (yacc, pgs, lalr, bison, ...) konstruieren $\mathrm{LALR}(1)$-Automaten.

$\mathrm{LALR}(k)$-Übergangsfunktion

Für einen charakteristischen Automaten C' in dem alle Zustände mit gleichem Kern zusammengelegt sind, $q \in Q$, $t \in V$ und $v \in T^{\leq k-1}$:

$$f(q,tv) = \begin{cases} \text{next}(q,t,\omega) & \text{wenn} & [X \to \mu \cdot t\gamma;\omega] \in q \wedge \\ & & (v \in \mathrm{Anf}'_{k-1}(\gamma\omega) \vee k < 2) \\ \text{RED}(X \to x) & \text{wenn} & [X \to x\cdot; k : tv] \in q \\ \text{HALT} & \text{wenn} & t = \# \wedge [Z \to S\cdot;\#] \in q \\ \text{FEHLER} & \text{sonst} & \end{cases}$$

erhalten wir die LALR(k)-Übergangsfunktion.

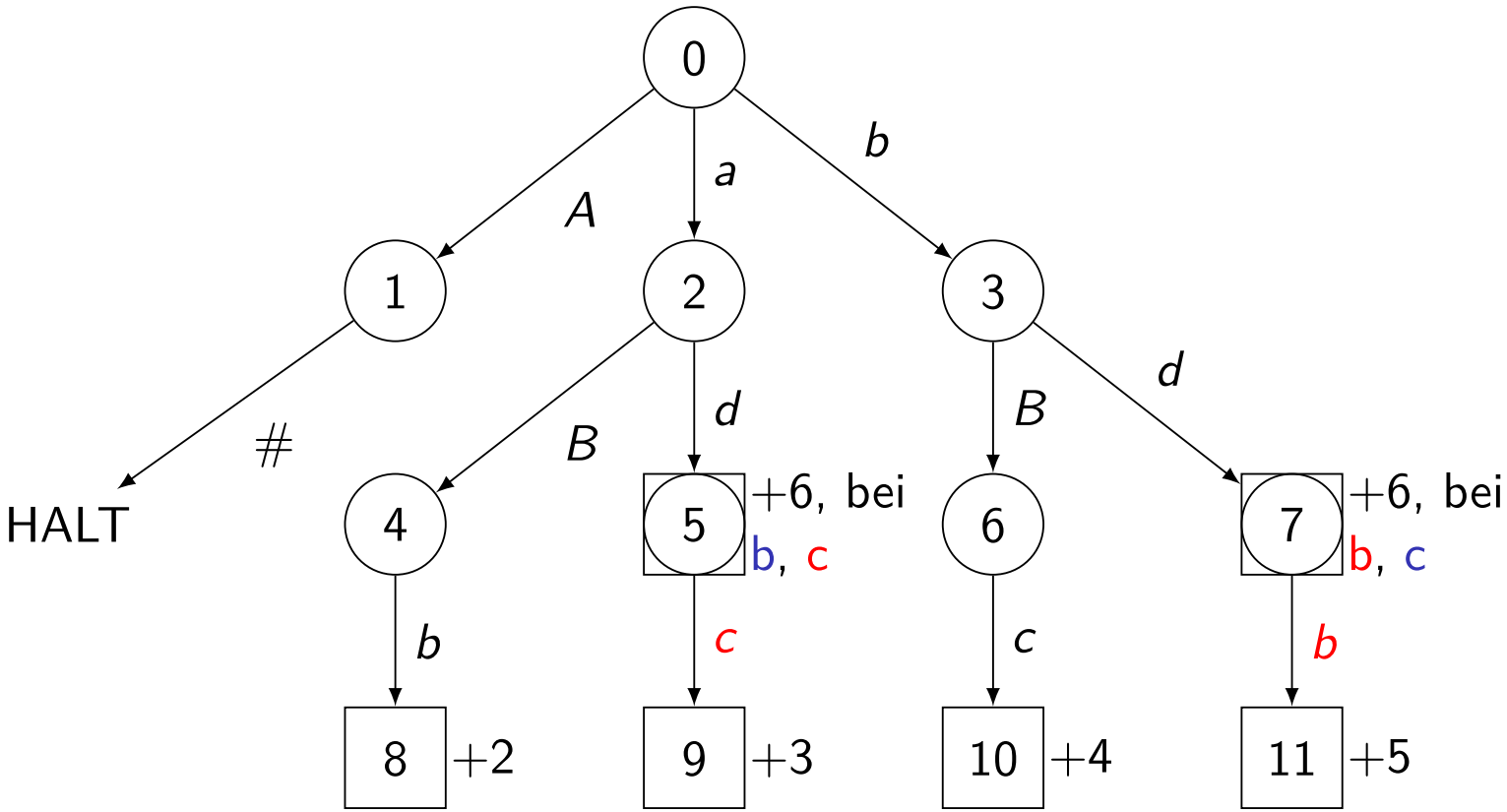
- rot: Unterschied $\mathrm{LR}(k)$ und $\mathrm{LALR}(k)$
  Einziger Unterschied: Zusammenlegen der Zustände „modulo Kern“
- Im Unterschied zu $\mathrm{SLR}(k)$ benutzt $\mathrm{LALR}(k)$ einen „echten“ Rechtskontext von $X \to x$, der schärfer trennt als $\mathrm{Folge}_k(X)$.

nicht SLR, aber LALR

1: $Z \rightarrow A$ 2: $A \rightarrow aBb$ 3: $A \rightarrow adc$

4: $A \rightarrow bBc$ 5: $A \rightarrow bdb$ 6: $B \rightarrow d$

$\text{Folge}_1(B) = \{b, c\}$



praktisches Beispiel (Algol 60): Unterscheide ; id : und [id :

Behebung inadäquater Zustände

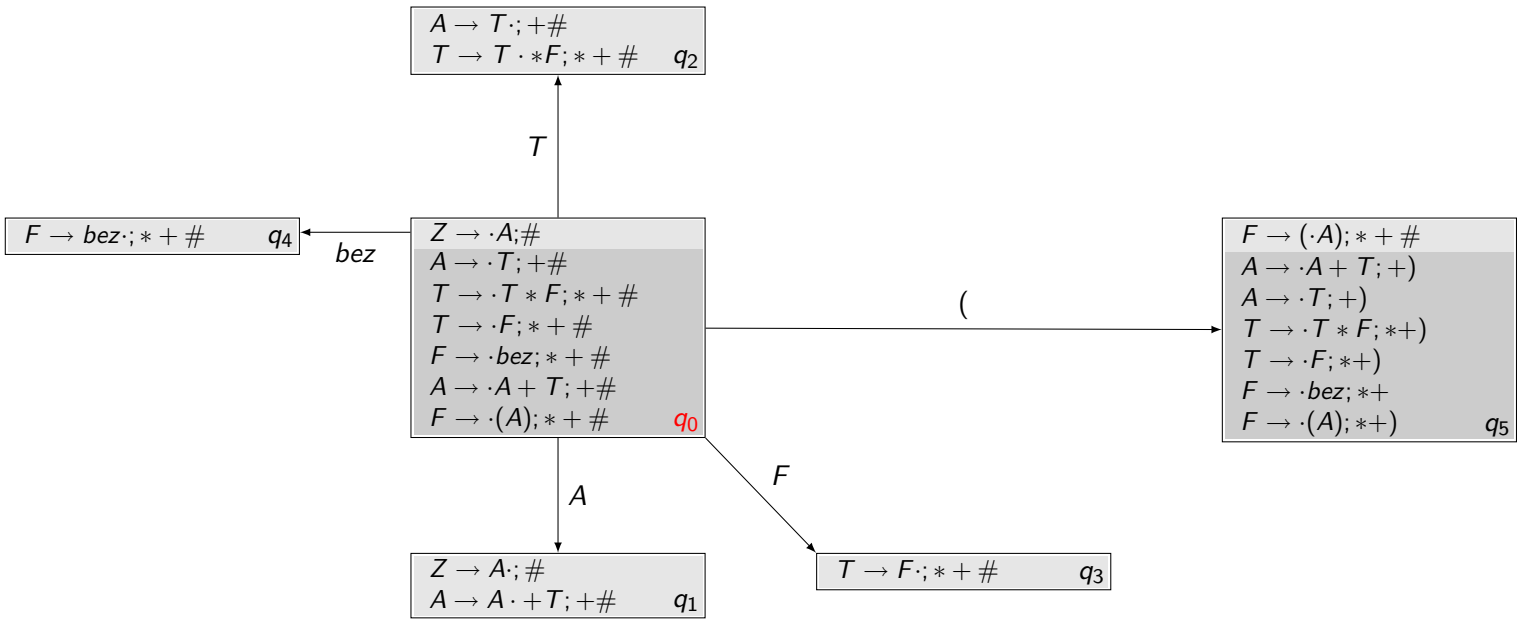
- Parsergenerator:
  - Schift-Reduzier-Konflikt: Schiften wird bevorzugt
  - Reduzier-Reduzier-Konflikt: Zuerst spezifizierte Reduktion wird häufig bevorzugt. Meist fehlerbehaftet.
- Maßnahmen:
  - Automatische Konfliktauflösung auf Korrektheit prüfen!
  - „Faktorisieren“ gemeinsamer Produktionsteile (siehe if-then-else)
  - Verwendung von Präzedenzen (z.B. bei bison)
  - Vergrößern der erkannten Sprache, nachher Einschränken mit semantischer Analyse
  - An Beispielen lernen
  - Mächtigeren Generator (höheres k, LR statt LALR) verwenden, allerdings in der Regel nicht hilfreich

Achtung: Es gibt keine allgemeingültigen Verfahren

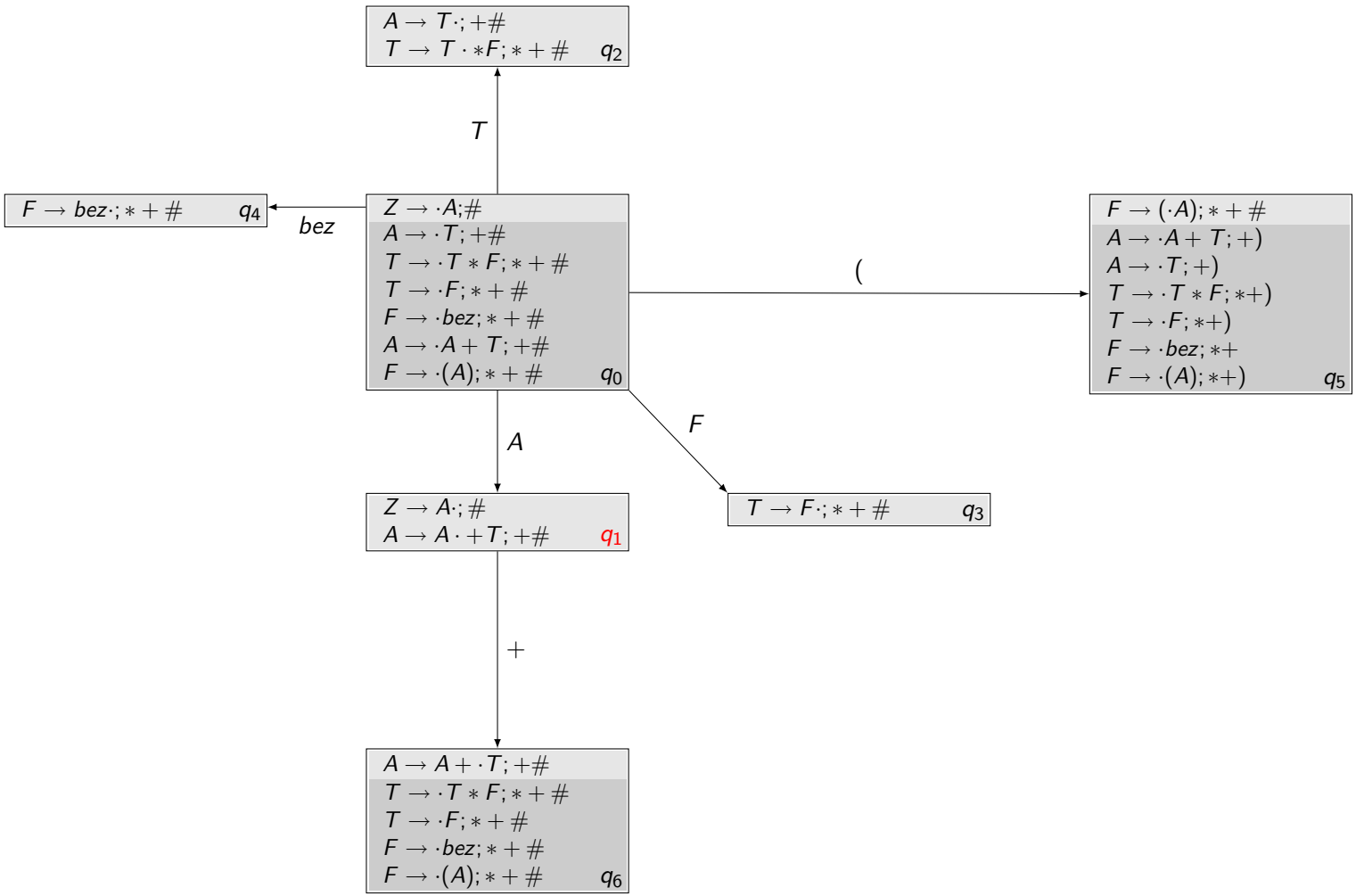
LALR(1)-Konstruktion am Beispiel

1: $Z \rightarrow A$
2: $A \rightarrow A + T$
3: $A \rightarrow T$
4: $T \rightarrow T * F$
5: $T \rightarrow F$
6: $F \rightarrow bez$
7: $F \rightarrow (A)$

