HANSER

Leseprobe

Jürgen Plate

Der Perl-Programmierer

Perl lernen - Professionell anwenden - Lösungen nutzen

ISBN: 978-3-446-41688-8

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-41688-8

sowie im Buchhandel.

6

Reguläre Ausdrücke

Dieses Kapitel behandelt die folgenden Themen:

- Reguläre Ausdrücke
- Suchen und Ersetzen in Strings
- m//, s/// und tr

Some people, when confronted with a problem,
think „I know, I will use regular expressions".
Now they have two problems.

Jamie Zawinski

Reguläre Ausdrücke (Regular Expressions) werden oft für geheimnisvolle Dinge gehalten, die nur ein wahrer Guru verstehen kann. Sie sind nicht besonders lesbar, wenn man mit ihrer Syntax nicht vertraut ist. In Wirklichkeit sind reguläre Ausdrücke ganz einfach und mit einem kleinen Spickzettel recht leicht zu erstellen.

„Regular Expressions" wurden schon 1956 von dem Mathematiker Stephen Kleene eingeführt. Entstanden sind sie als Menge von Syntaxregeln für die Suche nach übereinstimmenden *Mustern* in Zeichenfolgen. Später adaptierte Kleene sie an die neue Informationstechnik, um sie automatisch abarbeiten zu können. Seither haben Reguläre Ausdrücke

eine Reihe von Veränderungen durchgemacht. Der derzeitige Standard wird von der ISO (International Standards Organization) gehütet, festgelegt wird er von „The Open Group“, einem Zusammenschluss verschiedener Non-Profit-Organisationen aus dem technischen Bereich.

Reguläre Ausdrücke beschreiben Zeichenkettenmuster. Sie werden beispielsweise benutzt, um festzustellen, ob in einem Text ein vorgegebenes Zeichenmuster (engl. pattern) vorkommt oder nicht. Es gibt noch andere Operationen mit regulären Ausdrücken, die Mustersuche (das Auffinden eines Musters im Suchtext) ist jedoch die wichtigste davon. Die Beschreibung eines Musters könnte etwa lauten: „1-3 Buchstaben, ein Strich, 1-2 Buchstaben, 1-4 Ziffern“. Was kann mit diesem Pattern wohl gemeint sein? Richtig: ein deutsches Autokennzeichen.

6.1 Die Basis

Perl beherrscht als eingebautes Feature reguläre Ausdrücke. Für deren Anwendung kennt Perl zwei Operatoren:

- den Mustervergleichs-Operator „`/foo/`“ bzw. „`m/foo/`“ und
- den Ersetzungs-Operator „`s/foo/bar/`“.

Dabei sind die Begrenzer (oben wurden Schrägstriche verwendet) nicht Teil des regulären Ausdrucks.

Reguläre Ausdrücke in Perl basieren auf einem NFA (nicht-deterministischer finiter Automat), der folgendermaßen vorgeht: Er merkt sich die Stellen, an denen mehr als eine Möglichkeit zu kontrollieren ist. Stellt er beim Testen einer Variante fest, dass der Gesamtausdruck nicht mehr zutrifft, geht er zurück zum „Scheideweg“ und prüft die Alternative. Erst wenn alle abgehakt sind, entscheidet der NFA, ob der Ausdruck zutrifft oder nicht. Durch dieses „Backtracking“ genannte Vorgehen beherrscht Perl nummerierte Rückbezüge wie `s/(Eins) (Zwei)/\2\1/g` (ich komme gleich noch darauf). Hier sorgen die Klammern dafür, dass Perl sich jedes „Eins“ und jedes „Zwei“ merkt. Im zweiten Teil vertauscht dann `\2\1` die beiden miteinander.

In einem Perl-Programm muss man irgendwie Anfang und Ende eines regulären Ausdruckes markieren. Man verwendet dazu Zeichen, die man als „Begrenzer“ bezeichnet, zumeist der Schrägstrich. Die Begrenzer gehören selbst nicht mehr zum regulären Ausdruck. Logischerweise kann man die Begrenzer-Zeichen im regulären Ausdruck nicht als literale Zeichen verwenden, man muss sie dort wie die Metazeichen durch einen vorangestellten Backslash maskieren oder einen anderen Begrenzer wählen (auch dazu später noch mehr).

Perl versucht normalerweise, den frühesten Treffer im String zu finden. Kommt aber einer der unten aufgeführten Quantifizierer (z. B. *) ins Spiel, will der NFA so viel wie möglich finden, er wird gierig (greedy). Dabei ist die „Gierigkeit“ stärker als die Links-Bindung. Es gilt daher:

Prinzip 1: Insgesamt liefert ein regulärer Ausdruck die am weitesten links stehende Fundposition im Suchstring.

Prinzip 2: Bei einer Alternative im Suchstring (`a|b|c...`, siehe unten) wird die erste passende Alternative in der Folge verwendet.

Prinzip 3: Die Quantifizierer `,` `*,` `+` und `{n,m}` sind „greedy"; sie nehmen soviel vom String wie irgend geht, um das Suchmuster zu erfüllen.

Prinzip 4: Sind mehrere Elemente im Muster enthalten, holt sich das erste „gefräßige" Element, soviel es bekommen kann (es muss natürlich noch „matchen"), das nächste Element holt sich das Maximum aus dem Reststring usw. Dabei muss der reguläre Ausdruck immer in seiner Gesamtheit erfüllt sein.

