6 Formale Sprachen und Parser

Eine wichtige Klasse von Algorithmen in der Informatik befasst sich damit, Texte zu durchsuchen, zu erkennen und zu analysieren. Die Überprüfung, ob ein Text einer bestimmten – häufig in Form einer formalen Grammatik festgelegten – Struktur entspricht, bezeichnet man als *Syntaxanalyse* oder synonym als *Parsing*. Diese Algorithmen werden beispielsweise im Umfeld des sog. Data Mining oder im Compilerbau zur formalen Analyse von Programmtexten eingesetzt. Ein Compiler übersetzt ein Programm einer höheren Programmiersprache auf Basis seiner formalen Struktur (die oft in Form eines Syntaxbaums repräsentiert wird) in Maschinensprache, also derjenigen Sprache, die direkt vom Prozessor eines Computers verstanden werden kann.

In Abschnitt 6.1 beschäftigen wir uns mit den Grundlagen formaler Syntaxbeschreibungen: mit formalen Sprachen und Grammatiken, den mathematischen Pendants der „natürlichen“ Sprachen und Grammatiken. Besonders interessant für uns sind die sog. Typ-2-Sprachen und die in gewissem Sinne weniger komplexen Typ-3-Sprachen. Abschnitt 6.2 beschreibt die Repräsentation von Grammatiken in Python und zeigt die Implementierung einiger grundlegender Funktionen auf den Nichtterminalen von Grammatiken, nämlich FIRST und FOLLOW; diese werden in den darauffolgenden Abschnitten benötigt.

Die folgenden Abschnitte 6.3 und 6.4 beschreiben die in der Praxis am häufigsten verwendeten Algorithmen zum Erkennen und Analysieren von Programmiersprachen: Zum Einen prädiktive Parser, insbesondere Recursive-Descent-Parser, in Abschnitt 6.3; zum Anderen LR-Parser in Abschnitt 6.4 wie sie etwa in Parsergeneratoren wie Yacc zum Einsatz kommen.

6.1 Formale Sprachen und Grammatiken

6.1.1 Formales Alphabet, formale Sprache

Wir benötigen im restlichen Kapitel die folgenden Definitionen:

- Ein (*formales*) *Alphabet* A ist eine nicht-leere endliche Menge. Folgende Mengen sind beispielsweise Alphabete:
  $A_1 = \{a, b, \ldots, z\}$, $A_2 = \{0, 1\}$, $A_3 = \{\textbf{if}, \textbf{then}, \textbf{begin}, \textbf{end}, \mathit{stmt}, \mathit{ausdr}\}$
- Das *leere Wort*, das aus keinen Buchstaben besteht, wird als ε bezeichnet.
- Ein *Buchstabe* ist ein Element eines Alphabets. Beispiele: 0 ist also ein Buchstabe aus A_2; **then** ist ein Buchstabe aus A_3.

- Ein *Wort* entsteht durch Hintereinanderschreiben mehrerer Buchstaben eines Alphabets. Beispiele: *aabax* ist ein Wort über dem Alphabet A_1; 010001 ist ein Wort über dem Alphabet A_2.

Folgende Operatoren auf Wörtern und Alphabeten sind relevant:

- Sei w ein Wort; $|w|$ ist die Anzahl der Buchstaben in w. Beispiele: $|001| = 3$, $|\varepsilon| = 0$, $|\textbf{if } ausdr \textbf{ then } stmt| = 4$.
- Sei A ein Alphabet. Dann ist A^* die Menge aller Wörter mit Buchstaben aus A. Es gilt immer auch $\varepsilon \in A^*$. Beispiel:

$$\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots\}$$

- Gilt $L \subseteq A^*$, so nennt man L auch *Sprache* über dem Alphabet A.
- Ist $w \in A^*$ ein Wort über dem Alphabet A. Dann ist w^n das Wort, das durch n-maliges Hintereinanderschreiben des Wortes w entsteht. Offensichtlich gilt $|w^n| = n \cdot |w|$.

Aufgabe 6.1

Geben Sie den Wert der folgenden Ausdrücke an:

(a) $\{\varepsilon\}^*$ (b) $|\{\ w \in \{a, b, c\}^* \mid |w| = 2\ \}|$ (c) $|\{0, 1\}^*|$

6.1.2 Grammatik, Ableitung, akzeptierte Sprache, Syntaxbaum

Eine formale (Typ-2-)Grammatik[1] den allgemeinsten Typ-0-Grammatiken$G$ besteht aus vier Komponenten, mathematisch beschrieben als 4-Tupel (S, T, V, P), wobei

- T die Menge der sog. *Terminalsymbole* ist,
- V die Menge der sog. *Nichtterminalsymbole* ist, (manchmal auch *Variablen* oder *Metasymbole* genannt)
- $S \in V$ das Startsymbol ist,

[1] Tatsächlich kann man eine ganze Hierarchie von Grammatik-Typen definieren, die über die Form der jeweils zugelassenen Produktionen definiert werden kann. Bei Typ-0-Grammatiken unterliegen die Produktionen keinerlei Einschränkungen: Linke und rechte Seite der Produktionen dürfen beliebige Zeichenfolgen aus $V \cup T$ sein. Bei Typ-1-Grammatiken darf die rechte Seite einer Produktion nicht kürzer sein als die linke Seite (ausgenommen sind Produktionen, deren rechte Seite ε ist). Bei Typ-2-Grammatiken darf die Linke Seite jeder Regel aus nur einer Variablen bestehen und bei Typ-3-Grammatiken gibt es zusätzliche Einschränkungen für die rechte Seite.

- $P \subseteq V \times (T \cup V)^*$ die Menge der sog. Produktionen ist; Produktionen sind also Tupel, deren erste Komponente ein Element aus V und deren zweite Komponente eine Sequenz von Elementen aus $T \cup V$ ist.

Die Elemente von P sind mathematisch zwar als Tupel (siehe Anhang B) modelliert, die beiden Tupel-Komponenten werden jedoch i. A. mit einem „$\rightarrow$“ als Trenner notiert; für $(A, abA) \in P$ schreibt man also üblicherweise $A \rightarrow abA \ \in P$.

Beispiel 6.1: *Grammatik*

Die Grammatik $G = (S, \{ausdr, ziffer\}, \{\texttt{+}, \texttt{-}, \texttt{0}, \ldots, \texttt{9}\}, P)$ mit

$$\begin{aligned} P = \{ \quad & ausdr \rightarrow ausdr\ \texttt{+}\ ausdr \\ & ausdr \rightarrow ausdr\ \texttt{-}\ ausdr \\ & ausdr \rightarrow ziffer \\ & ziffer \rightarrow \texttt{0} \\ & \ldots \rightarrow \ldots \\ & ziffer \rightarrow \texttt{9} \quad \} \end{aligned}$$

beschreibt einfache arithmetische Ausdrücke.

Ableitung. Informell ausgedrückt, ist die „Bedeutung“ einer Produktion $A \rightarrow \alpha$ mit $A \in V$ und $\alpha \in (V \cup T)^*$ die, dass man jedes Vorkommen von A in einem Wort $w \in (V \cup T)^*$ durch die rechte Seite der Produktion α ersetzen darf. Dies wird durch den Begriff des *Ableitungsschritts* in Form der Relation „$\Rightarrow$“ zum Ausdruck gebracht. Es gilt:

$$\begin{aligned} x \Rightarrow y \text{ gdw. } & \exists \beta, \gamma \in (V \cup T)^*, \text{mit } x = \beta A \gamma,\ y = \beta \alpha \gamma \\ & \text{und } A \rightarrow \alpha \in P \end{aligned} \tag{6.1}$$

Der Begriff der *Ableitung* wird durch die transitive Hülle (siehe Abschnitt B.1.3 für eine Definition des Begriffs der transitiven Hülle) von $\Rightarrow$ modelliert, d. h. durch die „kleinste“ transitive Relation, in der $\Rightarrow$ enthalten ist. Die transitive Hülle von „$\Rightarrow$“ schreibt man als „$\stackrel{*}{\Rightarrow}$“. Man kann die Relation „$\stackrel{*}{\Rightarrow}$“ auch direkt folgendermaßen definieren:

$$\begin{aligned} x \stackrel{*}{\Rightarrow} y \text{ gdw. } & x = y \text{ oder } x \Rightarrow y \\ & \text{oder } \exists w_0, \ldots, w_n \colon x \Rightarrow w_0 \Rightarrow \ldots \Rightarrow w_n \Rightarrow y \end{aligned} \tag{6.2}$$

Die durch eine Grammatik $G = (S, T, V, P)$ *erzeugte Sprache* $L(G)$ ist folgendermaßen definiert:

$$L(G) := \{w \in T^* \mid S \stackrel{*}{\Rightarrow} w\}$$

Die Sprache $L(G)$ besteht also aus allen Wörtern (d. h. Folgen von Terminalzeichen, d. h. Elementen aus T^*), die aus der Startvariablen S ableitbar sind.

Da es sich bei den in diesem Abschnitt behandelten Grammatiken eigentlich um sog. Typ-2-Grammatiken handelt, nennen wir gelegentlich auch eine durch eine solche Grammatik erzeubare Sprache eine *Typ-2-Sprache*.

Ein aus sowohl Terminalen als auch Nichtterminalen bestehende Zeichenfolge, die in einem Zwischenschritt einer Ableitung auftaucht, nennt man *Satzform*.

Beispiel 6.2: *Ableitungsschritt, Ableitung, Sprache*

Sei G die in Beispiel 6.1 definierte Grammatik. Dann gelten beispielsweise folgende Aussagen:

ziffer $\Rightarrow$ `0`	denn:	mit $\beta, \gamma = \varepsilon$ und $\alpha =$ `0` und $A =$ *ziffer* gilt Voraussetzung aus Definition 6.1.
ausdr `+` *ausdr* $\Rightarrow^*$ *ziffer* `+ 9`	denn:	es gilt *ausdr* `+` *ausdr* $\Rightarrow$ *ziffer* `+` *ausdr* $\Rightarrow$ *ziffer* `+` *ziffer* $\Rightarrow$ *ziffer* `+ 9`.
`9 + 4 - 2` $\in L(G)$	denn:	Das Wort lässt sich aus dem Startsymbol *ausdr* ableiten, d. h. *ausdr* $\Rightarrow^*$ `9 + 4 - 2` und das Wort besteht nur aus Terminalsymbolen, d. h. `9 + 4 - 2` $\in T^*$.
`9 -` *ziffer* $\notin L(G)$	denn:	Es gilt zwar *ausdr* $\Rightarrow^*$ `9 -` *ziffer* aber `9 -` *ziffer* $\notin T^*$.