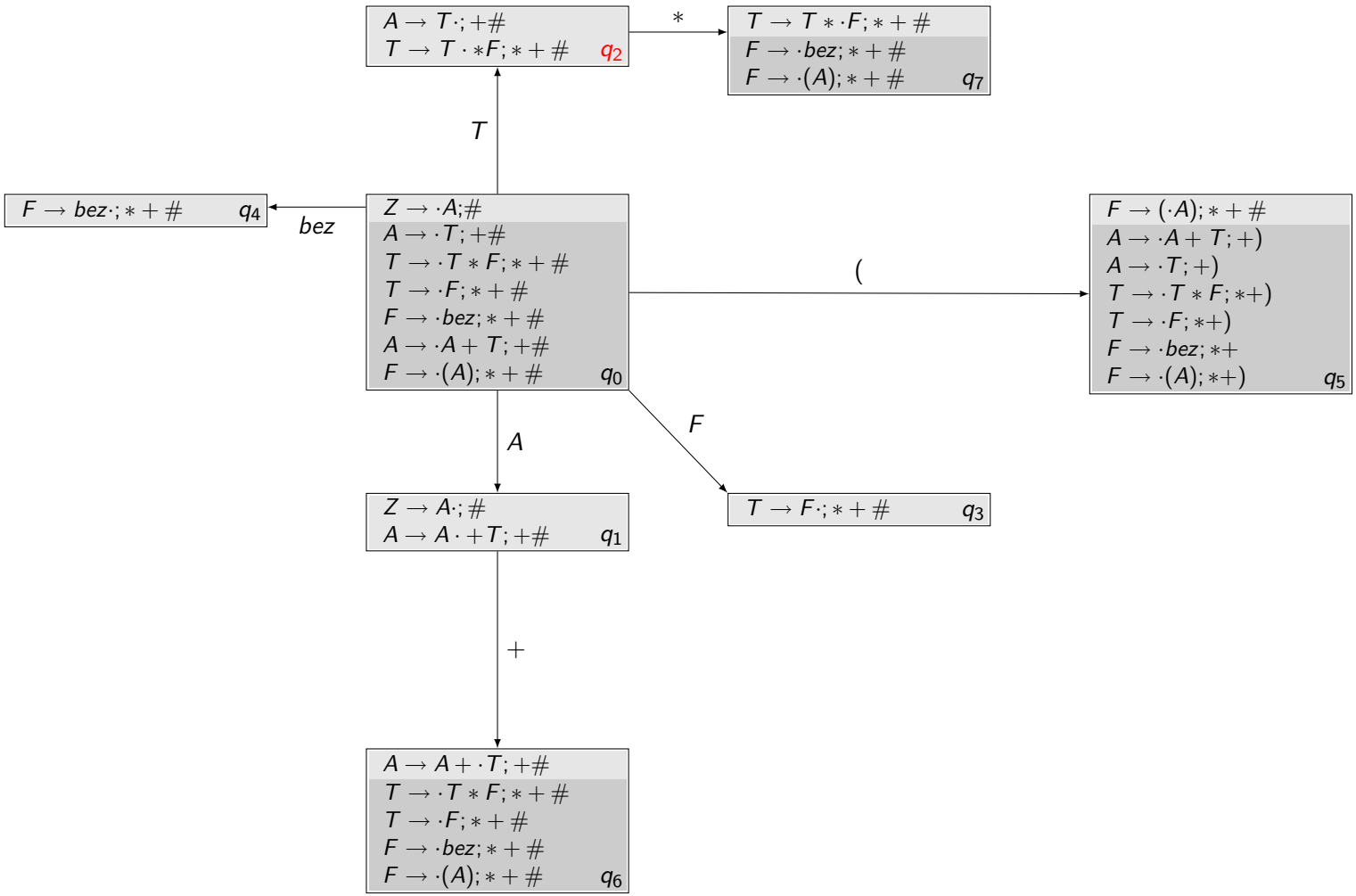
LALR(1)-Konstruktion am Beispiel



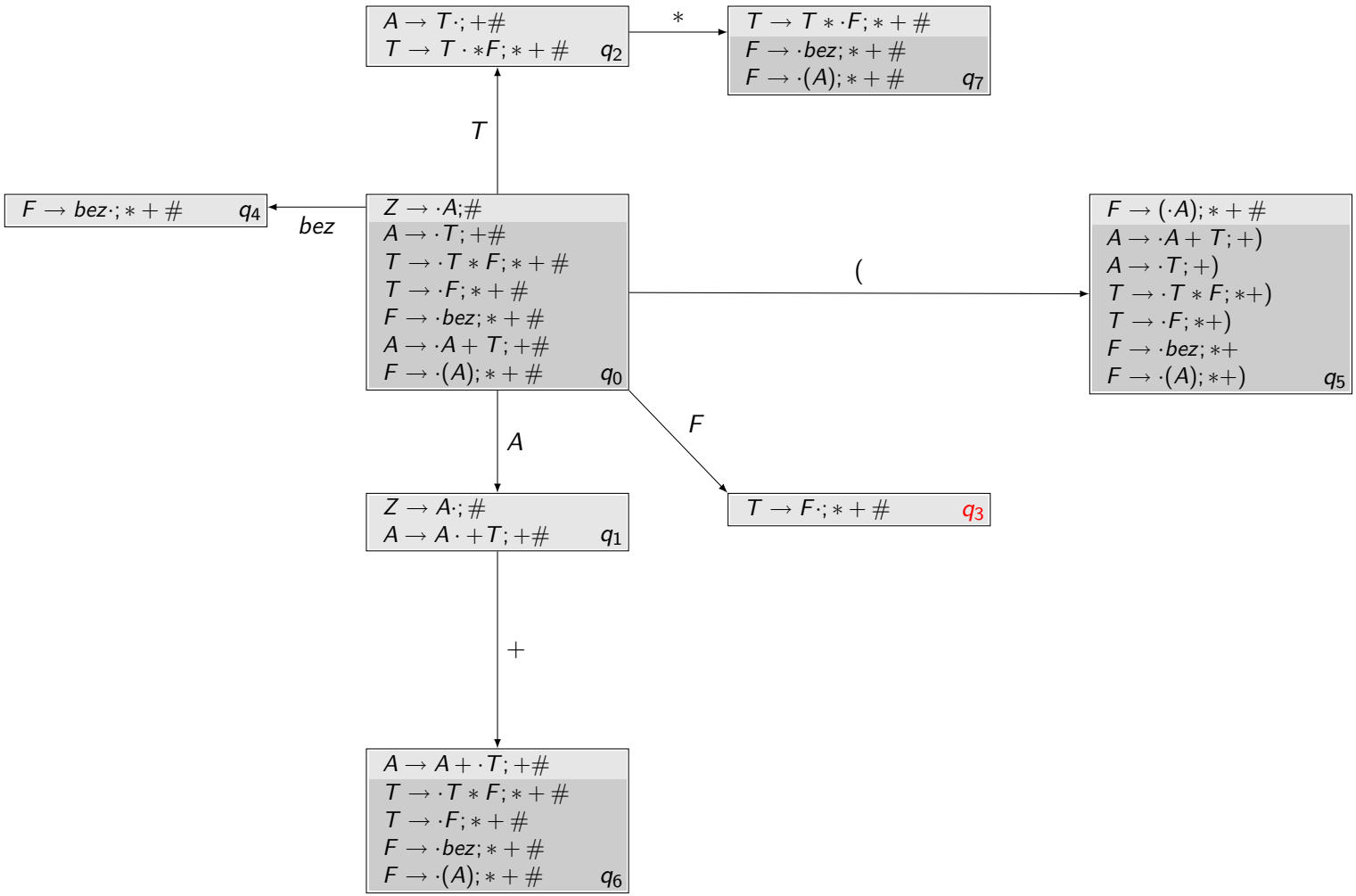
LALR(1)-Konstruktion am Beispiel



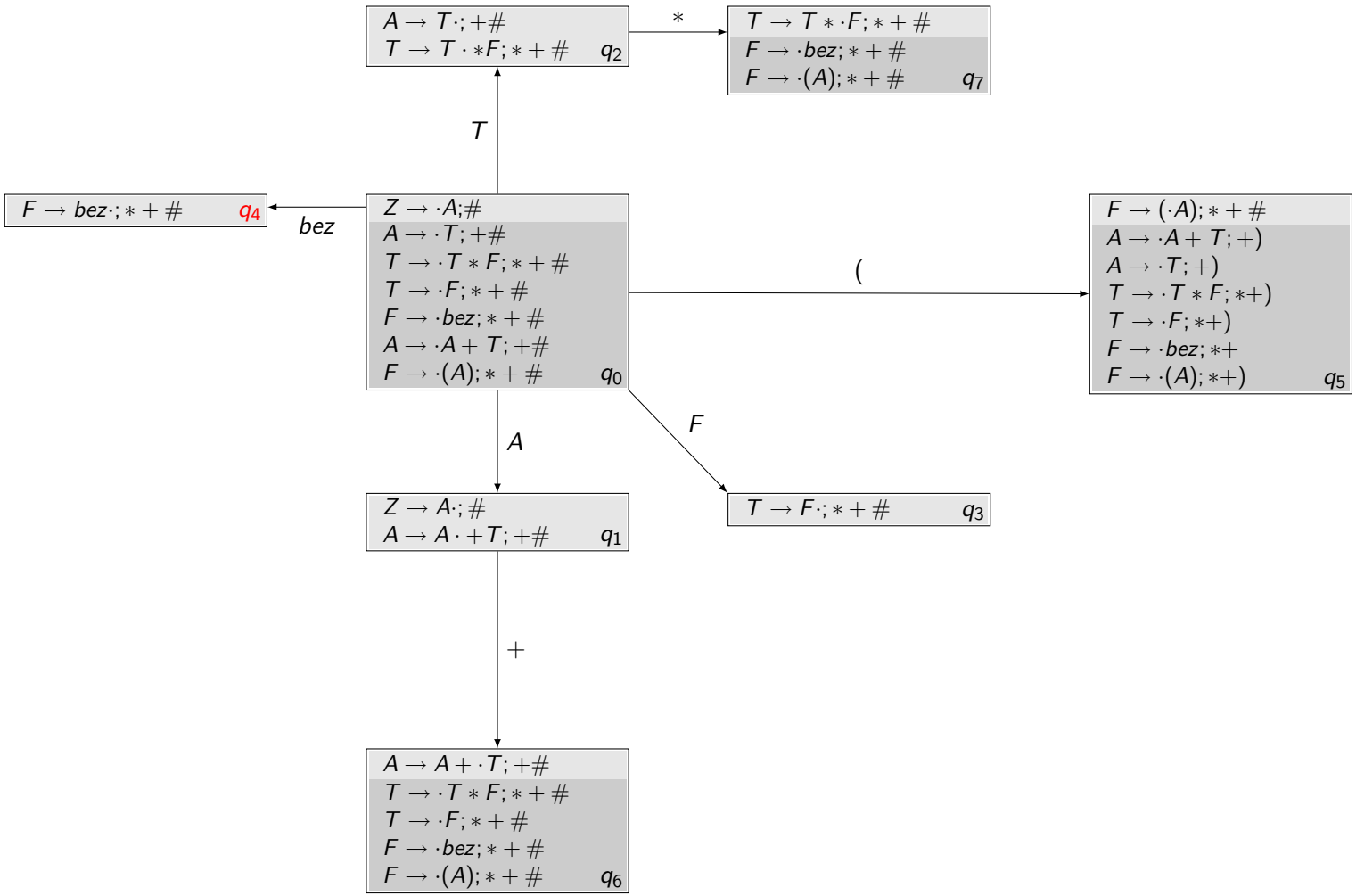
LALR(1)-Konstruktion am Beispiel



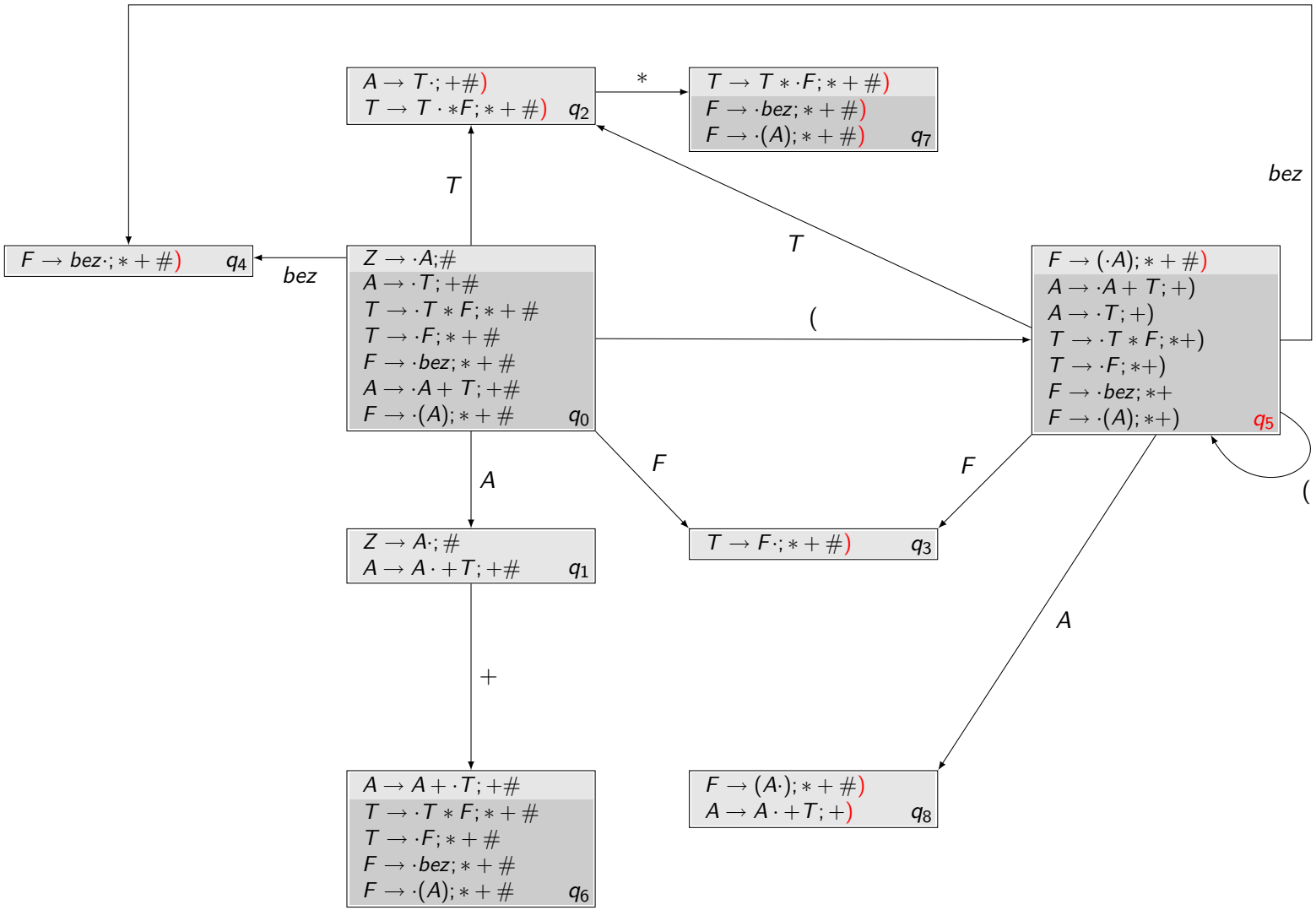
LALR(1)-Konstruktion am Beispiel



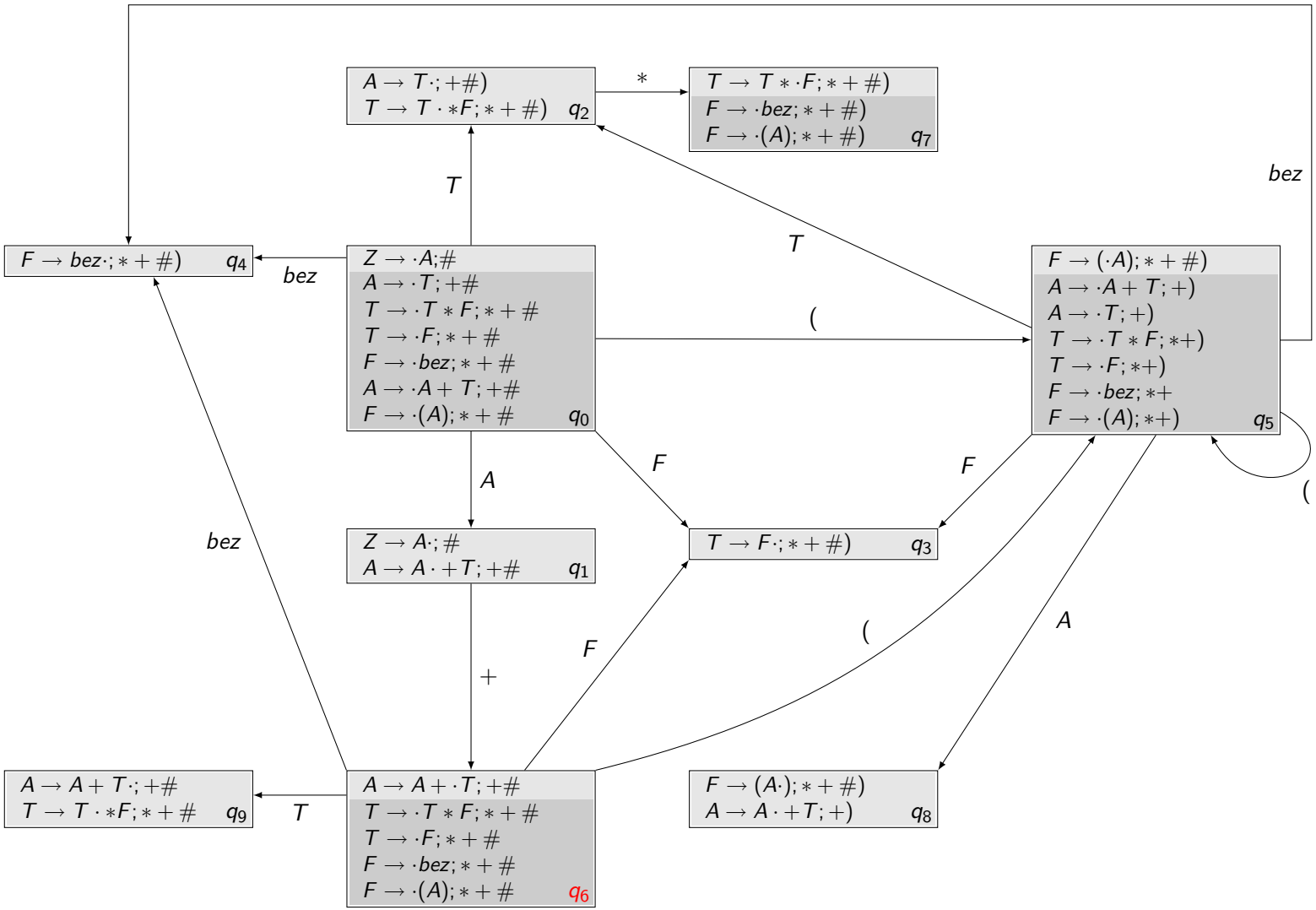
LALR(1)-Konstruktion am Beispiel



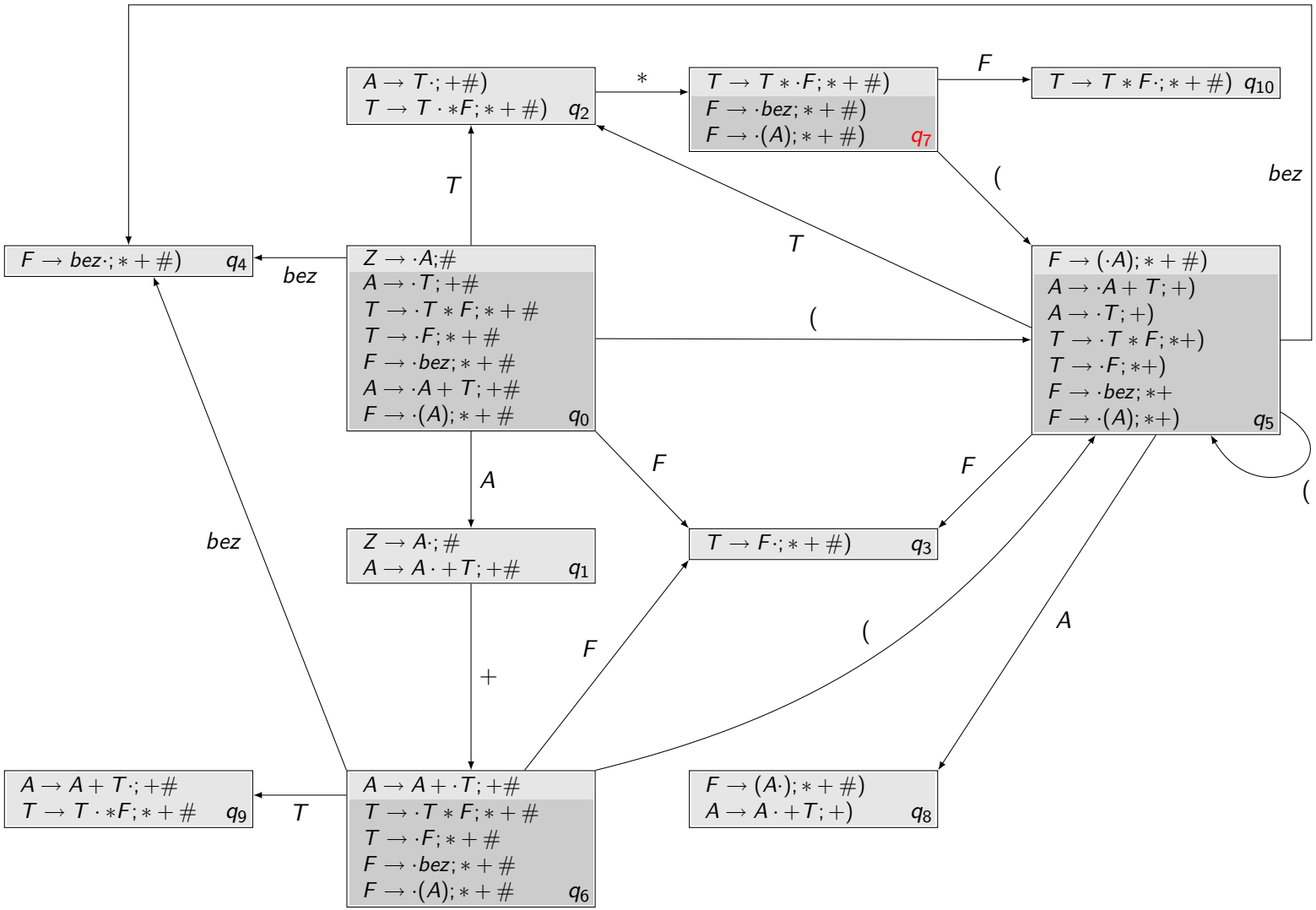
LALR(1)-Konstruktion am Beispiel



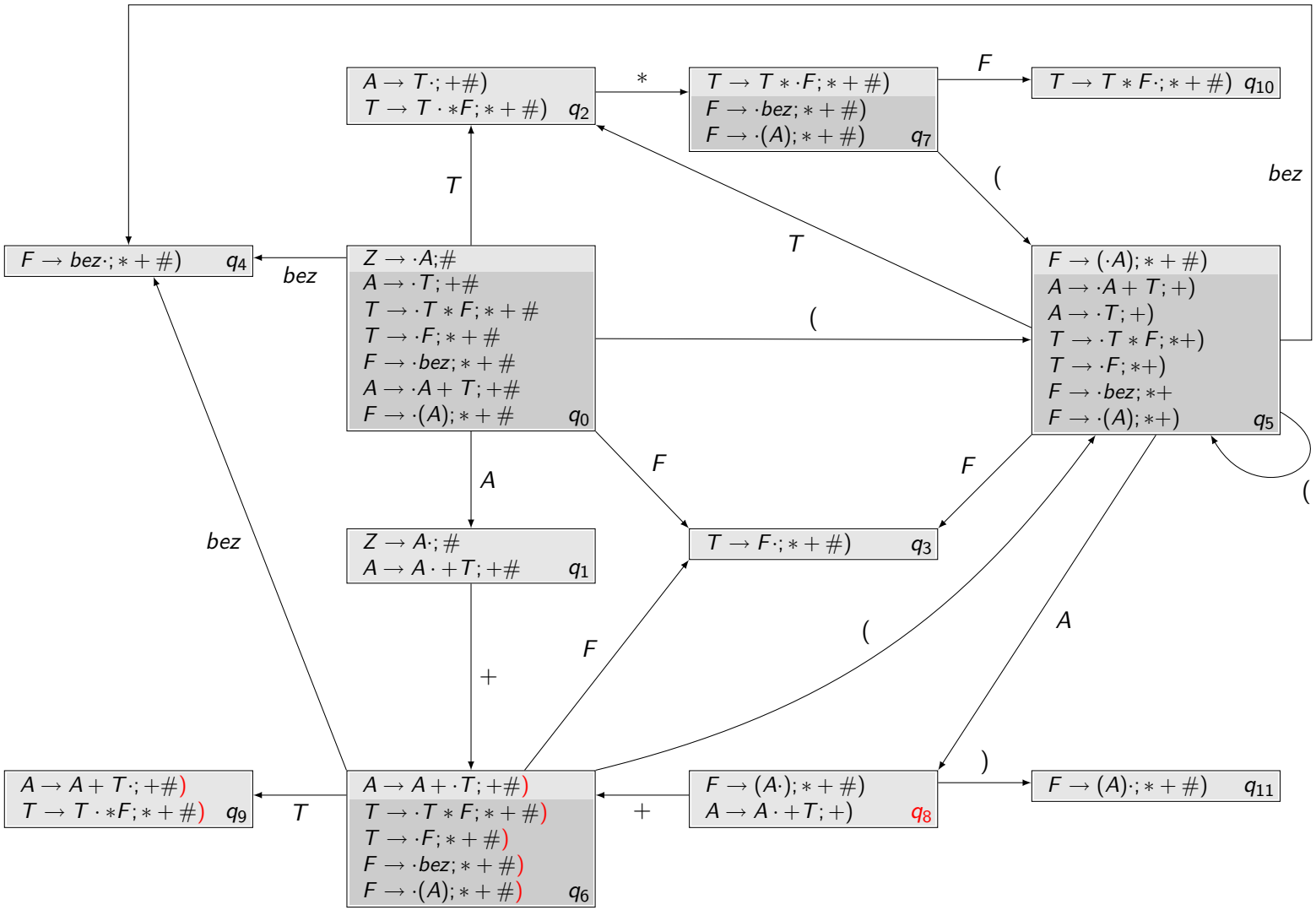
LALR(1)-Konstruktion am Beispiel



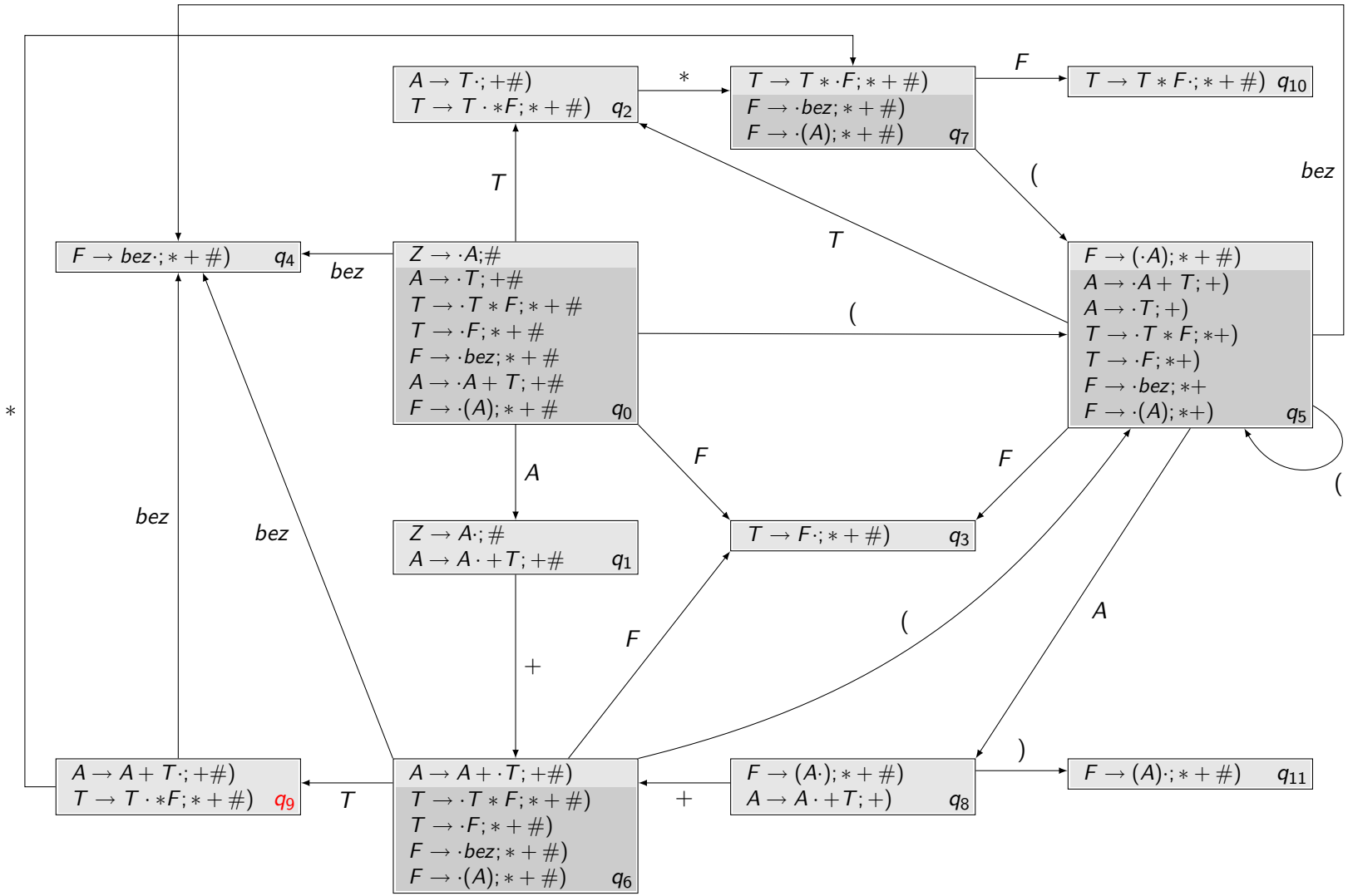
LALR(1)-Konstruktion am Beispiel



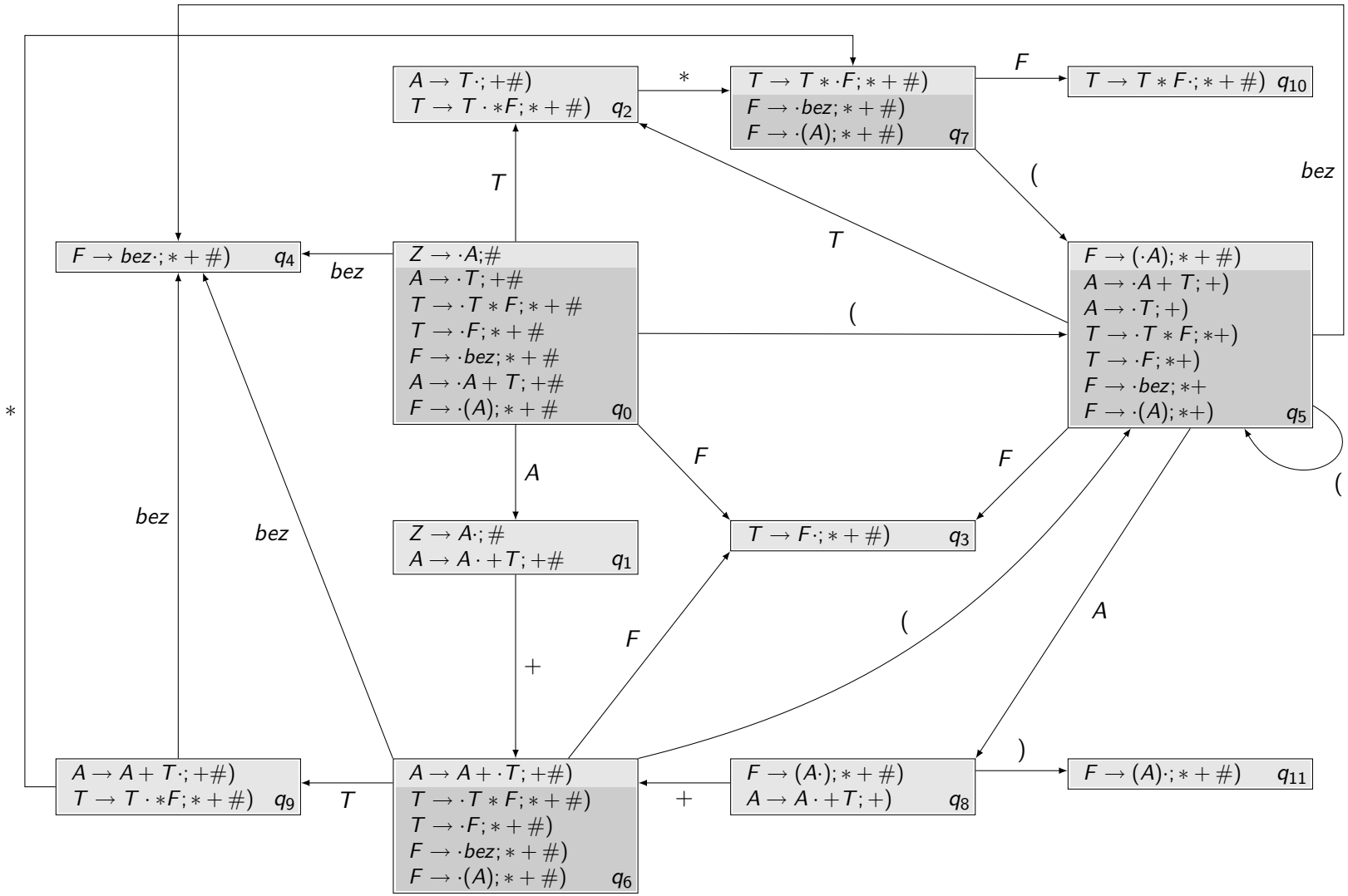
LALR(1)-Konstruktion am Beispiel



LALR(1)-Konstruktion am Beispiel



LALR(1)-Konstruktion am Beispiel



Übergangsfunktionen

q_0 $f(q_0, A) = q_1,\ f(q_0, T) = q2,\ f(q_0, F) = q3,$
$f(q_0, bez) = q4,\ f(q_0, () = q5$

q_1 $f(q_1, \#) = \text{HALT},\ f(q_1, +) = q6$

q_2 $f(q_2, +\#)) = \textit{Reduziere}(A \rightarrow T) f(q_2, *) = q7$

q_3 $f(q_3, * + \#)) = \textit{Reduziere}(T \rightarrow F)$

q_4 $f(q_4, * + \#)) = \textit{Reduziere}(F \rightarrow bez)$

