2

Lexikalische Analyse

Die lexikalische Analyse hat die Aufgabe, ein als Zeichenkette in einer Datei vorliegendes *Quellprogramm* in eine Kette von *Lexemen* zu zerlegen. (Lexem ist ein linguistischer Begriff für ein *lexikalisches Element*; in natürlichen Sprachen sind dies die Wörter; im Übersetzerbau werden Lexeme manchmal *Symbol* oder (engl.) *token* genannt.)

Wie fast alle anderen Phasen eines Übersetzers gliedert sich die lexikalische Analyse in zwei Teilaufgaben:

1. Das Erkennen eines Lexems im Quelltext mit einem *Leseprogramm*, engl. *Scanner* genannt.
2. Das Übersetzen des erkannten Lexems für Weiterverarbeitung in der Syntaxanalyse, mit einem *Sieber*, engl. *Screener*.

Die lexikalische Analyse zerfällt also auch in eine Analysephase – den Scanner – und eine Synthesephase – den Screener.

In diesem Kapitel diskutieren wir zunächst Zeichensätze und Lexeme, wiederholen wir dann einige Grundlagen der formalen Sprachen, führen reguläre Ausdrücke und Definitionen ein, mit denen die Lexeme von Programmiersprachen beschrieben werden können, und behandeln dann die Erkennung von Lexemen mit (deterministischen) endlichen Automaten. Schließlich gehen wir noch auf einge Aspekte des Siebens ein, insbesondere auf die Behandlung von Bezeichnern, und betrachten praktische Implementierungsfragen, bevor wir zum Schluss die Erzeugung von Scannern (und Siebern) mit dem UNIX-Werkzug `lex` beschreiben.

2.1 Zeichen und Lexeme

2.1.1 Zeichensätze

2.1.2 Lexemarten

2.2 Sprachen und Grammatiken

In diesem Abschnitt wiederholen wir einige Grundbegriffe aus den *formalen Sprachen*, die aus dem Kurs *Theoretische Informatik I* bekannt sein sollten. Wir be-

schränken uns dabei auf die im Übersetzerbau benötigten Definitionen und Ergebnisse.

2.2.1 Wörter

Wir gehen aus von einer nicht leeren endlichen Menge V, die *Vokabular* genannt wird. Die Elemente von V werden *Zeichen* genannt. Für die lexikalischen Analyse ist V ein Zeichensatz wie ASCII oder UNICODE, für die Syntaxanalyse das Vokabular der Lexeme.

Die *Wörter über* V sind definiert als die Menge der endlichen Zeichenketten

$$V^* = \{v_1 \cdots v_n \mid v_i \in V, 1 \leqslant i \leqslant n, n \geqslant 0\}$$

Mit $|w|$ bezeichnen wir die Länge eines Wortes $w \in V^*$; wir schreiben ε für das (einzige) Wort der Länge 0. V^n bezeichnet die *Wörter der Länge* $n \geqslant 0$ *über* V; also ist $V^0 = \{\varepsilon\}$, $V^1 = V$ und $V^* = \bigcup_{n \geqslant 0} V^i$. Ferner bezeichnet $V^+ = \bigcup_{n \geqslant 1} V^i$ die Menge der *nicht leeren Wörter über* V.

Die *Konkatenation* von zwei Wörtern $w, w' \in V^*$ wird durch Hintereinanderschreiben ww' ausgedrückt. Mit w^n bezeichnen wir das Wort, das aus der Konkatenation von n Kopien von w entsteht.

In einem Wort der Form $w = w'w''$ ist w' ein *Präfix* (und w'' ein *Suffix*) von w; Dieser Präfix w' (und Suffix w'') ist *echt*, wenn weder w noch w' (bzw. w'') leer ist.

