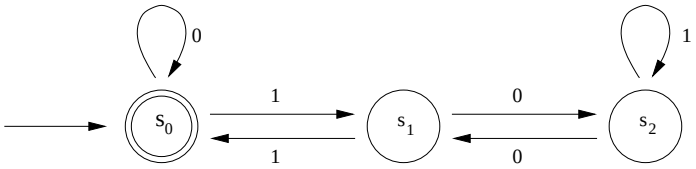
Technische Universität München
Fakultät für Informatik
Dr. M. Schneider, Dr. M. Pizka,
Dr. A. Vilbig, Dr. R. Letz,
G. Bauer

SS 2003
Lösungsvorschläge zu Blatt 3
2. Mai 2003

Übungen zur Vorlesung Einführung in die Informatik IV

Aufgabe 9 Endliche Automaten und reguläre Grammatiken

(a) Gegeben sei der folgende endliche Automat A:



Aus einem (deterministischen) ε-freien endlichen Automaten $A = (S, T, s_0, S_Z, \delta)$, kann man nach folgendem Schema eine reduktive linkslineare Grammatik $G = (T, N, \rightarrow, Z)$ konstruieren, so dass $L(G) = L(A) \backslash \varepsilon$ gilt:

- Für jeden Zustand führen wir ein Nichtterminal ein, $N = S$.
- Für jeden Übergang $\delta(s_0, a) = s_i$ führen wir eine Regel $a \rightarrow s_i$ ein.
- Für jeden Übergang $\delta(s_i, a) = s_j$ führen wir eine Regel $s_i a \rightarrow s_j$ ein.
- Zusätzlich führen wir das Axiom Z ein und für jeden akzeptierenden Zustand $s_i \in S_Z$ die Regel $s_i \rightarrow Z$.

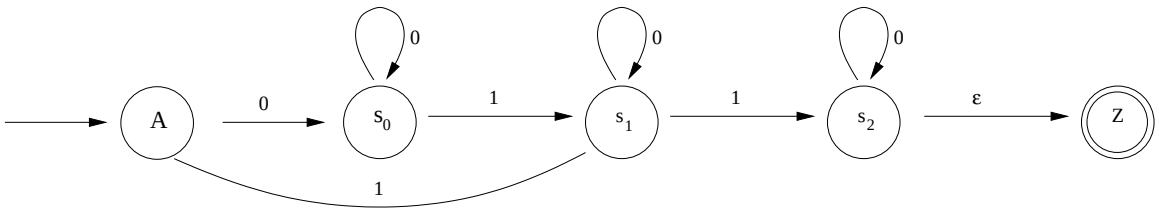
Somit ergibt sich die Grammatik $G = (T, N, \rightarrow, Z)$ wobei

$$\begin{aligned} T &= \{0, 1\} \\ N &= \{s_0, \ldots, s_2, Z\} \\ \rightarrow &= \{0 \rightarrow s_0, \\ &\quad 1 \rightarrow s_1, \\ &\quad s_0 0 \rightarrow s_0 \\ &\quad s_0 1 \rightarrow s_1 \\ &\quad s_1 0 \rightarrow s_2 \\ &\quad s_1 1 \rightarrow s_0 \\ &\quad s_2 0 \rightarrow s_1 \\ &\quad s_2 1 \rightarrow s_2 \\ &\quad s_0 \rightarrow Z\} \end{aligned}$$

(b) Aus einer reduktiven linkslinearen Grammatik $G = (T, N, \rightarrow, Z)$ kann man nach folgendem Schema einem (deterministischen) ε-freien endlichen Automaten $A = (S, T, s_0, S_Z, \delta)$, konstruieren, so dass $L(G) = L(A)$ gilt:

- Für jedes Nichtterminal führen wir einen Zustand ein und zusätzlich einen Anfansgzustand A, also $S = N \cup \{A\}$.
- Für jede Regel $s_i a \to s_j$ führen wir einen Übergang $\delta(s_i, a) = s_j$ ein.
- Für jede Regel $s_i \to s_j$ führen wir einen Übergang $\delta(s_i, \varepsilon) = s_j$ ein.
- Für jede Regel $a \to s_j$ führen wir einen Übergang $\delta(A, a) = s_j$ ein.
- Als Endzustand wählen wir Z.

