Kapitel 7

Lexikalische Analyse

In diesem Kapitel behandeln wir zuerst die Aufgabe der lexikalischen Analyse, dann die formale Spezifikation dieser Aufgabe, schließlich ein Generierungsverfahren für lexikalische Analysatoren (Scanner) und geben am Ende einige Implementierungshinweise. Unter dem letzteren Punkt handeln wir auch den Sieber ab, der allein nur ein sehr kleines Kapitel füllen würde. Es sei schon darauf hingewiesen, daß ein weiteres für die Praxis relevanteres Generierungsverfahren im Kapitel 8, Syntaktische Analyse, beschrieben wird, und zwar im Anschluß an die Generierungsverfahren für LR(k)-Analysatoren.

7.1 Die Aufgabe der lexikalischen Analyse

Die lexikalische Analyse, realisiert in dem **Scanner**-Modul, soll das als Folge von Zeichen von einer Datei eingelesene Quellprogramm in eine Folge von lexikalischen Einheiten, **Symbole** genannt, zerlegen. Der Scanner liest diese Zeichenfolge von links nach rechts. Bei verschränkter Arbeitsweise von Scanner, Sieber und Parser ruft der Parser die Kombination Scanner-Sieber auf, um das nächste Symbol zu erhalten. Der Scanner beginnt die Analyse mit dem Zeichen, welches auf das Ende des zuletzt gefundenen Symbols folgt, und sucht den längsten Präfix der restlichen Eingabe, der ein Symbol der Sprache ist. Eine Darstellung dieses Symbols gibt er an den Sieber zurück, der feststellt, ob dieses Symbol für den Parser relevant ist oder ignoriert werden soll. Ist es nicht relevant, so stößt der Sieber den Scanner erneut an. Andernfalls gibt er eine eventuell veränderte Darstellung des Symbols an den Parser zurück.

Der Scanner muß i.a. in der Lage sein, unendlich viele oder zumindest sehr viele verschiedene Symbole zu erkennen. Zweckmäßigerweise teilt er diese Menge in endlich viele Klassen ein. Symbole verwandter Struktur bzw. gleicher syntaktischer Funktion fallen dabei in eine **Symbolklasse**. Damit unterscheiden wir jetzt

- **Symbole**, das sind Worte über einem Alphabet von Zeichen, Σ, etwa xyz12, 125, begin, "abc",
- **Symbolklassen**, das sind Mengen von Symbolen, etwa die Menge der Identifier, die Menge der integer-Konstanten und die der Zeichenketten, bezeichnet durch die Namen id, intconst, string, und

- Die unterstrichenen Zeichen $\underline{(}$, $\underline{)}$, $\underline{|}$, $\underline{*}$, $\underline{\emptyset}$ und $\underline{\varepsilon}$ sind Zeichen des Beschreibungsmechanismus „reguläre Ausdrücke“ und nicht der durch die regulären Ausdrücke beschriebenen regulären Sprachen. Deshalb werden sie **Metazeichen** genannt. Jedes System, welches Beschreibungen von regulären Sprachen durch reguläre Ausdrücke akzeptiert, hat folgendes Problem zu lösen: Wegen des beschränkten Vorrats an darstellbaren Zeichen fallen Metazeichen mit Zeichen aus Σ zusammen. Zudem muß für die nicht darstellbaren Zeichen $\emptyset$ und ε eine Darstellung gefunden werden. Wir werden auf dieses Problem im praktischen Teil zurückkommen. In den Beispielen gibt es keine Verwechselungsmöglichkeiten zwischen Metazeichen und Zeichen aus Σ. Deshalb verzichten wir auf die Unterstreichung der Metazeichen. Außerdem betrachten wir im weiteren die leere Menge und das sie darstellende Symbol $\emptyset$ nicht mehr, da es für sie keinen sinnvollen Gebrauch in unserem Kontext gibt.

Beispiel 7.2.1

reg. Ausdruck	beschriebene reg. Sprache	Elemente aus der reg. Sprache
$a\|b$	$\{a, b\}$	a, b
ab^*a	$\{a\}\{b\}^*\{a\}$	$aa, aba, abba, abbba, \ldots$
$(ab)^*$	$\{ab\}^*$	$\varepsilon, ab, abab, \ldots$
$abba$	$\{abba\}$	$abba$

Akzeptoren, d.h. mathematische Maschinen zum Erkennen regulärer Sprachen, sind endliche Automaten. Sie sind charakterisiert durch eine sehr eingeschränkte Merkfähigkeit, realisiert durch eine Variable, die nur endlich viele verschiedene Zustände annehmen kann. Wie Abbildung 7.1 zeigt, verfügt ein endlicher Automat außerdem über einen Lesekopf, mit dem er das Eingabeband von links nach rechts überstreichen kann. Die Übergangsrelation Δ bildet die Kontrolle des Automaten.

Definition 7.2.3 (nichtdeterministischer endlicher Automat)
Ein **(nichtdeterministischer) endlicher Automat (NEA)** ist ein Tupel $M = (\Sigma, Q, \Delta, q_0, F)$, wobei

- Σ ein endliches Alphabet, das **Eingabealphabet**, ist,
- Q eine endliche Menge von **Zuständen** ist,
- $q_0 \in Q$ der **Anfangszustand** ist,
- $F \subseteq Q$ die Menge der **Endzustände** ist, und
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ die **Übergangsrelation** ist. □

Wir erläutern jetzt die Arbeitsweise eines NEA und eines als Scanner eingesetzten NEA. Ein NEA soll Eingabeworte daraufhin prüfen, ob sie in einer gegebenen Sprache sind oder nicht. Er akzeptiert ein Wort, wenn er nach Lesen

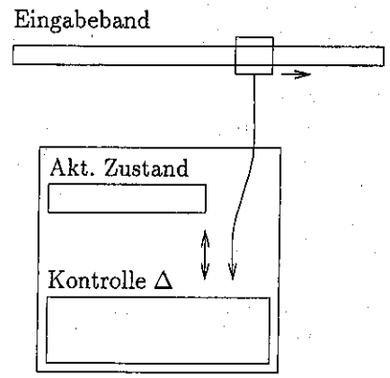


Abb. 7.1: Endlicher Automat

des ganzen Wortes in einem Endzustand gelandet ist. Ein als Scanner eingesetzter endlicher Automat zerlegt ein Eingabewort stückweise in Teilworte der gegebenen Sprache. Jedes Teilwort bringt ihn also von seinem Anfangszustand in einen Endzustand. Er hat dabei eventuell Probleme, das Ende des Teilwortes festzustellen.

Der endliche Automat wird in seinem Anfangszustand gestartet. Sein Lesekopf steht dabei am Anfang des Eingabebandes.

Bei Einsatz eines endlichen Automaten als Scanner steht er auf dem ersten noch nicht „konsumierten“ Zeichen.

Dann macht er eine Folge von Schritten. Ein Schritt hängt jeweils vom aktuellen Zustand und eventuell vom nächsten Eingabezeichen ab. Er besteht im Einnehmen eines neuen Zustands und, falls das Eingabezeichen gelesen wurde, im Bewegen des Lesekopfes auf das nächste Zeichen. Der Automat akzeptiert das Eingabewort, wenn die Eingabe erschöpft ist und der aktuelle Zustand ein Endzustand ist.

Der Scanner meldet das Finden eines Symbols, wenn er in einem Endzustand ist und unter dem nächsten Eingabezeichen keinen Übergang hat. Hat er aus dem aktuellen Zustand keinen Übergang, ohne daß dieser Zustand ein Endzustand ist, so muß er die letzten Zustandsübergänge rückgängig machen bis zu dem letzten durchlaufenen Endzustand. Gibt es für das aktuelle Symbol noch keinen solchen, so liegt ein Fehler vor.

Das zukünftige Verhalten eines NEA wird jeweils bestimmt vom aktuellen Zustand und der restlichen Eingabe. Diese beiden bilden zusammen die aktuelle Konfiguration des Automaten.

