```java
import java.util.StringTokenizer;
import java.util.Vector;
import java.lang.Integer;

/**
 * Die Klasse LingDaten ist eine der beiden Hauptklassen des
 * Projektes (die andere ist ShiftReduceParser).<p>
 * Hier werden alle linguistischen Daten gespeichert:
 * <ul>
 * <li> Die Grammatik (aus dem Eingabebereich)
 * <li> Das Lexikon (aus dem Eingabebereich)
 * <li> Der Eingabesatz (aus dem Eingabebereich)
 * <li> Die Aktions- und Sprungtabelle für den Parser, die aus
 * Grammatik und Lexikon erzeugt werden.
 * </ul>
 * Ferner werden mit der Methode 'erzeugeTabellen' aus Grammatik
 * und Lexikon die Aktions- und Sprungtabelle erzeugt. Als
 * Zwischenschritt dieser Erzeugung wird die Zustandsliste erzeugt
 * (Untermethode erzeugeZustaende).
 * @author Michiko Tamakoshi
 * @version     1.0, 9.11.2003
 * @see ShiftReduceParser
 * @see #erzeugeTabellen()
 * @see #erzeugeZustaende()
 */
public class LingDaten
{
      // Daten auf Bildschirm
      /**
       * Satz, wie er im Eingabefeld auf dem Bildschirm
       * erscheint
      */
      String guisatz;
      /**
       * Lexikon, wie es im Eingabefeld auf dem Bildschirm
       * erscheint
      */
      String guilexikon;
      /**
       * Grammatik, wie sie im Eingabefeld auf dem Bildschirm
       * erscheint
      */
      String guigrammatik;

      // formatierte Daten
      /**
       * formatierter Satz: aus dem Satz, wie er im
       * Eingabefeld auf dem Bildschirm erscheint
       * (Attribut 'guisatz'), wird durch Zergliederung
       * in einzelne Wörter (Trennzeichen sind Leerzeichen,
       * Komma etc.) eine Liste der Wörter erzeugt.
       * @see LingDaten#guisatz
      */
      public String[] satz;
      /**
       * formatiertes Lexikon: aus dem Lexikon, wie es im
       * Eingabefeld auf dem Bildschirm erscheint
       * (Attribut 'guilexikon'), wird durch Zergliederung
       * in einzelne Zeilen (Trennzeichen ist das Zeilenende)
       * eine Liste von Lexikoneinträgen erzeugt. Jede dieser
       * Einträge wird selber erneut (wie beim Satz) in
       * einzelne Wörter zerlegt.<p>
       * Das erste Symbol eines Lexikoneintrags ist
       * immer das Terminalsymbol,
```

```
 * dass den Schlüssel des Lexikoneintrags darstellt.
 * @see LingDaten#guilexikon
*/
public String[][] lexikon;
/**
 * formatierte Grammatik: aus der Grammatik, wie sie im
 * Eingabefeld auf dem Bildschirm erscheint
 * (Attribut 'guigrammatik'), wird durch Zergliederung
 * in einzelne Zeilen (Trennzeichen ist das Zeilenende)
 * eine Liste von Regeln erzeugt. Jede dieser
 * Regeln wird selber erneut (wie beim Satz) in
 * einzelne Wörter zerlegt.<p>
 * Das erste Symbol einer Regel ist immer das Regelsymbol,
 * dass die Nummer der Regel repräsentiert.
 * @see LingDaten#guigrammatik
*/
public String[][] grammatik;

// Attribute für Erzeugung
/**
 * Liste der erzeugten Zustände. Diese Liste wird
 * in der Methode 'erzeugeZustaende' erzeugt.<p>
 * Jeder Zustand selbst besteht wiederrum aus
 * einem Kopfelement gefolgt
 * einer Liste (Vector) von sogenannten "Punktregeln",
 * d.h. Regeln der Grammatik mit einem Punkt
 * vor dem in dem Zustand zu lesenden Symbol (Terminal
 * oder Nichtterminal). Im Kopfelement ist das erste
 * Symbol der Zustandsname. Danach folgen, in der Reihenfolge
 * der Punktregeln, die Folgezustände, wenn das Symbol
 * an der Position hinter dem Punkt gelesen wird
 * (Shift oder Goto).
 * @see #erzeugeZustaende()
*/
public Vector zustand;
/**
 * Konstante, die einem Punkt-String entspricht (".")
*/
private static final String punkt = new String(".");
/**
 * Startsymbol der Grammatik. Das Startsymbol wird dadurch
 * bestimmt, dass die erste Regel der Grammatik angeschaut
 * wird. Das (Nichtterminal-)Symbol auf der linken Seite
 * der Produktion wird als Startsymbol gesetzt.
*/
private String startsymbol;
/**
 * Aktionstabelle, die durch die Methode 'erzeugeTabellen'
 * erzeugt wird. Es handelt sich um ein 2-dimensionales
 * Array von Feldern (Typ String). In jedem Feld kann ein
 * oder mehrere Aktionen (wie shift oder reduce) gespeichert
 * sein.<p>
 * Die erste Zeile der Tabelle enthält immer die Liste der
 * Terminalsymbole, für die diese Aktion gelten soll.
 * Der Parser kann aufgrund dieser Kopfzeile die Spalte
 * bestimmen, die für das aktuelle Symbol der Eingabe
 * gelesen werden soll.<p>
 * Wenn eine Grammatik eindeutig ist, steht in jedem Feld
 * der Aktionstabelle immer nur eine Aktion. Ist die Grammatik
 * nicht mehrdeutig, können in einem Feld mehrere Aktionen
 * stehen.
*/
public String[][] aktionstabelle;
/**
```

```
     * Sprungtabelle, die durch die Methode 'erzeugeTabellen'
     * erzeugt wird. Es handelt sich um ein 2-dimensionales
     * Array von Feldern (Typ String). In jedem Feld kann ein
     * oder mehrere Aktionen (nur goto) gespeichert
     * sein.<p>
     * Die erste Zeile der Tabelle enthält immer die Liste der
     * Nichtterminalsymbole, für die diese Aktion gelten soll.
     * Der Parser kann aufgrund dieser Kopfzeile die Spalte
     * bestimmen, die für das aktuelle Symbol (nach einer Reduktion)
     * gelesen werden soll.<p>
     * Wenn eine Grammatik eindeutig ist, steht in jedem Feld
     * der Aktionstabelle immer nur eine Aktion. Ist die Grammatik
     * nicht mehrdeutig, können in einem Feld mehrere Aktionen
     * (Goto-Aktionen) stehen.