Der folgende Automat akzeptiert dieselbe Sprache wie G.



Aufgabe 10 **Endliche Automaten und reguläre Ausdrücke**

(a)

$$\begin{aligned} \varepsilon|s|ss^*s &= s^* \\ t|ss^*t &= s^*t \\ t|ts^*s &= ts^* \\ (ts^*s)\{\} &= \{\} \\ \{\}|(ts^*s) &= ts^*s \end{aligned}$$

(b) Der Algorithmus berechnet den entsprechenden Ausdruck nach der Formel

(i) $\quad E(i,j,0) = a_1|\cdots|a_q, \qquad$ wobei für alle $a \in \{a_1,\ldots,a_q\}$ gilt $\delta(s_i,a) = \{s_j\}$

(ii) $\quad E(i,j,k+1) = E(i,j,k) \mid E(i,k+1,k)\, E(k+1,k+1,k)^*\, E(k+1,j,k).$

$E(i,j,k)$ steht für den regulären Ausdruck der Menge der Wörter mit denen man von Zustand s_i zum Zustand s_j über Zwischenzustände nur aus $\{s_1,\ldots,s_k\}$ gelangen kann.

Falls $s_2 \notin S_Z$ wird die vom Automaten akzeptierte Sprache durch den regulären Ausdruck

$$RA(A) = E(1,z_1,n)|\ldots|E(1,z_e,n)$$

beschrieben.

Falls $s_2 \in S_Z$ wird die vom Automaten akzeptierte Sprache durch den regulären Ausdruck

$$RA(A) = E(1,z_1,n)|\ldots|E(1,z_e,n)|\varepsilon$$

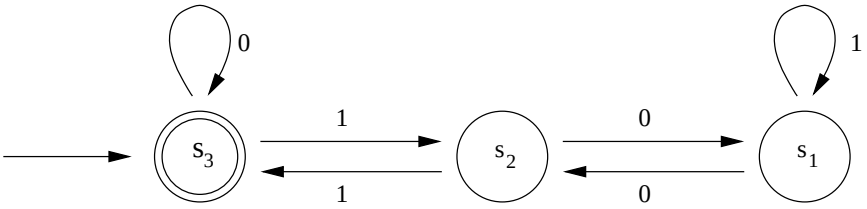
beschrieben.

Hierbei ist s_1 der Anfangszustand, $\{s_{z_1},\ldots s_{z_e}\} = S_Z$ die Menge der Endzustände und n die Anzahl aller Zustände.

Die entstehenden regulären Ausdrücke sind abhängig von der Reihenfolge, in der die Zustände durchlaufen werden. Sie sind jedoch alle äquivalent.

In diesem Fall ist der entstehende reguläre Ausdruck am einfachsten, wenn die Zustände von rechts nach links durchlaufen werden. (Der beispielsweise beim Durchlaufen von links nach rechts entstehende reguläre Ausdruck ist viel komplizierter.)

Dazu müssen die Zustände umnummeriert werden, denn der Algorithmus durchläuft die Zustände von s_1 bis s_n.



Es werden die folgenden Ausdrücke berechnet, wobei nicht alle für das Ergebnis gebraucht werden:

$$
\begin{array}{lll}
E(1,1,0)=1 & E(2,1,0)=0 & E(3,1,0)=\varnothing \\
E(1,2,0)=0 & E(2,2,0)=\varnothing & E(3,2,0)=1 \\
E(1,3,0)=\varnothing & E(2,3,0)=1 & E(3,3,0)=0
\end{array}
$$

