
Regex-Methoden aus der Praxis

Sie kennen jetzt die Grundlagen zum Schreiben von regulären Ausdrücken. Jetzt möchte ich dies anwenden und mit weiterführenden Methoden komplexere Ausdrücke konstruieren. Bei jeder Regex gibt es eine delikate Balance zwischen dem Finden von erwünschten Treffern und dem Ablehnen dessen, was nicht passen soll. Sie haben bereits einige Beispiele kennengelernt, bei denen wir das gierige Verhalten zum eigenen Nutzen anwenden konnten, wir haben aber auch einige mögliche Fehlerquellen dabei gesehen; solche Beispiele gibt es in diesem Kapitel wieder.

Bei einer NFA-Maschine gehört auch die Effizienz der Mustersuche zu dieser Balance – das ist vor allem im nächsten Kapitel ein Thema. Eine schlecht konzipierte Regex – auch eine, die an sich korrekt ist – kann eine Mustersuche enorm langsam werden lassen.

Dieses Kapitel besteht vornehmlich aus Beispielen, die generelle Überlegungen bei der Problemlösung veranschaulichen sollen. Sie sind deshalb auch dann nützlich, wenn das behandelte Problem für Ihre aktuelle Arbeit (noch) nicht von Belang ist.

Auch wenn Sie z. B. nichts mit HTML zu tun haben, können die HTML-Beispiele dennoch hilfreich sein. Das Schreiben von guten regulären Ausdrücken ist nicht nur eine Fertigkeit, es ist eine kleine Kunstform. Um in dieser Kunst zur Meisterschaft zu gelangen, reicht es nicht, ein paar Regeln auswendig zu lernen, dazu bedarf es längerer Übung. Die Beispiele sind Illustrationen zu den Erfahrungen, die ich über die Jahre mit regulären Ausdrücken gemacht habe.

Sie brauchen Ihre eigenen Erfahrungen, um das Gelernte zu verstehen und zu absorbieren, und ich hoffe, daß die Beschäftigung mit meinen Beispielen dazu beiträgt.

Die ausgewogene Regex

Um gute reguläre Ausdrücke zu schreiben, bedarf es einer Ausgewogenheit zwischen verschiedenen und teilweise antagonistischen Zielen. Ein guter regulärer Ausdruck muß folgende Kriterien erfüllen:

- Die Regex soll genau das finden, was Sie suchen, und nichts anderes.
- Die Regex soll verständlich und leicht zu warten sein.
- Bei einem NFA soll die Regex effizient sein (schnell zu einem Treffer bzw. zu einem Fehlschlag führen).

Diese Anforderungen sind oft von der Umgebung abhängig. Wenn ich auf der Eingabezeile nur schnell etwas »greppen« will, um mir einen Überblick zu verschaffen, ist es mir meist egal, ob eine Zeile zuviel ausgegeben wird; also verwende ich kaum Zeit darauf, *exakt* die richtige Regex zu formulieren. Ich erlaube mir Nachlässigkeiten, um Zeit zu sparen, und weil ich mir die ausgegebenen Zeilen sowieso selbst ansehe. Wenn ich dagegen an einem wichtigen Skript arbeite, ist die Zeit, eine Regex genau richtig zu formulieren, gut investiert: Wenn das Problem eine komplizierte Regex erfordert, dann muß sie auch so geschrieben werden. Man muß lernen, diese verschiedenen Aspekte unter einen Hut zu bringen.

Sogar in einem Programm sind Effizienz-Überlegungen von der Umgebung abhängig. Eine lange Alternation zum Prüfen von Argumenten von der Befehlszeile wie zum Beispiel `^(display|geometry|cemap|...|quick24|random|raw)$` ist für einen NFA wenig erfreulich, aber da es nur um Argumente geht (etwas, das nur einmal in einem Programm ausgeführt wird), spielt es keine Rolle, ob es hundertmal langsamer abläuft als die schnellstmögliche Version. Wenn der Ausdruck dagegen auf jede Zeile einer möglicherweise großen Datei angewandt wird, dann wirkt sich die Ineffizienz hier auf das ganze Programm aus.

Einige kleine Beispiele

Fortsetzungszeilen

Im letzten Kapitel hatten Sie beim Fortsetzungszeilen-Beispiel von Seite 181 gesehen, daß ein traditioneller NFA mit `^\w+=.*(\\n.*)*` und dem Text

```
SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c \
missing.c msg.c node.c re.c version.c
```

nicht beide Zeilen findet. Das liegt daran, daß das erste `.*` alles bis hinter den ersten Backslash erkennt und diesen vom eigentlich dafür vorgesehenen `(\\n.*)*` verbirgt. Wenn wir nur bis zum Backslash vorgehen wollen, müssen wir das in der Regex auch sagen: Wir können dazu den Punkt durch `[^\\n\\]` ersetzen. (Sie bemerken, daß in der negierten Zeichenklasse auch `\n` auftritt. Wir hatten angenommen, daß beim verwendeten Dialekt der Punkt nicht auf das Newline paßt, das verlangen wir auch von dem Unterausdruck, der den Punkt ersetzt; ↪ 121.)

Wir erhalten so:

```
^\\w+=([\\n\\])*([\\n([\\n\\])*]*)*
```

Das funktioniert mit den Beispielzeilen aus dem Makefile, aber mit dieser Änderung haben wir uns unter Umständen ein neues Problem eingehandelt: Backslashes sind nun nur am Ende einer Zeile erlaubt. Wenn in den Daten mitten in einer Zeile ein Backslash vorkommt, funktioniert unsere Regex nicht mehr richtig; wir fordern aber, daß auch dieser Fall richtig behandelt wird.

Bis jetzt sind wir immer nach dem Rezept vorgegangen: »Finde eine Zeile, und wenn eine Fortsetzungszeile folgt, auch diese.« Probieren wir jetzt einen ganz anderen Ansatz aus, den ich oft schon mit Erfolg verwendet habe. Wir konzentrieren uns an jedem Punkt auf die Zeichen, die erlaubt sein sollen. Auf der Zeile selbst sollen »normale« Zeichen erlaubt sein (weder Newline noch Backslash) oder aber eine Kombination von Backslash und irgend etwas. Wenn wir dafür `^\\.` nehmen und der »Punkt-paßt-auf-alles«-Modus eingeschaltet ist, dann paßt das auch auf die Kombination von Backslash und Newline.

Der Ausdruck wird so zu `^\\w+=([\\n\\]|\\.)*` – unter der Voraussetzung, daß der Punkt wirklich alle Zeichen erkennt. Je nach dem Verwendungszweck muß wegen des `^` am Anfang der Regex auch der Mehrzeilenmodus aktiviert sein (☞ 114).

Wir sind dennoch noch nicht ganz fertig mit diesem Beispiel – im nächsten Kapitel werden wir dazu Überlegungen zur Effizienz anstellen (☞ 276).

Eine IP-Adresse erkennen

Ein weiteres Beispiel, das wir jetzt wesentlich ausführlicher behandeln, ist das Erkennen von IP-(Internet Protocol-)Adressen: vier durch Punkte getrennte Zahlen, wie etwa 1.2.3.4. Oft werden diese Zahlen wie bei 001.002.003.004 mit führenden Nullen angegeben. Ein simpler Test wäre `[0-9]*\\. [0-9]*\\. [0-9]*\\. [0-9]*`, aber das ist so ungenau, daß sogar »und dann...?« erkannt wird – die Regex *erzwingt* keine einzige Ziffer; die einzige Forderung sind drei Punkte, mit Ziffern dazwischen *oder auch nicht*.

Um das zu verbessern, ersetzen wir zunächst die Sterne durch Pluszeichen, weil jede Zahl aus mindestens einer Ziffer bestehen muß. Damit der String außer der IP-Adresse nichts anderes enthalten darf, umschließen wir den Ausdruck mit `^...$`. Das ergibt:

```
^[0-9]+\\. [0-9]+\\. [0-9]+\\. [0-9]+$
```

Wenn wir statt der Zeichenklasse `[0-9]` die Abkürzung `\\d` verwenden, erhalten wir das etwas besser lesbare¹ `^\\d+\\. \\d+\\. \\d+\\. \\d+$`, aber das läßt noch immer Dinge zu, die keine IP-Adressen sind, wie etwa »1234.5678.9101112.131415« (bei IP-Adressen muß jede Zahl im Bereich 0 – 255 liegen). Dreistellige Zahlen ließen sich mit `^\\d\\d\\d\\. \\d\\d\\d\\. \\d\\d\\d\\. \\d\\d\\d$`

¹ Oder auch nicht – das hängt davon ab, woran man sich gewöhnt hat. In einer komplizierten Regex finde ich `\\d` leichter lesbar als `[0-9]`, aber auf bestimmten Systemen sind die zwei nicht exakt äquivalent. Auf Unicode-fähigen Systemen kann `\\d` auch auf Zifferzeichen außerhalb des ASCII-Codes passen (☞ 122).

erzwingen, aber das ist des Guten zuviel, weil nun ein- und zweistellige Zahlen (wie bei 1.2.3.4) nicht mehr passen. Wenn der verwendete Regex-Dialekt die $\{min,max\}$ -Notation kennt, kann man $^{\wedge}\{1,3\}\.\{1,3\}\.\{1,3\}\.\{1,3\}\$$ benutzen, sonst gibt es immer die Möglichkeit von $\{1,3\}$ oder $\{1,3\}$ für jeden Teil. Jede Regex erlaubt eine bis drei Ziffern, aber auf verschiedenen Wegen.

Je nach Anforderung genügt eine der verschiedenen Genauigkeitsstufen. Wer sehr genau sein will, muß sich darum kümmern, daß $\{1,3\}$ auch auf 999 paßt, was viel größer als 255 und damit als Komponente einer IP-Adresse nicht zulässig ist.

Es gibt verschiedene Möglichkeiten, die Zahl auf den Bereich von 0 bis 255 einzuschränken. Ein dummer Ansatz wäre $\{0|1|2|3|\dots|253|254|255\}$, der nicht einmal korrekt ist, weil die Zahlen mit führenden Nullen nicht erwischt werden; also bräuchte man $\{0|00|000|1|01|001|\dots\}$, was vollends lächerlich wäre. Für einen DFA ist es nur darum lächerlich, weil der Ausdruck sehr lang ist – aber er ist genauso effizient wie ein anderer Ausdruck, der dasselbe beschreibt. Bei einem NFA dagegen machen die vielen Alternativen den Ausdruck sehr ineffizient.

Ein realistischerer Ansatz konzentriert sich darauf, welche Ziffern an welcher Stelle zugelassen sind. Bei ein- und zweistelligen Zahlen spielt der Bereich keine Rolle, sie können also mit $\{1,3\}$ abgedeckt werden. Auch dreistellige Zahlen, die mit 0 oder 1 beginnen, sind problemlos – das entspricht Zahlen im erlaubten Bereich von 000 – 199. Es kommt also $\{01\}$ dazu, und wir erhalten die Regex $\{1,3\}\{01\}$. Dieser Ausdruck ruft vielleicht Erinnerungen an das Uhrzeit-Beispiel aus Kapitel 1 (☞ 28) oder an das Beispiel mit Kalenderdaten aus dem letzten Kapitel (☞ 180) wach.

Wenn eine dreistellige Zahl mit einer 2 beginnt, muß die Zahl kleiner als 255 sein, also sind an der zweiten Stelle alle Ziffern kleiner 5 erlaubt. Wenn die zweite Ziffer eine 5 ist, muß die dritte kleiner als 6 sein. Das kann mit $\{2[0-4]\{25[0-5]\}$ ausgedrückt werden.

Das mag zunächst verwirren, aber dieser Ansatz hat durchaus seine Logik. Das Resultat ist $\{1,3\}\{01\}\{2[0-4]\{25[0-5]\}$. Die ersten drei Alternativen können wir sogar zu einer verschmelzen und erhalten $\{01\}\{2[0-4]\{25[0-5]\}$. Dies ist bei einem NFA effizienter, weil jede nicht passende Alternative ein Backtracking auslöst. Wenn wir in der ersten Alternative $\{1,3\}$ statt $\{1,3\}$ benutzen, hat ein NFA ein kleines bißchen weniger zu tun, wenn gar keine Ziffer im String auftaucht. Die Analyse überlasse ich Ihnen – ein kleines Testbeispiel sollte den Unterschied klarmachen. In diesem Teil des regulären Ausdrucks gibt es noch mehr Optimierungsmöglichkeiten, aber diese verschiebe ich auf das nächste Kapitel.

Nun haben wir einen Unterausdruck, der auf eine einzelne Zahl zwischen 0 und 255 paßt. Diesen können wir in Klammern einpacken und anstelle jedes $\{1,3\}$ in die frühere Regex einsetzen. Das ergibt (auf zwei Zeilen aufgeteilt):

```
^([01]?[2[0-4]{25[0-5]})\.( [01]?[2[0-4]{25[0-5]})\
([01]?[2[0-4]{25[0-5]})\.( [01]?[2[0-4]{25[0-5]})$
```

Ein dicker Brocken! War das die Mühe wert? Das müssen Sie aufgrund der gestellten Anforderungen entscheiden. Der Ausdruck erlaubt nur noch syntaktisch korrekte IP-Adressen, aber darunter sind doch noch welche, die *semantisch* falsch sind, zum Beispiel 0.0.0.0 (es dürfen nicht alle Bytes null sein). Mit einem Lookahead-Konstrukt (☞ 136) könnte man sogar diesen Sonderfall ausschließen und nach dem `^(?!0+\.0+\.0+\.0+)$` einfügen, aber irgendwann müssen Sie sich entscheiden, ob Sie noch spezifischer sein wollen – das Kosten-Nutzen-Verhältnis stimmt dann vielleicht nicht mehr. Manchmal ist es einfacher, der Regex Arbeit abzunehmen. Zum Beispiel könnte man auf die einfachere und klare Regex `^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$` zurückgreifen und jede Komponente einklammern, so daß die gefundenen Zahlen in \$1, \$2, \$3 und \$4 verfügbar werden. Diese können dann mit anderen Mitteln der Programmiersprache auf den erlaubten Bereich geprüft werden.

