Reguläre Ausdrücke

Hans Jürgen Ohlbach

Keywords: Reguläre Ausdrücke, reguläre Sprachen, Automaten, Implementierung in Java, Abschlusseigenschaften

Empfohlene Vorkenntnisse: Funktionen und Relationen, Formale Sprachen, Typ 3-Sprachen

Inhaltsverzeichnis

1 Einführung **2**

2 Reguläre Ausdrücke: Die Basismenge an Operatoren **2**

3 Reguläre Ausdrücke $\mapsto$ Automaten **4**

4 Syntaxerweiterungen **7**

5 Reguläre Ausdrücke in Java **7**

5.1 Carpturing Groups 8

5.2 Syntaxerweiterungen in Java 9

6 Abschlusseigenschaften **11**

1 Einführung

Typ 3-Sprachen können auf verschiedene Weise spezifiziert werden, durch eine Typ 3-Grammatik, durch einen Automaten, und, wie wir jetzt kennen lernen, durch reguläre Ausdrücke. Reguläre Ausdrücke sind Bestandteil vieler Programmiersprachen. Daher sind sie eigentlich die nützlichsten Spezifikationsmechnismen für Typ 3-Sprachen. So wie Typ 3-Grammatiken können auch reguläre Ausdrücke in endliche Automaten übersetzt werden, die man dann zum Parsen von Wörtern benutzen kann.

Die Sprache der Regulären Ausdrücken besteht zunächst aus einer absolut minimalen Anzahl von Operatoren. Mit diesen lassen sich dann neue Operatoren als Abkürzungen definieren. Manche Programmiersprachen haben allerdings Operatoren hinzugefügt, die aus den Typ 3-Sprachen herausführen, aber immer noch effizient geparst werden können.

2 Reguläre Ausdrücke: Die Basismenge an Operatoren

Reguläre Ausdrücke werden über einem Alphabet Σ gebildet. Sie benutzen nur die Elemente von Σ, zusammen mit ein paar Sonderzeichen. Insbesondere werden keine Variablen gebraucht.

Definition 1 (Reguläre Ausdrücke) *Reguläre Ausdrücke über einem Alphabet Σ können auf folgende Weise gebildet werden:*

- $\emptyset$ *ist ein regulärer Ausdruck (die leere Menge)*
- ϵ *ist ein regulärer Ausdruck (der leere String)*
- *jedes* $x \in \Sigma$ *ist ein regulärer Ausdruck*
- *Falls* α *und* β *reguläre Ausdrücke sind, dann sind auch*
  - $\alpha\beta$ *reguläre Ausdrücke (Hintereinanderhängen, Konkatenation)*
  - $(\alpha|\beta)$ *reguläre Ausdrücke (Alternative)*
  - $(\alpha)^*$ *reguläre Ausdrücke (beliebige Wiederholung, incl. 0 mal, von* α*).*

Um die Formeln lesbarer zu machen, lassen wir im Folgenden unnötige Klammerungen weg.

Als erste Beispiele definieren wir Intergerzahlen mit regulären Ausdrücken über dem Alphabet $(0, \ldots, 9, +, -)$:

Integer = $(+|-|\epsilon)(0|1|2|3|4|5|6|7|8|9)^*$

$(+|-|\epsilon)$ startet mit + oder - als Vorzeichen, oder keinem Vorzeichen (ϵ). Danach kommen beliebig viele Ziffern. Konkrete Zahlen, die diesem Muster genügen sind z.B. 1234 oder -456 oder +3. Allerdings wären auch führende Nullen erlaubt, z.B. 0012. Sogar der leere String wäre erlaubt (weil der *-Operator auch 0 Wiederholungen zulässt), oder auch nur das + oder das -.

Man könnte das folgendermaßen verfeinern:

Integer = $(+|-|\epsilon)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$

Jetzt sind keine führenden Nullen erlaubt, da die erste Ziffer zwischen 1 und 9 sein muss. Allerdings ist die Zahl 0 auch ausgeschlossen. Diese müsste man als Alternative hinzufügen:

Integer = $(0|((+|-|\epsilon)(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*))$

Jetzt wäre die 0 erlaubt, und zwar ohne Vorzeichen.

