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.

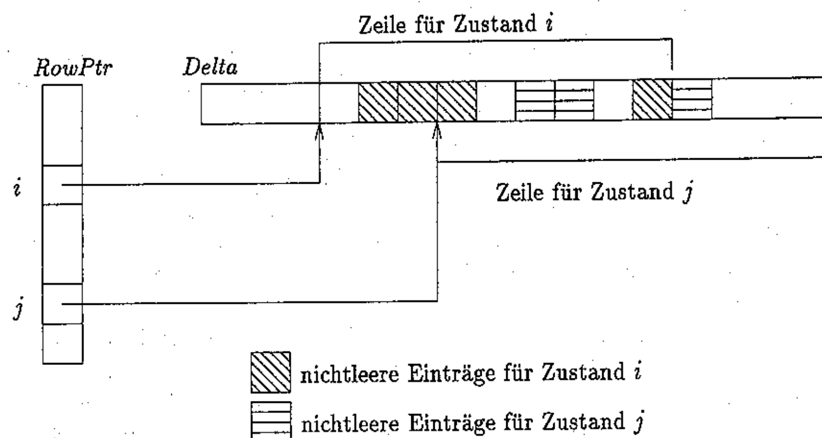

Abb. 7.11: Komprimierte Darstellung der Übergangsfunktion eines DEA

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

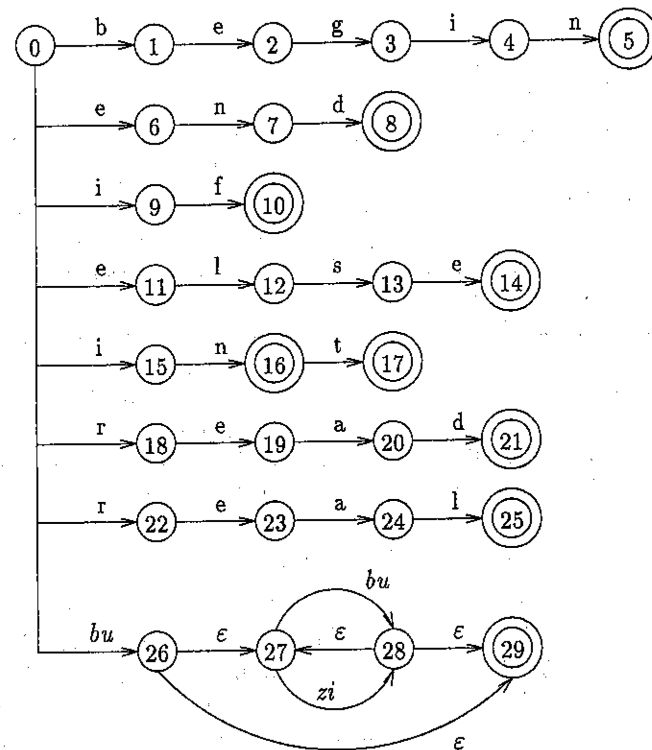

Abb. 7.12: Ein NEA zur Erkennung von Bezeichnern und von den Schlüsselwörtern begin, end, if, else, int, read, real.

eindeutigen Klassencode zuordnen. Dieser Klassencode wird beim Finden eines Symbols der Klasse durch den generierten Scanner an den Parser weitergegeben.

Beispiel 7.5.1

% Zeichenklassen
bu = a - z
zi = 0 - 9
% Symbolklassen
AddOp = + | −
MulOp = * | /
VglOp = < | > | = | <> | >= | <=
} definiert durch Aufzählung „Aufzählungsklassen"

Id = $bu(bu\,|\,zi)^*$
IntConst = $zi\ zi^*$
} definiert durch reg. Ausdruck mit Iteration „unendliche Klasse" □

Natürlich ist es für die semantische Analyse und für die Codeerzeugung unbe-

dingt notwendig zu wissen, welches Element einer Symbolklasse gefunden worden ist. Deshalb gibt der Scanner/Sieber außer dem Klassencode auch noch einen Relativcode für das gefundene Symbol weiter, welches i.a. vom Parser nicht benutzt, aber zur Weiterverwendung gemerkt wird.

