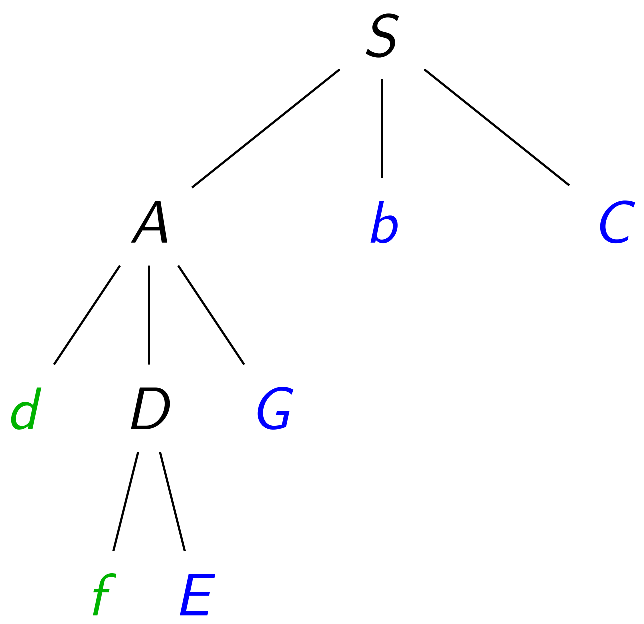
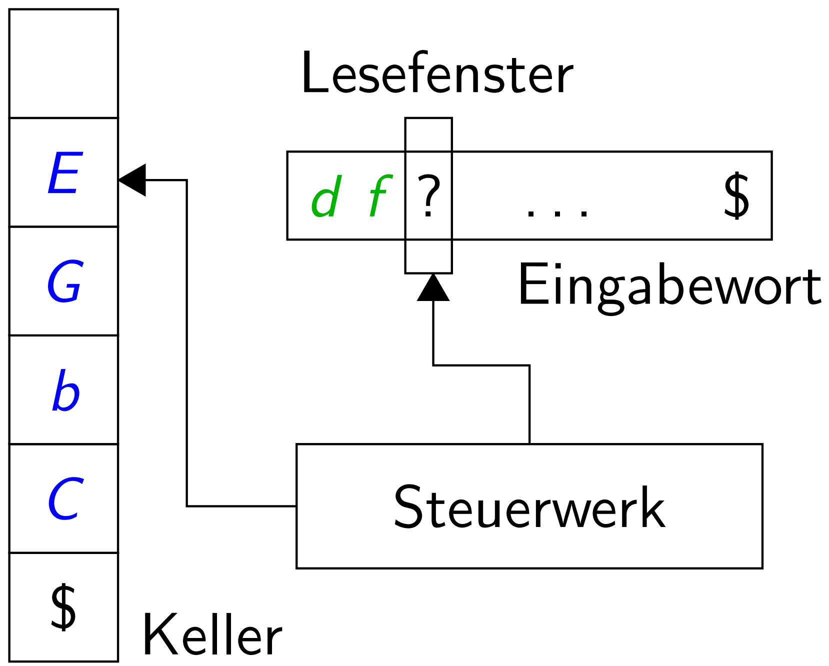
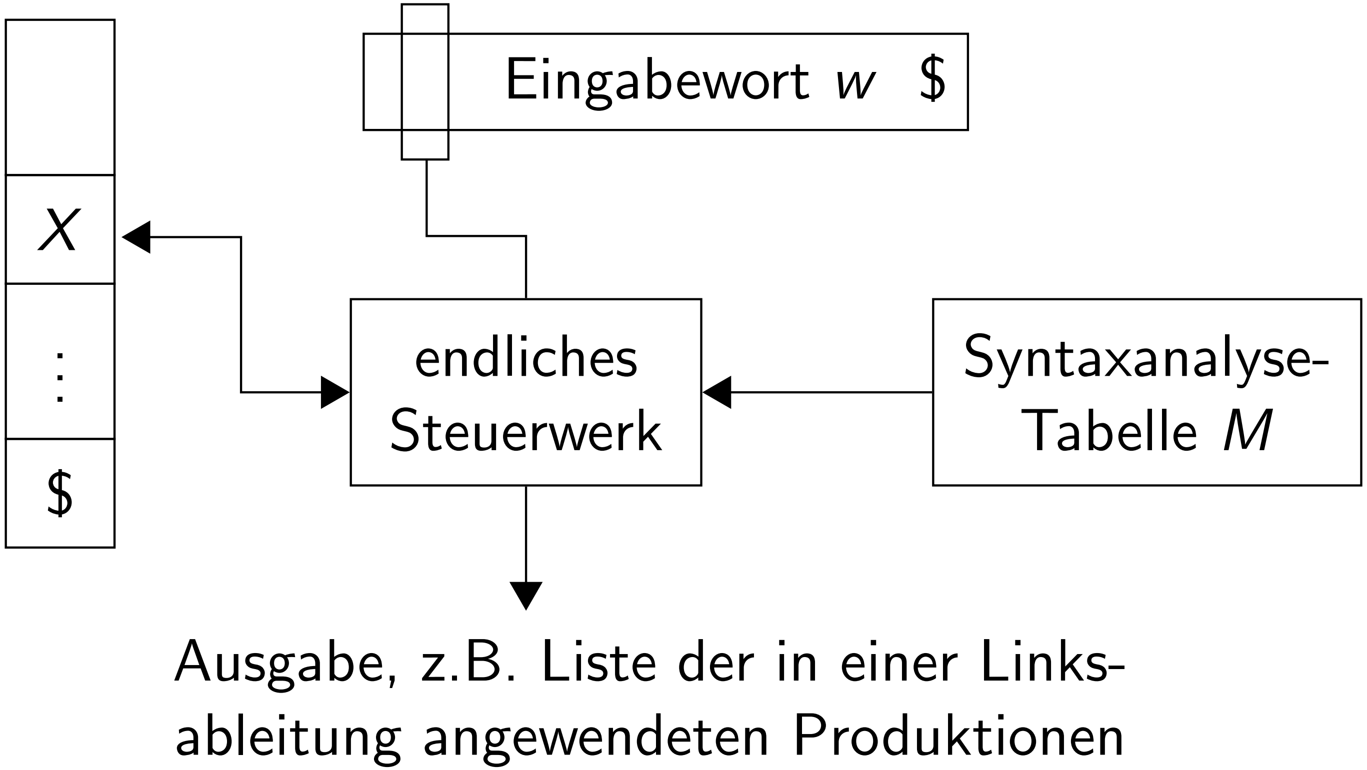
Ein tabellengesteuerter Top-Down-Parser

Idee: Der noch abzuleitende Teil des Ableitungsbaumes wird in einem Keller gespeichert:





Aufbau eines tabellengesteuerten Top-Down Parsers



Konstruktion der Syntaxanalyse-Tabelle

- Die Zeilen der Syntaxanalyse-Tabelle sind mit den nichtterminalen Symbolen der Grammatik markiert.
- Die Spalten der Tabelle sind mit den Vorschausymbolen markiert (also einem terminalen Symbol bzw. der Endmarke \$).
- Jeder Tabelleneintrag enthält eine Produktion der Grammatik oder bleibt leer.

Konstruktion der Syntaxanalyse-Tabelle M:

- Für jede Produktion $A \to \alpha$ in P und für alle $f \in$ **Erwartet**$(A \to \alpha)$ setze: $M(A, f) := A \to \alpha$

Die LL(1)-Bedingung garantiert, dass jeder Tabellenplatz *höchstens* einmal besetzt wird!
Jedes leere Feld beschreibt eine Fehlersituation.

Beispiel (Grammatik G_2)

Produktionen: $S \to (S)R \mid aR$
$R \to +S \mid *S \mid \varepsilon$

$\mathbf{First}(S) = \{a, (\},$ $\mathbf{Follow}(S) = \{\$,)\},$
$\mathbf{First}(R) = \{\varepsilon, +, *\}$ $\mathbf{Follow}(R) = \{\$,)\}$

$$\mathbf{Erwartet}(A \to \alpha) = \begin{cases} \mathbf{First}(\alpha) & \text{, falls } \varepsilon \notin \mathbf{First}(\alpha) \\ (\mathbf{First}(\alpha) \setminus \{\varepsilon\}) \cup \mathbf{Follow}(A) & \text{, falls } \varepsilon \in \mathbf{First}(\alpha) \end{cases}$$

$\mathbf{Erwartet}(S \to (S)R) = \{(\}$
$\mathbf{Erwartet}(S \to aR) = \{a\}$
$\mathbf{Erwartet}(R \to +S) = \{+\}$
$\mathbf{Erwartet}(R \to *S) = \{*\}$
$\mathbf{Erwartet}(R \to \varepsilon) = \{\$,)\}$

Konstruktion der Syntaxanalyse-Tabelle

Beispiel

Für die Grammatik mit den Produktionen

1: $S \to (S)R$, 2: $S \to aR$, 3: $R \to +S$, 4: $R \to *S$ und 5: $R \to \varepsilon$,

erhält man die folgende Syntaxanalyse-Tabelle M:

M	a	(	)	$+$	$*$	\$
S	$S \to aR$	$S \to (S)R$				
R			$R \to \varepsilon$	$R \to +S$	$R \to *S$	$R \to \varepsilon$

bzw. die vereinfachte Form mit Produktionsnummern:

M	a	(	)	$+$	$*$	\$
S	2	1				
R			5	3	4	5

Arbeitsweise eines tabellengesteuerten Top-Down Parsers

Initiale Situation

- Auf dem Eingabeband steht das zu analysierende Wort w gefolgt von der Endmarke \$.
- Das Lesefenster steht auf dem ersten Zeichen von w.
- Der Keller enthält als unterstes Symbol \$ und darüber das Startsymbol S der Grammatik.