Die Umgebung richtig einschätzen

Die zwei Anker bei dem obigen Ausdruck sind wichtig, und es ist auch wichtig, das zu verstehen. Ohne sie würde der Ausdruck auch auf `ip=72123.3.21.993` passen; wenn ein traditioneller NFA verwendet wird, sogar auf `ip=123.3.21.223`.

In diesem zweiten Fall wird nicht einmal die ganze letzte Zahl 223 erkannt, obwohl die Regex das zuläßt. Nun, es ist *erlaubt*, aber es gibt in dem Ausdruck nichts (wie einen abschließenden Punkt oder einen Anker), was drei Ziffern *erzwingt*. Die erste Alternative der letzten Gruppe, `[01]?d\d?`, findet die ersten zwei Ziffern und ist dann am Ende der Regex angelangt. Wie beim Kalenderdaten-Problem auf Seite 179 können wir aber die Alternativen so umordnen, daß sich der gewünschte Effekt einstellt. Hier bedeutet das, die Alternative für drei Ziffern als erste aufzuführen. Damit werden alle zulässigen dreistelligen Zahlen gefunden, bevor die Maschine zur nächsten Alternative geht. (Bei einem DFA oder einem POSIX-NFA ist ein solches Umordnen natürlich unnötig und auch sinnlos, weil diese ohnehin die Alternative mit dem längsten Treffer auswählen.)

Umgeordnet oder nicht – der erste falsche Treffer bleibt ein Problem. Vielleicht denken Sie jetzt: »Ah! Wortgrenzen-Anker lösen das Problem«. Leider nicht, denn so eine Regex würde noch immer auf Texte wie `1.2.3.4.5.6` passen. Um solche Treffer mittendrin auszuschließen, müssen Sie sicherstellen, daß links und rechts in der Umgebung weder alphanumerische Zeichen noch Punkte vorkommen. Wenn Lookaround unterstützt ist, können Sie die gesamte Regex in `(?<![\w.])…(?![\w.])` verpacken und so fordern, daß die Zeichen vor bzw. nach dem gewünschten Treffer nicht auf `[\w.]` passen dürfen. Ohne Lookaround kann je nach Situation auch `^(^|•)…(•|$)` genügen.

Umgang mit Dateinamen

Beim Umgang mit Dateinamen ergeben sich viele gute Beispiele für reguläre Ausdrücke, sei es mit Pfadnamen unter Unix wie `/usr/local/bin/perl` oder mit Windows-Dateinamen wie `\Program Files\Yahoo!\Messenger`. Ausprobieren ist kurzweiliger als Lesen, deshalb baue ich diese Ausdrücke in Programmbeispiele in Perl, Java und VB.NET ein. Wenn Sie an einer bestimmten Sprache nicht interessiert sind, lassen Sie die Programmlistings aus – es sind die regulären Ausdrücke, die hier zählen.

Verzeichnisangabe aus Pfadnamen entfernen

Als erstes Beispiel wollen wir die Verzeichnisangabe eines Pfadnamens entfernen, so daß nur der Dateiname übrigbleibt, also beispielsweise `/usr/local/bin/gcc` in `gcc` verwandeln. Probleme auf eine Art zu formulieren, die eine Lösung schon nahelegt, ist oft schon die halbe Miete. In diesem Fall wollen wir alles bis und mit dem letzten Slash (Backslash bei Windows) entfernen. Wenn kein Slash vorhanden ist, ist das auch gut; dann brauchen wir gar nichts zu tun. Ich habe mehrfach vor zu leichtfertigem Gebrauch von `^.*` gewarnt, aber in diesem Fall nutzen wir gerade das gierige Verhalten aus. In der Regex `^.*\/` verbraucht das `^.*` zunächst die ganze Zeile, aber die Maschine geht dann bis zum letzten Slash zurück (Backtracking), damit ein globaler Treffer gefunden wird.

In der folgenden Tabelle enthält die Variable `f` einen Pfadnamen. Dieser wird durch seine letzte Komponente, den Dateinamen, ersetzt. Hier geschah das in unseren drei Beispielsprachen für Unix-Pfadnamen:

Programmiersprache	Code
Perl	<code>\$f =~ s{^.*\/}{};</code>
java.util.regex	<code>f = f.replaceFirst("^.*\/", "");</code>
VB.NET	<code>f = Regex.Replace(f, "^.*\/", "")</code>

Der reguläre Ausdruck (bzw. der String, der als Regex interpretiert wird), ist unterstrichen; die mit der Mustersuche zusammenhängenden Komponenten sind halbfett dargestellt.

Zum Vergleich die Version für Windows-Pfadnamen:

Programmiersprache	Code
Perl	<code>\$f =~ s/^.*\\\/;</code>
java.util.regex	<code>f = f.replaceFirst("^.*\\\\\\", "");</code>
VB.NET	<code>f = Regex.Replace(f, "^.*\\", "")</code>

Die Unterschiede in den einzelnen Sprachen für Unix und für Windows sind schon bemerkenswert, insbesondere der vervierfachte Backslash in Java (☞ 104).

Und was ist nun, wenn die Regex nicht paßt? Wenn der Pfadname in `f` keinen Slash enthält, paßt die Regex nicht, die Substitution wird nicht ausgeführt, und der String bleibt, wie er war. Genau das wollen wir.

Für Effizienz-Betrachtungen muß man untersuchen, wie die Regex-Maschine vorgeht (wenn es sich um einen NFA handelt). Schauen wir, was geschieht, wenn wir den Zirkumflex am Anfang weglassen (das kann leicht passieren) und dann die Regex auf einen String ohne Slash anwenden. Die Maschine beginnt wie immer am String-Anfang. Das `^.*` läuft sofort bis ans Ende des Strings, dann muß jedes einzelne erkannte Zeichen zurückgenommen werden, bis ein Slash gefunden wird. Irgendwann wird durch dieses Backtracking wieder der Ausgangspunkt erreicht, und es wurde noch immer kein Treffer erzielt. Die Maschine entscheidet, daß es kein Matching *vom Anfang des Strings aus* gibt. Damit ist die Suche aber noch nicht beendet.

Das »Getriebe« geht zum nächsten Zeichen im String und setzt die Regex-Maschine darauf an. Dies muß (theoretisch) für jede Position im String wiederholt werden. Dateinamen sind meist kurz, aber bei anderen ähnlichen Fällen kann der Text sehr lang sein und erzeugt damit eine große Zahl von Backtracking-Operationen. (Bei einem DFA tritt dieses Problem natürlich nicht auf.) In der Praxis »merkt« ein gutes Getriebe, daß eine Regex nie passen kann, wenn sie mit `^.*` beginnt und schon am Anfang des Strings nicht paßt. Es spart sich die Verschiebungen zu den weiteren Positionen im String und gibt auf (☞ 251). Trotzdem ist es klüger, den Zeilenanker explizit hinzuschreiben, wie wir das getan haben.

Dateinamen aus einem Pfadnamen herauslösen

Wir können auch ganz anders vorgehen: Wir suchen das letzte Element – den Dateinamen – im Pfadnamen und speichern den gefundenen String in einer anderen Variablen. Der Dateiname ist »alles am Ende, das nicht ein Slash ist«: `^[^/]*$`. Diesmal ist der Anker nicht nur eine Optimierung; wir brauchen das Dollarzeichen am Ende wirklich. Wir können in Perl etwa schreiben:

```
$Pfad =~ m{([^\/]*)$};      # Variable $Pfad mit Regex testen
$DateiName = $1;          # gefundenen Text abspeichern
```

Es fällt auf, daß ich gar nicht überprüfe, ob das Matching erfolgreich war, weil ich *weiß*, daß die Regex immer paßt. Die einzige *zwingende* Vorschrift in der Regex ist die, daß der Treffer am Ende des Strings enden muß, und sogar der leere String hat ein Ende. Wenn ich also nachher `$1` benutze, um den von den Klammern eingefangenen Wert abzuspeichern, bin ich sicher, daß etwas gefunden wurde – auch wenn es der leere String ist, wenn der Pfadname auf einen Slash endet.

Noch eine Bemerkung zur Effizienz: Bei einem NFA ist ein Ausdruck wie `^[^/]*$` sehr ineffizient. Wenn man die einzelnen Schritte einer NFA-Maschine verfolgt, findet man eine große Anzahl von Backtrackings. Sogar bei einem relativ kurzen Beispiel wie `>/usr/local/bin/perl<` sind bereits über 40 Backtrackings involviert, bis der Treffer gefunden wird. Betrachten wir

einen Matching-Versuch, der bei `...local/...` beginnt. `^[^/]*` paßt auf alles bis zum zweiten `/`, dann wird das `$1` mit dem Slash verglichen (negativ), und für jedes der Zeichen `l`, `a`, `c`, `o`, `l` muß ein gespeicherter Zustand hervorgeholt werden. Nicht genug, der größte Teil wird gleich beim nächsten Versuch, der von `...local/...` ausgeht, wiederholt; dann nochmals bei `...local/...` usw.

Das soll uns aber in diesem Fall keine Sorgen machen, denn Dateinamen sind kurz, und 40 Backtrackings sind wenig – problematisch wird es bei 40 Millionen Backtrackings! Der Gedanke kann aber wichtig sein, wenn mit viel größeren Suchtexten umgegangen wird.

Auch wenn dies ein Buch über reguläre Ausdrücke ist, muß ich an dieser Stelle betonen, daß reguläre Ausdrücke nicht immer die Antwort auf alle Probleme sind. Die meisten Programmiersprachen bieten spezialisierte Routinen für den Umgang mit Pfadnamen. Trotzdem, um des Beispiels willen, gehe ich sogar noch weiter.

Sowohl Verzeichnis- als auch Dateinamen herauslösen

Die nächste Stufe ist das Zerlegen eines vollständigen Pfadnamens in eine Dateinamen- und eine Verzeichniskomponente. Dazu gibt es viele Möglichkeiten, je nachdem, was gefordert ist. Zunächst könnte man versucht sein, `^(.*)/(.*)$` zu verwenden und mit `$1` und `$2` auf die entsprechenden Teile zuzugreifen. Die Regex sieht hübsch ausgeglichen aus, und mit dem, was wir über gierige Quantifier wissen, sind wir sicher, daß nie etwas mit einem Slash in `$2` landen wird. Der einzige Grund, daß das erste `.*` überhaupt etwas übrigläßt, ist der Slash, der das Backtracking auslöst. Für das zweite `.*` bleiben genau die Zeichen übrig, die das erste beim Backtracking zurückgeben mußte. Damit erhalten wir den Verzeichnisteil in `$1` und den Dateinamen (oder jedenfalls die letzte Komponente des Pfadnamens) in `$2`.

Wir verlassen uns hier auf das gierige erste `^(.*)/` und sind nur deshalb sicher, daß das zweite `^(.*)` keinen Slash einfangen wird. Wir verstehen gierige Quantifier und können das tun. Ich will mich aber genauer ausdrücken und lasse mit `^[^/]*` für den Dateinamen-Teil explizit keinen Slash zu: `^(.*)/([^\/]*)$`. Dieser Ausdruck ist zudem besser selbstdokumentierend.

Ein Problem mit dieser Regex ist, daß sie mindestens einen Slash braucht, damit sie einen Treffer findet. Wenn wir sie auf etwas wie `datei.txt` anwenden, paßt sie nicht. Das kann gewollt sein, wenn dies durch das Programm abgefangen wird:

```
if ( $Pfad =~ m!^(.*)/([^\/]*)$! ) {
    # Treffer -- $1 und $2 sind gültig.
    $Verzeichnis = $1;
    $DateiName = $2;
} else {
    # Kein Treffer, also enthält der Pfadname kein /.
    $Verzeichnis = "."; # "datei.txt" wird zu "./datei.txt" ( "." ist das Arbeitsverzeichnis).
    $DateiName = $Pfad;
}
```

Verschachtelte Klammerpaare

Die Mustersuche nach paarweise vorkommenden Elementen wie Klammern (runden, eckigen, geschweiften) usw. ist ein häufiges Problem. Es stellt sich meist, wenn Konfigurationsdateien oder Programme verarbeitet werden sollen. Nehmen wir an, wir wollten die Argumentenliste einer Funktion aus einem C-Programm herausholen. Argumente folgen dem Funktionsnamen und sind eingeklammert, die Argumente selbst können aber wieder Klammern enthalten, entweder weil sie selbst Funktionsaufrufe sind oder weil Klammern bei algebraischen Ausdrücken auftreten. Zunächst würde man, unter Mißachtung von verschachtelten Klammern, etwa folgendes versuchen: `\bfoo\([^\)]*\)`, aber das funktioniert nicht.