Syntaxbäume. Ein Syntaxbaum für ein Wort $w \in L(G)$ ist ein Baum, dessen innere Knoten mit Nichtterminalen beschriftet sind, dessen Blätter mit Buchstaben aus w beschriftet sind, dessen Wurzel mit dem Startsymbol der Grammatik beschriftet ist und jeder der inneren Knoten in folgender Weise einer Produktion $A \to x_0 \dots x_n$ (mit $x_i \in V \cup T$) der Grammatik entspricht: Der innere Knoten ist mit „A“ beschriftet und die Kinder sind in der Reihenfolge von links nach rechts mit jeweils $x_0, \dots, x_n$ beschriftet. Abbildung 6.1 zeigt für die Grammatik aus Beispiel 6.1 einen Syntaxbaum für das Wort `9 + 4 - 3`.

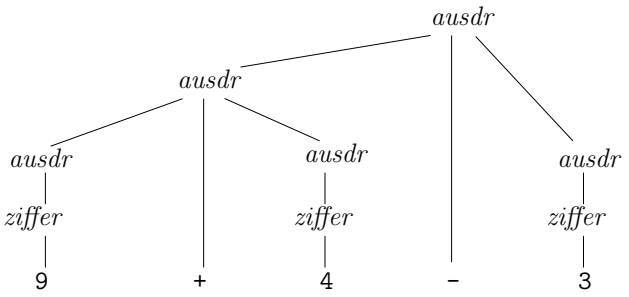


Abb. 6.1: *Ein Syntaxbaum für das Wort `9 + 4 - 3`.*

Eine Grammatik G heißt *mehrdeutig*, falls es ein Wort $w \in L(G)$ gibt, für die es zwei verschiedene Syntaxbäume gibt. Die Grammatik aus Beispiel 6.1 ist beispielsweise mehrdeutig (siehe auch Aufgabe 6.2).

Aufgabe 6.2

Für das Wort `9 + 4 - 3` gibt es neben dem in Abbildung 6.1 abgebildeten Syntaxbaum noch einen weiteren Syntaxbaum. Zeichnen Sie diesen auf.

Beispiel 6.3: *Grammatik für verschachtelte Listen*

Wir beschreiben eine Grammatik G_{Liste}, die alle gültigen möglicherweise verschachtelten Python-Ziffernlisten erzeugt; also folgende Wörter sollten beispielsweise in $L(G_{Liste})$ enthalten sein:

`[]`, `[1,5,2,6]`, `[1,[[2]],[9,2],[],[[]],[0]]`

Die folgende Grammatik $G_{Liste} = (S, V, T, P)$ mit

$$\begin{aligned} S &= \mathit{Liste}\ , \\ V &= \{\mathit{Liste}, \mathit{elemente}, \mathit{element}, \mathit{ziffer}\}\ , \\ T &= \{\texttt{,}, \texttt{[}, \texttt{]}, \texttt{0}, \ldots, \texttt{9}\} \end{aligned}$$

und einer Menge P, bestehend aus den folgenden Produktionen, beschreibt eine solche Sprache:

$$\begin{aligned} \mathit{Liste} &\to \texttt{[}\ \mathit{elemente}\ \texttt{]} \mid \texttt{[ ]} \\ \mathit{elemente} &\to \mathit{element} \mid \mathit{element}\ \texttt{,}\ \mathit{elemente} \\ \mathit{element} &\to \mathit{Liste} \mid \mathit{ziffer} \\ \mathit{ziffer} &\to \texttt{0} \mid \ldots \mid \texttt{9} \end{aligned}$$

Die erste Produktion beschreibt eine Liste als entweder zwischen den Terminalen `[` und `]` eingeschlossene Wörter, die durch das Nichtterminal *elemente* erzeugt werden oder als das Wort „`[ ]`“. Die zweite Produktion beschreibt das Nichtterminal *elemente*: Dieses ist entweder ein einzelnes Element, beschrieben durch das Nichtterminal *element*, oder eine durch Kommata getrennte Liste von Elementen. Man beachte, dass das Nichtterminal *elemente* rekursiv definiert ist; zum Verständnis hilft auch hier das in Abschnitt 1.2.1 beschriebene Denk„rezept“ für die Erstellung rekursiver Funktionen: Ausgehend von der Annahme, das Nichtterminal *elemente* auf der *rechten* Seite der Produktion erzeugt die gewünschten Wörter, so müssen wir die Produktionen so wählen, dass unter dieser Annahme die gewünschten Wörter erzeugt werden können.

Die Produktionen für das Nichtterminal *element* beschreiben die Tatsache, dass ein einzelnes Element schließlich wiederum eine vollständige Liste ist (auch hier gehen wir gemäß des eben schon erwähnten Denkrezepts davon aus, dass durch das Nichtterminal *Liste* auf der rechten Seite alle wohlgeformten Listen erzeugt werden können) oder eine einzelne Ziffer.

Das Wort `[ 1 , [ 5 , 1 ] , [ ] ]` hat beispielsweise den in Abbildung 6.2 gezeigten Syntaxbaum.

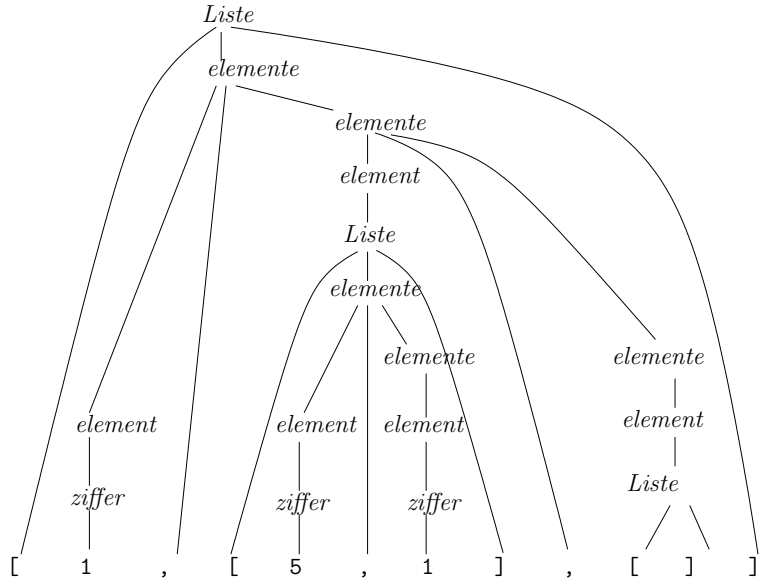


Abb. 6.2: *Syntaxbaum des Wortes* `[ 1 , [ 5 , 1 ] , [ ] ]` .

Aufgabe 6.3

Erweitern Sie die Grammatik so, dass alle (möglicherweise geschachtelte) Ziffer-Tupellisten (in Python-Notation) erzeugt werden. Folgende Wörter sollten beispielsweise in der durch die Grammatik erzeugten Sprache enthalten sein:

```
([1,(1,2)],(2,),[2],[],())  ([1],)  [1,2]  (1,[2])
```

Beachten Sie, dass ein-elementige Tupel mit einem Komma am Ende notiert werden.

Aufgabe 6.4

Zeichnen Sie den Syntaxbaum für das Wort `[ [ [ 1 , [ ] ] ] , 1 ]`.

6.2 Repräsentation einer Grammatik in Python

Wir repräsentieren eine Grammatik folgendermaßen als Python-Klasse:

```
1 class Grammatik(object):
2   def __init__ ( self ,S,V,T,P=[]):
3     if '$' not in T: T.append('$')
4     if S not in V: V.append(S)
5     self.S = S ; self.V = V ; self.T = T ; self.P = []
6     for p in P: self._addP(p)
```

```
7
8   def _addP(self, s):
9       (l, _, r) = s.partition('->')
10      l = l.split()[0] ; r = r.split()
11      assert all([x in self.V+self.T for x in [l]+r]):
12      self.P.append((l,r))
```

Wir gehen davon aus, dass V und T jeweils Stringlisten sind und P eine Liste von Tupeln darstellt, deren erste Komponente die jeweilige linke Seite eine Produktion und deren zweite Komponente die rechte Seite einer Produktion in Form einer Stringliste enthält. Die Anweisungen in den Zeilen 3 und 4 stellen sicher, dass sich die Startvariable S auch tatsächlich in der Variablenmenge V befindet und dass sich das Endesymbol `'$'` auch tatsächlich in der Menge der Terminalsymbol befindet – wir gehen nämlich (aus praktischen Gründen) davon aus, dass jede Eingabe mit dem Endezeichen `'$'` abschließt. In den Zeilen 5 und 6 werden die Objektattribute S, V und T gesetzt.

In Zeile 6 werden schließlich die Produktionen dem Objektattribut P hinzugefügt. Dies erfolgt über die interne Methode *_addP*, die es erlaubt, eine Produktion nach dem Schema „*linkeSeite*`'->'`*rechteSeite*“ zu übergeben. Mittels *s.partition*(`'->'`) wird die linke Seite l und die rechte Seite r getrennt. Mittels *l.split*() bzw. *r.split*() werden anschließend die einzelnen Symbole getrennt. Damit eine Trennung der einzelnen Grammatiksymbole mittels *split* funktioniert, sollten Terminale und Nichtterminale immer über Leerzeichen getrennt übergeben werden.