Definition 7.2.4 (Konfiguration, Schritt, akzeptierte Sprache)
Sei $M = (\Sigma, Q, \Delta, q_0, F)$ ein NEA. Ein Paar (q, w) mit $q \in Q$ und $w \in \Sigma^*$ heißt eine **Konfiguration** von M, (q_0, w) eine **Anfangskonfiguration** und (q_f, ε) mit $q_f \in F$ eine **Endkonfiguration.**

Die **Schritt**-Relation ist eine binäre Relation $\vdash_M$ auf $(Q \times \Sigma^*) \times (Q \times \Sigma^*)$. Es gilt $(q, aw) \vdash_M (p, w)$ genau dann, wenn $(q, a, p) \in \Delta$ für $q, p \in Q$ und $a \in \Sigma \cup \{\varepsilon\}$. $\vdash_M^*$ bezeichnet die reflexive, transitive Hülle dieser Relation $\vdash_M$. Die vom NEA M **akzeptierte Sprache** ist
$L(M) = \{w \in \Sigma^* | (q_0, w) \vdash_M^* (q_f, \varepsilon) \text{ mit } q_f \in F\}$. □

Der endliche Automat M akzeptiert also solche Worte, unter denen er einen Weg von einer Anfangs- in eine Endkonfiguration hat.

Der Scanner ist immer dann in einer Endkonfiguration, wenn er in einem Endzustand ist. Entsprechend akzeptiert er all die Worte als Symbole, die ihn von seinem Anfangszustand in einen Endzustand bringen.

Beispiel 7.2.2 In Tabelle 7.1 ist die Übergangsrelation eines NEA M_0 in Form einer zweidimensionalen Matrix T_{M_0} dargestellt. Die Zeilen entsprechen den durch ganze Zahlen dargestellten Zuständen, die Spalten den Elementen aus $\Sigma \cup \{\varepsilon\}$. Das Element $T_{M_0}[q, a]$ enthält die Menge der Zustände p, für die gilt: $(q, a, p) \in \Delta$. Der Automat erkennt Integer- und Realkonstanten, letztere mit optionalem zweistelligem Exponenten. □

Tabelle 7.1: NEA zum Erkennen von Integer- und Realkonstanten. Das Alphabet ist $\{0, \ldots, 9, ., E\}$. Die erste Spalte stellt 10 identische Spalten jeweils unter einer der Ziffern 0,1,..., 9 dar.

	0,1,..., 9	.	E	ε
0	{1,2}	∅	∅	∅
1	{1}	∅	∅	∅
2	{2}	{3}	∅	∅
3	{4}	∅	∅	∅
4	{4}	∅	{5}	{7}
5	{6}	∅	∅	∅
6	{7}	∅	∅	∅
7	∅	∅	∅	∅

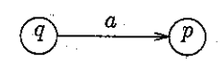
$q_0 = 0 \qquad F = \{1, 7\}$

Ein zu den nichtdeterministischen endlichen Automaten äquivalenter Mechanismus, der häufig zu ihrer Darstellung verwendet wird, ist der des (endlichen) Übergangsdiagramms.

Definition 7.2.5 (Übergangsdiagramm)
Ein **Übergangsdiagramm** ist ein endlicher, gerichteter, kantenmarkierter Graph $UD = (V, E, T, v_0, V_f)$, wobei V eine endliche Menge von Knoten, E eine endliche

Menge von markierten Kanten, $v_0 \in V$ ein ausgezeichneter Knoten, der **Startknoten**, und $V_f \subseteq V$ die Menge der Endknoten (graphisch doppelt umrandet dargestellt) sind. Die Markierung der Kanten stammt dabei aus der Menge T. Für $w \in T^*$ ist ein w-**Weg** in UD ein Weg von einem Knoten q zu einem Knoten p, so daß die Konkatenation der Kantenmarkierungen w ist. Die **von dem Übergangsdiagramm akzeptierte Sprache** ist
$L(UD) = \{w \in T^* |$ es gibt einen w-Weg von v_0 zu einem $v_f \in V_f$ in $UD\}$ □

Die Korrespondenz zwischen den beiden Mechanismen ist offensichtlich. Wenn man von einem NEA M ausgeht, nimmt man als Knoten des Übergangsdiagramms von M UD_M die Zustände, als Anfangsknoten den Anfangszustand und als Endknoten die Endzustände. Eine Kante



zieht man, wenn $(q, a, p) \in \Delta$ ist. Umgekehrt läßt sich aus einem Übergangsdiagramm UD eindeutig ein zu UD gehörender NEA M_{UD} gewinnen.

Beispiel 7.2.3
Das zum NEA von Beispiel 7.2.2 gehörende Übergangsdiagramm ist in Abbildung 7.2 dargestellt. □

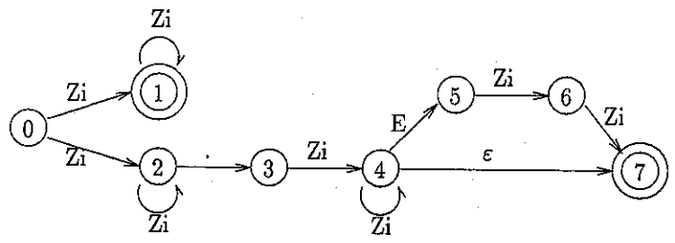
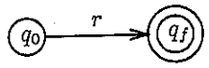


Abb. 7.2: Übergangsdiagramm zum NEA aus Beispiel 7.2.2, Zi steht für die Menge $\{0, 1, \ldots, 9\}$. Eine mit Zi markierte Kante ersetzt 10 mit 0, 1, ... bzw. 9 markierte Kanten mit gleichem Eingangs- und Ausgangsknoten.

Satz 7.2.1
Zu jedem regulären Ausdruck r gibt es einen nichtdeterministischen endlichen Automaten, der die von r beschriebene reguläre Menge akzeptiert. □

Der Beweis ist konstruktiv und beschreibt einen Teil des Scannergenerierungsverfahrens. Der folgende Algorithmus **RA→NEA** beschreibt die Konstruktion des Übergangsdiagramms eines NEA aus einem regulären Ausdruck r über einem Alphabet Σ. Er startet mit dem Anfangszustand und dem Endzustand und mit einer Kante dazwischen, die mit r markiert ist. Dann wird r gemäß seiner syntaktischen Struktur zerlegt, wobei das aktuelle Übergangsdiagramm verfeinert wird. Diese Zerlegung und Verfeinerung wird fortgesetzt, bis nur noch Kanten übrig sind, die mit Zeichen aus Σ oder ε markiert sind.

Algorithmus RA → NEA
Eingabe: regulärer Ausdruck r über Σ.
Ausgabe: Übergangsdiagramm eines NEA.
Methode: Start:



Wende die Regeln aus Abbildung 7.3 solange auf das jeweils aktuelle Übergangsdiagramm an, bis alle Kanten mit Zeichen aus Σ oder ε markiert sind. Die Knoten in den linken Seiten der Regeln werden identifiziert mit Knoten im aktuellen Übergangsdiagramm. Alle in der rechten Seite einer Regel neu auftretenden Knoten entsprechen neu kreierten Knoten, also neuen Zuständen.