Arbeitsweise eines tabellengesteuerten Top-Down Parsers

Arbeitsschritt

Sei X das oberste Kellersymbol und a im Lesefenster der Eingabe.

(i) Ist $X \in T \cup \{\$\}$, dann vergleiche X und a:

- (a) Ist $X \neq a$, dann beende das Parsen: $w \notin L(G)$.
- (b) Sonst ist $X = a$.
  - (α) Ist $a = \$$, dann beende das Parsen: $w \in L(G)$.
  - (β) Ist $a \in T$, dann lösche X vom Keller und setze das Lesefenster ein Zeichen weiter.

(ii) Ist $X \in N$, dann betrachte den Tabelleneintrag $M(X, a)$:

- (a) Ist $M(X, a)$ leer, dann beende das Parsen: $w \notin L(G)$.
- (b) Enthält $M(X, a)$ die Produktion $X \to A_1 \ldots A_r$, dann
  - lösche X,
  - schreibe $A_r, \ldots, A_1$ (in dieser Reihenfolge!) auf den Keller
  - und gib die Produktion aus.

Arbeitsweise eines tabellengesteuerten Top-Down Parsers

Erste Schritte des Parsers für die Eingabe $a * (a + a)$ zur Grammatik G_2:

Anfangssituation

$$\begin{array}{|c|}\hline \\ \hline S \\ \hline \$ \\ \hline \end{array} \qquad \boxed{\boxed{\textcolor{red}{\mathrm{a}}} * (a+a)\$}$$

1. Schritt, (iib)

$$\begin{array}{|c|}\hline \\ \hline a \\ \hline R \\ \hline \$ \\ \hline \end{array} \qquad \boxed{\boxed{\textcolor{red}{\mathrm{a}}} * (a+a)\$}$$

Ausgabe:
$S \rightarrow aR$

2. Schritt, (ibβ)

$$\begin{array}{|c|}\hline \\ \hline R \\ \hline \$ \\ \hline \end{array} \qquad \boxed{a \boxed{\textcolor{red}{*}} (a+a)\$}$$

Das Vorschau-Symbol ist rot markiert.

Konfigurationen des Parsers

Um die Arbeitsweise eines tabellengesteuerten Top-Down-Parsers kompakt darzustellen, wird

- der momentane Kellerinhalt (mit dem obersten Symbol X) und
- die restliche Eingabe (ab dem Vorschau-Symbol a)

als **Konfiguration**, also als Wortpaar $[X \ldots \$, \quad a \ldots \$]$ beschrieben.

Jeder Schritt des Parsers erzeugt eine neue Konfiguration.

Ein Schritt der Form (ibβ) wird bezeichnet durch

$$[aY \ldots \$, \quad ab \ldots \$] \vdash\!\!\!\longrightarrow [Y \ldots \$, \quad b \ldots \$]$$

und ein Schritt der Form (iib) durch

$$[XY \ldots \$, \quad a \ldots \$] \underset{(i)}{\vdash\!\!\!\longrightarrow} [A_1 \ldots A_r Y \ldots \$, \quad a \ldots \$]$$

i gibt die Nummer der angewendeten Produktion $X \rightarrow A_1 \ldots A_r$ an.

Beispiel (Konfigurationen des Top-Down-Parsers)

Produktionen:
$S \to (S)R \mid aR,$
$R \to +S \mid *S \mid \varepsilon$

(fehlerhafte) Eingabe:
`((a+a)a)$`

M	a	(	)	$+$	$*$	\$
S	2	1				
R			5	3	4	5

$a \notin \textbf{First}(R) = \{\varepsilon, +, *\}$
$a \notin \textbf{Follow}(R) = \{), \$\}$

mögliche Fehlermeldung:
$+$, $*$,) oder eof erwartet

$$
\begin{array}{lll}
 & \texttt{[S\$} & \texttt{,((a+a)a)\$]} \\
\vdash_{(1)} & \texttt{[(S)R\$} & \texttt{,((a+a)a)\$]} \\
\vdash & \texttt{[S)R\$} & \texttt{, (a+a)a)\$]} \\
\vdash_{(1)} & \texttt{[(S)R)R\$} & \texttt{, (a+a)a)\$]} \\
\vdash & \texttt{[S)R)R\$} & \texttt{, a+a)a)\$]} \\
\vdash_{(2)} & \texttt{[aR)R)R\$} & \texttt{, a+a)a)\$]} \\
\vdash & \texttt{[R)R)R\$} & \texttt{, +a)a)\$]} \\
\vdash_{(3)} & \texttt{[+S)R)R\$} & \texttt{, +a)a)\$]} \\
\vdash & \texttt{[S)R)R\$} & \texttt{, a)a)\$]} \\
\vdash_{(2)} & \texttt{[aR)R)R\$} & \texttt{, a)a)\$]} \\
\vdash & \texttt{[R)R)R\$} & \texttt{,)a)\$]} \\
\vdash_{(5)} & \texttt{[)R)R\$} & \texttt{,)a)\$]} \\
\vdash & \texttt{[R)R\$} & \texttt{, a)\$]} \\
\vdash & \texttt{error} &
\end{array}
$$

Implementation eines Top-Down-Parsers nach der Methode des rekursiven Abstiegs (*recursive descent parser*)

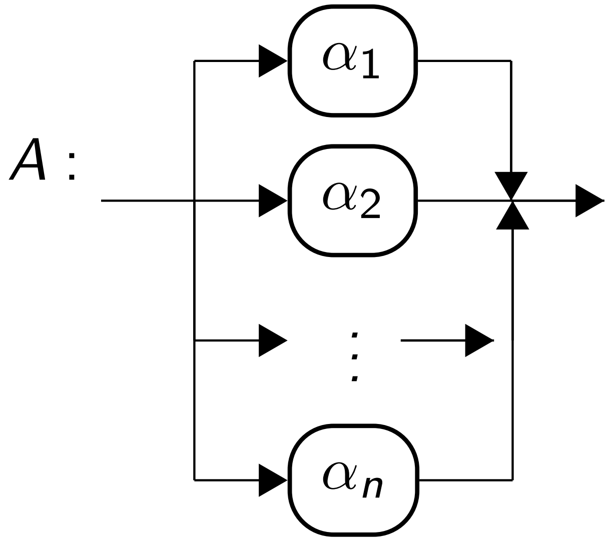
Idee: Man benutzt
Syntaxgraphen zur Darstellung der Grammatik der Programmiersprache und interpretiert sie als Ablaufpläne für rekursive Prozeduren.

Gegeben: eine kontextfreie Grammatik $G = (N, T, P, S)$.

Für jedes $A \in N$ wird ein Syntaxgraph mit Namen A erzeugt.
Jeder Syntaxgraph hat *eine* Eingangskante und *eine* Ausgangskante.

Konstruktion von Syntaxgraphen aus einer Grammatik

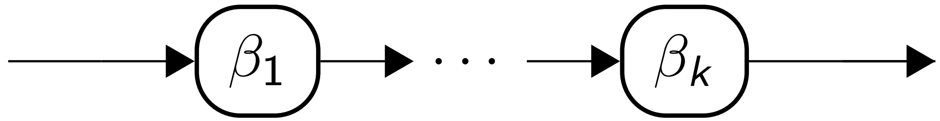
1. Sind $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$ alle A-Produktionen, erzeugen wir diesen Graphen A:



wobei die Teilgraphen $\boxed{\alpha_i}$ wie folgt bestimmt werden:

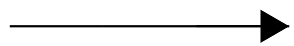
Konstruktion von Syntaxgraphen aus einer Grammatik

2. Ist $\alpha_i = \beta_1 \ldots \beta_k$ mit $\beta_i \in N \cup T$, dann hat der Teilgraph die Form



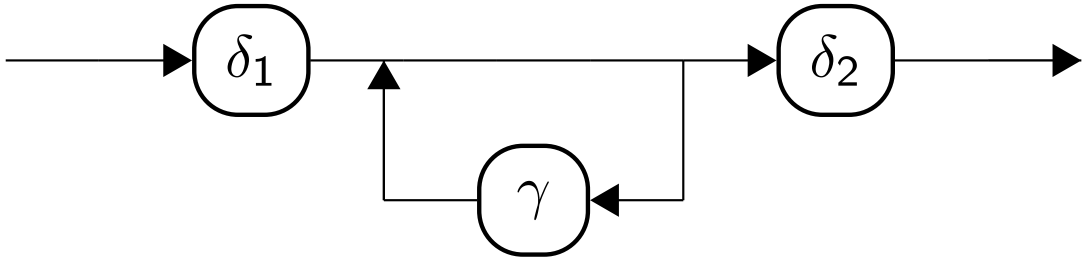
$$\text{mit } \beta_i = \begin{cases} \boxed{B} & \text{, falls } \beta_i = B \in N \\ x & \text{, falls } \beta_i = x \in T. \end{cases}$$

