Kapitel 3

Syntaktische Analyse

Kapitel 3: Syntaktische Analyse

Syntaktische Analyse

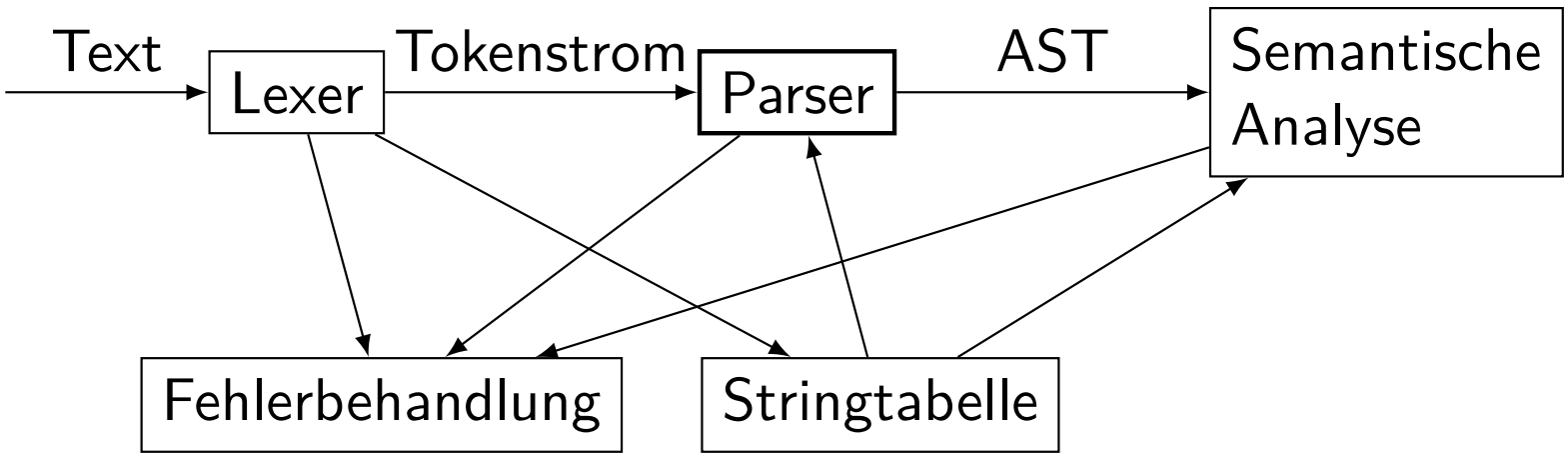
vorgegeben:

- 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:

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: $A \in N$.

Im Folgenden ist $V = T \cup N$.

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

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

$$\begin{aligned} T &= \{\mathbf{id}, +, *, (,)\} \\ N &= \{Z, E, T, F\} \\ P &= \{Z \to E, E \to T, T \to F, F \to \mathrm{id}, \\ &\qquad E \to E + T, T \to T * F, F \to (E)\} \end{aligned}$$

ist eine kontextfreie Grammatik. **id** steht für die Menge aller Bezeichner (engl. Identifier).

Linksableitung für a+(b+c):

E ⇒ E+T ⇒ T+T ⇒ F+T ⇒ a+T ⇒ a+(E) ⇒
a+(E+T) ⇒ a+(T+T) ⇒ a+(F+T) ⇒ a+(b+T) ⇒
a+(b+F) ⇒ a+(b+c)

Rechtsableitung für a+(b+c):

E ⇒ E+T ⇒ E+F ⇒ E+(E) ⇒ E+(E+T) ⇒ E+(E+F) ⇒
E+(E+c) ⇒ E+(T+c) ⇒ E+(F+c) ⇒ E+(b+c) ⇒
T+(b+c) ⇒ F+(b+c) ⇒ a+(b+c)

Reduzierte Grammatiken

Ein Nichtterminal A heißt erreichbar, wenn

$$Z \Rightarrow^* \mu A \chi \qquad \mu, \chi \in V^*$$

Ein Grammatik heißt reduziert, wenn alle Nichtterminale erreichbar sind. Beispiel für nicht-reduziert:

$$Z \to a \qquad A \to a|\epsilon$$

Entfernen aller unerreichbaren Nichtterminale einer Grammatik G ergibt reduzierte Grammatik G'. Da $L(G) = L(G')$ gehen wir im Folgenden davon aus, dass alle Grammatiken reduziert sind.

Schreibweise der Produktionen

in der Theorie: $A \to x|y|\ldots$, $A \in N$, x, $y \in V^*$

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)

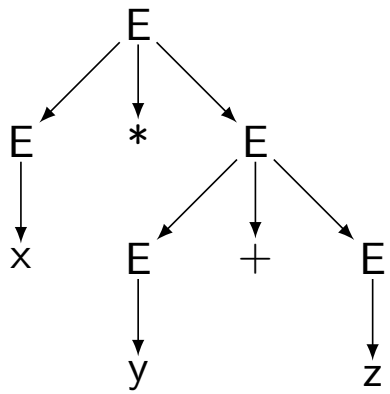
Forderungen:

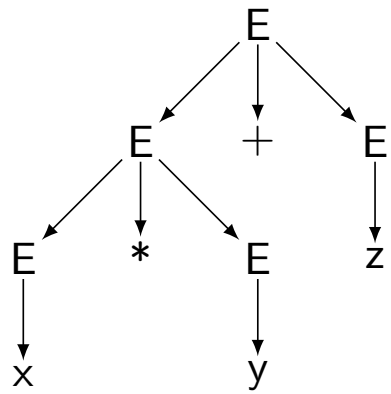
- deterministische Grammatik: zu einer Eingabe existiert höchstens ein Syntaxbaum
- Operatorprioritäten: Grammatik erzeugt Syntaxbaum gemäß Prioritäten

Gegenbeispiel:

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

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



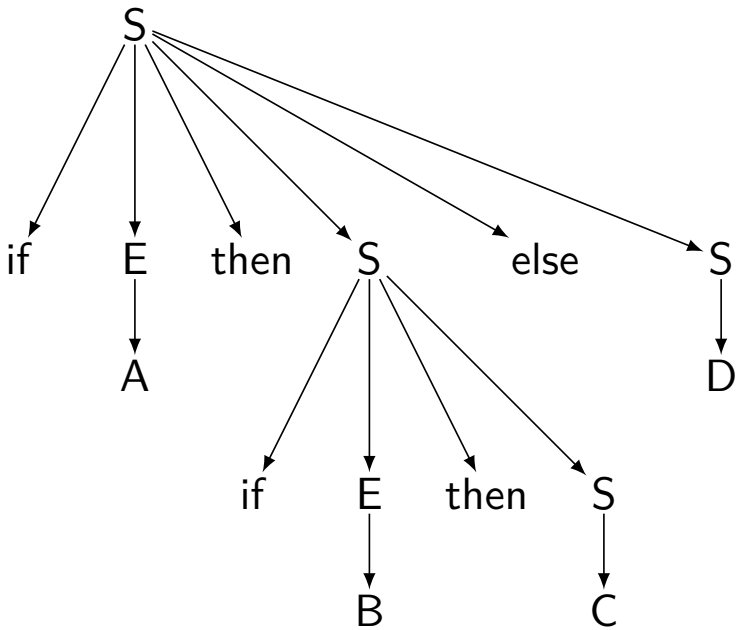


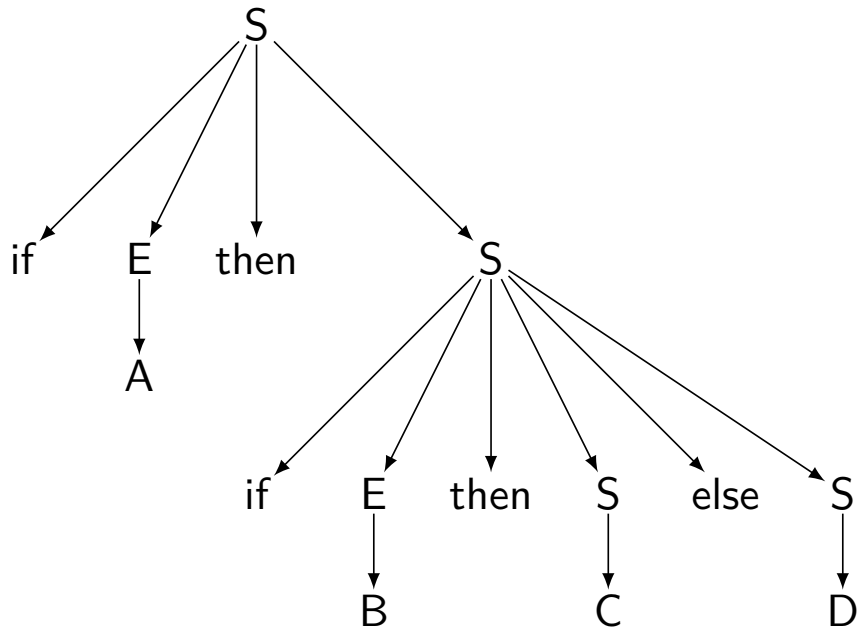
Grammar Engineering (2/3)

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.

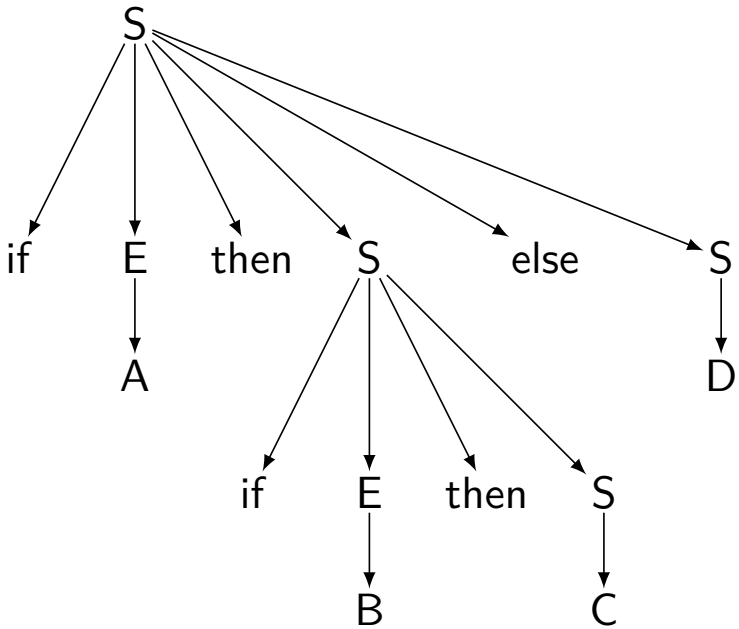


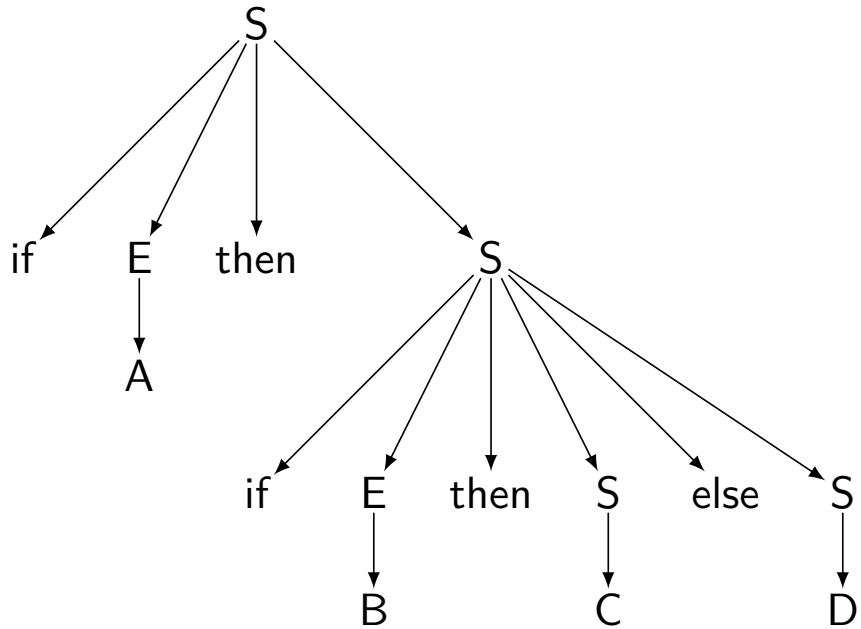


Grammar Engineering (3/3)

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:

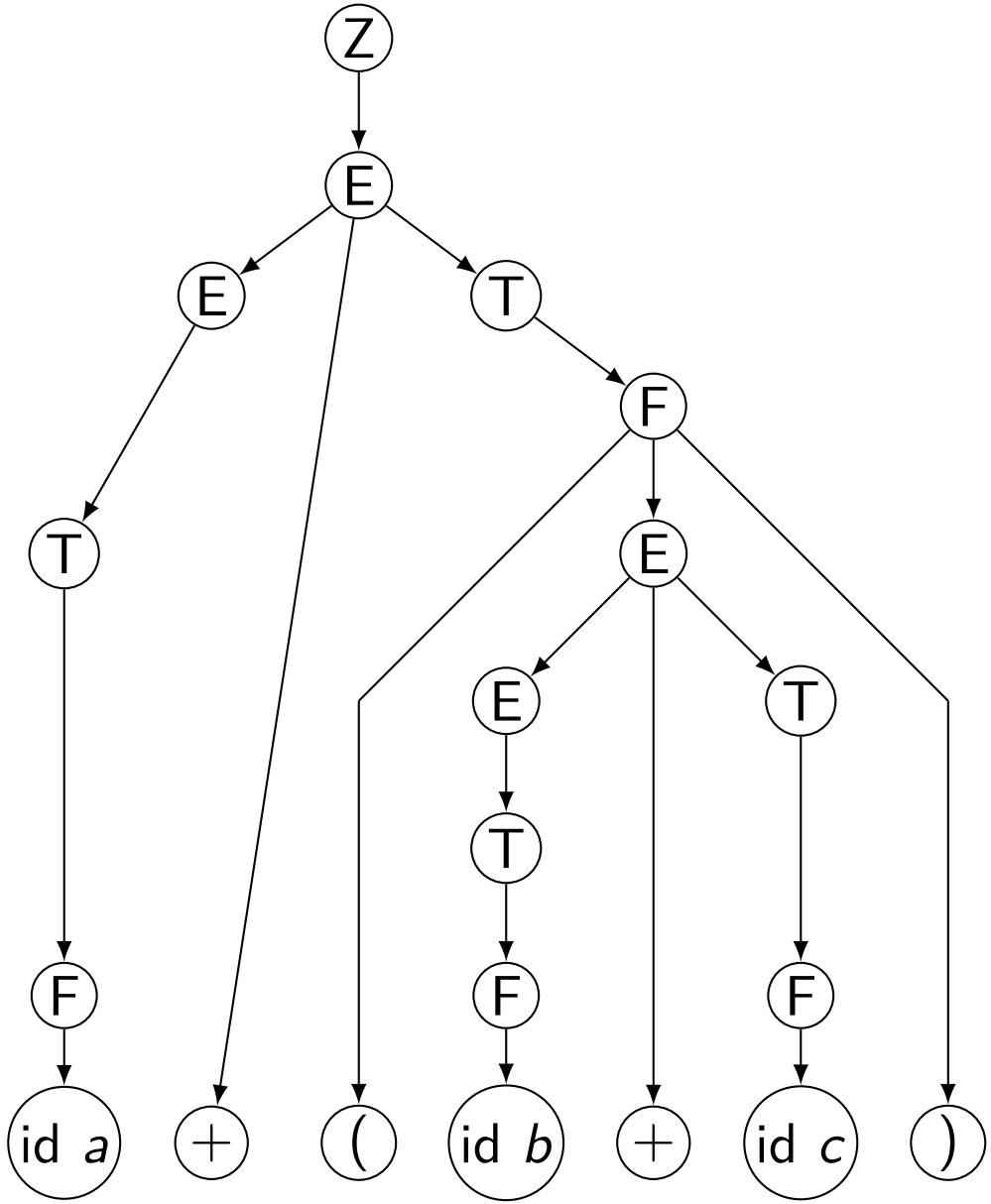
(0) $Z \to E$
(1) $E \to T$
(2) $T \to F$
(3) $F \to \mathrm{id}$
(4) $E \to E + T$
(5) $T \to T * F$
(6) $F \to (E)$

EBNF:

(0) Z ::= E.
(1) E ::= T ('+' T)∗.
(2) T ::= F ('∗' F)∗.
(3) F ::= id | '(' E ')'.

Ausdruck $a + (b + c)$

Konkreter Strukturbaum:



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 $A \to \varepsilon$ in alle rechten Seiten der Form $X \to \alpha A \beta$, 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

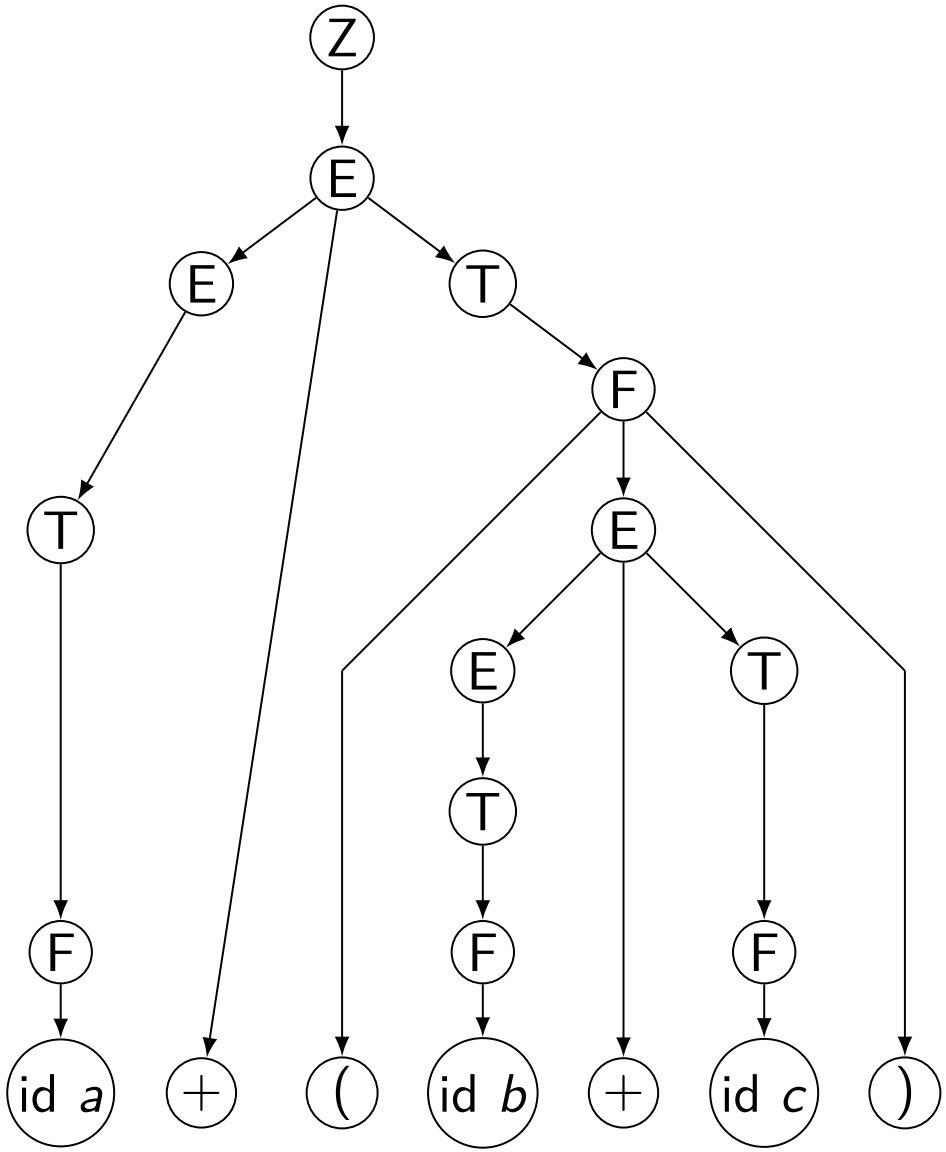
$S \rightarrow$ if E then S endif
$S \rightarrow$ if E then S else S endif