2.2.2 Sprachen

Eine Teilmenge $L \subseteq V^*$ heißt *Sprache über* V; meistens nehmen wir an, dass L nicht leer ist. Oft erwarten wir auch, dass eine Sprache nicht das leere Wort ε enthält. (Aus Sicht eines Übersetzerbauers macht dass durchaus Sinn, weil auch ein "leerer Text" mindestens ein Zeichen enthät, der das Ende des Textes markiert.)

Wir benutzen die folgenden Operationen auf Sprachen L und L': Das *Komplement* von L ist die Sprache $\bar{L} = V^* \setminus L$. und die *Verkettung* zweier Sprachen L und L' ist die Sprache $L \cdot L' = \{ww' \mid w \in L, w' \in L'\}$. Weiterhin ist $L^0 = V^0 = \{\varepsilon\}$ und $L^{n+1} = L \cdot L^n$ für $n \geqslant 0$. Schließlich ist der *transitive Abschluss* von L gegeben als $L^+ = \bigcup_{n \geqslant 1} L^i$, und ihr *transitive-reflexiver Abschluss* als $L^* = \bigcup_{n \geqslant 0} L^i$.

2.2.3 Grammatiken

Grammatiken generieren Sprachen durch Ableitungen nach Regeln. Eine Grammatik ist gegeben als $G = (V, T, R, S)$; hierin ist

- V ein Vokabular,
- die Teilmenge $T \subseteq V$ das *terminale Vokabular*, woraus sich die Menge von *nichtterminalen Zeichen* als $N = V \setminus T$ ergibt,
- $R \subseteq V^* \times V^*$ eine endliche Menge von *Regeln*, und
- $s \in N$ das *Startsymbol*.

Wir setzen voraus, dass in einer Regel $(\alpha, \beta) \in R$ die linke Seite α mindestens ein Nonterminal enthält.

Im Zusammenhang mit Grammatiken verwenden wir folgende Konventionen: Buchstaben u, w bezeichnen terminale Wörter aus T^*. Griechische Buchstaben α, β, γ bezeichnen allgemeine Wörter aus V^*. Die Buchstaben n, m, z bezeichnen Nichtterminale aus N. Buchstaben a, b, c bezeichenen Terminale aus T.

Die *Ableitungsschritte* mit den Regeln R einer Grammatik sind induktiv als die kleinste Relation $\Rightarrow_R \subseteq V \times V$ mit folgenden Eigenschaften definiert:

- Für jede Regel $(\alpha, \beta) \in R$ gilt $\alpha \Rightarrow_R \beta$.
- Wenn γ, δ beliebige Worte über V sind und $\alpha \Rightarrow_R \beta$ gilt, so gilt auch $\gamma\alpha\delta \Rightarrow_R \gamma\beta\delta$.

Die *Ableitungsrelation* $\Rightarrow_R^* \subseteq V \times V$ ist der transitiv-reflexive Abschluss der Ableitungsschritt-Relation. Das heißt, wir schreiben $\alpha \Rightarrow_R^* \beta$ wenn entweder $\alpha = \beta$ gilt, oder es eine Folge von Ableitungsschritten $\alpha_0 \Rightarrow_R \alpha_1 \Rightarrow_R \cdots \Rightarrow_R \alpha_n$ mit $\alpha = \alpha_0$ und $\alpha_n = \beta$ gibt, wobei $\alpha_0, \alpha_1, \ldots, \alpha_n \in V^*$.

Dann ist die von einer Grammatik G erzeugte *Sprache* als die Menge aller terminalen Wörter definiert, die sich mit ihren Regeln aus dem Startsymbol ableiten lassen:

$$\mathcal{L}(G) = \{w \in T^* \mid z \Rightarrow_R^* w\}$$

2.2.4 Die Chomsky-Hierarchie

Eine Grammatik werden nach der Form ihrer Regeln in vier Typen eingeteilt:

- Sind alle Regeln von der Form (n, t) oder (n, tm) (bzw. von der Form (n, t) oder (n, mt)), die eine Grammatik vom *Typ 3* und heißt *regulär*.
- Sind alle Regeln von der Form (n, α), ist die Grammatik vom *Typ 2* und heißt *kontextfrei*.
- Gilt für alle Regeln (α, β) dass $|\beta| le |\alpha|$, ist die Grammatik vom *Typ 1* und heißt *kontextsensitiv*.
- Sonst ist die Grammatik vom *Typ 0* und heißt *rekursiv aufzählbar*.