Die regulären Ausdrücke definieren Muster für konkrete Wörter einer Sprache.

Definition 2 (Generierte Sprache) *Ein regulärer Ausdruck γ generiert folgende Sprache $L(\gamma)$:*

- $L(\emptyset) = \emptyset$
- $L(\epsilon) = \epsilon$ *(leerer String)*
- $L(\alpha) = \alpha$ *(falls $\gamma = \alpha \in \Sigma$)*
- $L(\alpha\beta) = L(\alpha)L(\beta)$ *(jedes Wort aus $L(\alpha)$ wird mit jedem Wort aus $L(\beta)$ zusammengehängt)*
- $L((\alpha|\beta)) = L(a) \cup L(b)$
- $L((\alpha)^*) = L(\alpha)^*$ *(alle Worte aus $L(\alpha)$ werden beliebig oft (incl. 0 mal) aneinandergehängt)*

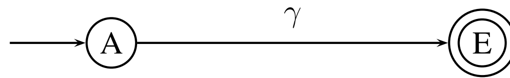
Die Definitionen illustrieren wir mit ein paar Beispielen:
Für $\alpha = a$ ist $L((\alpha)^*) = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$.

Für $\alpha = a$ und $\beta = b$ ist $L(\alpha|\beta) = \{a, b\}$.
$L(\alpha^*\beta^*) = \{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \ldots\}$,
also Wörter bestehend aus beliebig vielen (incl. 0) as gefolgt von beliebig vielen bs.

3 Reguläre Ausdrücke $\mapsto$ Automaten

Ein Regulärer Ausdruck γ kann auf einfache Weise in endliche Automaten übersetzt werden, die dann die Sprache $L(\gamma)$ akzeptiert.

Sei also γ ein regulärer Ausdruck. Wir starten mit einem Automaten:



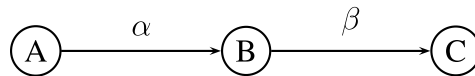
Jetzt werden die verschiedenen Fälle betrachtet, wie γ aufgebaut sein kann.

Fall $\gamma = \emptyset$. In diesem Fall wird der γ-Übergang einfach unterbrochen.

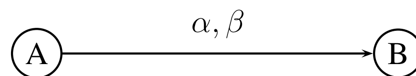
Fall $\gamma = \epsilon$. In diesem Fall wird für γ ein ϵ-Übergang eingeführt.

Fall γ **ist ein Buchstabe aus** Σ. In diesem Fall wird einfach der Buchstabe als Label benutzt.

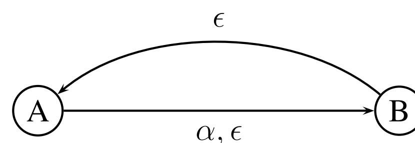
Fall $\gamma = \alpha\beta$ (Konkatenation): Die γ-Kante von A nach C wird aufgespalten in eine α-Kante und eine β-Kante.



Fall $\gamma = (\alpha|\beta)$ (Alternative): Die γ-Kante von A nach B bekommt die zwei Labels α und β.



Fall $\gamma = (\alpha)^*$ (beliebige Wiederholung von α)
Der Automat bekommt einen Zyklus.

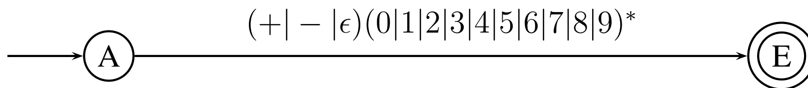


Der ϵ-Übergang von A nach B modelliert, dass α gar nicht wiederholt wird.