Will man beispielsweise in HTML-Code ein bestimmtes Tag erwischen, heißt der erste Versuch vermutlich: `/<.*>/`. übersetzt: „Suche beliebig viele (auch gar kein) Zeichen, umschlossen von spitzen Klammern." Was würde Perl nun in der Zeile

```
<B>Wir</B> sind die <B>Champions</B>!
```

finden? Alles von der ersten spitzen Klammer bis zur letzten vor dem Ausrufezeichen. Da ein Quantifizierer dabei ist, gilt nicht mehr das Suchen nach einem Treffer so weit links wie möglich (also das erste `<B>`), sondern es siegt die Gier, und es bleibt nur das Ausrufezeichen übrig. Die Gierigkeit lässt sich jedoch durch ein hinter `+` oder `*` gesetztes Fragezeichen beschränken. Benutzt man im obigen Beispiel `<.*?>`, wird es `<B>` finden. In solchen Fällen hilft ebenfalls: `/<[^>]+>/`. Dieser Ausdruck sucht ein <, dann etwas, was kein > ist, davon mindestens eines, schließlich ein >. Ähnlich geht man zum Beispiel vor, um Worte in Anführungszeichen zu finden: `/"[^"]+"/` erledigt diesen Job besser als `/".*"/`. Auf solche Quantifizierer und andere Spezialitäten wird in diesem Kapitel noch im Detail eingegangen.

6.2 Operatoren für reguläre Ausdrücke

Wenden wir uns den Operatoren für reguläre Ausdrücke zu. Bei Perl werden reguläre Ausdrücke in Verbindung mit den folgenden Operatoren verwendet:

`=~ m/text/`	„text" ist enthalten (das „m" darf auch fehlen)
`!~ m/text/`	„text" ist nicht enthalten (das „m" darf auch fehlen)
`=~ s/text1/text2/`	Textersetzung „text1" durch „text2"
`=~ tr/././`	Zeichenersetzung

Diese sogenannten Bindungsoperatoren erlauben die Anwendung auf beliebige Variableninhalte. Gleich mal ein einfaches Beispiel dazu, das auch praktisch nutzbar ist. Es werden Umlaute HTML-gerecht ersetzt:

```
$string = "Test ä ö ü Ä Ö Ü ß";

$string =~ s/ä/&auml;/g;
$string =~ s/ö/&ouml;/g;
$string =~ s/ü/&uuml;/g;
$string =~ s/Ä/&Auml;/g;
$string =~ s/Ö/&Ouml;/g;
$string =~ s/Ü/&Uuml;/g;
$string =~ s/ß/&szlig;/g;

print $string, "\n";
```

Es wird jeweils nach dem ersten, zwischen den Begrenzern stehenden Ausdruck gesucht (z. B. „ä“) und dieser dann durch den zweiten (ebenfalls begrenzten) Ausdruck ersetzt (z. B. `&auml;`). Das „g“ am Ende ist ein sogenannter Modifizierer (auch darüber erfahren Sie später mehr). Er legt fest, dass alle „ä“ im Text ersetzt werden sollen – wenn er fehlt, wird nur das erste „ä“ ersetzt. Die Ausgabe lautet dann übrigens:

```
Test &auml; &ouml; &uuml; &Auml; &Ouml; &Uuml; &szlig;
```

Dabei müssen die regulären Ausdrücke (und bei „s“ der Ersetzungsstring) keine konstanten Zeichenketten sein, sondern sie dürfen auch in Variablen gespeichert werden. Ebenso kann auf der linken Seite des Bindungsoperators auch ein Ausdruck stehen, der einen String als Ergebnis hat.

Die Operation liefert einen Wert, der vom Kontext abhängt, in jedem Falle aber eine Interpretation als „wahr“ oder „falsch“ zulässt. „Wahr“, wenn das Muster im Suchtext gefunden wird, sonst „falsch“.

Ein skalarer Kontext liegt beispielsweise in den beiden folgenden Fällen vor:

```
if ($string =~ m/$regex1/) ...
$status = $string =~ m/$regex1/;
```

Das Ergebnis ist jeweils 1 oder 0, was im Booleschen Kontext als „wahr“ oder „falsch“ interpretiert werden kann. Es wird nur nach dem ersten Treffer im Suchtext gesucht.

Ein Listenkontext liegt z. B. in folgenden Fällen vor:

```
@treffer = $pi =~ m/9/g;   # Suche nach Neunen
print join(", ",@treffer), "\n";
# geht auch "perlischer":
print join(", ", $pi =~ m/9/g), "\n";
```

Der Listenkontext ergibt sich aus der links stehenden Operation, also der Listenzuweisung im ersten Falle und der `print`-Anweisung im zweiten. Hier nun das komplette Beispiel zu obigen Ausführungen:

```
$regex1 = 'ttt';          # ganz einfache regulaere Ausdruecke
$regex2 = '926';

$string = "Betttruhe";    # gruseliges Wort (neue deut. Rechtschreibung)
$_      = "Bettruhe";     # nicht so gruselig
$pi     = 3.14159265358979323846264338327795;

# $string untersuchen
if ($string =~ m/$regex1/)
  { print "Schauder!\n"; }
else
  { print "Gute Nacht!\n"; }

$status = $string =~ m/$regex1/;
print "Status: $status\n";

# default $_ untersuchen
if (m/$regex1/)
  { print "Schauder!\n"; }
else
  { print "Gute Nacht!\n"; }

                          # sprintf liefert: "3.1415926536"
if (sprintf("%12.10f\n",$pi) =~ m/$regex2/)
  { print "$regex2 ist drin!\n"; }

# Suche nach Neunen, 'g' als Modifizierer
@treffer = $pi =~ m/9/g;
print join(", ",@treffer), "\n";

# geht auch "perlischer":
print join(", ", $pi =~ m/9/g), "\n";
```

Eine typische Anwendung der Mustersuche ist die Ausgabe aller Zeilen einer Datei, die ein bestimmtes Muster aufweisen. Für diese spezielle Aufgabe (und das Durchsuchen von Arrays) gibt es die spezielle Funktion `grep()`, die später behandelt wird, im Moment soll das Problem mit einfacher Mustersuche gelöst werden. Gegeben ist eine Datei mit IP-Adressenbereichen und Länderkennzeichnungen dazu. Wir suchen mal nach Adressen aus den USA:

```
open(DAT, "<", "data") || die "Kann data nicht lesen: $!\n";

while(my $string = <DAT>)
  { print $string if ($string =~ m/USA/); }

close(DAT);
```