Nicht alle kontextfreie Regeln sind kontext-sensitiv, denn die rechte Seite α so einer Regel darf leer sein. Dann wäre ihre linke Seite länger als ihre rechte, und die Regel nicht kontext-sensitiv. Allerdings kann man jede kontextfreie Grammatik so umformen, dass sie nur dann eine solche *leere Regel* enthält, wenn auch ihre Sprache das leere Wort enthält, und in diesem Fall reicht eine einzige leere Regel (z, ε) aus. Alle anderen kontextfreien Regeln der Grammatik sind dann auch kontextsensitiv. Abgesehen von dieser Ausnahme sind die Regeln höheren Typs immer spezieller als die niedrigeren Typs.

Eine Sprache L ist *vom Typ i* – bzw. *regulär*, *kontextfrei*, *kontextsensitiv* oder *rekursiv aufzählbar* – wenn sie von einer Grammatik des Typs i erzeugt wird. Die Mengen der Sprachen $\mathcal{L}^i$ vom Typ i bilden eine echte Hierarchie:

$$\mathcal{L}^3 \subsetneq \mathcal{L}^2 \subsetneq \mathcal{L}^1 \subsetneq \mathcal{L}^0$$

(Auch hier gilt die – unwesentliche – Einschränkung, dass kontextfreie Sprachen, die das leere Wort enthalten, nicht kontextsensitiv sind.) Für den Übersetzerbau sind vor allem reguläre und kontextfreie Sprachen und Grammatiken von Bedeutung. Reguläre Grammatiken (oder eher noch die gleich mächtigen *regulären Ausdrücke*)

werden zur Spezifikation der lexikalischen Einheiten und zur Erzeugung von *Leseprogrammen* (*Scanner*) für Übersetzer benutzt. Kontextfreie Grammatiken werden zur Definition der Syntax und zur Erzeugung von Parsierern für die Quellsprache eines Übersetzers verwendet.

Auch wenn einige Aspekte der Syntax einer Programmiersprache mit kontextfreien Grammatiken nicht beschrieben werden können, werden hierfür trotzdem nicht kontextsensitive Grammatiken benutzt, sondern Erweiterungen von kontextfreien Grammatiken wie z.B. Attributgrammatiken, mit denen solche Eigenschaften intuitver spezifiziert werden können, und aus denen sich die Kontextanalyse eines Übersetzers besser generieren lässt.

2.3 Reguläre Ausdrücke

Reguläre Ausdrücke bzw. reguläre Definitionen sind ein Standard zur Definition von regulären Sprachen; damit können Lexeme etwas kompakter beschrieben werden als mit regulären Grammatiken.

Die Syntax regulärer Ausdrücke ist wie folgt definiert:

1. Das Symbol v ist ein regulärer Ausdruck, für jedes $v \in V$.
2. Das leere Wort ε ist ein regulärer Ausdruck.
3. Sind R_1 und R_2 reguläre Ausdrücke, so ist auch
   a) die *Alternative* $R_1 \mid R_2$ und
   b) die *Verkettung* $R_1\, R_2$
   ein regulärer Ausdruck.
4. It R ein regulärer Ausdruck, so ist auch
   a) die *Option* $R^?$,
   b) die *nicht leere Iteration* R^+ und
   c) die *Iteration* R^*
   ein regulärer Ausdruck.