q_5 $f(q_5, A) = q8,\ f(q_5, T) = q2,\ f(q_5, F) = q3,$
$f(q_5, bez) = q4,\ f(q_5, () = q5$

q_6 $f(q_6, T) = q9,\ f(q_6, F) = q3,\ f(q_6, bez) = q4,$
$f(q_6, () = q5$

q_7 $f(q_7, F) = q10,\ f(q_7, bez) = q4,\ f(q_7, () = q5$

q_8 $f(q_8,)) = q11,\ f(q_8, +) = q6$

q_9 $f(q_9, +\#)) = \textit{Reduziere}(A \rightarrow A + T),\ f(q_9, *) = q7$

q_{10} $f(q_1 0, * + \#)) = \textit{Reduziere}(T \rightarrow T * F)$

q_{11} $f(q_1 1, * + \#)) = \textit{Reduziere}(F \rightarrow (A))$

Übergangstabelle

	bez	(	)	+	*	#	A	T	F
0	4	5	-	-	-	-	1	2	3
1	-	-	-	6	-	*			
2	-	-	+3	+3	7	+3			
3	+5	+5	+5	+5	+5	+5			
4	+6	+6	+6	+6	+6	+6			
5	4	5	-	-	-	-	8	2	3
6	4	5	-	-	-	-		9	3
7	4	5	-	-	-	-			10
8	-	-	11	6	-	-			
9	-	-	+2	+2	7	+2			
10	+4	+4	+4	+4	+4	+4			
11	+7	+7	+7	+7	+7	+7			

- Fehler

i Schifte, neuer Zustand: i

+r Reduziere mit Regel r

„ “ Don't care

Fakten zu LR

- Automat $qt \to tq$, $t \in T$, $x_1 \ldots x_n q \to Xq$, $X \to x_1 \ldots x_n \in P$ deterministisch machen
  - dazu Reduktionsklassen bestimmen, Kellerklassen ableiten
  - Kellerklassen sind regulär, Produktionen mit Situationen als Nichtterminale