Ist $\alpha_i = \varepsilon$, so hat der Teilgraph das Aussehen

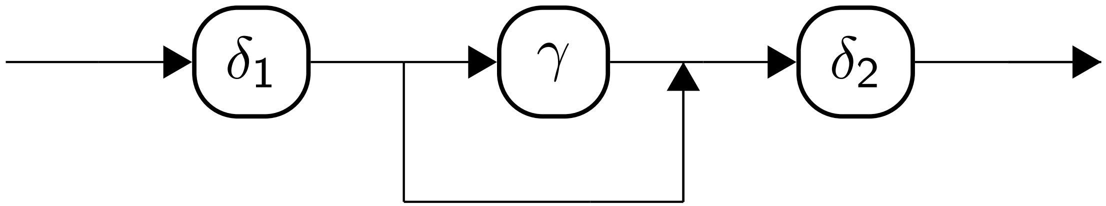


Konstruktion von Syntaxgraphen aus einer Grammatik

Ist $\alpha_i = \delta_1 \, \{\gamma\} \, \delta_2$, so hat der Teilgraph die Form



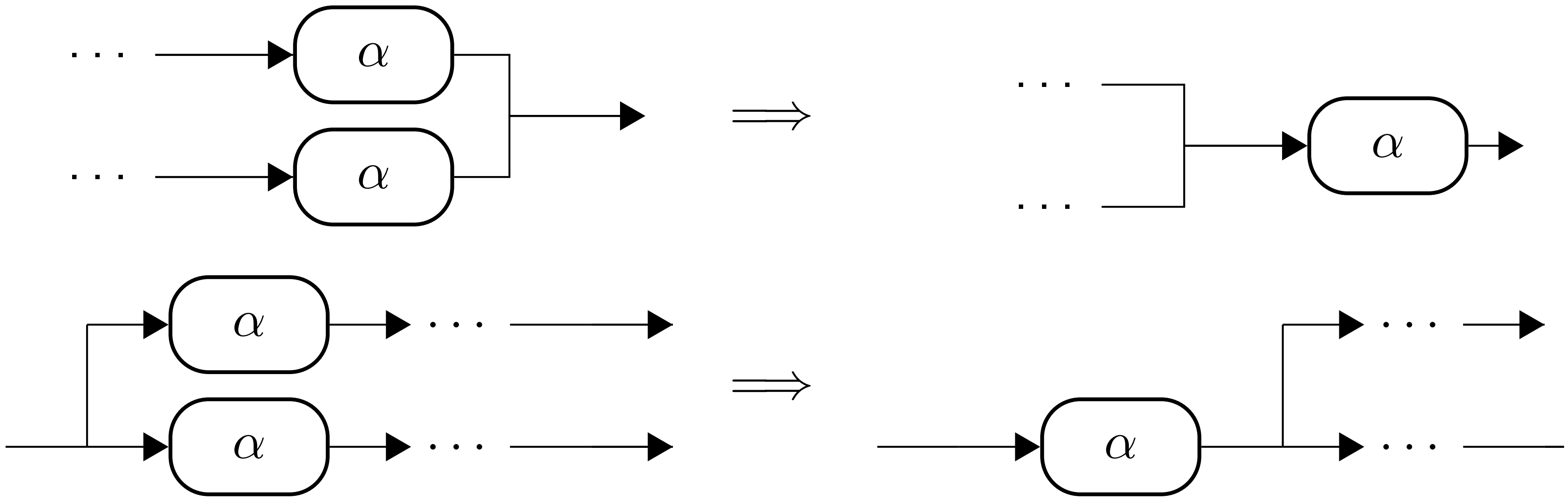
Ist $\alpha_i = \delta_1 \, [\gamma] \, \delta_2$, so hat der Teilgraph die Form



wobei die Teilgraphen $\boxed{\delta_1}$, $\boxed{\delta_2}$ und $\boxed{\gamma}$ wieder wie in 2) bestimmt werden.

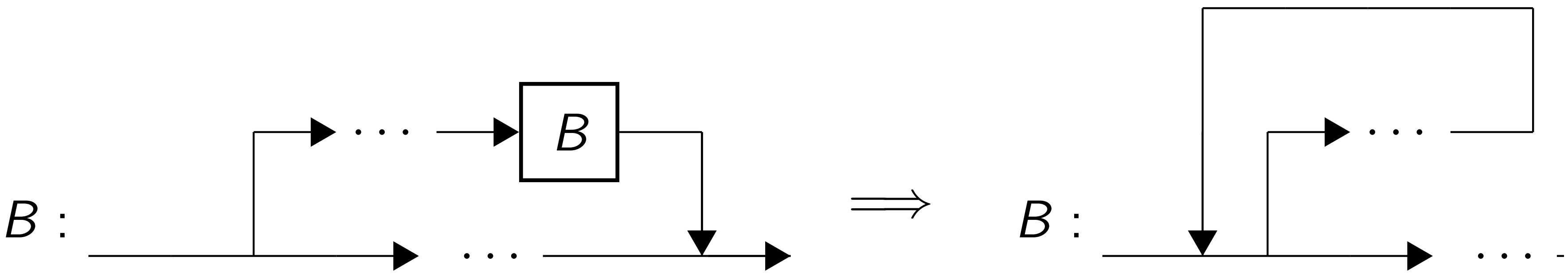
Vereinfachung der Syntaxgraphen

1. Ein Teilgraph $\boxed{B}$ mit $B \in N$ kann durch den Syntaxgraphen für B ersetzt werden (*Substitutionsregel*).
2. Identische Teilgraphen mit gemeinsamem Ausgang bzw. Eingang können zusammengefasst werden:



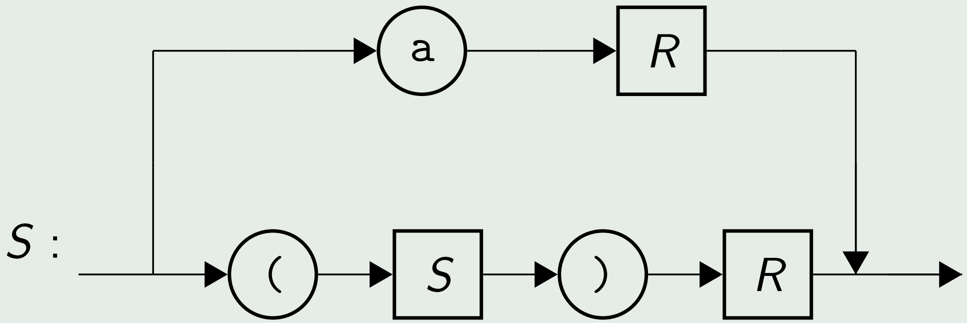
Vereinfachung der Syntaxgraphen

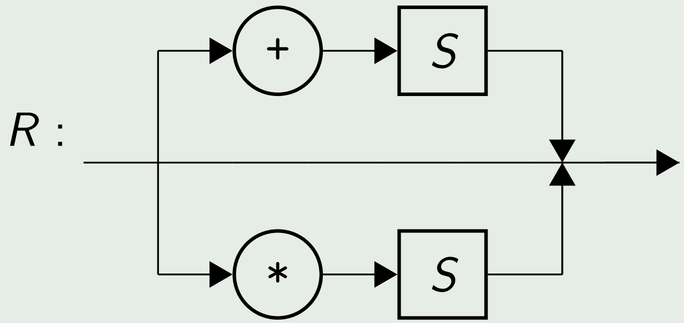
3. Tritt am Ausgang eines Syntaxgraphen für B der Knoten $\boxed{B}$ auf, so ersetzt man diesen durch eine Kante auf den Eingang des Syntaxgraphens (*Iteration* statt *Endrekursion*):



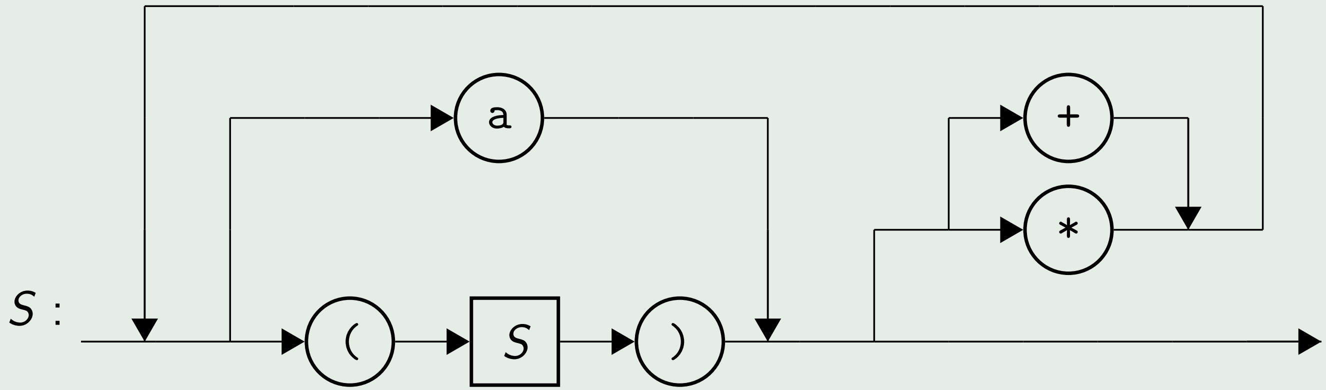
Beispiel

Die Syntaxgraphen für die Grammatik G_2 aus dem letzten Beispiel sind:





Die Vereinfachung ergibt:



Definition

Ein Syntaxgraph heißt **deterministisch**, wenn für jede Verzweigung gilt, dass die Mengen der ersten terminalen Zeichen, die auf jedem Ast der Verzweigung zu erreichen sind, paarweise disjunkt sind.