Beispiel 6.4

Wir können die Grammatik $G = (D, \{D, E, T, F\}, \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \textbf{id}\}, P)$ mit

$$\begin{aligned} P = \{ \quad & D \rightarrow E \mid E \texttt{ + } T \mid T \\ & T \rightarrow T \texttt{ * } F \mid F \\ & F \rightarrow \texttt{(} E \texttt{)} \mid \textbf{id} \quad \} \end{aligned}$$

also folgendermaßen in der Pythonklasse *Grammatik* repräsentieren:

```
>>> G = Grammatik('D', list('DETF'), ['id']+list('+*()'), '''D -> E
    E -> E + T
    E -> T
    T -> T * F
    T -> F
    F -> ( E )
    F -> id''' .split('\n'))
```

Die Produktionen sind anschließend im Grammatik-Objekt G folgendermaßen repräsentiert:

```
>>> G.P
[ ('D', ['E']), ('E', ['E', '+', 'T']), ('E', ['T']), ('T', ['T', '*', 'F']),
  ('T', ['F']), ('F', ['(', 'E', ')']), ('F', ['id']) ]
```

Aufgabe 6.5

Schreiben Sie für die Klasse *Grammatik* die Methode *__repr__*, um eine angemessene String-Repräsentation einer Grammatik zu definieren. Orientieren Sie sich an folgender Ausgabe:

```
>>> print G
D --> E
E --> E + T
E --> T
T --> T * F
T --> F
F --> ( E )
F --> id
```

6.2.1 Berechnung der FIRST-Mengen

Einige Algorithmen auf Grammatiken benötigen für jedes Nichtterminal (bzw. für jede Satzform) die sog. FIRST- und FOLLOW-Mengen. Hierbei steht $\mathsf{FIRST}(A)$ für die Menge aller Anfangssymbole von Wörtern, die aus A ableitbar sind. Meist geht man davon aus, dass die FIRST-Funktion auch über Satzformen $\alpha \in V \cup T$ definiert ist; $\mathsf{FIRST}(\alpha)$ steht entsprechend für die Menge aller Anfangssymbole von Wörtern, die aus α ableitbar sind. Formaler:

$$\mathsf{FIRST}(\alpha) := \{\ a \in T \mid \exists w : \alpha \Rightarrow^* w\ \wedge\ w \in T^*\ \wedge\ w \text{ beginnt mit } a\ \}$$

Beispiel 6.5: *FIRST-Mengen*

Für die Grammatik aus Beispiel 6.4 gilt: $\mathsf{FIRST}(D) = \mathsf{FIRST}(E) = \mathsf{FIRST}(T) = \mathsf{FIRST}(F) = \{(, \mathbf{id}\}$

Aufgabe 6.6

Gegeben sei die folgende Grammatik $G = (S, \{a, b, c\}, \{\}, P)$ gegeben, wobei P aus folgenden Produktionen besteht:

$$\begin{aligned} S &\to XYX \mid c \\ X &\to aXa \mid \varepsilon \\ Y &\to Yb \mid \varepsilon \end{aligned}$$

Berechnen Sie $\mathsf{FIRST}(S), \mathsf{FIRST}(X)$ und $\mathsf{FIRST}(Y)$.

Wiederhole die folgenden Schritte für alle $X \in V$, bis sich keine der Mengen $\mathsf{FIRST}(X)$ mehr verändert.

1. Gibt es eine Produktion $X \to \varepsilon$, so setze $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \{\varepsilon\}$
2. Gibt es eine Produktion $X \to Y_0 Y_1 \ldots Y_n$, dann:
   (a) Falls $Y_0 \in V$: Setze $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \mathsf{FIRST}(Y_0)$
   Falls $Y_0 \in T$: Setze $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \{Y_0\}$
   (b) Für alle $i \in \{1, \ldots, n\}$: Falls $\varepsilon \in \mathsf{FIRST}(Y_0), \ldots, \varepsilon \in \mathsf{FIRST}(Y_{i-1})$:
   Falls $Y_i \in V$: Setze $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \mathsf{FIRST}(Y_i)$
   Falls $Y_i \in T$: Setze $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \{Y_i\}$
   (c) Falls $\varepsilon \in \mathsf{FIRST}(Y_i)$ für $i = 0, \ldots, n$, so setze $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \{\varepsilon\}$.

Abb. 6.3: *Algorithmus zur Berechnung von $\mathsf{FIRST}(X)$ für $X \in V$.*

Da diese Definition noch kein Berechnungsverfahren festlegt, geben wir zusätzlich in Abbildung 6.3 einen Algorithmus zur Berechnung von $\mathsf{FIRST}(X)$, für $X \in V$ an. Wie man sieht, müssen wir zur Berechnung der FIRST-Menge eines Nichtterminals also sukzessive alle rechten Seiten der Produktionen für dieses Terminal untersuchen (Fall 2.) und – falls die jeweilige rechte Seite mit einem Terminal beginnt, diese in die FIRST-Menge mit aufnehmen (Fall 2(a)). Beginnt die rechte Seite mit einem Nichtterminal, so müssen alle Elemente der FIRST-Menge dieses Nichtterminals in die FIRST-Menge mit aufgenommen werden. Dies gilt auch für folgende Nichtterminale, falls alle linksstehenden Nichtterminale ε ableiten (Fall 2(b)).

Dies kann man direkt in Python umsetzen; wir speichern die berechneten FIRST-Mengen in einem Dict-Objekt *self. first* ab, dessen Schlüssel die Nichtterminale der Grammatik sind und die dazugehörenden Werte die FIRST-Mengen. Listing 6.2.1 zeigt die notwendigen Ergänzungen in Form von vier zusätzlichen Zeilen in der *__init__* -Funktion:

```
1  def __init__ ( self,S,V,T,P=[]):
2      ... # Code von Listing 6.2
3      self. first  = {}
4      for X in self. V:
5          self. first [X] = set()
6      self. firstCalc ()
```

In Zeile 4 werden alle Einträge von *self. first* auf die leere Menge *set*() gesetzt. Wie schon durch den Algorithmus in Abbildung 6.3 angedeutet, werden wir häufig die Vereinigungs-Operation benötigen; der *set*-Typ eignet sich hier folglich besser als der *list* -Typ.

In Zeile 6 wird die Methode *firstCalc* verwendet, um *self. first* [X] für alle $X \in$ *self. V* zu berechnen. Das folgende Listing 6.1 zeigt die Implementierung dieser *firstCalc*-Methode:

```
1  def firstCalc( self ):
2    while True:
3      oldFirst = deepcopy(self. first )
4      for X,alpha in self.P:
5        for Y in alpha:
6          if Y in self.T:
7            self. first [X].add( Y)
8            break
9          if Y in self.V:
10           self. first [X] = self. first [X].union( self. first [Y])
11           if '' not in self. first [Y]:
12             break
13       if all( [Y in self.V and '' in self. first [Y] for Y in alpha]):
14           self. first [X].add('')
15     if oldFirst == self. first :
16       break
```

Listing 6.1: *Python-Implementierung des in Abbildung 6.3 gezeigten Algorithmus.*

Zunächst wird in Zeile 3 eine Kopie *aller* momentanen FIRST-Mengen erstellt, um am Ende in Zeile 15 und 16 feststellen zu können, ob das Abbruchkriterium erfüllt ist: Abgebrochen wird nämlich dann, wenn sich keine der FIRST-Mengen mehr verändert hat. Ohne die Verwendung der *deepcopy*-Funktion würde lediglich die Referenz auf das *self. first*-Dictionary kopiert und ein Gleichheitstest mittels des Vergleichsoperators „==“ würde entsprechend immer „*True*“ liefern. Die Verwendung der *deepcopy*-Funktion erzwingt das Erstellen einer tatsächlichen vollständigen Kopie.

Die **for**-Schleife in Zeile 4 läuft über alle Produktionen p; die linke Seite wird jeweils an die Variable X, die rechte Seite an die Variable *alpha* gebunden. Für jede Produktion werden alle Symbole Y der rechten Seite *alpha* durchlaufen; dies geschieht in der **for**-Schleife in Zeile 5. Es werden zwei Fälle unterschieden:

- **if**-Anweisung in Zeile 6: Ist Y ein Terminal, wird dieses Terminal der Menge *first* $[X]$ hinzugefügt – dies entspricht der Zuweisung $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \{Y_i\}$ in Algorithmus aus Abbildung 6.3. Die weiteren Symbole aus *alpha* brauchen dann nicht mehr betrachtet zu werden, und die **for**-Schleife wird mittels **break** verlassen.

- **if**-Anweisung in Zeile 9: Ist Y dagegen ein Nichtterminal, so wird jedes Element aus *first* $[Y]$ in *first* $[X]$ eingefügt – dies entspricht der Zuweisung $\mathsf{FIRST}(X) := \mathsf{FIRST}(X) \cup \mathsf{FIRST}(Y_i)$ in Algorithmus aus Abbildung 6.3. Sollte ε nicht in *first* $[Y]$ enthalten sein (Prüfung in Zeile 11), so brauchen die nachfolgenden Symbole aus *alpha* nicht weiter betrachtet zu werden und die **for**-Schleife wird mittels **break** verlassen.

Aufgabe 6.7

Wo und wie genau wird der Fall 1. in dem in Abbildung 6.3 dargestellten Algorithmus in der in Listing 6.1 Implementierung abgedeckt.

Einige Parse-Algorithmen benötigen die FIRST-Menge einer Satzform. Basierend auf dem dict-Objekt *first* lässt sich einfach eine Methode *firstSatzform* implementieren, die die entsprechende FIRST-Menge einer Satzform α zurückliefert – siehe hierzu auch Aufgabe 6.8.

Aufgabe 6.8

Erstellen Sie eine Methode *firstSatzform* der Klasse *Grammatik*. Hierbei soll *firstSatzform*(α) die FIRST-Menge der Satzform α zurückliefern.

6.2.2 Berechnung der FOLLOW-Mengen

Die Menge $\mathsf{FOLLOW}(X)$ einer Grammatik $G = (S, V, T, P)$ für ein Nichtterminal X enthält alle Terminalsymbole, die in irgendeinem Ableitungsschritt unmittelbar rechts von X stehen können. Formaler:

$$\mathsf{FOLLOW}(X) := \{\ a \in T \mid \exists \alpha, \beta :\ S \Rightarrow^* \alpha X a \beta\ \}$$

Man beachte, dass $\$ \in \mathsf{FOLLOW}(X)$, falls $S \Rightarrow^* \alpha X$.

Da diese Definition noch kein Berechnungsverfahren festlegt, geben wir zusätzlich in Abbildung 6.4 einen Algorithmus zur Berechnung von $\mathsf{FOLLOW}(Y)$, für alle $Y \in V$ an.