- charakteristischen Automaten herleiten:
  nicht-deterministischen Automaten in deterministischen überführen (Hüllenbildung)
- $\mathrm{LR}(k)$-Automat zu groß: $\mathrm{SLR}(k)$-Klasse (Vorschau: $\mathrm{Folge}_k(A)$) ungenügend
- $\mathrm{LALR}(k)$: Zustände des $\mathrm{LR}(k)$-Automaten mit gleichem Kern verschmelzen
- SLR- und LALR-Automat hat gleiche Zustandsmenge wie $\mathrm{LR}(0)$-Automat

Fragen zu LR

- Wie lautet die Übergangsfunktion?
- Wie bestimmt man Zustände und Übergänge des LALR(1)-Automaten von Hand?
- Wie korrigiert man Grammatik, wenn es inadäquate Zustände gibt?

Generatoren für die syntaktische Analyse

- Parsergeneratoren ermöglichen eine kompakte Spezifikation syntaktischen Regeln und Aktionen beim erkennen solcher Regeln (eine Domain Specific Language).
- Eingabe: Ein Tokenstrom
- Ausgabe: Bei erkannten Regeln lassen sich beliebige Aktionen durchführen, die in der Beschreibungssprache spezifiziert werden. Mögliche Anbindungen an einen Compiler:
  - Aufbau des AST als Datenstruktur.
  - unmittelbare semantische Analyse und Codeausgabe (Single-Pass-Compiler).
  - Erzeugen von Zwischensprachen.