Die Sprache eines regulären Ausdrucks ist rekursiv wie folgt definiert:

1. $L(v) = \{v\}$ für alle $v \in V$.
2. $L(\varepsilon) = \{\varepsilon\}$.
3. Für alle regulären Ausdrücke R_1, R_2' gilt
   a) $L(RR') = L(R) \cdot L(R')$ und
   b) $L(R|R') = L(R) \cup L(R')$.
4. Für alle regulären Ausdrücke R gilt
   a) $L(R^?) = L(R) \cup \{\varepsilon\}$,
   b) $L(R^+) = L(R)^+$ und
   c) $L(R^*) = L(R)^*$.

Reguläre Ausdrücke generieren tatsächlich die reguläre Sprachen, sind also äquivalent – aber besser benutzbar – als reguläre Grammatiken.

Für das praktische Erstellen von regulären Ausdrücken erlauben wir noch die Benutzungvon Hilfsdefinitionen. dann können wir die regulären AusdrÜcke für Zahlen und Bezeichner

$(0|1|2|3|4|5|6|7|8|9)^+$

(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)
(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
|0|1|2|3|4|5|6|7|8|9)$^+$

übersichtlicher definieren als

$$
\begin{aligned}
\mathit{digit} &= 0|1|2|3|4|5|6|7|8|9 \\
\mathit{letter} &= \mathrm{a}|\mathrm{b}|\mathrm{c}|\mathrm{d}|\mathrm{e}|\mathrm{f}|\mathrm{g}|\mathrm{h}|\mathrm{i}|\mathrm{j}|\mathrm{k}|\mathrm{l}|\mathrm{m}|\mathrm{n}|\mathrm{o}|\mathrm{p}|\mathrm{q}|\mathrm{r}|\mathrm{s}|\mathrm{t}|\mathrm{u}|\mathrm{v}|\mathrm{w}|\mathrm{x}|\mathrm{y}|\mathrm{z} \\
&\quad | \ \mathrm{A}|\mathrm{B}|\mathrm{C}|\mathrm{D}|\mathrm{E}|\mathrm{F}|\mathrm{G}|\mathrm{H}|\mathrm{I}|\mathrm{J}|\mathrm{K}|\mathrm{L}|\mathrm{M}|\mathrm{N}|\mathrm{O}|\mathrm{P}|\mathrm{Q}|\mathrm{R}|\mathrm{S}|\mathrm{T}|\mathrm{U}|\mathrm{V}|\mathrm{W}|\mathrm{X}|\mathrm{Y}|\mathrm{Z} \\
\mathit{letgit} &= \mathit{letter} \mid \mathit{digit} \\
\mathit{number} &= \mathit{digit}^+ \\
\mathit{id} &= \mathit{letter}\ \mathit{letgit}^*
\end{aligned}
$$

Hilfsbegriffe wie *digit* und *number* dürfen aber nicht rekursiv verwendet werden. (Solche Definitionen wären nicht mehr regulär, sondern schon *kontextfrei*!)

2.4 Endliche Automaten

Ein endlicher Automat $A = (V, Q, \Delta, q_0, F)$ besteht aus folgenden Komponenten:

- V ist ein *Vokabular*.
- Q ist eine endliche *Zustandsmenge*.
- $\Delta \subseteq Q \times (V \cup \{\varepsilon\}) \times Q$ ist eine (endliche) Menge von *Zustandsübergängen*.
- $q_0 \in Q$ ist der *Startzustand*.
- $F \subseteq Q$ ist eine Menge von *Endzuständen*.

Ein endlicher Automat A wie oben ist *deterministisch* wenn gilt:

1. Es gibt keine mit ε markierten Übergänge, und
2. Für alle Übergänge (q, v, q') und (q, v', q'') in Δ folgt aus $v = v'$ dass gilt $q' = q''$.

Sonst heißt der Automat *nichtdeterministisch*.