Aber nicht nur die Suche ist oft gefragt, sondern auch das Ändern von Dateiinhalten. Das folgende Programm liest eine Datei, führt eine Stringersetzung in allen Zeilen aus und

schreibt das Ergebnis in eine neue Datei. Diese Datei erhält den Namen der vorhandenen Datei, die vorher zusätzlich die Endung „bak“ erhält. Die Parameter werden von der Kommandozeile übernommen.

```
die "usage: $0 file oldstring newstring\n"
  if ($#ARGV != 2);


my $oldfile = $ARGV[0] . ".bak";
my $newfile = $ARGV[0];
my $old = $ARGV[1];
my $new = $ARGV[2];
my $line;

rename($newfile, $oldfile);
open(OF, "<", $oldfile);
open(NF, ">", $newfile);

# Datei zeilenweise bearbeiten
while ($line = <OF>)
  {
  $line =~ s/$old/$new/;
  print NF $line;
  }
close(OF);
close(NF);
```

6.3 Der Transliterationsoperator tr

Für einfache Aufgaben wie Suchen und Ersetzen oder das Zählen von einzelnen Zeichen gibt es in Perl die Möglichkeit, ohne reguläre Ausdrücke zu arbeiten. Dafür gibt es einen speziellen Transliterationsoperator `tr`. Eine Transliteration besteht darin, einzelne Zeichen durch andere Zeichen zu ersetzen. Das Schema für eine Transliteration in Perl lautet:

```
$string =~ tr/Suchmuster/Ersatzmuster/[Optionen];
```

Dabei sind sowohl das Such- als auch das Ersetzungsmuster keine Zeichenketten, sondern nur einzelne Zeichenfolgen oder Zeichenbereiche. Die Muster sind demnach zwei Listen von Zeichen. Wird in `$string` ein Zeichen aus der Suchliste gefunden, ersetzt es `tr` durch das Zeichen, das an derselben Position in der Ersetzungsliste steht. Zum Beispiel:

```
my $chinesisch = "Rüde Rowdies rauben rüstigem Rentner rotes Rad.";
$chinesisch =~ tr/Rr/Ll/;
print "$chinesisch\n";
# --> Lüde Lowdies lauben lüstigem Lentnel lotes Lad.
```

```
my $string = "ALLES GEHT MIT PERL!";
$string =~ tr/A-Z/a-z/;
print "$string\n";
# --> alles geht mit perl!
```

Die meisten der speziellen Codes für reguläre Ausdrücke können in der Funktion `tr()` nicht verwendet werden. Die folgende Anweisung zählt die Anzahl Sterne in der Variablen `$string` und speichert sie in `$count` ab.

```
my $string = "Fischers * Fritze * fischt * frische * Fische.";
my $count = ($string =~ tr/*/*/);
print "Der Stern kommt $count mal vor!\n";
# --> Der Stern kommt 4 mal vor!
```

Man kann auch Zeichen oder Zeichenklassen zählen, indem auf der Ersetzungseite nichts steht. Auch das zeigt sich am besten im Beispiel:

```
my $string = "Fischers Fritze fischt frische Fische.";
my $count = ($string =~ tr/Ff//);
print "F kommt $count mal vor!\n";
# --> F kommt 5 mal vor!
```

Die Optionen von `tr` erlauben weitere Ersetzungen:

c (complement): Diese Option bedeutet für die Suchliste „alle Zeichen“ außer den angegebenen. Dabei ist mit „alle Zeichen“ der ASCII-Zeichensatz gemeint. Zum Beispiel:

```
my $Variable = "Sügömälüxämülä";
$Variable =~ tr/A-Za-z_0-9/_/c;
print "Variable: $Variable\n";
# --> S_g_m_l_x_m_l_
```

s (squash down): Werden aufeinander folgende Zeichen durch das gleiche Symbol ersetzt, so wird bei Verwendung dieser Option dieses Symbol nur einmal gesetzt. Zum Beispiel:

```
my $string = "Ein   Satz    mit  überflüssigen    Leerzeichen";
$string =~ tr/ //s;
print "$string\n";
# --> Ein Satz mit überflüssigen Leerzeichen
```

d (delete): Ist die Ersetzungsliste kürzer als die Suchliste, wird sie automatisch verlängert. Beispielsweise wird so `tr/abcde/xy/` zu `tr/abcde/xyyyy/`. Die Option d bewirkt, dass Zeichen der Suchliste gelöscht werden, wenn sie nicht in der Ersetzungsliste auftauchen. Zum Beispiel:

```
my $string = "THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG";
$string =~ tr/AEIOU//d;
print "$string\n";
# --> TH QCK BRWN FX JMPS VR TH LZY DG
```

Anstelle der Schrägstriche dürfen beim `tr`-Operator auch andere Zeichen gesetzt werden, wobei Klammern gesondert behandelt werden. Die Zeichen können auch oktal oder hexadezimal angegeben werden, z. B.:

```
my $Wort = "überflüssig";
$Wort =~ tr/\200-\377/\000-\177/;
print "$Wort\n";
# --> |berfl|ssig
```

Der `tr`-Operator kann sogar für schlichte Verschlüsselung verwendet werden. In den Zeiten vor dem World Wide Web, als man Texte und Bilder in der Regel über Newsgruppen verteilte, hat sich eine Simpel-Verschlüsselung namens „rot13“ eingebürgert [1]. Die einzelnen Buchstaben werden einfach durch denjenigen Buchstaben ersetzt, der 13 Positionen weiter rechts im Alphabet steht (natürlich „rundum“, nach „z“ kommt wieder „a“). Da das Alphabet 26 Buchstaben hat, ergibt das zweimalige Anwenden hintereinander wieder den ursprünglichen Text. Man braucht also nur ein Programm für Ver- und Entschlüsselung – oder einen Perl-Einzeiler[2]:

```
my $Satz = "Vs vg qenjf oybbq, vg'f uneqjner.";
$Satz =~ tr/a-zA-Z/n-za-mN-ZA-M/;
print "$Satz\n";
```

6.4 Anker und Zeichenklassen

Es gibt Elemente im regulären Ausdruck, die durch ein einziges Zeichen angegeben werden, und solche, für die mehrere Zeichen gebraucht werden. Zeichen, die keine andere Bedeutung haben, als sich selbst darzustellen, nennt man „literale Zeichen“ oder kurz „Literale“. Man verwendet diesen Ausdruck auch für Elemente, die mit einem Zeichen beschrieben werden, das nach genau diesem Zeichen im Suchtext sucht.