Generatoren Kriterien

Heute existieren sehr viele unterschiedliche Parsergeneratoren. Die Liste auf

http://wikipedia.org/wiki/Comparison_of_parser_generators

hat zur Zeit 101 Einträge!
Bei der Auswahl eines Generators hilft es sich an den folgenden Kriterien zu orientieren:

- Parsingtechnik: $\mathrm{LALR}(1)$, $\mathrm{LL}(k)$, GLR, Packrat
- Programmiersprache(n) in denen der Parser erzeugt werden kann
- Anbindung an Werkzeuge zur lexikalischen Analyse und attributierter Grammatiken
- Einbettung in Entwicklungsumgebungen

Generatoren (eine Auswahl)

- *Yacc*: Erzeugen $\mathrm{LALR}(1)$-Parser, bekannt aus dem Unix-/C-Umfeld.
- *Bison*: Erweiterte und Verbesserte Open-Source Variante von Yacc.
- Weitere $\mathrm{LALR}(1)$-Parsergeneratoren: *SableCC*, *cup*
- Einer bekannter Generator im Java-Umfeld ist *antlr*. Dieser erzeugt trotz seines Namens $\mathrm{LL}(k)$ Generatoren. Er kann auch Parser in anderen Sprachen als Java erzeugen.
- Weitere $\mathrm{LL}(k)$-Generatoren: *JavaCC* (Java), *Coco/R* (Java, C#, weitere), *happy* (haskell), *ocamlyacc* (ocaml)

Parser für XML

```
<?xml version='1.0'?>
<!-- my personal books -->
<books>
    <book name='Goedel, Escher, Bach'/>
</books>
```

$$
\begin{aligned}
\textit{input} &\rightarrow \textit{prolog element} \mid \textit{element} \\
\textit{prolog} &\rightarrow \texttt{<?}\ \textbf{xml version} = \textbf{string}\ \texttt{?>} \\
\textit{element} &\rightarrow \textit{simple\_tag} \mid \textit{start\_end\_tags} \\
\textit{start\_end\_tags} &\rightarrow \texttt{<}\ \textbf{name}\ \textit{attributes}\ \texttt{>}\ \textit{content}\ \texttt{</}\ \textbf{name}\ \texttt{>} \\
\textit{simple\_tag} &\rightarrow \texttt{<}\ \textbf{name}\ \textit{attributes}\ \texttt{/>} \\
\textit{attributes} &\rightarrow \varepsilon \mid \textit{attribute attributes} \\
\textit{attribute} &\rightarrow \textbf{name} = \textbf{string} \\
\textit{content} &\rightarrow \varepsilon \mid \textit{element content}
\end{aligned}
$$

Parser für XML

```
<?xml version='1.0'?>
<!-- my personal books -->
<books>
    <book name='Goedel, Escher, Bach'/>
</books>
```

$$
\begin{aligned}
\textit{input} &\rightarrow \textit{prolog}\ \textit{element} \mid \textit{element} \\
\textit{prolog} &\rightarrow <?\ \textbf{xml version} = \textbf{string}\ ?> \\
\textit{element} &\rightarrow \textit{simple\_tag} \mid \textit{start\_end\_tags} \\
\textit{start\_end\_tags} &\rightarrow <\ \textbf{name}\ \textit{attributes} >\ \textit{content} <\ /\ \textbf{name} > \\
\textit{simple\_tag} &\rightarrow <\ \textbf{name}\ \textit{attributes}\ /\ > \\
\textit{attributes} &\rightarrow \varepsilon \mid \textit{attribute}\ \textit{attributes} \\
\textit{attribute} &\rightarrow \textbf{name} = \textbf{string} \\
\textit{content} &\rightarrow \varepsilon \mid \textit{element}\ \textit{content}
\end{aligned}
$$

Deklarationen

Der erste Teil einer Bison Spezifikation besteht aus Deklarationen:

- Der Typ eines Tokens auf dem Stack wird als C-union mit %union angegeben.
- Die Tokenbezeichner werden mit Hilfe von %token festgelegt
- Vor einem Token kann dessen Typ in spitzen Klammern angegeben werden.
- Nach dem Token kann eine optionale Umschreibung als String angegeben werden.
- Alternativ kann man auch %left und %right angeben um eine Präzedenz zur Auflösung von Konflikten bei mehrdeutigen Grammatiken anzugeben.

Deklarationen: Beispiel

```
/* token value definitions */
%union {
  const char *string;
}

/* token definitions */
%token <string> T_STRING
%token <string> T_NAME
%token T_XML T_VERSION
%token T_LESS_QUESTIONMARK "<?"
%token T_LESS_SLASH "</"
%token T_SLASH_GREATER "/>"
%token T_QUESTIONMASK_GREATER "?>"
```

Regelspezifikation

Eine grammatische Regel der Form:

$$Kopf \rightarrow Rumpf_1 \mid Rumpf_2 \mid \ldots \mid Rump_n$$

wird folgendermaßen spezifiziert:

```
kopf : rumpf1 { /* semantische Aktion 1 */ }
   | rumpf2 { /* semantische Aktion 2 */ }
     ...
   | rumpfn { /* semantische Aktion n */ }
;
```

In einem Rumpf wird eine Folge von Terminalen und Nichtterminalen angegeben. Nichtterminale werden durch ihren Bezeichner angegeben. Terminale durch Bezeichner, **'x'** bei einzelnen Zeichen, oder ¨xx¨ (falls xx als Umschreibung eines Tokens definiert wurde).

Beispiel: Statements

```
statement : if_statement | while_statement | for_statement
   | expression_statement | compound_statement
;

if_statement : T_IF '(' expression ')' statement
   | T_IF '(' expression ')' statement T_ELSE statement
;

while_statement: T_WHILE '(' expression ')' statement ;

for_statement: T_FOR '(' expression ';' expression ';' expression ')' statement ;

expression_statement: expression ';'

compound_statement: '{' statements '}' ;

statements: /* empty */ | statement statements ;
```

Beispiel: Expressions

```
expression: expression '+' expression
   | expression '-' expression
   | expression '*' expression
   | expression '/' expression
   | call_expression
   | '(' expression ')'
   | T_IDENT
   | T_LITERAL
;

call_expression: expression '(' arguments ')' ;

arguments: /* empty */ | expression | multi_arguments ;

multi_arguments: expression
   | expression ',' multi_arguments ;
```

Beachte: Mehrdeutigkeiten werden hier durch Spezifikation von Tokenpräzedenzen aufgelöst.

Beispiel: XML Parser

input: prolog element | element ;

prolog: "<?" T_XML T_VERSION '=' T_STRING "?>" ;

element: start_end_tags | simple_tag ;

start_end_tags: '<' T_NAME attributes '>'
  content
  "</" T_NAME '>'
;

attributes: /* empty */ | attributes attribute ;

attribute: T_NAME '=' T_STRING ;

simple_tag: '<' T_NAME attributes "/>" ;

content: /* empty */ | content element ;

Grammatik 1: antlr, Java (1/2)

```
// Definition of a Java class
jClassDefinition[int mods]
  returns [JClassDeclaration self = null]
{
  String superClass = null;
  CClassType[] interfaces = CClassType.EMPTY;
  CParseClassContext context = new CParseClassContext();
  TokenReference sourceRef = buildTokenReference();
  JavadocComment javadoc = getJavadocComment();
  JavaStyleComment[] comments = getStatementComment();
}
:
...
```

Grammatik 1: antlr, Java (2/2)

```
...
  "class" ident:IDENT
  superClass = jSuperClassClause[]
  interfaces = jImplementsClause[]
  jClassBlock[context] // the body of the class
  {
    self = new JClassDeclaration(sourceRef,
    mods, ident.getText(),
    superClass, interfaces,
    context.getFields(),
    context.getMethods(),
    context.getInnerClasses(),
    context.getBody(),
    javadoc, comments);
  };
```

Grammatik 2: bison, Java (1/3)

```
/* 19.8.1 Production from §8.1: Class Declaration */
class_declaration:
modifiers CLASS_TK identifier super interfaces
   { create_class ($1, $3, $4, $5); }
class_body
   {;}
| CLASS_TK identifier super interfaces
{ create_class (0, $2, $3, $4); }
...
```

Grammatik 2: bison, Java (2/3)

```
...
class_body
   {;}
| modifiers CLASS_TK error
   { yyerror ("Missing class name"); RECOVER; }
| CLASS_TK error
   { yyerror ("Missing class name"); RECOVER; }
| CLASS_TK identifier error
   { if (!ctxp->class_err)
        yyerror ("'{' expected");
     DRECOVER(class1);
   }
| modifiers CLASS_TK identifier error
   { if (!ctxp->class_err)
        yyerror ("'{' expected"); RECOVER;}
;
...
```

Grammatik 2: bison, Java (3/3)

```
...
super:
   { $$ = NULL; }
| EXTENDS_TK class_type { $$ = $2; }
| EXTENDS_TK class_type error
   {yyerror ("'{' expected"); ctxp->class_err=1;}
| EXTENDS_TK error
   {
     yyerror ("Missing super class name");
     ctxp->class_err=1;
   }
;
```

Tabellenoptimierung

- $\mathrm{LR}(0)$-Reduktionszustand: Zustand, in dem auf jeden Fall reduziert wird (Kontext unerheblich)
- $\mathrm{LR}(0)$-Reduktionszustände können beseitigt werden, indem im vorigen Zustand bereits reduziert wird (Schift-Reduktionszustand)
- Kettenproduktionen eliminieren
- echte und unechte (don't care) Fehlerübergange unterscheiden.
- Unecht: Übergang kann nie erreicht werden, z.B. alle leeren Übergänge mit Nichtterminalen
- Fehlerübergänge ausfaktorisieren in Fehlermatrix F:

  $f(q,t) =$ **if** $F[q,t]$ then Fehler **else** Eintrag_in_Übergangsmatrix
- Übergangsmatrix komprimieren: leere Übergänge berücksichtigen
- Übergangsmatrix weiter komprimieren (siehe nächste Folie)

Tabellenkompression

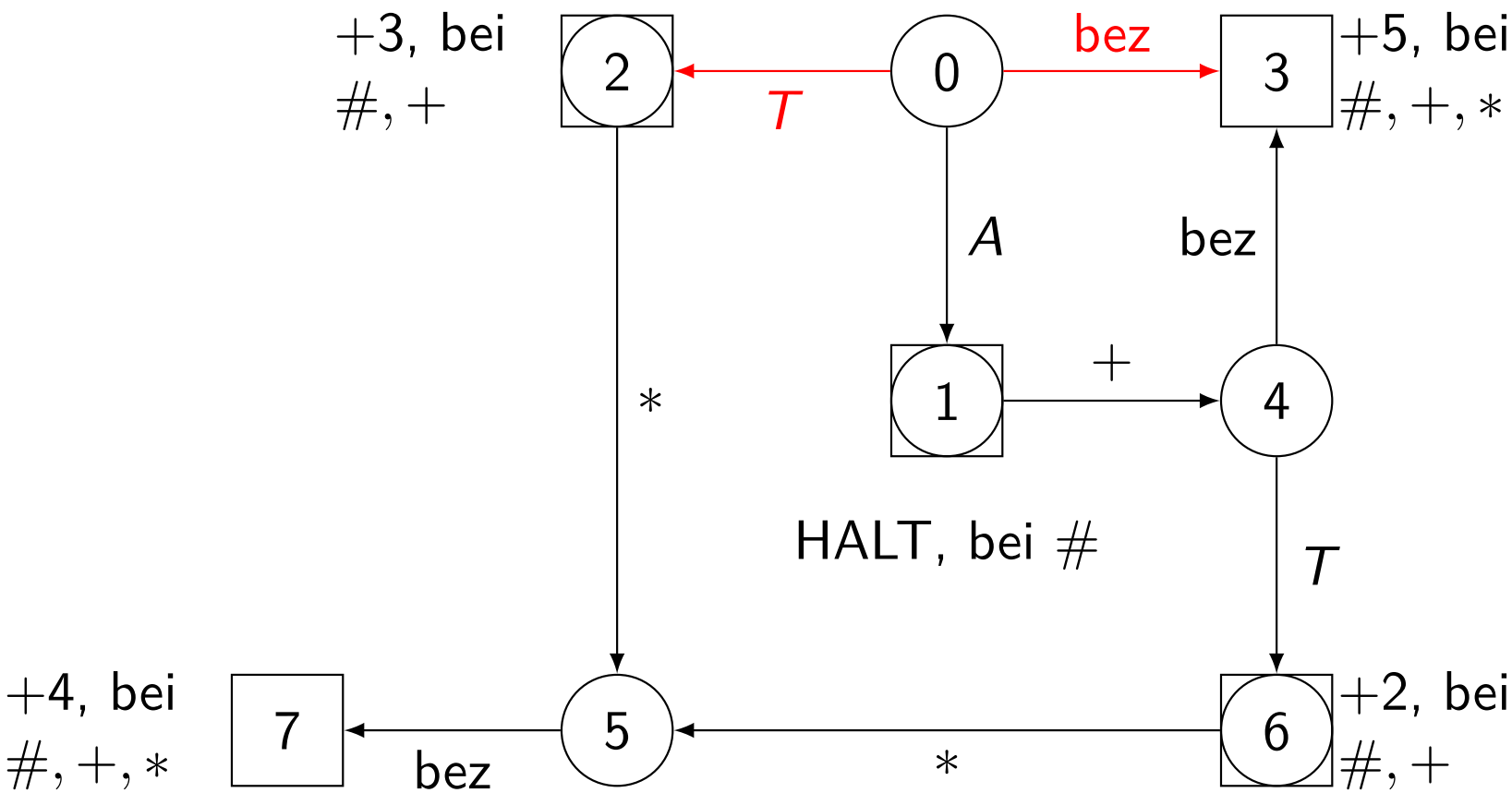
Methoden zur weiteren Kompression der Übergangsmatrix:

- Graphenfärben: Unverträglichkeitskanten zwischen ungleichen/unverträglichen Tabelleneinträgen. Gleichgefärbte Zeilen/Spalten(-Teile) können zusammengelegt werden.
- Index-Zugriffs-Methoden: Umsortieren und Kombinieren der Einträge, so dass mit konstant vielen Indizierungen der Eintrag findbar ist.
- Listen-Suche: Suche nach dem richtigen Eintrag via Schlüssel und variabler Vergleichsanzahl.
- Hinweise:
  - Verwendbar bei beliebigen dünn/gleichförmig besetzten Tabellen
  - Historisch: Bedingung für Anwendbarkeit ($|\text{Tabelle}| <$ Hauptspeicher) Heute: Geschwindigkeit ($|\text{Tabelle}| <$ Cachegröße)