Bemerkung

Diese terminalen Zeichen können dann als **Wegweiser** *dienen.*

Bemerkung

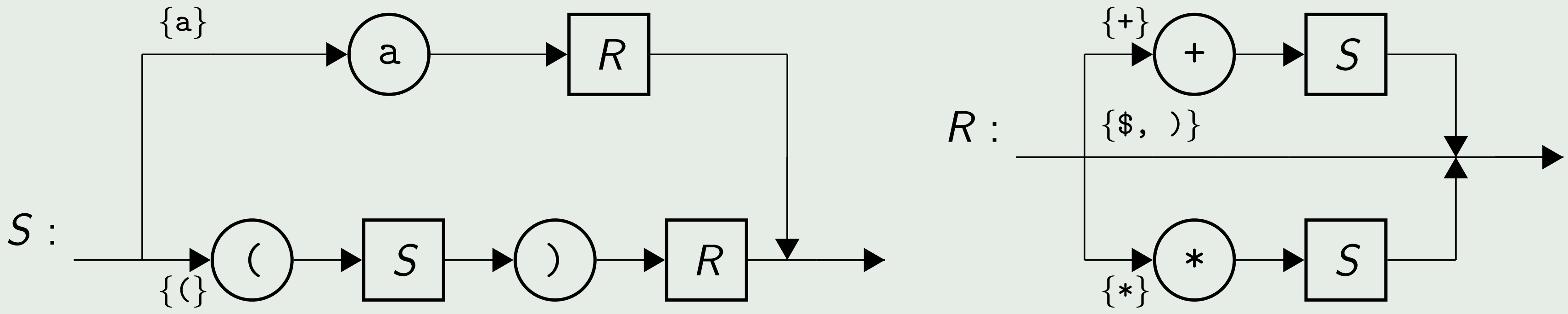
Der Wegweiser für den Ast der Verzweigung α_i in Schritt 2) der Konstruktion des Syntaxgraphen für ein Nonterminal A ist **Erwartet**$(A \to \alpha_i)$.

Bemerkung

Aus LL(1)-Grammatiken konstruierte Syntaxgraphen sind deterministisch.

Beispiel

Die Syntaxgraphen für die Grammatik G_2 sind deterministisch:



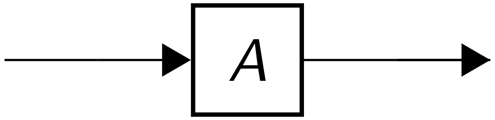
Konstruktion eines Parsers aus deterministischen Syntaxgraphen

- `getToken()` sei ein Unterprogramm, das bei jedem Aufruf das nächste Token aus der Eingabe bestimmt und die Tokenklasse in der globalen Variablen `token` abspeichert.
- Der Tokenwert wird –falls vorhanden– in der globalen Variable `tokenValue` gespeichert.
- `error()` sei eine Fehlerprozedur, die nach Ausgabe einer Fehlermeldung den Parsingprozess abbricht.

Konstruktion eines Parsers aus deterministischen Syntaxgraphen

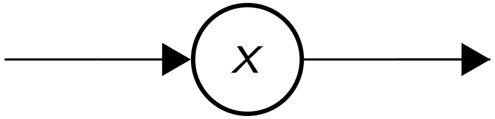
Für jeden erstellten und vereinfachten Syntaxgraphen erzeuge man ein Unterprogramm ohne Parameter gemäß folgender Regeln:

1. Einem Graphen mit Nichtterminal



wird zugeordnet ein Aufruf des Unterprogramms A: `A( );`

2. Einem Graphen mit Terminal

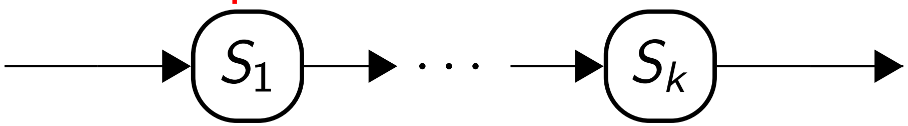


wird das Programmstück

```
if (token == x) getToken();
  else error();
```

zugeordnet.

3. Einer Sequenz

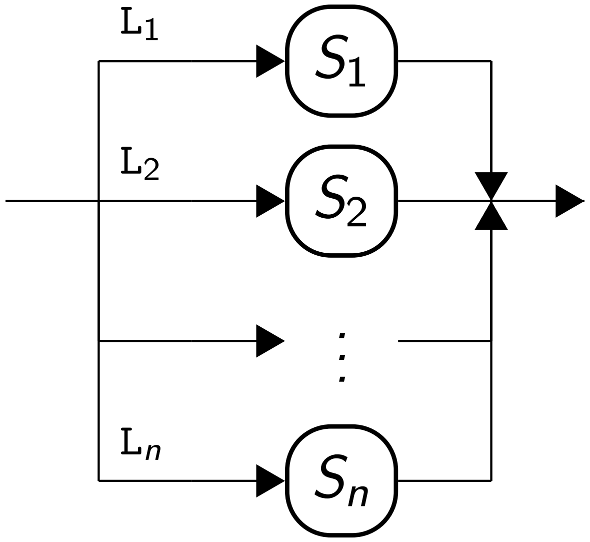


wird folgende Folge von Programmstücken zugeordnet:
`{ P(S1); ... ; P(Sk); }`

wobei `P(Si)` das `Si` zugeordnete Programmstück ist.

Konstruktion eines Parsers aus deterministischen Syntaxgraphen

4. Einer Verzweigung



wird das Programmstück

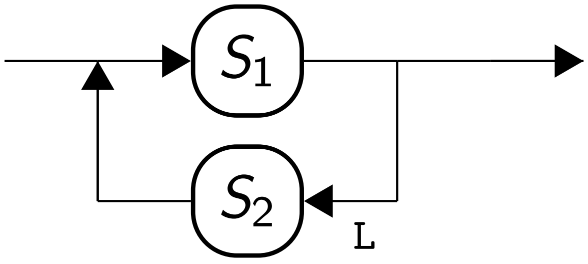
```
switch (token) {
  case in L1: P(S1); break;
     ⋮
  case in Ln: P(Sn); break;
  default: error();
}
```

zugeordnet.

Die $\mathtt{L}_i$ sind dabei die Wegweiser.

Konstruktion eines Parsers aus deterministischen Syntaxgraphen

5. Einer Schleife



wird folgendes Programmstück zugeordnet:

```
while (true) {
   P(S1);
   if (token != L) break;
   P(S2);
}
```

Konstruktion eines Parsers aus deterministischen Syntaxgraphen

Die so erhaltenen Unterprogramme werden vereinfacht (z.B. überflüssige Abfragen werden entfernt).

Der eigentliche Parser hat dann die Form:

```
getToken();
S();                              /* S ist das Startsymbol der Grammatik */
if (token == '$')
   printf("Wort gehoert zur Sprache L(G).\n");
else error();
```

Ein Parser für eine einfache „Programmiersprache“

Als Variablennamen sind nur a, b, . . . , z erlaubt;
als Zahlen treten nur ganze Zahlen auf.

Der Scanner arbeitet folgendermaßen:

- Er erkennt die Schlüsselwörter

`begin`	und liefert dann das Token **begin**,
`end`	und liefert dann das Token **end** und
`print`	und liefert dann das Token **print**.

- Bei Variablen liefert er das Paar (**ident**, index), wobei der Index des *i*-ten Buchstabens im Alphabet *i* ist, und
- bei Zahlen liefert er das Paar (**number**, Zahlenwert).

Ein Parser für eine einfache „Programmiersprache“

Am Ende der Eingabe (durch „$“ markiert) gibt der Scanner das spezielle Token **eof** zurück.

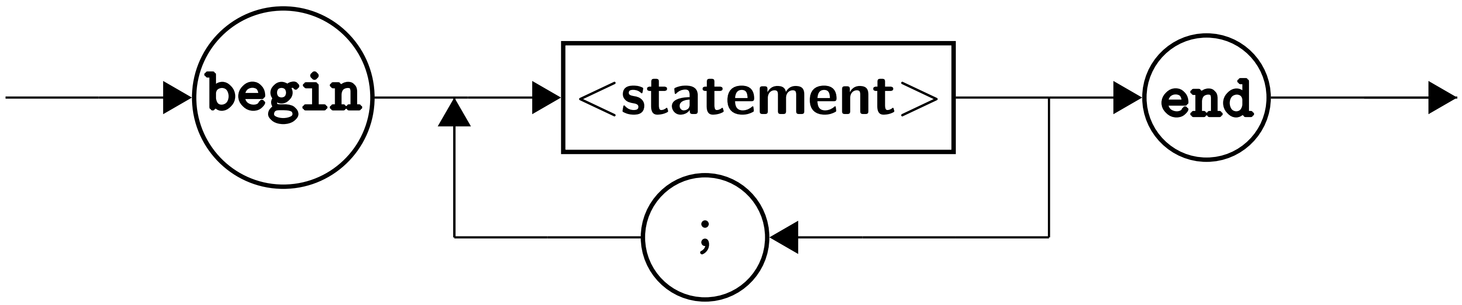
Alle anderen Zeichen werden direkt übergeben. Die Rückgabewerte des Scanners stehen in den Variablen `token` bzw. `tokenValue`.

Bei Erkennen eines Syntaxfehlers wird die Prozedur `error("...")` aufgerufen.