Alle Buchstaben und Ziffern sind, sofern sie in einem regulären Ausdruck als Element auftreten, literale Zeichen, es gibt darüber hinaus noch viele andere literale Zeichen. „X“ ist ein literales Zeichen, als Element in einem regulären Ausdruck sucht es nach einem Zeichen „X“. Das bedeutet jedoch nicht, dass jedes „X“ in einem regulären Ausdruck auch ein solches Element angibt. Im Folgenden werden solche Metazeichen nach und

---

1 Damit keiner sagen konnte, er sei zufällig über einen Frauen-, Männer-, Tier- oder Sonstwas-feindlichen Witz gestolpert. Man musste aktiv rot13 anwenden, um den Klartext zu lesen.

2 If it draws blood, it's hardware.

nach vorgestellt. Beginnen wollen wir jedoch mit den ersten einfachen Elementen eines regulären Ausdrucks.

Die Übersicht in der folgenden Tabelle 6.1 zeigt die Anker von regulären Ausdrücken. „Anker“ bedeutet in diesem Fall, dass der Suchbegriff nicht irgendwo im Text stehen darf, um gefunden zu werden, sondern dass er an einer bestimmten Stelle im Text „verankert“ ist. Die beiden am meisten verwendeten Anker sind der Zeilen- bzw. Stringanfang und das Zeilen- bzw. Stringende. Für einige Zeichenklassen wie „0 – 9“ oder „A – Z“ hat Perl noch einige Abkürzungen.

Tabelle 6.1: Anker bei regulären Ausdrücken

Ausdruck	Beschreibung
^	Dieser Anker steht für den Zeilenbeginn.
	`/^Meier/` adressiert Zeile, die mit „Meier“ beginnt
$	dieser Anker steht für das Zeilenende.
	`/Meier$/` adressiert Zeile, die mit „Meier“ endet.
	`/^$/` adressiert die nächste Leerzeile.
`\b`	Dieser Anker steht für eine Wortgrenze.
	Wortgrenzen findet man am Anfang und am Ende eines Wortes sowie am Textanfang und Textende – so auch rechts und links von einem Leerzeichen zwischen zwei Worten.
`[ ]`	Definiert einen Buchstaben aus dem Bereich in `[ ]`. Beginnt der Bereich mit ^, wird nach einen Zeichen gesucht, das **nicht** im Bereich enthalten ist (Bei Dateinamen war dies das !-Zeichen).
	`[ABC]`: einer der Buchstaben A, B oder C
	`[A-Z]`: Großbuchstaben
	`[A-Za-z]`: alle Buchstaben
	`[^0-9]`: keine Ziffer
.	Der Punkt steht für ein beliebiges Zeichen außer Newline; Er passt im Single-Line-Mode auch auf das `\n`-Zeichen.
(Wort)	Ein gefangenes „Wort“, wird in `$i` gespeichert; es kann sich mittels `\i` im Ausdruck darauf bezogen werden.
(?:Wort)	Gruppieren, aber nicht speichern in `$i` oder `\i`.
x(?=y)	Gefunden wird x wenn y folgt; die Klammer speichert nicht in `$i` oder `\i`.
x(?!y)	Gefunden wird x wenn kein y folgt; die Klammern speichert nicht in `$i` oder `\i`.
`w`	Wortzeichen
`w+`	Ganzes Wort (Wort mit folgendem Leer- oder Satzzeichen)
`W`	Nicht-Wortzeichen: Satzzeichen, Leerzeichen und so weiter
`s`	Leerraum: Leerzeichen, Tabulator, Newline
`S`	Alles, was kein Leerraum ist: Buchstaben, Ziffern etc.
`d`	Ziffer
`D`	Nicht-Ziffer

Einige Beispiele für Zeichenklassen:

```
M[ae][iy]e?r                          Findet alle Kombinationen des Namens
[A-Z]{1,3}-[A-Z]{1,2}[0-9]{1,4}       Filtert deutsche Autokennzeichen
\b([A-Za-z]+)\b +\b\1\b               Findet Wortdopplungen
[0123456789]                          identisch mit der Zeichenklasse \d
[\^0123456789]                        identisch mit der Zeichenklasse \D
[0-9A-Za-z\_]                         identisch mit der Zeichenklasse \w
```

Die Zeichenklasse `[a e i o u]` ist identisch mit der Zeichenklasse `[ aeiou]`. Beachten Sie, dass die Leerstelle ein Listenelement und kein Listentrenner ist, die mehrfache Auflistung eines Elementes hat aber keine Auswirkung.

Zeichenklassentricks

Manchmal kann es Probleme mit Zeichenklassen geben, meist wenn Zeichen wie „–, `^`, `]`, `[`, `*`“ etc. mit in der Zeichenklasse enthalten sein sollen. Da gilt dann:

- „`^`“ darf nicht am Anfang stehen, am besten nach ganz hinten.
- „-“ wird für Bereichen verwendet, entweder nach ganz vorne oder nach ganz hinten.
- Alle anderen Metazeichen werden mit `\` maskiert.
- Es gibt Ersatzdarstellungen für Newline (`\n`), Carriage Return `\r`), Tabulator `\t`) und den Backslash `\\`).

In vielen neueren Implementationen können innerhalb der eckigen Klammern nach POSIX auch Klassen angegeben werden, die selbst wieder eckige Klammern enthalten. In Tabelle 6.2 sind die wichtigsten aufgeführt.