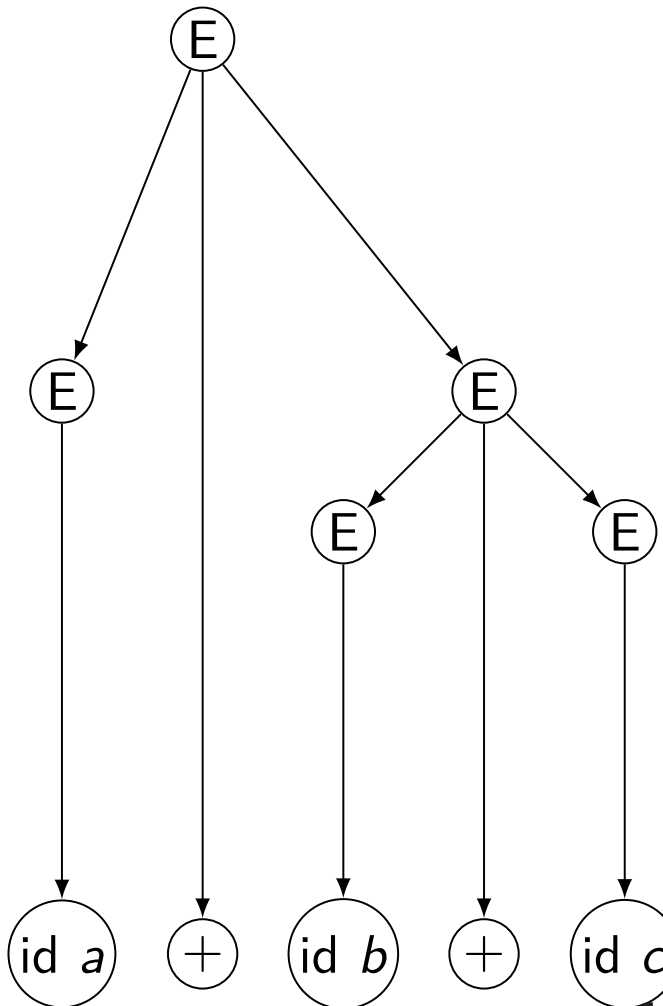
haben gemeinsamen Anfang if E then S.
Linksfaktorisierung ergibt:

$S \rightarrow$ if E then S X
$X \rightarrow$ endif | else S endif

Konkrete und abstrakte Syntax

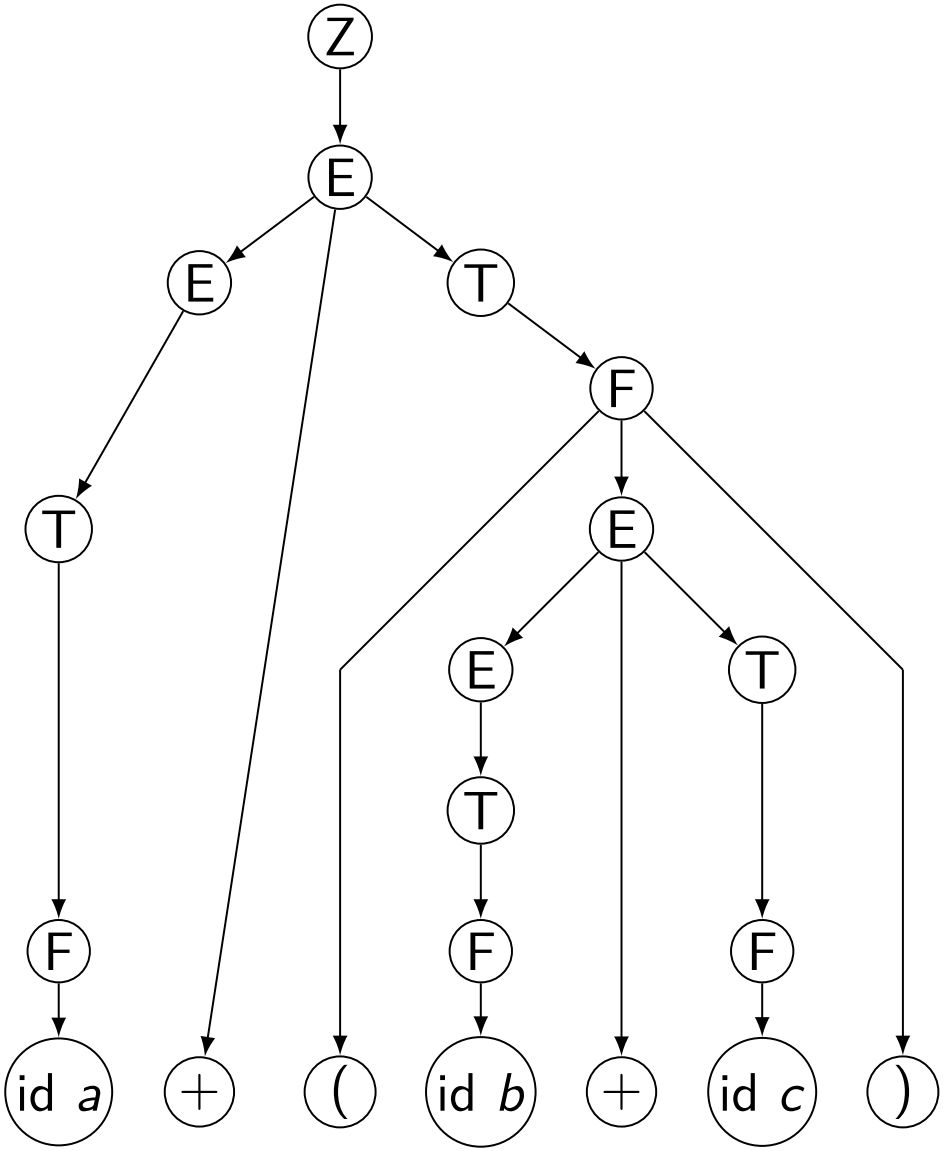
Prinzip der abstrakten Syntax: nur die für die Semantik wichtige Struktur behalten: $E \to E + E | E * E | \mathrm{id}$

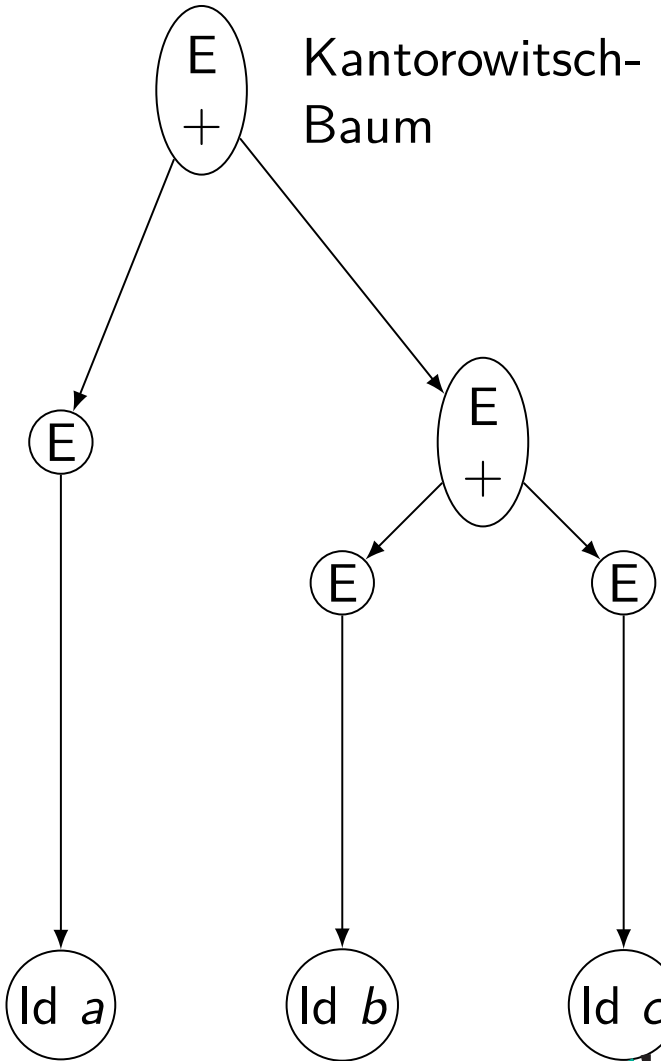




Konkrete und abstrakte Syntax

Prinzip der abstrakten Syntax: nur die für die Semantik wichtige Struktur behalten: $E \to E + E | E * E | \mathrm{id}$





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 $A \to B$ bzw. $A \to B \mid C \mid \ldots$
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

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

Abstrakte Syntax als abstrakte Algebra

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

Beispiele

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

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

Implementierung abstrakte Syntax

Objektorientiert:

- je 1 Klasse pro syntaktische Kategorie
- Alternativregeln

$$X = X1 \mid X2 \mid \ldots$$

werden zu Unterklassen:

```
class X { /* ... */ }
class X1 extends X { /* ... */ }
class X2 extends X { /* ... */ }
```

- Baumaufbauregeln

$$X :: Y1 \;\; Y2$$

werden zu Konstruktorfunktionen:

```
class X {
public X(Y1 y1, Y2 y2) { /* ... */ }
}
```

Sonderfälle in abstrakter Syntax

- Bezeichner:
  - $E \to \mathrm{id}$ 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

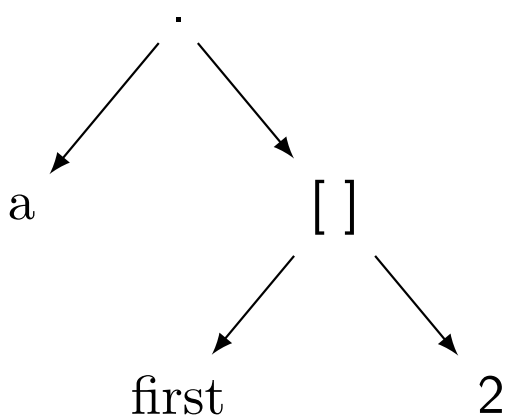
Typunabhängiges Parsen

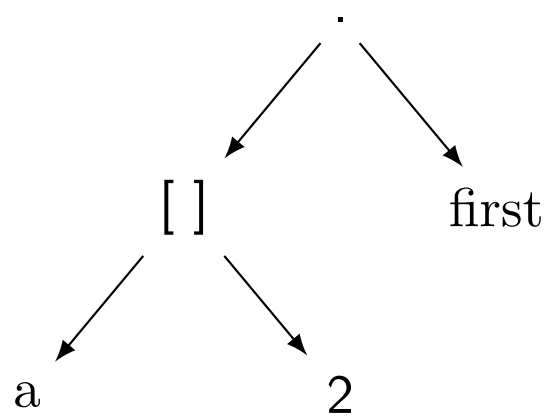
Typunabhängiges Parsen

- 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

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 Postfixform

Ausgaberoutinen:

addop	gib '+' aus
mulop	gib '*' aus
bezeichner	gib id aus
merke	merke id
bez_aus	gib gemerkte id aus, falls vorhanden

Postfixform, d.h. abstrakter Syntaxbaum als Rechtsableitung:

1. $E ::= T\ (\ '+'\ \ T\ \%\text{addop}\ )^*$
2. $T ::= F\ (\ '*'\ \ F\ \%\text{mulop}\ )^*$
3. $F ::= \text{id}\ \&\text{bezeichner} \mid '('\ E\ ')'$

Beispiel Postfixform

1. $E ::= T\ (\ '+'\ \ T\ \%\text{addop}\ )^*$
2. $T ::= F\ (\ '*'\ \ F\ \%\text{mulop}\ )^*$
3. $F ::= \text{id}\ \&\text{bezeichner} \mid '('\ E\ ')'$

Ableitung für x∗y+z	Ausgabe
E	
$\Rightarrow$ 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+

Kellerautomaten

- Kellerautomat $A = (T, Q, R, q_0, F, S, s_0)$
  - T Eingabealphabet (Tokens)
  - Q Zustandsmenge
  - R Menge von Regeln $sqx \to s'q'x'$, $s, s' \in S^*$, $q, q' \in Q$, $x, x', x'' \in T^*$, $x = x''x'$
  - 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 $sqx \to s'q'x'$ 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

Kapitel 3: Syntaktische Analyse

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 Token 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 Token t, $x = x'tx''$, so gibt es einen Satz $y \in L(G)$ mit $y = x'y'$.
  - Alternative: Erkennen des Fehlers einige Token später, keine syntaktische Fehlerlokalisierung möglich.

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)
$tqt \to q,\ t \in T$	$qt \to tq,\ t \in T$
$Xq \to x_n \dots x_1 q,$	$x_1 \dots x_n q \to Xq,$
$X \to x_1 \dots x_n \in P$	$X \to x_1 \dots x_n \in P$

- mache Kellerautomat deterministisch durch Hinzunahme Rechtskontext, also Vorhersage $Xqx' \to x_n \dots x_1 qx'$ bzw. Reduktion $x_1 \dots x_n qx' \to Xqx'$
  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):	Shift:
$tqt \to q,\ t \in T$	$qt \to tq,\ t \in T$
Vorhersage (produce):	Reduktion (reduce):
$Xq \to x_n \dots x_1 q,$	$x_1 \dots x_n q \to Xq,$
$X \to x_1 \dots x_n \in P$	$X \to x_1 \dots x_n \in P$

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 (ε-free 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

Berechnung von Anf_1 und Folge_1

Anf_1

1. Wenn $\gamma \in T^*$, so $1 : \gamma \in \mathrm{Anf}_1(\gamma)$
2. Wenn $X \to \alpha \in P$, so $\mathrm{Anf}_1(\alpha) \subseteq \mathrm{Anf}_1(X)$
3. $\mathrm{Anf}_1(\alpha) \setminus \{\#\} \subseteq \mathrm{Anf}_1(\alpha\beta)$
4. Wenn $\alpha \Rightarrow^* \varepsilon$, so $\mathrm{Anf}_1(\beta) \subseteq \mathrm{Anf}_1(\alpha\beta)$

Folge_1

1. Wenn $X \to \alpha Y \beta \in P$, so $\mathrm{Anf}_1(\beta) \subseteq \mathrm{Folge}_1(Y)$
2. Wenn $X \to \alpha \in P$, so $\mathrm{Folge}_1(X) \subseteq \mathrm{Folge}_1(\alpha)$
3. $\mathrm{Folge}_1(\alpha\beta) \subseteq \mathrm{Folge}_1(\beta)$
4. Wenn $\beta \Rightarrow^* \varepsilon$, so $\mathrm{Folge}_1(\alpha\beta) \subseteq \mathrm{Folge}_1(\alpha)$

Diese Regeln werden wiederholt angewendet, bis Anf_1 bzw. Folge_1 stabil sind (Fixpunktiteration).

Übung: Geben Sie die allgemeinen Formeln für Anf_k und Folge_k an. Verwenden Sie dazu Anf_{k-1} bzw. Folge_{k-1}.

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 Token kann unter Berücksichtigung des Kellerinhalts die nächste anzuwendende Produktion eindeutig vorhergesagt werden.

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

Beispiele von LL-Grammatiken, Anf- und Folge-Mengen

Grammatik:

E $\rightarrow$ TE'
E' $\rightarrow$ ε | +TE'
T $\rightarrow$ FT'
T' $\rightarrow$ ε | $*$FT'
F $\rightarrow$ id | (E)

$\text{Anf}_1(E) = \text{Anf}_1(T) = \text{Anf}_1(F) = \{\textbf{id}, \textbf{(}\}$
$\text{Anf}_1(E') = \{\varepsilon, \textbf{+}\} \quad \text{Anf}_1(T') = \{\varepsilon, \textbf{*}\}$
$\text{Folge}_1(E) = \text{Folge}_1(E') = \{\varepsilon, \textbf{)}\}$
$\text{Folge}_1(T) = \text{Folge}_1(T') = \{\varepsilon, \textbf{)}, \textbf{+}\}$
$\text{Folge}_1(F) = \{\varepsilon, \textbf{)}, \textbf{+}, \textbf{*}\}$

Beispiele von LL-Grammatiken, LL-Definition

Grammatik:

E $\rightarrow$ TE'

E' $\rightarrow$ ε | $+$TE'

T $\rightarrow$ FT'

T' $\rightarrow$ ε | $*$FT'

F $\rightarrow$ id | (E)

Grammatik ist $\mathrm{LL}(1)$ nach Definition. Betrachte etwa
$E \Rightarrow TE' \Rightarrow_L^* \mu + TE'\chi \Rightarrow^* \mu\gamma$ (hier ist $\nu = +TE'$)
$E \Rightarrow TE' \Rightarrow_L^* \mu\varepsilon\chi' \Rightarrow^* \mu\gamma'$ (hier ist $\omega = \varepsilon$)
$k : \gamma = k : \gamma'$ bedeutet: beide fangen mit $+$ an. Dann kommt die 2. Möglichkeit nicht in Frage, da $+ \notin \mathrm{Folge}_1(E)$. Deshalb ist $\nu = \omega$. Für die anderen Produktionen wird analog argumentiert.

Bemerkung:

In der Praxis verwendet man stets das SLL-Kriterium (s.u.)

Beispiele von LL-Grammatiken, LL(2)

Grammatik

Z $\rightarrow$ aAab | bAbb A $\rightarrow$ ε | a

$\mathrm{Anf}_1(A) = \{\mathbf{a}\}$, $\mathrm{Folge}_1(A) = \{\mathbf{a}, \mathbf{b}\}$, also nicht LL(1) da $A \Rightarrow^* \varepsilon$.
Probiere LL(2):

$$\mathrm{Anf}_2(\varepsilon \mathrm{Folge}_2(A)) = \{\mathbf{ab}, \mathbf{bb}\}$$

$$\mathrm{Anf}_2(\mathbf{a}\mathrm{Folge}_2(A)) = \{\mathbf{aa}, \mathbf{ab}\}$$

also nicht SLL(2) (s.u.) aber LL(2):

$$Z \Rightarrow \mathbf{a}A\mathbf{ab} \Rightarrow \mathbf{aab}, Z \Rightarrow \mathbf{a}A\mathbf{ab} \Rightarrow \mathbf{aaab}$$

$2 : \mathbf{ab} \neq 2 : \mathbf{aab}$, deshalb LL(2) Kriterium nicht verletzt; analog

$$Z \Rightarrow \mathbf{b}A\mathbf{bb} \Rightarrow \mathbf{bbb}, Z \Rightarrow \mathbf{b}A\mathbf{bb} \Rightarrow \mathbf{babb}$$

Man beachte, dass die „Vergangenheit“ (gewählte Z-Produktion) bekannt sein muss. Falls diese nicht bekannt ist, kann trotz 2 Token Vorausschau nicht entschieden werden, welche A-Produktion verwendet werden muss!