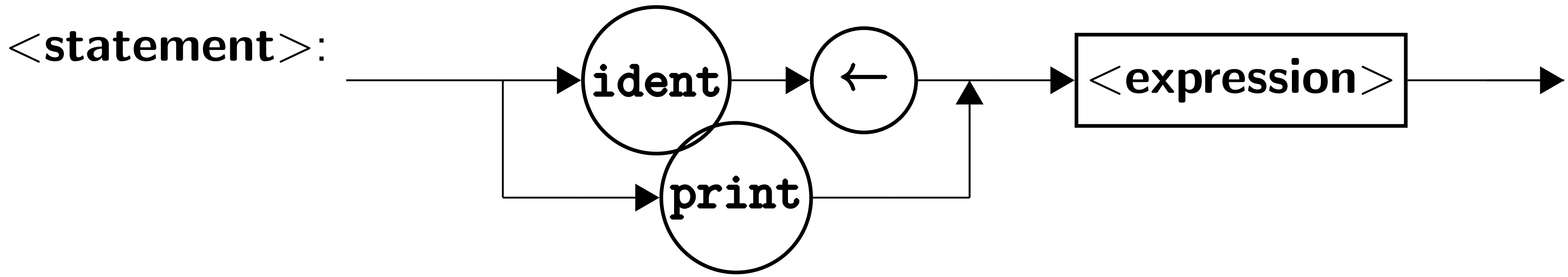
Beispielgrammatik

<program>	::=	**begin** <statementlist> **end**
<statementlist>	::=	<statement> | <statement>**;** <statementlist>
<statement>	::=	**ident** ← <expression> | **print** <expression>
<expression>	::=	<sign><termlist>
<sign>	::=	**+** | **−** | ε
<termlist>	::=	<term> | <term> **+** <termlist> | <term> **−** <termlist>
<term>	::=	<factorlist>
<factorlist>	::=	<factor> | <factor> ***** <factorlist> | <factor> **/** <factorlist>
<factor>	::=	**ident** | **number** | **(** <expression> **)**

<**program**>:

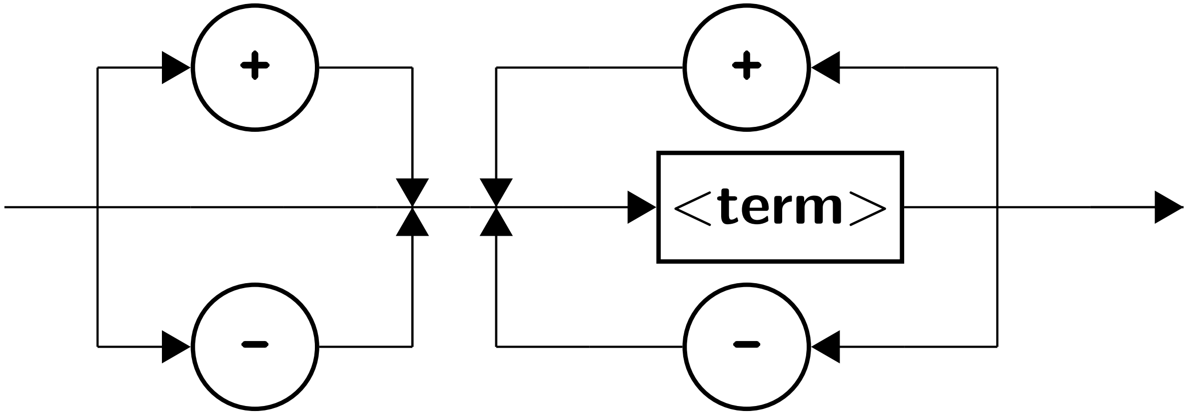



```
void program() {
    if (token != begin) error("program: begin expected");
    do {
        getToken();
        statement();
    } while (token == ';');
    if (token != end) error("program: end expected");
    getToken();
}
```



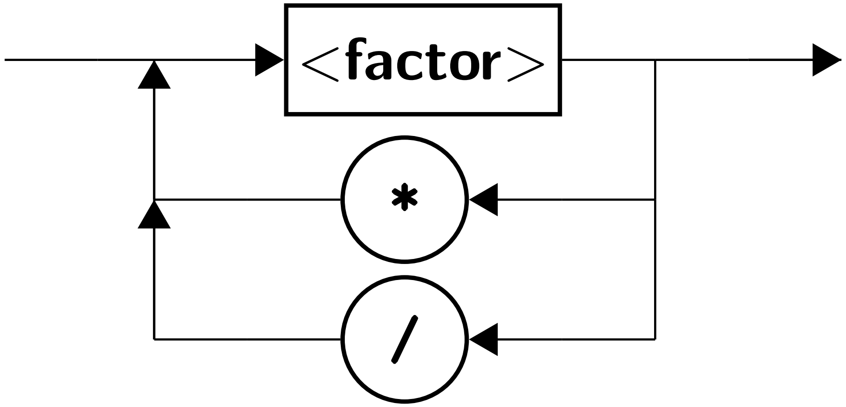

```
void statement() {
    if (token == ident) {
        getToken();
        if (token != '←') error("statement: ← expected");
    }
    else if (token != print) error("statement: print expected");
    getToken();
    expression();
}
```

<**expression**>:



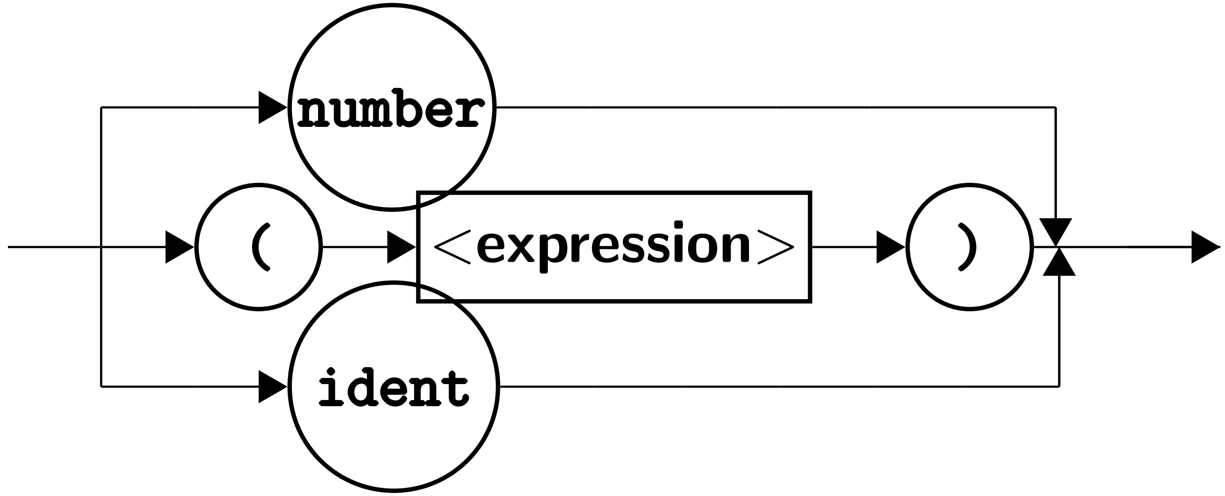

```
void expression() {
    if ((token == '-') || (token == '+')) getToken();
    do {
        term();
        if ((token != '-') && (token != '+')) break;
        getToken();
    } while (true);
}
```

<**term**>:




```
void term() {
    do {
        factor();
        if ((token != '*') && (token != '/')) break;
        getToken();
    } while (true);
}
```

<**factor**>:




```
void factor() {
    switch (token) {
        case ident:
        case number: getToken();
                     break;
        case '(':    getToken();
                     expression();
                     if (token != ')') error("factor: ) expected");
                     getToken();
                     break;
        default:     error ("factor: identifier, number or ( expected");
    }
}
```

Das Hauptprogramm könnte dann folgendermaßen aussehen:

```
void main() {
    getToken();
    program();
    if (token == eof)
        printf("Wort gehoert zur Sprache\n");
    else
        error("main: eof expected");
}
```

Man kann die Parserprozeduren leicht erweitern, so dass die erkannten arithmetischen Ausdrücke gleich während des Parsens ausgewertet werden, und erhält einen **Interpretierer** (Beispiel im Skript).

Linksreduzierende Syntaxanalyse (*bottom-up parsing*)

- Bei der **linksreduzierenden Syntaxanalyse** (*bottom-up parsing*) wird das Eingabewort ebenfalls von links nach rechts gelesen.

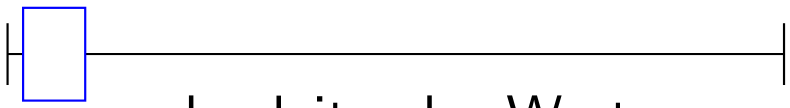
  Es wird jedoch eine **Linksreduktion** erzeugt,
  d.h. der Ableitungsbaum wird von unten nach oben erzeugt.

  Dabei erhält man die Produktionen einer **Rechtsableitung**, allerdings in umgedrehter Reihenfolge.

- Kontextfreie Grammatiken, die mit dieser Methode **deterministisch** behandelt werden können, heißen **LR(k)-Grammatiken**.