$$
\begin{array}{lll}
E(1,1,1)=\ldots & E(2,1,1)=\ldots & E(3,1,1)=\ldots \\
E(1,2,1)=\ldots & E(2,2,1)=01^*0 & E(3,2,1)=1 \\
E(1,3,1)=\ldots & E(2,3,1)=1 & E(3,3,1)=0
\end{array}
$$

$$
\begin{array}{lll}
E(1,1,2)=\ldots & E(2,1,2)=\ldots & E(3,1,2)=\ldots \\
E(1,2,2)=\ldots & E(2,2,2)=\ldots & E(3,2,2)=\ldots \\
E(1,3,2)=\ldots & E(2,3,2)=\ldots & E(3,3,2)=0|1[01^*0]^*1
\end{array}
$$

$$
\begin{array}{lll}
E(1,1,3)=\ldots & E(2,1,3)=\ldots & E(3,1,3)=\ldots \\
E(1,2,3)=\ldots & E(2,2,3)=\ldots & E(3,2,3)=\ldots \\
E(1,3,3)=\ldots & E(2,3,3)=\ldots & E(3,3,3)=(0|1[01^*0]^*1)\ (0|1[01^*0]^*1)^*
\end{array}
$$

Der reguläre Ausdruck, der die vom Automaten A akzeptierte Sprache beschreibt, ist also:

$$RA(A) = [0 \mid 1\ [01^*0]^*\ 1]\ \ [0 \mid 1\ [01^*0]^*\ 1]^* \mid \varepsilon = [0 \mid 1\ [01^*0]^*\ 1]^*$$

Den folgenden regulären Ausdruck erhält man, wenn man die Zustände von links nach rechts durchläuft:

$$RA(A) = 0^* \mid 0^*\ 1\ [10^*1]^*\ 1\ 0^* \mid 0^*\ 1\ [10^*1]^*\ 0\ [1 \mid 0\ [10^*1]^*\ 0]^*\ 0\ [10^*1]^*\ 1\ 0^*$$

(c) Der Algorithmus lässt sich in das folgende Gofer-Programm umsetzen: reguläre Ausdrücke werden einfach als Strings kodiert.

Zunächst werden Operatoren für das Hintereinanderschreiben von regulären Ausdrücken (#) und für alternative reguläre Ausdrücke (|), sowie der Kleene-Operator (*) definiert.

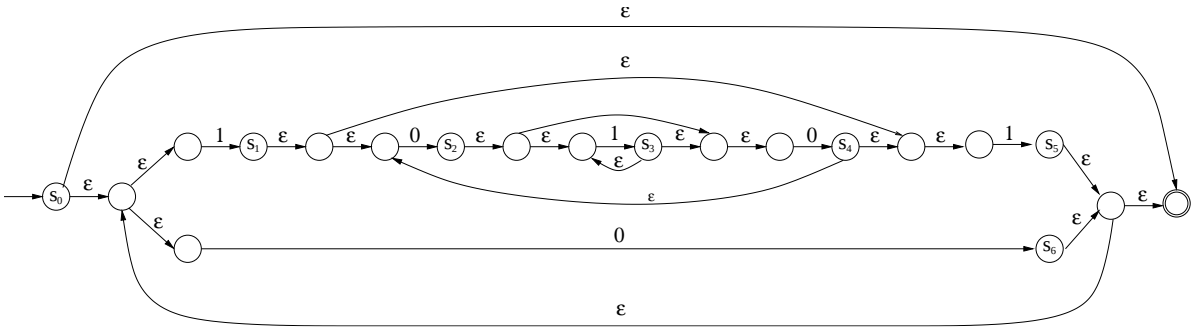
Bei der Berechnung der rekursiven Ausdrücke kann man die folgende Vereinfachung verwenden:

$$E(i,j,k+1) = \begin{cases} E(i,j,k)^* , & \text{falls } i = j = k+1 \\ E(k+1,k+1,k)^* E(k+1,j,k) , & \text{falls } i = k+1 \\ E(i,k+1,k) E(k+1,k+1,k)^* , & \text{falls } j = j+1 \\ E(i,j,k) | E(i,k+1,k) E(k+1,k+1,k)^* E(k+1,j,k) , & \text{sonst.} \end{cases}$$

```
-- Gegeben die Übergangsfunktion des Automaten
a :: (Int,Int) -> String
a(1,1) = "1"
a(1,2) = "0"
a(1,3) = ""
a(2,1) = "0"
a(2,2) = ""
a(2,3) = "1"
a(3,1) = ""
a(3,2) = "1"
a(3,3) = "0"

-- der reguläre Ausdruck s t
(#) :: String -> String -> String
"" # s = ""
s # "" = ""
s # t = s ++ " " ++  t

-- der regulären Ausdruck s | t
alt :: String -> String -> String
"" `alt` s = s
s `alt` "" = s
s `alt` t = s ++ " | " ++ t

-- der reguläre Ausdruck s*
k :: String -> String
k s = "[" ++ s ++ "]" ++ "*"

-- der zum Automat äquivalente reguläre Ausdruck
e :: (Int,Int,Int) -> String
e (i,j,0) = a (i,j)
e (i,j,n+1) | i == j && i == n+1 = k (e(n+1,n+1,n))
            | i == n+1 = k(e(n+1,n+1,n)) # e(n+1,j,n)
            | j == n+1 = e(i,n+1,n) # k(e(n+1,n+1,n))
            | otherwise = e(i,j,n) `alt`
                          ((e(i,n+1,n) # k(e(n+1,n+1,n))) # e(n+1,j,n))

-- zwei Testfunktionen
test = "\n"++e(3,3,3)++"\n"

testall = concat ["\n"++"e("++show a++","++show b++","++show c++") = "
                  ++e(a,b,c) | c<-[0..3], a<-[1..3], b<-[1..3] ]
```

Aufgabe 11 **Reguläre Ausdrücke und endliche Automaten**

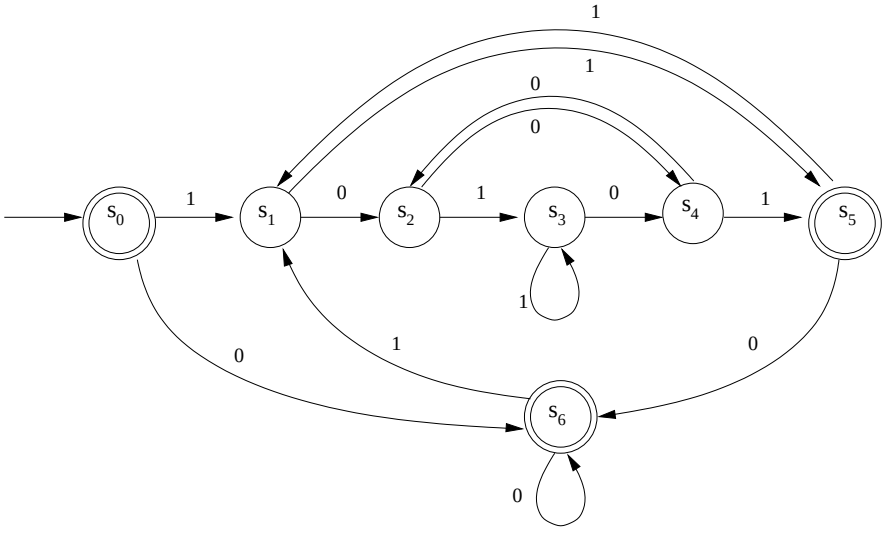
(a) Unter Verwendung der aus der Vorlesung bekannten Vorgehensweise erhalten wir folgenden endlichen Automaten:



(b) Die Eliminierung von ε-Übergängen erfolgt nach folgendem Prinzip:

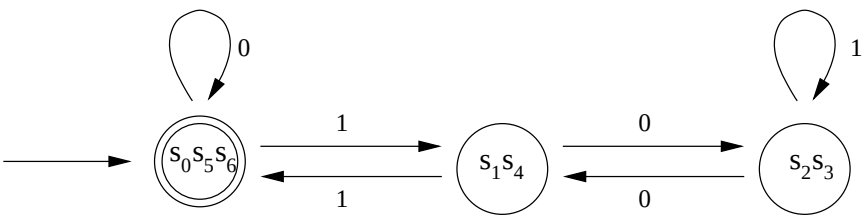
- Zusammenfassen von Knoten auf ε-Zyklen zu Äquivalenzklassen
- Weglassen von ε-Schlingen
- Durchschalten von ε-Übergängen. Wir betrachten einen Knoten k, von dem keine ε-Übergänge ausgehen, an dem jedoch ε-Übergänge enden. Geht von einem Knoten i ein ε-Übergang zu einem solchen Knoten k und ein weiterer Übergang von k mit der Markierung a zu einem Knoten j, so ersetzen wir diesen Pfad durch eine Kante mit der Markierung a von i zu j. Ist k Endzustand, so wird auch i Endzustand. Auf diese Weise eliminieren wir sukzessive die ε-Übergänge.

Dadurch ergibt sich folgender Automat:



Dieser Automat ist bereits deterministisch; im Allgemeinen ist dies nicht der Fall.

(c) Diesen Automaten kann man noch minimieren, indem man Zustände, die gleiches Verhalten haben zu einem zusammenfasst. Man erhält wieder den folgenden Automat:

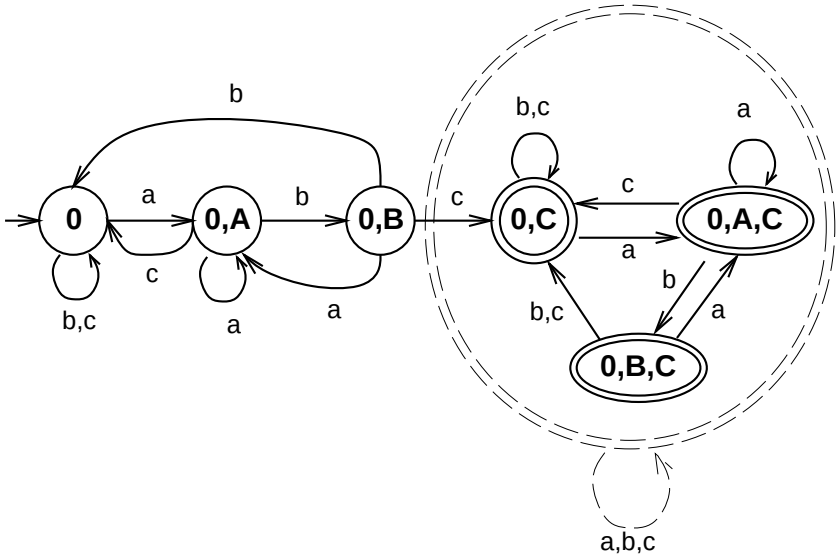


Aufgabe 12 **(Lösungsvorschlag)**

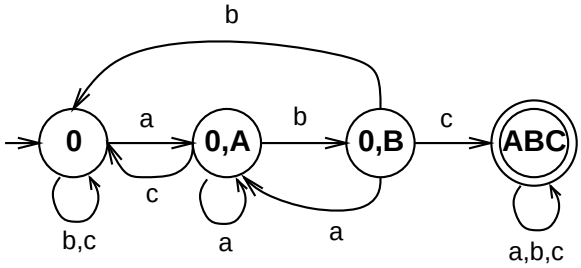
(a) Um den nichtdeterministischen endlichen Automaten deterministisch zu machen, kann man die sog. „Myhill-Konstruktion“ durchführen. Dabei geht man so vor, daß ausgehend vom Startzustand die Zustandsmengen errechnet werden, die mit den Eingabezeichen erreichbar sind. Dieses Verfahren wird für jede neu entstehende Zustandsmenge ausgeführt, bis keine neue mehr entsteht. Diese Zustandsmengen bilden dann die Zustände des deterministischen endlichen Automaten. Jede Zustandsmenge, die einen Endzustand (hier Zustand *C*) enthält, wird zum Endzustand im deterministischen Automaten.

