Proseminar Theoretische Informatik 03.11.2015

Anwendungen regulärer Sprachen und endlicher Automaten

Aiko Pipo *Wolfgang Mulzer*

1 Formale Sprachen

Die Aufgabe Probleme und Sachverhalte in einer präzisen und eindeutigen Sprache darzustellen, motiviert die Definition einer abstrakten Struktur, welche gewissen Regeln unterliegt und als mathematisches Objekt untersucht werden kann.

Definition 1. *Sei Σ eine Menge von eindeutigen Symbolen (=Buchstaben), Σ^* die Kleenesche Hülle von Σ und $L \subseteq \Sigma^*$. Wir nennen dann Σ Alphabet und L eine formale Sprache über Σ.*

1.1 Formale Grammatik und Chomsky-Hierarchie

Um verschiedene Sprachen nach wesentlichen Eigenschaften zu sortieren, empfiehlt sich das Einführen von Bildungsregeln (Grammatiken), welche zusammen mit einem Alphabet eine bestimmte Sprachklasse induzieren.

Definition 2. *Sei V eine endliche Menge, $\Sigma \subset V$ ein Alphabet sowie $P \subset (V^* \setminus \Sigma^*) \times V^*$ eine Menge von Produktionsregeln und $S \in V \setminus \Sigma$ ein Startsymbol. Dann nennen wir das Tupel $G \coloneqq (V, \Sigma, P, S)$ eine formale Grammatik.*

Die Chomsky-Hierarchie ordnet die Menge dieser Grammatiken in einer Hierarchie, sodass für alle Grammatiken innerhalb einer Klasse gewisse Eigenschaften äquivalent sind.

1.2 Reguläre Sprachen

Die reguläre Grammatik (Typ-3-Grammtik) wird in der Chomsky-Hierarchie folgendermaßen beschrieben:

Definition 3. *Sei $G = (V, \Sigma, P, S)$ eine Grammatik, $N = V \setminus \Sigma$ die Menge der Nichtterminale und ϵ das leere Wort. G ist regulär, falls genau eine der beiden folgenden Voraussetzungen gilt:*

1. $(linksregulär)$ $\forall (w_1 \to w_2) \in P : (w_1 \in N) \wedge (w_2 \in \Sigma \cup N\Sigma)$
2. $(rechtsregulär)$ $\forall (w_1 \to w_2) \in P : (w_1 \in N) \wedge (w_2 \in \Sigma \cup \Sigma N \cup \{\epsilon\})$

Reguläre Sprachen sind demnach genau jene, welche durch reguläre Grammatiken erzeugt werden. Außerdem können reguläre Sprachen auch durch reguläre Ausdrücke erzeugt werden, Dazu mehr in den Anwendungen.

2 Endliche Automaten

Ein Automat ist ein abstraktes theoretisches Modell, welches die Arbeit eines Rechners mithilfe von Zuständen und Übergangskriterien simuliert. Ein Spezialfall des Automaten ist der *endliche Automat* , welcher nur eine endliche Menge von Zuständen kennt. Man nennt einen Automaten *deterministisch*, falls der Zustandsübergang durch den Ausgangszustand und die Eingabe eindeutig festgelegt ist.

2.1 Deterministischer Endlicher Automat (DEA *oder engl. DFA*)

Definition 4. *Sei Q eine endliche Menge von Zuständen, Σ ein Alphabet, $\delta : Q \times \Sigma \to Q$ eine Übergangsfunktion, $q_0 \in Q$ der Startzustand, und $F \subseteq Q$ die Menge der Endzustände. Dann beschreibt das Tupel $M \coloneqq (Q, \Sigma, \delta, q_0, F)$ einen deterministischen endlichen Automaten.*

Die Menge aller Wörter die vom einem DEA akzeptiert werden (Eingabe führt zu einem Endzustand) nennt man die *Sprache des Automaten.*

Definition 5. *Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein DEA. Die Menge $L(M) \coloneqq \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$ ist dann die Sprache von M, wobei $\delta^* : Q \times \Sigma^* \to Q$ folgendermaßen definiert wird:*

1. $\delta^*(q, \epsilon) = q$
2. $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$, *für* $a \in \Sigma, w \in \Sigma^*$

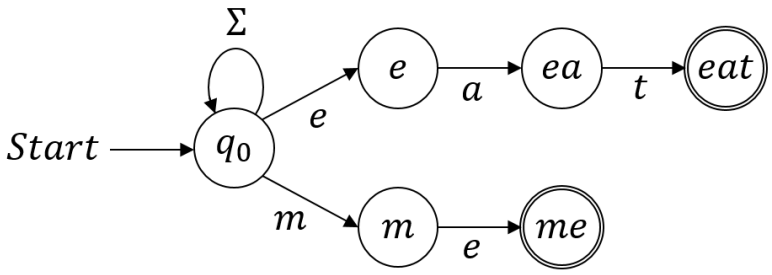
Im Gegensatz zu den DEA's gibt es auch nichtdeterministische EA (NEA), bei denen ein Folgezustand nach einem Übergang nicht eindeutig definiert ist. Ein NEA hat im Gegensatz zu einem DEA anstatt der Übergangsfunktion δ eine *Übergangsrelation* Δ, welche verschiedene Folgezustände erlaubt. Mithilfe der sogenannten Potenzmengenkonstruktion ist es möglich einen NEA in einen äquivalenten DEA umzuwandeln.