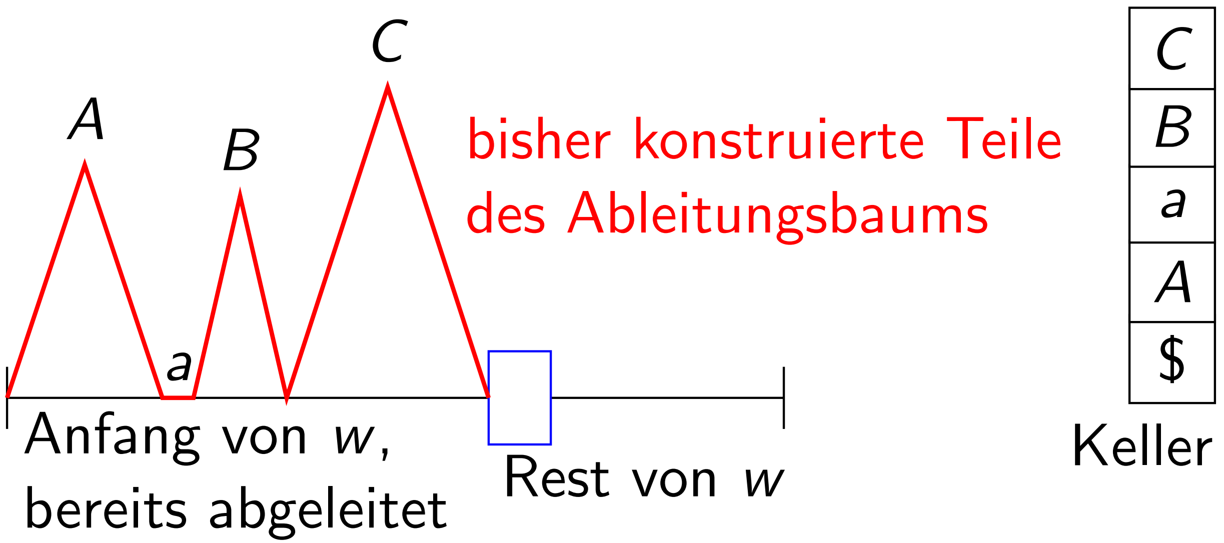
Anfangssituation beim Bottom-Up Parsing

Lesefenster für Vorschau-Symbol,
zeigt zu Anfang auf das erste
Symbol der Eingabe



abzuleitendes Wort *w*

Situation im Bottom-Up Parsing



Reduktionsstelle (Griff)

Wir müssen im abzuleitenden Wort w bzw. im bereits konstruierten Teil des Ableitungsbaumes $\gamma\alpha x$ die **rechte Seite α einer Produktion $A \to \alpha$** identifizieren und **durch die linke Seite A dieser Produktion ersetzen**.

Für eine Rechtsableitung $S \stackrel{*}{\Longrightarrow} \gamma A x \Longrightarrow \gamma\alpha x$ ($\stackrel{*}{\Longrightarrow} w$) und eine Produktion $A \to \alpha$ heißt (die Position von) α die **Reduktionsstelle** oder der **Griff** (*handle*) von $\gamma\alpha x$.

Zu einem Wort einer eindeutigen Grammatik gibt es genau einen Ableitungsbaum (eine Rechtsableitung, eine Linksreduktion), d.h. es gibt in jedem Reduktionsschritt nur eine „richtige“ zu ersetzende rechte Seite.

Diese Reduktionsstelle zu finden, ist nicht immer einfach:
Es ist nicht immer der linkeste reduzierbare Teilstring!

Es ist der linkeste reduzierbare Teilstring, der durch weitere Reduktionsschritte bis zum Startsymbol führt.

Operationen des tabellengesteuerten Bottom-Up Parsers:

- Eine **shift**-Operation schreibt das Vorschau-Symbol auf den Keller und schiebt das Lesefenster um ein Zeichen weiter.
- Eine **reduce**-Operation ist nur anwendbar, wenn die obersten Symbole auf dem Keller die rechte Seite einer Produktion bilden. Diese Symbole werden gelöscht und durch das Symbol auf der linken Seite der Produktion ersetzt.
- Die **accept**-Operation wird ausgeführt, wenn der Parser das Ende der Eingabe erkennt und auf dem Keller nur noch das Startsymbol der Grammatik gefolgt von der Endmarke $ steht.
- Eine **error**-Operation wird ausgeführt, wenn der Parser einen Syntaxfehler in der Eingabe erkennt.

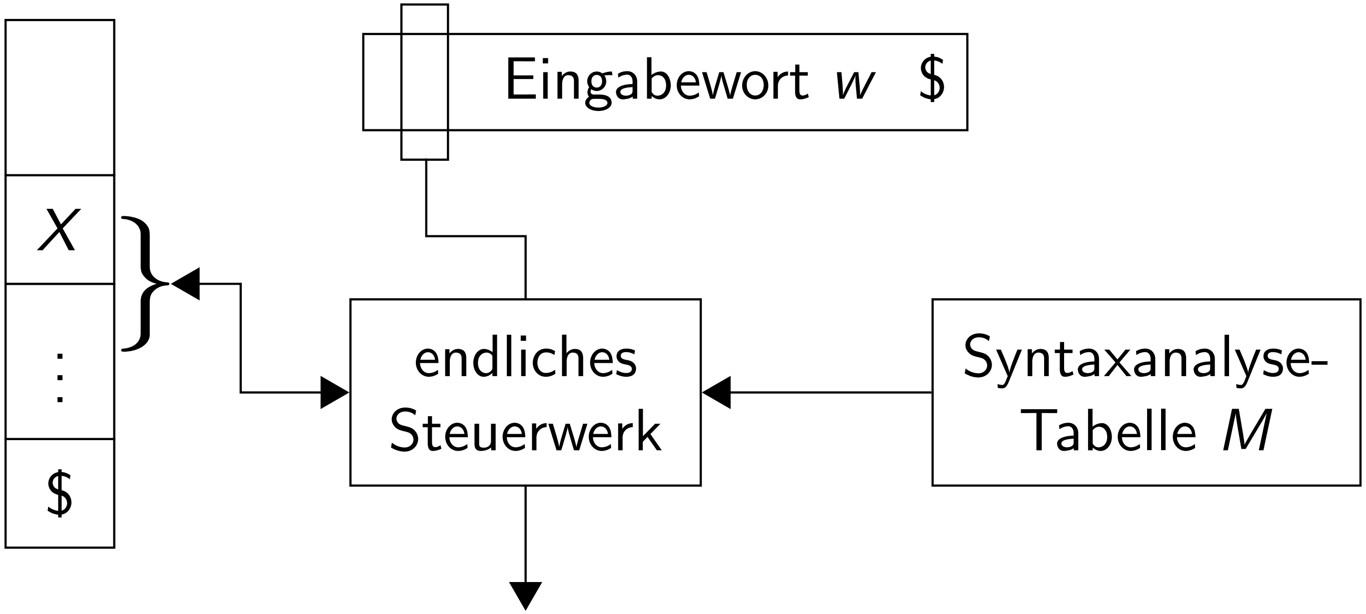
Tabellengesteuerter Bottom-Up-Parser

Auch ein Bottom-up Parser speichert die noch zu bearbeitenden Teile des Ableitungsbaumes im Keller.

Wir benötigen jetzt Zugriff auf *mehrere* Symbole am oberen Ende des Kellers.

Wir steuern den Parser wieder durch eine Syntaxanalyse-Tabelle.

Tabellengesteuerter Bottom-Up-Parser



Ausgabe, z.B. umgedrehte Liste der in einer Rechtsableitung angewendeten Produktionen

Konfigurationen

Die Arbeitsweise eines Bottom-Up-Parsers wird wieder durch **Konfigurationen** dargestellt.

Der momentane Kellerinhalt $\$ \ldots z$ mit dem obersten Kellersymbol z und die restliche Eingabe ab dem Vorschau-Symbol a wird wieder als Wortpaar $[\$ \ldots z, \quad a \ldots \$]$ notiert.

Bemerkung

Der Kellerinhalt wird hier um 90 Grad im Uhrzeigersinn gedreht; das oberste Kellerelement (der aktuelle Zustand) steht rechts.

Beispiel (Grammatik für arithmetische Ausdrücke: $G_4 = (N, T', P, E)$)

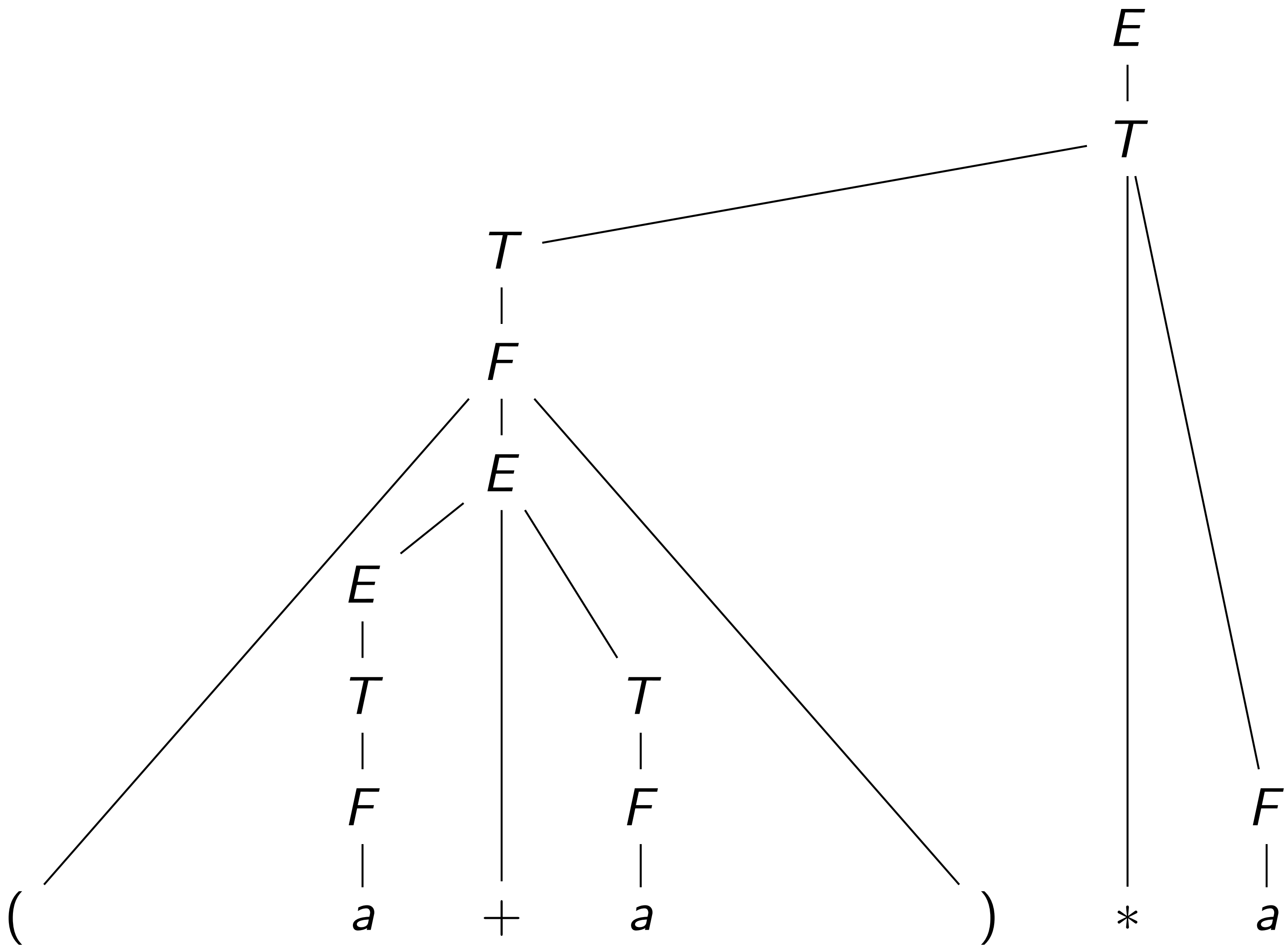
$N = \{E, T, F\}$, $T' = \{a, +, *, (,)\}$ und P enthalte die Produktionen:

$$\underset{(1)}{E \to E + T} \mid \underset{(2)}{T} \qquad \underset{(3)}{T \to T * F} \mid \underset{(4)}{F} \qquad \underset{(5)}{F \to (E)} \mid \underset{(6)}{a}$$

Linksrekursive Produktionen stören beim Bottom-up-Parsen nicht!

Betrachte das Wort $(a + a) * a$.

(eindeutiger) Syntaxbaum für $(a + a) * a$ bzgl. G_4



Rechtsableitung und Linksreduktion

Beispiel (Grammatik für arithmetische Ausdrücke: $G_4 = (N, T', P, E)$)

$$E \to \underset{(1)}{E + T} \mid \underset{(2)}{T} \qquad T \to \underset{(3)}{T * F} \mid \underset{(4)}{F} \qquad F \to \underset{(5)}{(E)} \mid \underset{(6)}{a}$$

Rechtsableitung:

$$E \overset{(2)}{\Longrightarrow} T \overset{(3)}{\Longrightarrow} T * F \overset{(6)}{\Longrightarrow} T * a \overset{(4)}{\Longrightarrow} F * a \overset{(5)}{\Longrightarrow} (E) * a \overset{(1)}{\Longrightarrow} (E + T) * a \overset{(4)}{\Longrightarrow}$$
$$(E + F) * a \overset{(6)}{\Longrightarrow} (E + a) * a \overset{(2)}{\Longrightarrow} (T + a) * a \overset{(4)}{\Longrightarrow} (F + a) * a \overset{(6)}{\Longrightarrow} (a + a) * a$$

Linksreduktion:

$$(a+a)*a \overset{(6)}{\Longleftarrow} (F+a)*a \overset{(4)}{\Longleftarrow} (T+a)*a \overset{(2)}{\Longleftarrow} (E+a)*a \overset{(6)}{\Longleftarrow} (E+F)*a \overset{(4)}{\Longleftarrow}$$
$$(E + T) * a \overset{(1)}{\Longleftarrow} (E) * a \overset{(5)}{\Longleftarrow} F * a \overset{(4)}{\Longleftarrow} T * a \overset{(6)}{\Longleftarrow} T * F \overset{(3)}{\Longleftarrow} T \overset{(2)}{\Longleftarrow} E$$

Arbeitsweise von kfr. Grammatik und Kellerautomat

kfr. Grammatik	Kellerautomat		
$\downarrow$ Linksreduktion	Keller	, Resteingabe	Operation
	$\$$	, $(a+a)*a\$$	shift (
	$\$($	, $a+a)*a\$$	shift a
$(a+a)*a \overset{(6)}{\Longleftarrow}$	$\$(a$	, $+a)*a\$$	reduce $F \rightarrow a$
$(F+a)*a \overset{(4)}{\Longleftarrow}$	$\$(F$	, $+a)*a\$$	reduce $T \rightarrow F$
$(T+a)*a \overset{(2)}{\Longleftarrow}$	$\$(T$	, $+a)*a\$$	reduce $E \rightarrow T$
	$\$(E$	, $+a)*a\$$	shift $+$
	$\$(E+$	, $a)*a\$$	shift a
$(E+a)*a \overset{(6)}{\Longleftarrow}$	$\$(E+a$	, $)*a\$$	reduce $F \rightarrow a$
$(E+F)*a \overset{(4)}{\Longleftarrow}$	$\$(E+F$	, $)*a\$$	reduce $T \rightarrow F$
$(E+T)*a \overset{(1)}{\Longleftarrow}$	$\$(E+T$	, $)*a\$$	reduce $E \rightarrow E+T$
$\uparrow$ Rechtsableitung	Keller	, Resteingabe	Operation

Arbeitsweise von kfr. Grammatik und Kellerautomat

kfr. Grammatik		Kellerautomat		
$\downarrow$ Linksreduktion		Keller	, Resteingabe	Operation
		$\$(E$	, $)*a\$$	shift)
$(E)*a$	$\stackrel{(5)}{\Longleftarrow}$	$\$(E)$	, $*a\$$	reduce $F \to (E)$
$F*a$	$\stackrel{(4)}{\Longleftarrow}$	$\$F$	, $*a\$$	reduce $T \to F$
		$\$T$	, $*a\$$	shift $*$
		$\$T*$	, $a\$$	shift a
$T*a$	$\stackrel{(6)}{\Longleftarrow}$	$\$T*a$	, $\$$	reduce $F \to a$
$T*F$	$\stackrel{(3)}{\Longleftarrow}$	$\$T*F$	, $\$$	reduce $T \to T*F$
T	$\stackrel{(2)}{\Longleftarrow}$	$\$T$	, $\$$	reduce $E \to T$
E		$\$E$	, $\$$	accept
$\uparrow$ Rechtsableitung		Keller	, Resteingabe	Operation

Der Kellerautomat zeigt –wie die Grammatik– jeden Reduktionsschritt, darüberhinaus aber auch jede shift-Operation.

Im 9. Reduktionsschritt (rot) wird nicht mit der Produktion $E \to T$ reduziert, da dieser Schritt in eine **Sackgasse**

$$T * a \Leftarrow E * a \Leftarrow E * F \Leftarrow E * T$$

führen würde, also nicht bis zum Startsymbol der Grammatik verlängert werden kann!

Natürlich soll der Bottom-up-Parser **deterministisch** arbeiten, also auch Sackgassen vermeiden.

Parser-Generatoren und LR(1)-Grammatiken

Kontextfreie Grammatiken, deren Bottom-up Parser deterministisch ist, heißen **LR(k)-Grammatiken**.

Die theoretischen Grundlagen zum Bottom-up Parsen sind komplizierter als beim Top-down-Parsen.
Viele **Parser-Generatoren** arbeiten jedoch mit dieser Methode, da viel mehr Sprachen durch LR(1)-Grammatiken als durch LL(1)-Grammatiken beschreibbar sind.

Syntaxanalyse-Tabellen zu LR(1)-Grammatiken werden jedoch sehr groß.
Parser-Generatoren beschränken sich daher oft –wie `yacc` und `Bison`– auf die Teilklasse der **LALR(1)-Grammatiken**.