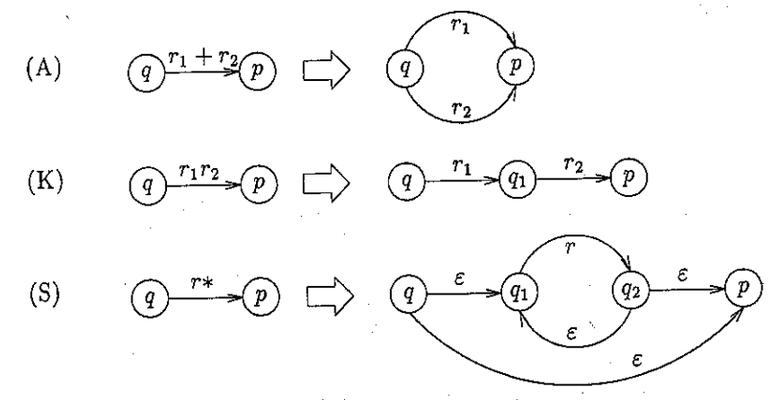


Abb. 7.3: Regeln des Algorithmus **RA → NEA**

Beispiel 7.2.4
Sei $\Sigma = \{a, 0\}$. Der reguläre Ausdruck $a(a|0)^*$ beschreibt die Menge der Worte über $\{a, 0\}$, die mit einem a beginnen. Die Konstruktion des NEA, der diese Sprache akzeptiert, läuft wie in Abbildung 7.4 beschrieben. □

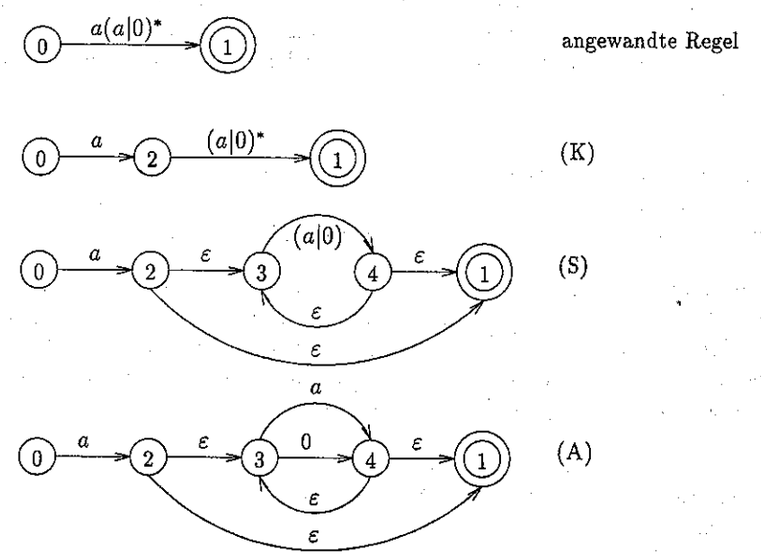


Abb. 7.4: Konstruktion eines NEA aus dem regulären Ausdruck $a(a|0)^*$

Definition 7.2.6 (deterministischer endlicher Automat, DEA)
Sei $M = (\Sigma, Q, \Delta, q_0, F)$ ein NEA. M heißt **deterministischer endlicher Automat** (DEA), wenn Δ eine partielle Funktion $\delta : Q \times \Sigma \rightsquigarrow Q$ ist. □

Ein DEA kennt also keine Übergänge unter ε und für jedes Paar (q, a) mit $q \in Q$ und $a \in \Sigma$ höchstens einen Nachfolgezustand. Dadurch gibt es auch für jedes Wort $w \in \Sigma^*$ höchstens einen w-Weg im Übergangsdiagramm von M. Wenn w im Sprachschatz von M ist, so führt dieser Weg vom Anfangszustand in einen Endzustand, und zwar ohne daß M die Ratefähigkeit eines nichtdeterministischen Automaten ausnutzen müßte. Deshalb werden für den praktischen Einsatz DEA bevorzugt. Glücklicherweise gilt Satz 7.2.2.

Satz 7.2.2
Wird eine Sprache L von einem NEA akzeptiert, so gibt es einen DEA, der L akzeptiert. □

Beweis:
Der Beweis ist konstruktiv und beschreibt einen weiteren Teil des Generierungsverfahrens für Scanner. Er benutzt die sogenannte **Teilmengenkonstruktion**. Die Zustände des konstruierten DEA sind Teilmengen der Zustandsmenge des Ausgangs-NEA, und zwar fallen dabei zwei NEA-Zustände p und q in dieselbe

Teilmenge, wenn es ein Wort w gibt, welches den NEA aus seinem Anfangszustand sowohl nach p als auch nach q bringt. Diese Konstruktion ist im Algorithmus **NEA → DEA** in Abbildung 7.5 beschrieben. Sie benutzt die folgenden beiden Begriffe.

Algorithmus NEA → DEA
Eingabe: NEA $M = (\Sigma, Q, \Delta, q_0, F)$
Ausgabe: DEA $M' = (\Sigma, Q', \delta, q_0', F')$ gemäß Definition 7.2.8

Methode: Zustände in M' sind Mengen von Zuständen von M. Startzustand von M' ist ε-$FZ(q_0)$. Die zu Q' hinzuerzeugten Zustände werden markiert, sobald ihre Folgezustände bzw. Übergänge unter allen Symbolen aus Σ erzeugt worden sind. Die Markierung von erzeugten Zuständen wird in der partiellen Funktion *marked*: $\mathcal{P}(Q) \rightsquigarrow$ **bool** festgehalten.

$q_0' := \varepsilon\text{-}FZ(q_0)$; $Q' := \{q_0'\}$; $marked(q_0') :=$ **false**; $\delta := \emptyset$;
while existiert $S \in Q'$ **and** $marked(S) =$ **false do**
  $marked(S) :=$ **true**;
  foreach $a \in \Sigma$ **do**
    $T := \varepsilon\text{-}FZ(\{p \in Q | (q, a, p) \in \Delta \text{ und } q \in S\})$;
    if $T \notin Q'$
    then $Q' := Q' \cup \{T\}$; (* neuer Zustand *)
      $marked(T) :=$ **false**
    fi;
    $\delta := \delta \cup \{(S, a) \mapsto T\}$ (* neuer Übergang *)
  od
od;

Abb. 7.5: Algorithmus NEA→DEA:

Definition 7.2.7 (ε-Folgezustände)
Sei $M = (\Sigma, Q, \Delta, q_0, F)$ ein NEA, und sei $q \in Q$.
Die Menge der **ε-Folgezustände** von q ist

$$\varepsilon\text{-}FZ(q) = \{p | (q, \varepsilon) \vdash_M^* (p, \varepsilon)\},$$

also die Menge aller Zustände p, inklusive q, für die es einen ε-Weg im Übergangsdiagramm zu M von q nach p gibt. Wir erweitern ε-FZ auf Mengen von Zuständen $S \subseteq Q$.

$$\varepsilon\text{-}FZ(S) = \bigcup_{q \in S} \varepsilon\text{-}FZ(q)$$

□

Der deterministische endliche Automat M' zu einem nichtdeterministischen endlichen Automaten M ist definiert durch:

Definition 7.2.8 (der zu einem NEA gehörende DEA)
Sei $M = (\Sigma, Q, \Delta, q_0, F)$ ein NEA. Der zu M **gehörende DEA**
$M' = (\Sigma, Q', \delta, q_0', F')$ ist definiert durch:
$Q' \subseteq \mathcal{P}(Q)$ (Potenzmenge von Q),
$q_0' = \varepsilon\text{-}FZ(q_0)$,
$F' = \{S \subseteq Q | S \cap F \neq \emptyset\}$ und
$\delta(S, a) = \varepsilon\text{-}FZ(\{p | (q, a, p) \in \Delta \text{ für } q \in S\})$ für $a \in \Sigma$. □

Den Nachfolgezustand zu einem Zustand S unter einem Zeichen a in M' erhält man also, indem man die Nachfolgezustände aller Zustände $q \in S$ unter a zusammenfaßt und ihre ε-Folgezustände hinzufügt.

Beispiel 7.2.5
Der Algorithmus **NEA → DEA**, angewendet auf den NEA von Beispiel 7.2.4 (siehe Abbildung 7.4) läuft in den in Abbildung 7.6 beschriebenen Schritten ab. Die Zustände des zu konstruierenden DEA sind „gestrichene" natürliche Zahlen $0', 1', \ldots$. Der Anfangszustand ist $0'$. Die markierten Zustände in Q' sind unterstrichen. Der Zustand $\emptyset$ ist der Fehlerzustand. Er ist der Nachfolgezustand eines Zustandes q unter a, wenn es keinen Übergang unter a aus q heraus gibt. □

Die in den beiden Schritten aus regulären Ausdrücken erzeugten deterministischen endlichen Automaten sind i.a. nicht die kleinstmöglichen, die die Ausgangssprache akzeptieren. Es kann nämlich noch Zustände mit gleichem „Akzeptanzverhalten" geben. Dieses trifft für Zustände p und q zu, wenn der Automat aus p und q unter allen Eingabewörtern entweder immer oder nie in einen Endzustand geht. Das in Abbildung 7.8 vorgestellte Verfahren produziert einen DEA mit minimaler Anzahl von Zuständen für die gleiche Sprache.