Ein endlicher Automat liest Wörter Zeichen für Zeichen von links nach rechts. In einem Zustand q kann der Automat einen Zustandsübergang (q, v, q') vollziehen, wenn der noch zu lesende Teil des Worts mit v beginnt; im Folgezustand q' wird dann mit dem Rest des Worts fortgefahren. Der Automat beginnt mit dem Lesen des gesamten Worts im Startzustand q_0. Ist der Automat nach dem vollstängigen Lesen des Wortes in einem Endzustand, gehört es zu der von ihm erkannten Sprache. Kann das Wort nicht vollständig gelesen werden, weil es in einem der Zustände keinen passenden Zustandsübergang gibt, oder stoppt der Automat in einem anderen als einem Endzustand, gehört es nicht zur Sprache, ist "fehlerhaft".

Wir stellen einen endlichen Automaten als *Zustandsübergangsdiagramm* dar. Dies ist ein gerichteter Graph mit markierten Kangten, der wie folgt aufgebaut ist:

- Seine Knoten stellen die Zustände dar, die meist mit fortlaufenden Nummern benannt werden.
- Seine Endzustände werden mit Doppellinien gezeichnet.

- Eine mit einem Zeichen $v \in V \cup \{\varepsilon\}$ markierte Kante von einem Knoten q zu einem Knoten q' stellt einen Zustandsübergang $(q, v, q') \in \Delta$ dar.
- Der Knoten mit der Inschrift 0 ist der Startzustand.

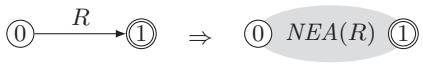
Wir schreiben "$q \rightarrow_v q'$" wenn es einen Übergang von Zustand q in Zustand q' unter dem Zeichen $v \in T \cup \{\varepsilon\}$ gibt und erweitern dies auf Pfade beliebiger Länge, indem wir $q \rightarrow^*_\varepsilon q$ schreiben für Pfade der Länge 0, und $q \rightarrow^*_{vw} q'$ wenn es einen Zustand q'' gibt so dass $q \rightarrow_v q''$ und $q'' \rightarrow^*_w q'$.

Im Zustandsübergangsdiagramm bilden die Kantenmarkierungen an jedem Pfad vom Startknoten zu einem der Endknoten ein vom Automaten erkannten Wort: Wenn $q_0 \rightarrow^*_w q_f$ für $q_f \in F$, so erkennt A das Wort w. Indem man vom Startzustand den Kanten auf beliebigen Pfaden bis zu einem Endzustand folgt, kann man also die von ihm erkannten Wörter bilden.

Endliche Automaten erkennen reguläre Sprachen. Insofern ist es naheliegend, zu fragem, welcher Zusammenhang zwischen regulären Ausdrücken (die ja äquivalent zu regulären Grammatiken sind) und endliche Automaten besteht. Tatsächlich kann man für jeden regulären Ausdruck R konstrutiv einen endlichen Automaten A_R bestimmen, der die von R beschriebene Sprache erkennt. In der Regel ist dieser Automat zunächst nicht deterministisch. Die sogenannte *Potenzmengenkonstruktion* erlaubt es dann, zu jedem nicht determinitsischen Automaten einen äquivalenten deterministischen Automaten herzuleiten. Deterministische endliche Automaten können außerdem in Hinsicht auf die Anzahl ihrer Zustände minimiert werden. Schließlich kann man deterministische endliche Automaten auf zweierlei Weise programmieren: Durch sehr einfache iterative Programme (die im Wesentlichen aus bedingten Sprüngen bestehen) und als Tabellen, die mit einem allgemeinen, ebenfalls sehr einfachen iterativen Programm interpretiert werden.

2.4.1 Übersetzung regulärer Ausdrücke in endliche Automaten

Für jeden regulären Ausdruck R kann ein endlicher Automat angegeben werden, der die von R definierte Sprache erkennt. Wir beschreiben diese Konstruktion mit den Transformationsregeln in Abb. 2.1 auf *erweiterten Zustandsübergangsdiagrammen*, deren Kanten nicht nur mit $V \cup \{\varepsilon\}$, sondern zunächst mit beliebigen regulären Ausdrücken R markiert sein können. Die mit regulären Ausdrücken beschrifteten Kanten werden solang durch Teilautomaten (mit "einfache") beschrifteten Zustandsübergängen ersetzt, bis das Diagramm ein gewöhnliches Zustandsdiagramm ist. (Siehe [5, Abschnitt 7.2.2]) Beginnt man mit einem Automat, in dem der einzige Startzustand durch eine mit R beschriftete Kante mit dem einzigen Endzustand verbunden ist, erhält man so einen nichtdeterministischen Automaten für R:



Die für reguläre Ausdrücke der Form R^* und R^+ eingesetzten Teilautomaten sehen zunächst komplizierter aus als nötig. Es ist jedoch zu beachten, dass diese Automaten an beliebigen Stellen in regulären Ausdrücken eingesetzt werden. In manchen Fällen wird so ein komplizierter Teilautomat wirklich benötigt.

Man ist vielleicht versucht, die Knoten s und m sowie n und t zu identifizieren, um die ε-Übergänge zu sparen. Dann würde jedoch ein Ausdruck der Form $R_1^+ \mid R_2$ so transformiert:

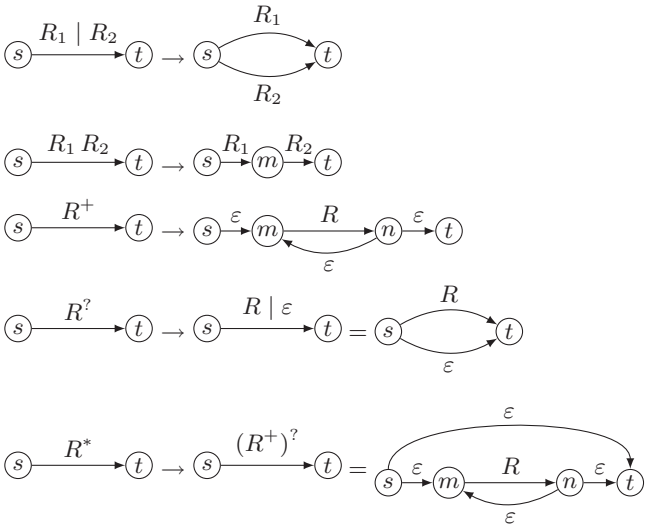
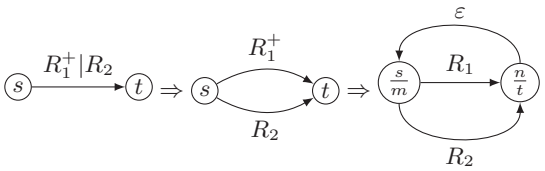


Abb. 2.1. Transformation von regulären Ausdrücken in nichtdeterministische Automaten (nach [5, Abschnitt 7.2.2])

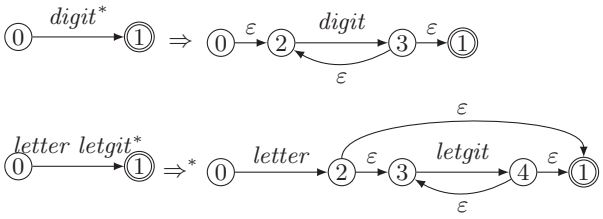


Dieser Automat erkennt den Ausdruck $(R \mid R')^+$, *nicht* den Ausdruck $R^+ \mid R'$.

Beispiel 2.1 (Nichtdeterministischer Automat für Zahlen und Bezeichner). Werte ganzer Zahlen und einfache Bezeichner können mit folgenden regulären Ausdrücken definiert werden:

digit$^+$ bzw. *letter letgit**

Mit Hilfe der Transformationsregeln aus Abbildung 2.1 entsteht folgender nichtdeterministische Automat:



2.4.2 Die Potenzmengenkonstruktion

Für einen nichtdeterministischen endlichen Automat kann ein deterministischer Automat konstruiert werden.