1. Setze $\mathsf{FOLLOW}(S) := \{\$\}$
2. Wiederhole die folgenden Schritte für alle $Y \in V$, bis sich keine der Mengen $\mathsf{FOLLOW}(Y)$ mehr verändert.
   (a) Für jede Produktion der Form $X \to \alpha Y \beta$:
   setze $\mathsf{FOLLOW}(Y) := \mathsf{FOLLOW}(Y) \cup \mathsf{FIRST}(\beta) \setminus \{\varepsilon\}$
   (b) Für jede Produktion der Form $X \to \alpha Y$ oder $X \to \alpha Y \beta$, mit $\beta \Rightarrow^* \varepsilon$:
   setze $\mathsf{FOLLOW}(Y) := \mathsf{FOLLOW}(Y) \cup \mathsf{FOLLOW}(X)$.

Abb. 6.4: *Algorithmus zur Berechnung von* $\mathsf{FOLLOW}(X)$ *für alle* $X \in V$.

Aufgabe 6.9

Sind die beiden Fälle 2(a) und 2(b) des in Abbildung 6.4 gezeigten Algorithmus disjunkt?

Auch die Berechnung der FOLLOW-Mengen können wir direkt in Python umsetzen. Zunächst erweitern wir die *__init__*-Methode der Klasse *Grammatik* um die folgenden Zeilen:

```
1  def __init__(self,S,V,T,P=[]):
2      ... # Code von Listing 6.2 und Listing 6.2.1
3      self.follow = {}
4      for X in self.V:
5          self.follow[X] = set()
6      self.followCalc()
```

Analog zur Repräsentation der FIRST-Mengen, verwenden wir auch bei der Repräsentation der FOLLOW-Mengen ein Dictionary-Objekt, dessen Schlüssel Elemente aus $self.T$ und dessen Werte *set*-Objekte sind. Zunächst werden in den Zeilen 4 und 5 alle Einträge $self.follow[X]$, für $X \in T$ auf die leere Menge $set()$ gesetzt.

Die in Listing 6.2 gezeigte Methode *followCalc* implementiert die eigentliche Berechnung der FOLLOW-Mengen.

```
1  def followCalc(self):
2      oldFollow = {}
3      self.follow[self.S].add('$')  # Fall 1.
4      while oldFollow != self.follow:
5          oldFollow = deepcopy(self.follow)
6          for (X,Y,beta) in [(p[0],p[1][i],p[1][i+1:]) for p in self.P
7                                                       for i in range(len(p[1]))
8                                                         if p[1][i] in self.V]:
9              firstBeta = self.firstSatzform(beta)
10             if beta:  # Fall 2.(a)
11                 firstBetaD = firstBeta.difference(['ε'])
12                 self.follow[Y] = self.follow[Y].union(firstBetaD)
13             if not beta or 'ε' in firstBeta: # Fall 2.(b)
14                 self.follow[Y] = self.follow[Y].union(self.follow[X])
```

Listing 6.2: *Python-Implementierung des in Abbildung 6.4 gezeigten Algorithmus.*

Ähnlich wie bei der Berechnung der FIRST-Mengen, wird auch hier in jeder Iteration mittels *deepcopy* eine vollständige Kopie der FOLLOW-Mengen angelegt und nur dann eine weitere Iteration durchgeführt, wenn sich mindestens eine der FOLLOW-Mengen verändert hat. Die **for**-Schleife in Zeile 6 durchläuft alle Variablen X und Y, für die es eine Produktion der Form $X \to \alpha Y \beta$ gibt. Die Variable *firstBeta* wird in Zeile 9 auf $\mathsf{FIRST}(\beta)$ gesetzt. Ist $\beta \neq \varepsilon$ (dies entspricht der **if**-Abfrage in Zeile 10), so tritt der in Algorithmus 6.4 unter 2(a) beschriebene Fall ein und es wird der Menge $\mathsf{FOLLOW}(Y)$ die Menge $\mathsf{FIRST}(\beta) \setminus \{\varepsilon\}$ hinzugefügt – diese geschieht in Zeile 12. Ist $\beta = \varepsilon$ oder $\varepsilon \in \mathsf{FIRST}(\beta)$ (entspricht der **if**-Abfrage in Zeile 13), so tritt der in Algorithmus 6.4 unter 2(b) beschriebene Fall ein und es wird der Menge $\mathsf{FOLLOW}(Y)$ die Menge $\mathsf{FOLLOW}(X)$ hinzugefügt.

Aufgabe 6.10

Gegeben sei die Grammatik $G = (Z, \{a, b, c\}, \{Z, S, A, B\}, P\}$, wobei P aus den folgenden Produktionen besteht:

$$
\begin{aligned}
Z &\to S \mid \varepsilon \\
S &\to BASc \mid aSa \\
A &\to bAb \\
B &\to cBc \mid \varepsilon
\end{aligned}
$$

Berechnen Sie die FOLLOW-Mengen aller Nichtterminale.

6.3 Recursive-Descent-Parser

Wir führen in diesem Abschnitt die vielleicht einfachste Art der Syntaxüberprüfung für Typ-2-Sprache ein: Die Erstellung eines Recursive-Descent-Parsers. Ein Recursive-Descent-Parser benötigt keine explizite Repräsentation der Grammatik wie im letzten Abschnitt gezeigt, sondern repräsentiert eine Grammatik in Form einer Sammlung von (eigens für die jeweilige Grammatik) erstellten Prozeduren; eine Ableitung wird durch Aufrufen von rekursiven Prozeduren „simuliert“. Entsprechend ist ein Recursive-Descent-Parser auch nicht generisch, sondern immer auf eine bestimmte Grammatik zugeschnitten.

Ganz anders verhält es sich mit dem in Abschnitt 6.4 vorgestellten LR-Parser; dieser ist generisch und eben nicht auf eine bestimmte Grammatik beschränkt; er erwartet als Eingabe eine in Python repräsentierte Grammatik in der in Abschnitt 6.2 vorgestellten Form und erstellt daraus automatisch einen Parser; LR-Parser sind beliebte Methoden Parsergeneratoren (wie beispielsweise Yacc einer ist) herzustellen.

6.3.1 Top-Down-Parsing

Es gibt zwei grundsätzlich unterschiedliche Vorgehensweisen, einen Text basierend auf einer formalen Grammatik zu parsen und einen entsprechenden Syntaxbaum zu erzeugen:

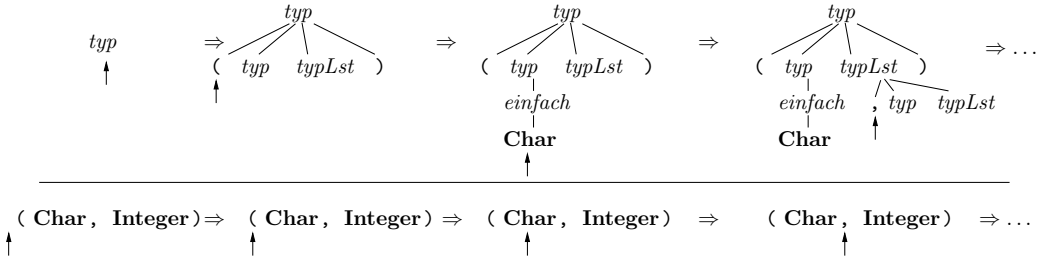
- Bottom-Up-Parsing: Hier wird der Syntaxbaum von unten nach oben erzeugt und die Produktionen der Grammatik quasi von links nach rechts angewendet, solange bis man bei der Startvariablen (also der Wurzel des Syntaxbaums) angelangt ist. Bottom-Up-Parser (wie etwa der in Abschnitt 6.4 vorgestellte LR-Parser) sind i. A. komplexer zu programmieren, können aber eine größere Teilmenge von Typ-2-Sprachen erkennen.
- Top-Down-Parsing: Hier wird der Syntaxbaum von oben nach unten erzeugt und zunächst mit der Startproduktion begonnen. Top-Down-Parser sind i. A. leicht zu programmieren, können jedoch nur eine verhältnismäßig kleine Teilmenge von Typ-2-Sprachen erkennen.

Beispiel 6.6: *Top-Down-Parsing*

Gegeben sei die folgende Grammatik, die die syntaktische Struktur einfache Haskell[2]-Datentypen beschreibt.

$$\begin{array}{lcl} \mathit{typ} & \rightarrow & \mathit{einfach} \mid \texttt{[}\ \mathit{typ}\ \texttt{]} \mid \texttt{(}\ \mathit{typ}\ \mathit{typLst}\ \texttt{)} \\ \mathit{einfach} & \rightarrow & \textbf{Integer} \mid \textbf{Char} \mid \textbf{Bool} \\ \mathit{typLst} & \rightarrow & \texttt{,}\ \mathit{typ}\ \mathit{typLst} \mid \varepsilon \end{array}$$

Folgende Abbildung zeigt den Anfangsteil eines Top-Down-Parsevorgangs für die Erkennung des Wortes „**(Char , Integer)**“. Die obere Hälfte zeigt einen Teil des Syntaxbaums der bisher aufgebaut wurde; die untere Hälfte zeigt die jeweiligen Positionen im Eingabewort an der sich der Parsevorgang befindet.



6.3.2 Prädiktives Parsen

Im allgemeinen Fall ist nicht sichergestellt, dass beim Betrachten des nächsten Eingabezeichens eindeutig klar ist, welche Produktion ausgewählt werden muss. Für eine allgemeine Typ-2-Grammatik muss ein solcher Parser möglicherweise mit Backtracking arbeiten: Sollte es sich im weiteren Verlauf des Parsevorgangs herausstellen, dass die Auswahl einer Produktion (aus mehreren möglichen) falsch war, so muss der Parsevorgang zurückgesetzt werden, eine andere Alternative gewählt und mit dieser fortgefahren werden. Dies entspricht einer Tiefensuche durch den Baum aller möglichen Parse-Wege, die im schlechtesten Fall exponentielle Laufzeit haben kann. Eigentlich möchte man, dass immer nur höchstens eine mögliche Produktion zur Auswahl steht, dass also der „Baum“ aller möglichen Parse-Wege eine simple Liste ist. Welche Eigenschaften muss eine entsprechende Grammatik haben um ein solches sog. *Prädiktives Parsen* zu ermöglichen?