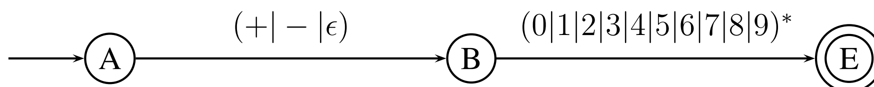
Bei der Umwandlung in den Automaten bleiben die am Anfang eingeführten Start- und Endknoten bis zum Schluss unverändert Start- und Endknoten.

Beispiel:
Wir wandeln den Ausdruck für die einfachen Integer $\gamma = (+|-|\epsilon)(0|1|2|3|4|5|6|7|8|9)^*$ in einen Automaten um.

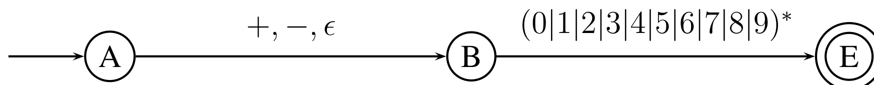
Schritt 1: Start



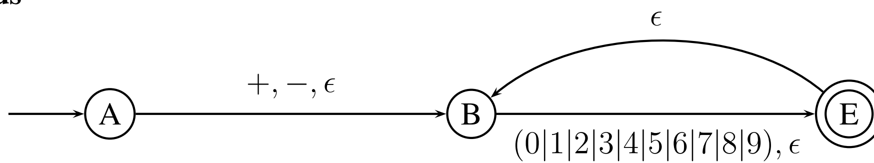
Schritt 2: Konkatenation



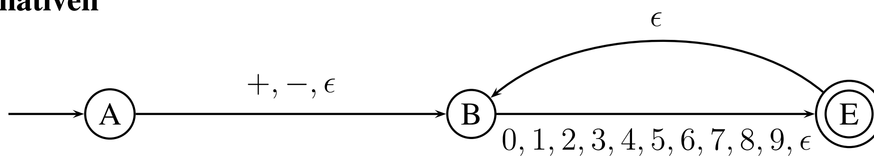
Schritt 3: Alternativen



Schritt 4: Zyklus



Schritt 5: Alternativen

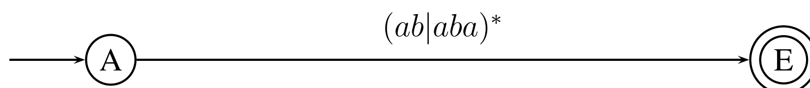


Theorem 1 (Kleene) *Die von einem regulären Ausdruck beschriebenen Sprache sind genau die regulären Sprachen. die von dem generierten Automaten akzeptierte Sprache.*

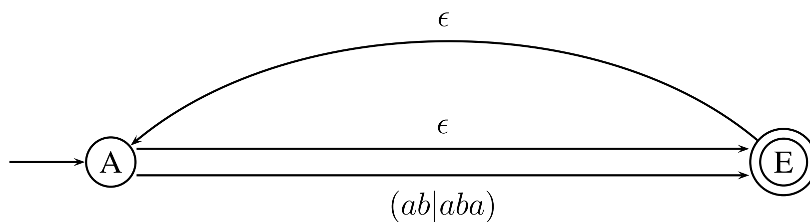
Im Beweis zeigt man durch strukturelle Induktion über den Aufbau der regulären Ausdrücke, dass die Sprache der regulären Ausdrücke genau die Sprache ist, die der daraus generierte endliche Automat akzeptiert. Dann nutzt man die Tatsache, dass die endlichen Automaten genau die regulären Sprachen akzeptieren.

Der aus dem regulären Ausdruck generierte Automat ist leider i.A. weder deterministisch, noch minimal, wie das Beispiel $(ab|aba)^*$ zeigt:

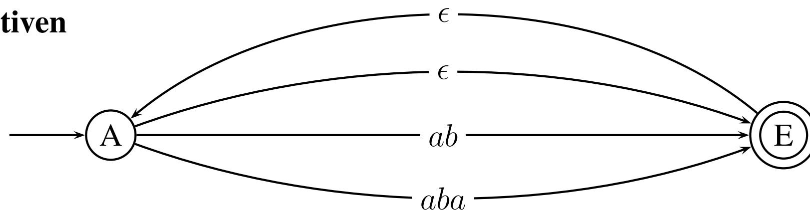
Schritt 1: Start



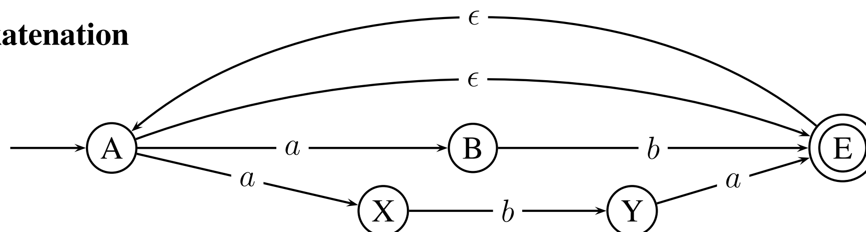
Schritt 2: Zyklus



Schritt 3: Alternativen



Schritt 4: Konkatenation



Der Knoten A ist nichtdeterministisch. Für den Buchstaben a hat er Übergänge nach B und X.

Um die so generierten Automaten zum Parsen von Wörten zu verwenden muss man sie daher noch deterministisch machen und minimieren.

4 Syntaxerweiterungen

Die Basismenge an Operatoren für reguläre Ausdrücke ist oft noch sehr unhandlich. Daher hat man Abkürzungen eingeführt, die zwar keine neuen Ausdrucksmöglichkeiten schaffen, aber die Schreibarbeit erleichtern.

Abkürzung	Expandiert	Bedeutung
X^+	XX^*	beliebig oft, aber mindestens einmal
$X?$	$(\epsilon\|X)$	einmal oder keinmal
$X\{3\}$	XXX	3 mal, allgemein n-mal
$X\{3,\}$	$XXXX^*$	mindestens 3 mal, allgemein mindestens n mal
$X\{3,5\}$	$XXX(\epsilon\|X\|XX)$	3 bis 5 mal, allgemein n bis m mal

Weitere Abkürzungen werden von den jeweiligen Programmiersystemen unterstützt, z.B. steht $[a-k]$ i.A. für $(a|\ldots|k)$.

5 Reguläre Ausdrücke in Java

Reguläre Ausdrücke sind ein extrem nützliches Hilfsmittel für die Textverarbeitung. Daher gibt es sie in fast jeder Programmiersprache, u.A. auch in Java. Die Implementierungen bieten noch weitaus mehr Möglichkeiten, als die Theorie der regulären Ausdrücke enthält. An dieser Stelle soll aber keine umfassende Dokumentation der Java-Version von regulären Ausdrücken vorgestellt werden, sondern eher ein erster Eindruck ihrer Möglichkeiten vermittelt werden.

Ein regulärer Ausdruck ist zunächst ein String, z.B.
```
String regExp = "(ab|aba)+";
```
Mit
```
Pattern p = Pattern.compile(regExp);
```
wird der Ausdruck in einen Automaten übersetzt.

Jetzt kann man den Automaten auf einen Eingabestring anwenden,
```
Matcher m = p.matcher("ababab");
```

Ein Matcher ist zunächst noch nicht das Ergebnis eines Durchlaufs durch den Automaten, sondern er bereitet den Automaten nur vor. Neben der Lösung des Wortproblems (in Java-Sprech matching genannt) kann er noch weitere Operationen. Insgesamt sind es vier Operationen die er kann. Sie werden mit folgenden Methoden aufgerufen:

matches: matcht den ganzen Eingabestring gegen das Pattern. Das ist die Lösung des Wortproblems.
```
System.out.println(m.matches());
```
druckt in diesem Fall `true` aus.

lookingAt: testet, ob ein Anfangsstück des Eingabestrings gegen das Pattern matcht.
```
p.matcher("ababab").lookingAt() ↦ true
p.matcher("abcabab").lookingAt() ↦ true
```

`p.matcher("cababab").lookingAt()` ↦ `false`
Im positiven Fall kann man sich mit den Methoden start() und end(), sowie group() auch den Bereich ausgeben lassen, wo der Match gefunden wurde.

find: sucht den nächsten Teilstring, der gegen das Pattern matcht.
`p.matcher("cababab").find()` ↦ `true`
`p.matcher("cacb").find()` ↦ `false`
Auch hier kann man sich den Bereich ausgeben lassen, wo der Match gefunden wurde.