Beispiele von LL-Grammatiken, weitere Beispiele

- 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

- E $\rightarrow$ T | E + T , T $\rightarrow$ F | T $*$ F, F $\rightarrow$ id | (E) ist linksrekursiv
- Ersetzung: Schritt 1 leer, Schritt 2: E $\rightarrow$ T | E + T durch E $\rightarrow$ T E', E' $\rightarrow$ ε | + T E' ersetzen; T $\rightarrow$ F | T $*$ F analog. Dies entspricht der EBNF E ::= T ('+' T)*, T ::= F ('$*$' F)*, F ::= id | '(' E ')'.
- Andere mögliche Ersetzung: E $\rightarrow$ T | T E', E' $\rightarrow$ + T | + T E'
- Vorsicht: Die Ersetzung durch E $\rightarrow$ T | T + E 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)

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

Für $k \geq 1$ heißt eine kfG $G = (T, N, P, Z)$ eine $\mathrm{SLL}(k)$-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 Token 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 Tokenreihen mit mindestens k Token ableitbar sind, trägt $\mathrm{Folge}_k(\mathrm{A})$ 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 Token

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

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:

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

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

Abbildung: Parsertabelle

Parsertabelle

Grammatik:

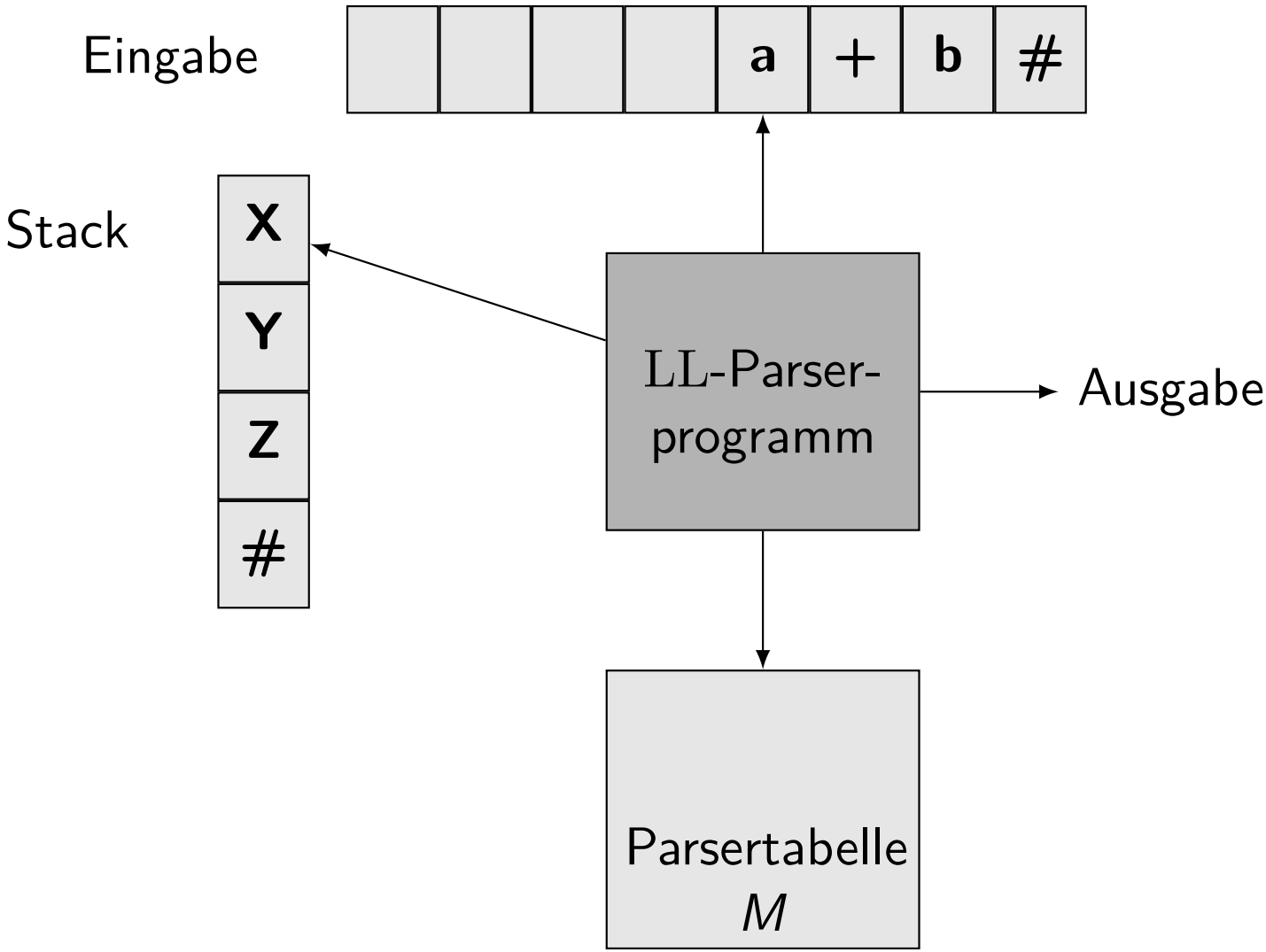
$$
\begin{aligned}
S &\rightarrow \mathbf{i}\ E\ \mathbf{t}\ S\ Z \mid \mathbf{a} \\
Z &\rightarrow \mathbf{e}\ S \mid \varepsilon \\
E &\rightarrow \mathbf{b}
\end{aligned}
$$

Parsertabelle:

Nichtterminal	Anf_1	Folge_1	**Eingabesymbol**					
			a	**b**	**e**	**i**	**t**	**#**
S	**i**, **a**	**e**, ε	$S \rightarrow \mathbf{a}$			$S \rightarrow \mathbf{i}E\mathbf{t}SS'$		
Z	**e**, ε	**e**, ε			$Z \rightarrow \mathbf{e}S$			$Z \rightarrow \varepsilon$
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 $E \to TE'$
	$FT'E'\#$	**id+id*id#**	Ausgabe von $T \to FT'$
	id$T'E'\#$	**id+id*id#**	Ausgabe von $F \to$ **id**
id	$T'E'\#$	**+id*id#**	Übereinstimmung mit **id**
id	$E'\#$	**+id*id#**	Ausgabe von $T' \to \varepsilon$
id	$+TE'\#$	**+id*id#**	Ausgabe von $E' \to +TE'$
id+	$TE'\#$	**id*id#**	Übereinstimmung mit **+**
id+	$FT'E'\#$	**id*id#**	Ausgabe von $T \to FT'$
id+	**id**$T'E'\#$	**id*id#**	Ausgabe von $F \to$ **id**
id+id	$T'E'\#$	***id#**	Übereinstimmung mit **id**
id+id	***$FT'E'\#$	***id#**	Ausgabe von $T' \to$ ***FT'
id+id*	$FT'E'\#$	**id#**	Übereinstimmung mit *****
id+id*	**id**$T'E'\#$	**id#**	Ausgabe von $F \to$ **id**
id+id*id	$T'E'\#$	**#**	Übereinstimmung mit **id**
id+id*id	$E'\#$	**#**	Ausgabe von $T' \to \varepsilon$
id+id*id	**#**	**#**	Ausgabe von $E' \to \varepsilon$

LL(1)-Parseralgorithmus

```
push('#'); push(Z); t = next_token();
while (t != '#') {
    if (stackEmpty()) { error("superfluous tokens found"); }
    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 : add_error(...);
   $\}$
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 = next_token() **else** add_error(...);

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

      wenn $Y_i \in N$

Parser aus Grammatik in EBNF

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

Beispielgrammatik

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

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

Parser für Beispielgrammatik (1/2)

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

Hinweis: Additionen/Multiplikationen werden im AST linksassoziativ interpretiert.

Parser für Beispielgrammatik (2/2)

```
AST F() {
    AST res = null;
    if (t == id) {
        res = new AST(t); t=next_token();
    }
    else if (t == '(') {
        t = next_token(); res = E();
        if (t == ')')
            t = next_token();
        else
            add_error(missing_closing_parenthesis, t.pos);
    }
    else
        add_error(invalid_token, t.pos);
    return res;
}
```

Praxis des rekursiven Abstiegs

- Einfügung von semantischen Aktionen:
  Semantische Aktion formal wie Produktion $A \rightarrow \varepsilon$ 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

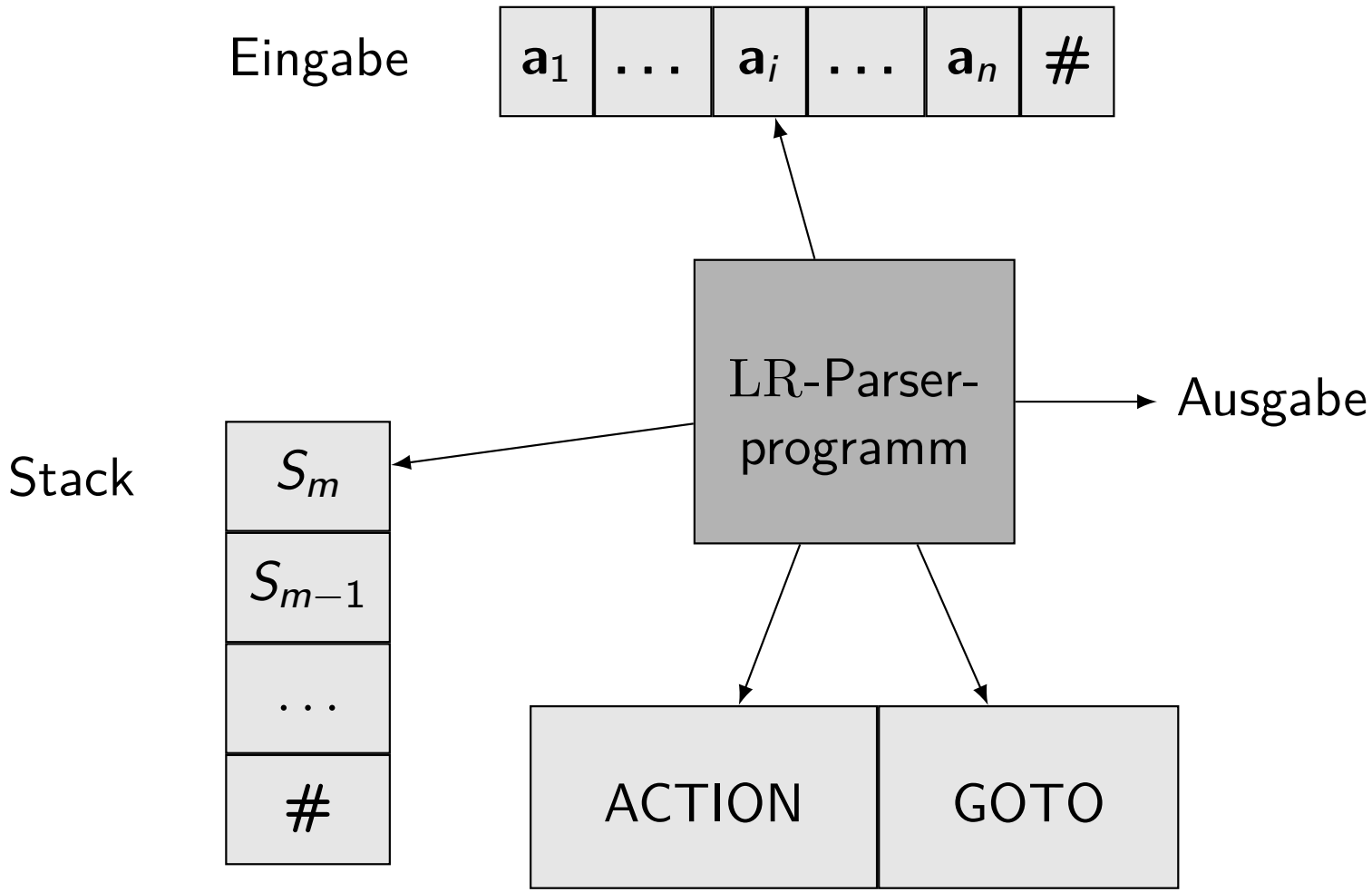
Für LL: (prädiktiv)	Für LR: (reduzierend)
Vergleich (compare):	Shift:
$tqt \to q,\ t \in T$	$qt \to tq,\ t \in T$
Vorhersage (produce):	Reduktion (reduce):
$Xq \to x_n \dots x_1 q,$	$x_1 \dots x_n q \to Xq,$
$X \to x_1 \dots x_n \in P$	$X \to x_1 \dots x_n \in P$

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



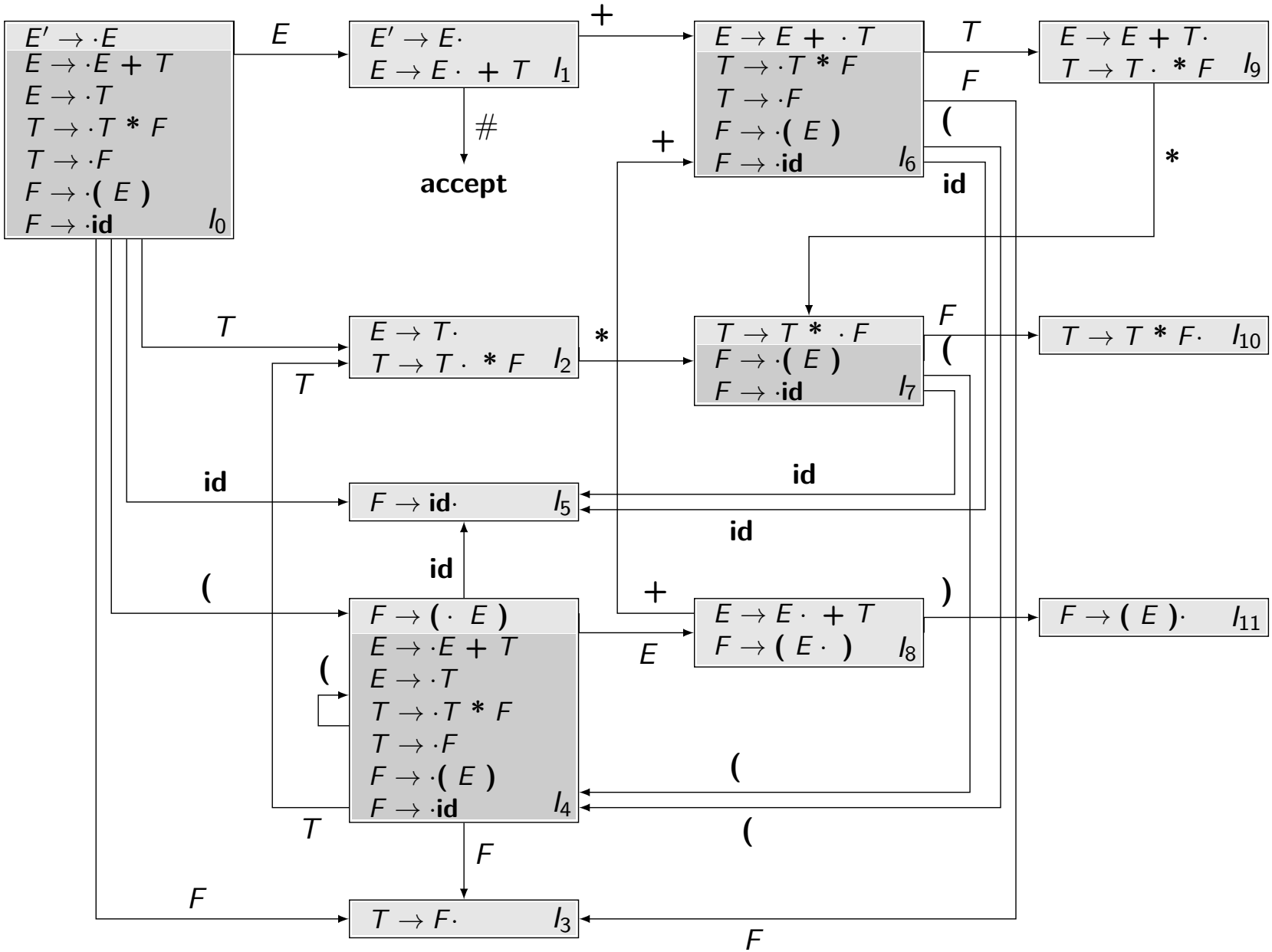
Beispiel: Grammatik G

1. $E' \to E$
2. $E \to E + T$
3. $E \to T$
4. $T \to T * F$
5. $T \to F$
6. $F \to (E)$
7. $F \to \mathbf{id}$

Verhalten eines Shift-Reduce-Parsers

Stack	**Eingabe**	**Aktion**
#	$\mathbf{id}_1$ * $\mathbf{id}_2$ #	Shift
# $\mathbf{id}_1$	* $\mathbf{id}_2$ #	Reduzieren durch $F \to \mathbf{id}$
# F	* $\mathbf{id}_2$ #	Reduzieren durch $T \to F$
# T	* $\mathbf{id}_2$ #	Shift
# T *	$\mathbf{id}_2$ #	Shift
# T * $\mathbf{id}_2$	#	Reduzieren durch $F \to \mathbf{id}$
# T * F	#	Reduzieren durch $T \to T$ * F
# T	#	Reduzieren durch $E \to T$
# 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 #**	Shift
0 5	# **id**	*** id #**	Reduzieren durch $F \rightarrow$ **id**
0 3	# F	*** id #**	Reduzieren durch $T \rightarrow F$
0 2	# T	*** id #**	Shift
0 2 7	# T *	**id #**	Shift
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

LR(1)-Parseralgorithmus

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

Verhalten eines LR-Parsers

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

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

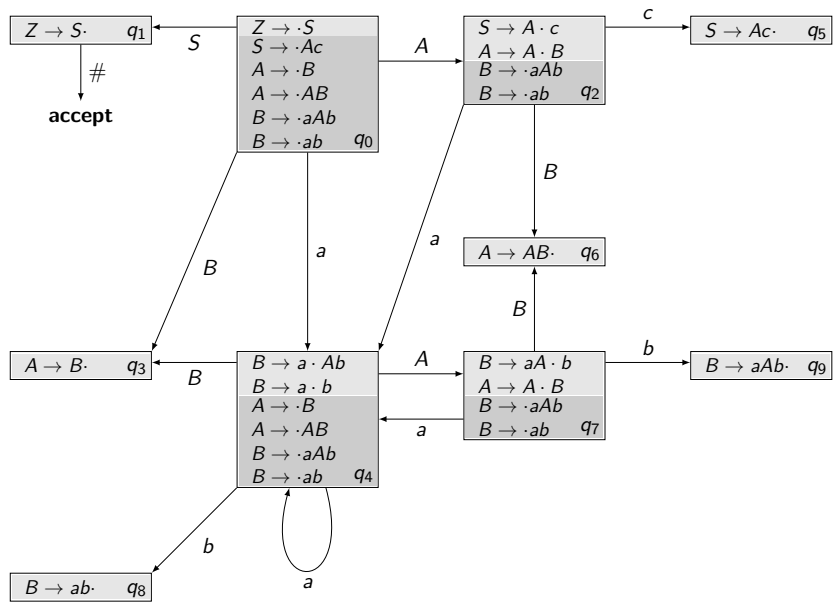
	a	b	c	#	A	B	S
0	s4	-	-	-	s2	s3	s1
1	-	-	-	#	-	-	-
2	s4	-	s5	-	-	s6	-
3	r4	r4	r4	-	r4	r4	r4
4	s4	s8	-	-	s7	s3	-
5	r2	r2	r2	-	r2	r2	r2
6	r3	r3	r3	-	r3	r3	r3
7	s4	s9	-	-	-	s6	-
8	r6	r6	r6	-	r6	r6	r6
9	r5	r5	r5	-	r5	r5	r5

-	Fehler
rX	Reduziere mit Regel X
sY	Shifte zu Zustand Y
#	Akzeptieren

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

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

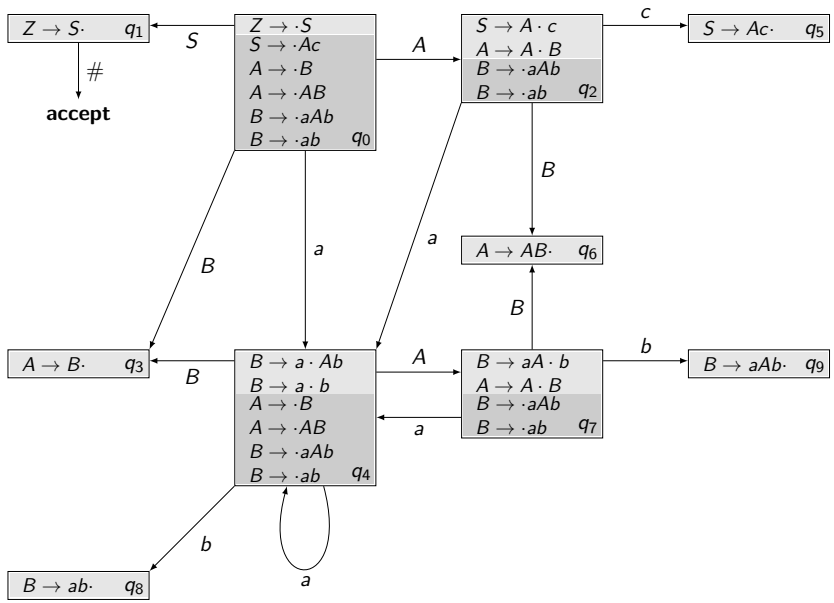
5: $B \to aAb$ 6: $B \to ab$



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

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

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



rot: aktuelle Symbole bzw. zu reduzierender Kellerteil

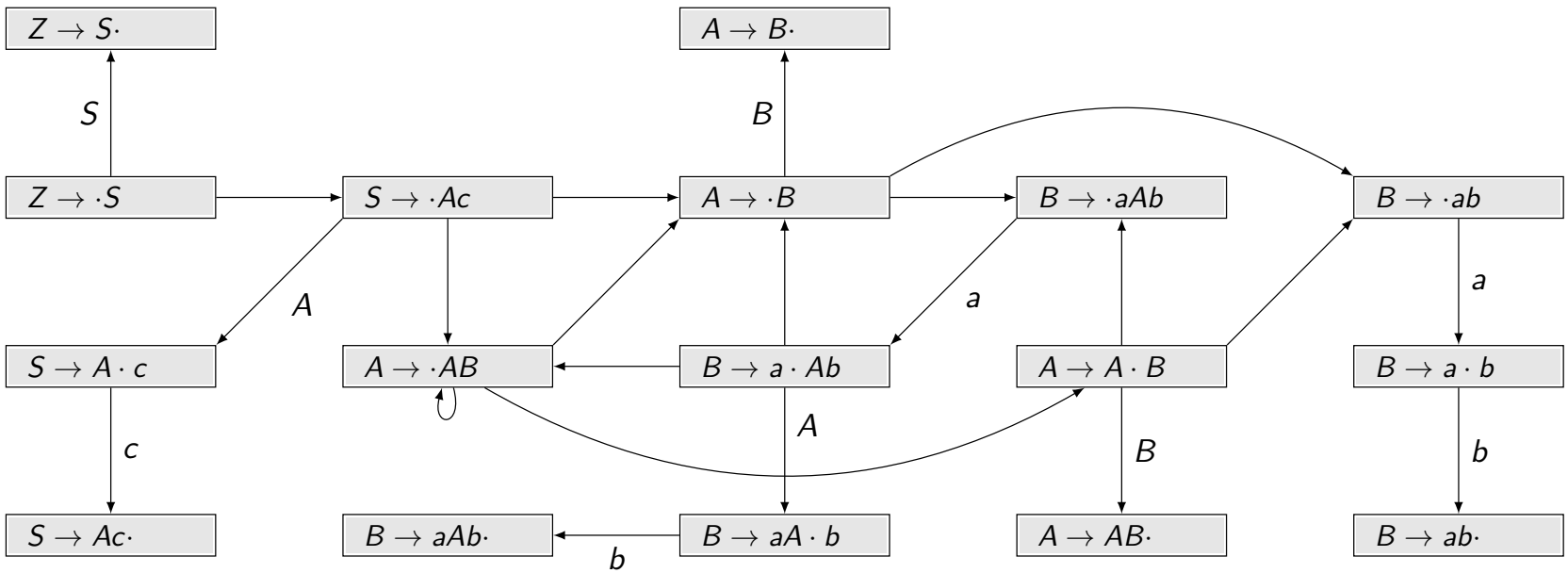
blau: Reduktionssymbol mit Shift im nächsten Schritt

Situationen (Items) und Automatenkonstruktion

- $\mathrm{LR}(0)$ Automat erkennt alle rechten Seiten von Produktionen als Strings von Terminalen und Nichtterminalen
- Position in rechter Seite wird durch Punkt angegeben. $X \to x' \cdot x''$ nennt man eine Situation oder ein $\mathrm{LR}(0)$-item
- Automat zunächst stark nichtdeterministsch (Menge von Kettenautomaten mit gemeinsamen Startzustand)
- deterministisch machen (Potenzmengenkonstruktion) bringt Zustände, die Mengen von items enthalten
- Spezialfälle von Items: $X \to \cdot\varepsilon$, $X \to \cdot\mathrm{AB}$, $X \to AB\cdot$ letztere heissen Reduktionsitems
  - LR-Analyse ruft quasi den Automaten für jede rechte Seite rekursiv auf
  - Rekursion wird nicht explizit, sondern durch Keller mit geretteten Zuständen realisiert
    zusätzlich Symbole von (teil)erkannten rechten Seiten im Keller

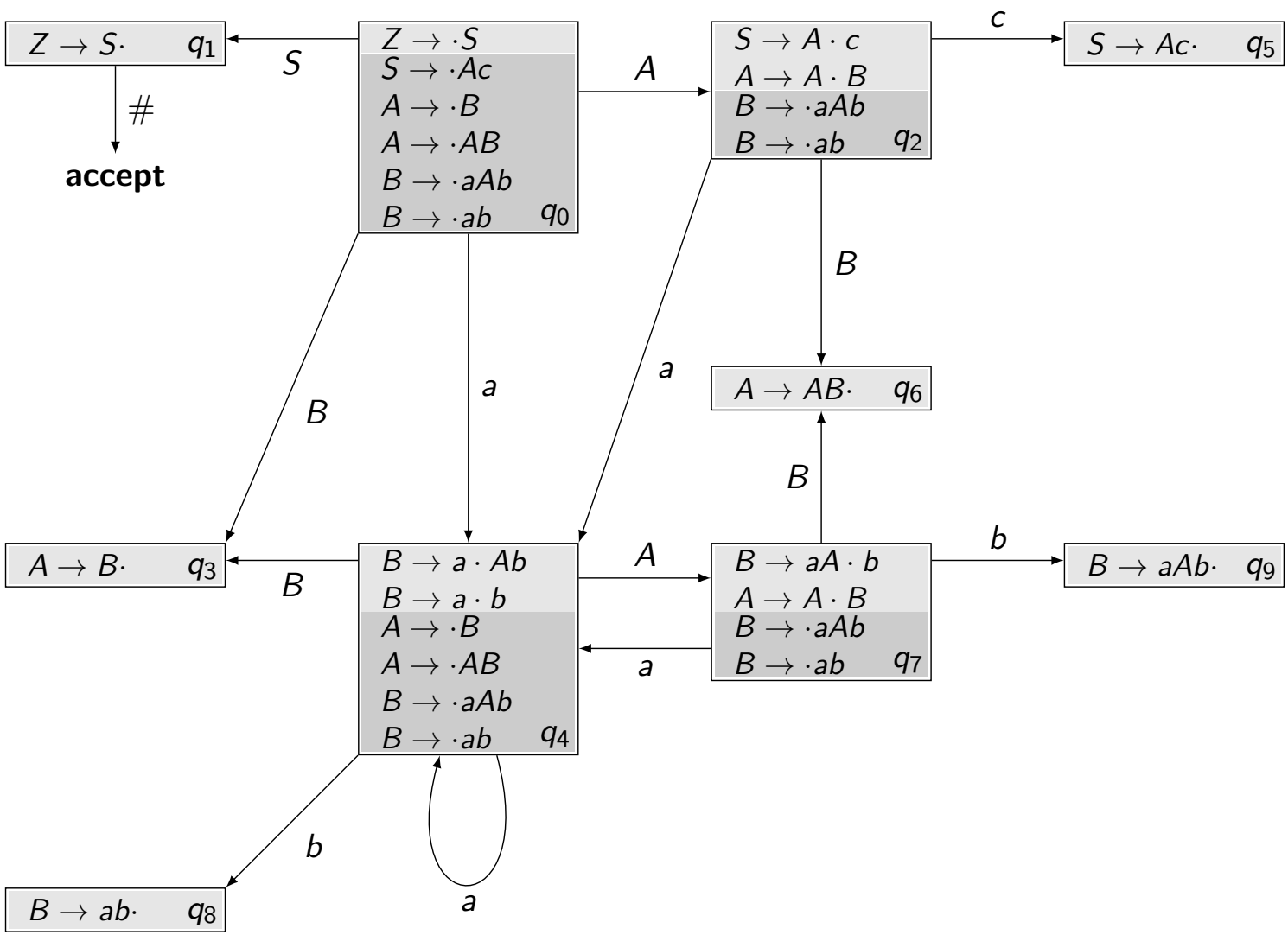
Nichtdeterministischer, endlicher Automat C

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



Charakteristischer Automat für $k = 0$

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



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

$$
\begin{array}{lcl}
E' & \to & E \\
E & \to & E + T \mid T \\
T & \to & T * F \mid F \\
F & \to & (E) \mid \mathbf{id}
\end{array}
$$

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

$$
\begin{array}{lcl}
E' & \to & \cdot E \\
E & \to & \cdot E + T \\
E & \to & \cdot T \\
T & \to & \cdot T * F \\
T & \to & \cdot F \\
F & \to & \cdot (\ E \ ) \\
F & \to & \cdot \mathbf{id}
\end{array}
$$

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:

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

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);
}
```