	a	b	c
0	**0,A**	0	0
0,A	0,A	**0,B**	0
0,B	0,A	0	**0,C**
0,C	**0,A,C**	0,C	0,C
0,A,C	0,A,C	**0,B,C**	0,C
0,B,C	0,A,C	0,C	0,C

Also ergibt sich folgende graphische Darstellung des endlichen Automaten:



(b) Wie man jedoch sehen kann, können nach Erreichen des Zustands **0,C** nur noch Endzustände erreicht werden (jeder Endzustand hat zudem Übergänge für sämtliche Zeichen des Eingabealphabets, d.h. der Fangzustand wird ebenfalls nicht von dort aus erreicht). Deshalb kann man die 3 Endzustände **0,C**, **0,A,C** und **0,B,C** zu einem Endzustand **ABC** zusammenfassen, der erreicht wird, nachdem *abc* gelesen wurde und dann nicht mehr verlassen wird. Durch diese „Minimierung“ ergibt sich folgender Automat:



Aufgabe 13 (P) Reguläre Grammatik, nichtdeterministischer endlicher Automat

Die vorgestellte Lösung ist verhältnismäßig umfangreich, jedoch wird ein Großteil der benutzten Klassen als Vorlage im WWW zur Verfügung gestellt. Im folgenden sind daher nur die eigens zu implementierenden Teile der Klasse `Automaton` aufgeführt. In einer folgenden Aufgabe wird diese allgemeine Repräsentation eines endlichen Automaten für dessen Minimierung genutzt.

```
import java.util.*;
import java.io.*;

/**
 * This class represents a finite automaton in both
 * non-determinstic and determinstic variants.
 *
 */
public class Automaton {
    // Tabular representation of automaton transition matrix
    private Set[][] automaton;
    private int numStates, numLabels;
    // Automaton consists of states and labels, transitions are managed by states
    private Set states, labels;
    private State currentState;
    // Utility data structure to map unique names to corresponding instance
    private Map nameToState, nameToLabel;
    // Utility data structure to map integer id to states and labels
    private List stateList, labelList;

    // Teilaufgabe a) ------------------------------------------------------

    public Automaton() {
        numStates = 0; numLabels = 0;
        states = new HashSet(); labels = new HashSet();
        nameToState = new HashMap(); nameToLabel = new HashMap();
        stateList = new Vector(); labelList = new Vector();
    }

    // Add a new state to the automaton, possibly as start or terminal state
    public void addState(String name, boolean isStart, boolean isTerminal) {
        State s = (State) nameToState.get(name);
        if (s == null) {
            s = new State(name, numStates);
            s.setStart(isStart); s.setTerminal(isTerminal);
            states.add(s);
            stateList.add(s);
            nameToState.put(name, s);
            numStates++;
        }
    }
```

```
// Add a new transition to the automaton between possibly existing states
public void addTransition(String from, String to, String label) {
    State origin = (State) nameToState.get(from);
    State target = (State) nameToState.get(to);
    if (origin == null) {
        addState(from, false, false);
        origin = (State) nameToState.get(from);
    }
    if (target == null) {
        addState(to, false, false);
        target = (State) nameToState.get(to);
    }
    // Create label if necessary
    Label l = (Label) nameToLabel.get(label);
    if (l == null) {
        l = new Label(label, numLabels);
        labels.add(l);
        labelList.add(l);
        nameToLabel.put(label, l);
        numLabels++;
    }
    // Create transition itself
    Transition t = new Transition(origin, target, l);
    origin.getOutgoingTransitions().add(t);
    target.getIncomingTransitions().add(t);
}

// Initialize the automaton, called before execution
public void init() {
    initTransitionMatrix();
    // Choose an initial state
    Iterator sit = states.iterator();
    while (sit.hasNext()) {
        State s = (State) sit.next();
        if (s.isStart()) {
            currentState = s;
            break;
        }
    }
    if (currentState == null)
        System.out.println("*** No initial state!");
}

/**
 *  Execute a single step of the automaton by providing a language symbol,
 *  not changing its internal state (call setCurrentState() later to do so)