Angenommen, das nächste zu expandierende Nichtterminal-Symbol sei A und das nächste Eingabezeichen sei x; die Produktionen der verwendeten Grammatik, deren linke Seite A ist, seien $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_n$. Es ist klar, dass eine solche Produktion ausgewählt werden muss aus deren rechter Seite α_i das Terminal x als erstes Zeichen ableitbar ist; in anderen Worten: Es muss eine Produktion $A \rightarrow \alpha_i$ gewählt werden mit $x \in \mathsf{FIRST}(\alpha_i)$. Ist diese als Nächstes zu wählende Produktion immer eindeutig bestimmt, so bezeichnet man die Grammatik als prädiktiv.

[2]Die Programmiersprache Haskell ist wohl der prominenteste Vertreter der reinen funktionalen Programmiersprachen.

Es ist klar, dass für jede prädiktive Grammatik folgende Bedingung gelten muss: Für je zwei Produktionen $A \to \alpha$ und $A \to \beta$ mit gleichen linken Seiten A muss gelten, dass

$$\mathsf{FIRST}(\alpha) \cap \mathsf{FIRST}(\beta) = \emptyset$$

d. h. die FIRST-Mengen der rechten Seiten müssen paarweise disjunkt sein.

6.3.3 Implementierung eines Recursive-Descent-Parsers

Ein Recursive-Descent-Parser arbeitet das Eingabewort über den Aufruf rekursiver Prozeduren ab. Jedes Nichtterminal der Grammatik wird als Prozedur implementiert. Um einen Recursive-Descent-Parser für die Grammatik aus Beispiel 6.6 (die Sprache einfacher Haskell-Typen) zu erstellen, müssen Prozeduren *typ*, *typLst* und *einfach* erstellt werden. Jedem Nichtterminal auf der rechten Seite einer Produktion entspricht ein Prozeduraufruf, jedem Terminal auf der rechten Seite einer Prozedur entspricht einer Match-Operation, die prüft, ob das aktuelle Zeichen der Eingabe mit dem entsprechenden Terminalsymbol übereinstimmt.

Listing 6.3 zeigt die Implementierung eines Recursive-Descent-Parsers für die Grammatik aus Beispiel 6.6. Auf der linken Seite sind immer die zum jeweiligen Code-Fragment passenden Produktionen der Grammatik zu sehen.

Die in Zeile 27 durch Benutzereingabe definierte Variable s enthält die Liste der zu parsenden Eingabesymbole; es wird immer der Wert *None* an das Ende dieser Liste angehängt; dieser Wert wird von dem Parser als Ende-Symbol interpretiert und entspricht dem `'$'`-Symbol in der in Abschnitt 6.2 präsentieren Grammatik. Die Variable *lookahead* zeigt immer auf das nächste vom Parser zu lesende Symbol aus s.

Der Parse-Vorgang wird durch das Ausführen der Prozedur S – die dem Startsymbol S entspricht – in Gang gesetzt. Man beachte: Ähnlich wie bei dem im nächsten Abschnitt beschriebenen LR-Parser ist man auch hier angehalten, für das Startsymbol – in diesem Fall: *typ* – eine zusätzliche spezielle Produktion – in diesem Fall: $S \to$ *typ* `$` – einzufügen, die das Ende der Eingabe erkennt.

Dieser Recursive-Descent-Parser ist tatsächlich auch ein prädiktiver Parser: In jeder Prozedur – dies trifft insbesondere für die Prozedur *typ* zu – kann durch Lesen des nächsten Eingabesymbols $s[lookahead]$ immer eindeutig die passende Produktion ausgewählt werden.

```
                                  1  def match(c):
                                  2    global lookahead
                                  3    if s[lookahead] == c: lookahead += 1
                                  4    else: print "Syntaxfehler"
                                  5
                                  6  def S():
                        S → typ   7    typ() ; match(None)
                                  8
                                  9  def typ():
                                 10    if s[lookahead] in ['Integer','Char','Bool']:
                   typ → einfach 11      einfach()
                                 12    elif s[lookahead] == '[':
                   typ → [ typ ] 13      match('[') ; typ() ; match(']')
                                 14    elif s[lookahead] == '(':
typ → ( typ typLst ) 15      match('(') ; typ() ; typLst() ; match(')')
                                 16    else: print "Syntaxfehler"
                                 17
                                 18  def einfach():
einfach → Integer | ... 19    match(s[lookahead])
                                 20
                                 21  def typLst():
                                 22    if s[lookahead] == ',':
typLst → , typ typLst 23      match(',') ; typ() ; typLst()
                                 24    else:
                    typLst → ε 25      pass
                                 26
                                 27  s = raw_input('Haskell-Typ? ').split() + [None]
                                 28  lookahead = 0
                                 29  S()
```

Listing 6.3: *Recursive-Descent-Parser Wörter der Grammatik aus Beispiel 6.6 erkennt.*

6.3.4 Vorsicht: Linksrekursion

Eine Produktion heißt *linksrekursiv*, falls das am weitesten links stehende Symbol der rechten Seite mit dem Symbol der linken Seite identisch ist; eine Grammatik heißt linksrekursiv, falls sie linksrekursive Produktionen enthält.

Beispiel 6.7: *Linksrekursive Grammatik*

Folgende linksrekursive Grammatik beschreibt die Syntax einfacher arithmetischer Ausdrücke, bestehend aus `+`, `-`, `0`, ... `9`.

$$\begin{aligned} &\textit{ausdr} \rightarrow \textit{ausdr}\ \texttt{+}\ \textit{ziffer} \mid \textit{ausdr}\ \texttt{-}\ \textit{ziffer} \\ &\textit{ausdr} \rightarrow \textit{ziffer} \\ &\textit{ziffer} \rightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \end{aligned}$$

Aufgabe 6.11

Erstellen Sie einen Syntaxbaum für den Ausdruck

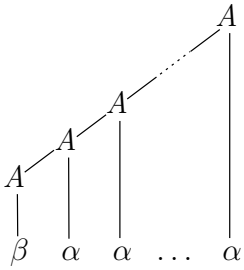
`9 + 5 - 2`

basierend auf der Grammatik aus Beispiel 6.7.

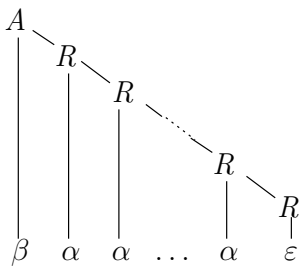
Betrachten wir einen linksrekursiven „Teil“ einer Grammatik, d. h. eine Produktion der Form $A \rightarrow\ A\alpha \mid \beta$ mit dem linksrekursiven „Fall“ $A\alpha$ und dem „Abbruch“-Fall β – wobei wir voraussetzen, dass $\alpha, \beta \in V \cup T$ und α und β nicht mit dem Nichtterminal A beginnen. Diese Produktion erzeugt beliebig lange Folgen von αs, die mit einem β beginnen. Ein entsprechender Syntaxbaum ist in Abbildung 6.5(a) zu sehen. Eine solche mit einem β beginnende α-Folge, könnte man aber auch mit den nicht linksrekursiven Produktionen $A \rightarrow \beta R$, $R \rightarrow \alpha R \mid \varepsilon$ erzeugen; ein entsprechender Syntaxbaum ist in Abbildung 6.5(b) zu sehen. Enthalten die Produktionen für ein Nichtterminal mehrere linksrekursive Fälle, so können diese in analoger Weise in nicht linksrekursive Produktionen umgewandelt werden; Tabelle 6.2 zeigt nochmals diese in und die oben beschriebene Transformation zur Elimination von Linksrekursion.

Linksrekursive Produktionen		**Nicht linksrekursive Prod.**
$A \rightarrow\ A\alpha \mid \beta$	$\Longrightarrow$	$A \rightarrow \beta R$ $R \rightarrow \alpha R \mid \varepsilon$
$A \rightarrow\ A\alpha \mid A\beta \mid \gamma$	$\Longrightarrow$	$A \rightarrow \gamma R$ $R \rightarrow \alpha R \mid \beta R \mid \varepsilon$

Tabelle 6.2: *Transformationsschemata zur Elimination von Linksrekursion aus einer Grammatik. Hierbei gilt, dass $\alpha, \beta \in V \cup T$ und sowohl α als auch β beginnen nicht mit dem Nichtterminal A.*



(a) Syntaxbaum des Wortes $\beta\alpha\alpha\ldots\alpha$ durch die linksrekursive Produktion.



(b) Syntaxbaum des Wortes $\beta\alpha\alpha\ldots\alpha$ durch die Produktion mit eliminierter Linksrekursion.

Abb. 6.5: *Syntaxbäume des Worts $\beta\alpha\alpha\ldots\alpha$ für die linksrekursiven Produktionen $A \to A\,\alpha \mid \beta$ und für die entsprechenden nicht linkrekursiven Produktionen $A \to \beta\,R;\ R \to \alpha\,R \mid \varepsilon$.*

Aufgabe 6.12

Gegeben sei die folgende linksrekursive Grammatik

$$
\begin{aligned}
\mathit{ausdr} &\to \mathit{ausdr}\ \texttt{+}\ \mathit{term}\\
\mathit{ausdr} &\to \mathit{ausdr}\ \texttt{-}\ \mathit{term}\\
\mathit{ausdr} &\to \mathit{term}\\
\mathit{term} &\to \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9}
\end{aligned}
$$

(a) Eliminieren Sie die Linksrekursion aus dieser Grammatik.

(b) Implementieren Sie einen Recursive-Descent-Parser, der die durch diese Grammatik beschriebene Sprache erkennt.

6.4 Ein LR-Parsergenerator

Ein LR-Parser arbeitet seine Eingabe von links nach rechts ab (daher das „L“) und erzeugt Rechtsableitungen (daher das „R“), d. h. immer das am weitesten rechts stehende Nichtterminal wird durch die rechte Seite einer Produktion ersetzt.