$\mathrm{SLR}(k)$-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 $v \in T^{\leq k-1}$:

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

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

SLR(1)-Übergangsfunktion am Beispiel

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

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

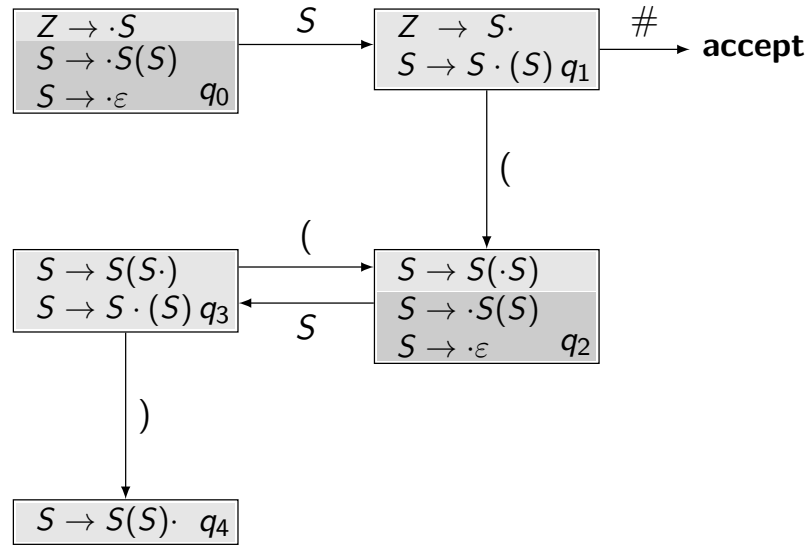
Es gilt $\mathrm{Folge}_1(E') = \{\#\}$. SLR(1)-Übergangsfunktion für I_1 ergibt:

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

Beispiel: LR(0)-Automat für Dyck-Sprache D_1

Keller	Eingabe	Bemerkungen
0	()()#	reduziere($S \to \varepsilon$)
0	()()#	shift(S)
01	(()()#	shift(()
012	)()#	reduziere($S \to \varepsilon$)
012	)()#	shift(S)
0123	)()#	shift())
01234	()#	reduziere($S \to S(S)$)
0	()#	shift(S)
01	(()#	shift(()
012	)#	reduziere($S \to \varepsilon$)
012	)#	shift(S)
0123	)#	shift())
01234	#	reduziere($S \to S(S)$)
0	#	shift(S)
01	#	HALT

1: $Z \to S$
2: $S \to S(S)$
3: $S \to \varepsilon$

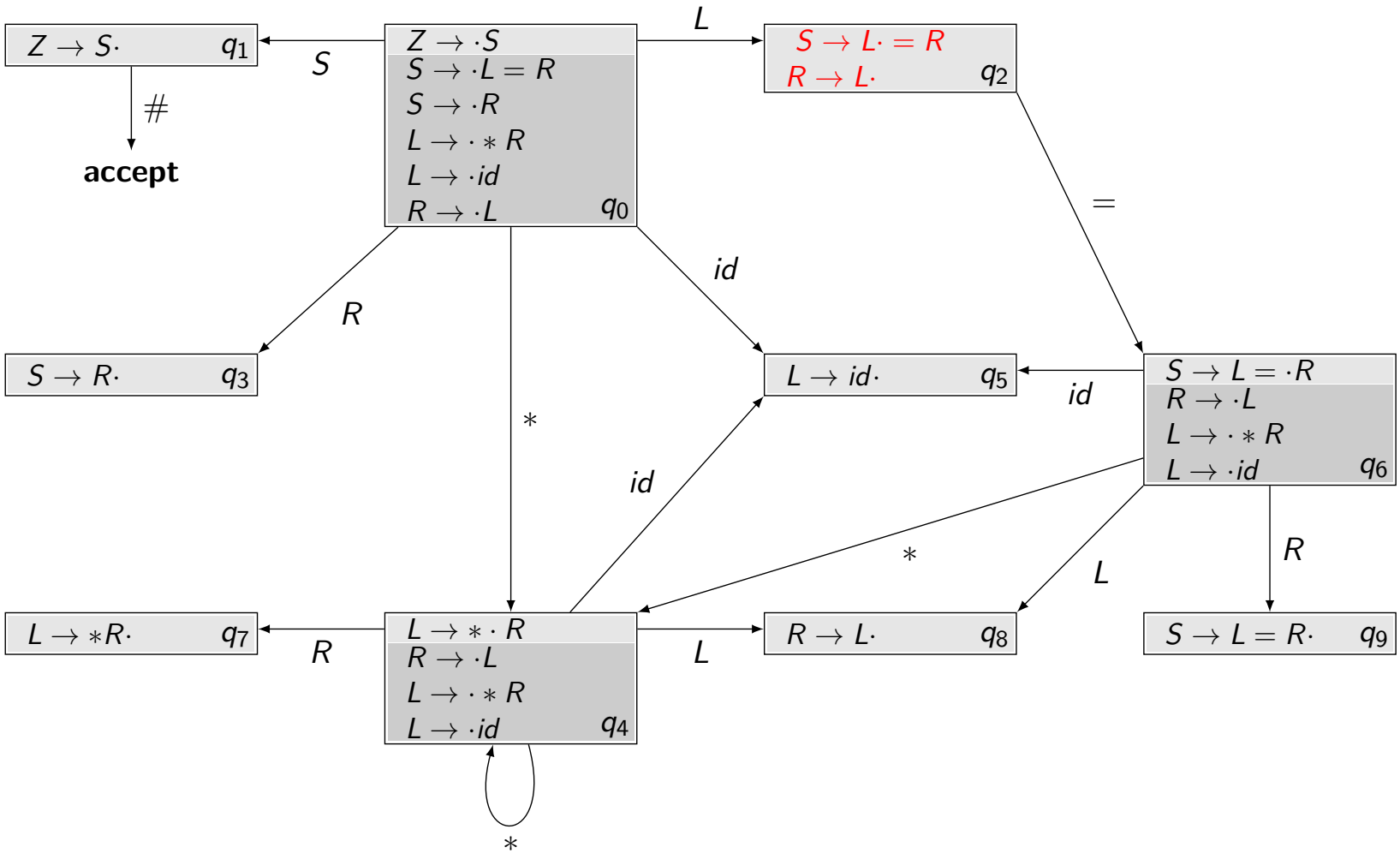


rot: aktuelle Symbole bzw. zu reduzierender Kellerteil
blau: Reduktionssymbol mit Shift im nächsten Schritt

Beispiel für Konflikt bei SLR(1)

Die bisher gezeigten Beispiele sind alle SLR(1). Die folgende Grammatik jedoch nicht:

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

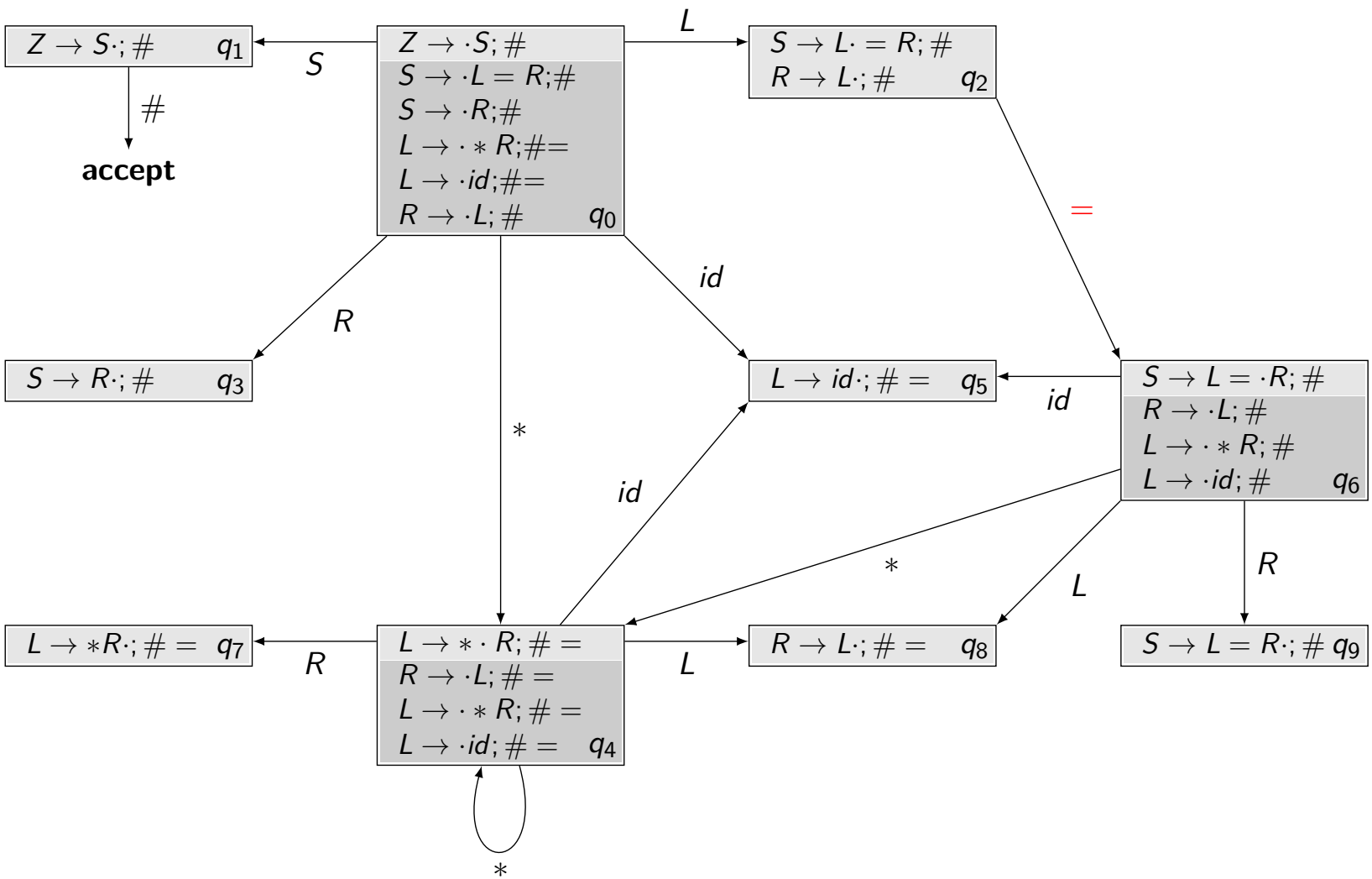


Grundidee LR-Grammatiken

- SLR Kriterium zur Konfliktauflösung zu schwach
- Grund: Folgemengen gelten für ganze Grammatik, berücksichtigen nicht spezifische Situation im Automat
- Idee: ergänze items um Situations-spezifische Folgemengen (Rechtskontexte) Schreibweise: $X \to A \cdot B;\ t$
- nur diese werden zur Konfliktauflösung herangezogen: falls irgendwann die Reduktion $X \to AB\cdot$ ansteht, wird diese nur gemacht, wenn das nächste Token $= t$ ist
- Rechtskontexte werden bei Closure-Berechnung bestimmt aus den tatsächlichen items
- Beispiel $X \to A \cdot BC; t$ bei z.B. $B \to DE$ führt zur Closure-Ergänzung $B \to \cdot DE,\ u$ mit $u \in \mathrm{Anf}_k(Ct)$
- Trennschärfer, da $\mathrm{Anf}_k(Ct)$ meist echt kleiner als $\mathrm{Folge}_k(B)$ bzw. $\mathrm{Folge}_k(X)$!
- Rechtskontexte bleiben bei Zustandsübergängen im Automaten unverändert (nur Punkt im item wird veschoben)

Beispiel für Rechtskontexte

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



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

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:

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

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 (ε-free 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 shift / reduzieren?
maximal verfügbare Information:
$\text{Reduktionsklasse} = \{(\text{Kellerinhalt}, \text{Eingaberest})\}$

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

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

In R_0 wird wegen R' also nur geshiftet mit r, falls nicht $A \Rightarrow^* Brs$, $B \Rightarrow^* \varepsilon$!

$R_i \cap R_j = \emptyset$ für $i \neq j$ bedeutet: Shiften bzw. Reduzieren mit Produktion p kann stets eindeutig entschieden werden.

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

$L(G) = \{cb, b\}$ und
$P = \{1\text{: } Z \rightarrow S,\ 2\text{: } S \rightarrow Ab,\ 3 : A \rightarrow c,\ 4\text{: } A \rightarrow \varepsilon\ \}$
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 Token b geshiftet wird.
Ohne Unterscheidung zwischen $\Rightarrow_{R'}$ und $\Rightarrow_R$: $(\varepsilon, b\#)$ gehört zu R_0, also wird geshiftet aber Konfiguration $bq\#$ kann keiner Reduktionsklasse zugeordnet werden, d.h. Sackgasse

k-Kellerklassen

Idee: Beschränke den betrachteten Eingaberest auf $k \geq 0$ Token:
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 $B \rightarrow b \in P$, $\tau \in \mathrm{Anf}_k(y\omega)$
- Endzustände sind belanglos für das weitere Vorgehen

Closure und GOTO für $\mathrm{LR}(k)$

Für Menge von Situationen I und (Nicht-)Terminal X sei

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

mit

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

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

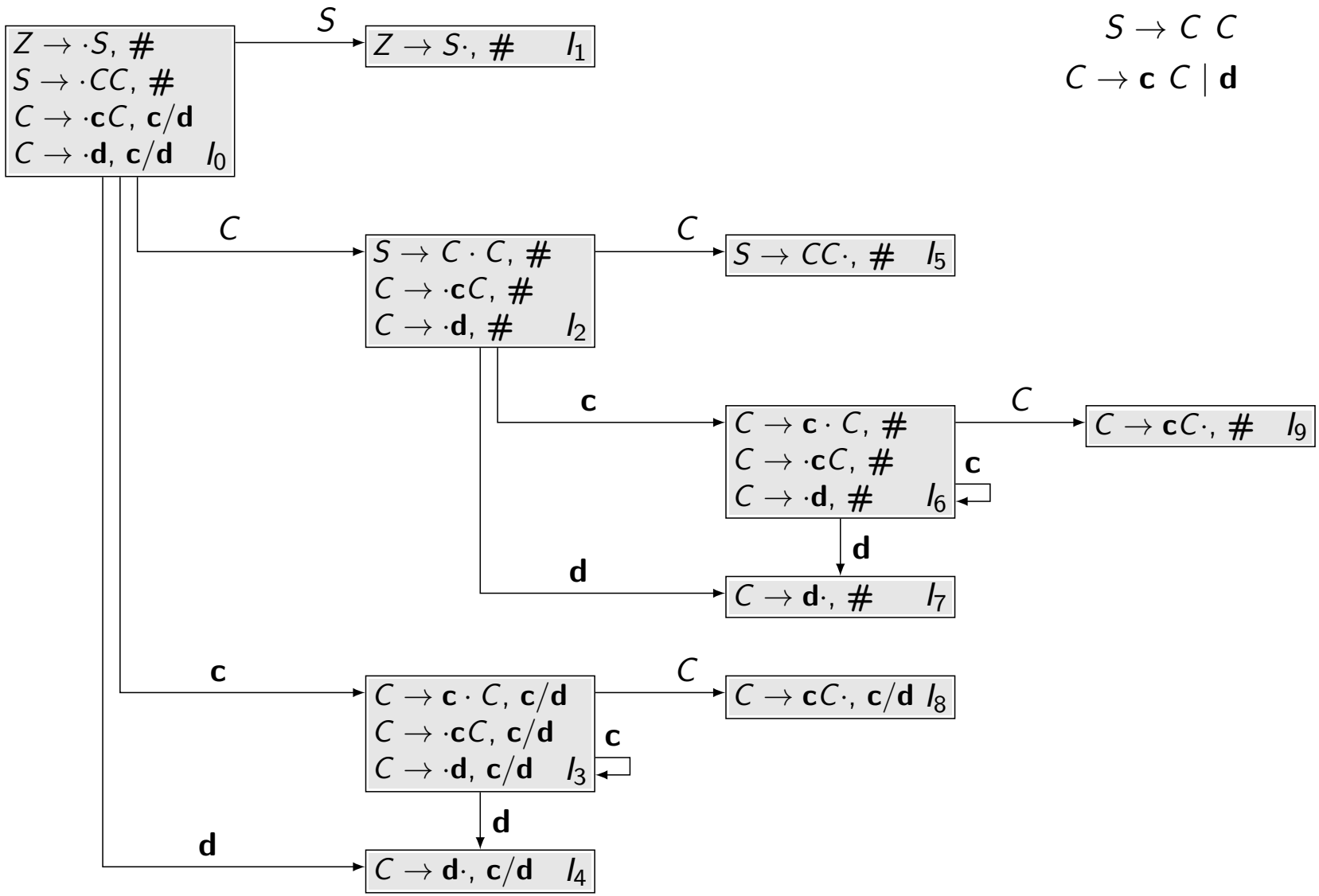
Für einen charakteristischen Automaten C, $q \in Q$, $t \in V$ und $v \in T^{\leq k-1}$:

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

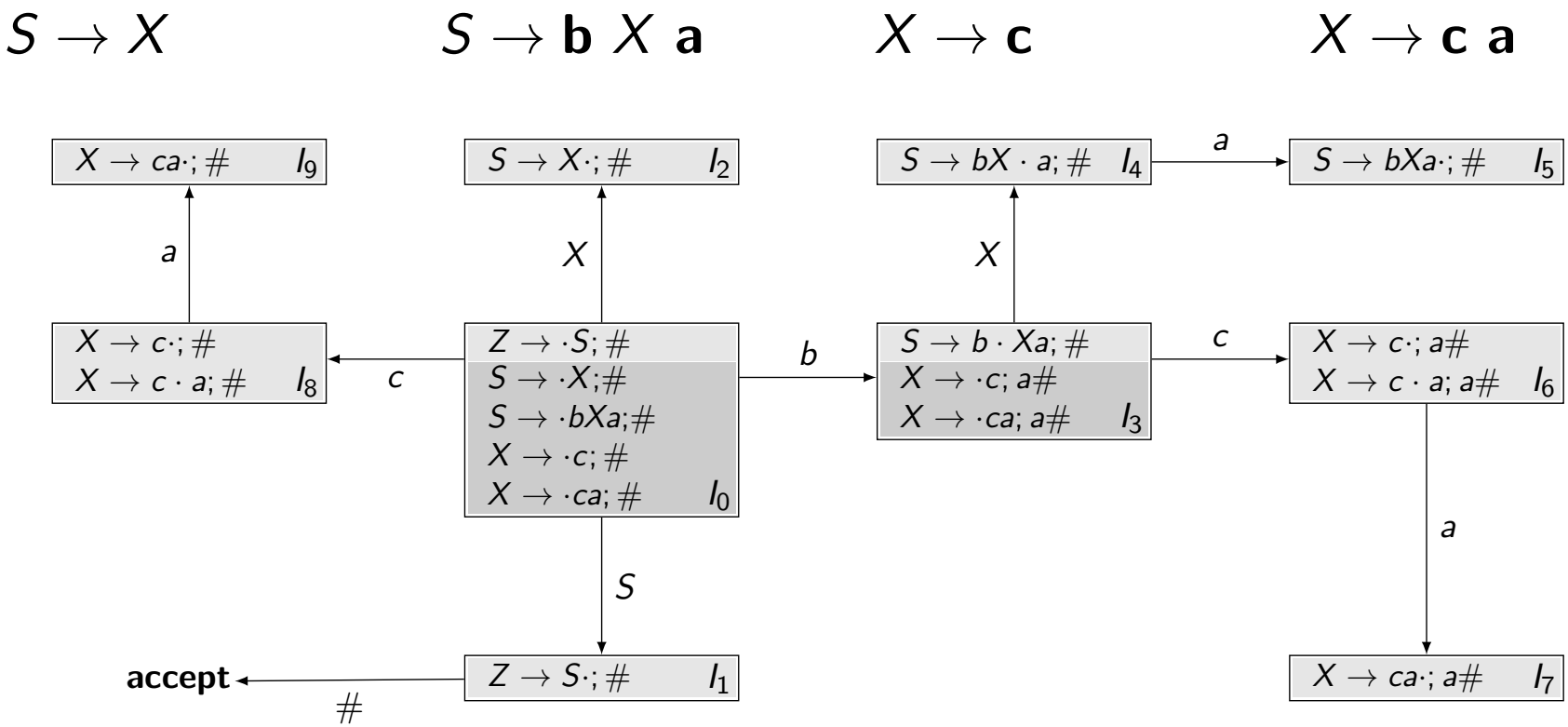
- Zustand q heißt inadäquat, wenn Übergangsfunktion $f(q, tv)$ für irgendein tv nicht eindeutig die Fälle eins, zwei und drei unterscheiden kann. Nur zwei Möglichkeiten existieren:
  - Shift-Reduktionskonflikt: shiften und reduzieren möglich
  - Reduktions-Reduktionskonflikt: Reduktion mit zwei verschiedenen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

GOTO-Graph für die Grammatik G'

$$Z \to S$$
$$S \to C\ C$$
$$C \to \mathbf{c}\ C \mid \mathbf{d}$$



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



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

1: $S \to X$, 2: $S \to bXa$, 3: $X \to c$, 4: $X \to ca$

Keller	Eingabe	Bemerkungen
0	bcaa#	shift(b)
03	caa#	shift(c)
036	aa#	shift(a)
0367	a#	reduziere($X \to ca$)
03	a#	shift(X)
034	a#	shift(a)
0345	#	reduziere($S \to bXa$)
0	#	shift(S)
01	#	Akzeptieren

rot: aktuelle Symbole bzw. zu reduzierender Kellerteil

grün: Vorschau

blau: Reduktionssymbol mit Shift im nächsten Schritt

	bc	c#	ca	a#	aa	#	S#	X#	Xa
0	s3	s8	s8	-	-	-	s1	s2	-
1	-	-	-	-	-	#	-	-	-
2	-	-	-	-	-	r1	-	-	-
3	-	-	s6	-	-	-	-	-	s4
4	-	-	-	s5	-	-	-	-	-
5	-	-	-	-	-	r2	-	-	-
6	-	-	-	r3	s7	-	-	-	-
7	-	-	-	r4	-	-	-	-	-
8	-	-	-	s9	-	r3	-	-	-
9	-	-	-	-	-	r4	-	-	-

-	Fehler
rX	Reduziere mit Regel X
sY	Shifte zu Zustand Y
#	Akzeptieren

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

$\mathrm{SLR}(k)$-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 $v \in T^{\leq k-1}$:

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

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

$\mathrm{LALR}(k)$-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 $v \in T^{\leq k-1}$:

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

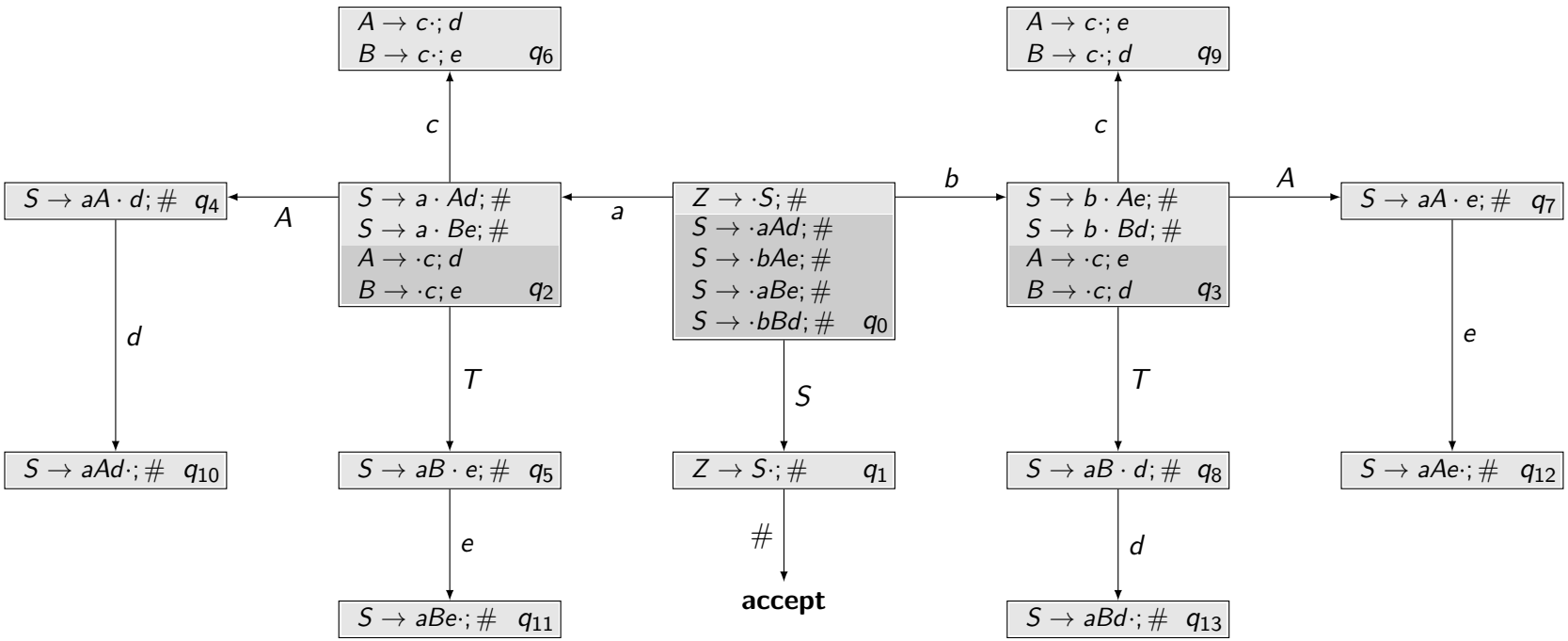
erhalten wir die LALR(k)-Übergangsfunktion.

- 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 LALR, aber LR

Die bisherigen Beispiele sind alle $\mathrm{LALR}(1)$, die folgende Grammatik jedoch nicht:

1: $S \to aAd$	2: $S \to bAe$	3: $A \to c$
4: $S \to aBe$	5: $S \to bBd$	6: $B \to c$



Verschmelzen von q_6 und q_9 würde Reduce-Konflikte erzeugen!