Dazu werden Zustände des nichtdeterministischen Automaten, zwischen denen Übergänge unter ε existieren, als äquivalent aufgefasst, genau so wie Folgezustände, zu denen man aus einem dieser Äquivalenten Zustände unter demselben Zeichen gelangt.

Sei $E(q)$ die Menge aller zu einem Zustand äquivalenten Zustände:

$$E(q) = \{q' \in Q \mid q \rightarrow_{\varepsilon}^{*} q'\},$$

und Sei $F(P,t)$ die Menge aller Folgezustände unter einem Zeichen $v \in V$ aus einer Menge $P \subseteq Q$ von (äquivalenten) Zuständen:

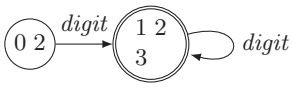
$$F(P,t) = \{q' \in Q \mid q \rightarrow_{v}^{*} q', q \in P\}.$$

Der Potenzmengenautomat eines Automaten $A = (V, Q, \Delta, q_0, F)$ hat den Startzustand $P_0 = E(q_0)$. Für jedes Zeichen $v \in T$ mit nicht leerer Menge $P_i = F(P_0, t)$ ist P_i ein Folgezustand von P_0 mit $P_0 \rightarrow_v P_i$. Mit der Bestimmung von Folgezuständen wird soweit fortgefahren, bis keine neuen mehr entstehen. Dies muss irgendwann der Fall sein, weil es nur endlich viele Teilmengen der endlichen Menge Q gibt. Ein Zustand P in diesem Automaten ist ein Endzustand, wenn $F \cap P \neq \emptyset$.

Beispiel 2.2 (Deterministische Automaten für Zahlen und Bezeichner). Für den nichtdeterministischen Automaten für Zahlen aus Beispiel 2.1 werden folgende Zustandsmengen berechnet:

1. $P_0 = E(0) = \{0, 2\}$.
2. $P_1 = F(P_0, \mathit{digit}) = \{1, 2, 3\}$.
3. $P_2 = F(P_1, \mathit{digit}) = \{1, 2, 3\} = P_1$.

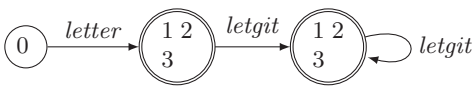
Also entsteht folgender deterministischer Automat:



Für den nichtdeterministischen Automaten für Bezeichner aus Beispiel 2.1 werden folgende Zustandsmengen berechnet:

1. $P_0 = E(0) = \{0\}$.
2. $P_1 = F(P_0, \mathit{letter}) = \{1, 2, 3\}$.
3. $P_2 = F(P_1, \mathit{letgit}) = \{1, 3, 4\}$.
4. $F(P_2, \mathit{letgit}) = \{1, 3, 4\} = P_2$.

Damit entsteht folgender deterministischer Automat:



2.4.3 Minimierung deterministischer endlicher Automaten

Automaten und ihre Zustandsübergangsdiagramme können Teile enthalten, die nutzlos sind: Nur Zustände, die wirklich auf einem Pfad vom Startzustand zu einem Endzudtand liegen, sind nützlich. Alle anderen können entfernt werden.

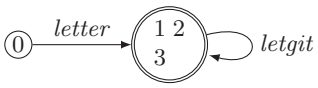
Zudem kann es Zustände geben, die äquivalent sind und zusammengelegt werden können. Zwei Zustände q_1 und q_2 sollen *äquivalent* heißen, geschrieben $q_1 \equiv q_2'$, wenn gilt:

1. Für alle $v \in T \cup \varepsilon\}$ gilt $q_1 \rightarrow_v q_1'$ genau dann wenn $q_2 \rightarrow q_2'$ und $q_1' \equiv q_2'$, und
2. $Q_1 \in F$ genau dann wenn $q_2 \in F$.

Äquivalente Zustände entdeckt man, indem man zunächst eine Äquivalenzrelation auf den Zuständes des Automaten postuliert und überprüft, ob sie tatsächlich die oben stehenden Eigenschaften erfüllt. Ein Algorithmus hierfür kann in [5, Abschnitt 7.2.5] nachgelesen werden; im Beispiel sieht man das leicht, oder kann es durch Ausprobieren leicht herausfinden.