Technik: Wir erstellen zunächst einen endlichen Automaten (sehr ähnlich einem sog. Kreuzprodukt-Automaten), der gültige Präfixe der durch die Grammatik beschriebenen Sprache erkennt. Auf dessen Basis erstellen wir schließlich eine Syntaxtabelle, mit deren Hilfe die Sprache effizient erkannt werden kann.

6.4.1 LR(0)-Elemente

Ein LR(0)-Element einer Grammatik G ist eine Produktion aus G zusammen mit einer Position auf der rechten Seite dieser Produktion; diese Position markieren wir mit einem „`.`“. Ein LR(0)-Element der Grammatik aus Beispiel 6.4 wäre etwa

$$\langle F \to \texttt{(} \ . \ E \ \texttt{)} \rangle$$

Ein LR(0)-Element enthält Informationen darüber, an welcher „Stelle“ sich ein Parse-Vorgang befindet. Wir können uns also vorstellen, dass das LR(0)-Element $\langle F \to \texttt{(} \ . \ E \ \texttt{)} \rangle$ den Status eines Parsevorgangs widerspiegelt, der gerade dabei ist zu versuchen, das Nichtterminal F zu erkennen und vorher schon das Terminal `(` erkannt hat und als Nächstes versuchen wird das Nichtterminal E zu erkennen.

In Python kann ein LR(0)-Element einer Grammatik *Grammatik*(S,T,V,P) als Tupel (i,j) repräsentiert werden, wobei $i \in$ *range*(*len*(*self*.P)) die Nummer der entsprechenden Produktion und $j \in$ *range*(*len*(*self*.$P[i]\,[1]$) +1) die Position des „`.`“ auf der rechten Seite der Produktion spezifiziert. Geht man von der in Beispiel 6.4 gezeigten Repräsentation der Grammatik aus, so würde man das LR(0)-Element $\langle F \to \texttt{(} \ . \ E \ \texttt{)} \rangle$ durch das Tupel (5,1) repräsentieren – die „5“ steht für den Indexposition der Produktion $F \to \texttt{(}\,E\,\texttt{)}$ innerhalb der Produktionenliste *self*.P und die „1“ steht für die Position des „`.`“-Zeichens (nämlich rechts des erstens Symbols der rechten Seite der Produktion).

Aufgabe 6.13

Implementieren Sie für die Klasse *Grammatik* eine Methode *printElement*(i,j), das das durch (i,j) repräsentierte LR(0)-Element in gut lesbarer Form auf dem Bildschirm ausgibt, wie etwa in folgender Beispielanwendung:

```
>>> G.printElement(5,1)
'F -> ( . E )'
```

6.4.2 Die Hüllenoperation

Befindet sich ein Parsevorgang vor einem Nichtterminal Y – haben wir es also mit einem LR(0)-Element der Form $\langle X \to \alpha \ . \ Y\beta \rangle$ zu tun – versucht der Parser als Nächstes, das Nichtterminal Y zu erkennen. Befindet sich in dieser Grammatik eine Produktion der Form $Y \to \gamma$, so entspricht die Situation das Nichtterminal Y zu erkennen, auch dem LR(0)-Element $\langle Y \to \ . \gamma \rangle$.

Die Hüllenoperation *huelle*(E) führt eine entsprechende Erweiterung einer Sammlung E von LR(0)-Elementen durch; *huelle*(E) enthält also immer LR(0)-Elemente, die derselben Parse„situation“ entsprechen. Listing 6.4 zeigt eine Implementierung in Python.

Die Methode *huelle* sammelt in der Listenvariablen *E_huelle* alle zur Hülle der LR(0)-Elemente gehörenden Elemente auf; in *E_neu* befinden sich immer in der jeweiligen Iteration neu hinzugekommenen Elemente. Die **while**-Schleife in Zeile 5 wird so lange

wiederholt, bis keine weiteren Elemente hinzukommen. Zu Beginn jeder Iteration werden zunächst alle in der letzten Iteration neu aufgesammelten LR(0)-Elemente zu *E_huelle* hinzugefügt. Die in Zeile 7 definierte Liste *Ys* enthält alle Elemente $Y \in V$ für die es ein LR(0)-Element der Form $\langle X \to \alpha \,.\, Y\beta \rangle \in$ *E_neu* gibt. Die in Zeile 9 definierte Liste *E_neu* sammelt nun alle LR(0)-Elemente der Form $\langle Y \to \,.\gamma \rangle$ auf, die sich noch nicht in der bisher berechneten Hülle befinden. Können der Hülle keine weiteren Elemente hinzugefügt werden, so bricht die **while**-Schleife ab und es wird *E_huelle* als set-Struktur zurückgeliefert.

```
1  class Grammatik(object):
2    ...
3    def huelle(self,E):
4      E_huelle = [] ; E_neu = E[:]
5      while E_neu:
6        E_huelle += E_neu
7        Ys = [self.P[i][1][j] for (i,j) in E_neu
8                              if j<len(self.P[i][1]) and self.P[i][1][j] in self.V]
9        E_neu = [(i,0) for i in range(len(self.P))
10                     if self.P[i][0] in Ys and (i,0) not in E_huelle]
11     return set(E_huelle)
```

Listing 6.4: *Implementierung der Hüllenoperation*

6.4.3 Die GOTO-Operation

Entscheidend für die Konstruktion eines LR-Parsers ist die GOTO-Operation: Für $Y \in (V \cup T)$ (d. h. Terminal oder Nichtterminal) ist $\mathsf{GOTO}(E, Y)$ definiert als die Hülle aller LR(0)-Elemente $\langle X \to \alpha Y \,.\, \beta \rangle$, mit $\langle X \to \alpha \,.\, Y\beta \rangle \in E$. Listing 6.5 zeigt die Implementierung in Python.

```
1  class Grammatik(object):
2    ...
3    def goto(self,E,Y):
4        return self.huelle([(i,j+1) for (i,j) in E
5                                    if j<len(self.P[i][1]) and self.P[i][1][j]==Y])
```

Listing 6.5: *Implementierung der GOTO-Operation in Python*

Das Tupel (i, j) durchläuft alle LR(0)-Elemente aus E, deren rechte Seiten an der Position neben dem „ . “ (das ist die Position j) das Symbol Y stehen haben. Ist (i, j) die Repräsentation des LR(0)-Elements $\langle X \to \alpha \,.\, Y\beta \rangle$, so ist $(i, j+1)$ die Repräsentation des entsprechenden LR(0)-Elements in $\mathsf{GOTO}(Y)$. Die Methode *goto* liefert nun einfach die Hülle all dieser LR(0)-Elemente zurück.

6.4.4 Erzeugung des Präfix-Automaten

Als nächsten Schritt auf dem Weg hin zu einem LR-Parser konstruieren wir einen deterministischen endlichen Automaten (kurz: DEA), der Präfixe aller aus dem Startsymbol rechts-ableitbaren Satzformen erkennt – vorausgesetzt jeder Zustand wird als möglicher Endzustand interpretiert.

Nach Ausführung der in Listing 6.6 ab Zeile 16 gezeigten Methode *automaton*() enthält Attribut *self.Es* die Sammlung von Elementmengen, die die Zustände des Präfixautomaten darstellen. Jede dieser Elementmengen repräsentiert einen Zustand während des Parsevorgangs.

```
1  class Grammatik(object):
2    ...
3    def automatonRek(self,state):
4      for X in self.V+self.T:
5        goto = self.goto(state,X)
6        if not goto: continue
7        if goto not in self.Es:
8          self.Es.append(goto)
9          gotoInd = len(self.Es)-1
10         self.edges[gotoInd] = []
11         self.edges[self.Es.index(state)].append((X,gotoInd))
12         self.automatonRek(goto)
13       else:
14         self.edges[self.Es.index(state)].append((X,self.Es.index(goto)))
15
16   def automaton(self):
17     start = self.closure([(0,0)])
18     self.Es.append(start)
19     self.edges[0]=[]
20     self.automatonRek(start)
```

Listing 6.6: *Erzeugung des Präfix-erkennenden Automaten*

Als Startzustand interpretieren wir die Hülle des initialen LR(0)-Elements $\langle S' \to \cdot S \rangle$, repräsentiert durch das Tupel (0,0). In Zeile 18 wird diese in die (zu erzeugende) Menge von Elementen *self.Es* eingefügt und die Methode *automatonRek* mit diesem Zustand gestartet. Eine Randbemerkung ist an dieser Stelle angebracht: Will die von einer gegebenen Grammatik mit Startsymbol S erzeugte Sprache durch einen Parser erkennen, so sollte man grundsätzlich eine zusätzliche „künstliche“ Produktion $S' \to S$ einführen; nur mittels dieser künstlichen Produktion kann der Parser erkennen, dass die Eingabe beendet ist; auch bei dem Recursive-Descent-Parser aus Abschnitt 6.3 war dies notwendig.

Die Methode *automatonRek* konstruiert nun rekursiv den Präfixautomaten wie folgt: In der **for**-Schleife in Zeile 4 wird für jedes Grammatiksymbol $X \in V \cup T$ die Elementmenge $\mathsf{GOTO}(state, X)$ berechnet, die – falls nichtleer – einem weiteren Zustand des

Präfixautomaten entspricht. Es können drei Fälle unterschieden werden:

1. Zeile 6: Es gilt $\mathsf{GOTO}(state, X) = \emptyset$, d. h. es gibt keine von *state* ausgehende mit X beschriftete Kante. In diesem Fall ist nichts weiter zu tun und die **for**-Schleife wird mittels der **continue**-Anweisung mit dem nächsten Symbol $X \in V \cup T$ fortgesetzt.
2. Zeile 7: Die Elementmenge $\mathsf{GOTO}(state, X)$ befindet sich noch nicht in *self*.*Es*: $\mathsf{GOTO}(state, X)$ wird in *self*.*Es* eingefügt (Zeile 8) und die Variable *gotoInd* auf die Nummer dieses neuen Zustands gesetzt (Zeile 9). Der neue Zustand enthält noch keine ausgehenden Kanten, d. h. *self*.*edges*[*gotoInd*] wird die leere Liste [] zugewiesen und in die Sammlung der ausgehenden Kanten des Zustands *state* wird der neue Zustand zusammen mit dem Grammatiksymbol X mit aufgenommen (Zeile 11). In Zeile 12 erfolgt die Rekursion: *self*.*automatonRek*(*goto*) erzeugt rekursiv alle vom neu erzeugten Zustand ausgehenden Kanten und die folgenden Zustände.
3. Zeile 13: Die Elementmenge $\mathsf{GOTO}(state, X)$ befindet sich bereits in *self*.*Es*. In diesem Fall wird lediglich die Kante vom Zustand *state* (mit Nummer *self*.*Es*.*index*(*state*)) zum Zustand $\mathsf{GOTO}(state, X)$ in *self*.*edges* eingefügt.