Ersetzungen: `p.matcher("abcabab").replaceAll("@")` ↦ `@c@`
ersetzt alle Bereiche, die matchen, durch @. Neben replaceAll gibt es noch weitere Ersetzungsmethoden.

Die Java-Implementierung von regulären Ausdrücken hat eine ganze Reihe von Erweiterungen:

5.1 Capturing Groups

Geklammerte Teilstrings in regulären Ausdrücken erlauben den Zugriff auf den Matchenden Teil des Eingabestrings.

```
Pattern p = Pattern.compile("(ab|aba)+(cd)+");
Matcher m = p.matcher("ababcdcd");
m.matches();
System.out.println(m.groupCount());
System.out.println(m.group(1));
System.out.println(m.group(2));
```

druckt nacheinander 2, ab und cd aus.

Die Nummerierung der Gruppen richtet sich nach der Position der öffnenden Klammer im regulären Ausdruck.

Eine echte Erweiterung, die über die formale Definition der regulären Ausdrücke hinausgeht, sind die Referenzen auf die Gruppennummern. Im Ausdruck `(a+)b(\1)` referiert `\1` auf den Inhalt der ersten Klammer. Nach dem b muss also genau das gleiche kommen wie in der ersten Klammer.

```
Pattern p = Pattern.compile("(a+)b(\1)");
Matcher m = p.matcher("aaba");
System.out.println(m.matches());
```

druckt `false` aus, während

```
Pattern p = Pattern.compile("((a+)b(\1)");
Matcher m = p.matcher("aabaa");
System.out.println(m.matches());
```

`true` ausdruckt.

5.2 Syntaxerweiterungen in Java

Eine umfassende Liste der Syntaxerweiterungen von regulären Ausdrücken findet man in der Dokumentation der Pattern-Klasse. Wir illustrieren sie an ein paar Beispielen:

Characters: Einzelne Unicode-Buchstaben können mit Oktal- oder Hexadezimalzahlen spezifiziert werden. `\u00B5` steht z.B. für den griechischen Buchstaben μ. Ganz bestimmte Zeichen haben spezielle Namen, z.B. \r steht für den carriage-return.

Character Klassen: Ganze Gruppen von Zeichen können u.a. folgendermaßen definiert werden:
Der Punkt . steht für ein beliebiges Zeichen.
`[abc]` steht für `(a|b|c)`
`[^abc]` steht für alle Zeichen außer, a,b oder c
`[a-zB-X]` steht für Buchstaben von a bis z oder B bis X
`[a-z&&[^m-p]]` steht für Buchstaben a bis l, oder q-z.

Wiederum gibt es einige vordefinierte Klassen, z.B. \d steht für die Ziffern 0 - 9. \s steht für alle Leerzeichen (Whitespace) usw.

Begrenzungszeichen: Die Methoden `lookingAt` und `find` suchen matchende Teilstrings. Für diese Methoden gibt es Sonderzeichen, die genauer kontrollieren lassen, wo diese Teilstrings zu finden sind.

Eines davon ist das Zeichen ^. Es steht für den Zeilenbeginn
Beispiel:

```
String ls = System.getProperty("line.separator");
Pattern p = Pattern.compile("^a+b",Pattern.MULTILINE);
Matcher m = p.matcher("c"+ls+"aabc");
System.out.println(m.find());
System.out.println(m.group());
```

druckt `true` und `aab` aus, weil `aab` genau an einer neuen Zeile anfängt.