In guter C-Tradition heißt die Funktion im Beispiel natürlich `foo`. Der hervorgehobene Teil der Regex ist der, der auf die Argumente der Funktion passen soll. Mit einem Suchtext wie `foo(2, 4.0)` und `foo(somevar, 3.7)` funktioniert das, wie gewünscht. Leider findet die Regex auch `foo(bar(somevar), 3.7)`, also nicht das Gewünschte. Wir brauchen etwas »Schlaueres« als `[^\)]*`; denkbar wären etwa:

Regex	Beschreibung
1. <code>\(.*\)</code>	Literale Klammern mit irgend etwas dazwischen
2. <code>\([^\)]*\)</code>	Von einer öffnenden Klammer bis zur nächsten schließenden Klammer
3. <code>\([^\(\)]*\)</code>	Von einer öffnenden Klammer bis zur nächsten schließenden Klammer, ohne irgendwelche Klammern zwischendrin

Abbildung 5-1 zeigt, was diese Ausdrücke bei einem Beispiel-Text finden würden.

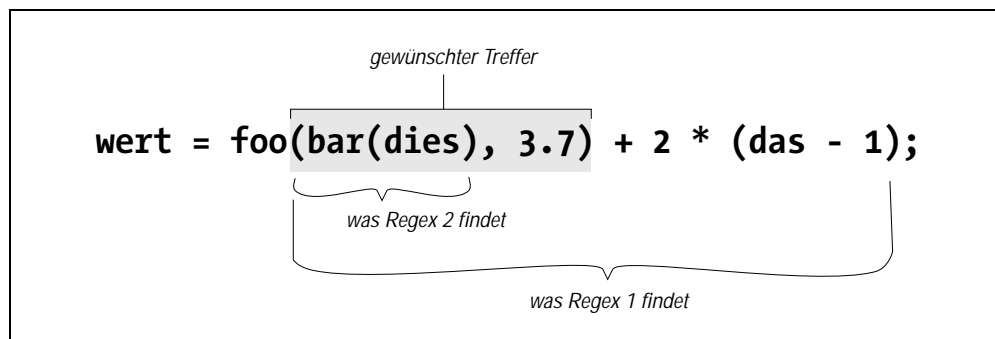


Abbildung 5-1: Was unsere Beispiel-Ausdrücke finden würden

Wie Sie sehen, erkennt Regex 1 zuviel Text² und Regex 2 zuwenig. Regex 3 paßt überhaupt nicht – isoliert würde `>(dies)<` zwar erkannt, aber die öffnende Klammer muß unmittelbar auf den Funktionsnamen `foo` folgen. Also erfüllt keiner der drei Ausdrücke den Zweck.

² Bei `[^\)]*` sollte bei Ihnen ein Warnlicht aufleuchten: Vorsicht, ist der Punkt wirklich das, worauf Sie den Stern anwenden wollen? Manchmal stimmt das ja, aber häufig wird `[^\)]*` falsch benutzt.

Das hat seinen Grund: *Mit regulären Ausdrücken können beliebig verschachtelte Klammerausdrücke nicht erkannt werden. Es geht einfach nicht.* Das war bis vor kurzem ein universell gültiger Satz. Neuerdings gibt es jedoch sowohl in Perl als auch in den .NET-Sprachen Konstrukte, die genau dies dennoch ermöglichen (siehe dazu die Seiten 337 und 443). Auch ohne diese Erweiterungen kann man immerhin einen regulären Ausdruck konstruieren, der *bis zu einer bestimmten Verschachtelungstiefe* funktioniert, aber nicht für beliebig tiefe Verschachtelungen. Ein regulärer Ausdruck, der nur gerade eine Verschachtelung zulässt, ist zum Beispiel:

```
^\([^()]*\(\([^()]*\)\)*\)*\)
```

Der Gedanke, tiefere Verschachtelungen zuzulassen, wird damit schon beängstigend. Und doch – das folgende kleine Perl-Programmstück konstruiert zu einer Verschachtelungstiefe \$depth die entsprechende Regex. Es verwendet dazu den »String x Anzahl«-Operator, der einen String konstruiert, der aus Anzahl zusammengesetzten Strings besteht.

```
$regex = '\(' . '(:[^()])|\(' x $depth . '[^()]*' . '\))*' x $depth . '\)';
```

Die Analyse bleibt Ihnen überlassen.

Ungewollte Treffer vermeiden

Es wird leicht übersehen, was passiert, wenn der untersuchte Text nicht so aussieht, wie man es sich vorgestellt hat. Nehmen wir an, Sie schreiben ein Filter-Programm, das nackten Text in HTML übersetzen soll. Mehrere Bindestriche sollen dabei durch ein <HR>-Tag ersetzt werden, das einem Balken quer über die Seite entspricht (»HR« steht für »horizontal rule«, waagrechter Strich). Mit der Substitution `s/-*/<HR>/` werden tatsächlich Bindestrich-Sequenzen durch Balken ersetzt, aber nur am Beginn einer Zeile. Erstaunt? Nun, `s/-*/<HR>/` setzt bei *jeder* Zeile ein <HR> vorne an, ob nun Bindestriche vorhanden sind oder nicht.

Noch einmal: Jedes Element der Regex, das nicht zwingend vorgeschrieben ist, paßt immer. Wenn `[-*]` auf einen String angesetzt wird, paßt der Ausdruck auf einen oder mehrere Bindestriche am Anfang des Strings. Wenn da aber kein Bindestrich ist, paßt der Ausdruck *auch*, er paßt auf das »Garnichts« am Anfang des Strings. Darum geht es ja beim Stern.

Ein ähnliches Beispiel habe ich im Buch eines bekannten Autors gefunden. Darin wird eine Regex entwickelt, die Zahlen erkennen soll, auch Fließkommazahlen. Zulässige Zahlen beginnen dabei mit einem optionalen Minuszeichen, gefolgt von einer beliebigen Anzahl Ziffern, einem optionalen Dezimalpunkt und beliebig vielen Nachkommastellen. Seine Regex ist `[-]?[0-9]*\.\?[0-9]*`.

Tatsächlich erkennt dies Texte wie `1`, `-272.37`, `129238843.`, `.191919` und sogar so etwas wie `-.0`. Das ist ja auch, was erwartet wird.

Paßt das auch auf »Dieser•Test•enthält•keine•Ziffern«, »Nix•da« oder gar auf den leeren String? Betrachten Sie die Regex genau – *alles* darin ist optional. *Wenn* eine Ziffer vorkommt und *wenn* diese am Anfang des Strings vorkommt, dann wird sie erkannt, aber das ist alles gar nicht vorgeschrieben! Diese Regex paßt auf alle drei der nicht-numerischen Texte, weil sie auf das »Garnichts« am Anfang jedes Strings paßt. Die Regex paßt auch auf dieses Garnichts am Anfang von »Nummer•123«, weil das Garnichts früher paßt als die Zahl 123.

Also: Es ist wichtig zu sagen, was man wirklich meint. Eine Fließkommazahl *muß* mindestens eine Ziffer enthalten, sonst ist es keine Zahl. Beim Aufbau einer besseren Regex fordern wir zunächst, daß vor dem Dezimalpunkt mindestens eine Ziffer erscheinen muß. Dazu verwenden wir das Pluszeichen: `^[0-9]+`. Im Unterausdruck für den fakultativen Nachkomma-Teil ist es wichtig, daß Ziffern nur vorkommen dürfen, wenn auch ein Punkt da ist. Wenn wir naiverweise einfach `^\.[0-9]*` nehmen, dann kann `[0-9]*` passen, ohne daß zuvor ein Punkt erkannt wurde.

Der richtige Weg ist, genau zu sagen, was wir eigentlich wollen. Ein Dezimalpunkt (und mögliche Ziffern danach) ist optional: `(\.[0-9]*)?`. Hier quantifiziert (oder: beschreibt) das Fragezeichen nicht mehr nur den Dezimalpunkt, sondern die ganze Kombination aus Dezimalpunkt und Nachkommastellen. *Innerhalb* dieser Kombination ist der Punkt vorgeschrieben; wenn er fehlt, wird die Maschine nie zu `[0-9]*` vordringen.

Zusammengesetzt lautet der Ausdruck `^-?[0-9]+(\.[0-9]*)?`. Das erkennt Dinge wie `>.007<` nicht, weil unsere Regex mindestens eine Ziffer vor dem Dezimalpunkt verlangt. Wenn wir die linke Seite so ändern, daß auch null Ziffern zugelassen sind, müssen wir entsprechend mindestens eine Nachkommaziffer fordern – wir müssen im gesamten Ausdruck mindestens eine Ziffer haben (sonst hätten wir das Problem, von dem wir ausgegangen sind).

Die hier verfolgte Lösung fügt eine Alternative hinzu, die den bisher unzulässigen Fall abdeckt: `^-?[0-9]+(\.[0-9]*)?|-?\.[0-9]+`. Das erlaubt nun auch nur einen Dezimalpunkt, gefolgt von einer Ziffer oder mehreren. Und ein optionales Minus davor, so etwas vergißt man leicht. Details, Details. Man kann die Alternation auch einklammern und das `^-?` nur einmal davor angeben: `^-?([0-9]+(\.[0-9]*)?)|-\.[0-9]+`.

Das ist zweifellos ein Fortschritt gegenüber dem Original, aber auch dieser Ausdruck kann noch auf `>2003.04.12<` passen. Oft wird angenommen, daß die Regex auf ganz bestimmte Daten angewandt wird oder daß sie als Teil einer größeren Regex gebraucht wird, die vor und nach der Zahl bestimmte Zeichen nicht zuläßt. Unsere Fließkommazahl-Regex funktioniert nur dann, wenn sie in eine größere Regex eingebaut wird, zum Beispiel in `^\^...$` oder auch in `Wert\s*=\s*...$`.

Eingefassten Text erkennen

Einen Text in Anführungszeichen oder eine IP-Adresse zu erkennen sind nur zwei Aufgaben aus einer ganzen Klasse von ähnlich gelagerten Problemen: Man will Text erkennen, der durch bestimmte Begrenzer eingefasst (oder getrennt) ist. Die Begrenzer bestehen häufig aus mehreren Zeichen. Beispiele dazu:

- Kommentare in C, die durch `*/` und `*/` begrenzt werden
- HTML-Tags, also Text, der durch spitze Klammern (`<...>`) wie bei `<CODE>` eingefasst ist
- Text *zwischen* HTML-Tags herausholen, z. B. den Text `>Hier=klicken!` aus dem Link `Hier=klicken!`
- Eine Zeile aus einer `.mailrc`-Datei erkennen. Diese Datei kann E-Mail-Aliases in der Art `alias Kürzel volle-Adresse` enthalten, wie etwa `>alias jeff jfriedl@regex.info`. (Hier sind die Begrenzungszeichen einfach die Leerzeichen zwischen den Wörtern sowie auch die Zeilenenden.)
- Text zwischen Anführungszeichen erkennen und dabei zulassen, daß mittendrin mittels Backslash geschützte Anführungszeichen auftreten, wie im Beispiel:
`a passport needs a "2\"x3\" photo" of the holder.`
- Das Parsen von CSV-Dateien (»comma-separated values«, durch Kommas getrennte Felder, ein häufiges Austausch-Format bei Datenbanken)

Das Problem kann allgemein so angegangen werden:

1. Finde die öffnenden Begrenzungszeichen.
2. Finde den eigentlichen Text
(das heißt: »Alles, was nicht zum schließenden Begrenzer gehört«).
3. Finde die schließenden Begrenzungszeichen.

Wie Sie schon gesehen haben, kann es bei Punkt 2, »Alles, was nicht zum schließenden Begrenzer gehört«, schwierig werden, wenn die Begrenzer aus mehreren Zeichen bestehen oder wenn sie im eigentlichen Text vorkommen können.

Geschützte Anführungszeichen in Strings in Anführungszeichen zulassen

Im Beispiel `2\"x3\"` ist der Begrenzer das Anführungszeichen ("Gänsefüßchen"), aber mittendrin können wieder Anführungszeichen vorkommen, sofern sie mit einem Backslash geschützt sind. Die öffnenden und schließenden Begrenzungszeichen sind einzelne Zeichen und damit einfach zu behandeln. Schwieriger ist es mit dem Text dazwischen. Klares Denken hilft hier: Wenn ein Zeichen kein Anführungszeichen ist (also `[^"]`), dann paßt es sicher. Wenn es ein Anführungszeichen *ist*, dann ist es nur dann zugelassen, wenn davor ein Backslash steht. Wenn wir das »wenn davor« wörtlich in eine Lookbehind-Zusicherung (☞ 136) übersetzen, erhalten wir `"([^\"]|(?<=\\))"`, dies erkennt das `2\"x3\"`-Beispiel korrekt.

Das ist aber auch das perfekte Beispiel, wie sich ungewollte Treffer in eine scheinbar korrekte Regex einschleichen können. Die Regex funktioniert nicht in allen Fällen. Wir möchten, daß die Regex auf den unterstrichenen Teil in diesem etwas albernen Text paßt:

Darth-Symbol: "/-|-\\\" oder "[^~^]"

Aber tatsächlich wird folgendes gefunden:

Darth-Symbol: "/-|-\\\" oder "[^~^]"

Das liegt daran, daß vor dem schließenden Begrenzungszeichen im ersten String tatsächlich ein Backslash auftritt. Dieser Backslash ist selbst durch einen weiteren Backslash geschützt, er schützt nicht das nachfolgende Anführungszeichen (und dieses Anführungszeichen ist tatsächlich das schließende Begrenzungszeichen). Unser Lookbehind-Konstrukt erkennt nicht, daß das Escape-Zeichen selbst geschützt ist, und es kann ja auch sein, daß davor eine ganze Reihe von `>\\<`-Sequenzen auftritt – vielleicht war das Lookbehind doch nicht so eine gute Idee. Wir sollten die geschützten Zeichen schon beim ersten Durchsuchen des Strings erkennen, nicht erst hinterher.