 *  Returns a set of states which is either
 *    null, if no valid symbol has been provided,
 *    empty, if no valid transition exists (i.e. the word is not accepted),
 *    a single state, if the automaton is deterministic for given symbol,
 *    a number of states, if the automaton is nondeterministic for given symbol.
```

```
 * */
public Set step(String symbol) {
    Label label = (Label) nameToLabel.get(symbol);
    if (label == null) {
        System.out.println("*** Token \"" + symbol + "\" not recognized!");
        return null;
    }
    Set result = automaton[currentState.getId()][label.getId()];
    if (result.isEmpty()) {
        System.out.println("\n*** Token \"" + symbol + "\" not valid in state \""
        + currentState.getName() +"\"!");
        System.out.print("*** Expected symbol(s): ");
        for (int j = 0; j < numLabels; j++) {
            if (automaton[currentState.getId()][j].size() > 0)
                System.out.print(labelList.get(j) + " ");
        }
        System.out.print("\n");
        return result;
    }
    return result;
}

// Teilaufgabe b) ------------------------------------------------------

// Construct automaton from given regular grammar with productions of form
// <nonterminal> -> <epsilon>|terminal|terminal <nonterminal>
public void readGrammar(String fileName) {
    // Initialize private fields
    numStates = 0; numLabels = 0; currentState = null;
    nameToState = new HashMap(); nameToLabel = new HashMap();
    states = new HashSet(); labels = new HashSet();

    // Iterate all productions in given grammar (no error checking)
    String buffer = new String();
    String lhs = new String(), rhs1 = new String(), rhs2 = new String();
    try {
        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        while (reader.ready()) {
            // Separate productions into left- and right-hand side parts
            buffer = reader.readLine();
            StringTokenizer tokenizer = new StringTokenizer(buffer);
            lhs = tokenizer.nextToken();
            tokenizer.nextToken();
            rhs1 = tokenizer.nextToken();
            if (tokenizer.hasMoreTokens())
                rhs2 = tokenizer.nextToken();
            else
                rhs2 = "";

            // Process originating state i.e. nonterminal on left-hand side
            addState(lhs, isStart(lhs), false);
```

```
                if (isEpsilon(rhs1)) {
                    // Production: '<nonterminal> -> 'epsilon>'
                    State origin = (State) nameToState.get(lhs);
                    origin.setTerminal(true);
                } else {
                    // Process target state if necessary
                    if (rhs2.length() > 0) {
                        // Production: '<nonterminal> -> terminal <nonterminal>'
                        addState(rhs2, false, false);
                        addTransition(lhs, rhs2, rhs1);
                    } else {
                        // Production: '<nonterminal> -> terminal'
                        // Add dedicated final state
                        addState("<F>", false, true);
                        addTransition(lhs, "<F>", rhs1);
                    }
                }
            }
            reader.close();
        }
        catch (IOException ex) {
            ex.printStackTrace();
            System.exit(1);
        }
    }

    // Initialize transition matrix after change to automaton structure
    private void initTransitionMatrix() {
        automaton = new Set [numStates][numLabels];
        for (int i = 0; i < numStates; i++) {
            for (int j = 0; j < numLabels; j++) {
                automaton[i][j] = new HashSet();
            }
        }
        Iterator sit = states.iterator();
        while (sit.hasNext()) {
            State s = (State) sit.next();
            Iterator tit = s.getOutgoingTransitions().iterator();
            while (tit.hasNext()) {
                Transition t = (Transition) tit.next();
                automaton[s.getId()][t.getLabel().getId()].add(t.getTarget());
            }
        }
    }
}
```