Gibt es verschiedene, jedoch syntaktisch und semantisch äquivalente Symbole, so können diese jeweils innerhalb einer Symbolklassendefinition zusammengefaßt werden.

Beispiel 7.5.2

VglOp = (<,*lt*) | (>,*gt*) | (=,*eq*) | (<>,*neq*) | (>=,*ge*) | (<=,*le*) □

Die Relativcodes für Elemente von Aufzählungsklassen, das sind Klassen, deren endlich viele Elemente durch Auflistung in der Klassendefinition gegeben sind, ergeben sich aus der Position der Elemente in dieser Auflistung. Elemente unendlicher Klassen werden durchnumeriert in der Reihenfolge, in der sie in einem zu analysierenden Programm auftreten. Dabei kann ein komfortabler Scannergenerator dem Benutzer noch die Option anbieten,

- jedem Element der Klasse einen eindeutigen Relativcode zuzuordnen; dann muß der Scanner/Sieber verschiedene Vorkommen desselben Symbols identifizieren;
- jedem Vorkommen eines Symbols einen eindeutigen Relativcode zuzuordnen.

Die erste Option wird man typischerweise für die Symbolklasse der Bezeichner wählen. Für eine Symbolklasse wie die der Zeichenfolgen (strings) würde man vermutlich die zweite Option wählen, weil der Aufwand für den Vergleich jeder neu gefundenen Zeichenfolge mit allen vorhandenen beträchtlich und der Gewinn mäßig wäre.

Es ist aber klar, daß ein solcher Sieber die Aufgabe hätte, über die Elemente unendlicher Klassen Buch zu führen. Er wird also die gefundenen Symbole in einer geeigneten Datenstruktur abspeichern. Die Datenstruktur muß folgende Operationen erlauben: Eintragen mit Vergabe eines neuen Relativcodes, Test auf vorhandenen Eintrag mit Rückgabe des Relativcodes im Erfolgsfall, Finden der Externdarstellung des Symbols mithilfe von Klassencode und Relativcode. Solch eine Datenstruktur würde initialisiert mit vordefinierten Symbolen aus Aufzählungsklassen wie etwa den reservierten Symbolen.

Es gibt weitere Direktiven an einen Sieber für die Behandlung der Elemente einer Symbolklasse. Z.B. sollte es möglich sein zu spezifizieren, daß jedes Element einer bestimmten Klasse ignoriert, d.h. nicht an den Parser weitergegeben wird.

Beispiel 7.5.3

% Einzelzeichenklassen
open = (
close =)
star = *

% Symbolklassen
- Comment = open star until (star close)
- Separator = (Blank | NL)$^+$
Das Minuszeichen vor dem Klassennamen gibt an, daß alle Elemente der beiden Klassen ignoriert werden sollen. □

7.6 Flex, ein Scanner-Generator unter UNIX

In diesem Abschnitt werden die wesentlichen Eigenschaften eines Scanner-Generators beschrieben, der unter UNIX als Public Domain Software zur Verfügung steht.

7.6.1 Die Arbeitsweise von Flex-generierten Scannern

Der Kern des Generators ist die Erzeugung eines deterministischen endlichen Automaten zur Erkennung einer regulären Sprache. Der erzeugte DEA erkennt wie der vorher beschriebene DEA maximal lange Präfixe der restlichen Eingabe als Symbole (token). Es gibt einige Möglichkeiten, auf das „Konsumverhalten“ des DEA Einfluß zu nehmen, z.B. gelesene Zeichen wieder vor die restliche Eingabe einzufügen oder weitere Anfangszeichen der restlichen Eingabe zu konsumieren.

Es gibt keinen Sieber, wie er im vorangegangenen Abschnitt beschrieben worden ist. Statt dessen gibt es eine Schnittstelle zu C; Benutzerdefinierte C-Funktionen können bei Erreichen jedes Endzustandes aufgerufen werden. Dise sind für die Weiterverarbeitung von gefundenen Symbolen zuständig. Das zuletzt gefundene Symbol steht über einen globalen Zeiger (`yytext`) zur Verfügung, seine Länge in einer globalen integer-Variablen (`yyleng`). Der ganze erzeugte Scanner ist eine C-Funktion `yylex`, welche von anderen Programmen, etwa einem Parser aufgerufen werden kann.