Wir konzentrieren uns jetzt auf die Dinge, die wir innerhalb der öffnenden und schließenden Begrenzern zulassen wollen. Jedes geschützte Zeichen ist zulässig (`(\\.)`), und außerdem jedes Zeichen außer dem Begrenzungszeichen (`(["'])`). Das ergibt `"(\\.|[^\"])*"`.

Toll, Problem gelöst! Leider stimmt hier schon wieder etwas nicht. Auch hier gibt es Strings, die unerwünschte Treffer ergeben, zum Beispiel, wenn das schließende Anführungszeichen fehlt:

"You need a 2\"x3\" photo.

Warum wird hier ein Treffer gefunden? Erinnern wir uns an die Erfahrung aus dem Abschnitt »Gierig oder genügsam – der Treffer geht vor« (☞ 170). Zunächst wird in der Tat der gesamte Text bis zum Punkt erkannt. Danach aber geht die Maschine, weil kein schließendes Anführungszeichen gefunden wird, mittels Backtracking zurück bis zum Zustand:

im Text: <u>>...2\"x3\"</u> <...<	in der Regex: <code>(\\. [^\"])</code>
--------------------------------------	--

An diesem Punkt paßt `(["'])` auf den Backslash, und das Anführungszeichen danach wird fälschlich als schließendes Zeichen erkannt.

Wir lernen daraus:

Wenn mittels Backtracking ein unerwünschter Treffer aus einer Alternation gefunden wird, ist das ein Zeichen dafür, daß auch die erwünschten Treffer nur aus der Anordnung der Alternativen entstehen.

Wären die Alternativen in der Regex vertauscht, würden die geschützten Anführungszeichen nicht übersprungen. Das Problem ergibt sich daraus, daß die eine Alternative auf etwas paßt, was wir eigentlich der anderen Alternative überlassen wollten.

Wie können wir das reparieren? Wie beim Problem mit den Fortsetzungszeilen von Seite 190 müssen wir sicherstellen, daß es keine andere Möglichkeit als die vorgesehene gibt, einen Backslash zu erkennen. Wir ändern also `["^"]` in `["^\\"]`. Das berücksichtigt, daß sowohl das Anführungszeichen als auch der Backslash in diesem Zusammenhang »speziell« sind und entsprechend behandelt werden müssen. Das Resultat `["(\\. |["^\\"])*"]` funktioniert denn auch. (Auch eine funktionierende Regex kann für einem NFA oft noch effizienter gemacht werden; Sie werden dieses Beispiel im nächsten Kapitel erneut antreffen, ☞ 226.)

Aus dem Beispiel können Sie also folgendes lernen:

Beim Aufbau einer Regex müssen auch die Fälle berücksichtigt werden, die *nicht* erkannt werden sollen, besonders auch unerwartete oder »falsche« Daten.

Die Regex funktioniert, aber es gäbe auch andere Reparaturmöglichkeiten. Wenn possessive Quantoren (☞ 144) oder atomare Gruppen (☞ 141) unterstützt werden, könnte man die Regex auch durch `["(\\. |["^"])*+"]` oder `["(?:\\. |["^"])*"]` ersetzen. Beides funktioniert, und doch wird das Problem hier eher vertuscht als gelöst. Es wird einfach verhindert, daß die Maschine mittels Backtracking dorthin zurückkriecht, wo Probleme entstehen könnten.

Wenn Sie verstehen, wie der possessive Quantor oder die atomare Gruppe hier hilft, ist das sehr wertvoll. Dennoch werden wir mit der ersten Reparaturmöglichkeit fortfahren, weil sie eher anwendbar ist. Eigentlich könnte man hier zur ersten Reparatur *auch noch* den possessiven Quantor oder die atomare Klammer anwenden – nicht um das Problem zu lösen, denn das ist schon beseitigt, sondern um die Mustersuche effizienter zu machen, denn so wird ein Fehlschlag viel schneller erkannt.

Erwartete Daten und Annahmen

Dies ist ein guter Zeitpunkt, um auf generelle Punkte beim Aufbau von regulären Ausdrücken zu sprechen zu kommen. Einiges habe ich schon kurz angesprochen. Beim Gebrauch von regulären Ausdrücken und bei deren Anwendung auf Daten in bestimmten Situationen werden meist stillschweigend Annahmen getroffen, und es ist oft wichtig, sich diese bewußt zu machen. Sogar bei etwas ganz Einfachem wie `['a']` werden Annahmen über den verwendeten Zeichensatz getroffen (☞ 30). Meist genügt gesunder Menschenverstand, und deshalb habe ich nicht viel darüber gesprochen.

Nun sind aber viele Annahmen für eine Person völlig klar, aber für eine zweite überhaupt nicht. Unsere Lösung aus dem vorherigen Abschnitt nimmt beispielsweise stillschweigend an, daß geschützte Newlines nicht erkannt werden sollen (oder nur im Modus »Punkt paßt auf alles«, ☞ 113). Wenn der verwendete Dialekt dies unterstützt, kann man dazu in der Regex einfach den Punkt durch ein `['(?:s:.)']` ersetzen.

Auch können die Daten ganz anders geartet sein, als wir es erwarten. Wenn man unsere Regex auf einen Programmtext in fast jeder Programmiersprache anwendet, wird man sein blaues Wunder erleben: Sie scheitert, weil in Kommentaren Anführungszeichen vorkommen können, die nicht gepaart zu sein brauchen.

Es ist nichts Schlechtes dabei, Annahmen über die erwarteten Daten zu treffen oder darüber, wie eine Regex eingesetzt wird. Probleme entstehen erst, wenn diese Annahmen zu optimistisch sind oder wenn eine Regex in Situationen eingesetzt wird, die der Autor nicht eingeplant hat.

Leerzeichen am Anfang und am Ende entfernen

Das Entfernen von Leerzeichen am Anfang und am Ende eines Strings ist nun wirklich keine große Herausforderung, aber das Problem taucht immer wieder auf. Die mit Abstand beste Lösung für dieses Problem ist die einfache und offensichtliche:

```
s/^\s+//;  
s/\s+$//;
```

Als kleine Optimierung wird hier `+` statt `*` benutzt, weil man die Substitution besser gar nicht ausführt, wenn es nichts zu entfernen gibt.

Aus welchen Gründen auch immer scheint es *die* Herausforderung zu sein, eine Lösung zu finden, die beides in einem Schritt erledigt. Ich empfehle so etwas nicht, aber es ist aufschlußreich zu sehen, wie solche Lösungsansätze funktionieren und warum sie nicht zu empfehlen sind.

```
s/\s*(.*?)\s*$/1/s
```

Als die nicht-gierigen Quantoren neu waren, war dies *das* Paradebeispiel; in der Zwischenzeit hat man gemerkt, daß diese Lösung viel langsamer ist als die einfache (in Perl etwa fünfmal langsamer). Das liegt daran, daß bei jedem Zeichen getestet werden muß, ob das auf `*?` Folgende passen kann. Das erfordert eine große Menge von Backtrack-Vorgängen.

```
s/^\s*((?:.*\S?))\s*$/1/s
```

Das sieht viel komplizierter als die vorherige Lösung aus, aber die Mustersuche verläuft hier viel geradliniger und ist nur noch etwa doppelt so langsam wie die kanonische Lösung. Mit `^\s*` wird der Whitespace am Anfang übersprungen, und das `.*?` in der Mitte erkennt alle Zeichen bis zum String-Ende. Mit dem `\S?` wird Backtracking bis zum letzten Nicht-Whitespace-Zeichen erzwungen. Der Whitespace am Ende wird von `\s*` erkannt, das außerhalb der einfangenden Klammern steht.

Das Fragezeichen ist notwendig, damit der Ausdruck auch bei Zeilen funktioniert, die nur Whitespace enthalten. Ohne das Fragezeichen würde die Regex fehlschlagen, und die Zeile würde die Leerzeilen behalten.

```
s/^\s*|\s*$//g
```

Dies ist eine häufig vorgeschlagene Lösung: Sie ist nicht falsch (keine der hier vorgestellten Lösungen ist falsch), aber die Alternation verhindert eine Reihe von sonst möglichen Optimierungen.

Der /g-Modifier wird benutzt, damit beide der Alternativen passen können, aber /g scheint mir etwas übertrieben, wenn wir wissen, daß höchstens zwei Treffer mit der jeweils anderen Alternative möglich sind. Auch diese Lösung ist ziemlich langsam, etwa viermal langsamer als die einfache.

Die Geschwindigkeit dieser Methoden ist oft von der Art der Daten abhängig. In seltenen Fällen, bei sehr langen Strings mit wenig Whitespace an beiden Enden, kann die zweitletzte Methode etwas schneller sein als die einfache. In meinen Programmen benutze ich dennoch

```
s/^\s+//;  
s/\s+$//;
```

oder eine äquivalente Formulierung in der benutzten Programmiersprache, weil es fast immer schneller ist und weil es zweifellos die Absicht viel klarer ausdrückt.

Beispiele zu HTML

In Kapitel 2 hatten wir ein Programm entwickelt, das nackten ASCII-Text in HTML umsetzt und dabei mit regulären Ausdrücken E-Mail-Adressen und http-URLs in entsprechende HTML-Elemente verwandelt. In diesem Abschnitt geht es um ganz ähnliche Aufgaben im Zusammenhang mit HTML.

HTML-Tags erkennen

Man sieht oft den regulären Ausdruck `<[^>+>`, der ein HTML-Tag erkennen soll. Meist funktioniert das auch, zum Beispiel in diesem Perl-Programmstück:

```
$html =~ s/<[^>+>+//g;
```

Es gibt allerdings ein Problem, wenn innerhalb des Tags ein `>><` vorkommt, z. B. bei diesem völlig korrekten HTML-Tag: `><`. Das ist nicht sehr häufig und wird auch nicht empfohlen, aber in einem Tag-Attribut dürfen durchaus `><<` oder `>><` vorkommen, sofern sie in Anführungszeichen stehen. Unser einfaches `<[^>+>` berücksichtigt das nicht, also müssen wir es schlauer machen.

Innerhalb von `>...<` sind Strings in Anführungszeichen erlaubt, sowie »anderes Zeug«, das nicht in Anführungszeichen zu stehen braucht, nämlich alles außer `>><` und den Anführungszeichen. In HTML kann man für die Anführungszeichen wahlweise die Gänsefüßchen oder die Hochkommas nehmen, dagegen kann man diese einfachen oder doppelten Anführungszeichen nicht mit einem Backslash schützen. Wir können daher relativ einfache reguläre Ausdrücke wie `"[^"]*"'` und `'[^']*'` verwenden.

Zusammengesetzt mit der Regex `['^">]` für das »andere Zeug«, erhalten wir:

```
<("[^"]*"|'[^']*'|['^">])*>
```

Das ist ziemlich verwirrend, darum zeige ich dasselbe noch einmal im Modus »Freie Form«:

```
<          # Öffnendes "<"
(          # Jede Anzahl von ...
  "[^"]*"  # Strings in Anführungszeichen
  |        # oder ...
  "'[^']*'" # Strings in Hochkommas
  |        # oder ...
  "[^">"]  # "anderes Zeug".
)*
>          # Schließendes ">"
```

Der Ansatz ist recht elegant. Jeder String in Anführungszeichen wird als Einheit betrachtet, und dabei wird klar gesagt, welche Zeichen an welcher Stelle erlaubt sind. Ein bestimmtes Zeichen kann immer nur auf eine Alternative passen, es gibt hier keine Mehrdeutigkeit, und so besteht auch keine Gefahr, daß sich hier wie bei früheren Beispielen plötzlich ungewollte Treffer einschleichen.

In den ersten zwei Alternativen wird `[*]` und nicht `[+]` benutzt, denn ein String in Hochkommas oder Anführungszeichen darf auch leer sein (z. B. `alt=""`). Dagegen darf in der dritten Alternative weder `[*]` noch `[+]` auftreten, denn einen Quantor gibt es schon für die Klammer, die die Alternation umschließt. Mit einem weiteren Quantor, beispielsweise mit `([">"]+)*`, bekämen wir eine böse Überraschung, auf die ich im nächsten Kapitel im Detail eingehe (☞ 230).

Noch einige Überlegungen zur Effizienz bei einem NFA: Wenn wir den von den Klammern eingefangenen Text nicht benötigen, können wir die Klammern durch nicht-einfangende Klammern ersetzen (☞ 139). Weil es keine Mehrdeutigkeiten zwischen den Alternativen gibt, ist ein erneutes Ausprobieren der anderen Alternativen sinnlos, wenn das `>` am Ende nicht gefunden wird. Wenn eine Alternative gepaßt hat, kann an der gleichen Stelle keine der anderen passen. Wir können deshalb die gespeicherten Zustände ohne Schaden wegwerfen; sie würden ohnehin nur zu Fehlschlägen führen. Wir können also die nicht-einfangenden Klammern gerade auch durch `(?>...)`, also durch atomare Klammern, ersetzen (oder sie mit einem possessiven Stern quantisieren).