3 Anwendungen

Mit diesem Zusammenhang sind wir in der Lage Problemstellungen zu lösen, welche durch reguläre Sprachen zu beschreiben und von (deterministischen) endlichen Automaten zu lösen sind. In der praktischen Informatik gibt es eine Vielzahl solcher Anwendungen, von denen einige hier vorgestellt werden.

3.1 Textsuche

Ein häufiges Problem: In einer großen Menge von Wörtern werden bestimmte Zeichenketten - meist *Schlüsselwörter* genannt - gesucht. Für die Lösung können wir uns zunächst einen 'einfachen' NEA konstruieren und daraus einen DEA ableiten, dessen Arbeitsweise dann implementiert werden kann. Wir suchen also einen NEA A, dessen Sprache $L(A)$ nur Schlüsselwörter enthält. Das bedeutet, dass nur bei einer Eingabe eines Schlüsselwortes ein Endzustand erreicht werden kann.
Wir wählen zum Beispiel die Schlüsselwörter *eat* und *me*. Der zugehörige NEA sieht dann wie folgt aus:



Mit folgender rekursiver Definition lassen sich die Zustände des äquivalenten DEA berechnen:

1. $Q_0 = \{\{q_0\}\}$
2. $Q_{i+1} = Q_i \cup \{\delta(S, a) \mid S \in Q_i, a \in \Sigma\}$, wobei $\delta =$ Übergangsfunktion des DEA
3. $\delta(S, a) = \bigcup_{q \in S} \Delta(q, a)$, wobei $\Delta =$ Übergangsrelation des NEA

Wir erhalten einen DEA, welcher nun leicht zu implementieren ist:

$$\begin{aligned} Q_0 &= \{\{q_0\}\} \\ Q_1 &= \{\{q_0\}\} \cup \{\{q_0, e\}, \{q_0, m\}\} \\ Q_2 &= \{\{q_0\}, \{q_0, e\}, \{q_0, m\}\} \cup \{\{q_0, ea\}, \{q_0, e, me\}\} \\ Q_3 &= \{\{q_0\}, \{q_0, e\}, \{q_0, m\}\, \{q_0, ea\}, \{q_0, e, me\}\} \cup \{\{q_0, eat\}\} \\ &= \{\{q_0\}, \{q_0, e\}, \{q_0, m\}\, \{q_0, ea\}, \{q_0, e, me\}, \{q_0, eat\}\} \end{aligned}$$

Die Darstellung des DEA ist aber schon bei diesen kurzen Zeichenketten sehr umfangreich, da die Anzahl der Übergänge wesentlich größer ist als beim NEA.

3.2 Textmuster und reguläre Ausdrücke

Das untersuchen von Textmustern und Zeichenketten ist eine zentrale Aufgabe bei der Verwaltung von Dokumenten und Datenbanken. Um individuelle Suchanfragen zu realisieren, empfiehlt es sich, einen endlichen Automaten zu genieren, welcher die gesuchte Klasse von Zeichenketten akzeptiert. Das Ziel besteht also darin, eine möglichst einfache Darstellung einer regulären Sprache zu finden. Dazu werden reguläre Ausdrücke verwendet, welche äquivalent zu einer regulären Sprache sind.
Bei der konkreten Implementierung regulärer Ausdrücke gibt es durchaus Unterschiede. Das verwendete Alphabet kann sich zum Beispiel unterscheiden, oder die Definition eines Zeilenumbruchs ist anders. Im Allgemeinen werden die Konventionen aber in den meisten Implementierungen berücksichtigt.