Beispiel 2.3 (Minimaler Deterministischer Automat für ganzzahlige Literale). Der deterministische Automat für Zahlen aus Beispiel 2.2 ist minimal. Zwar haben seine Zustände äquivalente Übergänge, aber nur einer von beiden ist ein Endzustand.

Im deterministischen Automaten für Bezeichner aus Beispiel 2.2 gilt $\{1, 2, 3\} \equiv \{1, 3, 4\}$. Also sieht der minimale Automat so aus:



2.4.4 Die Definition des Scanners

Bisher haben wir einzelne Lexeme mit regulären Ausdrücken und regulären Definitionen beschrieben. Der *Scanner* eines Übersetzers muss aber *alle* Lexeme der Quellsprache erkennen, und zwar eine Sequnz dieser Lexeme.

Wenn die einzzelnen Lexeme der Quellsprache mit regulären Ausdrücken $R_1, \ldots, R_n$ beschrieben werden können, muss also insgesant der reguläre Ausdruck

$$(R_1 \mid \cdots \mid R_n)^*$$

erkannt werden. Durch das Iterieren von Lexemen wird die Zerlegung einer Zeichenkette leicht *ehrdeutig*: Die Ziffernfolge 12345 kann als eine Folge von bis zu 5 Zahlen aufgefasst werden.

Diese sehr einfache Mehrdeutigkeit wird durch die Strategie des *longest match* aufgelöst: die Automatren für reguläre Ausdrücke versuchen immer so lange zu lesen wie möglich. Demnach würde also 12345 als eine Zahl erkannt.

Manchmal führt die Strategie des *longest match* aber auch in die Irre, wie folgendes Beispiel aud PASCAL zeigt.

Beispiel 2.4 (Intervalle und Zahlen in PASCAL*).* In *Padscal* wird mit 3..14 ein Intervall von ganzen Zahlen bezeichnet. Gebrochene zahlen können in Dezimalschreibweise geschrieben werden, z.B. 3.14.

Arbeitet ein Automat blind mit *longest match*, würde er bei der Erkennung von 3..14 nach dem Lesen des ersten Punktes einen Fehler melden, weil danach kein Nenner folgt. Nun könnte man fordern, dass Intervalle so geaschrieben wereden: 3..14 und wäre das Problem los, weil das Leerzeichen das Ende der Zahl eindeutig kennzeichnet.

Ein besserer Ausweg – ohne Änderung der Quellsprache – besteht darin, in diesem Fall *longest match* nicht blind zu machen, sondern nur, wenn es angebracht ist. Dazu führen wir einen weiteren regulären Operator ein: R_1/R_2 erkennt den regulären Ausdruck R_2 nur dann, wenn darauf der reguläre Ausdruck R_1 folgt. Im Beispiel würde man den regulären Ausdruck für Zahlen so definieren:

$$digit^+(./digit)\,digit^+$$

Nun wird der Punkt nur dann gelesen, wenn auf ihn eine Ziffer folgt, und damit klar ist, dass die Zahl weitergeht.

Der Vorschauoperator R_1/R_2 verändert *nicht* das von einem einzelnen regulären Ausdruck beschriebene Lexem; er verändert nur die von dem iterierten regulären Ausdruck beschriebene *Lexem-Sequenz*, bzw. die vom entsprechenden Automaten erkannte Sequenz von Lexemen.

...

2.4.5 Problematische und pathologische Lexemdefinitionen

...

2.4.6 Programmierung deterministischer endlicher Automaten

- Tabellen
- `goto`-Programme
- Daran denken, dass `lex` das Lesen aller Zeichen seit dem letzten Endzustand rückgängig macht.

...

2.5 Scanner

...

2.6 Sieben

...

2.7 Scanner erzeugen mit `lex`

...