Tabelle 6.2: POSIX-Erweiterung der Zeichenklassen

POSIX	Bedeutung
`[:alnum:]`	Alphanumerische Zeichen: [:alpha:] und [:digit:]
`[:alpha:]`	Buchstaben: [:lower:] und [:upper:]
`[:blank:]`	Leerzeichen und Tabulator
`[:cntrl:]`	Steuerzeichen. Bei ASCII sind das die Zeichen 00 bis 1F, und 7F (DEL)
`[:digit:]`	Ziffern: 0, 1, 2,... bis 9
`[:graph:]`	Grafische Zeichen: [:alnum:] und [:punct:]
`[:lower:]`	Kleinbuchstaben1: nicht notwendigerweise nur von a bis z
`[:print:]`	Druckbare Zeichen: [:alnum:], [:punct:] und Leerzeichen
`[:punct:]`	Interpunktionszeichen
`[:space:]`	Whitespace: Horizontaler und vertikaler Tabulator, Zeilen- und Seitenvorschub, Wagenrücklauf und Leerzeichen
`[:upper:]`	Großbuchstaben1: nicht notwendigerweise nur von A bis Z
`[:xdigit:]`	Hexadezimale Ziffern: 0 bis 9, A bis F, a bis f

Was Buchstaben sind, ist im Allgemeinen *locale*-abhängig, ebenso Sonderzeichen außerhalb des ASCII-Bereichs. Die *locale* wirkt sich auf reguläre Ausdrücke vor allem bei

Zeichenklassen sowie bei den Abkürzungen `\w` und `\W` aus. Wendet man `/\w+/` auf den String „München“ an, findet Perl in der Regel keinen Treffer, dagegen führt die Zeichenklasse `/[A-Za-zäöüÄÖÜß]+/` zum Ziel.

Der Ausdruck `/.+/` akzeptiert Umlaute ebenfalls je nach Einstellungen. Vergessen Sie daher nicht die folgenden Zeilen am Anfang Ihres Programms:

```
use POSIX;
use locale;
# Nur falls nicht schon gesetzt:
setlocale(LC_CTYPE, "de_DE.ISO-8859-1");
```

Das nächste Beispiel druckt für eine beliebige Zahl die einzelnen Ziffern in Worten, beispielsweise zum Bedrucken eines Schecks (Sie erinnern sich noch an diese Papierchen?).

```
my (@ziffern, $zahl, $i);

print 'Geben Sie die zu buchstabierende Zahl ein: ';
chomp($zahl = <STDIN>);
@ziffern = split(//,$zahl);

print "-";
foreach $i (@ziffern)
  { print ziffer($i),"-"; }
print "\n";

sub ziffer
  {
  my $zif = shift;
  $zif =~ m/1/ && print 'eins';
  $zif =~ m/2/ && print 'zwei';
  $zif =~ m/3/ && print 'drei';
  $zif =~ m/4/ && print 'vier';
  $zif =~ m/5/ && print 'fünf';
  $zif =~ m/6/ && print 'sechs';
  $zif =~ m/7/ && print 'sieben';
  $zif =~ m/8/ && print 'acht';
  $zif =~ m/9/ && print 'neun';
  $zif =~ m/0/ && print 'null';
  }
```

In den folgenden Beispielen wird von einer Telefonliste ausgegangen, die aus Namen, Vornamen und Telefonnummern besteht:

```
Huber Karl      123
Meier Hans      231
Gaukeley Gundel 666
Schulze Maria   256
Klever Klaas    400
```

Die Telefonnummern sollen nun um den Text `Tel.:` ergänzt werden. Dazu wird ein weiteres Feature der `s`-Anweisung verwendet: der oben beschriebene Rückbezug.

```
my @data = (
      "Huber Karl      123",
      "Meier Hans      231",
      "Gaukeley Gundel 666",
      "Schulze Maria   256",
      "Klever Klaas    400",
      );

for my $i (0..4)
  {
  $data[$i] =~ s/([0-9][0-9][0-9])/Tel.: \1/;
  print $data[$i], "\n";
  }
```

Das Ergebnis sieht dann folgendermaßen aus:

```
Huber Karl      Tel.: 123
Meier Hans      Tel.: 231
Gaukeley Gundel Tel.: 781
Schulze Maria   Tel.: 256
Klever Klaas    Tel.: 400
```

Jetzt sollen Nachname und Vorname vertauscht werden Beachten Sie die Leerzeichen zwischen den Gruppen. Das „+“-Zeichen wird weiter unten erklärt:

```
my @data = (
      "Huber Karl      123",
      "Meier Hans      231",
      "Gaukeley Gundel 666",
      "Schulze Maria   256",
      "Klever Klaas    400",
      );

for my $i (0..4)
  {
  $data[$i] =~ s/([0-9][0-9][0-9])/Tel.: \1/;
  $data[$i] =~ s/([A-Za-z]+) ([A-Za-z]+)/\2 \1/;
  print $data[$i], "\n";
  }
```

Das Ergebnis:

```
Karl Huber      Tel.: 123
Hans Meier      Tel.: 231
Gundel Gaukeley Tel.: 781
Maria Schulze   Tel.: 256
Klaas Klever    Tel.: 400
```

Aus Gründen der Übersichtlichkeit wurden die Umlaute und das scharfe S weggelassen – in einer realen Anwendung müssen die natürlich mit in die eckige Klammer. Gegebenenfalls sind auch noch weitere Buchstaben zu berücksichtigen, etwa die Vokale mit Akzent.

6.5 Quantifizierer

Mit dem bisher Gezeigten kann man zwar schon schöne Ausdrücke schreiben, aber irgendwie fehlt noch etwas. Wie das letzte Beispiel gezeigt hat, ist es doch relativ umständlich, mehrere aufeinander folgende Zeichen anzugeben ([0-9][0-9][0-9]). Hier helfen die Quantifizierer (auch „Quantoren“ genannt).

Da wäre als Erstes der altbekannte Stern, der für „keinmal bis beliebig oft“ steht (kennt man von der Kommandozeile). Das Suchmuster `/^\d*$/` würde beispielsweise alle Strings finden, die aus 0 bis beliebig vielen Ziffern bestehen. Wie man an dem Beispiel sieht, muss der Quantifizierer immer hinter dem Zeichen bzw. der Zeichenmenge stehen. Neben dem Allquantor * gibt es noch etliche andere. Die Tabelle 6.3 fasst die wichtigsten Quantifizierer zusammen.