    */
    public String[][] sprungtabelle;
    /**
     * Liste der First-Mengen. Eine First-Menge besteht aus
     * dem First-Symbol, sowie einer Menge aus Terminalen,
     * die diesem First-Symbol zugeordnet sind.<p>
     * Diese Menge
     * wird berechnet, indem das First-Symbol solange gemäß
     * der Grammatikregeln expandiert wird, bis man auf
     * Ebene der Terminale angekommen ist. First-Symbole
     * sind die an erster Stelle kommenden Terminale der
     * angewendeten Produktionen.<p>
     * Beispiel: sei NP -> n vt n <br>
     * eine Regel der Grammatik. Dann ist First(NP) = {n}
    */
    private Vector first;
    /**
     * Liste der Follow-Mengen. Eine Follow-Menge besteht aus
     * dem Follow-Symbol, sowie einer Menge aus Terminalen
     * oder Nicht-Terminalen,
     * die diesem Follow-Symbol zugeordnet sind.<p>
     * Diese Menge
     * besteht aus:<p>
     * 1. allen Symbolen, die dem Follow-Symbol aufgrund der
     * Grammatik folgen können. Hierbei handelt es sich um
     * Punktregeln, bei denen der Punkt zwischen zwei Symbolen
     * steht, wobei das erste Symbol das Follow-Symbol ist. <p>
     * Beispiel: sei S -> NP . VP <br>
     * eine Punktregel. Dann ist Follow(NP) = {VP}<p>
     * 2. allen Symbolen, die in der Follow-Menge des Elternknotens
     * der Produktion der Grammatik stehen, wobei das letzte Symbol
     * der Produktion das Follow-Symbol ist. Also: <br>
     * Follow(Follow-Symbol) = Follow(Elternknoten)<p>
     * Beispiel: sei S -> NP VP . und Follow(S) = {präp, $}<br>
     * dann ist Follow(VP) = {präp, $}, da die Followmenge
     * des Elternknotens quasi kopiert wird.
    */
    private Vector follow;

////////////////////////////////////////
// Teil A: Methoden für die Generierung //
////////////////////////////////////////

    /**
     * neues linguistisches Datenobjekt erzeugen. Hierbei
     * werden die linguistischen Eingabedaten vom Bildschirm
     * übernommen. Sie können in der Folge von der Methode
     * 'formatiereDaten' weiterverarbeitet werden.
   * @param   psatz   Satz, wie er im Eingabefeld auf dem Bildschirm
     * erscheint
```

```
     * @param   plexikon   Lexikon, wie es im Eingabefeld auf dem
     * Bildschirm erscheint
     * @param   pgrammatik   Grammatik, wie sie im Eingabefeld auf
     * dem Bildschirm erscheint
      * @see #formatiereDaten()
     */
      public LingDaten(String psatz,String plexikon,
            String pgrammatik)
      {
            guisatz = psatz;
            guilexikon = plexikon;
            guigrammatik = pgrammatik;
      }

      /**
       * Diese Methode räumt das Datenobjekt auf.
       * Hiermit werden jedoch lediglich die Referenzen
       * auf die Eingabedaten zurückgenommen.
       * Dies wird dann benötigt,
       * wenn ein begonnener Parsevorgang beendet ist bzw.
       * der Parsevorgang abgebrochen wird. Ein Beispiel ist,
       * dass der Benutzer
       * den Knopf "Tabellen erzeugen" gedrückt hat.
       * Hierdurch sollen die Eingabedaten erneut verarbeitet
       * und die Tabellen erstellt werden.
    */
      public void raeumeAuf()
      {
            guisatz      = null;
            guilexikon   = null;
            guigrammatik = null;
      }

      /**
       * Diese Methode übernimmt und formatiert
       * den als Parameter übergebenen Eingabesatz vom
       * Bildschirm. Bei der Formatierung werden die lexikalischen
       * Kategorien der Eingabewörter bestimmt.<p>
       * Dies hat dieselbe Funktion wie der Konstruktur
       * und anschliessender Aufruf der Methode 'formatiereDaten',
       * jedoch bezieht sich diese Methode nur auf den Satz.
       * Diese Methode wird aufgerufen, wenn der Benutzer lediglich
       * den Eingabesatz ändert, nicht aber Lexikon oder Grammatik.
     * @param   pguisatz   Satz, wie er im Eingabefeld auf dem
Bildschirm
       * erscheint
       * @see #formatiereDaten()
       * @see #LingDaten(String,String,String)
    */
      public void aktualisiereSatz(String pguisatz)
      {
            guisatz = pguisatz;
            // Satz formatieren
            int i = 0;
            StringTokenizer st = new StringTokenizer
                  (guisatz, ",. ");
            satz = new String[st.countTokens()+1];
            while (st.hasMoreTokens())
            {
                  satz[i] = st.nextToken();
                  i++;
            }
            satz[i] = new String("$");
      }
```

```
  /**
   * Diese Methode formatiert den Eingabesatz,
   * sowie Lexikon und Grammatik im Eingabebereich,
   * die vorher im Konstruktor gesetzt wurden.<p>
   * Formatierung bedeutet, dass die lediglich im
   * Stringformat vorhandenen Daten (Attribute
   * 'guisatz', 'guilexikon', 'guigrammatik') in
   * ein formatiertes Format gebracht werden
   * (Attribute 'satz', 'lexikon', 'grammatik').<p>
   * Siehe weitere Dokumentation bei den Attributen.
   * @see LingDaten#guisatz
   * @see LingDaten#guilexikon
   * @see LingDaten#guigrammatik
   * @see LingDaten#satz
   * @see LingDaten#lexikon
   * @see LingDaten#grammatik
*/
  public void formatiereDaten()
  {
        int i = 0;

        // Satz formatieren
        StringTokenizer st = new StringTokenizer
              (guisatz, ",. ");
        satz = new String[st.countTokens()+1];
        while (st.hasMoreTokens())
        {
              satz[i] = st.nextToken();
              i++;
        }
        satz[i] = new String("$");

        // Lexikon formatieren
        StringTokenizer lex = new StringTokenizer
              (guilexikon, "\n");
        lexikon = new String[lex.countTokens()][];
        int j;
        i=0;
        while (lex.hasMoreTokens())
        {
              StringTokenizer lexeintrag = new StringTokenizer
                    (lex.nextToken(), " ={},");
              lexikon[i] = new String[lexeintrag.countTokens()];
              j=0;
              while (lexeintrag.hasMoreTokens())
              {
                    lexikon[i][j] = lexeintrag.nextToken();
                    j++;
              }
              i++;
        }

        // Grammatik formatieren
        StringTokenizer gramm = new StringTokenizer
              (guigrammatik, "\n");
        grammatik = new String[gramm.countTokens()][];
        i=0;
        while (gramm.hasMoreTokens())
        {
              StringTokenizer grammeintrag = new StringTokenizer
                    (gramm.nextToken(), " ()->");
              grammatik[i] = new String[grammeintrag.countTokens()];
              j=0;
```

```
                while (grammeintrag.hasMoreTokens())
                {
                        grammatik[i][j] = grammeintrag.nextToken();
                        j++;
                }
                i++;
        }
    }

    /**
     * Diese Methode ist die wichtigste Methode
     * (Haupteinstiegspunkt von außen). Sie ermöglicht
     * die Generierung der Aktions- und Sprungtabelle,
     * wenn zuvor Eingabesatz, Lexikon und Grammatik
     * gesetzt und formatiert sind. <p>
     * Die Methode ist High-Level und bedient sich weiterer
     * Methoden zur Erledigung ihrer Aufgaben:
     * <ul>
     * <li> erzeugeZustaende: Erzeugung aller Zustände
     * <li> vervollstaendigeFollow: Follow-Mengen werden
     * vervollständigt
     * <li> erzeugeAktionsGotoReduceTabellen: eigentliches Schreiben
     * der Tabellen
     * </ul>
     * Als erstes werden noch die Attribute
     * 'first' und 'follow' initialisert.