Aufgabe 6.14

Durchläuft der in Listing 6.6 gezeigte Algorithmus die Zustände des Präfixautomaten ...

(a) ... in der Reihenfolge einer Tiefensuche?

(b) ... in der Reihenfolge einer Breitensuche?

(c) ... weder in der Reihenfolge einer Tiefen- noch der Reihenfolge einer Breitensuche?

Beispiel 6.8

Abbildung 6.6 zeigt den Präfixautomaten für die Grammatik $G = (E', T, V, P)$ mit $T = \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{id}\}$ und $V = \{E', E, T, F\}$ und folgenden in P enthaltenen Produktionen:

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

Wir spielen die ersten Schritte bei der Erstellung des Präfixautomaten durch. Wir starten mit der als Anfangszustand betrachteten Elementmenge

HÜLLE($\langle E' \to \texttt{.}\, E \rangle$) – in Abbildung 6.6 entspricht dies genau der als Zustand E_0 bezeichneten Elementmenge. Nun werden die GOTO-Mengen berechnet, angefangen mit GOTO(E_0, E'), die jedoch leer ist. Darum wird zum nächsten Nichtterminal E übergegangen und GOTO$(E_0, E) = \{\langle E' \to E\,\texttt{.} \rangle, \langle E \to E\,\texttt{.}\,\texttt{+}\,T \rangle\}$ berechnet und ein neuer dieser Elementmenge entsprechender Zustand erzeugt; in Abbildung 6.6 entspricht dies Zustand E_1. Durch einen rekursiven Aufruf (Zeile 12 in Listing 6.6) werden zunächst alle vom Zustand E_1 ausgehenden Kanten und die nachfolgenden Zustände bestimmt. Danach wird mit dem nächsten Nichtterminal T fortgefahren und GOTO(E_0, T) zu $\{\langle T \to T\,\texttt{.}\,\texttt{*}\,F \rangle, \langle E \to T\,\texttt{.} \rangle\}$ berechnet; diese Elementmenge entspricht dem Zustand E_9 in Abbildung 6.6. Diese Erzeugung von Kanten und entsprechenden Zuständen wird so für die verbleibenden Symbole aus $V \cup T$ fortgeführt und die Erstellung des Präfixautomaten anschließend beendet.

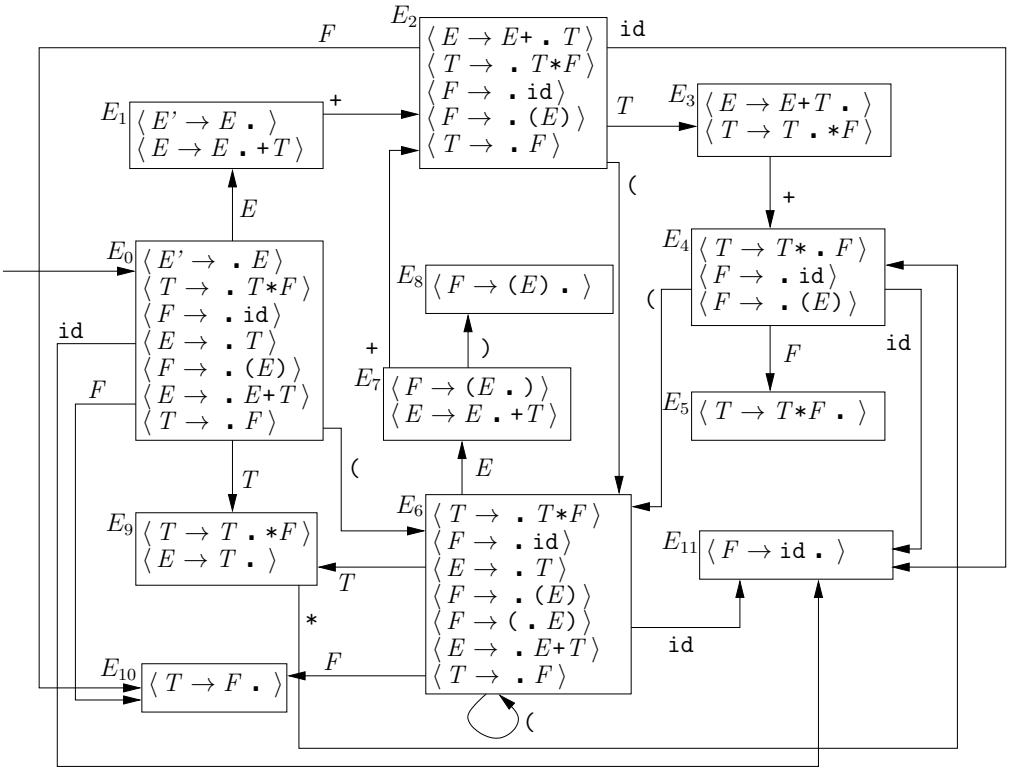


Abb. 6.6: *Präfixautomat der Grammatik G. Die Elementmengen E_0, E_1, ..., E_{11} entsprechen genau den durch in Listing 6.6 gezeigten Funktion automaton() berechneten Elementmengen self.Es[0], self.Es[1], ..., self.Es[11].*

Aufgabe 6.15

Welche der folgenden Satzformen (d. h. Wörter aus $V \cup T$) der in Beispiel 6.8 verwendeten Grammatik können durch den Präfixautomaten erkannt werden? – vorausgesetzt, jeder Zustand des Präfixautomaten ist ein aktzeptierender Zustand.
(a) **id+id** (b) E`+`**id** (c) **id** (d) E`+`T`+(`
(e) `(`E`+((` (f) `(`T`*(`

6.4.5 Berechnung der Syntaxanalysetabelle

Aus dem Präfixautomaten kann nun die Syntaxanalysetabelle erstellt werden, auf der das eigentliche LR-Parsing basiert. Die Syntaxanalysetabelle besteht aus zwei Teilen: der *Aktionstabelle* (in der Implementierung aus Listing 6.7: *self.aktionTab*) und der *Sprungtabelle* (in der Implementierung aus Listing 6.7: *self.sprungTab*). Tabelle 6.3 zeigt Aktions- und Sprungtabelle für die Grammatik aus Beispiel 6.8. Der eigentliche LR-Parser arbeitet als Kellerautomat (dessen Implementierung wir im nächsten Abschnitt vorstellen): Trifft der Parser auf einen Shift-Eintrag – das sind die mit „s“ beginnenden Einträge in Tabelle 6.3 –, so wird der entsprechende Zustand auf den Keller geladen; trifft der Parser auf einen Reduce-Eintrag – das sind die mit „r“ beginnenden Einträge –, so wird mit der entsprechenden Produktion reduziert, d. h. die rechte Seite α (die sich zu diesem Zeitpunkt auf dem Stack befinden sollte) der Produktion $A \to \alpha$ wird durch die Variable A ersetzt; für den Kellerautomaten bedeutet dies, dass die zu den Symbolen der rechten Seite gehörenden Zustände vom Keller entfernt werden und dadurch der Keller um $|\alpha|$ Einträge schrumpft.

Befindet sich der LR-Parser beispielsweise im Zustand E_2 und liest das „`(`“-Zeichen, so lädt er den Zustand 6 (bzw. in dem von uns implementierten Kellerautomaten das Tupel $(6, \texttt{'('})$) auf den Keller. Befindet sich der LR-Parser beispielsweise im Zustand E_3 und liest das „`)`“-Zeichen, so reduziert er mit Produktion *self.P*[1], also der Produktion $E \to E\texttt{+}T$.

Aufgabe 6.16

Erklären Sie einige der Einträge der Syntaxanalysetabelle (Tabelle 6.3):

(a) Warum ist **Aktionstabelle**$[E_0, \texttt{(}] = \text{s6}$?
(b) Warum ist **Sprungtabelle**$[E_6, T] = 9$?
(c) Warum ist **Aktionstabelle**$[E_8, \texttt{)}] = \text{r5}$?

Das Skript in Listing 6.7 berechnet die Syntaxanalysetabelle und verwendet dabei den in Listing 6.6 berechneten Präfixautomaten bestehend aus den Knoten *self.Es* und den Kanten *self.edges*.

	Aktionstabelle						**Sprungtabelle**			
	+	(	)	**id**	*	$	*E'*	*E*	*T*	*F*
E_0		s6		s11				1	9	10
E_1	s2					Acc				
E_2		s6		s11					3	10
E_3	r1		r1		s4	r1				
E_4		s6		s11						5
E_5	r3		r3		r3	r3				
E_6		s6		s11				7	9	10
E_7	s2		s8							
E_8	r5		r5		r5	r5				
E_9	r2		r2		s4	r2				
E_{10}	r4		r4		r4	r4				
E_{11}	r6		r6		r6	r6				

Tabelle 6.3: *Syntaxanalysetabelle für die Grammatik aus Beispiel 6.8 basierend auf dem entsprechenden Präfixautomaten aus Abbildung 6.6*

```
1  class Grammatik(object):
2    ...
3    def tabCalc(self):
4      for i in range(len(self.Es)):
5        for X,j in self.edges[i]:
6          if X in self.T: self.aktionTab[i][X] = (SHIFT, j)
7          if X in self.V: self.sprungTab[i][X] = j
8        if (0,1) in self.Es[i]: self.aktionTab[i]['$'] = ACCEPT
9        for (aS,jS) in [(a,j) for (j,k) in self.Es[i]
10                               if k == len(self.P[j][1]) and self.P[j][0] != self.V[0]
11                             for a in self.follow[self.P[j][0]]]:
12         self.aktionTab[i][aS] = (REDUCE, jS)
```

Listing 6.7: *Berechnung der Syntaxanalysetabelle*

Für jeden Zustand E_i in der durch *automaton*() berechneten Sammlung von Elementmengen *self*.*Es* (das ist die „**for** *i*“-Schleife in Zeile 4) wird für jede mit einem Symbol X beschriftete ausgehende Kante zu einem Zustand j (das ist die „**for** *X*,*j*“-Schleife in Zeile 5) ein Eintrag in der Syntaxanalysetabelle erzeugt: Falls $X \in$ *self*.*T* so wird der Eintrag „sj“ in der Aktionstabelle erzeugt (Zeile 6); falls $X \in$ *self*.*V* wird ein Eintrag „j“ in der Sprungtabelle erzeugt (Zeile 7). In Zeile 8 wird der Eintrag *ACCEPT* in der Syntaxanalysetabelle erzeugt: Befindet sich der Automat in einem Zustand, der das LR(0)-Element $S' \to S\,\mathbf{.}$ enthält und erhält der Automat als nächste Eingabe das Endezeichen `'$'`, so wird die Eingabe erkannt.