Einen HTML-Link erkennen

Wir wollen aus einer HTML-Datei zu jedem Link jeweils die URL und den dazugehörigen Link-Text extrahieren, also die hier unterstrichenen Teile:

```
...<a href="http://www.oreilly.de">O'Reilly Verlag</a>...
```

Weil der Inhalt eines `<A>`-Tags sehr vielgestaltig sein kann, gehe ich das Problem in zwei Schritten an. Im ersten operiere ich die »Innereien« des `<A>` und den Link-Text heraus, im zweiten pflücke ich aus diesen `<A>`-Innereien die URL heraus.

In einem ersten Ansatz verwende ich eine Mustersuche ohne Rücksicht auf Groß- und Kleinschreibung mit dem Modus »Punkt paßt auf alles« und einem nicht-gierigen Stern: `<a\b(?:>+)(.*?)`. Die »Innereien« landen in \$1 und der Link-Text in \$2. Natürlich sollte ich statt `[>+]` den im letzten Abschnitt aufgebauten Ausdruck benutzen; ich verwende hier die einfache Form nur, damit das Beispiel übersichtlich bleibt.

Wenn wir den Inhalt des `<A>`-Tags in einem String haben, können wir ihn mit einer weiteren Regex untersuchen. Die URL kommt darin im Attribut `href=Wert` vor. In HTML ist auf beiden Seiten des Gleichheitszeichens Whitespace erlaubt, und der Wert kann in Anführungszeichen stehen oder auch nicht, wie wir das im letzten Abschnitt gesehen haben. Das folgende Perl-Programm gibt die URLs und die Link-Texte aller Links in der Variablen \$Html aus:

```
# Die Regex in der while-Bedingung ist zu einfach -- siehe Text
while ($Html =~ m{<a\b(?:>+)(.*?)</a>}ig)
{
    my $Innereien = $1; # Resultate aus dem ersten Matching werden in separate ...
    my $LinkText  = $2; # ... Variablen abgespeichert.

    if ($Innereien =~ m{
        \b HREF      # "href"-Attribut.
        \s* = \s*    # Whitespace zu beiden Seiten des = erlaubt.
        (?
            "([^\"]*)" # Der Wert ist ...
            |          # oder ...
            '([^\']*)' # ein String in Hochkommas
            |          # oder ...
            ([^">\s]+) # "anderes Zeug".
        )
    }xi)
    {
        my $Url = $+; # Der gefundene Klammerinhalt mit der höchsten Nummer von $1, $2 usw.
        print "URL $Url mit Link-Text: $LinkText\n";
    }
}
```

Dazu ein paar Anmerkungen:

- Diesmal ist jede Alternative eingeklammert, weil wir den Wert daraus benötigen.
- Wo wir den eingefangenen Text nicht brauchen – bei der Klammer, die die Alternation umschließt –, verwenden wir dagegen nicht-eingefangene Klammern, das ist klarer und effizienter.
- In der Alternative »anderes Zeug« ist diesmal außer Hochkomma, Anführungszeichen und `><` auch kein Whitespace erlaubt, denn dieser trennt die »Attribut=Wert«-Paare.
- Diesmal wird bei der »anderes-Zeug«-Klasse wirklich der Quantor `[+]` benutzt. Damit wird der gesamte Wert des href-Attributs erfaßt. Handeln wir uns damit nicht die genannte »böse Überraschung« ein wie in der »Anderes-Zeug«-Alternative von Seite 205?

Nein, denn diesmal gibt es keinen äußeren Quantor, der mit dem Quantor der Zeichenklasse in Konkurrenz steht. Auch dieses Beispiel werden wir im nächsten Kapitel wiederfinden.

Je nach Art des HTML-Texts wird die URL in \$1, \$2 oder \$3 abgespeichert, die anderen sind leer oder undefiniert. Perl hat genau für diesen Zweck eine besondere Variable, \$+, die den Wert enthält, der vom letzten Klammerpaar aufgefangen wurde, das tatsächlich Text abgekriggt hat. In diesem Fall ist das gerade die URL.

In Perl wird man \$+ benutzen, in anderen Programmiersprachen gibt es andere Möglichkeiten. Man kann mit den normalen Mitteln der Programmiersprache die eingefangenen Textgruppen untersuchen und die letzte nehmen, die nicht leer ist. Falls benannte Klammerausdrücke (☞ 140) unterstützt sind, sind sie hier ideal geeignet, wie beim Programmstück in VB.NET auf der nächsten Seite. (Glücklicherweise unterstützen die .NET-Sprachen benannte Klammerausdrücke, denn das \$+ funktioniert bei ihnen nicht richtig, ☞ 431.)

Eine HTTP-URL auseinandernehmen

Jetzt haben wir eine URL. Wir testen nun, ob es sich um eine HTTP-URL handelt, und wenn ja, zerpfücken wir sie in ihre Komponenten Servername, Portnummer und Pfad. Weil wir bereits wissen, daß es sich um die eine oder andere Art von URL handelt, ist die Aufgabe viel einfacher, als wenn wir in einem beliebigen Text eine URL *identifizieren* müßten. Das ist viel schwieriger; wir werden uns diese Aufgabe weiter hinten in diesem Kapitel vornehmen.

Zu einer gegebenen URL brauchen wir nur die Bestandteile zu identifizieren. Der Server oder Hostname ist alles nach dem `^http://`, aber vor dem nächsten Slash (wenn es denn einen solchen gibt), der Pfad ist alles danach: `^http://([^/]+)(/.*)?$`

Die Portnummer haben wir vergessen. Diese ist optional, steht vor dem Slash und wird vom Servernamen durch einen Doppelpunkt abgetrennt: `^http://([^/:]+(:(\d+))?)(/.*)?$`

Das folgende Perl-Programm zerlegt eine HTTP-URL nach diesem Rezept:

```
if ($url =~ m{^http://([^/:]+(:(\d+))?)(/.*)?$})
{
    my $host = $1;
    my $port = $3 || 80; # $3, falls definiert und nicht leer, sonst 80.
    my $pfad = $4 || "/"; # $4, falls definiert und nicht leer, sonst "/".
    print "Host: $host\n";
    print "Port: $port\n";
    print "Pfad: $pfad\n";
} else {
    print "Keine HTTP-URL\n";
}
```

Ein Link-Checker in VB.NET

Dieses Programm gibt die URLs und Link-Texte aus der Variablen Html aus:

```
Imports System.Text.RegularExpressions
' Reguläre Ausdrücke definieren, die später in der Schleife verwendet werden.
Dim A_Regex as Regex = New Regex("<a\b(?:<Innerei>[^\>]+)>(?:<Link>.*?)</a>", _
    RegexOptions.IgnoreCase)

Dim InnereiRegex as Regex = New Regex( _
    "\b HREF          (?# 'href'-Attribut                )" & _
    "\s* = \s*        (?# '=' mit Whitespace              )" & _
    "(?:            (?# Der Wert ist ...                  )" & _
    "  ""(?:<url>[^\"]*)""  (?# ein String in Anführungszeichen )" & _
    "  |                (?# oder ...                      )" & _
    "  '(?:<url>[^\']*)'   (?# ein String in Hochkommas       )" & _
    "  |                (?# oder ...                      )" & _
    "  (?:<url>[^\"]*>\s+)" (?# 'anderes Zeug'.              )" & _
    ")") & _
    RegexOptions.IgnoreCase OR RegexOptions.IgnorePatternWhitespace)

' Die »Html«-Variable absuchen ...
Dim CheckA as Match = A_Regex.Match(Html)

' Für jeden gefundenen Treffer ...
While CheckA.Success
    ' <a>-Tag gefunden, jetzt extrahieren wir die URL daraus.
    Dim UrlCheck as Match = InnereiRegex.Match(CheckA.Groups("Innerei").Value)
    If UrlCheck.Success
        ' Treffer, also haben wir ein URL/Link-Paar.
        Console.WriteLine("Url " & UrlCheck.Groups("url").Value & _
            " mit Link-Text: " & CheckA.Groups("Link").Value)
    End If
    CheckA = CheckA.NextMatch
End While
```

Dazu ein paar Bemerkungen:

- In VB.NET muß die Regex-Bibliothek zuerst mit Imports eingebunden werden.
- Ich benutze in der Regex Kommentare vom Typ `(?#...)`, weil es ziemlich schwierig ist, in einen VB.NET-String ein Newline einzubauen. Normale Kommentare mit `»#«` reichen entweder bis zum nächsten Newline oder bis ans Ende des Strings; damit würde der ganze Rest der Regex auskommentiert. Für normale Kommentare mit `!#...!` benötigte man ein `&chr(10)` am Ende jeder Zeile (☞ 426).
- Jedes `»"«` in der Regex erfordert ein `»""«` im literalen String (☞ 105).
- In beiden regulären Ausdrücken werden benannte Unterausdrücke eingesetzt, das ergibt aussagekräftigere Namen als bloß `Groups(1)`, `Groups(2)` usw.

Einen Hostnamen auf syntaktische Korrektheit prüfen

Im vorherigen Beispiel hatten wir für den Servernamen nur `['^/:']+`, verwendet, in Kapitel 2 (☞ 78) dagegen das kompliziertere `['-a-z]+(\.[-a-z]+)*\.(com|edu|\.\.\.|info)`. Warum treiben wir derart unterschiedlichen Aufwand für das gleiche Resultat?

Nun, beide Unterausdrücke »passen auf einen Hostnamen«, aber in ganz verschiedenem Zusammenhang. Aus einem String mit bekanntem Aufbau einen Teil herauszupicken ist relativ einfach (wenn wir beispielsweise wissen, daß es sich um eine URL handelt), es ist aber ungleich schwieriger, sicher und eindeutig das gleiche Teil in einem beliebigen String zu identifizieren. Im vorherigen Beispiel haben wir angenommen, daß nach einem `>http://<` ein Hostname kommt, da war das einfache `['^/:']+` angemessen. Im Beispiel aus Kapitel 2 dagegen mußten wir eine Regex entwickeln, die in einem beliebigen Text die Hostnamen erkennt. Darum mußten wir schon spezifischer vorgehen.

Nun wollen wir aber ein drittes Mal und diesmal ganz exakt mit einem regulären Ausdruck prüfen, ob ein wohlgeformter, syntaktisch korrekter Hostname vorliegt. Offiziell besteht ein Internet-Hostname aus mehreren Teilen, die durch Punkte voneinander getrennt sind. Jeder Teil besteht aus ASCII-Buchstaben, -Ziffern und dem Bindestrich. Der Bindestrich darf jedoch weder das erste noch das letzte Zeichen sein. Einen Hostname-Teil können wir also (wir ignorieren Groß- und Kleinschreibung) mit `['a-z0-9']|['a-z0-9'][-a-z0-9]*['a-z0-9']` erkennen. Für den letzten Teil (`>com<`, `>edu<`, `>de<` usw.) gibt es nur eine eingeschränkte Anzahl von Möglichkeiten, wie nebenher in Kapitel 2 erwähnt wurde. Damit bekommen wir den folgenden regulären Ausdruck für einen syntaktisch korrekten Hostnamen:

```
^
(?: # Groß- und Kleinschreibung nicht beachten.
# Ein oder mehrere Teile, durch Punkte getrennt...
(?: ['a-z0-9']\. | ['a-z0-9'][-a-z0-9]*['a-z0-9']\. )+
# Top-level-Domains...
(?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|['a-z']['a-z'] )
$
```

Was auf diese Regex paßt, muß noch nicht unbedingt korrekt sein. Es gibt auch ein Längenlimit für die einzelnen Teile. Diese dürfen nicht länger als 63 Zeichen sein. Wir ersetzen dazu den Stern in `['a-z0-9']*` durch `{0,61}`.

Wenn es ganz offiziell sein soll, müssen wir noch etwas berücksichtigen. Auch ein Name, der nur aus dem obersten Domain-Namen besteht (also `>com<`, `>edu<` usw.), ist an sich syntaktisch korrekt. In der Praxis gibt es solche Rechner für die »Namen«-Domains aber nicht, nur bei manchen zweibuchstabigen Länder-Domains antwortet ein Rechner. Anguilla hat beispielsweise einen Webserver mit der Adresse `http://ai/`, auch bei den Domains `cc`, `co`, `dk`, `mm`, `ph`, `tj`, `tv` und `tw` findet man einen Webserver.

Wenn wir auch das zulassen wollen, ersetzen wir den Plus-Quantor beim mittleren Teil `(?:...)+` durch `(?:...)*`, und wir erhalten:

```
^
  (?i) # Groß- und Kleinschreibung nicht beachten.
  # Ein oder mehrere Teile, durch Punkte getrennt ...
  (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. )*
  # Top-level-Domains ...
  (?: com|edu|gov|int|mil|net|org|biz|info|name|museum|coop|aero|[a-z][a-z] )
$
```

Das funktioniert nun erstklassig bei einem String, der nur aus einem Hostnamen besteht. Da dieses der genaueste unserer drei Hostnamen-Ausdrücke ist, könnten wir versucht sein, auch die Anker vorn und hinten weglassen. So ist das ganz und gar nicht. Diese Regex paßt auf jedes zweibuchstabile Wort: In dieser Hinsicht ist sogar die weniger spezifische Regex aus Kapitel 2 besser. Je nach Kontext und Verwendungszweck ist aber dieser reguläre Ausdruck noch nicht gut genug.