     * Siehe weitere Dokumentation bei den aufgerufenen Methoden.
     * @see LingDaten#first
     * @see LingDaten#follow
     * @see #erzeugeZustaende()
     * @see #vervollstaendigeFollow()
     * @see LingDaten#lexikon
     * @see #erzeugeAktionsGotoReduceTabellen()
*/
    public void erzeugeTabellen()
    {
        first = new Vector();
        follow = new Vector();
        erzeugeZustaende();
        vervollstaendigeFollow();
        erzeugeAktionsGotoReduceTabellen();
    }

    /**
     * Diese Methode erzeugt aus der Grammatik
     * eine Menge von Zuständen, die wiederrum
     * aus einer Liste von Punktregeln besteht.
     * Zur Datenstruktur siehe Dokumentation des
     * Attributes 'zustand'. <p>
     * Diese Methode wird von Methode 'erzeugeTabellen'
     * zuerst aufgerufen, bevor die Tabellengenerierung
     * aufgerufen wird. Dies ist notwendig, da die
     * Tabellengenerierung auf der Zustandsliste basiert.<p>
     * Ablauf der Methode:
     * <ul>
     * <li>Erzeugung des initialen Zustands (Z0)
     * <li>Expansion von Z0 (Aufruf Methode
     * 'expandiereNichtterminal'), Ergebnis ist,
     * dass weitere Punktregeln in Z0 erfasst werden
     * (gemäß Grammatik)
     * <li>Erzeugungsschleife: Die Schleife beginnt beim
     * initialen Zustand Z0 und führt für jede der Punktregeln
     * die gleiche Operation durch: Weiterrücken des Punktes
     * (Aufruf der Methode 'rueckePunktVor'), wodurch neue
```

```
         * Zustände erzeugt oder bestehende Zustände um weitere
         * Punktregeln erweitert werden können.
         * Nach Beendigung von Zustand Z0 wird mit Zustand Z1
         * fortgefahren etc., bis kein weiterer Zustand mehr
         * vorhanden ist. Achtung: Da die Liste der Zustände
         * gleichsam dynamisch wächst, ist das Ende der Schleife
         * auch nicht statisch, sondern nur dynamisch mit der
         * Anzahl der Zustände bestimmt.<p>
         * Am Ende der Methode werden auf Standard-Out noch
         * die Zustände textuell ausgegeben, aus Testgründen.
         * </ul>
         * @see LingDaten#zustand
         * @see #erzeugeTabellen()
         * @see #erzeugeAktionsGotoReduceTabellen()
         * @see #expandiereNichtterminal(String,int,Vector,Vector)
         * @see #rueckePunktVor(String[],Vector)
     */
       private void erzeugeZustaende()
       {
             // initialisiere
             if (zustand == null)
                   zustand = new Vector();
             else
                   zustand.clear();
             startsymbol = grammatik[0][1];

             // erzeuge Zustand 0
             Vector zustand0 = new Vector();
             Vector zustKopf = new Vector();
             String zustName;
             String[] zust;
             zustName = new String("Z0");
             zustKopf.add(zustName);
             zustand0.add(zustKopf);
             zust = new String[3];
             zust[0] = new String(startsymbol + "'");
             zust[1] = punkt;
             zust[2] = startsymbol;
             zustand0.add(zust);
             zustand.add(zustand0);

             // Erzeugungsschleife
             boolean nochZustaendeVorhanden = true;
             int aktZustand = 0;
             int i;
             Vector aktuellerZustand;
             String[] punktregel;

             // prüfe auf follow
             Vector followStartsymbol = new Vector();
             followStartsymbol.add(startsymbol);
             followStartsymbol.add(new String("$"));
             follow.add(followStartsymbol);


       expandiereNichtterminal(startsymbol,0,null,followStartsymbol);
             while (nochZustaendeVorhanden)
             {
                   aktuellerZustand =
(Vector)zustand.elementAt(aktZustand);
                   for (i=1; i<aktuellerZustand.size();i++)
                   {
                         punktregel = (String[])
aktuellerZustand.elementAt(i);
```

```
                        rueckePunktVor(punktregel,aktuellerZustand);
                  }
                  aktZustand++;
                  nochZustaendeVorhanden = (aktZustand <
zustand.size());
            }
            zustaendeAusgeben();
      }

      /**
       * Diese Methode führt die Operation des Weiterrückens
       * des Punktes innerhalb einer Punktregel durch.
       * Zu diesem Zweck werden ihr als Parameter die
       * Punktregel selbst und der aktuelle Zustand, in
       * dem die Punktregel vorhanden ist, übergeben. <p>
       * Nach Weiterrücken des Punktes (dies erzeugt quasi
       * eine neue Punktregel) gibt es 3 Möglichkeiten:
       * <ul>
       * <li>die neue Punktregel existiert bereits in einem
       * Zustand. In diesem Falle wird lediglich die
       * Nummer dieses Folgezustands
       * in das Kopfelement des aktuellen Zustands geschrieben.
       * Siehe auch Struktur der Zustände (Attribut 'zustand').
       * <li>die neue Punktregel existiert zwar nicht, aber
       * es existiert sozusagen eine "ähnliche" Punktregel, d.h.
       * eine Punktregel, die bis zum Punkt identisch mit der
       * neuen Punktregel ist (hinter dem Punkt sich aber
       * unterscheidet, ansonsten wäre sie ja komplett identisch).
       * In diesem Fall muss die neue Punktregel an den Zustand,
       * der die ähnliche Punktregel enthält, angefügt werden.
       * <li>die neue Punktregel existiert in keinem Zustand und
       * es existiert auch keine ähnliche Punktregel. In diesem
       * Fall muss ein neuer Zustand angelegt werden und die
       * neue Punktregel wird diesem Zustand angefügt.
       * </ul><p>
       * Diese Methode wird verwendet von der Methode
       * 'erzeugeZustaende' (siehe weiteres dort).<p>
       * Die Methode sorgt ferner noch dafür, dass eine hinzugefügte
       * Punktregel in dem entsprechenden Zustand auch noch
       * expandiert wird (Aufruf der Methode 'expandiereNichtterminal').