Beispiel für reguläre Ausdrücke:

- [abc] - entspricht dem (kleinen) Buchstaben a,b oder c
- [ab][cd] - Kombination von 2 (kleinen) Buchstaben: ac, ad, bc oder bd
- [0-9] - Zahl von 0 bis 9: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- [A-Za-z0-9] - beliebiger Buchstabe oder Zahl: 0, T, h, 5, p, R, 7...
- $[a_0 \dots a_n]\{x, y\}$ - mindestens x, maximal y mal ein Element aus $\{a_0, \dots, a_n\}$
  - $[a]+ \equiv [a]\{1,\}$ - $n \geq 1$ mal
  - $[a]^* \equiv [a]\{0,\}$ - $n \geq 0$ mal (beliebig oft)
  - $[a]? \equiv [a]\{0,1\}$ - optionaler Ausdruck

Auf diese Weise lassen sich zum Beispiel Adressen darstellen:

[A-Z][a-z]* [0-9]{1,3}[a-z]?, [0-9]{5} [A-Z][$a - z$]*

Dieser Ausdruck fordert zunächst einen Straßennamen, welcher mit einem Großbuchstaben beginnt, dann eine Hausnummer im Bereich 1-999 mit optionaler Angabe (a,b,c...) gefolgt von einem Komma, eine fünfstellige Postleitzahl und letztendlich einen Stadtnamen. Eine gültige Eingabe, welche vom entsprechenden Automaten akzeptiert werden würde, könnte folgendermaßen aussehen:

Arnimallee 3, 14195 Berlin

Leider hat diese Angabe noch Schwächen. So wird zum Beispiel die gültige Adresse

Edwin-Redslob-Straße 25, 14195 Berlin

nicht akzeptiert, da der Straßenname das Symbol '-' enthält, welches wir mit unserem Ausdruck nicht erlaubt haben. Ein korrekte Angabe des regulären Ausdrucks zu finden ist häufig schwer, manchmal sogar unmöglich oder nicht empfehlenswert/effizient. Allerdings lassen sich durch kleine Korrekturen einzelne Fehler schnell beheben. Die überarbeitete Version hier akzeptiert nun auch korrekterweise die obige Adresse.

[A-Z][-A-Za-z]* [0-9]{1,3}[a-z]?, [0-9]{5} [A-Z][$a - z$]*

3.3 Lexikalische Analyse

Dank modernen Programmierwerkzeugen und Entwicklungsumgebungen, wie z.B. Eclipse, NetBeans und anderen IDE's ist das Entwerfen und Debuggen von Programmen heute einfacher denn je. Quellcode kann einfach in anschaulichen Programmiersprachen entwickelt werden und wird anschließend automatisch überprüft und kompiliert. Doch bis ein Quellcode auf die unterste Ebene der Maschinensprache gebracht ist, werden viele Schritte durchlaufen. Der erste davon ist die *lexikalische Analyse*.

Bei der lexikalischen Analyse wird der eingegebene Quelltext in sogenannten *Token* übersetzt, welche einer regulären Grammatik folgen. Die lexikalische Analyse kann daher mit einem endlichen Automaten bearbeitet werden. Jeder Token beschreibt dabei ein Terminalsymbol der Sprache des *Parsers*.

Beispiele für eine Übersetzung in Token:

Quellcode (Lexem)	**Token**
else	ELSE
if	IF
!=	NEQ
>=	GE
(	LPAREN
[A-Za-z][A-Za-z0-9]*	ID('*Bezeichner*')

Das zeitgleiche Erkennen mehrerer Token kann eindeutig über die Reihenfolge der Ausdrücke geregelt werden. So erhält z.B. kann man Schlüsselwörter wie 'else' vor der Definition für Bezeichner angeben, um sie zu reservieren.
Ein zentrales Problem dieser Herangehensweise ist, dass der Scanner beim Einlesen einzelner Zeichen nicht erkennen kann, ob der Ausdruck bereits beendet ist. Eine mögliche Lösung ist ein *Look-Ahead* Verfahren, welches immer den längst möglichen gültigen Ausdruck einliest. Dabei können jedoch weniger Fehler gefunden werden, da z.B. ungültige Bezeichner falsch zerlegt werden. Dies fällt dann erst später in der Kompilierung auf.

3.4 Weitere Anwendungen

Neben der Textsuche und Zeichenerkennung werden endliche Automaten in vielen weiteren Anwendungsbereichen verwendet. Dazu gehören vor allem digitale Schaltungen und Steuerungen, mit denen das Verhalten von automatisierten Prozessen gesteuert werden kann.

Literatur

[1] Andreas Abel, Ulrich Schöpp: *Compilerbau*. Materialien zum Praktikum

[2] Christoph Kreitz, Jens Otten: *Theoretische Informatik I*. Materialien zur Vorlesung, Universität Potsdam, Sommersemester 2004

[3] Jürgen Giesl: *Formale Systeme, Automaten, Prozesse*. Materialien zur Vorlesung, RWTH Aachen, Sommersemester 2010

[4] Andreas Göbel: *Hilfe für reguläre Ausdrücke*. `http://www.regexe.de/hilfe.jsp`, 2008, Zugriff 01.11.2015 14:57 Uhr

[5] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compiler: Prinzipien, Techniken Werkzeuge.* Pearson Studium, 2. Auflage, 2008

[6] J. E. Hopcroft, R. Motwani, J.D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexität.* Pearson Studium, 3. Auflage, 2011.