Beispiel 7.2.6

Minimierung des DEA aus Beispiel 7.2.5.

Partition	Klasse	Aufspaltung
$\{\{0', \emptyset\}, \{1', 2'\}\}$		
	$\{0', \emptyset\}$	$\{0'\}, \{\emptyset\}$
	$\{1', 2'\}$	nein
$\{\{0'\}, \{\emptyset\}, \{1', 2'\}\}$	keine weitere Aufspaltung	

$\{1', 2'\}$ bilden zusammen einen neuen Zustand
$\{\emptyset\}$ ist ein toter Zustand, da er nicht Endzustand ist,
und alle Übergänge aus ihm heraus wieder in ihn hineingehen.

Der resultierende minimale Automat ist in Abbildung 7.7 dargestellt. □

$0' = \{0\}; Q' = \{0'\}$

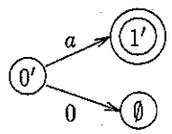
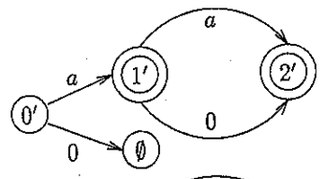
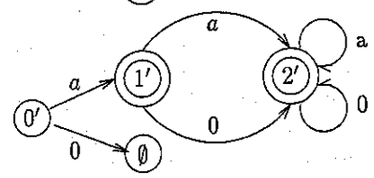
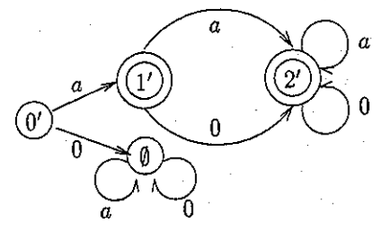
ausgewählter Zustand	neues Q'	neuer (Teil-) DEA
$0'$	$\{\underline{0'}, 1', \emptyset\}$ mit $1' = \{1,2,3\}$	
$1'$	$\{\underline{0'}, \underline{1'}, 2', \emptyset\}$ mit $2' = \{1,3,4\}$	
$2'$	$\{\underline{0'}, \underline{1'}, \underline{2'}, \emptyset\}$	
$\emptyset$	$\{\underline{0'}, \underline{1'}, \underline{2'}, \underline{\emptyset}\}$	

Abb. 7.6: Konstruktionsschritte zu Beispiel 7.2.5

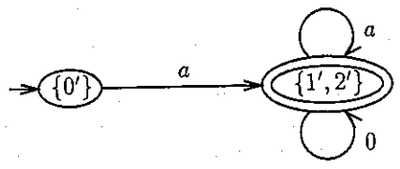


Abb. 7.7: Minimaler endlicher Automat aus Beispiel 7.2.6

Algorithmus MinDEA
Eingabe: DEA $M = (\Sigma, Q, \delta, q_0, F)$.
Ausgabe: DEA $M_{min} = (\Sigma, Q_{min}, \delta_{min}, q_{0,min}, F_{min})$ mit $L(M) = L(M_{min})$ und Q_{min} minimal.

Methode: Die Zustandsmenge von M wird in eine Partition aufgeteilt, die schrittweise verfeinert wird. Für Zustände in verschiedenen Klassen einer Partition ist schon bekannt, daß sie „verschiedenes Akzeptanzverhalten" zeigen, d.h. daß es mindestens ein Wort w gibt, unter welchem aus einem der Zustände ein Endzustand erreicht wird und aus dem anderen nicht. Deshalb beginnt man mit der Partition $\Pi = \{F, Q - F\}$. Der Algorithmus hält, wenn in einem Schritt die Partition nicht mehr verfeinert wurde. Da in jedem Iterationsschritt nur Klassen der aktuellen Partition evtl. in Vereinigungen neuer Klassen zerlegt werden, Q und damit $\mathcal{P}(Q)$ aber endlich sind, terminiert das Verfahren. Die Klassen der dann gefundenen Partition sind die Zustände von M_{min}. Es gibt einen Übergang zwischen zwei neuen Zuständen P und R unter einem Zeichen $a \in \Sigma$, wenn es einen Übergang $\delta(p, a) = r$ mit $p \in P$ und $r \in R$ in M gab. Es folgt das Programm.

$\Pi := \{F, Q - F\}$;
do changed := **false**;
  $\Pi' := \Pi$;
  foreach $K \in \Pi$ **do**
    $\Pi' := (\Pi' - \{K\}) \cup \{\{K_i\}_{1 \le i \le n}\}$, wobei die K_i maximal sind mit $K = \bigcup_{1 \le i \le n} K_i$; und $\forall a \in \Sigma : \exists K_i' \in \Pi : \forall q \in K_i : \delta(q, a) \in K_i'$
    if $n > 1$ **then** changed := **true** **fi** (* K wurde aufgespalten *)
  od;
$\Pi := \Pi'$
until **not** changed ;
$Q_{min} = \Pi -$ (Tot $\cup$ Unerreichbar);

$q_{0,min}$	die Klasse in Π, in der q_0 ist.
F_{min}	die Klassen, die ein Element aus F enthalten.
$\delta_{min}(K, a) = K'$,	wenn $\delta(q, a) = p$ mit $a \in \Sigma$ und $p \in K'$ für ein (und damit für alle) $q \in K$.
$K \in$ Tot,	wenn K kein Endzustand ist und nur Übergänge in sich selbst enthält.
$K \in$ Unerreichbar,	wenn es keinen Weg vom Anfangszustand nach K gibt.

Abb. 7.8: Algorithmus MinDEA

7.3 Eine Sprache zur Spezifikation der lexikalischen Analyse

Mit den regulären Ausdrücken steht der prinzipielle Beschreibungsformalismus für die lexikalische Analyse zur Verfügung. Allerdings ist er allein zu unhandlich für praktische Zwecke.

Beispiel 7.3.1
Ein regulärer Ausdruck, der die in den Beispielen 7.2.2 und 7.2.3 durch NEA bzw. Übergangsdiagramme akzeptierte Sprache der Integer- bzw. Realkonstanten beschreibt, ist
(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
(ε|.(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
(ε|E(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)))
Beachten Sie daß sich die Konkatenation der drei Teilausdrücke über die zwei Zeilenenden erstreckt. □

In den folgenden Abschnitten werden schrittweise Erweiterungen des Beschreibungsformalismus vorgenommen, die den Komfort erhöhen, aber die Mächtigkeit, d.h. die beschreibbare Sprachklasse nicht erweitern.

7.3.1 Zeichenklassen

Eine Spezifikation der lexikalischen Analyse sollte es erlauben, Mengen von Zeichen zu Klassen zusammenzufassen, wenn diese in Symbolen ausgetauscht werden können, ohne daß dadurch die entstehenden Symbole in verschiedene Symbolklassen eingeordnet würden.

Beispiel 7.3.2
bu = a - z A - Z
zi = 0 - 9
or = |
open = (
close =)
star = *
□

Die ersten beiden Zeichenklassendefinitionen erfüllen den oben geschilderten Zweck. Sie definieren Mengen von Zeichen durch Angabe von Intervallen im zugrundeliegenden Zeichencode, z.b. ASCII. Jetzt ließe sich z.B. die übliche Definition der Symbolklasse der Bezeichner angeben:

id = *bu* (*bu* | *zi*)*

Die weiteren vier Zeichenklassendefinitionen lösen (teilweise) die Metazeichenproblematik. Wenn in der Scannerspezifikation die Zeichenklassendefinitionen syntaktisch von der Definition der Symbole getrennt sind, so können die Zeichen, die gleichzeitig Metazeichen und benutzbare Zeichen sein sollen, mit Zeichenklassennamen versehen werden. In den Symboldefinitionen werden dann

für die benutzbaren Zeichen diese Namen verwendet, um sie von den Metazeichen zu unterscheiden. Wir kommen in den Zeichenklassendefinitionen mit nur drei Metazeichen aus, nämlich '=', '−' und dem Leerzeichen.