     * @param   punktregel   Punktregel, auf die die Operation
     * des Weiterrückens des Punktes angewendet werden soll
     * @param   aktuellerZustand   Zustand, der diese Punktregel
     * enthält (Parameter punktregel)
       * @see LingDaten#zustand
       * @see #erzeugeZustaende()
       * @see #expandiereNichtterminal(String,int,Vector,Vector)
    */
      private void rueckePunktVor(String[] punktregel,
            Vector aktuellerZustand)
      {
            Vector aktuellerZustKopf =
                  (Vector)aktuellerZustand.elementAt(0);
            if (punktregel[punktregel.length-1].equals(punkt))
            {
                  aktuellerZustKopf.add(new Integer(-1));
                  return;
            }
            String[] neuePunktregel = new String[punktregel.length];
            int i = 0;
            // neue Punktregel erzeugen
            while (i<punktregel.length)
            {
                  if (punktregel[i].equals(punkt))
```

```
	{
		neuePunktregel[i] = punktregel[i+1];
		neuePunktregel[i+1] = punkt;
		// gibt es ein Folgesymbol?
		if (i<punktregel.length-2 &&
			punktregel[i+1].charAt(0)>='A' &&
			punktregel[i+1].charAt(0)<='Z')
			fuegeZuFollowHinzu
				(punktregel[i+1],punktregel[i+2]);
		i=i+2;
	}
	else // nur übernehmen
	{
		neuePunktregel[i] = punktregel[i];
		i++;
	}
}
// prüfe ob Punktregel schon vorhanden
boolean aehnlich=false;
int aehnlicherZustand = 0;
Vector aehnlZustand;
int neuZustand = 0;
for (i=0; i<zustand.size(); i++)
{
	if (!regelNichtVorhanden(neuePunktregel,
		(Vector)zustand.elementAt(i))) // vorhanden?
		{
			aktuellerZustKopf.add(new Integer(i));
			return;
		}
	if (aehnlicheRegelVorhanden(neuePunktregel,
		(Vector)zustand.elementAt(i))) // ähnlich?
	{
		aehnlicherZustand = i;
		aehnlich = true;
		break;
	}
}
if (!aehnlich)
{
	// nun neuen Zustand erzeugen
	neuZustand = zustand.size();
	Vector neuerZustand = new Vector();
	Vector zustName = new Vector();
	String zstName = new String("Z" + neuZustand);
	zustName.add(zstName);
	neuerZustand.add(zustName);
	neuerZustand.add(neuePunktregel);
	zustand.add(neuerZustand);
	aktuellerZustKopf.add(new Integer(neuZustand));
}
else // hinzufügen
{
	aehnlZustand = (Vector)zustand.elementAt
		(aehnlicherZustand);
	aehnlZustand.add(neuePunktregel);
	aktuellerZustKopf.add(new Integer(aehnlicherZustand));
}
i=0;
// Expansion, falls notwendig
// Punkt in neuer Punktregel finden
while (!neuePunktregel[i].equals(punkt))
	i++;
// expandieren? (Nichtterminal und Punkt nicht am Ende)
```

```
            if (neuePunktregel[neuePunktregel.length-1].equals(punkt))
                  return;
            char zeichen = neuePunktregel[i+1].charAt(0);
            if (zeichen<'A' || zeichen>'Z')
                  return;
            if (!aehnlich)
                  expandiereNichtterminal(neuePunktregel[i+1],
                        neuZustand,null,null);
            else
                  expandiereNichtterminal(neuePunktregel[i+1],
                        aehnlicherZustand,null,null);
      }

      /**
       * Diese Methode führt die Operation der Expansion
       * eines Nichtterminals in einem Zustand durch.
       * Parameter sind demzufolge das zu expandierende
       * Nichtterminal und die Nummer des Zustands, in
       * dem das Nichtterminal expandiert werden soll.
       * 2 weitere Parameter werden für den Aufbau der
       * First- und Follow-Mengen benötigt.<p>
       * Die Expansion geschieht anhand der vorhandenen
       * Regeln der Grammatik und rekursiv. Rekursiv
       * deshalb, weil das Nichtterminal auch in mehreren
       * Stufen expandiert werden kann, bis es auf
       * Ebene der Terminale "ankommt".<p>
       * Ablauf der Methode:
       * <ul>
       * <li> Erzeugen einer First-Menge für das Nichtterminal,
       * wenn es noch keine First-Menge gibt (siehe Attribut
       * 'first').
       * <li> Gleiches für die Follow-Menge des Nichtterminals
       * <li> Expansion: es wird eine Regel der Grammatik gesucht,
       * die auf das Nichtterminal paßt. Dann wird versucht,
       * hieraus eine Punktregel zu erzeugen und diese dem
       * aktuellen Zustand hinzuzufügen. Ist diese noch nicht
       * vorhanden, wird rekursiv die Expansion fortgesetzt.
       * </ul>
      * @param   nichtTerminal   Nichtterminal, das expandiert
      * werden soll
      * @param   aktuellerZustand   Zustand, in dem die Expansion
      * durchgeführt werden soll
      * @param   firstzeileOberknoten   bezeichnet die First-Menge
      * des Vaterknotens (Symbol auf der linken Seite, wobei das
      * aktuelle Nichtterminal rechts steht)
      * @param   followzeileOberknoten   entsprechend Follow-Menge
      * des Vaterknotens
       * @see LingDaten#first
       * @see LingDaten#follow
     */
      private void expandiereNichtterminal(String nichtTerminal,
            int aktuellerZustand,Vector firstzeileOberknoten,
            Vector followzeileOberknoten)
      {
            int i,j;
            String[] elmt = null;
            String punkt = new String(".");
            char zeichen = ' ';
            boolean hinzugefuegt = false;
            String np;
            Vector aktZustand =
(Vector)zustand.elementAt(aktuellerZustand);

            // pruefe ob nichtTerminal in first vorhanden
```

```
	Vector firstzeile;
	boolean firstgefunden = false;
	for (i=0; i<first.size(); i++)
	{
		firstzeile = (Vector) first.elementAt(i);
		np = (String)firstzeile.elementAt(0);
		if (np.equals(nichtTerminal))
		{ firstgefunden = true; break; }
	}
	if (!firstgefunden)
	{
		firstzeile = new Vector();
		firstzeile.add(nichtTerminal);
		first.add(firstzeile);
	}
	else
		firstzeile = null;

	// prüfe auf follow
	Vector followzeile;
	if (followzeileOberknoten == null)
		followzeile = neueFollowzeile(nichtTerminal);

	// Expansion
	for (i=0; i<grammatik.length; i++)
	{
		// Regel gefunden?
		if (nichtTerminal.equals(grammatik[i][1]))
			hinzugefuegt = fuegeRegelHinzu(i,aktZustand);
	}
	if (hinzugefuegt)
	{
		for (i=1; i<aktZustand.size(); i++)
		{
			elmt = (String[]) aktZustand.elementAt(i);
			for (j=1; j<elmt.length-1; j++)
			{
				zeichen = elmt[j+1].charAt(0);
				if (!elmt[j].equals(punkt))
					continue;
				// Nichtterminal?
				if (zeichen>='A' && zeichen<='Z')

expandiereNichtterminal(elmt[j+1],aktuellerZustand,
							firstzeile,null);
				else // Terminal

fuegeZuFirstHinzu(firstzeile,elmt[j+1]);
			}
		}
	}
	// first für Oberknoten übernehmen
	if (!firstgefunden && firstzeileOberknoten != null)
	{
		for (i=1; i<firstzeile.size(); i++)

firstzeileOberknoten.add(firstzeile.elementAt(i));
	}
}

/**
 * Ein Symbol der First-Menge, die übergeben wird,
 * hinzufügen.