Kettenproduktionen

1: $Z \rightarrow A$ 2: $A \rightarrow A + T$ 3: $A \rightarrow T$

4: $T \rightarrow T * \mathbf{bez}$ 5: $T \rightarrow \mathbf{bez}$



Kettenproduktionen Eliminieren

1: $Z \rightarrow A$ 2: $A \rightarrow A + T$ 3: $A \rightarrow T$

4: $T \rightarrow T * \mathbf{bez}$ 5: $T \rightarrow \mathbf{bez}$

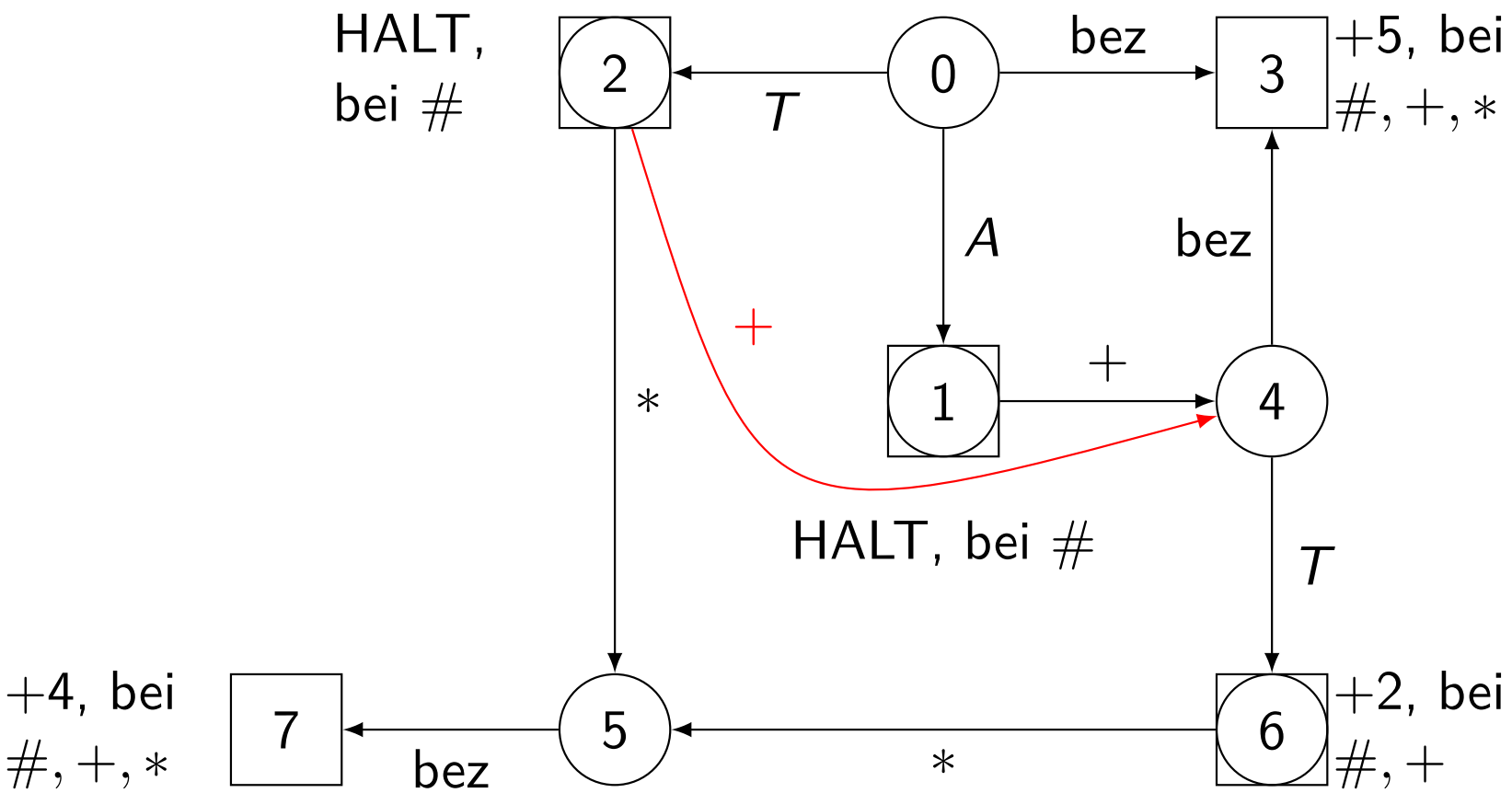


Tabelle mit Schift-Reduktionen, ohne Kettenproduktionen

	bez	(	)	+	∗	#	A	T	F
0	-6	3	-	-	-	-	1	2	2
1			-	4		*			
2	-	-	-	4	5	*			
3	-6	3	-	-	-	-	6	7	7
4	-6	3	-	-	-	-		8	8
5	-6	3	-	-	-	-			-4
6			-7	4		-			
7	-	-	-7	4	5	-			
8	-	-	+2	+2	5	+2			

- Fehler

i Schifte, neuer Zustand: i

„ “ Don’t care

+r Reduziere mit Regel r

-r Schifte, reduziere dann mit Regel r

Fehlermatrix und komprimierte Übergangsmatrix

	bez	(	)	+	*	#
0	f	f	t	t	t	t
1			t	f		f
2	t	t	t	f	f	f
3	f	f	t	t	t	t
4	f	f	t	t	t	t
5	f	f	t	t	t	t
6			f	f		t
7	t	t	f	f	f	t
8	t	t	f	f	f	f

	A	T,F
0,1,2	1	2
3	6	7
4		8
5,6,7,8		-4

	bez	(	)	+	*	#
0,1,2,3	-6	3	-7	4	5	*
4,5,6,7						
8			+2	+2	5	+2

Tabellengröße

Tabellengröße nach Elimination von Ketten (Beispiel ADA 83):

- 95 Terminale, 252 Nichtterminale
- 540 Zustände (kodiert in 2 Bytes)
- Tabellengröße: 374.760 Bytes

Reduktion des Platzbedarfes durch:

- Einfache Tabellenkomprimierung
- Elimination der Fehlerübergänge bei Nichtterminalen

Tabellengröße nach diesen Reduktionen 22.584 Bytes (ca. 6%)

Vergleich verschiedener Parsergeneratoren

	Bison	yacc	PGS	Lalr	Ell
Grammatik	$\mathrm{LALR}(1)$	$\mathrm{LALR}(1)$	$\mathrm{LALR}(1)$	$\mathrm{LALR}(1)$	$\mathrm{LL}(1)$
Spezifiziert in	BNF	BNF	EBNF	EBNF	EBNF
Geschwindigkeit in [10^3 Symbol / Sekunde]	8.93	15.94	17.32	34.94	54.64
Geschwindigkeit in [10^3 Zeilen / Minute]	150	270	290	580	910
Tabellengröße in [bytes] (komprimiert)	7724	9968	9832	9620	-
Parsergröße in [bytes]	10900	12200	14140	16492	18048

Eingabe: Modula-2 Code.

Hardware: PCS Cadmus mit MC68020 Prozessor (16.7 MHz).

Komplexität Parsen in $\mathcal{O}(n)$

Für alle Klassen gilt:

- Jeder Ableitungsschritt benötigt, da niemals ein Rücksetzen notwendig ist, konstanten Aufwand $\mathcal{O}(1)$.
- Parsen benötigt Aufwand $\mathcal{O}(n)$ wobei n die Anzahl der Ableitungsschritte ist.

Beweis über Grad der Knoten und mögliche Höhe des Baumes (Kettenproduktionen und Epsilonproduktionen berücksichtigen)

Komplexität

n : Anzahl der Produktionen, k : Länge der Vorausschau

Grammatiktyp	Parsergenerierung	Test der Grammatik
$\mathrm{LL}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$
$\mathrm{SLL}(k)$	$\mathcal{O}(n^{k+1})$	$\mathcal{O}(n^{k+1})$
$\mathrm{LL}(k)$	$\mathcal{O}(2^{n^{k+1}+(k+1)\log n})$	$\mathcal{O}(n^{2k})$
$\mathrm{SLR}(1)$	$\mathcal{O}(2^{n+\log n})$	$\mathcal{O}(n^2)$
$\mathrm{SLR}(k)$	$\mathcal{O}(2^{n+k\log n})$	$\mathcal{O}(n^{k+2})$
$\mathrm{LR}(k)$	$\mathcal{O}(2^{n^{k+1}+k\log n})$	$\mathcal{O}(n^{2(k+1)})$

Sätze über kontextfreie Grammatiken

Satz 1: Für jede $\mathrm{LR}(k)$-Grammatik G mit $k > 1$ gibt es eine $\mathrm{LR}(1)$-Grammatik G' mit $L(G) = L(G')$.
Beweis durch Rechtsfaktorisierung.

Satz 2: Jede $\mathrm{LL}(k)$-Grammatik ist auch $\mathrm{LR}(k)$.

Satz 3: Es gibt $\mathrm{LR}(k)$-Grammatiken, die für kein k' $\mathrm{LL}(k')$ sind.

Satz 4: Es ist entscheidbar, ob es für eine gegebene $\mathrm{LR}(k)$-Grammatik G ein k' gibt, so dass G $\mathrm{LL}(k')$ ist.

Satz 5: Es ist unentscheidbar, ob für eine Sprache L eine Grammatik G existiert, so dass G $\mathrm{LL}(1)$ ist.

Satz 6: Es ist unentscheidbar, ob es für eine Sprache L eine Grammatik G gibt, so dass G $\mathrm{LL}(k)$ oder $\mathrm{LR}(k)$ ist.

Kapitel 3: Syntaktische Analyse

Fehlerbehandlung (1/2)

Ziel: Analyse soll formal korrekte Ergebnisse liefern (oder wegen zu vieler Fehler abbrechen), um möglichst viele Fehler zu finden.

- Gefahr von Folgefehlern wegen unpassender Korrektur unvermeidbar
- Unterscheide Fehler (= Ursache) und Fehlersymptom (= beobachtbare Wirkung)

Reaktionen:

- Bericht: Immer notwendig
- Reparatur, wenn Fehlerursache feststellbar und korrigierbar
- Wiederaufsetzen: Internen Zustand konsistent machen und weitere Fehler suchen
- Abbruch bei zu vielen Fehlern oder bei zu hohem Ressourcenverbrauch (Tabellenüberlauf, ...)

Hinweis: Dieser Abschnitt behandelt Fehlermeldungen aller Übersetzerphasen, nicht nur syntaktische!

Fehlerbehandlung (2/2)

Fehler: Nicht (vom Programmierer) intendiertes Programm

Fehlersymptom:

- Sichtbare Auswirkung des Fehlers: Verletzung der Sprachdefinition
- parserdefinierte Fehlerstelle.

Diagnose:

- Versuch, den Fehler auf der Grundlage des Symptoms zu erkennen.
- Entsprechende Fehlermeldung, Reparatur oder Wiederaufsetzen.

Beispiel

Zuweisung:	$x := (a + b * c;$
Fehler (vermutlich):	) nach b fehlt.
Fehlersymptom:	) fehlt vor ;

Position des Fehlersymptoms ist nicht die Fehlerstelle!

Klassifikation

Fehlerklassen:

- Anomalien (verdächtig, aber nicht gefährlich)
  - Notiz, z.B. keine Standardkonstruktion
  - Kommentar, z.B. unpassender Programmierstil
  - Warnung bei möglichen Fehler, z.B. unbenutzte Variable
- Fehler
  - einfacher Fehler: reparierbar, Code kann erzeugt werden
  - fataler Fehler: kein Code erzeugbar, nur Wiederaufsetzen möglich
  - Abbruchfehler: Übersetzer gibt auf (z.B. wegen Ressourcenbeschränkung)

Fehlermeldung

- Ausgabe von
  - Position (Datei, Zeile, Spalte),
  - Fehlerklasse,
  - Meldung
- Intern: Meldung kodiert durch ganze Zahl
- Meldungstexte in getrennter Datei zur Anpassung der Sprache
- Meldungen fallen nicht in der Reihenfolge des Eingabetexts an:
  - Meldungen aufsammeln
  - möglichst nach Position sortiert ausgeben

Unterscheidung nach Übersetzerphase

1. Symbolfehler: unzulässiges Eingabezeichen, Abschluß Textkonstante oder Kommentar fehlt (falls erkennbar), verfrühtes Eingabeende: kein Endzustand des Symbolentschlüsselungsautomaten erreicht.
   Reparatur: falsche Zeichen oder vorhandenen Symbolanfang ignorieren
2. Syntaktischer Fehler: Satz gehört nicht zu der durch den Parser definierten Obermenge der Sprache.
3. Semantischer Fehler: Fehler in der statischen Semantik.
4. Semantischer Fehler, der erst in der Optimierung entdeckt wird, z.B. Verletzung Indexgrenzen bei Reihungsindizierung.
5. Ressourcenbeschränkung verletzt (stets Abbruch), in allen Phasen möglich.

Parserdefinierte Fehlerstelle

parserdefinierte Fehlerstelle t:

$$\forall r \in \Sigma^* : str \notin L \text{ und } \exists s' : ss' \in L.$$

LL-, SLL-, LR-, SLR- und LALR-Parser finden die parserdefinierte Fehlerstelle.