Beispiel 7.3.3
Der reguläre Ausdruck für die Integer- und Realkonstanten vereinfacht sich durch die Zeichenklassendefinition $zi = 0 - 9$ zu:
$zi\ zi^*(\varepsilon\ |.zi\ zi^*(\varepsilon\ |Ezi\ zi))$ □

7.3.2 Folgen von regulären Definitionen

Symbole einer Programmiersprache können Symbole anderer Symbolklassen als Teilworte enthalten. Deshalb ist es wünschenswert, regulären Ausdrücken, die Symbolklassen definieren, Namen geben zu können, damit diese Namen in anderen regulären Ausdrücken zur Abkürzung verwendet werden können.

Definition 7.3.1 (Sequenz von regulären Ausdrücken)
Eine **Sequenz von regulären Ausdrücken** über Σ ist eine Folge von Definitionen

$$\begin{aligned} A_1 &= R_1 \\ A_2 &= R_2 \\ &\vdots \\ A_n &= R_n \end{aligned}$$

wobei die $A_1, \ldots, A_n$ paarweise verschiedene Bezeichner und die R_i reguläre Ausdrücke über $\Sigma \cup \{A_1, \ldots A_{i-1}\}$ sind $(1 \leq i \leq n)$. □

Der zu R_i korrespondierende reguläre Ausdruck über Σ ist

$$R_i' = [R_1'/A_1][[R_2'/A_2][\ldots[R_{i-1}'/A_{i-1}]R_i]\ldots],$$

wobei $[R_j/A_j]R$ die Ersetzung aller Vorkommen von A_j durch R_j in R bedeutet.

Beispiel 7.3.4
intconst = *zi zi**
realconst = *intconst.intconst* (*Ezi zi*|ε)
Die Definition der Integerkonstanten kann in der Definition der Realkonstanten verwendet werden. □

Die Einschränkung, daß R_i nur Bezeichner aus der Menge $\{A_1, \ldots, A_{i-1}\}$ enthalten darf, ist wesentlich und bedarf einer Erläuterung. Die Grenze zwischen lexikalischer und syntaktischer Analyse, welche beide die syntaktische Struktur eines Programms analysieren, ist fließend und läßt sich aus pragmatischen Gründen verschieben. Wesentlich allerdings ist, daß die der lexikalischen Analyse zugeordnete Aufgabe durch einen endlichen Automaten erledigt werden kann, denn dieser läßt sich effizienter realisieren als die zur Syntaxanalyse verwendeten Kellerautomaten. Deshalb müssen die Symbolklassen der lexikalischen Analyse reguläre Sprachen sein. Wenn die obige Einschränkung für Sequenzen regulärer Definitionen fehlt, so ist auch eine Definition $A = (aAb|\varepsilon)$ möglich, welche die bekanntermaßen nicht reguläre Sprache $\{a^n b^n | n \geq 0\}$ beschreibt.

7.3.3 Nichtrekursive Klammerung

Programmiersprachen enthalten lexikalische Einheiten, welche durch die sie begrenzenden Klammern charakterisiert sind, z.B. Zeichenketten (strings) und Kommentare. Im Falle der Kommentare können die Klammern durchaus aus mehreren Zeichen zusammengesetzt sein: (* und *), /* und */, **cobegin** und **coend**, — — und NL (Zeilenwechsel). Zwischen den öffnenden und schließenden Klammern können nahezu beliebige Worte stehen. Dies ist nicht sehr einfach zu beschreiben. Eine abkürzende Schreibweise dafür ist:

$$R_1 \, until(R_2)$$

Dabei muß man verlangen, daß die von R_2 beschriebene reguläre Sprache das leere Wort nicht enthält. Die dadurch definierte reguläre Sprache ist

$$R_1 \, \overline{\Sigma^* R_2 \Sigma^*} \, R_2$$

7.4 Die Generierung eines Scanners

Die Erzeugung eines minimalen deterministischen endlichen Automaten aus einem regulären Ausdruck haben wir in Abschnitt 7.2 behandelt. Die dort beschriebenen Schritte bilden das Herzstück eines möglichen Scannergenerierungsverfahrens. Noch zu erläutern bleiben die Erweiterungen des Beschreibungsformalismus, die im letzten Abschnitt behandelt wurden.

7.4.1 Zeichenklassen

Zeichenklassen wurden eingeführt, um das Aufschreiben und die entstehenden Automaten zu verkleinern. Die Klassendefinitionen

bu = a - z
zi = 0 - 9

erlauben es, 26 Übergänge unter Buchstaben zwischen zwei Zuständen des Automaten für $id = bu(bu|zi)^*$ durch einen unter bu zu ersetzen.

Die Implementierung ist nicht schwer. Es wird ein Feld angelegt, dessen Komponenten mit den Maschinendarstellungen der Zeichen indiziert werden. Der Inhalt einer Feldkomponente ist der Code für die Zeichenklasse, zu der das Zeichen gehört. Für Zeichen, die nicht explizit in einer Zeichenklasse vorkommen und für solche, die in einer Symboldefinition explizit auftreten, wird implizit eine eigene Klasse definiert. Ein Problem tritt auf, wenn zwei Klassen A und B nicht disjunkt sind. Dann ersetzt der Generator diese durch die drei Klassen $A-B$, $B-A$ und $A \cap B$. Im Automaten gibt es dann überall, wo es einen Übergang unter A bzw. B gäbe, jeweils zwei Übergänge unter $A-B$ und $A \cap B$ bzw. $B-A$ und $A \cap B$, sofern die so entstandenen Klassen nicht leer sind. Dieser Prozeß wird solange durchgeführt, bis alle Klassen paarweise disjunkt sind.

Beispiel 7.4.1
Hätten wir die beiden Klassen

bu = a - z
$buzi$ = a - z 0 - 9

definiert, um dann die Symbolklasse
$id = bu \; buzi^*$ zu definieren, so müßte der Generator eine Aufspaltung in die Zeichenklassen

$zi' = buzi - bu$
$bu' = bu \cap buzi = bu$

vornehmen. Damit ergäbe sich die (offensichtlich sinnvolle) oben angegebene Klasseneinteilung. □

7.4.2 Folgen von regulären Definitionen

Gegeben sei eine Folge von regulären Definitionen.

$$\begin{aligned} A_1 &= R_1 \\ A_2 &= R_2 \\ &\vdots \\ A_n &= R_n \end{aligned}$$

Die Generierung eines Scanners zu dieser Folge erfolgt in den Schritten:

1. Erzeugung von NEAs $M_i = (\Sigma, Q_i, \Delta_i, q_{0,i}, F_i)$ für die (durch Einsetzung entstandenen) regulären Ausdrücke R_i', wobei die Q_i paarweise disjunkt seien.

2. Konstruktion des NEA $M = (\Sigma, Q, \Delta, q_0, F)$ mit
   $Q = \bigcup_{i=1}^{n} Q_i \cup \{q_0\}, \quad q_0 \notin \bigcup_{i=1}^{n} Q_i, \quad F = \bigcup_{i=1}^{n} F_i$
   $\Delta = \bigcup_{i=1}^{n} \Delta_i \cup \{(q_0, \varepsilon, q_{0,i}) \mid 1 \leq i \leq n\}$.
   Man erhält also den NEA M für die Sequenz, indem man den neuen Anfangszustand q_0 mit einem ε-Übergang zu allen Anfangszuständen $q_{0,i}$ versieht.

3. Anwendung des Algorithmus **NEA→DEA** (siehe Abb. 7.5) auf M, Ergebnis DEA M'.

4. Eventuell Minimierung von M'. Dabei startet man den Algorithmus **Min-DEA** mit einer Partition $\Pi = \{F_1', F_2', \ldots, F_n', Q' - \bigcup_{i=1}^{n} F_i'\}$, wobei die F_i' die zur Symbolklasse A_i gehörenden Endzustände von M' sind. Genauer: $F_i' = \{S \mid S \cap F_i \neq \emptyset \; und \; S \cap F_j = \emptyset (1 \leq j < i)\}$. [1]

[1] Dies heißt, daß ein Symbol, welches den Automaten M' in zwei frühere Endzustände des M_i führt, der in der Definitionsliste früheren Klasse zugeordnet wird.