* @param   firstzeile   First-Menge, zu der das
```

```
 * Symbol hinzugefügt werden soll
 * @param   symbol   Hinzuzufügendes Symbol (Terminal)
   * @see LingDaten#first
*/
  // falls erforderlich Terminal zu Firstliste hinzufügen
  private void fuegeZuFirstHinzu(Vector firstzeile,String symbol)
  {
        int i;
        String elmt;
        // prüfe ob First vorhanden
        if (firstzeile == null)
              return;
        // prüfe ob schon vorhanden
        for (i=1; i<firstzeile.size(); i++)
        {
              elmt = (String)firstzeile.elementAt(i);
              if (elmt.equals(symbol))
                    return;
        }
        firstzeile.add(symbol);
  }

  /**
   * Eine neue Follow-Menge für das übergebene Nichtterminal
   * übergeben, wenn es noch keine Follow-Menge für dieses
   * gibt.
 * @param   nichtTerminal   Nichtterminal, für das
 * die Follow-Menge entweder bestimmt oder neu erzeugt
 * werden soll
   * @see LingDaten#follow
*/
  // falls erforderlich neue Follow-Zeile erzeugen
  private Vector neueFollowzeile(String nichtTerminal)
  {
        int i;
        Vector followzeile;
        String followsymbol;
        for (i=0; i<follow.size(); i++)
        {
              followzeile = (Vector)follow.elementAt(i);
              followsymbol = (String)followzeile.elementAt(0);
              // Followsymbol existiert bereits
              if (followsymbol.equals(nichtTerminal))
                    return followzeile;
        }
        // noch nicht in Followliste, dann füge hinzu
        followzeile = new Vector();
        followzeile.add(nichtTerminal);
        follow.add(followzeile);
        return followzeile;
  }

  /**
   * Ein Symbol der Follow-Menge, die übergeben wird,
   * hinzufügen.
 * @param   followsymbol   Follow-Symbol, zu dessen
 * Follow-Menge das Symbol hinzugefügt werden soll
 * @param   symbol   Hinzuzufügendes Symbol (Terminal
 * oder Nichtterminal)
   * @see LingDaten#first
*/
  // Follow eines Symbols erweitern
  private boolean fuegeZuFollowHinzu(String followsymbol,
        String symbol)
```

```
		{
			Vector followzeile;
			String erstSymbol,aktSymbol;
			boolean geaendert = false;
			int i,j;
			for (i=0; i<follow.size(); i++)
			{
				followzeile = (Vector)follow.elementAt(i);
				erstSymbol = (String)followzeile.elementAt(0);
				if (erstSymbol.equals(followsymbol))
				{
					for (j=1; j<followzeile.size(); j++)
					{
						aktSymbol =
(String)followzeile.elementAt(j);
						if (aktSymbol.equals(symbol))
							return false;
					}
					followzeile.add(symbol);
					return true;
				}
			}
			return false;
		}

	/**
	 * Beim Aufbau der Zustandsliste wird die Follow-Menge
	 * nicht komplett erzeugt: nur die rechts in der Produktion
	 * stehenden Symbole, denen ein weiteres Symbol folgt,
	 * werden dort aufgenommen. Daher wird in dieser Methode
	 * der Rest der Symbole hinzugefügt:
	 * es handelt sich um die Symbole am Ende einer Regel, die
	 * die Followmenge des Vaterknotens übernehmen:<p>
	 * Follow(Kind) = Follow(Vater)<p>
	 * Aufruf der Methode von der Methode 'erzeugeTabellen' aus.
	 * @see LingDaten#follow
	 * @see #erzeugeTabellen()
    */
	// füge hinzu Follows der Form follow(VP)=follow(S)
	private void vervollstaendigeFollow()
	{
		int i;
		String[] zeile;
		String symbol;
		char zeichen;
		boolean geaendert;
		boolean aenderung = true;
		while (aenderung)
		{
			aenderung = false;
			for (i=0; i<grammatik.length; i++)
			{
				zeile = grammatik[i];
				symbol = zeile[zeile.length-1];
				zeichen = symbol.charAt(0);
				// Nichtterminal?
				if (zeichen>='A' && zeichen<='Z')
				{
					geaendert =
uebernehmeFollow(zeile[1],symbol);
					if (geaendert)
						aenderung = true;
				}
			}
```

```
            }
      }

      /**
       * Bei der Vervollständigung der Follow-Menge müssen
       * die Symbole der Follow-Menge des Vaterknotens
       * in die Follow-Menge des Kindknotens übernommen werden,
       * jedoch nur diejenigen, die noch nicht vorhanden sind
       * (Duplikat-Filterung). Dies leistet diese Methode.<p>
       * Aufruf der Methode in 'vervollstaendigeFollow'.
     * @param   linkesSymbol   Vaterknoten, von dessen Follow-Menge
     * übernommen werden soll
     * @param   symbol   Kindknoten, zu dessen Follow-Menge
     * hinzugefügt werden soll
       * @see LingDaten#follow
       * @see #vervollstaendigeFollow()
    */
      private boolean uebernehmeFollow(String linkesSymbol,String
symbol)
      {
            int i,j;
            Vector zeile;
            String erstesSymbol;
            boolean geaendert;
            boolean aenderung = false;
            for (i=0; i<follow.size(); i++)
            {
                  zeile = (Vector)follow.elementAt(i);
                  erstesSymbol = (String)zeile.elementAt(0);
                  if (erstesSymbol.equals(linkesSymbol))
                  {
                        for (j=1; j<zeile.size(); j++)
                        {
                              geaendert =
fuegeZuFollowHinzu(symbol,(String)zeile.elementAt(j));
                              if (geaendert)
                                    aenderung = true;
                        }
                        return aenderung;
                  }
            }
            return false;
      }

      /**
       * Regel aus der Grammatik als Punktregel in einen
       * Zustand aufnehmen, jedoch nur, wenn diese Punktregel
       * noch nicht dort vorhanden ist.<p>
       * Aufruf der Methode von 'expandiereNichtterminal' aus.
     * @param   regelnr   Nummer der Regel der Grammatik,
     * die als Punktregel aufgenommen werden soll
     * @param   aktuellerZustand    aktueller Zustand,
     * zu dem die Punktregel hinzugefügt werden soll
       * @see #expandiereNichtterminal(String,int,Vector,Vector)
    */
      private boolean fuegeRegelHinzu(int regelnr,Vector
aktuellerZustand)
      {
            int i;
            String[] zust = new String[grammatik[regelnr].length];
            zust[0] = grammatik[regelnr][1];
            zust[1] = punkt;
            for (i=2; i<grammatik[regelnr].length; i++)
                  zust[i] = grammatik[regelnr][i];
```

```
        if (regelNichtVorhanden(zust,aktuellerZustand))
        {
              aktuellerZustand.add(zust);
              return true;
        }
        else
              return false;
  }

  /**
   * prüfe ob die übergebene Punktregel im übergebenen
   * Zustand bereits enthalten ist oder nicht.<p>
   * Aufruf der Methode von 'fuegeRegelHinzu' aus.