Eine URL in der Praxis erkennen

Bei meiner Arbeit für Yahoo! Finance schreibe ich Programme, die ankommende Finanz- und Börsen-News verarbeiten. Die Artikel werden normalerweise als nackte ASCII-Texte angeliefert, und meine Programme setzen diese in möglichst gut lesbares HTML um. (Auf <http://finance.yahoo.com> können Sie das Resultat begutachten.) Das ist nicht immer eine leichte Aufgabe, weil diese ankommenden Texte ziemlich beliebig (oder gar nicht) formatiert sind. Es ist viel schwieriger, in solchen Texten Hostnamen und URLs zu *identifizieren*, als sie auf syntaktische Korrektheit zu *prüfen*, wenn sie einmal aus den Texten herausgelöst sind. Im letzten Abschnitt wurde darauf angespielt; hier zeige ich Code, der bei Yahoo! wirklich im Einsatz ist.

Wir suchen nach verschiedenen URL-Typen: `mailto`, `http`, `https` und `ftp`. Wenn wir in einem Text ein `>http://<` vorfinden, können wir ziemlich sicher sein, daß es sich wirklich um eine URL handelt, und wir suchen mit einem relativ einfachen regulären Ausdruck wie `http://[-\w]+(\.\w[-\w]*)+` nach dem Hostnamen. Wir wissen, daß es sich um englische Texte in ASCII handelt, daher können wir ziemlich unbesorgt `[-\w]` statt `[-a-z0-9]` verwenden. `[\w]` erkennt auch den Unterstrich und bei manchen Systemen sämtliche Unicode-Zeichen, aber das spielt bei den hier vorliegenden Daten keine Rolle.

Dagegen werden Web-Adressen statt als richtige URLs oft in verkürzter Form ohne `http://` oder `mailto:` angegeben, z. B.:

```
...visit us at www.oreilly.com or mail to orders@oreilly.com.
```

In diesen Fällen müssen wir vorsichtiger vorgehen. Wir benutzen einen Ausdruck, der dem aus dem letzten Abschnitt ähnelt, sich aber in ein paar Punkten von ihm unterscheidet:

```
(?i: [a-z0-9] (?:[-a-z0-9]*[a-z0-9])? \. )+ # Subdomains.
# Top-level Domains -- wir verlangen hier Kleinbuchstaben.
(?-i: com\b
  | edu\b    | biz\b
  | org\b    | gov\b
  | in(?:t|fo)\b      # .int oder .info
  | mil\b    | net\b
  | name\b   | museum\b
  | coop\b   | aero\b
  | [a-z][a-z]\b     # Zweibuchstabige Ländercodes nach ISO.
)
```

In dieser Regex werden `(?-i:…)` und `(?i:…)` benutzt. In bestimmten Teilen der Regex wird also mit Berücksichtigung der Groß- und Kleinschreibung gesucht, in anderen ohne (☞ 137). Wir wollen URLs mit wechselnder Schreibung wie `www.OReilly.com` erkennen, nicht aber ein Börsenkürzel wie `NT.TO` (das Symbol für die Börsenkurse von Nortel Networks an der Börse von Toronto – bei unseren Finanzdaten gibt es solche Abkürzungen in großer Anzahl). Offiziell, nach RFC, darf der letzte Teil eines Hostnamens (z. B. `.com`) durchaus auch in Großbuchstaben geschrieben werden, aber in der Praxis ist das ganz selten der Fall, und wir vernachlässigen diese Möglichkeit. Wir müssen hier zwischen dem, was wir erkennen wollen (jede Art von URL), dem, was wir vermeiden wollen (Börsenkürzel), und der Komplexität abwägen. Wir könnten das `(?-i:…)` auch nur auf die zweibuchstabigen ISO-Ländercodes anwenden, aber in der Praxis sind durchgehend groß geschriebene URLs selten.

Das Folgende ist ein Programmgerüst zum Auffinden von URLs in Texten, in das Sie die einzelnen Unterausdrücke für den Hostnamen einsetzen können:

```
\b
# Protokollteil mit Hostname oder nur Hostname
(
  # Protokoll-Präfix ftp://, http:// oder https://
  (ftp|https?):/[-\w]+(\.\w[-\w]*)+
  |
  # Wenn ohne Präfix: ein spezifischerer Unterausdruck für den Hostnamen
  Volle-Hostnamen-Regex
)

# Optionale Portnummer
( : \d+ )?

# Der Rest der URL ist fakultativ; wenn vorhanden, beginnt er mit einem Slash.
(
  / Pfad-Teil
)?
```

Mit dem Pfad-Teil (also dem Teil der URL, der nach dem Hostnamen kommt und der in <http://www.oreilly.com/catalog/regex/> unterstrichen ist) haben wir uns noch nicht beschäftigt. Es stellt sich heraus, daß dies der Teil ist, der am schwierigsten zu erkennen ist;

hier müssen wir schon ein bißchen raten, damit die Regex in der Praxis etwas taugt. Wie Sie schon in Kapitel 2 gesehen haben, kommt nicht selten direkt nach der URL Text, der auch zulässigerweise Teil der URL sein könnte. Bei einem Text wie

Lesen Sie seine Kommentare auf http://www.oreilly.com/ask_tim/index.html. Er ...

sehen Sie schnell, daß der Punkt nach ›index.html‹ ein Satzpunkt ist und nicht zur URL gehört. Dagegen ist der Punkt *innerhalb* von ›index.html‹ sehr wohl Teil der URL.

Für einen menschlichen Leser liegt das auf der Hand, für ein Programm ist das sehr viel schwieriger; wir müssen hier heuristisch vorgehen. Beim Beispiel in Kapitel 2 hatten wir negatives Lookbehind verwendet und so sichergestellt, daß die URL nicht mit einem Satzzeichen endet. Die bei Yahoo! Finance verwendete Lösung wurde entwickelt, bevor es negatives Lookbehind gab. Sie ist komplizierter als die Methode von Kapitel 2, erreicht aber dasselbe. Der Ansatz ist im Listing auf dieser Seite wiedergegeben und unterscheidet sich in mehreren Punkten von dem aus dem Beispiel von Seite 77; ein Vergleich lohnt sich. Insbesondere die Java-Version der gleichen Regex im Kasten auf der nächsten Seite zeigt, wie die Regex aufgebaut ist.

```
\b
# Protokollteil mit Hostname oder nur Hostname
(
  # Protokoll-Präfix ftp://, http:// oder https://
  (ftp|https?)/?[-\w]+(\.\w[-\w]*)+
  |
  # Wenn ohne Präfix: ein spezifischerer Unterausdruck für den Hostnamen
  (?i: [a-z0-9] (?:[-a-z0-9]*[a-z0-9])? \. )+ # Subdomains
  # Top-level Domains -- wir verlangen hier Kleinbuchstaben.
  (?-i: com\b | edu\b
      | biz\b | gov\b
      | in(?:t|fo)\b # .int oder .info
      | mil\b | net\b
      | org\b | [a-z][a-z]\b # Zweibuchstabige Ländercodes nach ISO
  )
)

# Optionale Portnummer
( : \d+ )?

# Der Rest der URL ist fakultativ; wenn vorhanden, beginnt er mit einem Slash.
(
  /
  # Für den Rest gehen wir heuristisch vor: Was in der Praxis funktioniert, ist »korrekt«.
  [^!.,?;"'<>()\[\]\{\}\s\x7F-\xFF]*
  (?
    [!.,?]+ [^!.,?;"'<>()\[\]\{\}\s\x7F-\xFF]+
  )*
)?
```

Eine Regex aus Variablen aufbauen, in Java

```
String SubDomain = "(?i:[a-z0-9]|[a-z0-9][a-z0-9]*[a-z0-9])";
String TopDomains = "(?x-i:com\\b      \\n" +
                    "      |edu\\b      \\n" +
                    "      |biz\\b      \\n" +
                    "      |in(?:t|fo)\\b  \\n" +
                    "      |mil\\b      \\n" +
                    "      |net\\b      \\n" +
                    "      |org\\b      \\n" +
                    "      |[a-z][a-z]\\b  \\n" + // Ländercodes
                    ")
                    \\n";
String Hostname = "(?:" + SubDomain + "\\.)+" + TopDomains;

String NOT_IN = ";\\'<>()\\[\\]\\{\\}\\|\\s\\x7F-\\xFF";
String NOT_END = "!.,?";
String ANYWHERE = "[^" + NOT_IN + NOT_END + "]*";
String EMBEDDED = "[" + NOT_END + "]*";
String UriPath = "/" + ANYWHERE + "*" + EMBEDDED + "*" + ANYWHERE + "*" + ".*";

String Url =
    "(?x:
    " \\b \\n" +
    " ## Hostname-Teil erkennen \\n" +
    " ( \\n" +
    "   (?: ftp | http s? ): // [-\\w]+(\\.\\w[-\\w]*)+ \\n" +
    "   | \\n" +
    "   " + Hostname + " \\n" +
    " ) \\n" +
    " # Optionale Portnummer zulassen \\n" +
    " (?: \\d+ )? \\n" +
    " \\n" +
    " # Rest der URL ist optional und beginnt mit / \\n" +
    " (?: " + UriPath + " )? \\n" +
    ")";

// String-Darstellung der Regex in ein Regex-Objekt umwandeln.
Pattern UriRegex = Pattern.compile(Url);
// Jetzt können wir die Regex auf Textdateien anwenden ...
:
```

In Wirklichkeit würde ich kaum eine riesige Monster-Regex wie diese schreiben. Ich würde mir eher eine kleine Sammlung oder *Bibliothek* von regulären Ausdrücken aufbauen und diese zu einer größeren zusammensetzen. Ein einfaches Beispiel für diese Technik haben wir schon beim Gebrauch der Variablen `$HostnameRegex` auf Seite 78 benutzt, außerdem im Kasten auf dieser Seite bei der Java-Version des gleichen Programms.

Ausführliche Beispiele

Die nächsten Beispiele illustrieren einige interessante Verfahren bei regulären Ausdrücken. Sie sind ausführlicher, und der Gedankengang hinter den Programmen und auch die Irrtümer, die schließlich zum funktionierenden Programm führen, werden genauer dargestellt.

Mit den Daten im Takt bleiben

Wir betrachten ein längeres Beispiel, das etwas gesucht ist, aber ein paar wichtige Punkte sehr gut veranschaulicht: Es zeigt, warum es wichtig ist, daß die Regex mit den Daten, die abgesucht werden, im Takt bleibt (und dazu die Methoden liefert).

Nehmen wir an, die zu untersuchenden Daten seien eine Reihe von (deutschen, fünfstelligen) Postleitzahlen, die alle zusammengeschrieben sind. Die Aufgabe ist es, alle Postleitzahlen zu finden, die mit 66 beginnen. Hier sehen Sie einige Beispieldaten, die gesuchten Postleitzahlen sind fett gedruckt:

```
03824531669411615213661829503566706752010217663235
```

Wir gehen zunächst von der Idee aus, `{\d\d\d\d\d}` wiederholt anzuwenden. In Perl geht das ganz einfach: Mit `@plz = m/{\d\d\d\d\d}/g` wird ein Array erzeugt, bei dem jedes Element eine Postleitzahl ist (natürlich unter der Annahme, daß die Daten im Standardsuchraum `$_` vorliegen, ☞ 81). Bei anderen Sprachen wird dazu die »find«-Methode für reguläre Ausdrücke in einer Schleife aufgerufen. Ich konzentriere mich im Folgenden mehr auf die regulären Ausdrücke als darauf, wie sie in einer bestimmten Sprache angewendet werden; ich werde Perl für die Beispiele benutzen.

Zurück zum `{\d\d\d\d\d}`. Folgendes wird sich gleich als wichtig erweisen: Die Regex paßt immer und sofort, bis die ganzen Daten erschöpft sind – das Getriebe muß nie ein Zeichen weiterschalten, weil alle Daten Ziffern sind (ich nehme hier an, daß die Daten den Spezifikationen entsprechen, eine Annahme, die in der Praxis manchmal stimmt – und häufig nicht).

Also ist es ziemlich klar, daß eine Änderung von `{\d\d\d\d\d}` zu `{66\d\d\d\d}` gar nichts bringt – sobald ein Fehlschlag auftritt, schiebt das Getriebe die Regex um ein Zeichen weiter, und die Maschinerie gerät außer Takt: Es wird nach `{66\d\d\d}` gesucht und nicht nach dem Anfang einer Fünfergruppe. Mit `{66\d\d\d\d}` wird ein falscher Treffer bei `...5316694116...` gefunden, und das ist keine Postleitzahl.

Man könnte nun einen Zirkumflex oder ein `{^A}` vorne in die Regex einsetzen, aber das würde eine Postleitzahl nur dann erkennen, wenn sie zuvorderst im String auftritt. Wir müssen eine Methode finden, bei der die Regex im Takt bleibt, bei der aber unerwünschte Postleitzahlen ignoriert werden. Dazu müssen wir ganze Postleitzahlen »von Hand« überspringen – und nicht nur einzelne Ziffern, wie es das Getriebe automatisch tut.

Mit erwarteten Daten im Takt bleiben

Hier sind ein paar Möglichkeiten aufgelistet, wie die uninteressanten Postleitzahlen übersprungen werden können. Jeder der aufgezählten Ausdrücke kann vor dem Unterausdruck eingesetzt werden, der auf das paßt, was uns eigentlich interessiert (`(66\d\d\d)`). Wir verwenden nicht-einfangende Klammern für die Postleitzahlen, die uns nicht interessieren, so daß die interessanten mit den einfangenden Klammern in `$1` aufgefangen werden.