Bereiche: Ein regulärer Ausdruck kann auf verschiedene Stellen eines Strings passen. Mit folgenden Konstrukten hat man etwas Einfluss auf die Art der Stellen, auf die er passt. Das betrifft insbesondere die Wiederholungsoperatoren ?,*,+,{...}. Es gibt dabei drei Varianten:

greedy: so wirken die normalen Operatoren.

Beispiel:

```
Pattern p = Pattern.compile(".*b");
Matcher m = p.matcher("abaaaab");
System.out.println(m.find());
System.out.println(m.group());
```

druckt `true` und `abaaaab` aus. Der Operator .* matcht soviel Buchstaben wie möglich (greedy). Das wäre zunächst der komplette String. Da aber das b noch gebraucht wird, backtrackt er, um dem Rest des Patterns noch etwas übrig zu lassen.

reluctant:

Beispiel:

```
Pattern p = Pattern.compile(".*?b");
Matcher m = p.matcher("abaaaab");
System.out.println(m.find());
System.out.println(m.group());
```

druckt `true` und `ab` aus. Der Operator .*? matcht so wenig Buchstaben wie möglich (reluctant). Ihm reicht das a. Danach kommt das erste b.

possessive:

Beispiel:

```
Pattern p = Pattern.compile(".*+b");
Matcher m = p.matcher("abaaaab");
System.out.println(m.find());
```

druckt `false` aus. Der Operator .*+ matcht so viele Buchstaben wie möglich. Im Gegensatz zu greedy gibt er aber nichts mehr her (kein Backtracking). Er matcht daher den kompletten String, so dass für das b nichts mehr übrig bleibt.

Die weiteren Möglichkeiten, die die Klassen Pattern und Matcher bieten, sollte man in der Originaldokumentation nachlesen.

6 Abschlusseigenschaften

Mit Hilfe des Satzes von Kleene kann man jetzt eine ganze Reihe von Abschlusseigenschaften herleiten. Abschlusseigenschaften sagen aus, dass gewisse Operationen Objekte einer Klasse wieder in Objekte derselben Klasse transformieren, z.B. reguläre Sprachen in reguläre Sprachen.

Im folgenden Theorem nutzen wir aus, dass es zu jeder regulären Sprache einen regulärer Ausdruck gibt, der genau diese Sprache beschreibt.

Theorem 2 (Abschlusseigenschaften regulärer Sprachen)
Die regulären Sprachen sind abgeschlossen unter:

Vereinigung: *Sein α und β reguläre Ausdrücke für zwei reguläre Sprachen.*
Dann ist $L(\alpha) \cup L(\beta) = L(\alpha|\beta)$ (Definition).
$(\alpha|\beta)$ ist wieder ein regulärer Ausdruck und daher ist $L(\alpha|\beta)$ eine reguläre Sprache.

Komplement: *Um zu zeigen, dass das Komplement L' der Sprache L wieder regulär ist, nimmt man den endlichen Automaten für die Sprache L, vertauscht End- mit Nicht-Endzuständen, und erhält wieder einen endlichen Automaten, diesmal für L'.*

Schnitt: *Für den Schnitt $L_1 \cap L_2$ nutzt man den rein mengentheoretischen Zusammenhang: $L_1 \cap L_2 = (L_1' \cup L_2')'$. Da Komplement und Vereinigung abgeschlossen sind, ist auch der Schnitt abgeschlossen.*

Produkt (Konkatenation): *Sein α und β reguläre Ausdrücke für zwei reguläre Sprachen.*
Dann ist $L(\alpha)L(\beta) = L(\alpha\beta)$ (Definition).
$\alpha\beta$ ist wieder ein regulärer Ausdruck und daher ist $L(\alpha\beta)$ eine reguläre Sprache.

Stern (beliebige Wiederholung): *Sei α ein reguläre Ausdrücke für eine reguläre Sprache.*
Dann ist $L(\alpha)^* = L(\alpha^*)$ (Definition).
α^* ist wieder ein regulärer Ausdruck und daher ist $L(\alpha^*)$ eine reguläre Sprache.