 * @param   punktregel   Punktregel,
 * die überprüft werden soll
 * @param   aktuellerZustand    aktueller Zustand,
 * für den überprüft werden soll, ob die Punktregel
 * dort vorhanden ist
   * @see #fuegeRegelHinzu(int,Vector)
*/
  // prüfe ob Punktregel in akt. Zustand vorhaden
  private boolean regelNichtVorhanden(String[] punktregel,
        Vector aktuellerZustand)
  {
        int i,j;
        boolean gleich;
        String[] element;
        for (i=1; i<aktuellerZustand.size(); i++)
        {
              gleich = true;
              element = (String[])aktuellerZustand.elementAt(i);
              if (punktregel.length!=element.length)
                    continue;
              for (j=0; j<element.length; j++)
              {
                    if (!element[j].equals(punktregel[j]))
                    { gleich = false; break; }
              }
              if (gleich)
                    return false;
        }
        return true;
  }

  /**
   * prüfe ob eine zur übergebene Punktregel
   * ähnlichen Punktregel im übergebenen
   * Zustand bereits enthalten ist oder nicht.<p>
   * Aufruf der Methode von 'rueckePunktVor' aus.
 * @param   punktregel   Punktregel,
 * die überprüft werden soll
 * @param   aktuellerZustand    aktueller Zustand,
 * für den überprüft werden soll, ob eine ähnliche Punktregel
 * dort bereits vorhanden ist
   * @see #rueckePunktVor(String[],Vector)
*/
  // prüfe ob ähnliche Punktregel in akt. Zustand vorhaden
  private boolean aehnlicheRegelVorhanden(String[] punktregel,
        Vector aktuellerZustand)
  {
        int i,j;
        boolean aehnlich;
        String[] element;
        String punkt = new String(".");
```

```
            for (i=1; i<aktuellerZustand.size(); i++)
            {
                  aehnlich = true;
                  element = (String[])aktuellerZustand.elementAt(i);
                  // if (punktregel.length!=element.length)
                  //    continue;
                  for (j=0; j<element.length; j++)
                  {
                        if (!element[j].equals(punktregel[j]))
                        { aehnlich = false; break; }
                        if (element[j].equals(punkt) ||
                            punktregel[j].equals(punkt))
                            break;
                  }
                  if (aehnlich)
                        return true;
            }
            return false;
      }

      /**
       * Erzeugung der Aktions- und Sprungtabelle.
       * Diese Methode wird von der Methode
       * 'erzeugeTabellen' aufgerufen.<p>
       * Nach Ablauf einer Initialisierung
       * (Reservierung der Tabellen, sowie
       * schreiben der Kopfzeilen der Tabellen)
       * erfolgt die Generierungsschleife:
       * die äußere Schleife iteriert über
       * die Zustände, die innere Schleife
       * über die Punktregeln des aktuellen Zustands
       * (der durch die äußere Schleife bestimmt wird).
       * Für das Tripel aus Punktregel,
       * Zustandsnummer und Nummer der Punktregel
       * in diesem Zustand wird die Methode
       * 'schreibeShiftGotoReduce' aufgerufen, die
       * die tatsächliche Aktion bestimmt.
       * @see LingDaten#aktionstabelle
       * @see LingDaten#sprungtabelle
       * @see #erzeugeTabellen()
       * @see #schreibeShiftGotoReduce(String[],int,int)
    */
      // Aktionstabelle erzeugen
      private void erzeugeAktionsGotoReduceTabellen()
      {
            int i,j;
            Vector zust;
            String[] zeile;
            Vector followzeile;
            // Tabellen initialisieren
            aktionstabelle = new
                  String[zustand.size()+1][lexikon.length+1];
            sprungtabelle = new
                  String[zustand.size()+1][follow.size()];
            // Kopfzeilen erzeugen
            for (i=0; i<lexikon.length; i++)
                  aktionstabelle[0][i] = lexikon[i][0];
            aktionstabelle[0][lexikon.length] = new String("$");
            for (i=0; i<follow.size(); i++)
            {
                  followzeile = (Vector)follow.elementAt(i);
                  sprungtabelle[0][i] =
(String)followzeile.elementAt(0);
            }
```

```
		// Schleife über alle Zustände
		for (i=0; i<zustand.size(); i++)
		{
			zust = (Vector)zustand.elementAt(i);
			for (j=1; j<zust.size(); j++)
			{
				zeile = (String[])zust.elementAt(j);
				schreibeShiftGotoReduce(zeile,i,j);
			}
		}
	}

	/**
	 * Für einen gegebenen Zustand und eine gegebene Punktzeile,
	 * schreibe eines der möglichen Aktionen Shift, Goto
	 * oder Reduce in die Aktions- oder Sprungtabelle.<p>
	 * Die Methode steuert die verschiedenen Aktionstypen
	 * auseinander auf Basis folgender Überlegung:
	 * <ul>
	 * <li> Ein Punkt vor einem Nichtterminal --> Goto
	 * <li> Ein Punkt vor einem Terminal --> Shift
	 * <li> Ein Punkt am Regelende --> Reduce
	 * </ul>
	 * Abhängig vom Ergebnis der Prüfung werden die
	 * Methoden 'schreibeGoto', 'schreibeShift' oder
	 * 'schreibeReduce' aufgerufen.
	* @param   zeile   Punktregel,
	* die überprüft werden soll
	* @param   zustand aktueller Zustand,
	* in dem sich die Punktregel befindet
	* @param   nrPunktregel  Nummer der Punktregel innerhalb
	* des aktuellen Zustands
	 * @see #schreibeGoto(int,String,int)
	 * @see #schreibeShift(int,String,int)
	 * @see #schreibeReduce(int,String,int)
	*/
	private void schreibeShiftGotoReduce(String[] zeile,int zustand,
		int nrPunktregel)
	{
		int i;
		String symbol;
		char zeichen;
		for (i=1; i<zeile.length-1; i++)
		{
			symbol = (String)zeile[i];
			if (symbol.equals(punkt))
			{
				symbol = (String)zeile[i+1];
				zeichen = symbol.charAt(0);
				if (zeichen >= 'A' && zeichen <= 'Z')
					schreibeGoto(zustand,symbol,nrPunktregel);
				else

	schreibeShift(zustand,symbol,nrPunktregel);
			}
		}
		// prüfe auf Punkt am Schluss = Reduce
		symbol = zeile[0];
		if (zeile[zeile.length-1].equals(punkt))
			schreibeReduce(zustand,symbol,nrPunktregel);
	}

	/**
	 * Für einen gegebenen Zustand und eine gegebene Punktzeile,
```

```
   * schreibe eine Shift-Aktion für das übergebene Symbol
   * in die Aktionstabelle. Das Symbol ist das Symbol
   * der Punktregel, welches hinter dem Punkt steht.<p>
   * Ablauf:<br>
   * 1. Finden der Spaltennummer der Aktionstabelle, in der
   * das Symbol zu finden ist.<br>
   * 2. Bereitstellen der Nummer des Folgezustands (steht
   * im Kopfelement des aktuellen Zustands, siehe
   * Dokumentation des Attributs 'zustand').<br>
   * 3. Schreiben des Shift-Eintrags. Hierbei muss berücksichtigt
   * werden, dass bei Mehrdeutigkeit mehrere Einträge im Feld
   * der Tabelle möglich sind. Allerdings darf keine Aktion
   * doppelt geschrieben werden.
 * @param   zust    Nummer des aktuellen Zustands,
 * in dem sich die Punktregel befindet
 * @param   symbol   Symbol, das "geshiftet" werden soll,
 * d.h. dass als nächstes in der Eingabe stehen muss.