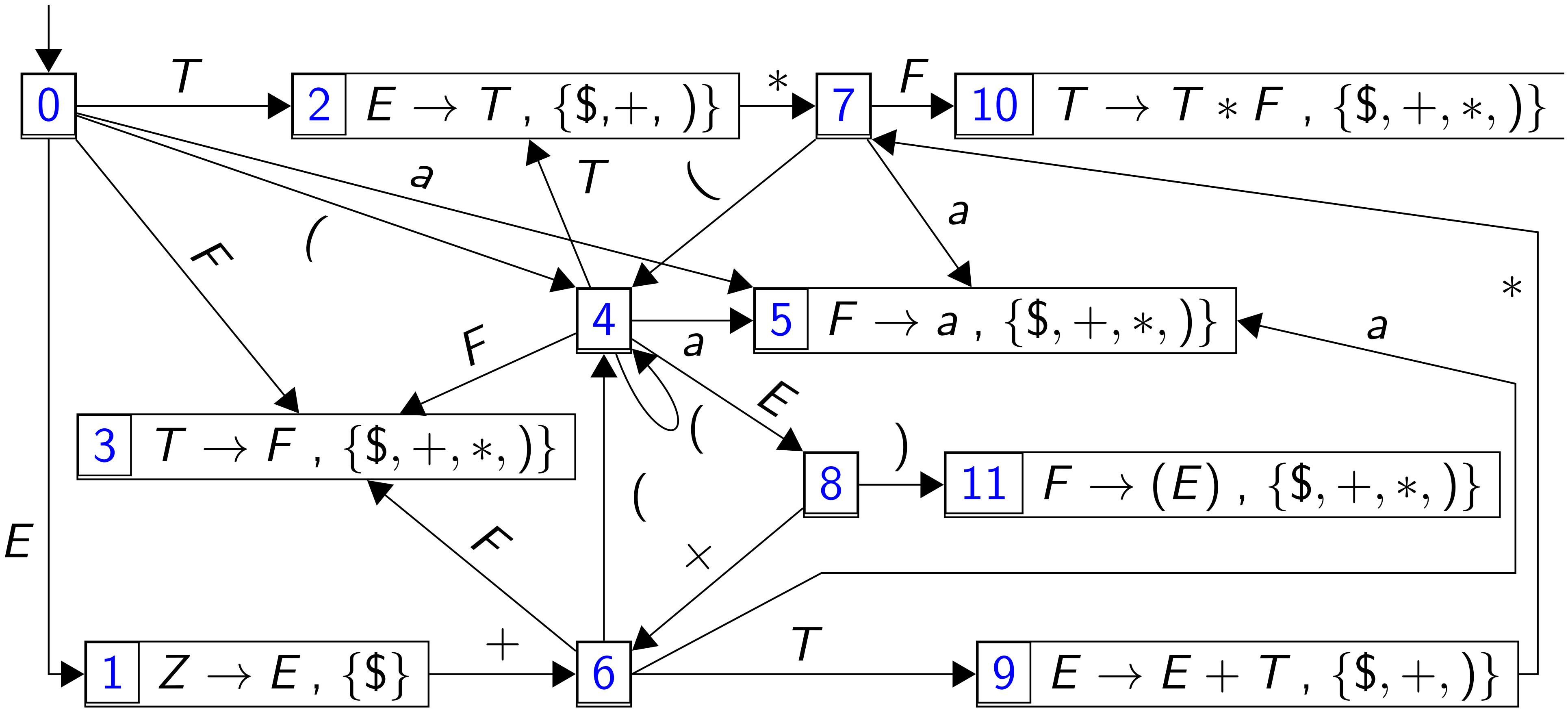
Parser-Generatoren und LR(1)-Grammatiken

Die **Menge der Kellerbelegungen** aller erfolgreichen Reduktionen bilden eine reguläre Sprache.

Für diese Sprache konstruiert ein Parser-Generator einen **deterministischen endlichen Automaten**, z.B.

für die Grammatik $G_4 = (N, T', P, E)$ der arithmetischen Ausdrücke mit
$N = \{E, T, F\}$, $T' = \{a, +, *, (,)\}$ und
$P = \{E \to E + T \mid T, \quad T \to T * F \mid F, \quad F \to (E) \mid a\}$

deterministischer endlicher Automat für G_4

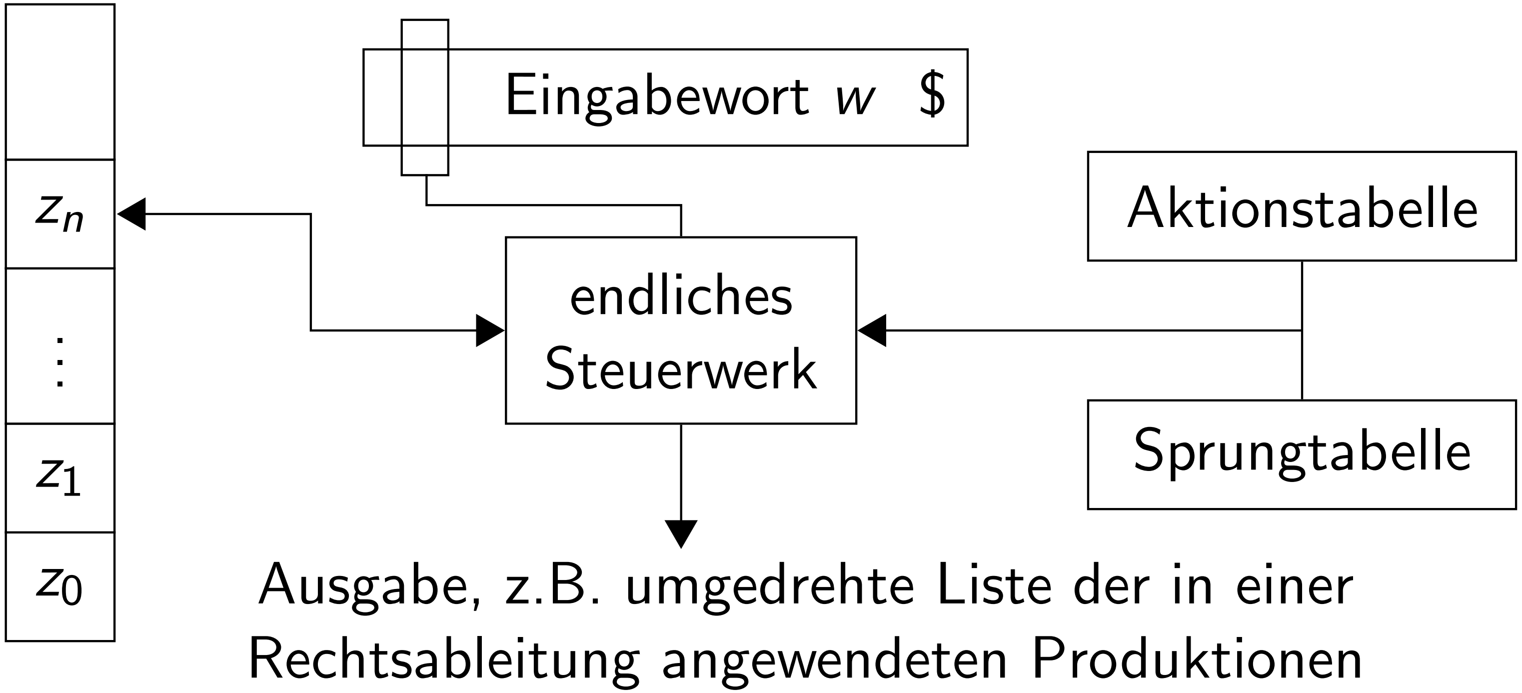


Kommt man in einen Zustand mit einer Produktion und liegt das Vorschausymbol in der angegebenen Menge, so reduziert man mit dieser Produktion.
Sonst führt man eine shift-Operation aus und das eingelesene/bearbeitete Symbol bestimmt den Übergang in den Nachfolgezustand.

Damit der Automat nach einer reduce-Operation nicht wieder über den bisherigen Kellerinhalt laufen muss, merkt man sich nach jeder Operation den erreichten Zustand – auf dem Keller.
Der Keller enthält also statt der Grammatiksymbole jetzt Zustände des endlichen Automaten.

Aus diesem endlichen Automaten leitet ein Parser-Generator die Syntaxanalyse-Tabelle ab, die jetzt aus zwei Teilen besteht, der **Aktionstabelle** und der **Sprungtabelle**.

Aufbau eines deterministischen LR-Parsers



Die z_i sind Zustände des endlichen Automaten.

Aufbau der Syntaxanalyse-Tabellen

- Die Zeilen beider Tabellen sind mit Zuständen markiert.
- Die Spalten der Aktionstabelle sind mit den Vorschausymbolen, also den terminalen Symbolen der Grammatik und der Endmarke $, markiert.
- Die Spalten der Sprungtabelle sind mit den nichtterminalen Symbolen der Grammatik markiert.

Die Produktionen der Grammatik sowie die Zustände des endlichen Automaten werden durchnummeriert. Der Startzustand ist 0.

Einträge in der Aktions- und Sprung-Tabelle

Ein Eintrag der **Aktionstabelle** (*action table*) hat eine der Formen:

si	(**shift** <neuer Zustand>) oder
ri	(**reduce** <Produktion>) oder
acc	(**accept**).

Ein Eintrag in der **Sprungtabelle** (*goto table*) hat die Form:

i	(<neuer Zustand>)

Freie Felder in den Tabellen bezeichnen Fehlerzustände.

Wie arbeitet der tabellengesteuerte LR-Parser?

Initiale Situation

Das Lesefenster befindet sich auf dem ersten Zeichen des Eingabeworts, das mit der Endmarke $ abgeschlossen ist.
Im Keller befindet sich nur der Startzustand 0.

Wie arbeitet der tabellengesteuerte LR-Parser?

Arbeitsschritt
Sei z der Zustand oben auf dem Keller und sei a das Vorschau-Symbol.

Ist aktion$[z, a] =$

- shift z', dann schreibe den neuen Zustand z' auf den Keller. Das Lesefenster wandert ein Symbol nach rechts.
- reduce $A \to \alpha$, dann lösche die obersten $|\alpha|$ Einträge vom Keller. Sei danach z' der Zustand oben auf dem Keller. Schreibe den Zustand sprung$[z', A]$ auf den Keller. Gib die Produktion $A \to \alpha$ aus.
- accept, dann beende das Parsing. Das Eingabewort gehört zur Sprache.
- error (leerer Eintrag in der Aktionstabelle), dann beende das Parsing mit einer Fehlermeldung. Das Eingabewort gehört nicht zur Sprache.

Beispiel (Grammatik für einfache arithmetische Ausdrücke)

Produktionen: $E \to E + T \mid T, \quad T \to T * F \mid F, \quad F \to (E) \mid a$

Ein Parser-Generator liefert dafür folgende Tabellen:

	aktion						sprung		
Zustand	a	$+$	$*$	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			