Andere Parser, z.B. für Präzedenzgrammatiken, finden sie nicht.

Beweisskizze für $\mathrm{LL}(k)$

Angenommen t parserdefinierte Fehlerstelle, d.h. $str \notin L$,
$\exists s' : ss' \in L$.
Sei s im Keller und $q = [P \rightarrow P_{anf} \cdot P_{rest}; \Omega]$

Terminal t ist parserdefinierte Fehlerstelle gdw:

$$\nexists k : ts \in \mathrm{Anf}_k(P_{rest}\Omega)$$

Ideale Korrektur

Minimale Anzahl von Operationen

- Einsetzen
- Streichen
- Ersetzen (= Streichen + Einsetzen)

ausführen

Tatsächliche Korrektur

- Panischer Modus: Wiederaufsetzen an Anweisungs- oder Vereinbarungsende
- Systematische Fortsetzung an parserdefinierter Fehlerstelle $str \notin L$: frühest möglichen Wiederaufsetzpunkt finden
- Totalkorrektur:
  - Eingabe $str = s_1 x_1 \ldots s_n x_n$, $s_1 = s$, n minimal, so aufteilen, dass alle s_i Ausschnitte einer korrekten Eingaben sind, die durch eventuell unbrauchbare Texte x_i verknüpft sind
  - die x_i durch korrekte Texte y_i so ersetzen, dass $s_1 y_1 \ldots s_n y_n$ korrekt ist
  - Nachteil: quadratischer Aufwand, praktisch nicht eingesetzt
- Die Zeichen, an denen wieder aufgesetzt werden kann, bilden die Ankermenge.

zahlreiche weitere Verfahren bekannt

Panischer Modus

alle Symbole bis Anweisungs- oder Vereinbarungsende streichen
Keller soweit abbauen, dass Folgesymbol ; end, }, ... akzeptiert wird
Fehlermeldung ausgeben und Analyse fortsetzen

Vorteil:

- einfache Implementierung

Nachteile:

- keine Analyse des Anweisungsrests
- Schwierigkeiten mit korrektem Abschluß von Klammerungen **if** ... then, then ... **else**, usw.

Fehlerbehandlung für LL-Parser

Panikmodus: Überlese Tokens bis nächstes Token in synchronisierender Menge (Ankermenge). Ankermengen werden beim rekursiven Abstieg mitübergeben bzw. stehen zu einem Nichtterminal auf dem Stack.
Berechnung der Ankermenge für Nichtterminal A: Ank(A)

1. erste Näherung: FOLLOW(A). Aber das reicht nicht: z.B. fehlendes Semikolon würde zu Folgefehlern führen.
2. Erweiterung von Ank(A) um solche Tokens, die „übergeordnete" Strukturen beginnen. Beispiel: *statement* ist übergeordnet zu *expression*: FIRST(*statement*) wird zu Ank(*expression*) hinzugefügt.
3. wenn $\varepsilon \in L(A)$, wird an Fehlerstelle $A \longrightarrow \varepsilon$ expandiert und normal fortgefahren.
4. wenn ein Topstack-Terminal nicht kommt, wird es gepoppt nebst Meldung „Terminal xx expected".

C (gcc) – Panischer Modus

```
int main ( ) {
    int j, i ;
    if (i<j) {{ /*FEHLER*/
        i = j ;
    } /*FEHLER–SYMPTOM 1: } erwartet*/
    else {
        if ( i != j ){
            j: = i ; /*FEHLER 2 bleibt unerkannt*/
        } /*Ende if */
    } /*Ende main*/
    return 0 ; /*FEHLER–SYMPTOM 2: return unerwartet*/
}
```

```
test.c: In function 'main':
test.c: 6: parse error before 'else'
test.c: At top level:
test.c: 11: parse error before 'return'
```

Systematische Korrektur (Röhrich)

gegeben parserdefinierter Fehlerstelle $str \notin L$

gesucht: Fortsetzung $sy \in L$:

- Bestimme eine Ankermenge $D = \{d \in \Sigma \mid sy = ss'ds'' \in L\}$
- Suche ein $d \in D$, so dass $r = r'dr''$ und $|r'|$ minimal ist.
- Ersetze $tr'dr''$ durch $s'dr''$. $\frac{str = str'dr''}{ss'dr''}$
- gib Fehlermeldung aus und setze Analyse fort
- Vorteile:
  - nahe bei Minimalkorrektur, einfache Fehlermeldung
  - fast vollautomatisch erzeugbar
  - terminiert, da Eingabe um d verkürzt.
- Nachteile:
  - Korrektur nicht zwangsläufig korrekt
  - Schwierigkeiten bei Listenkonstruktionen (Anweisungs-, Bezeichner-, Vereinbarungslisten, usw.)
  - bei rekursivem Abstieg Vorbereitung notwendig
  - bei LR-Analyse Adaption des Generators notwendig

ALGOL 60 - Systematische Korrektur (Röhrich)

```
BEGIN
  INTEGER ARRAY A,B(1...5 1...10);
                          ^
***Error in front of 1 <,> inserted
  INTEGER I,J,K,L;
  UP:I+J>K+L*4 THEN GO L1 ELSE K IS 2;
     ^         ^       ^
***Error in front of <+> <:=> inserted
***Error in front of THEN <;> inserted
    THEN deleted
***Error in front of <L1> TO inserted
    ELSE <K> IS <2> deleted
```

ADA (adac) - Fehlerbehandlung

```
procedure Main is
  I;J : Integer;
  begin
  if I > J then then /* FEHLER 1: I in Ankermenge,
                        then gestrichen */
    I:=J
  else
    if I /= J then
      J = ; I /* FEHLER 2: = unerwartet,
                  ; in Ankermenge, = gestrichen */
    end if /* SYMPTOM: ; erwartet, end in Ankermenge,
              ; eingefügt */
  end if;
end Main;
```

ADA (adac) - Fehlerausgabe

```
error.ada...
4, 19: Error syntax error
4, 19: Information expected tokens:
IDENTIFIER CHAR_STRING NULL PRAGMA CASE RETURN
FOR BEGIN EXIT GOTO DELAY ABORT RAISE REQUEUE IF
WHILE LOOP DECLARE ACCEPT SELECT <<
5, 3: Information     restart point
8, 7: Error           syntax error
8, 7: Information     expected tokens: . ; ( , : : =
8, 9: Information     restart point
8, 6: Error           syntax error
9, 6: Information     expected tokens: . ; ( , : : =
9, 6: Repair          token inserted : ;
```

Durchführung

1. Zeichne in jedem Zustand eine Situation aus, deren weitere Verfolgung zur Generierung der Fortsetzung führt. Die Fortsetzung muss terminieren. Ankermenge sind alle Symbole in dieser Fortsetzung.
2. Streiche in der Eingabe alle Zeichen bis zum ersten Zeichen in der Ankermenge.
3. Dieses Zeichen wird während der Verarbeitung der generierten Fortsetzung in einem Zustand q' akzeptiert und es gibt einen Übergang vom Fehlerzustand q nach q'
4. Setze die Zeichen ein, die den Automaten vom Zustand q nach q' bringen.

LL Durchführung

- Zeichne für jedes Nichtterminal eine Produktion aus, die in der Fortsetzungserzeugung vorhergesagt wird. Die Produktion darf nicht rekursiv sein!
- Bestimme für die laufenden und jede vorhergesagte Produktion die noch fehlenden Symbole. Diese bilden die Ankermenge. Die noch laufenden Produktionen sind aus dem Keller ersichtlich.
- Schwierigkeit bei rekursivem Abstieg: Keller nur nach Prozedurrückkehr zugänglich.
  - Abhilfe: Ankermenge bereits während der normalen Syntaktische Analyse aufbauen und in getrenntem Keller (Datenstruktur) ablegen oder als Argument bei Prozeduraufruf mitgeben.

LL Beispiel Automat

$$Z \rightarrow A,\ A \rightarrow FA',\ A' \rightarrow \varepsilon \mid +FA',\ F \rightarrow i \mid (A)$$

$q_0 : [Z \rightarrow \cdot A]$
$q_1 : [Z \rightarrow A\cdot]$
$q_2 : [Z \rightarrow \cdot FA']$
$q_3 : [A \rightarrow F \cdot A']$
$q_4 : [F \rightarrow \cdot i]$
$q_5 : [F \rightarrow \cdot(A)]$
$q_6 : [A \rightarrow FA'\cdot]$
$q_7 : [A' \rightarrow \cdot\varepsilon]$
$q_8 : [A' \rightarrow \cdot + FA']$
$q_9 : [F \rightarrow i\cdot]$
$q_{10} : [F \rightarrow (\cdot A)]$
$q_{11} : [A' \rightarrow + \cdot FA']$
$q_{12} : [F \rightarrow (A\cdot)]$
$q_{13} : [A' \rightarrow +F \cdot A']$
$q_{14} : [F \rightarrow (A)\cdot]$
$q_{15} : [A' \rightarrow +FA'\cdot]$

$*q_0 i \rightarrow q_1 q_2 i,\ q_0(\rightarrow q_1 q_2 (,$
$*q_1 \rightarrow \varepsilon,$
$*q_2 i \rightarrow q_3 q_4 i,\ q_2(\rightarrow q_3 q_5 (,$
$*q_3 \# \rightarrow q_6 q_7 \#,\ q_3) \rightarrow q_6 q_7),$
$q_3 \rightarrow q_6 q_8 +,\ *q_4 i \rightarrow q_9,$
$*q_5(\rightarrow q_{10},\ *q_6 \rightarrow \varepsilon,\ *q_7 \rightarrow \varepsilon,$
$*q_8 + \rightarrow q_1 1,\ *q_9 \rightarrow \varepsilon,$
$*q_{10} i \rightarrow q_{12} q_2 i,\ q_{10}(\rightarrow q_{12} q_2 (,$
$*q_{11} i \rightarrow q_{13} q_4 i,\ q_{11}(\rightarrow q_{13} q_5 (,$
$*q_{12}) \rightarrow q_{14},$
$*q_{13} \# \rightarrow q_{15} q_7 \#,$
$q_{13}) \rightarrow q_{15} q_7),\ q_{13} + \rightarrow q_{15} q_{15} +,$
$*q_{14} \rightarrow \varepsilon,\ *q_{15} \rightarrow \varepsilon$

$*$ zeichnet die vorherzusagenden Produktionen bzw. Zustandsübergänge aus

LL Beispiel $i + \#$

Parsen bis zum Fehler	Fortsetzung finden	Wiederaufsetzen	
$q_0 i + \#$	$q_1 q_6 q_{11}$ $D = \{i(\}$	$q_1 q_6 q_{11} \#$	
$q_1 q_2 i + \#$	$q_1 q_6 q_{13} q_4$	$q_1 q_6 q_{13} q_4 \#$	
$q_1 q_3 q_4 i + \#$	$q_1 q_6 q_{13} q_9$	$q_1 q_6 q_{13} q_9 \#$	i generiert mit
$q_1 q_3 q_9 + \#$	$q_1 q_6 q_{13}$ $D = \{i(\#)+\}$		$q_4 i \rightarrow q_9$
$q_1 q_3 + \#$	$q_1 q_6 q_{15} q_7$	$q_1 q_6 q_{13} \#$	weiter normal
$q_1 q_6 q_8 + \#$	$q_1 q_6 q_{15}$		
$q_1 q_6 q_{11} \#$	$q_1 q_6$		
	q_1		

Problem am Beispiel (1/2)

Grammatik:

1. $Z \rightarrow I \mid A$
2. $I \rightarrow$ if E then Z end
3. $A \rightarrow E := E$
4. $E \rightarrow id\ E_{rest}$
5. $E_{rest} \rightarrow \varepsilon \mid = id$

SLL-Analyse des fehlerhaften Satzes:
if a:=b then ... end
mit parserdefiniertem Fehler := landet in Situation
$[\mathrm{I} \rightarrow$ if $E \cdot$ then Z end; $\Omega]$

Beachte: $:= \in \mathrm{Folge}_1(E)$

Problem am Beispiel (2/2)

Grammatik:

1. $Z \rightarrow I \mid A$
2. $I \rightarrow$ if B then Z end
3. $A \rightarrow E := E$
4. redB $\rightarrow$ E
5. $E \rightarrow id\ E_{rest}$
6. $E_{rest} \rightarrow \varepsilon \mid = id$

Parsen von **if** a:=b then ... end
mit parserdefiniertem Fehler := landet in Situation
[I $\rightarrow$ if $E \cdot$ then Z end; Ω]

Beachte: $:= \notin \mathrm{Folge}_1(B)$

Fehlerbehandlung für LR-Parser

Beispiel für Grammatik:

$$E \rightarrow E{+}E \mid E{*}E \mid (E) \mid \mathbf{id}$$

Erzeuge 4 Aktionen zur Fehlerbehebung:

e1 **id** erwartet aber nicht gefunden.
Aktion: Shift 3; Nachricht „`Operand fehlt`".

e2 **)** gefunden ohne vorherige **(**.
Aktion: Überspringe Token; Nachricht „`) ohne Gegenstück`".

e3 Operator erwartet aber **id** oder **)** gefunden.
Aktion: Shift 4; Nachricht „`Operator fehlt`".

e4 # gefunden aber noch Klammern geöffnet.
Aktion: Nachricht „`) fehlt`".