Behebung inadäquater Zustände

- Parsergenerator:
  - Shift-Reduzier-Konflikt: Shiften 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

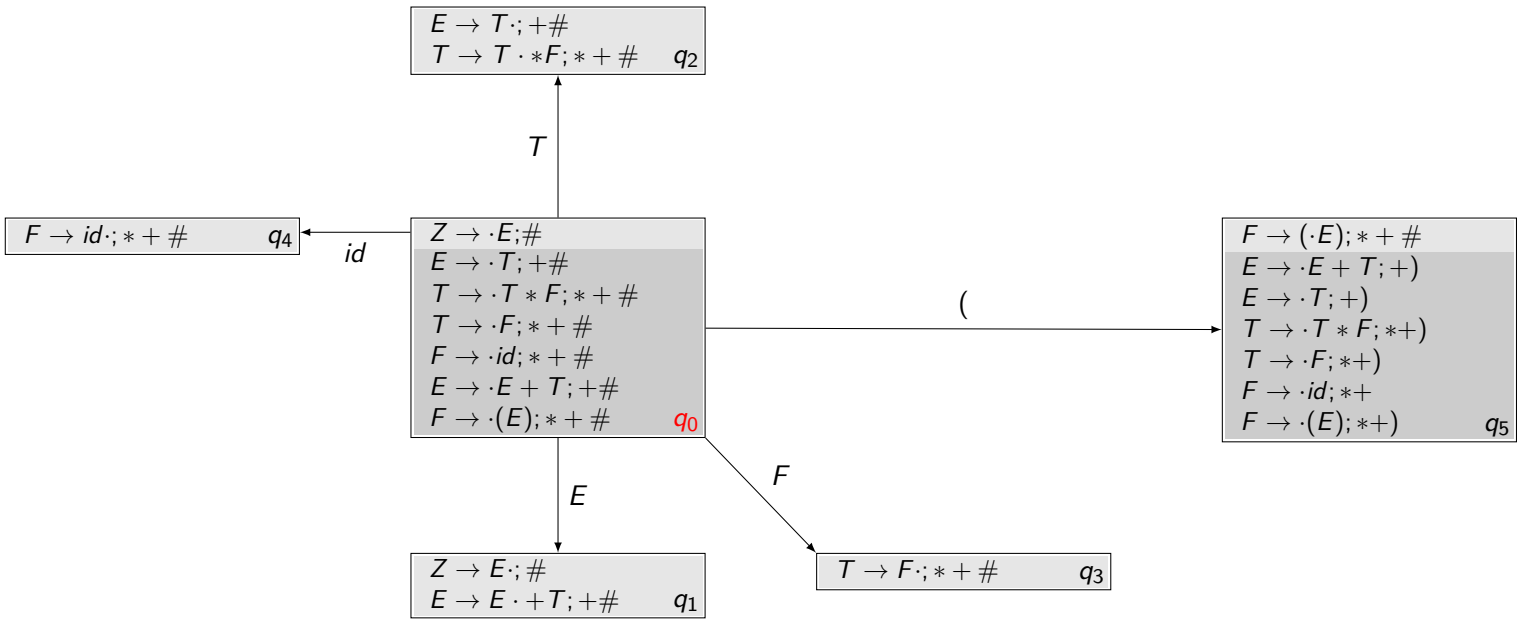
Achtung: Es gibt keine allgemeingültigen Verfahren

LALR(1)-Konstruktion am Beispiel

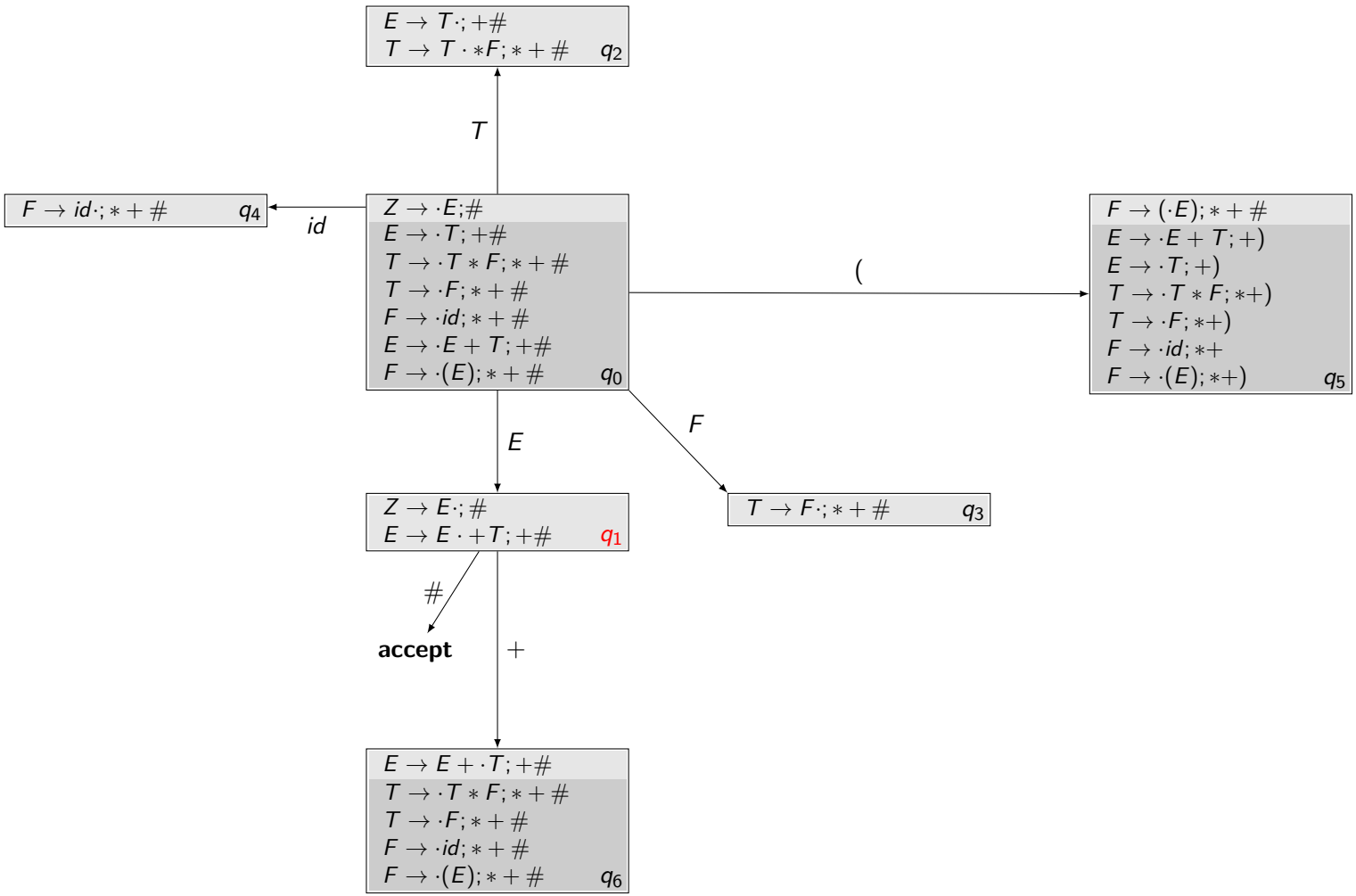
1: $Z \rightarrow E$
2: $E \rightarrow E + T$
3: $E \rightarrow T$
4: $T \rightarrow T * F$
5: $T \rightarrow F$
6: $F \rightarrow id$
7: $F \rightarrow (E)$