Beispiel 7.4.2
Seien die Einzelzeichenklassen

zi = 0 - 9
hex = A - F gegeben.

Die Folge von regulären Definitionen

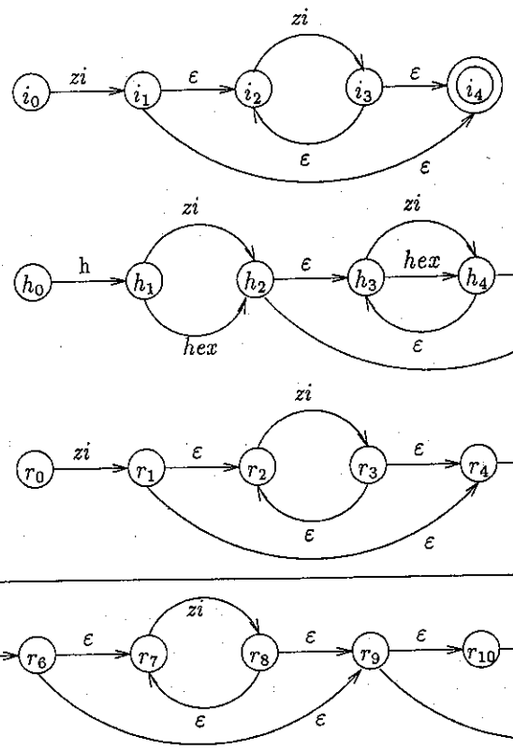
intconst = *zi zi**
hexconst = h (*zi*|*hex*) (*zi*|*hex*)*
realconst = *intconst* . *intconst* (e *zi zi*|ε)

wird in den folgenden Schritten bearbeitet:

- Ersetzung der beiden Vorkommen des Symbolklassennamens *intconst* durch seine Definition:

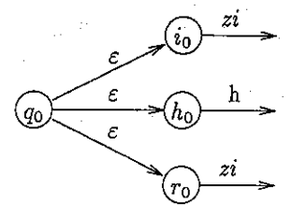
  intconst = *zi zi**
  hexconst = h (*zi*|*hex*) (*zi*|*hex*)*
  realconst = *zi zi** . *zi zi** (e *zi zi*|ε)

- Erzeugung von NEAs für die regulären Ausdrücke:

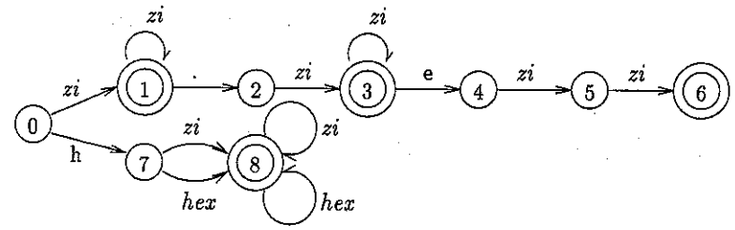


Beachten Sie, daß die drei Endzustände die folgende Interpretation haben: „*intconst* gefunden“ (i_4), „*hexconst* gefunden“ (h_5) bzw. „*realconst* gefunden“ (r_{12}).

- Kombinieren der drei NEAs mithilfe eines neuen Anfangszustandes q_0:



- Deterministisch machen dieses NEA.



- Minimierung des deterministischen endlichen Automaten

Nach der Konstruktion des DEA enthält der neue Endzustand 1 den alten Endzustand i_4 und hat damit u.a. die Interpretation „*intconst* gefunden“; die Endzustände 3 und 6 enthalten beide den alten Endzustand r_{12} und haben damit die Bedeutung „*realconst* gefunden“; der Endzustand 8 enthält h_5 und signalisiert damit „*hexconst* gefunden“. Denken sie daran, daß generierte Scanner immer die längsten Präfixe der restlichen Eingabe suchen, die in einen Endzustand führen. Der obige DEA wird also aus dem Endzustand 1 heraus einen Übergang machen, wenn dies möglich ist, d.h. wenn ein „.“ folgt. Folgt auf diesen Punkt allerdings keine Ziffer, so muß der DEA zum Endzustand 1 zurücksetzen und den Lesezeiger ein Feld zurückschieben. □

7.4.3 Implementierung des *until*-Konstrukts

Zur Erkennung eines Symbols, das durch R_1 *until*(R_2) beschrieben wird, muß der Scanner, nachdem er ein Exemplar für R_1 durch einen entsprechenden Automaten M_{R_1} erkannt hat, aus dessen Endzustand heraus ein Exemplar von R_2 finden und dann anhalten. Diese letzte Aufgabe ist eine Verallgemeinerung des Problems der Mustererkennung auf Zeichenketten (string pattern matching). Es gibt dazu Algorithmen, die für reguläre Muster die Mustererkennung in linearer Zeit in der Größe der zu durchmusternden Eingabe vornehmen. Diese werden z.B. in dem UNIX-Programm *egrep* verwendet. Sie konstruieren einen endlichen

Automaten für diese Aufgabe. Man könnte also die oben gestellte Aufgabe lösen, indem man einen solchen Mustererkennungsautomaten hinter den Automaten M_{R_1} hängt. Im folgenden soll ein solcher Automat auf ungewöhnliche Art und Weise konstruiert werden.

Der Ausgangspunkt ist die Definition der durch R_1 *until*(R_2) beschriebenen Sprache. Sie wird durch den regulären Ausdruck $R_1\ \overline{\Sigma^* R_2 \Sigma^*}\ R_2$ definiert. Wir benutzen nun die Konstruktionen, die aus den Beweisen der Abgeschlossenheitseigenschaften von regulären Sprachen bekannt sind. Die Automatenkonstruktion ist in den folgenden 7 Schritten beschrieben. In Abbildung 7.9 lassen sich diese Schritte an einem einfachen Beispiel verfolgen.

1. Konstruktion eines NEA M_{R_2} für R_2. Von diesem werden zwei Kopien in den Schritten 2 und 6 gebraucht.

2. Konstruktion eines NEA M_1 für $\Sigma^* R_2 \Sigma^*$. Hier benutzen wir eine Kopie von M_{R_2}.

   Dieser Automat akzeptiert (nichtdeterministisch) alle Worte über Σ, die mindestens ein Teilwort aus der von R_2 beschriebenen regulären Sprache enthalten.

3. Dieser NEA wird mithilfe von Algorithmus **NEA→DEA** in einen DEA M_2 umgewandelt. Die Zustände von M_2 haben also die Eigenschaft: Wenn es ein Wort $w \in \Sigma^*$ gibt und Zustände p und q in M_1 mit $(q_{0,1}, w) \vdash_{M_1}^* (p, \varepsilon)$ und $(q_{0,1}, w) \vdash_{M_1}^* (q, \varepsilon)$, dann liegen p und q in mindestens einem Zustand von M_2 gemeinsam. Besonders wichtig ist die Funktion des Zyklus $delta_1(q_{0,1}, a) = q_{0,1}$ für alle $a \in \Sigma$. Er ist dafür verantwortlich, daß $q_{0,1}$ in allen Zuständen von M_2 enthalten ist. Dies wiederum hat folgende Auswirkung. Nehmen wir an, daß ein Wort w aus der Sprache zu R_2 die Gestalt $xyxz$ hat, wobei $x \in \Sigma^+$ sei; ein Suffix x eines Präfixes xyx des Wortes w ist also auch Präfix von w. Wenn M_2 xyx gelesen hat und das nächste Eingabezeichen nicht der Anfang von z ist, so enthält der aktuelle Zustand von M_2 nicht nur die Hypothese, daß gerade der Präfix xyx von w gelesen wurde, sondern gleichzeitig die, daß der Präfix x von w gelesen wurde. Deshalb erlaubt er eine Fortsetzung unter dem Anfangszeichen von y. Wie kommt dies zustande? Gibt es in M_1 einen Weg unter x in den Zustand q_i, unter xy einen in q_k und unter x einen aus q_k in q_j, so wird jeweils jeder mit $q_{0,1}$ in einen neuen Zustand gepackt. Aus dem neuen Zustand, der $q_{0,1}$ und q_k enthält, gibt es deshalb einen x-Weg in einen Zustand, der u.a. q_i und q_j enthält. Diese Zusammenfassung von Zuständen ist wesentlich für die lineare Zeitkomplexität des *until*-Automaten verantwortlich.