Tabelle 6.3: Quantifizierer bei regulären Ausdrücken

Ausdruck	Beschreibung
`*`	Der Stern „*“ steht für eine beliebige Folge des **vorhergehenden** Zeichens (auch null Zeichen!). `a*`: Leer-String oder beliebige Folge von „a“ `aa*`: eine beliebige Folge von „a“ (mindestens eines) `[a-z]*`: Leer-String oder eine beliebige Folge von Kleinbuchstaben `[a-z][a-z]*`: eine beliebige Folge von Kleinbuchstaben (mindestens einer) `.*`: jede beliebige Zeichenfolge
`*?`	Der Stern ist recht „gefräßig“ (greedy), d. h. es wird versucht, maximal viele Zeichen in den regulären Ausdruck einzuschließen. Bei der Zeichenkette „aaa:bbb:ccc“ fände der Ausdruck „`.*:`“ die Zeichenkette „aaa:bbb:“. Durch das nachgestellte Fragezeichen wird diese Eigenschaft umgekehrt, es wird die minimale Teilzeichenkette genommen - also im obigen Beispiel „aaa:“.
`+`	Das Pluszeichen „+“ steht für eine beliebige Folge des **vorhergehenden** Zeichens, jedoch mindestens eines. `a+`: ein a oder beliebige Folge von a’s
`+?`	Die nicht-gierige Variante von „+“.
`?`	Nullmal oder einmal das **vorhergehende** Zeichen.
`\`	Hebt den Metazeichen-Charakter für das folgende Zeichen auf. `a*` steht für Leer-String oder beliebige Folge von a’s. `a\*` steht für die Zeichenfolge „a*“.
`{n}`	Vorangehender Ausdruck genau n-mal (nicht greedy: `{n}?`, entspricht `{n,n}?`, `{n,n}`, `{n}`).
`{n,}`	Vorangehender Ausdruck n-mal oder häufiger (nicht greedy: `{n,}`, möglichst wenig).
`{n,m}`	vorangehender Ausdruck n- bis m-mal (nicht greedy: `{n,m}?`, möglichst wenig).

Wie Sie sehen, können fast alle Quantoren durch ein nachgestelltes Fragezeichen von „gierig“ auf „genügsam“ geschaltet werden. Schließlich gibt es noch reguläre Ausdrücke mit Alternativen, dabei dient das Pipe-Symbol als Trenner:

- `m/a|b/` findet „a“ oder „b“, entspricht also [ab].
- `m/aus|ohne/` findet „aus“ und „Hausbau“, aber auch „ohne“ und „Bohne“.
- `m/a|bc|d/` findet „a“ oder „bc“ oder „d“.
- `m/(a|b)(c|d)/` findet „ac“ oder „ad“ oder „bc“ oder „bd“.
- `m/Er mag (Rot|Weiß)wein/` findet „Rotwein“ oder „Weißwein“.

Dazu einige Beispiele:
Suche nach einem Datum im Format tt.mm.jjjj (also Tag und Monat zweistellig, Jahr vierstellig):

```
my $text = "Am 09.06.2009 ist Donald Duck 75 Jahre alt geworden.";
if ($text =~ m/(\d{2}).(\d{2}).(\d{4})/)
  {
  print "Datum $1 - $2 - $3 gefunden";
  }
else
  {
  print "Text mit Datum nicht gefunden";
  }
```

Extrahiere zwei Zahlen aus einem String und multipliziere diese:

```
my $string = "12 mal 7 ergibt ";
$string =~ m/(\d+)[^\d]+(\d+)/;
print $string . ( $1 * $2 ) . "\n" ;
```

HTML-Tags aus einem Text entfernen:

```
my $string = "Dies ist ein <b>Text</b> mit <i>HTML</i>-Tags";
$string =~ s/<[^>]*>//g;
print "Nach Ersetzung: $string\n";
```

Dateinamen aus einer URL erzeugen; z. B. wird aus „http://www.cia.gov/data/secret.html“ der Dateiname „www.cia.gov-data-secret.html“.

```
$filename = $url;
$filename =~ s|^http://||;                  # nur den Domainnamen
$filename =~ s|/|-|g;                       # '/' --> '-'
$filename =~ s|-$||;                        # '-' am Ende weg
$filename .= '.html' unless ($filename =~ /html$/);
```

Noch einige Beispiele:

```
# Leerzeichen am Anfang und am Ende entfernen
$line =~ s/^\s+//;
$line =~ s/\s+$//;

# Testen, ob eine Jahreszahl vierstellig ist
# Format: nn/nn/nnnn oder nn-nn-nnnn --> true, falls OK
($line =~ m/[0-9]{2}[\/|-][0-9]{2}[\/|-][0-9]{4}/)
```

Schließlich gibt es noch ein paar ganz besondere Spezialzeichen (wie auch immer man sie nennen will). Sie sind in Tabelle 6.4 aufgeführt.

Tabelle 6.4: Sonderzeichen bei regulären Ausdrücken

Ausdruck	Beschreibung
\	Backslash, nimmt Metazeichen die Spezialbedeutung
\U	alle Zeichen bis \E als Großbuchstaben behandeln
\L	alle Zeichen bis \E als Kleinbuchstaben behandeln
\Q	alle Metazeichen bis \E mit \ schützen
\E	Ende von \U, \L, \Q

Achtung:

Die Metazeichen + – ? . * ^ $ ( ) [ ] { } | \ müssen durch einen davorstehenden Backslash (\) geschützt werden, wenn sie im Suchstring auftauchen.