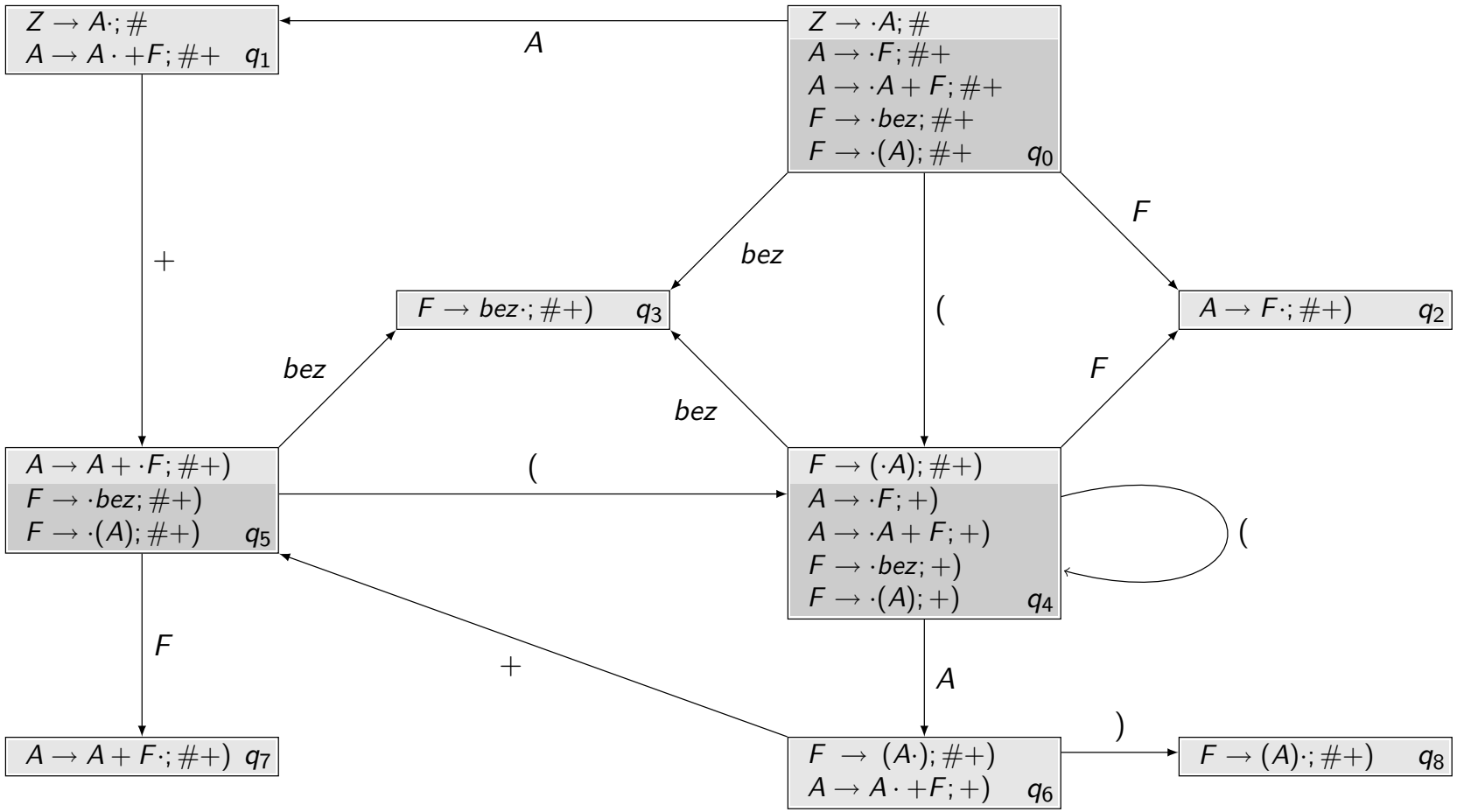
LR-Parsertabelle mit Fehlerroutinen

Zustand	ACTION						GOTO
	id	+	*	(	)	#	*E*
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc.	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

LR Durchführung

- Zeichne für jedes Nichtterminal eine nichtrekursive Produktion aus
- Zustände sind nicht Mengen, sondern geordnete Listen von Situationen. Die Situationen für ausgezeichnete Produktionen kommen jeweils vor allen anderen Situationen mit gleicher linker Seite.
- Erzeugung der Fortsetzung: Benutze in jedem Zustand jeweils die erste Situation und ihren Übergang.
- Die ausgezeichnete Situation gehört zur Basis des Zustandes.
- Verfahren terminiert wegen Nichtrekursivität der ausgezeichneten Produktionen
- Sonderbehandlung von Trennzeichen für Listen. Sie werden initial in entsprechende Ankermenge übernommen.

LR Beispiel-Automat



Übergänge

q	*bez*	(	)	+	#	A	F
0	-4	4	-	-	-	1	-3
1	-	-	-	5	*1		
2	-4	4	-	-	-	6	-3
4	-4	4	-	-	+4	6	-3
5	-4	4	-	-	+4		-2
6	-	-	-5	5	-		

Grau hinterlegt: Ausgezeichnete Übergänge

LR Beispiel $i +) \; i \; \#$

Parsen bis zum Fehler

$q_0 i +) i \#$
$q_0 q_1 +) i \#$
$q_0 q_1 q_5) i \#$

Fortsetzung finden

$q_0 q_1 q_5 \;\; D = \{ i (\}$
$q_0 q_1 \;\; D = \{ i (+ \# \}$

Wiederaufsetzen

$q_0 q_1 q_5 i \#$ wobei) gelöscht
weiter normal

Fehlerbehandlung in der semantischen Analyse

für jede Eigenschaft oder Wertetyp einen ausgezeichneten Wert *unbekannt* definieren

- Operationen mit unbekannt liefern wieder unbekannt
- Fehlermeldung nur, wenn unbekannt als Ergebnis, nicht als Operand erscheint.

Quellen fataler Fehler:

- fehlende Vereinbarung: Bezeichner erhält Typ unbekannt (oder Typ durch Typinferenz aus Anwendung erschließen)
- unzulässige Operation: Ergebnistyp ist unbekannt
- inkompatible Operanden oder sonstige Verletzung von Konsistenzbedingungen: Fehler melden, Ergebnistyp unbekannt, mit Analyse fortfahren

Kapitel 3: Syntaktische Analyse

Earley Parser

- Algorithmus zum parsen kontextfreier Grammatiken mit dynamischer Programmierung.
- Verarbeitet mehrdeutige Grammatiken und erzeugt alle möglichen Parsebäume eines Satzes.
- Laufzeit liegt im allgemeinen in $O(n^3)$; in $O(n^2)$ für eindeutige Grammatiken; in $O(n)$ für fast alle $\mathrm{LR}(k)$ Grammatiken.
- Häufiger Einsatz in der Computerlinguistik.

Algorithmus

- Erweitere die Grammatik um eine Regel $S' \rightarrow S$.
- Für jede Eingabeposition wird Menge S_i von Situationen $[X \rightarrow \alpha \cdot \beta, j]$ berechnet. j gibt die Eingabeposition an. Initiale Menge: $\{[S' \rightarrow \cdot S, 0]\}$.
- Für jede Eingabeposition i wende 3 Schritte iterativ bis zum Fixpunkt an:
  - **Prediction**: Für jede Situation $[X \rightarrow \alpha \cdot A\ \beta, j]$ in S_i füge eine neue Situation $[A \rightarrow \cdot\gamma, i]$ für jede Produktion der Grammatik mit A auf der linken Seite ein (transitive Hülle bilden).
  - **Completion**: Für jede Situation $[A \rightarrow \gamma\cdot, j]$ in S_i füge für jede Situation $[X \rightarrow \alpha \cdot A\ \beta, k]$ in S_j eine neue Situation $[X \rightarrow \alpha\ A \cdot \beta, k]$ in S_i ein.
  - **Scanning**: Sei **a** das nächste Token. Füge für jede Situation $[X \rightarrow \alpha \cdot \mathbf{a}\ \beta, j]$ in S_i eine Situation $[X \rightarrow \alpha\ \mathbf{a} \cdot \beta, j]$ in S_{i+1} ein.

Beispiel

Grammatik:

$$
\begin{aligned}
S &\rightarrow S \textbf{ und } S \mid S \textbf{ oder } S \\
S &\rightarrow \textbf{blau} \mid \textbf{gestreift} \mid \textbf{glatt} \mid \textbf{teuer}
\end{aligned}
$$

Eingabe: **blau und gestreift oder glatt und teuer**

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 0)

Situationen S_0:
$[S' \rightarrow \cdot S, 0]$

Iteration:
$[S \rightarrow \cdot S \textbf{ und } S, 0]$
$[S \rightarrow \cdot S \textbf{ oder } S, 0]$
$[S \rightarrow \cdot \textbf{blau}, 0]$
$[S \rightarrow \cdot \textbf{gestreift}, 0]$
$[S \rightarrow \cdot \textbf{glatt}, 0]$
$[S \rightarrow \cdot \textbf{teuer}, 0]$

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 1)

Situationen S_1:
$[S \rightarrow \textbf{blau}\cdot, 0]$

Iteration:
$[S' \rightarrow S\cdot, 0]$
$[S \rightarrow S \ \cdot \ \textbf{und}\ S, 0]$
$[S \rightarrow S \ \cdot \ \textbf{oder}\ S, 0]$

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 2)

Situationen S_2:
$[S \rightarrow S\ \textbf{und}\ \cdot\ S, 0]$

Iteration:
$[S \rightarrow \cdot S\ \textbf{und}\ S, 2]$
$[S \rightarrow \cdot S\ \textbf{oder}\ S, 2]$
$[S \rightarrow \cdot \textbf{blau}, 2]$
$[S \rightarrow \cdot \textbf{gestreift}, 2]$
$[S \rightarrow \cdot \textbf{glatt}, 2]$
$[S \rightarrow \cdot \textbf{teuer}, 2]$

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 3)

Situationen S_3:
$[S \rightarrow \textbf{gestreift}\cdot, 2]$

Iteration:
$[S \rightarrow S\ \textbf{und}\ S\cdot, 0]$
$[S' \rightarrow S\cdot, 0]$
$[S \rightarrow S\ \cdot\textbf{und}\ S, 0]$
$[S \rightarrow S\ \cdot\textbf{oder}\ S, 0]$
$[S \rightarrow S\ \cdot\textbf{und}\ S, 2]$
$[S \rightarrow S\ \cdot\textbf{oder}\ S, 2]$

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 4)

Situationen S_4:

$[S \rightarrow S$ **oder** $\cdot\, S, 0]$

$[S \rightarrow S$ **oder** $\cdot\, S, 2]$

Iteration:

$[S \rightarrow \cdot S$ **und** $S, 4]$

$[S \rightarrow \cdot S$ **oder** $S, 4]$

$[S \rightarrow \cdot$**blau**$, 4]$

$[S \rightarrow \cdot$**gestreift**$, 4]$

$[S \rightarrow \cdot$**glatt**$, 4]$

$[S \rightarrow \cdot$**teuer**$, 4]$

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 5)

Situationen S_5:
$[S \to \textbf{glatt}\cdot, 4]$

Iteration:

$[S \to S \textbf{ oder } S\cdot, 0]$
$[S \to S \textbf{ oder } S\cdot, 2]$
$[S \to S \ \cdot \textbf{und } S, 4]$
$[S \to S \ \cdot \textbf{oder } S, 4]$

$[S' \to S\cdot, 0]$
$[S \to S \ \cdot \textbf{und } S, 0]$
$[S \to S \ \cdot \textbf{oder } S, 0]$
$[S \to S \textbf{ und } S\cdot, 0]$
$[S \to S \ \cdot \textbf{und } S, 2]$
$[S \to S \ \cdot \textbf{oder } S, 2]$

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 6)

Situationen S_6:

$[S \rightarrow S\ \textbf{und}\ \cdot\ S, 4]$
$[S \rightarrow S\ \textbf{und}\ \cdot\ S, 0]$
$[S \rightarrow S\ \textbf{und}\ \cdot\ S, 2]$

Iteration:

$[S \rightarrow \cdot S\ \textbf{und}\ S, 6]$
$[S \rightarrow \cdot S\ \textbf{oder}\ S, 6]$
$[S \rightarrow \cdot \textbf{blau}, 6]$
$[S \rightarrow \cdot \textbf{gestreift}, 6]$
$[S \rightarrow \cdot \textbf{glatt}, 6]$
$[S \rightarrow \cdot \textbf{teuer}, 6]$

Beispiel

Eingabe: **blau und gestreift oder glatt und teuer** # (Pos. 7)

Situationen S_7:

$[S \rightarrow \textbf{teuer}\cdot, 6]$

Iteration:

$[S \rightarrow S \ \cdot \textbf{und}\ S, 6]$
$[S \rightarrow S \ \cdot \textbf{oder}\ S, 6]$
$[S \rightarrow S\ \textbf{und}\ S\cdot, 4]$
$[S \rightarrow S\ \textbf{und}\ S\cdot, 0]$
$[S \rightarrow S\ \textbf{und}\ S\cdot, 2]$

$[S \rightarrow S \ \cdot \textbf{und}\ S, 4]$
$[S \rightarrow S \ \cdot \textbf{oder}\ S, 4]$
$[S' \rightarrow S\cdot, 0]$
$[S \rightarrow S \ \cdot \textbf{und}\ S, 0]$
$[S \rightarrow S \ \cdot \textbf{oder}\ S, 0]$
$[S \rightarrow S \ \cdot \textbf{und}\ S, 2]$
$[S \rightarrow S \ \cdot \textbf{oder}\ S, 2]$