 * @param   nrPunktregel  Nummer der Punktregel innerhalb
 * des aktuellen Zustands
   * @see LingDaten#zustand
*/
  private void schreibeShift(int zust, String symbol,
        int nrPunktregel)
  {
        int i,spalte = 0;
        Vector kopfelement;
        Integer folgezustand;
        Vector zst = (Vector)zustand.elementAt(zust);
        for (i=0; i<lexikon.length; i++)
              if (aktionstabelle[0][i].equals(symbol))
              { spalte = i; break; }
        kopfelement = (Vector)zst.elementAt(0);
        folgezustand = (Integer)kopfelement.elementAt(nrPunktregel);
        // prüfe ob bereits geschrieben oder auf Mehrdeutigkeit
        String feld = aktionstabelle[zust+1][spalte];
        if (feld == null)
        {
              aktionstabelle[zust+1][spalte] =
                    new String("sh" + folgezustand.intValue());
              return;
        }
        StringTokenizer st = new StringTokenizer(feld,",");
        String neuesShift =
              new String("sh" + folgezustand.intValue());
        while (st.hasMoreTokens())
              if (st.nextToken().equals(neuesShift))
                    return;
        aktionstabelle[zust+1][spalte] =
              new String(feld + ",sh" + folgezustand.intValue());
  }

  /**
   * Für einen gegebenen Zustand und eine gegebene Punktzeile,
   * schreibe eine Goto-Aktion für das übergebene Symbol
   * in die Sprungtabelle. Das Symbol ist das Symbol
   * der Punktregel, welches hinter dem Punkt steht.<p>
   * Ablauf:<br>
   * 1. Finden der Spaltennummer der Sprungtabelle, in der
   * das Symbol zu finden ist.<br>
   * 2. Bereitstellen der Nummer des Folgezustands (steht
   * im Kopfelement des aktuellen Zustands, siehe
   * Dokumentation des Attributs 'zustand').<br>
   * 3. Schreiben des Goto-Eintrags. Hierbei muss berücksichtigt
   * werden, dass bei Mehrdeutigkeit mehrere Einträge im Feld
```

```
     * der Tabelle möglich sind. Allerdings darf keine Aktion
     * doppelt geschrieben werden.
   * @param   zust    Nummer des aktuellen Zustands,
   * in dem sich die Punktregel befindet
   * @param   symbol   Symbol, für das "gesprungen" werden soll,
   * d.h. dass aktuell im Arbeitsbereich vorne steht und vor
   * dem der Punkt in der Punktregel steht
   * @param   nrPunktregel  Nummer der Punktregel innerhalb
   * des aktuellen Zustands
     * @see LingDaten#zustand
  */
    private void schreibeGoto(int zust, String symbol,
          int nrPunktregel)
    {
          int i,spalte = 0;
          Vector kopfelement;
          Integer folgezustand;
          Vector zst = (Vector)zustand.elementAt(zust);
          for (i=0; i<follow.size(); i++)
                if (sprungtabelle[0][i].equals(symbol))
                { spalte = i; break; }
          kopfelement = (Vector)zst.elementAt(0);
          folgezustand = (Integer)kopfelement.elementAt(nrPunktregel);
          // prüfe ob bereits geschrieben oder auf Mehrdeutigkeit
          String feld = sprungtabelle[zust+1][spalte];
          if (feld == null)
          {
                sprungtabelle[zust+1][spalte] =
                      new String("go" + folgezustand.intValue());
                return;
          }
          StringTokenizer st = new StringTokenizer(feld,",");
          String neuesGoto =
                new String("go" + folgezustand.intValue());
          while (st.hasMoreTokens())
                if (st.nextToken().equals(neuesGoto))
                      return;
          sprungtabelle[zust+1][spalte] =
                new String(feld + ",go" + folgezustand.intValue());
    }

    /**
     * Für einen gegebenen Zustand und eine gegebene Punktzeile,
     * schreibe eine Reduce-Aktion für das übergebene Symbol
     * in die Aktionstabelle. Das Symbol ist das Symbol
     * auf der linken Seite der Punktregel (Produktion).<p>
     * Ablauf:<br>
     * 1. Initialisierung<br>
     * 2. Ermitteln der Menge First(Follow(symbol)): hierzu
     * wird zuerst die Follow-Menge des Symbols gefunden.
     * Danach wird für jedes Symbol der Follow-Menge die
     * First-Menge dieses Symbols der neuen Menge
     * First(Follow(symbol)) hinzugefügt. Hierbei müssen
     * aber noch Duplikate gefiltert werden.<br>
     * 3. Bestimmen der Regelnummer für die Punktregel anhand der
     * Grammatik<br>
     * 4. Schreiben von Reduce (Schritte 1. bis 3. waren
     * Voraussetzungen hierzu): für jedes Symbol der
     * Menge First(Follow(symbol)) wird die Aktion Reduce mit
     * der Regelnummer der Grammatik geschrieben.
     * Hierbei muss noch die Sonderregelgung für die Aktion
     * "acc" (accept) berücksichtigt werden.
   * @param   zust    Nummer des aktuellen Zustands,
   * in dem sich die Punktregel befindet
```

```
 * @param   symbol   Symbol, das reduziert, d.h.
 * das Reduktionssymbol, das auf der linken Seite
 * der Produktion steht.
 * @param   nrPunktregel  Nummer der Punktregel innerhalb
 * des aktuellen Zustands
   * @see LingDaten#first
   * @see LingDaten#follow
   * @see LingDaten#grammatik
*/
  private void schreibeReduce(int zust, String symbol,
        int nrPunktregel)
  {
        Vector firstVonFollow = new Vector();
        Vector followzeile = null;
        Vector firstzeile;
        String followsymbol;
        String firstsymbol;
        String sym;
        String string = new String("$");
        String feld;
        Vector zst = (Vector)zustand.elementAt(zust);
        String[] punktregel = (String[])zst.elementAt(nrPunktregel);
        int i,j;
        // Finde richtige Follow-Zeile
        for (i=0; i<follow.size(); i++)
        {
              followzeile = (Vector)follow.elementAt(i);
              if (symbol.equals((String)followzeile.elementAt(0)))
                    break;
              followzeile = null;
        }
        // Sonderfall: für Startsymbol' verwende
        //    die Follow-Menge des Startsymbols
        if (followzeile == null)
              followzeile = neueFollowzeile
                    (symbol.substring(0,symbol.length()-1));

        // prüfe alle Follow-Symbole
        char zeichen = ' ';
        for (i=1; i<followzeile.size(); i++)
        {
              followsymbol = (String)followzeile.elementAt(i);
              // Fall 1: followsymbol ist Terminal
              zeichen = followsymbol.charAt(0);
              if ((zeichen >= 'a' && zeichen <='z') ||
                   (zeichen == '$'))
                    if (!symbolInListe(firstVonFollow,followsymbol))
                    {
                          firstVonFollow.add(followsymbol);
                          continue;
                    }
              // Fall 2: followsymbol ist Nichtterminal
              for (j=0; j<first.size(); j++)
              {
                    firstzeile = (Vector)first.elementAt(j);
                    firstsymbol = (String)firstzeile.elementAt(0);
                    if (followsymbol.equals(firstsymbol))
                    {
                          fuegeZuFirstVonFollowHinzu
                                (firstVonFollow,firstzeile);
                          break;
                    }
              }
        }
```

```
            // Finde Regel in Grammatik
            int reducenr = -1;
            for (i=0; i<grammatik.length; i++)
            {
                  if (gleich(punktregel,grammatik[i]))
                  {
                        reducenr = i;
                        break;
                  }
            }
            // Schreibe Reduce
            for (i=0; i<firstVonFollow.size(); i++)
            {
                  sym = (String)firstVonFollow.elementAt(i);
                  for (j=0; j<aktionstabelle[0].length; j++)
                        if (sym.equals(aktionstabelle[0][j]))
                        {
                              // Startsymbol?