Manchmal kann man schwer auseinander halten, wann welches Zeichen eine Sonderbedeutung hat und wann nicht. Enthält ein Ausdruck viele Dollars, Schrägstriche oder Klammern, die für sich selbst stehen (und nicht für Zeilenende, Trenner oder Gruppierungen), müssen sie geschützt (quotiert) werden. In Einzelfällen genügt dafür ein vorangestellter Backslash. Bei langen Ausdrücken oder vielen Sonderzeichen erlaubt Perl mehrere Varianten des Quotierens, beispielsweise die Funktion `quotemeta()`. Innerhalb regulärer Ausdrücke dienen die oben erwähnten Zeichen \Q und \E diesem Zweck.

```
\Q(<- Irgendein String mit Sonderzeichen, z. B. /\/\/\ ->)\E.

/Von:\s*\Q$von\E/;
```

Die Musterverarbeitung mit `\Q` auszuschalten, ist auch für die Interpolation von Variablen innerhalb von Mustern nützlich (wie die zweite Zeile im Beispiel zeigt), um ungewöhnliche Ergebnisse bei der Eingabe von Suchmustern zu verhindern: Die Klammern und Schrägstriche verlieren ihre Sonderbedeutung und werden als normale Zeichen behandelt.

Vorausschauende reguläre Ausdrücke

Perls reguläre Ausdrücke können vorausschauen, ob ein String passen könnte (lookahead). Mit */Haus(?=bau)/* findet der Interpreter „Haus“ nur dann, wenn „bau“ folgt. Schließt sich an „Haus“ jedoch „herr“ an, trifft der Ausdruck nicht mehr zu. Das Ganze darf man auch verneinen: „Finde jedes 'Haus', aber nur, wenn dem kein 'bau' folgt“: `/Haus(?!bau)/`.

6.6 Klammern und Rückbezüge

Durch Klammerung kann man eine Elementfolge im regulären Ausdruck zu einem komplexen Element zusammenfassen, und durch einen auf dieses Element angewendeten Quantor lässt sich ein neues Element konstruieren. Dazu ein Beispiel:

```
my $string = "18.07.2009";

print "passt\n" if $string =~ m/\d{1,2}.\d{1,2}.\d{4}/;
print "passt\n" if $string =~ m/(\d{1,2}.){2}\d{4}/;
```

Die beiden Ausdrücke sind also äquivalent. Die Klammerung dient oft nicht nur der Zusammenfassung einer Elementenfolge zu einem komplexen Element, sondern auch zur Abspeicherung des Treffers. Die obigen Ausdrücke beschreiben daher zwar dasselbe Muster, sind aber trotzdem nicht vollkommen gleichwertig. Über die Klammen kann man Teilausdrücke als „Treffer“ bzw. Rückbezüge abspeichern. Diese Klammern sind also einfangende Klammern.

Für nummerierte Rückbezüge kennt Perl zwei Schreibweisen: Die Variablen `$1`, `$2`, `$3` und so weiter enthalten jeweils den Wert des in der ersten, zweiten, dritten ... einfangenden Klammer gefundenen Musters. Sie lassen sich außerhalb des regulären Ausdrucks irgendwo im Programm verwenden.

`\1`, `\2` sind dagegen Bestandteile der RegEx-Maschine. In ihnen steht ebenfalls der in den einfangenden Klammern gefundene Wert. Die Anzahl der Rückbezüge ist in beiden Fällen unbegrenzt. Dabei gelten folgende Regeln:

- Das Element in der i-ten Klammer (von links) wird in `\i` bzw. `$i` gespeichert.
- `\i` steht zur Verfügung, nachdem das betreffende Element ermittelt wurde, es ist also frühestens nach der schließenden Klammer des entsprechenden Elementes bekannt.
- Die Variable `$i` wird besetzt, nachdem die Mustersuche erfolgreich abgeschlossen wurde, sie erhält den letzten Wert von `\i`. Bleibt die Mustersuche erfolglos, so ist der Wert von `$i` undefiniert. Der Geltungsbereich des Namens ist automatisch lokal innerhalb des umgebenden Blocks.
- Bei einem Quantor außerhalb der Klammerung kann das geklammerte Element mehrere Treffer erzielen; es wird in der betreffenden Variablen nur der letzte dieser Treffer abgespeichert.

Vorsicht ist jedoch geboten: Aufgrund der Art und Weise, wie die RegEx-Maschine den Ausdruck interpoliert und compiliert, gibt es durchaus einen Unterschied zwischen `$i` und `\i`. Empfohlen wird, innerhalb des Suchausdrucks nur den Rückbezug mittels `$i` zu verwenden, da Variablen wie `$i` erst später ausgewertet werden und beim ersten Interpretieren eines Ausdrucks möglicherweise noch gar keinen Wert besitzen.

Neben `$1`, `$2`, ... belegt Perl bei jeder RegEx-Auswertung einige Spezialvariablen neu: In `$&` findet sich immer der letzte gültige Treffer, in `$&´` alles, was vor ihm und in `$&`` das, was nach ihm lag.

Perl erlaubt in seinen regulären Ausdrücken Ausdrücke und Funktionen, die einen korrekten String ergeben, wenn man den Modifizierer `e` benutzt. Er sorgt dafür, dass die RegEx-Maschine alle Variablen interpoliert, den Ausdruck übersetzt, `\1` usw. belegt und den Ersetzungsteil evaluiert.

Der folgende Ausdruck ersetzt beispielsweise alle Zeichenfolgen, die auf das Muster „%-Zeichen, gefolgt von zwei Sedezimalziffern“ passen, durch den Buchstaben, der den ASCII-Wert der Hexzahl hat:

```
$value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
```

Wie Sie sehen, kann der Ersetzungsteil auch einen Perl-Ausdruck enthalten, der sich seinerseits auf Teile des Suchstrings bezieht (über $1).