`^(?:[^6]\d\d\d\d|\^6]\d\d\d)*...`

Diese Methode überspringt Postleitzahlen, sofern sie mit etwas anderem als 66 beginnen (vorsichtiger wäre vielleicht `[0-57-9]` statt `[^6]`), aber ich nehme wie gesagt an, daß die Daten nur aus Ziffern bestehen). Übrigens – `^(?:[^6][^6]\d\d\d)*` würde nicht funktionieren, weil es unerwünschte Postleitzahlen wie 65432 nicht erkennt und damit nicht überspringt.

`^(?:(?!66)\d\d\d\d)*...`


Diese Methode benutzt negatives Lookahead und überspringt Postleitzahlen, sofern sie nicht mit 66 beginnen. Die deutsche Umschreibung sieht fast gleich aus wie bei der vorherigen Lösung, aber die regulären Ausdrücke unterscheiden sich ziemlich. In diesem Fall erzeugt eine erwünschte Postleitzahl (eine, die mit 66 beginnt) bei `(?!66)` einen Fehlschlag, und damit endet auch das Überspringen von uninteressanten Postleitzahlen.

`^(?:\d\d\d\d\d)*?...`

Bei dieser Methode wird ein nicht-gieriger Quantor benutzt, und Postleitzahlen werden nur übersprungen, wenn es nicht anders geht; das heißt, wenn es von einem später folgenden Unterausdruck, der passen *soll*, erzwungen wird. Aufgrund des minimalen Matchings wird `^(?:\d\d\d\d\d)` überhaupt nur angewandt, wenn der darauf folgende Teil nicht paßt (dann allerdings mehrfach, bis eben dieser folgende Teil einen lokalen Treffer findet).

Wir kombinieren die letzte Lösung mit `(66\d\d\d)` und bekommen:

```
@plz = m/(?:\d\d\d\d\d)*?(66\d\d\d)/g;
```

Das pickt die `>66xxx<`-er Postleitzahlen heraus und überspringt dazwischen aktiv (also nicht mit dem normalen Verhalten des Getriebes) die uninteressanten Zahlen. (In einem Listenkontext liefert `»@array = m/.../g«` eine Liste der Textstücke, die in allen Iterationen von einfangenden Klammerausdrücken erkannt wurden;  318.) Diese Regex funktioniert mit dem `/g`-Modifier, weil wir dafür gesorgt haben, daß die *»aktuelle Position«* nach jeder Iteration des `/g` eine Position am Anfang einer neuen Postleitzahl ist.

Den Takt auch bei Unerwartetem nicht verlieren

Haben wir wirklich sichergestellt, daß die Regex nur am Anfang einer Postleitzahl getestet wird? *Nein!* Wir überspringen zwar *»von Hand«* uninteressante Postleitzahlen *zwischen* jeweils zweien, die mit 66 beginnen, aber wenn die Regex bei der letzten interessanten PLZ

angelangt ist, paßt sie nicht mehr. Dann wird, wie immer, das Getriebe ein Zeichen weiterschalten, und die Regex bei einer Position mitten in einer Postleitzahl anwenden – und unser Ansatz verläßt sich darauf, daß das nie passiert!

Betrachten wir noch einmal unsere Beispieldaten:

03824531669411615213**6618295035**66706~~752010217~~**663235**

Hier sind die gefundenen Treffer fett gedruckt (der dritte davon ist der unerwünschte), die aktiv übersprungenen Postleitzahlen sind unterstrichen, und die durch das Weiterschalten des Getriebes erreichten Positionen sind markiert. Nach dem Treffer 66706 findet die eigentliche Regex keine weiteren Treffer mehr. Ist damit das Matching beendet? Nein, natürlich nicht. Das Getriebe nimmt seine Arbeit auf, schaltet Zeichen für Zeichen weiter und setzt die Regex bei jedem Zeichen neu an; wir geraten also außer Takt mit den eigentlichen Postleitzahlen. Nach dem vierten Zeichen überspringt die Regex 10217 und betrachtet 66323 fälschlicherweise als Postleitzahl.

Alle unsere drei Lösungen funktionieren nur dann zuverlässig, wenn sie am Anfang einer Postleitzahl angesetzt werden, aber das normale Verhalten des Getriebes durchkreuzt das. Man kann hier Abhilfe schaffen, indem man verhindert, daß das Getriebe weiterschaltet, oder dadurch, daß dieses Weiterschalten nicht zeichenweise passiert.

Bei den ersten zwei Methoden können wir das Weiterschalten dadurch verhindern, daß wir den Teil `^(66\d\d\d\d)?` durch Anhängen von `?` optional machen. Wir benutzen dazu die Tatsache, daß die vorangestellten Unterausdrücke `^(?:(!66)\d\d\d\d\d)*...` oder auch `^(?:[6]\d\d\d\d|[\d][6]\d\d\d\d)*...` nur dann verlassen werden, wenn wir gerade vor einer gesuchten Postleitzahl stehen oder wenn überhaupt keine Postleitzahlen mehr folgen (deshalb können wir diese Methode mit dem dritten Ansatz nicht verwenden). Das folgende `^(66\d\d\d\d)?` erkennt also eine der gesuchten Postleitzahlen, wenn eine vorliegt, erzwingt aber kein Backtracking.

Auch diese Lösung ist nicht ganz unproblematisch. Die Regex paßt jetzt auch (auf den Leerstring), wenn gar keine Postleitzahl gefunden wurde, deshalb müssen wir mit den Mitteln der Programmiersprache diese leeren Treffer ausschließen. Sie ist aber schnell, weil nur wenig Backtracking involviert ist und weil das Getriebe die Regex nie an einer neuen Position ansetzen muß.

Mit `\G` im Takt bleiben

Als allgemeinere Lösungsmethode können wir einfach ein `^\G` (☞ 131) vor irgendeinen der drei Ausdrücke setzen. Wir hatten jeden der drei Unterausdrücke so konzipiert, daß er immer gerade nach einer Postleitzahl endet, also nach einer Fünfergruppe. Mit dem `^\G` schaltet das Getriebe nicht weiter, denn bei den meisten Regex-Dialekten paßt es nur, wenn der neue Treffer direkt an den vorherigen anschließt, ohne daß das Getriebe weiterschaltet. (Bei Ruby allerdings bezieht sich das `^\G` auf den »Anfang des neuen Treffers«, nicht auf das »Ende des letzten Treffers«, ☞ 133.)

Mit der zweiten Variante erhalten wir eine Lösung, bei der wir nicht hinterher leere Treffer aussondern müssen:

```
@plz = m/\G(?:?!66)\d\d\d\d\d*(66\d\d\d\d)/g;
```

Dieses Beispiel mit \G im größeren Zusammenhang

Ich weiß sehr wohl, daß dieses Beispiel sehr konstruiert ist. Dennoch hat es uns eine Reihe von wertvollen Ideen vermittelt, wie man eine Regex mit den Daten synchronisiert. Wenn mir ein derartiges Problem in der Praxis begegnete, würde ich es wahrscheinlich nicht allein mit regulären Ausdrücken lösen. Vielleicht würde ich mit `^[\d\d\d\d\d]` Fünfergruppen von Ziffern herauslösen und testen, ob diese mit `>66<` beginnen. In Perl könnte das etwa so aussehen:

```
@plz = ( ); # Leeres Array.

while (m/(\d\d\d\d\d)/g) {
    $plz = $1;
    if (substr($plz, 0, 2) eq "66") {
        push @plz, $plz;
    }
}
```

Im Kasten auf Seite 134 ist ein besonderes interessantes Beispiel für den Gebrauch von \G beschrieben, das allerdings Features benutzt, die es im Moment nur in Perl gibt.

CSV-Dateien verarbeiten

Jeder, der schon einmal versucht hat, CSV-Dateien (Comma Separated Values, durch Komma getrennte Werte) zu parsen, wird bestätigen, daß das etwas verzwickelt ist. Das größte Problem dabei ist wahrscheinlich, daß jedes Programm seine eigene Auffassung davon hat, was *genau* das CSV-Format ist. Ich beginne mit einem Programm, das die Art von CSV-Dateien einliest, die Microsoft Excel erzeugt; wir werden das später für andere Varianten verallgemeinern.³

Das Microsoft-Format ist auch eines der einfacheren. Die durch Kommas getrennten Werte sind entweder »nackt« (stehen einfach zwischen Kommas) oder in Anführungszeichen »eingekleidet«; wenn innerhalb der Anführungszeichen wieder ein Anführungszeichen vorkommt, ist es verdoppelt.

Hier ein Beispiel:

```
Zehn Tausender,10000, 2710 ,, "10,000", "Das war'n ""10 große Scheine"", Baby",10K
```

3 Die Endversion für Microsoft-CSV-Daten finden Sie in Kapitel 6 (⇨ 277).

Die Zeile enthält sieben Felder:

```
Zehn=Tausender
10000
•2710•
  (Ein leeres Feld)
10,000
Das=war'n="10=große=Scheine",•Baby
10K
```

Um diese Zeile zu parsen, benötigen wir einen Ausdruck, der die zwei Typen von Werten erkennt. Die »nackten« Werte sind einfach – sie dürfen alles außer Anführungszeichen und Kommas enthalten; das erkennen wir mit `[^",,]+`.

Die Werte in Anführungszeichen dürfen Kommas, Leerzeichen und überhaupt alles außer Anführungszeichen enthalten. Sie dürfen außerdem zwei Anführungszeichen hintereinander enthalten, die für ein einzelnes Anführungszeichen im eigentlichen Wert stehen. Ein Feld mit Anführungszeichen können wir nach diesem Rezept durch eine beliebige Anzahl von `[^"]|"` innerhalb von `"..."` erkennen, das ergibt `"(?:[^\"]|")*"`. (Mit atomaren Gruppen, mit `(?>...)` statt `(?:...)`, wäre das noch effizienter. Mehr dazu im nächsten Kapitel, [☞ 265.](#))

Zusammengesetzt ergibt sich die Regex `[^",,]+|(?:[^\"]|")*`, die auf ein einzelnes Feld paßt. Das ist schon wieder so unübersichtlich, daß ich besser den Modus »Freie Form« verwende ([☞ 113](#)):

```
[^",,]+          # Entweder ein Feld ohne Kommas und Anführungszeichen ...
|              # ... oder ...
"             # ... ein Feld in Anführungszeichen:
  (?: [^\"] | "" )*  #   Öffnendes Anführungszeichen
                    #   Anführungszeichen nur verdoppelt erlaubt
                    #   Schließendes Anführungszeichen
```

Wir können diese Regex wiederholt auf eine CSV-Zeile anwenden, aber wenn wir wirklich Werte herausholen wollen, müssen wir wissen, welche der Alternativen gepaßt hat. Wenn es ein Feld in Anführungszeichen war, müssen wir die äußeren Anführungszeichen entfernen und die inneren, verdoppelten durch einzelne ersetzen.

Ich kann mir dazu zwei Ansätze vorstellen. Beim ersten betrachten wir das erste Zeichen: Wenn es ein Anführungszeichen ist, entfernen wir das erste und das letzte Zeichen (die äußeren Anführungszeichen) und ersetzen die inneren `>"<` durch `>"<`. Das ist recht einfach, aber mit einfangenden Klammern geht es noch einfacher. Nach dem Matching untersuchen wir die eingefangenen Texte und sehen dann, welche Alternative gepaßt hat:

```

( [^",]+ )      # Entweder ein Feld ohne Kommas und Anführungszeichen ...
|              # ... oder ...
"              # ... ein Feld in Anführungszeichen:
"              #   Öffnendes Anführungszeichen
" ( (? : [^"] | "" ) * ) #   Anführungszeichen nur verdoppelt erlaubt
"              #   Schließendes Anführungszeichen

```

Wenn wir feststellen, daß das erste Klammerpaar einen Text eingefangen hat, verwenden wir diesen als Wert. Wenn es das zweite war, müssen wir noch eventuell vorhandene >"< durch >"< ersetzen.

Das ergibt ein Perl-Programm, das ich später (nach etwas Debugging) auch in Java und in VB.NET umgeschrieben vorstelle. Wir nehmen an, daß die CSV-Zeile in der Variablen \$zeile vorliegt und daß das Newline am Ende schon entfernt wurde (das Newline gehört schließlich nicht zum letzten Wert in der Zeile).

```

while ($zeile =~ m{
    ( [^",]+ )      # Ein Feld ohne Kommas und Anführungszeichen ...
    |              # ... oder ...
    "              # ... ein Feld in Anführungszeichen:
    "              #   Öffnendes Anführungszeichen
    " ( (? : [^"] | "" ) * ) #   Anführungszeichen nur verdoppelt erlaubt
    "              #   Schließendes Anführungszeichen
    }gx)
{
    if (defined $1) {
        $feld = $1;
    } else {
        $feld = $2;
        $feld =~ s/"/"/g;
    }
    print "[$feld]"; # Feld für Debug-Zwecke ausgeben.
    : # Wert in $feld kann jetzt weiterverarbeitet werden.
}

```

Mit unseren Testdaten ergibt sich:

```
[Zehn•Tausender][10000][•2710•][10,000][Das•war'n•"10•große•Scheine",•Baby][10K]
```

Das sieht schon ganz gut aus, aber leider bekommen wir gar nichts für das leere vierte Feld. Wenn das Programm beispielsweise die Felder in ein Array abspeichert, wollen wir für das fünfte Array-Element »10,000« erhalten. Das funktioniert nur, wenn wir auch leere Felder erkennen und entsprechend leere Array-Elemente abspeichern.