                              if (reducenr==-1)
                                    aktionstabelle[zust+1][j] =
                                          new String("acc");
                              else
                              // reduce
                              {
                                    feld = aktionstabelle[zust+1][j];
                                    if (feld == null)
                                          aktionstabelle[zust+1][j] =
                                                new String("re" +
(reducenr+1));
                                    else // Mehrdeutigkeit
                                          aktionstabelle[zust+1][j] =
new
                                                String(feld + ",re" +
(reducenr+1));
                              }
                              break;
                        }
            }
      }

      /**
       * Hilfsmethode: prüft, ob eine Punktregel gleich einer
       * Regel der Grammatik ist (ohne Berücksichtigung des
       * Punktes)
     * @param   punktregel   Punktregel, die auf Gleichheit
     * geprüft werden soll
     * @param   grammatikregel      Regel der Grammatik, die auf
     * Gleichheit geprüft werden soll
    */
      // Punktregel und Grammatikregel gleich?
      private boolean gleich(String[] punktregel,
            String[] grammatikregel)
      {
            int i;
            int diff = 0;
            if (punktregel.length!=grammatikregel.length)
                  return false;
            for (i=1; i<grammatikregel.length; i++)
            {
                  if (punktregel[i-1].equals(punkt))
                  { diff = 1; continue; }
                  if (!grammatikregel[i].equals(punktregel[i-1+diff]))
                        return false;
            }
```

```
		return true;
	}

	/**
	 * Hilfsmethode: Fügt den Inhalt (d.h. die Symbole) einer
	 * First-Menge der übergegebenen Menge First(Follow(symbol))
	 * hinzu (repräsentiert durch einen Vector).
	 * Hierbei werden Duplikate gefiltert.
	* @param   firstVonFollow   First(Follow(symbol))-Menge,
	* zu der die Symbole hinzugefügt werden sollen
	* @param   firstzeile    First-Menge, deren Symbole
	* hinzugefügt werden sollen
	*/
	private void fuegeZuFirstVonFollowHinzu
		(Vector firstVonFollow, Vector firstzeile)
	{
		int i;
		String symbol;
		for (i=1; i<firstzeile.size(); i++)
		{
			symbol = (String)firstzeile.elementAt(i);
			if (!symbolInListe(firstVonFollow,symbol))
				firstVonFollow.add(symbol);
		}
	}

	/**
	 * Hilfsmethode: prüft, ob das übergebene Symbol in der
	 * Liste (Vector) enthalten ist oder nicht.
	* @param   liste   Liste von Symbolen, die geprüft werden
	* soll
	* @param   symbol  Symbol, dass auf Vorhandensein in der
	* Liste geprüft werden soll
	* @return  Ergebnis der Prüfung: wahr, wenn Symbol in der
	* Liste enthalten ist
	*/
	// Symbol in Liste vorhanden?
	private boolean symbolInListe(Vector liste,String symbol)
	{
		int i;
		String sym;
		for (i=0; i<liste.size(); i++)
		{
			sym = (String)liste.elementAt(i);
			if (sym.equals(symbol))
				return true;
		}
		return false;
	}

	/**
	 * Hilfsmethode (für Testzwecke): gibt alle Zustände auf
	 * Standard out aus.
	*/
	// alle Zustände ausgeben
	private void zustaendeAusgeben()
	{
		int i,j,k;
		Vector zust;
		Vector zustName;
		String[] punktregel = null;
		for (i=0; i<zustand.size(); i++)
		{
			zust = (Vector)zustand.elementAt(i);
```

```
                  zustName = (Vector)zust.elementAt(0);
                  System.out.println((String)zustName.elementAt(0));
                  for (j=1; j<zust.size(); j++)
                  {
                        punktregel = (String[])zust.elementAt(j);
                        for (k=0; k<punktregel.length; k++)
                              System.out.print(punktregel[k] + " ");
                        System.out.println("");
                  }
            }
      }

////////////////////////////////////////////////////
// Teil B: Service-Methoden für Parser oder Applet //
////////////////////////////////////////////////////

      /**
       * Service-Methode für den Parser: gibt die Spalte der
       * Aktionstabelle zurück, in deren Überschrift sich das
       * übergebene Symbol befindet.
     * @param   symbol  Symbol, dessen Spalte (aufgrund der
     * Überschrift) bestimmt werden soll
     * @return  Spalte, deren Überschrift das Symbol enthält.
     * -1, wenn nicht gefunden.
    */
      public int sucheEingabesymbolInAktionstabelle(String symbol)
      {
            int i;
            for (i=0; i<aktionstabelle[0].length; i++)
                  if (aktionstabelle[0][i].equals(symbol))
                        return i;
            return -1;
      }

      /**
       * Service-Methode für den Parser: gibt die Spalte der
       * Sprungtabelle zurück, in deren Überschrift sich das
       * übergebene Symbol befindet.
     * @param   symbol  Symbol, dessen Spalte (aufgrund der
     * Überschrift) bestimmt werden soll
     * @return  Spalte, deren Überschrift das Symbol enthält.
     * -1, wenn nicht gefunden.
    */
      public int sucheSymbolInSprungtabelle(String symbol)
      {
            int i;
            for (i=0; i<sprungtabelle[0].length; i++)
                  if (sprungtabelle[0][i].equals(symbol))
                        return i;
            return -1;
      }

      /**
       * Service-Methode für das Applet: listet alle Zustände
       * auf und gibt diese Liste in Form eines einzigen
       * Strings zurück. Dieser kann dann in einem Fenster
       * ausgegeben werden.
     * @return  Text als Liste aller Zustände
    */
      // Zustände aufzählen und als String zurückliefern
      public String holeZustandsliste()
      {
            StringBuffer liste = new StringBuffer(500);
            int i,j,k;
```

```
                Vector zust;
                String[] punktregel;
                for (i=0; i<zustand.size(); i++)
                {
                      zust = (Vector)zustand.elementAt(i);
                      liste.append("Zustand ");
                      liste.append(i);
                      liste.append(":\n");
                      for (j=1; j<zust.size(); j++)
                      {
                            punktregel = (String[])zust.elementAt(j);
                            for (k=0; k<punktregel.length; k++)
                            {
                                  liste.append(punktregel[k]);
                                  if (k==0)
                                        liste.append(" ->");
                                  liste.append(' ');
                            }
                            liste.append('\n');
                      }
                      liste.append('\n');
                }
                return liste.toString();
          }
}
```