Vorgehen: konstruiere LR(0) Automaten und ergänze danach Rechtskontexte

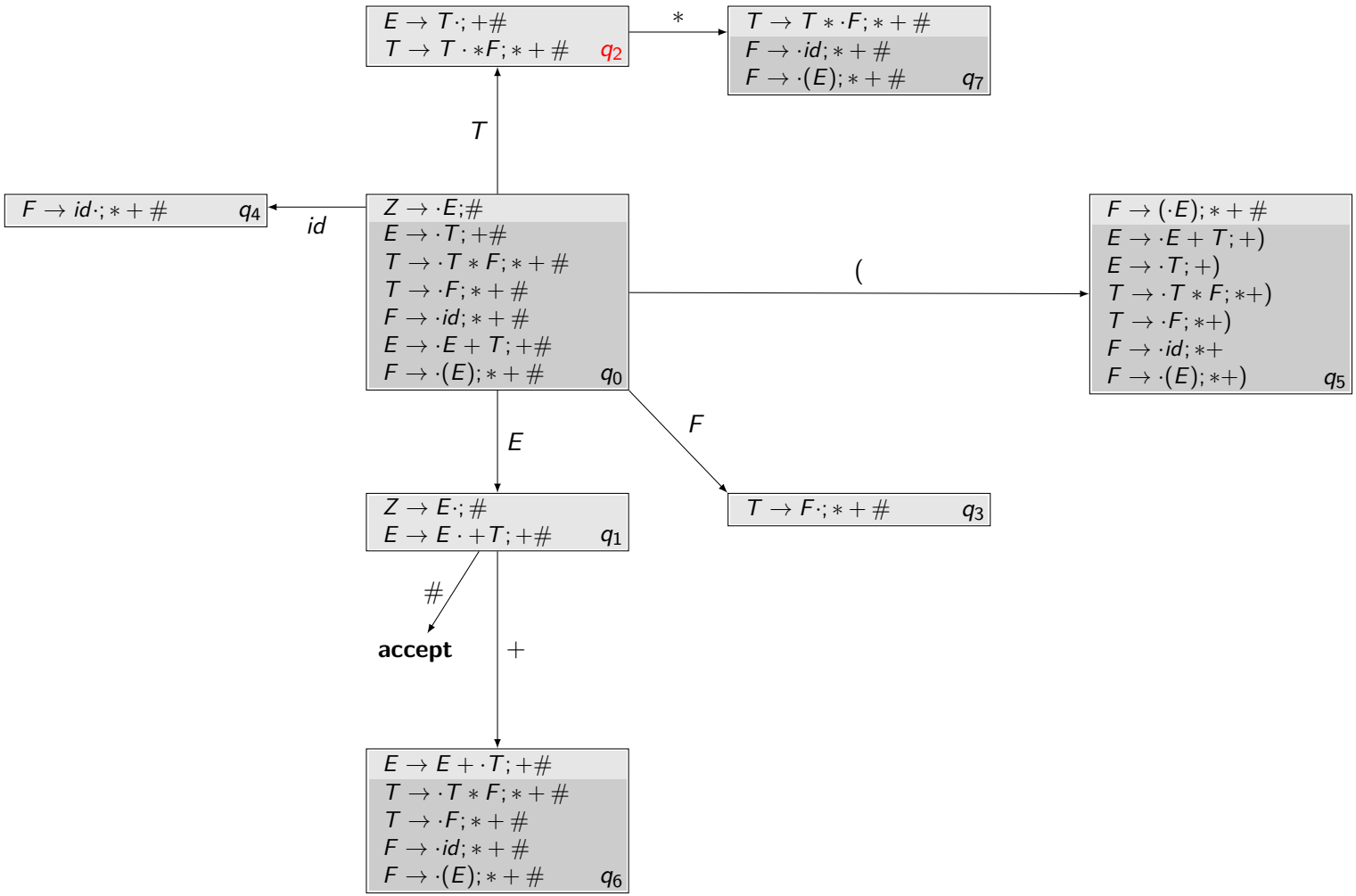
LALR(1)-Konstruktion am Beispiel



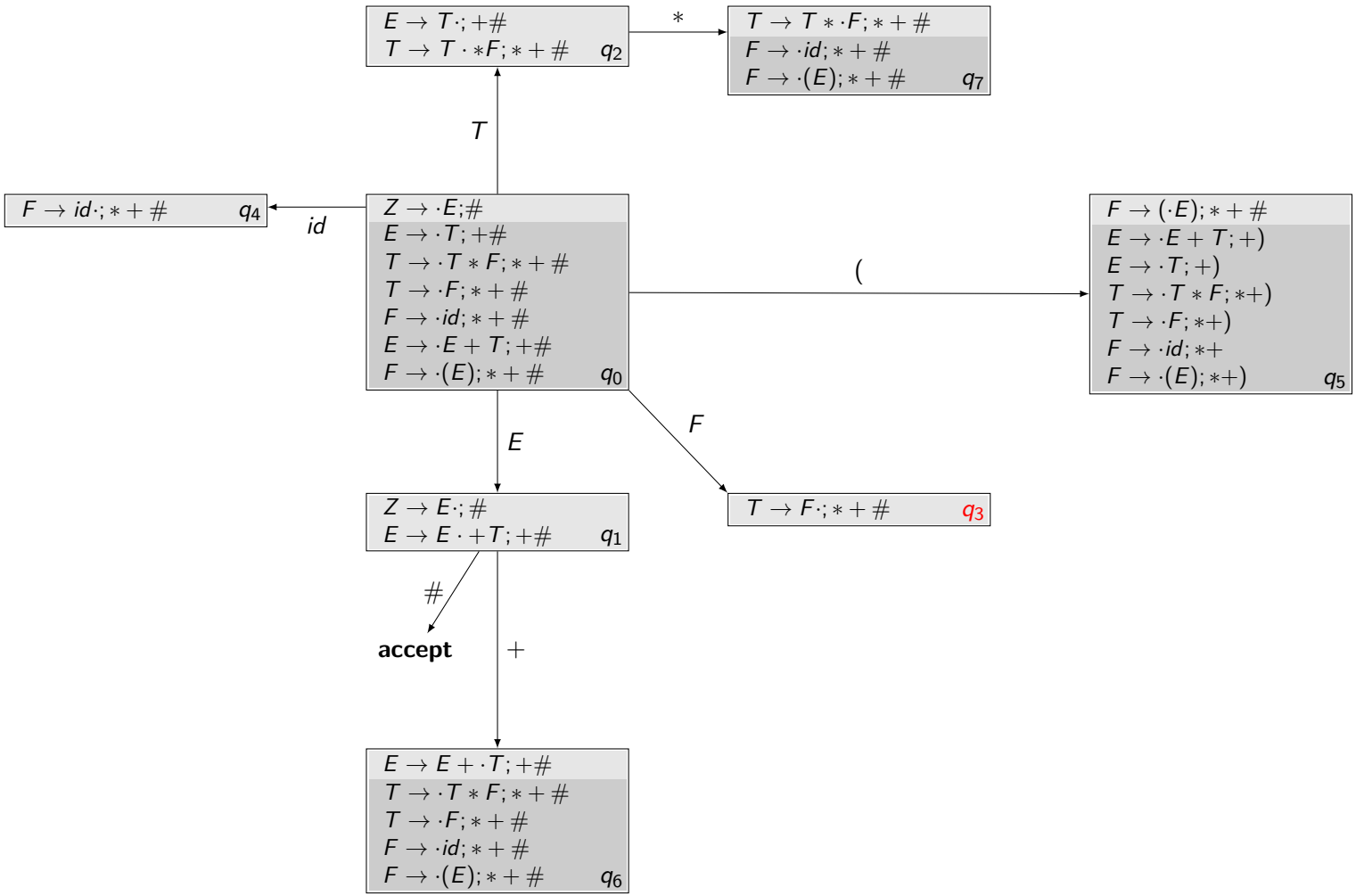
LALR(1)-Konstruktion am Beispiel



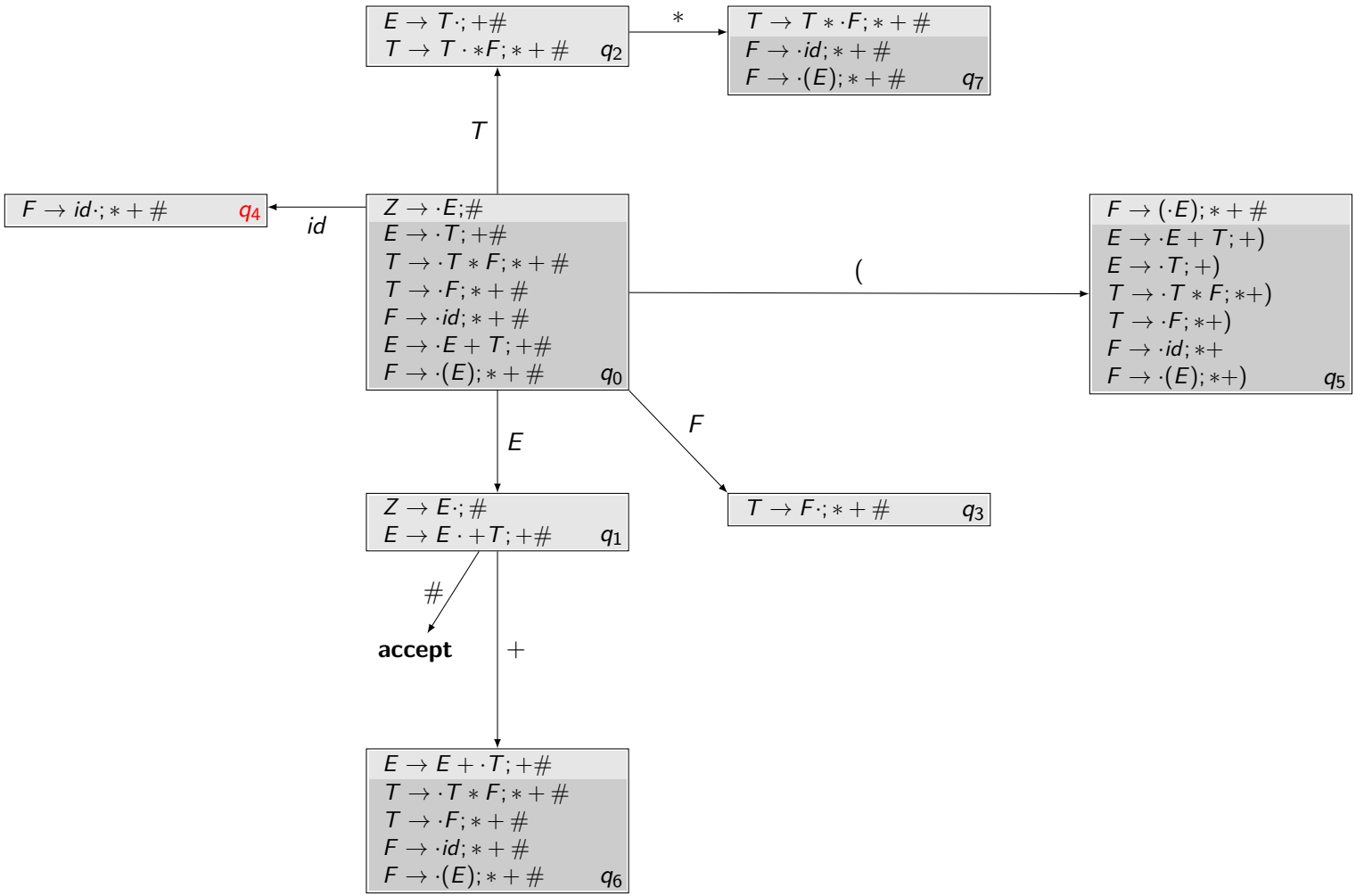
LALR(1)-Konstruktion am Beispiel



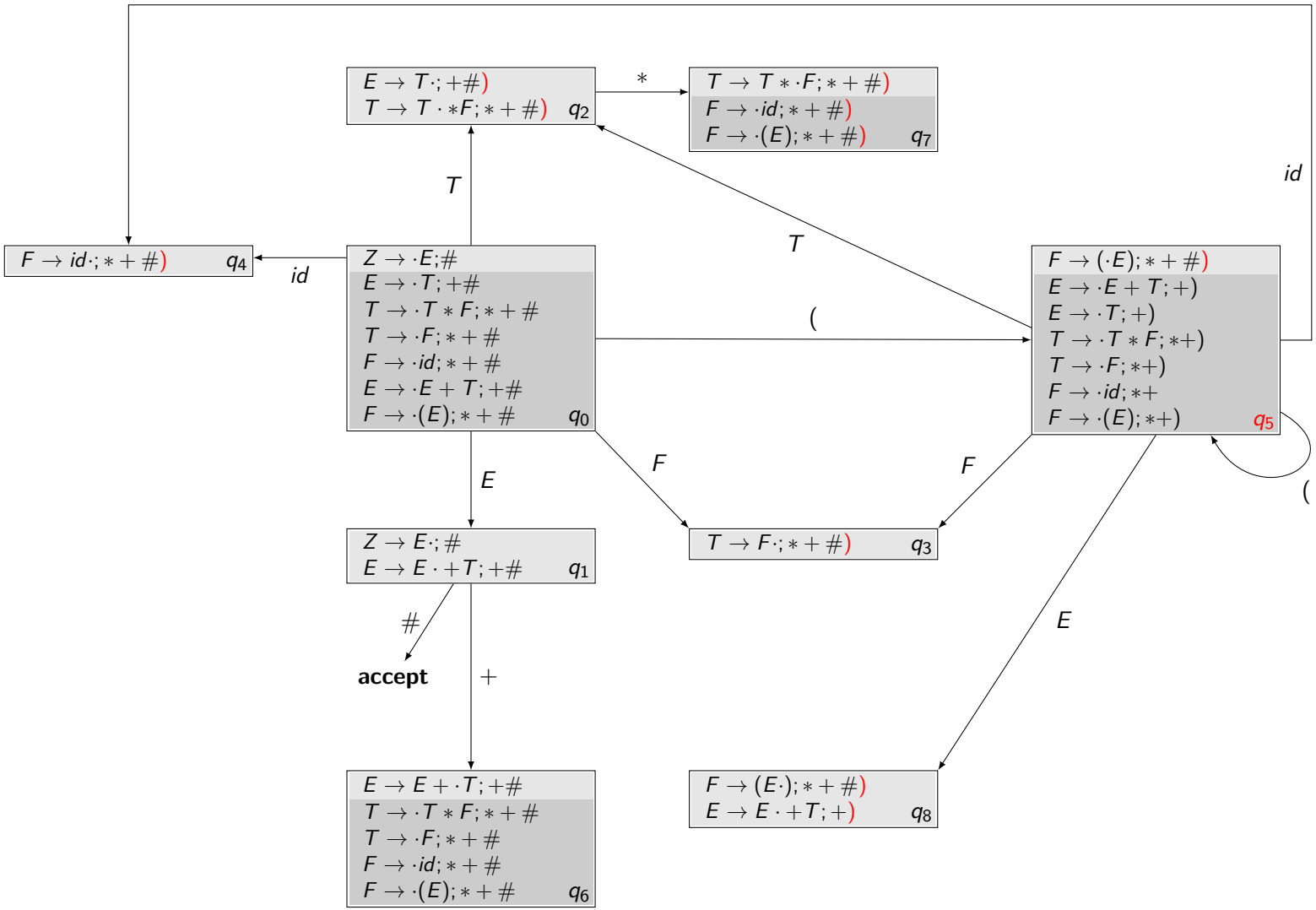
LALR(1)-Konstruktion am Beispiel



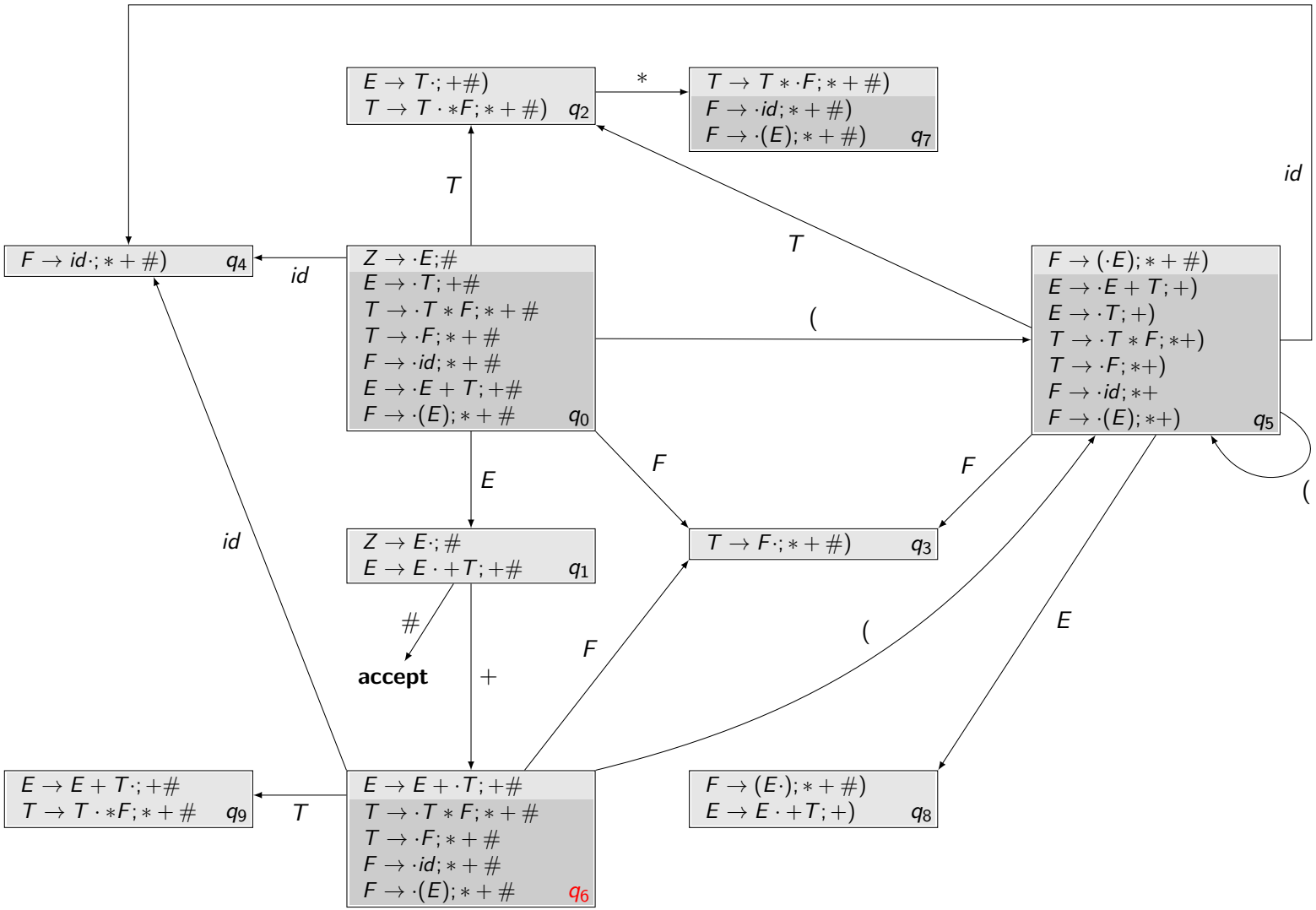
LALR(1)-Konstruktion am Beispiel



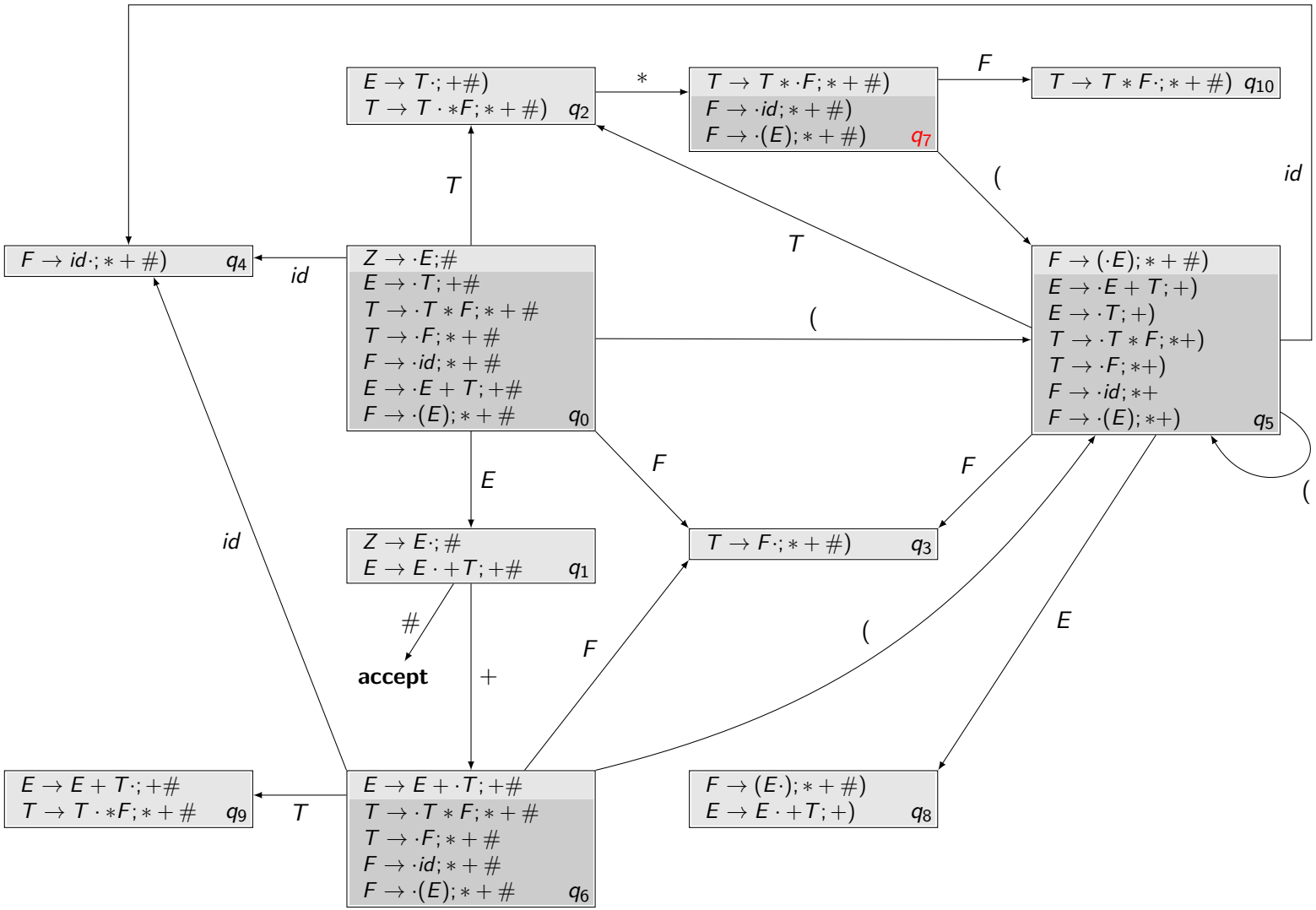
LALR(1)-Konstruktion am Beispiel



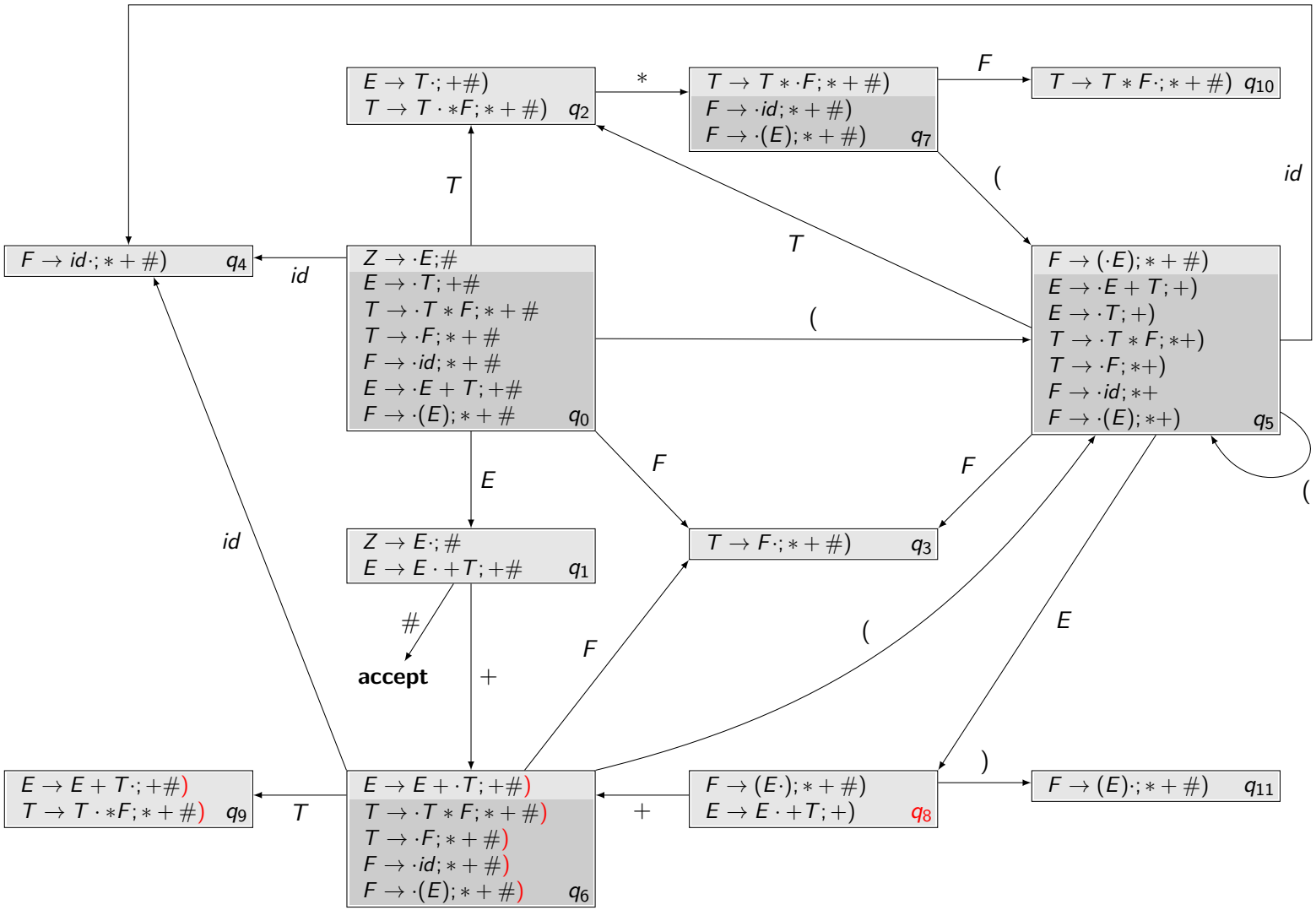
LALR(1)-Konstruktion am Beispiel



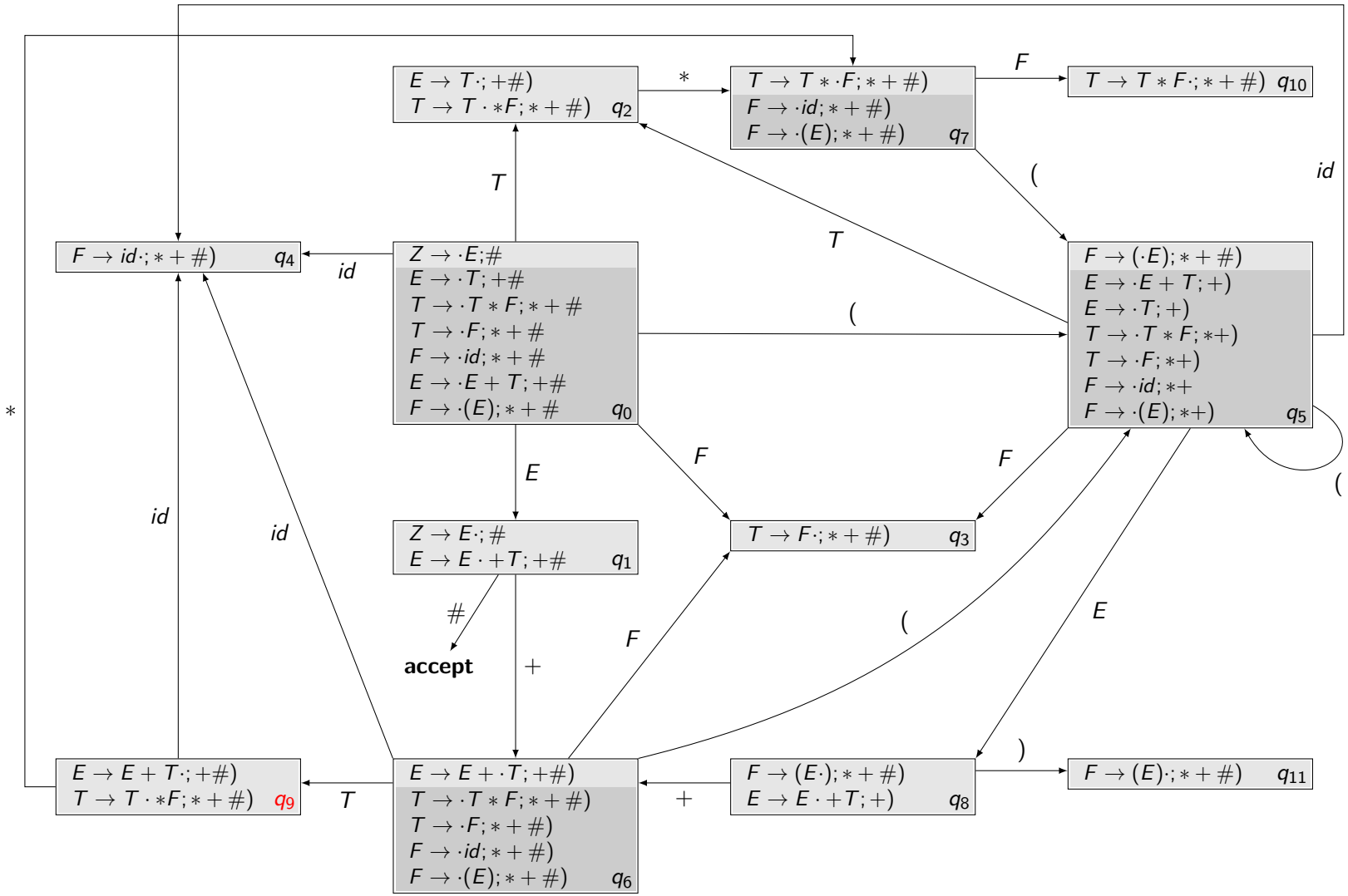
LALR(1)-Konstruktion am Beispiel



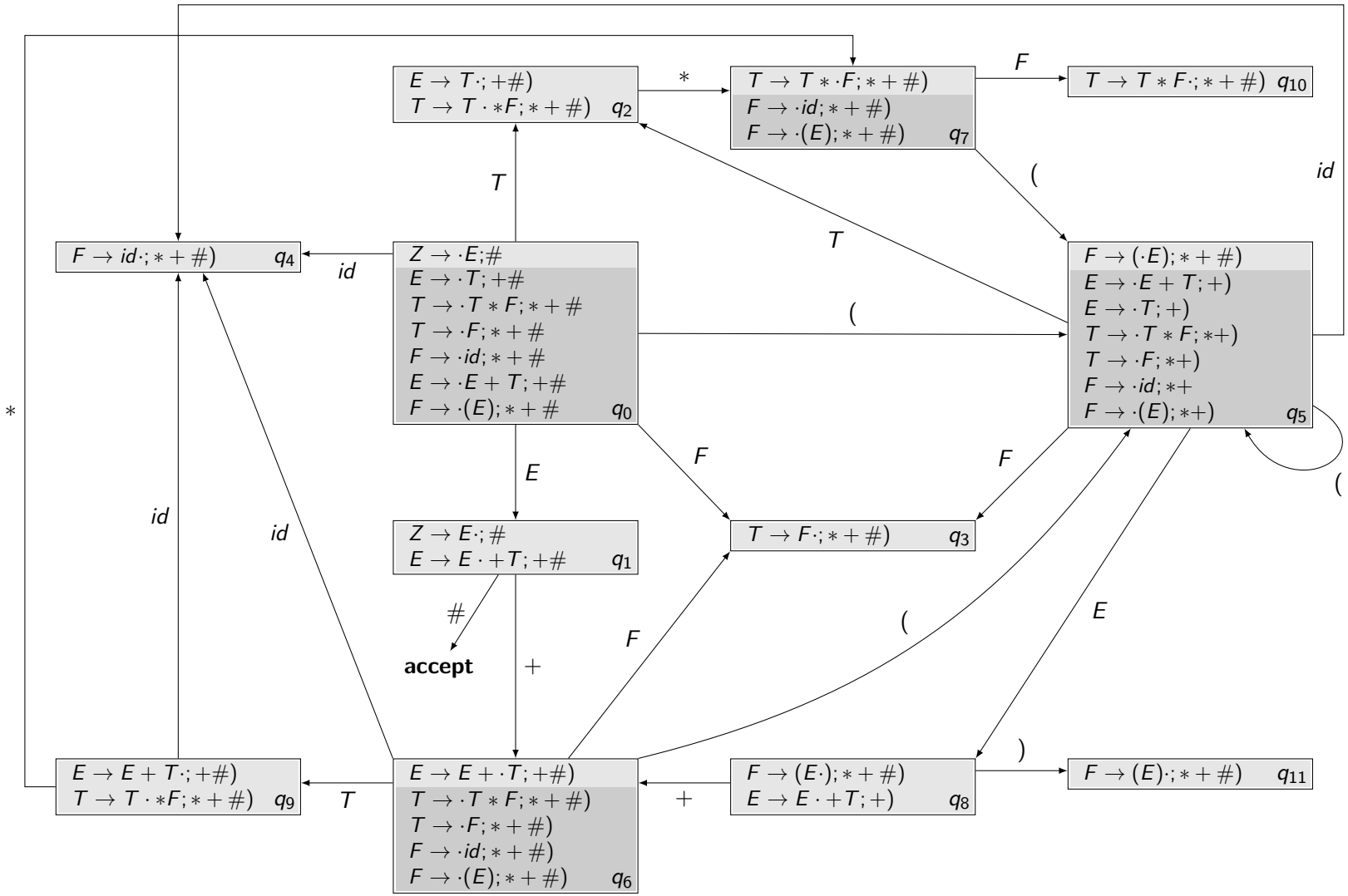
LALR(1)-Konstruktion am Beispiel



LALR(1)-Konstruktion am Beispiel



LALR(1)-Konstruktion am Beispiel



Übergangsfunktionen

q_0 $f(q_0, E) = q_1,\ f(q_0, T) = q2,\ f(q_0, F) = q3,$
$f(q_0, id) = q4,\ f(q_0, () = q5$