Wir könnten ja einfach `[^",]+` durch `[^",]*` ersetzen, so paßt die erste Alternative auch auf leere Felder. Das scheint klar auf der Hand zu liegen, aber funktioniert das auch?

Probieren wir es aus. Wir erhalten:

```
[Zehn•Tausender][][10000][][•2710•][][][10,000][][Das•war'n•"...",•Baby][][10K][]
```

Hoppla. Wir haben irgendwie eine Menge zusätzlicher leerer Felder bekommen! Nun ja, wenn man sich das recht überlegt, sollten wir nicht zu sehr überrascht sein. Mit `(...)*` ist auch ein Treffer möglich, der auf den Nullstring paßt. Das ist ganz in unserem Sinne, wenn tatsächlich ein leeres Feld vorkommt; aber nachdem das erste Feld erkannt wurde, beginnt die Regex-Maschine bei der zweiten Iteration an der Position ›Zehn-Tausender,10000...‹. Nichts in der Regex paßt auf das Komma, aber mit dem Stern ist nun auch ein leerer String ein erfolgreicher Treffer, der genau an der aktuellen Position paßt. Im Prinzip könnte dieser leere String unendlich oft gefunden werden, aber das Getriebe der Regex-Maschine ist so eingerichtet, daß es nach einem erfolgreichen, aber leeren Treffer ein Zeichen weiterschaltet (☞ 133). Daher bekommen wir zwischen allen wirklichen Treffern einen leeren, einen zusätzlichen leeren Treffer vor jedem Feld mit Anführungszeichen und einen leeren Treffer am Ende des Strings.

Das Getriebe ausschalten

Das Problem rührt daher, daß wir uns darauf verlassen haben, daß das Getriebe uns nach jeder Iteration des `/g` über das Komma hinweghilft. Wir können das Problem lösen, indem wir diese Aufgabe selbst übernehmen und das Getriebe gar nicht benutzen. Es gibt hier zwei Möglichkeiten:

- Wir suchen in der Regex explizit nach den Kommas. In diesem Fall gehört das Erkennen des Kommas eigentlich zum Erkennen des Feldes, und wir durchqueren den String komplett, wir erkennen jedes Zeichen.
- Wir könnten bei jedem Feld testen, ob wir an einer Stelle sind, wo ein Feld überhaupt anfangen kann. Felder beginnen entweder am Anfang des Strings oder nach einem Komma.

Oder noch besser – wir können beide Ansätze kombinieren. Den ersten Ansatz können wir so umsetzen, daß jedes Feld außer dem ersten mit einem Komma beginnen muß. Wir könnten auch umgekehrt vorgehen und fordern, daß jedes Feld außer dem letzten mit einem Komma enden muß. Wir setzen dazu vor unsere Regex ein `^(,|)` oder ein `$(,|)` dahinter und fügen die erforderlichen Klammern hinzu:

```
(?:^(,|)
(?:
  ( [^",]* )           # Entweder ein Feld ohne Kommas und Anführungszeichen ...
  |                   # ... oder ...
  "                   # ... ein Feld in Anführungszeichen:
  ( (? [^"] | "" )* ) #   Öffnendes Anführungszeichen
  "                   #   Anführungszeichen nur verdoppelt erlaubt
  )                   #   Schließendes Anführungszeichen
)
```

Das sollte nun wirklich funktionieren, aber wenn wir diese Änderung in unser Programm einbauen, erhalten wir:

```
[Zehn•Tausender][10000][•2710•][][][000][][•Baby][10K]
```

Erwartet hatten wir aber:

```
[Zehn•Tausender][10000][•2710•][][10,000][Das=war'n•"10=große=Scheine",•Baby][10K]
```

Warum hat es denn diesmal nicht funktioniert? Gibt es jetzt ein Problem bei den Feldern, die in Anführungszeichen eingekleidet sind? Nein, das Problem tritt schon vorher auf. Sie erinnern sich an die Lehre, die wir aus dem Beispiel von Seite 179 gezogen haben: *Wenn mehrere Alternativen auf den gleichen Text passen können, muß man sich die Reihenfolge der Alternativen genau überlegen.* Für die erste Alternative, `[^",]*`, ist auch ein leerer String ein erfolgreicher Treffer, daher wird die zweite Alternative nie verwendet, es sei denn, etwas anderes, das später in der Regex auftritt, erzwingt dies. Bei unserer Regex kommt aber nach den Alternativen gar nichts, daher wird die zweite Alternative niemals verwendet!

Man muß sich das noch einmal durch den Kopf gehen lassen. Na dann, probieren wir es noch einmal mit vertauschten Alternativen:

```
(?:^|,)  
(?:  
    "      # Entweder ein Feld in Anführungszeichen ...  
    (?: [^"] | "" )* #   Öffnendes Anführungszeichen  
    "      #   Anführungszeichen nur verdoppelt erlaubt  
    |     #   Schließendes Anführungszeichen  
    (?: [^,]+ ) # ... oder ...  
    )      # ... ein 'nacktes' Feld ohne Kommas und Anführungszeichen.
```

Es funktioniert! Nun ja, mit unseren Testdaten funktioniert es tatsächlich. Schlägt unser Verfahren vielleicht mit anderen Daten fehl? Sicher schadet es nichts, das Programm mit weiteren Daten zu testen. Dieser Abschnitt heißt aber »Das Getriebe ausschalten«, und mit dem Metazeichen `^\G` können wir sicherstellen, daß jede Iteration des `/g` genau da beginnt, wo die letzte geendet hat. Wir vermuten, daß dies ohnehin der Fall ist, weil wir die Regex so konstruiert haben, daß sie beim Durcharbeiten des Strings jedes Zeichen erkennt. Wenn wir am Anfang der Regex ein `^\G` einfügen, verbieten wir alle Treffer, für die das Getriebe weiter-schalten muß. Wir nehmen an, daß dieser Fall gar nicht auftritt, aber wenn er vorkommt, wird der Fehler dafür um so offensichtlicher. Hätten wir das bei der vorherigen, unkorrekten Regex getan, hätten wir statt

```
[Zehn•Tausender][10000][•2710•][][][000][][•Baby][10K]
```

das ebenso falsche

```
[Zehn•Tausender][10000][•2710•][][][
```

erhalten, und wir hätten sofort gemerkt, daß etwas faul ist.

CSV-Dateien mit Java parsen

Hier ist das CSV-Beispiel mit dem `java.util.regex`-Package von Sun. In diesem Beispiel geht es um Klarheit und gute Lesbarkeit; eine effizientere Version finden Sie in Kapitel 8 (☞ 397).

```
import java.util.regex.*;
Pattern FeldRegex = Pattern.compile(
    "\\G(?:^|,)"           \n"+
    "(?:"                 \n"+
    " # Entweder ein Feld in Anführungszeichen ... \n"+
    "  \" # Öffnendes Anführungszeichen           \n"+
    "   ( (?: [^\"]++ | \"\")*+ )                 \n"+
    "  \" # Schließendes Anführungszeichen       \n"+
    " # ... oder ...                               \n"+
    " |"                 \n"+
    " # ... ein Feld ohne Kommas und Anführungszeichen ... \n"+
    "  ( [^\",]* )                                     \n"+
    ") \n", Pattern.COMMENTS);
Pattern AnfzeichenRegex = Pattern.compile("\\\"");
// Aus dem CSV-String in 'Zeile' alle Felder extrahieren ...
Matcher m = FeldRegex.matcher(Zeile);
while (m.find())
{
    String Feld;
    if (m.group(1) != null) {
        Feld = AnfzeichenRegex.matcher(m.group(1)).replaceAll("\"");
    } else {
        Feld = m.group(2);
    }
    // Wert in 'Feld' kann jetzt weiterverarbeitet werden.
    System.out.println("[ " + Feld + " ]");
}
```

Ein anderer Ansatz

Am Anfang dieses Abschnitts wurden zwei Ansätze beschrieben, damit unsere Unterausdrücke mit den Feldern der CSV-Daten im Takt bleiben. Beim zweiten Ansatz wird sichergestellt, daß ein Treffer nur an einer Stelle beginnen darf, an der auch ein Feld beginnt. Oberflächlich betrachtet ist das das gleiche, wie wenn wir vor unsere Regex ein `^(^|,)` setzen, nur benutzen wir hier ein Lookbehind-Konstrukt, nämlich `(?<=^|,)`.

Leider unterstützen viele Programme, wenn sie überhaupt ein Lookbehind kennen, nur das Lookbehind mit Strings fester Länge (siehe Kapitel 3, ☞ 136), dann funktioniert dieser Ansatz nicht. Wenn es nur um die Strings fester Länge geht, könnte man `(?<=^|,)` durch `(?:^|(?<=,))` ersetzen, aber das ist schon übertrieben kompliziert. Außerdem müssen wir

CSV-Dateien mit VB.NET verarbeiten

```
Imports System.Text.RegularExpressions

Dim FeldRegex as Regex = New Regex( _
    "(?:^|,)"           " & _
    "(?:"             " & _
    "    (?# Entweder ein Feld in Anführungszeichen ... ) " & _
    "    "" ( ?# Öffnendes Anführungszeichen ) " & _
    "    ( (?> [^"]+ | """" ) * ) " & _
    "    "" ( ?# Schließendes Anführungszeichen ) " & _
    "    ( ?# ... oder ... ) " & _
    " | "             " & _
    "    ( ?# ... ein Feld ohne Kommas und Anführungszeichen. ) " & _
    "    ( [^",]* ) " & _
    " )", RegexOptions.IgnorePatternWhitespace)

Dim AnzfzRegex as Regex = New Regex("''''") 'Ein String mit zwei Anführungszeichen.
:
Dim FeldMatch as Match = FeldRegex.Match(Zeile)
While FeldMatch.Success
    Dim Feld as String
    If FeldMatch.Groups(1).Success
        Feld = AnzfzRegex.Replace(FeldMatch.Groups(1).Value, "")
    Else
        Feld = FeldMatch.Groups(2).Value
    End If

    Console.WriteLine("[ " & Feld & " ]")
    ' Wert in 'Feld' kann jetzt weiterverarbeitet werden.

    FeldMatch = FeldMatch.NextMatch
End While
```

uns dann wieder auf das Getriebe verlassen, das das Komma nach jedem Feld überspringen muß. Wenn an anderer Stelle ein Fehler auftritt, beginnt ein Matching fälschlicherweise bei einer Position wie >...*10,000*...<. Im ganzen erscheint der erste Ansatz sicherer.

Wir können diesen Ansatz doch noch retten – wir verlangen, daß ein Feld vor einem Komma (oder vor dem Ende des Strings) enden muß. Wenn wir `'(?:= $| ,)` an unsere Regex anhängen, sind wir ganz sicher, daß sie nur Felder findet, die an einer solchen Position enden. In der Praxis verzichtet man wohl eher darauf, aber es ist gut zu wissen, daß man das gleiche Ziel mit mehreren Methoden erreichen kann.

Eine effizientere Regex

Erst im nächsten Kapitel beschäftigen wir uns wirklich mit Effizienzüberlegungen. Dennoch möchte ich hier darauf hinweisen, daß man diese Regex mit atomaren Gruppen wesentlich verbessern kann (☞ 141). Falls unterstützt, kann man den Unterausdruck `(?:[^\"]|")*`, der die Werte in den Feldern mit Anführungszeichen erkennt, durch `(?>[^\"]+|")*` ersetzen. In der VB.NET-Version im Kasten auf der vorherigen Seite wird das so gemacht.

Wenn possessive Quantoren unterstützt sind (☞ 144), wie im Java-Regex-Package von Sun, kann man eine äquivalente Formulierung mit diesen benutzen. Im Kasten mit der Java-Version ist das so programmiert.

Die Überlegungen dahinter werden im nächsten Kapitel diskutiert und führen zu einer noch effizienteren Version, die auf Seite 277 angegeben ist.

Andere CSV-Formate

Das CSV-Format von Microsoft kommt häufig vor, weil es eben von Microsoft ist, aber bei anderen Programmen gibt es leichte Unterschiede. Mir sind schon etliche Variationen untergekommen:

- Manche benutzen ein anderes Trennzeichen, z. B. `;<` oder das Tabulatorzeichen. (Man kann sich dann fragen, ob man das immer noch CSV, *Comma-separated values*, nennen soll.)
- Manchmal ist nach dem Trennzeichen Whitespace erlaubt, der nicht zum Wert des Feldes gehört.
- Oft werden die inneren Anführungszeichen nicht durch Verdoppelung dargestellt, sondern durch einen vorangestellten Backslash geschützt (also `>\"` statt `>""` innerhalb von Anführungszeichen). Meist bedeutet das auch, daß ein Backslash vor allen anderen Zeichen ignoriert wird.

Diese Änderungen sind leicht zu implementieren. Für die erste ersetzt man jedes Komma in der Regex durch das entsprechende Trennzeichen; für die zweite hängt man an das erste Trennzeichen ein `\s*` an, die Regex beginnt also mit `(?:^|,\s*)`.

Für die dritte Änderung können wir auf Code zurückgreifen, den wir schon früher entwickelt haben (☞ 202), und `[^"]+|"` durch `["\\"]+|\\.` ersetzen. Dann müssen wir natürlich auch das folgende `s/"/"/g` durch das allgemeinere `s/\\(.)/$1/g` ersetzen oder durch eine äquivalente Formulierung in der benutzten Programmiersprache.