7.6.2 Flex-Spezifikationen

Flex-Eingabe gliedert sich in drei Abschnitte, getrennt jeweils durch '%%':

Definitionen
%%
Regeln
%%
C-Programme des Benutzers

Zur Vereinfachung der Beschreibung der Regeln führt der **Definitionen**-Abschnitt Namen für reguläre Ausdrücke ein. Diese Namen können, eingeschlossen in geschweifte Klammern, in späteren Definitionen und in Regeln anstelle dieser regulären Ausdrücke stehen.
Beispiel:

```
ZI [0--9]
BU [a - z]
ID {BU} ({BU} | {ZI})*
```


Diese Definitionen sollten selbsterklärend sein.

Die **Regeln** bestehen jeweils aus einem *Muster*, einem regulären Ausdruck, und einer *Aktion*, einem Stück C-Code. Dieser C-Code wird ausgeführt, wenn der erzeugte Scanner ein Symbol als zu diesem Muster gehörend erkennt.
Beispiel:

```
{ZI}+      printf ("Eine ganze Zahl: %s (%d)\n", yytext,
                                          atoi(yytext));
"+"|"-"|"*"|"/"  {
           printf("Ein Operator: %s\n", yytext);
           }
if|then|else|while|do|begin|end {
           printf ("Ein Schluesselwort: %s\n", yytext);
           }
{ID}       printf ("Ein Name: %s\n", yytext);
[ \t\n]    /* konsumiert Leerzeichen, Tabs und Zeilenenden */
```

Einige Bemerkungen zu diesem Beispiel:

- Zwischen eckigen Klammern, '[' und ']', kann man Zeichenklassen aufführen, wie im Definitionsteil auch durch Angabe von Bereichen. Einzelne Zeichen und Zeichenklassen können auch negiert auftreten.
- '*', '+', '—' als Metazeichen haben die übliche Bedeutung. Treten sie als Zeichen der spezifizierten Eingabe auf, werden sie durch Doppelapostroph geschützt.
- Die Schlüsselworte `if`, ..., `end` sind natürlich auch Mitglieder der Klasse `ID`. Der Konflikt zwischen den Endzuständen zu diesen regulären Ausdrücken wird zugunsten der Schlüsselwörter entschieden, da sie früher in der Spezifikation stehen.

Eine weitere Vereinfachung der Beschreibung ist mit *bedingten Regeln* möglich. Regeln, denen in der Spezifikation der Name einer Bedingung vorangestellt ist, werden nur benutzt, wenn die entsprechende Bedingung aktiviert worden ist.
Beispiel:

```
<STRING>[^"]*  { /* Code, der eine String-Konstante aufbaut */ }
```

Diese Regel muß aktiviert werden, wenn der einleitende Doppelapostroph einer String-Konstanten gefunden wurde. Sie liest dann solange, wie kein weiterer Doppelapostroph gefunden wird. Bedingte Regeln können an- und abgeschaltet werden, allein und in Gruppen.

7.7 Übungen

2.1 Sei Σ ein Alphabet, und für Mengen $P \subseteq \Sigma^*$ sei $P^* = \{p_1 \dots p_k \mid k \geq 0,\ p_i \in P\}$, außerdem $\emptyset^* = \{\varepsilon\}$.
Man zeige:
(a) $L \subseteq L^*$
(b) $\varepsilon \in L^*$
(c) $u, v \in L^* \Rightarrow uv \in L^*$ für alle Worte $u, v \in \Sigma^*$.
(d) L^* ist die kleinste Menge mit den Eigenschaften (a) - (c), d.h. wenn für eine Menge M gilt:
$L \subseteq M,\ \varepsilon \in M$ und $(u, v \in M \Rightarrow uv \in M)$, dann ist $L^* \subseteq M$.
(e) $L \subseteq M \Rightarrow L^* \subseteq M^*$
(f) $L^{**} = L^*$