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

q_2 $f(q_2, +\#)) = \mathit{Reduziere}(E \to T) f(q_2, *) = q7$

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

q_4 $f(q_4, * + \#)) = \mathit{Reduziere}(F \to id)$

q_5 $f(q_5, E) = q8,\ f(q_5, T) = q2,\ f(q_5, F) = q3,$
$f(q_5, id) = q4,\ f(q_5, () = q5$

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

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

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

q_9 $f(q_9, +\#)) = \mathit{Reduziere}(E \to E + T),\ f(q_9, *) = q7$

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

q_{11} $f(q_1 1, * + \#)) = \mathit{Reduziere}(F \to (E))$

Übergangstabelle

	id	(	)	+	*	#	E	T	F
0	s4	s5	-	-	-	-	s1	s2	s3
1	-	-	-	s6	-	*			
2	-	-	r3	r3	s7	r3			
3	r5	r5	r5	r5	r5	r5			
4	r6	r6	r6	r6	r6	r6			
5	s4	s5	-	-	-	-	s8	s2	s3
6	s4	s5	-	-	-	-		s9	s3
7	s4	s5	-	-	-	-			s10
8	-	-	s11	s6	-	-			
9	-	-	r2	r2	s7	r2			
10	r4	r4	r4	r4	r4	r4			
11	r7	r7	r7	r7	r7	r7			

- Fehler

sY Shifte, neuer Zustand: Y

rX Reduziere mit Regel X

„ “ Don’t care

Generatoren für die syntaktische Analyse

- Parsergeneratoren ermöglichen eine kompakte Spezifikation syntaktischer 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>
```

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

Parser für XML

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

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

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 Token, 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 ;
```

Verhalten von Yacc im Konfliktfall

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)
```

Mögliche Behebungen des Konflikts

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 ;
```

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 (Shift-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

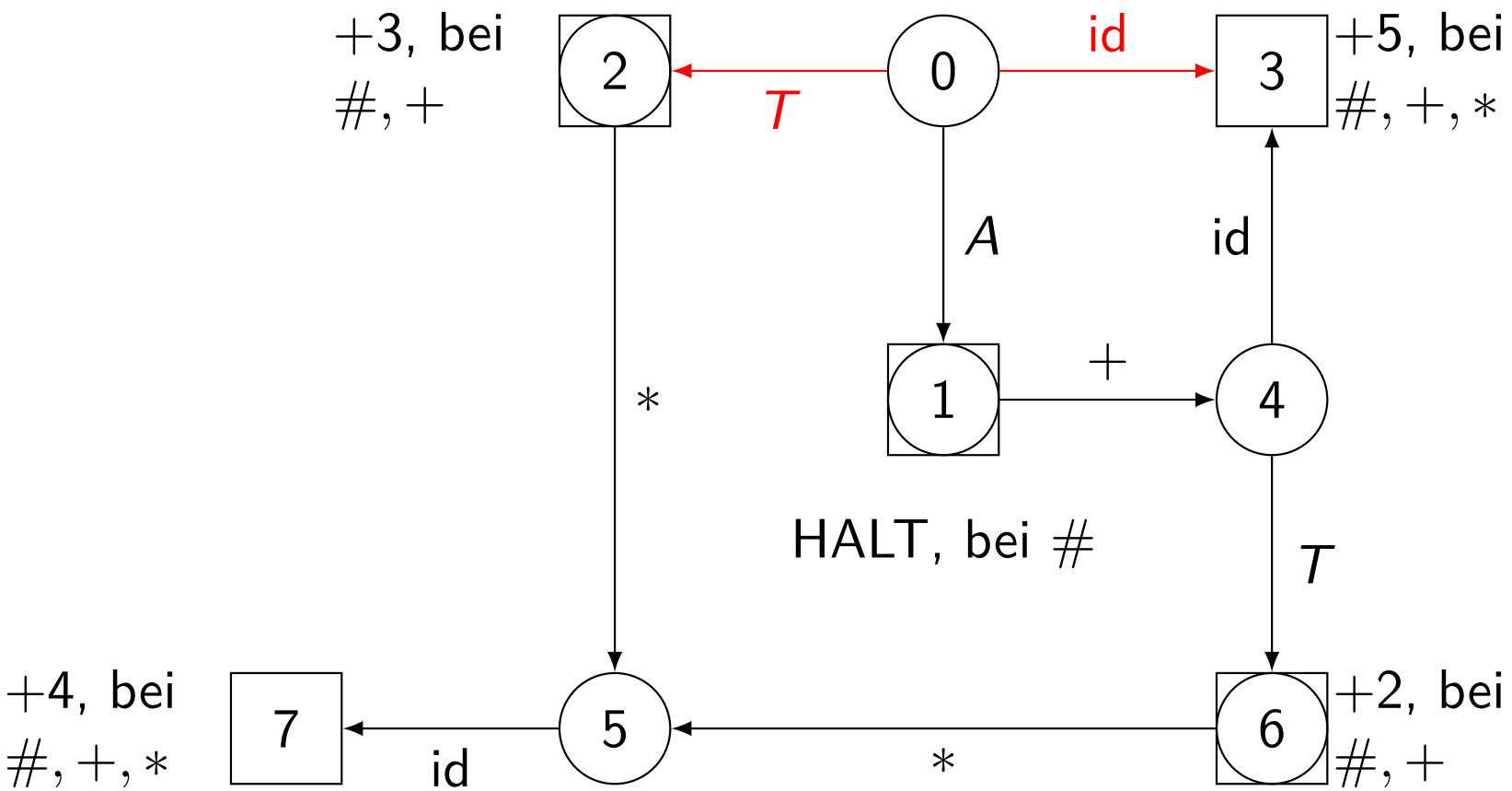
Methoden zur weiteren Kompression der Übergangsmatrix:

- 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: $Z \rightarrow E$ 2: $E \rightarrow E + T$ 3: $E \rightarrow T$

4: $T \rightarrow T * \mathrm{id}$ 5: $T \rightarrow \mathrm{id}$



Kettenproduktionen Eliminieren

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

4: $T \rightarrow T * \mathrm{id}$ 5: $T \rightarrow \mathrm{id}$

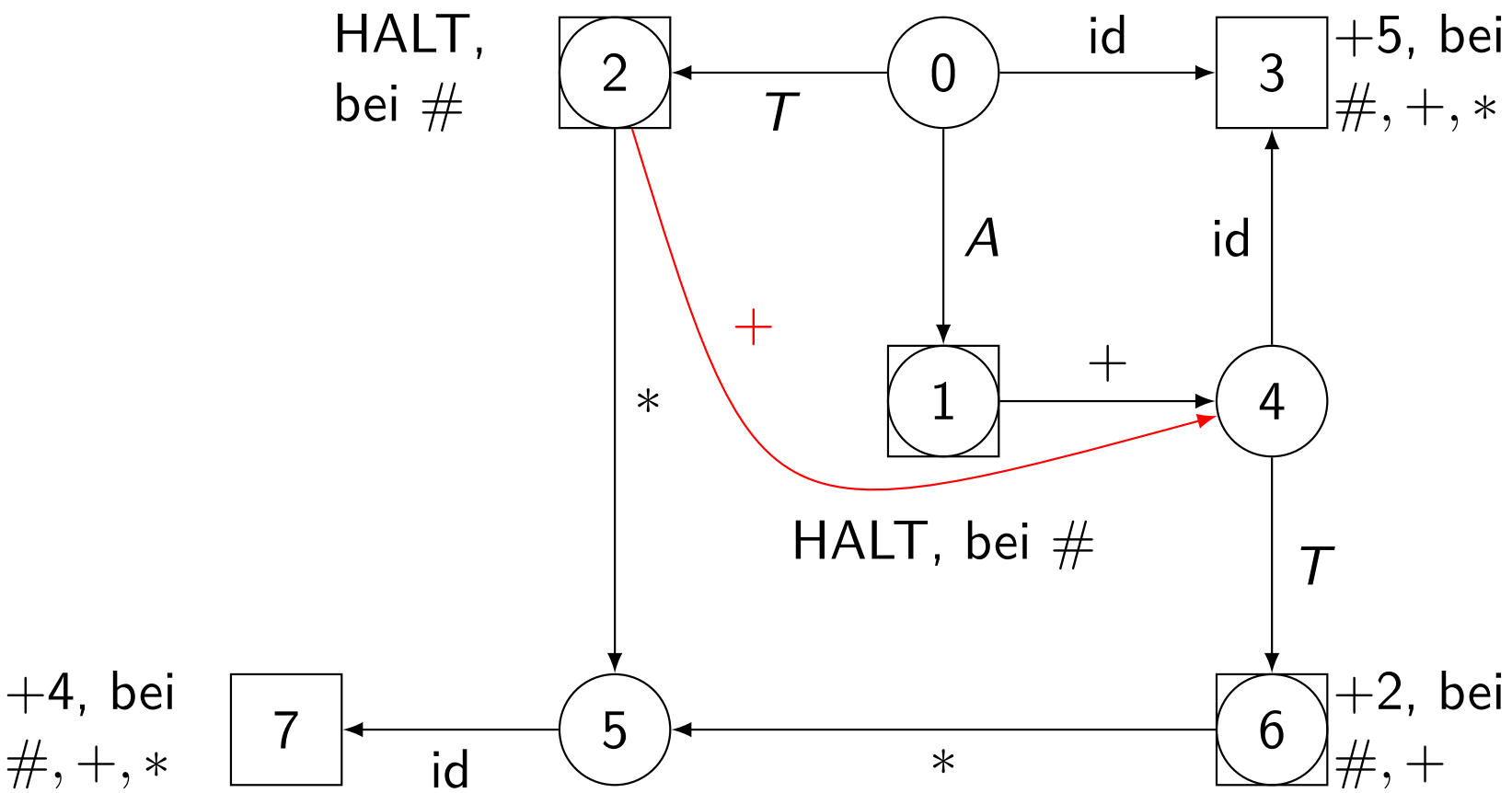


Tabelle mit Shift-Reduktionen, ohne Kettenproduktionen

	id	(	)	+	*	#	E	T	F
0	sr6	s3	-	-	-	-	s1	s2	s2
1			-	s4		*			
2	-	-	-	s4	s5	*			
3	sr6	s3	-	-	-	-	s6	s7	s7
4	sr6	s3	-	-	-	-		s8	s8
5	sr6	s3	-	-	-	-			sr4
6			sr7	s4		-			
7	-	-	sr7	s4	s5	-			
8	-	-	r2	r2	s5	r2			

- Fehler

sY Shifte, neuer Zustand: Y

„ “ Don’t care

rX Reduziere mit Regel X

srZ Shifte, reduziere dann mit Regel Z

Fehlermatrix und komprimierte Übergangsmatrix

	id	(	)	+	*	#
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

	E	T,F
0,1,2	s1	s2
3	s6	s7
4		s8
5,6,7,8		sr4

	id	(	)	+	*	#
0,1,2,3	r6	s3	r7	s4	s5	*
4,5,6,7						
8			r2	r2	s5	r2

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

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 Token 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 $q = [P \to P_{anf} \cdot P_{rest}; \Omega]$

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 Token, 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: $\mathrm{Folge}(A)$. 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*: $\mathrm{Anf}(\textit{statement})$ wird zu Ank(*expression*) hinzugefügt.
3. wenn $\varepsilon \in L(A)$, wird an Fehlerstelle $A \to \varepsilon$ 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

Fehlerbehandlung für LR-Parser

Beispiel für Grammatik:

$$E \to E{+}E \mid E{*}E \mid (E) \mid \textbf{id}$$

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	

Panischer Modus bei LR/SLR/LALR

Fehler wird entdeckt, falls in Zustand kein legaler Übergang des Automaten unter aktuellem Token
Berechnung der Ankermenge im Fehlerzustand q:

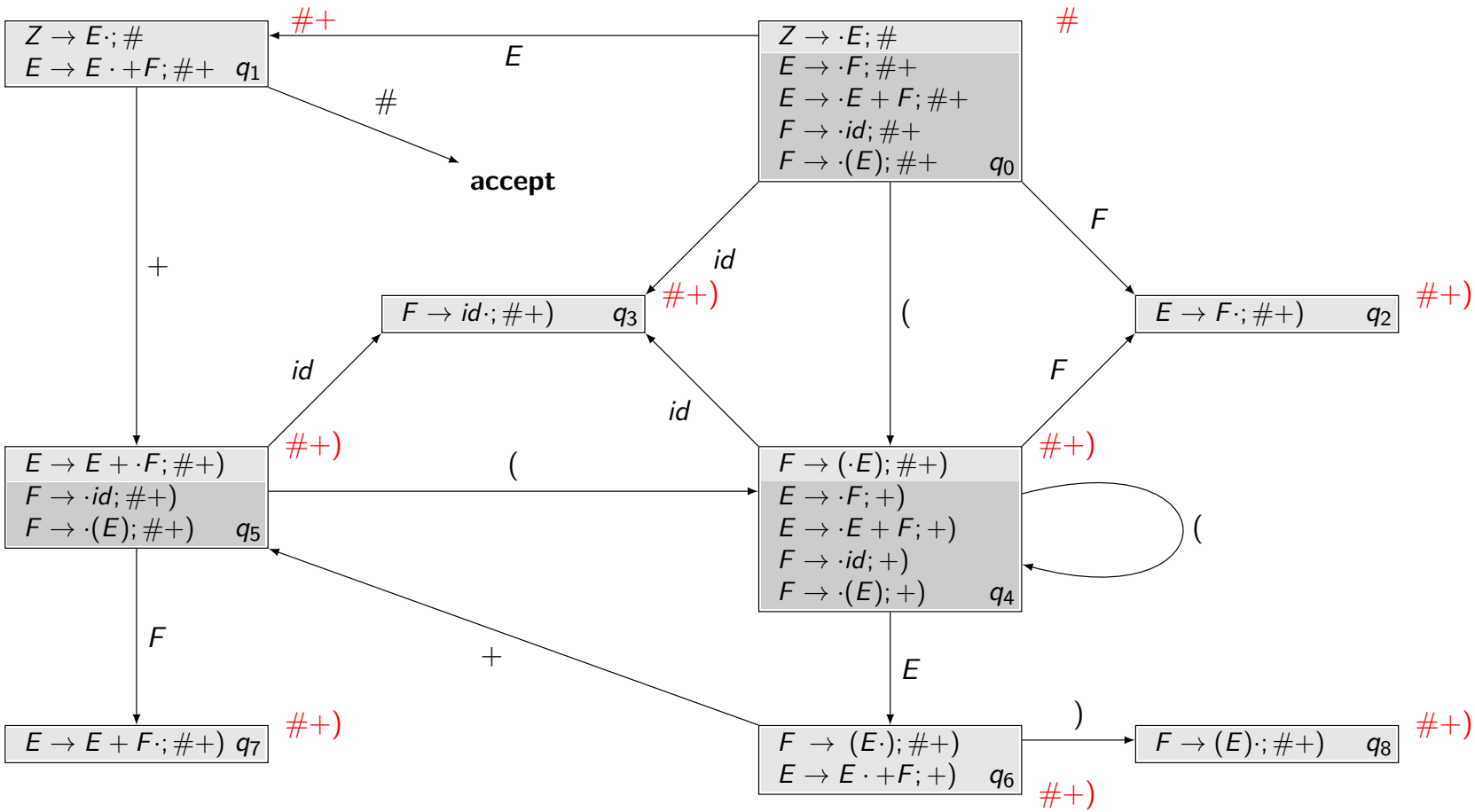
- Rechtskontexte in nicht-rekursiven Nicht-Closure-items
- Evtl. Terminale in Nicht-Closure-items rechts vom Punkt
- Auswahl kann manuell feingetunt werden

Recovery:

- Überspringe Eingabe bis Token in Ankermenge
- Mache Automatenübergänge in einen Reduktionszustand q', der Ankermenge als Rechtskontext hat
- Normal fortfahren, d.h. Reduziere

NB: AST-Aufbau etc nach Syntaxfehler nicht möglich

LR Beispiel-Automat



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 $Z \to S$.
- Tokens werden durchnummeriert, zu jeder Tokenposition i wird Menge S_i von Earley-Items berechnet
- Earley-items haben Form $[X \to \alpha.\beta, j]$, wobei j die Tokenposition des Anfangs von X angibt
- initiale Menge S_0: $\{[Z \to \cdot S, 0]\}$
- Algorithmus vervollständigt die S_i nach folgenden Regeln bis zum Fixpunkt:
  - **Prediction**: Für jede Situation $[X \to \alpha \cdot A\ \beta, j]$ in S_i füge eine neue Situation $[A \to \cdot\gamma, i]$ für jede Produktion der Grammatik mit A auf der linken Seite ein (transitive Hülle bilden).
  - **Completion**: Für jede Situation $[A \to \gamma\cdot, j]$ in S_i füge für jede Situation $[X \to \alpha \cdot A\ \beta, k]$ in S_j eine neue Situation $[X \to \alpha\ A \cdot \beta, k]$ in S_i ein.
  - **Scanning**: Sei **a** das nächste Token. Füge für jede Situation $[X \to \alpha \cdot \mathbf{a}\ \beta, j]$ in S_i eine Situation $[X \to \alpha\ \mathbf{a} \cdot \beta, j]$ 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:
$[Z \to \cdot S, 0]$

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

Beispiel

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

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

Iteration:
$[Z \to S\cdot, 0]$
$[S \to S \ \cdot \ \textbf{und}\ S, 0]$
$[S \to S \ \cdot \ \textbf{oder}\ S, 0]$

Beispiel

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

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

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

Beispiel

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

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

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

Beispiel

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

Situationen S_4:

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

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

Iteration:

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

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

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

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

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

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

Beispiel

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

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

Iteration:

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

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

Beispiel

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

Situationen S_6:

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

Iteration:

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

Beispiel

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

Situationen S_7:

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

Iteration:

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

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