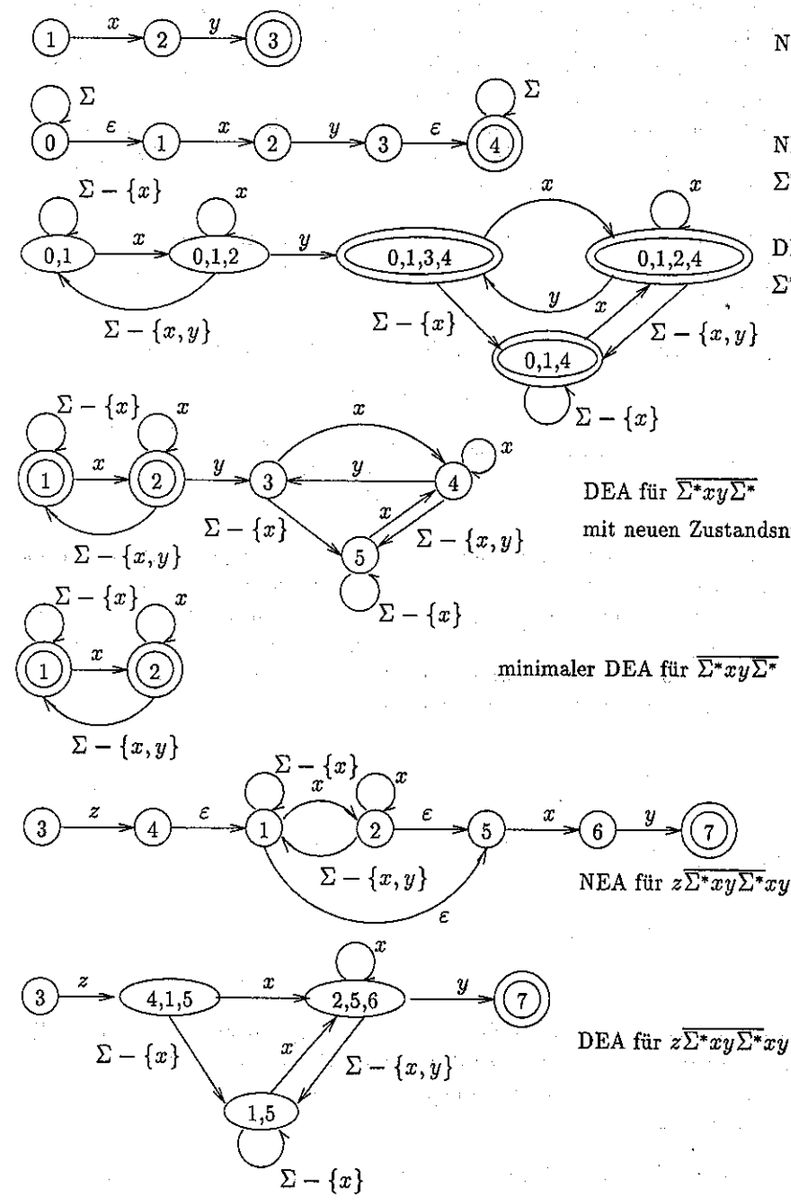
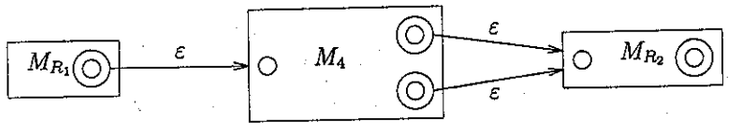


Abb. 7.9: Die Entwicklung des deterministischen endlichen Automaten für z *allbut*(xy) mit $x, y, z \in \Sigma$.

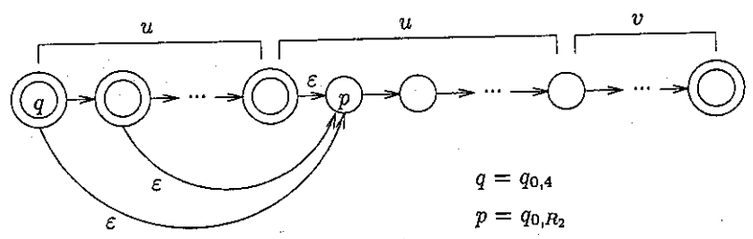
4. Zur Konstruktion des DEA M_3, der die Sprache zu $\overline{\Sigma^* R_2 \Sigma^*}$ akzeptiert, werden in M_2 die Mengen der Endzustände und der Nichtendzustände vertauscht. Jeder Zustand, der vorher Endzustand war, ist es jetzt nicht mehr, jeder, der vorher kein Endzustand war, ist es jetzt geworden. M_3 akzeptiert also z.B. alle echten Präfixe von Worten zu R_2, die nicht selbst Worte von R_2 sind. Insbesondere akzeptiert M_3 im Anfangszustand das leere Wort, da die Sprache zu R_2 dies nicht enthalten darf.

5. Dieser Automat wird in einen minimalen DEA M_4 umgewandelt. Dabei werden alle früheren Endzustände von M_2 in einer Klasse zusammengefaßt und anschließend als tot eliminiert.

6. Konstruktion eines NEA M_5 für die Sprache $R_1\ \overline{\Sigma^* R_2 \Sigma^*}\ R_2$.



Von jedem Endzustand von M_4, insbesondere vom Anfangszustand $q_{0,4}$, geht ein ε-Übergang zum Anfangszustand von M_{R_2}, das ist q_{0,R_2}. Von q_{0,R_2} führen Wege unter allen Worten w aus der Sprache von R_2 in den Endzustand von M_{R_2}. Von $q_{0,4}$ führen Wege unter echten Präfixen von Worten, die nicht in dieser Sprache sind, in Endzustände von M_4. Ist $w = uv \in R_2$ mit $u, v \in \Sigma^*$ und $v \neq \varepsilon$, so sieht das folgendermaßen aus:



7. M_5 wird in einen deterministischen Automaten M_6 umgewandelt. Durch die ε-Kante von $q_{0,4}$ nach q_{0,R_2} werden dabei alle vorhandenen w-Wege in M_4 und M_{R_2} zusammengefaßt. Solange es möglich ist, verfolgt man gewissermaßen die zwei Hypothesen

   H1: ein Suffix des gegenwärtig analysierten Symbols gehört zu R_2 und

   H2: kein Suffix des gegenwärtig analysierten Symbols gehört zu R_2

   parallel. M_{R_2} verfolgt dabei die Hypothese **H1**, M_4 die Hypothese **H2**. Kann der Suffix nicht zu einem Wort aus der Sprache von R_2 fortgesetzt werden, d.h. hat M_{R_2} unter dem nächsten Eingabezeichen keine Fortsetzung, ist aber

auch noch nicht im Endzustand, so muß die Hypothese **H1** für diesen Suffix verworfen werden. Dies wird u.a. durch die in M_5 vorhandene ε-Kante in den Anfangszustand von M_{R_2} repräsentiert. Diese besagt, daß die Hypothese **H1** mit leerem Suffix neu überprüft werden muß.

7.4.4 Die Darstellung eines Scanners

Für die Darstellung der Übergangsfunktion eines Scanners, der in unserem Fall ein deterministischer endlicher Automat ist, gibt es eine naheliegende Datenstruktur, nämlich ein zweidimensionales Feld *delta*, welches mit dem aktuellen Zustand und der Zeichenklasse des nächsten Eingabezeichens indiziert wird, um den neuen Zustand nachzuschlagen, in den der Automat übergeht. Dazu werden sowohl die Zustände als auch die Zeichenklassen als natürliche Zahlen codiert. Der Zugriff auf *delta*[*actstate, nextcl*] ist äußerst schnell. Ein Problem dagegen ist eventuell die Größe des Feldes *delta*, das Produkt aus Zahl der Zustände und Zahl der Einzelzeichenklassen.