2.2 Konstruieren Sie den minimalen deterministischen endlichen Automaten für die Folge von regulären Definitionen:

id	=	*bu* (*bu* | *zi*)*
sysid	=	*bu* & (*bu* | *zi*)*
comid	=	*bu bu* & (*bu* | *zi*)*

2.3 (a) Geben Sie eine Folge von regulären Definitionen für die römischen Zahlen an.
(b) Erzeugen Sie daraus einen deterministischen endlichen Automaten.
(c) Ergänzen Sie diesen Automaten auf folgende Weise um eine Möglichkeit den Dezimalwert einer römischen Zahl zu berechnen: Mit jedem Zustandsübergang darf der Automat eine Wertzuweisung an **eine** Variable w durchführen. Der Wert ergibt sich aus einem Ausdruck über w und Konstanten. w wird initialisiert zu 0. Geben Sie zu jedem Zustandsübergang eine geeignete Wertzuweisung an, so daß in jedem Endzustand w den Wert der erkannten Zahl enthält.

2.4 Machen Sie den NEA aus Abbildung 7.12 deterministisch.

2.5 Erweitern Sie den Algorithmus **RA→ NEA** aus Abbildung 7.3 so, daß er auch reguläre Ausdrücke r^+ und $[r]$ verarbeitet. r^+ steht für rr^* und $[r]$ für $(\varepsilon|r)$.

2.6 Erweitern Sie den Algorithmus **RA→ NEA** um die Verarbeitung zählender Iteration, d.h. um reguläre Ausdrücke der Form

$r\{u - o\}$	mindestens u und höchstens o aufeinanderfolgende Exemplare von r
$r\{u-\}$	mindestens u aufeinanderfolgende Exemplare von r
$r\{-o\}$	höchstens o aufeinanderfolgende Exemplare von r

3.1 Fortran erlaubt die implizite Deklaration von Bezeichnern nach ihrem Anfangsbuchstaben. Geben Sie Definitionen für die Symbolklassen *realid* und *intid* an.

4.1 Gegeben seien die folgenden Definitionen von Einzelzeichenklassen:

bu	=	a - z
zi	=	0 - 9

und die Symbolklassendefinitionen

bin	=	b (0|1)+
oct	=	o (0|1|2|3|4|5|6|7)+
hex	=	h (*zi*|*A*|*B*|*C*|*D*|*E*|*F*)+
intconst	=	*zi*+
id	=	*bu* (*bu* | *zi*)+

(a) Geben Sie die vom Scannergenerator berechnete Einteilung in Einzelzeichenklassen an.
(b) Beschreiben Sie den generierten NEA unter Benutzung dieser Einzelzeichenklasseneinteilung.
(c) Machen Sie diesen NEA deterministisch.

4.2 Geben Sie eine Implementierung des *until*-Konstrukts, R_1 *until* R_2 an.

4.3 Komprimieren Sie die Tabellen der von Ihnen „zu Fuß" erstellten deterministischen endlichen Automaten mittels des Verfahrens aus Abschnitt 7.2.

7.8 Literaturhinweise

Die konzeptionelle Trennung in Scanner und Sieber wurde von F. DeRemer vorgeschlagen [DeR74]. Die Generierung von Scannern aus regulären Ausdrücken wird in vielen sogenannten Übersetzergeneratoren unterstützt. Johnson u.a. [JPAR68] beschreiben ein solches System. Das entsprechende Dienstprogramm unter UNIX, LEX, wurde von M. Lesk entwickelt [Les75]. FLEX wurde von Vern Paxson geschrieben. Das in diesem Kapitel beschriebene Konzept lehnt sich an den Scannergenerator des POCO-Übersetzergenerators an [Eul88].

Kompressionsmethoden für schwach besetzte Tabellen, wie sie typischerweise bei der Scanner- und der Parsergenerierung erzeugt werden, werden in [TY79] und [DDH84] analysiert und verglichen.