Ab Zeile 9 werden die Reduce-Einträge in der mit E_i markierten Zeile erzeugt: Für jedes LR(0)-Element in E_i der Form $\langle\, A \to \alpha\,\mathbf{.}\,\rangle$ mit $A \neq S'$ (dies entspricht dem

Test $self.P[j][0] \neq self.V[0]$) wird in Spalte X ein Reduce-Eintrag erzeugt, falls $X \in \mathsf{FOLLOW}(A)$ – nur falls $X \in \mathsf{FOLLOW}(A)$ kann nämlich X ein erlaubtes Nächstes Zeichen im Parse-Prozess sein.

Aufgabe 6.17

Schreiben Sie eine Funktion *printTab* als Methode der Klasse *Grammatik*, die die durch *tabCalc* berechnete Syntaxanalysetabelle in lesbarer Form ausgibt. Beispiel:
>>> **print** *G.printTab*()

```
     |    +    (    )    id   *    $    E'   E    T    F
  0  |         s6        s11                   1    9    10
  1  |    s2                       acc
  2  |         s6        s11                   3    10
  3  |    r1        r1        s4   r1
     ...                      ...
```

6.4.6 Der Kellerautomat

Das in Listing 6.8 gezeigte Skript implementiert den eigentlichen Parser in Form eines Kellerautomaten. Dieser Kellerautomat greift in jedem Schritt auf Einträge der Syntaxanalysetabelle zu und bestimmt daraus die nächste auszuführende Aktion.

```
1  class Grammatik(object):
2    ...
3    def parse(self,s):
4      s = s.split() + ['$']
5      stack = [(0,None)] ; zustand = 0 ; i=0 ; prods=[]
6      while True:
7        x = s[i]
8        if x not in self.aktionTab[zustand]:
9          print "error at",x
10         return
11       aktion = self.aktionTab[zustand][x]
12       if aktion[0] == SHIFT:
13         stack.append((aktion[1], x))
14         i+=1
15       elif aktion[0] == REDUCE:
16         p = self.P[aktion[1]] # Reduktion mit p = A → α
17         prods.append(p)
18         stack = stack[:-len(p[1])] # Stack um |α| erniedrigen
19         stack.append((self.sprungTab[stack[-1][0]][p[0]],p[0])) # stack.append(GOTO(A),A)
20       elif aktion[0] == ACCEPT:
21         return prods
22       zustand = stack[-1][0]
```

Listing 6.8: *Implementierung des Kellerautomaten*

Die Variable s enthält das zu parsende Wort in Form einer Liste von Terminalsymbolen. Der Stack wird in der Variablen *stack* gehalten. Innerhalb der **while**-Schleife wird das Wort s durchlaufen. Hierbei enthält x immer den aktuellen Buchstaben. Der Zustand *zustand* des Kellerautomaten ist immer der im obersten Tupel des Stacks gespeicherte Zustand (siehe Zeile 22).

Sollte die Syntaxanalysetabelle für den aktuellen Zustand *zustand* und das aktuelle Zeichen x keinen Eintrag enthalten, (der entsprechende Test erfolgt in Zeile 8) so wird eine Fehlermeldung ausgegeben. Andernfalls enthält *aktion* die durchzuführende Aktion; hier sind 3 Fälle zu unterscheiden:

1. *aktion* ist eine Shift-Operation – dies wird in Zeile 12 geprüft: In diesem Fall wird der Zustand $aktion[1]$ zusammen mit dem aktuellen Eingabezeichen als Tupel auf den Stack gelegt.

2. *aktion* ist eine Reduce-Operation – dies wird in Zeile 15 geprüft: In diese Fall wird mit der Produktion $p = self.P[aktion[1]]$ reduziert. Hierbei wird zunächst der Stack um die Länge der rechten Seite α von p reduziert (Zeile 18); mit Hilfe des nun oben auf dem Stack liegenden Zustands $stack[-1][0]$ und der linken Seite A der Produktion p wird aus der Sprungtabelle der Folgezustand bestimmt und diesen zusammen mit A auf den Stack gelegt (Zeile 19).

3. *aktion* ist die Accept-Operation – dies wird in Zeile 20 geprüft. Dies bedeutet, dass die Eingabe akzeptiert wird und die *parse*-Funktion mit der Rückgabe der für den Parse-Vorgang verwendeten Produktionen abbricht.

Der vollständige Ablauf einer Parse-Operation des Beispielworts „(**id** * **id**)“ ist in Abbildung 6.7 gezeigt.

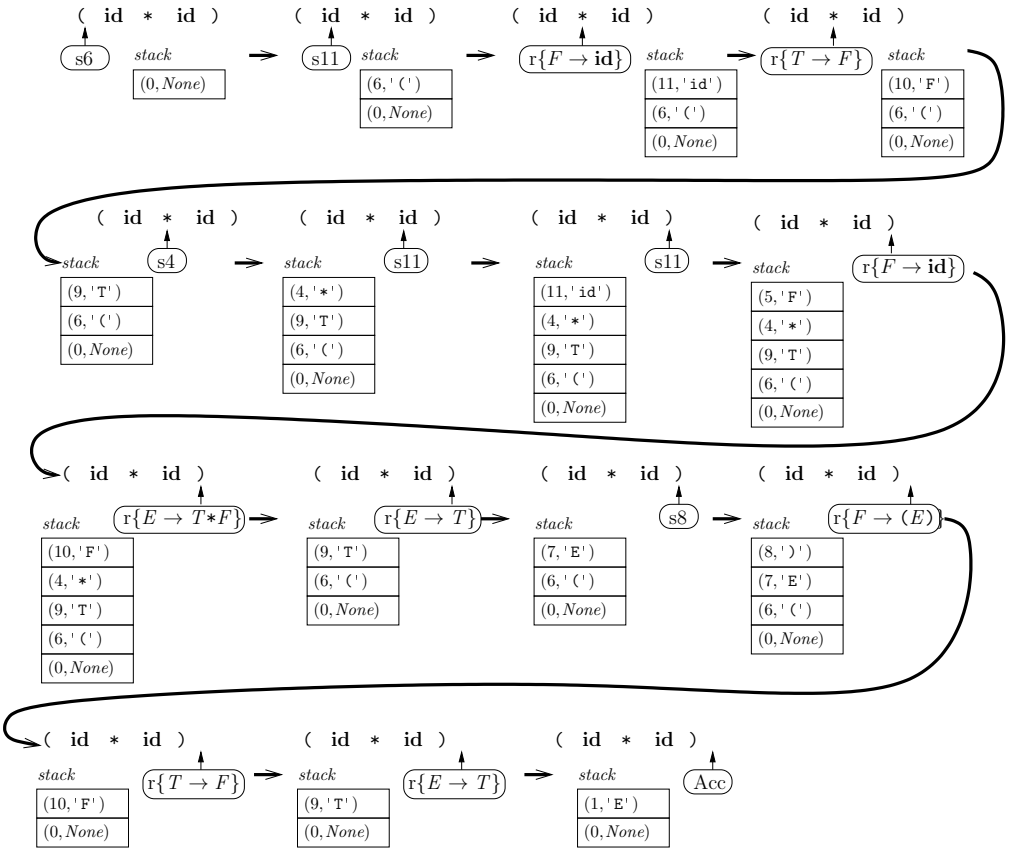


Abb. 6.7: *Darstellung aller Aktionen des Kellerautomaten, um das Wort „* **(id * id)** *“ zu parsen. In jedem Schritt wird der Zustand des Stacks, die momentane Position innerhalb des zu parsenden Wortes und die aus der Syntaxanalysetabelle ausgelesene Aktion (innerhalb des abgerundeten Kästchens) dargestellt. Die Shift-Operationen beginnen mit einem „s“, die Reduce-Operationen mit einem „r“. Bei den Reduce-Schritten wurde statt der Nummer der Produktion (mit der die auf dem Stack befindliche Satzform reduziert werden soll) aus Gründen der besseren Lesbarkeit jeweils gleich die Produktion selbst (statt deren Nummer innerhalb der Produktionenliste* $self.P$*) angegeben.*
Nehmen wir als Beispiel den dritten Schritt: Der Kellerautomat befindet sich immer in dem im obersten Stackelement enthaltenen Zustand, hier also in Zustand E_{11}*; es wird das Zeichen* `*` *gelesen. Der entsprechende Eintrag in der Syntaxanalysetabelle, also* $self.aktionTab[11]['*']$*, ist „r6“ (siehe auch Tabelle 6.3); der besseren Lesbarkeit halber ist die sechste Produktion, also* $self.P[6]$*, ausgeschrieben als „*$F \to \boldsymbol{id}$*“. Diese Reduce-Aktion bewirkt, dass der Stack zunächst um* $|\boldsymbol{id}| = 1$ *erniedrigt wird; mit dem Zustand, den das oberste Stackelement dann enthält, das ist hier der Zustand* E_6*, und mit der linken Seite der zu reduzierenden Produktion, das ist hier F, wird dann in der Sprungtabelle der Folgezustand* $self.sprungTab[6]['F']=10$ *nachgeschlagen; dieser wird zusammen mit F auf den Stack gelegt und mit dem nächsten Schritt fortgefahren.*