Da zusätzlich dieses Feld im allgemeinen schwach besetzt ist, kann man eine Reihe von Verfahren zur Komprimierung schwach besetzter Matrizen anwenden. Diese sparen meist sehr viel an Platz auf Kosten geringfügig erhöhter Zugriffszeit. Da die leeren Einträge aber zu undefinierten Übergängen gehören, die für die Analyse und die Fehlererkennung signifikant sind, muß die in ihnen enthaltene Information weiterhin verfügbar sein.

Betrachten wir einen solchen Komprimierungsalgorithmus. Wir stellen die Übergangsfunktion etwas anders dar als durch das oben definierte Feld *delta*, und zwar durch ein Feld *RowPtr*, welches indiziert wird mit Zuständen und dessen Komponenten Adressen der Zeilen von *delta* sind, siehe Abbildung 7.10.

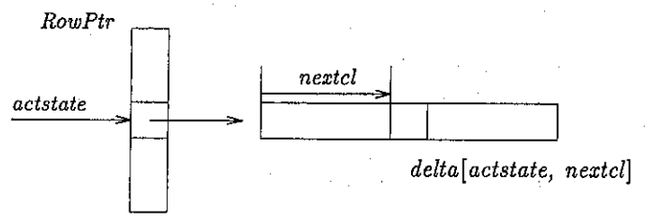


Abb. 7.10: Darstellung der Übergangsfunktion eines DEA

Noch haben wir nichts gewonnen, sondern nur beim Zugriff Geschwindigkeit eingebüßt. Die Zeilen, auf die in *RowPtr* verwiesen wird, sind vor allen Dingen fast leer. Deshalb werden die einzelnen Zeilen so in ein gemeinsames eindimensionales Feld *Delta* übereinander gelegt, daß nichtleere Einträge nicht miteinander kollidieren. Das kann man etwa für die einzelnen Zeilen nacheinander mit einer first-fit-Strategie machen. Man schiebt die nächste abzulegende Zeile so über das

Feld *Delta*, bis keine Kollisionen zwischen nichtleeren Einträgen dieser Zeile und nichtleeren Einträgen bereits abgelegter Zeilen vorhanden sind. Dann werden seine nichtleeren Einträge eingetragen. In *RowPtr*[i] wird dann der Index in *Delta* abgespeichert, ab dem die i-te Zeile in *Delta* abgelegt ist, siehe Abbildung 7.11.

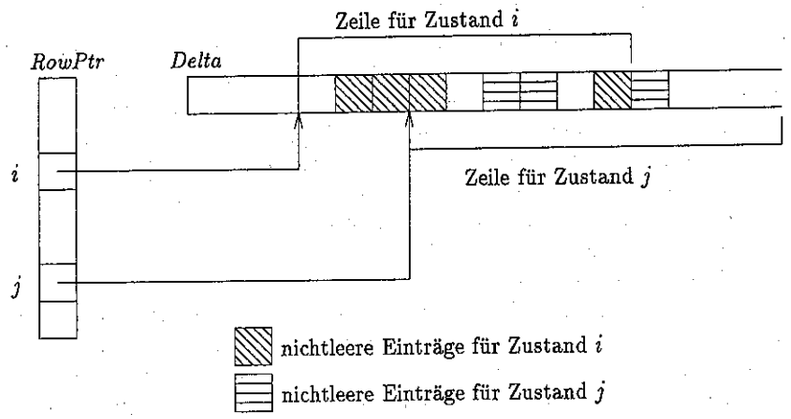


Abb. 7.11: Komprimierte Darstellung der Übergangsfunktion eines DEA

Jetzt hat allerdings der dargestellte Automat die Fähigkeit verloren, immer undefinierte Übergänge zu erkennen, etwa wenn $delta(i, a)$ undefiniert ist, aber $Delta[RowPtr[i] + a]$ einen nichtleeren Eintrag vom Zustand $j \neq i$ enthält[2]. Deshalb müssen wir ein zweites Feld *Pruef* der gleichen Länge wie *Delta* spendieren, welches angibt, zu welchen Zuständen Einträge in *Delta* gehören; d.h. $Pruef[RowPtr[i] + a] = i$, wenn $delta(i, a)$ definiert ist.

Die Übergangsfunktion des DEA kann dann durch eine Funktion *nextstate* wie folgt implementiert werden:

function *nextstate* (i : *state, a: charclass*) **returns** *state;*
  if $Pruef[RowPtr[i] + a] = i$
  then $Delta[RowPtr[i] + a]$
  else -1 (* steht für undefiniert *)
  fi

7.5 Der Sieber

Für die Verteilung der Aufgaben zwischen Scanner und Sieber und für die Funktionalität des Siebers gibt es viele Möglichkeiten, deren Vorteile bzw. Nachteile

[2] wir benutzen hier Zeichenklassen und ihre Nummer gleichwertig.

nicht ganz leicht zu beurteilen sind. Einige solche Alternativen werden im folgenden diskutiert.

7.5.1 Die Erkennung von Schlüsselwörtern

Nach der Aufgabenverteilung im letzten Kapitel kennt der Sieber die Menge der reservierten Bezeichnungen, auch Schlüsselwörter (keywords) genannt. Das setzt voraus, daß der Scanner eine oder mehrere Symbolklassen hat, in denen diese Symbole liegen. Dies ist z.B. dann der Fall, wenn wie in Pascal, C, Ada die Schlüsselwörter die gleiche Struktur wie Bezeichner haben. Bei obiger Aufgabenstellung wird dann der Scanner für jeden Bezeichner, ob reserviert oder nicht, das Vorliegen eines Bezeichners melden. Der Sieber wird anschließend feststellen, ob es sich dabei um ein reserviertes Symbol handelt. Diese Aufgabenverteilung hält die Mengen der Zustände und der Übergänge des Scannerautomaten klein. Allerdings muß der Sieber eine effiziente Möglichkeit des Erkennens von Schlüsselwörtern z.B. mithilfe einer Hashtabelle besitzen. Abbildung 7.12. zeigt einen NEA, der einige Schlüsselwörter in Endzuständen erkennt.

7.5.2 Scanner mit Aufrufschnittstelle zum Sieber

Ein Scannergenerator ist ein ziemlich universell einsetzbares Instrument. In vielen Bereichen gibt es Anwendungen für generierte Scanner, also die Aufgabe Exemplare regulärer Ausdrücke zu finden. Dann fallen oft die Aufgaben des Siebers gar nicht an. Ein solcher Scannergenerator bietet dann meist die Möglichkeit an, mit den Endzuständen eines Scannerautomaten Aufrufe von Funktionen in einer Programmiersprache zu assoziieren. Der Benutzer des Scannergenerators schreibt also die Funktionen zur Weiterverarbeitung der gefundenen Symbole, zur Buchführung etc. selbst. Von dieser Art ist etwa der Scannergenerator LEX unter UNIX.

Im Gegensatz dazu kennt man viele der Aufgaben, die in Übersetzern rund um die lexikalische Analyse anfallen, ganz gut. Deshalb lohnt es sich, Funktionen dafür in einem Sieber zu implementieren. Ein solcher Entwurf, wie er in dem Übersetzergenerator POCO [Eul88] realisiert wurde, wird im folgenden Abschnitt geschildert.

7.5.3 Symbolklassen

Symbolklassen sind Mengen von Symbolen, die für den „Konsumenten“ – im Übersetzer also den Parser – äquivalent sind. Zwei Symbole sind dann äquivalent, wenn der Parser in jedem Zustand unter beiden Symbolen den gleichen Übergang macht bzw. die gleiche Entscheidung trifft. Typische Symbolklassen sind die verschiedenen Klassen von Konstanten, die Bezeichner (ohne die reservierten Symbole), die Kommentare, arithmetische Operatoren gleicher Präzedenz und die Vergleichsoperatoren.

Der Beschreiber einer Scanner-Sieber-Kombination wird solche Klassen definieren. Er selbst explizit oder der Generator implizit werden jeder Klasse einen