Im folgenden Beispiel wird „Betriebssystem“ durch den Rückgabewert des Unterprogramms `os()` ersetzt. In diesem Beispiel werden nur Unix-artige Betriebssysteme berücksichtigt.

```
...
$system =~ s/Betriebssystem/&os/eg;
...

sub os
 {
 $string = `uname -a`;
 return $1 if ($string =~ /(^\w+)/);
 }
```

Tabelle 6.5: Spezialvariablen bei regulären Ausdrücken

Variable	Beschreibung
`\i`	Rückbezug auf die i-te einfangende Klammer
`$i`	Wert der i-ten einfangenden Klammer; nur im Ersetzungsteil verwendbar
`$&`	letzter gefundener Treffer
`$'`	Text nach dem Treffer
`` $` ``	Text vor dem Treffer
`$+`	Treffer der letzten einfangenden Klammer

Dazu ein winziges Beispiel:

```
$text = "Heute muss die Glocke werden";
print "|$`<$&>$'|\n" if ($text =~ /Glocke/);

# Ergebnis: |Heute muss die <Glocke> werden|
```

6.7 Modifizierer

Am Ende eines Operators für reguläre Ausdrücke (also die inzwischen hinreichend bekannten Operatoren `m/.../` und `s/.../.../`) können noch Modifizierer stehen, die das Gesamtverhalten regeln. So haben Sie bereits das „g“ kennengelernt, das dafür sorgt, dass nicht nur das erste Auftretens eines Suchstrings ersetzt wird, sondern alle. Die Tabelle 6.6 zeigt weitere Modifizierer.

Tabelle 6.6: Modifizierer bei regulären Ausdrücken

Modifizier	Beschreibung
g	jedes Vorkommen finden
i	Groß- und Kleinschreibung ignorieren
e	Ausdruck im Ersetzungsteil erst evaluieren, dann ersetzen
x	Kommentare und Leerzeichen im Suchen-Teil erlaubt
m	Multiline-Mode, ^ und $ passen auf logische Zeilenanfänge und -enden, der Punkt matched kein Newline
s	Singleline-Mode, ^ und $ erkennen Anfang/Ende des gesamten Strings, . matched Newline
sm	kombiniert: Logische Zeilen plus Newline

Durch den Modifizierer „i“ wird beispielsweise innerhalb des Suchstrings nicht zwischen Groß- und Kleinschreibung unterschieden, was manche Ausdrücke vereinfacht.

```
my $string = "HALLO WELT!";

$string =~ s/hallo/HEJ/i;
print $string, "\n";
# --> HEJ WELT!
```

Der Modifizierer „e“ ist wichtig, wenn man sich im Ersetzungsausdruck rückbezüglich nicht auf `\i`, sondern auf `$i` stützt. Das Beispiel von oben

```
$value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
```

würde ohne den Modifizierer nicht klappen, weil `$1` eventuell noch nicht ausgewertet ist, wenn die Funktion `pack()` aufgerufen wird. Bei Rückbezügen ist es günstig, auf jeden Fall „e“ zu verwenden.

Einige Zeichen ändern ihre Bedeutung allerdings je nach Zusammenhang, zum Beispiel das Leerzeichen. Normalerweise steht es für sich selbst und spielt beim Muster eine Rolle. Verwendet man jedoch den Modifizierer `x` am Ende des Ausdrucks, sind Leerzeichen und Kommentare im Ausdruck erlaubt. Das verbessert vor allem bei komplizierten Ausdrücken die Lesbarkeit:

```
/"      # Anfuehrungszeichen
[^"]+   # gefolgt von etwas, das kein Anfuehrungszeichen enthaelt
"\x     # gefolgt Anfuehrungszeichen
```

Wer in solchen Ausdrücken ein Leerzeichen braucht, muss es mit einem Backslash schützen. Die Zeichenklasse `\s` kann nur dann als Ersatz dienen, wenn es auf die Art des Leerraums nicht ankommt.

Noch ein Beispiel. Es soll der Dateipfad einen Bildes aus dem HTML-IMG-Tag extrahiert werden. Der reguläre Ausdruck `/<IMG\s+SRC="(.+?)".*?>/i` erschließt sich nicht unbedingt auf den ersten Blick. Verteilt man ihn dagegen auf mehrere Zeilen und kommentiert ihn, wird es schon übersichtlicher:

```
...
$t = "...<IMG SRC=\"pfad/bild.gif\" WIDTH=110>...";

$t =~ /<IMG          # HTML-Tag für Bild (Beginn)
           \s+         # Leerzeichen
           SRC="       # Hier kommt der URL
              (.+?)    # Diesen Pfad suchen wir
           "           # Ende des URL
           .*?         # vielleicht noch Optionen
         >           # Ende des HTML-Tags
      /ix;         # schreibweisenunabhängig, extended

print "$1\n";
...
```

Perl unterstützt mit den Modifizierern `s` und `m` zwei Möglichkeiten, wie bei der Zeichenklasse „.“ mit einem Newline-Zeichen umgegangen werden soll und wie sich Zeilenanfang/Stringanfang (`^`) Zeilenende/Stringende (`$`) verhalten. Diese beiden Modifizierer schalten den Single- beziehungsweise Multiline-Modus ein.

Ohne die Modifizierer trifft der Punkt kein Newline, und `$` nimmt das „natürliche“ Zeilenende Newline (`\n`). Schaltet man nun den Perl-Zeilentrenner `$/` zum Beispiel auf „X“ um, passt `$` nicht mehr auf das Newline, sondern auf „X“. Es lässt sich beispielsweise eine Datei als ein langer String in einem Skalar speichern, indem man den Zeilentrenner umdefiniert:

```
undef $/;
open (DAT, "<", "data") || die "No data: $!\n";
$string = <DAT>;
close(DAT);
```

Auf diese Weise gibt es keine einzelnen Zeilen, sondern in `$string` landet die ganze Datei. In diesem Fall passen `^` und `$` auf den Stringanfang beziehungsweise dessen Ende.

Im Multiline-Modus ändert sich das Verhalten der Anker `^` und `$` in Bezug auf Zeilenanfänge und -enden. Der Punkt, der für jedes beliebige Zeichen steht, findet normalerweise kein Newline.

Der Unterschied zwischen Multiline- und normalem Mode offenbart sich, wenn man `$/` verändert. Setzt man es wie oben auf „X“, passen `^` und `$` auf zwei Dinge: zum einen auf das durch `$/` neu definierte Zeilenende und zum anderen auf das logische Zeilenende und den logischen Zeilenanfang. Das Newline steckt ja noch in der Datei, deshalb ist der logische Zeilenanfang die Stelle hinter dem Newline und das logische Zeilenende das Zeichen vor dem Newline.

Die Newlines verschwinden ja nicht durch eine Änderung von `$/`, sondern sie markieren lediglich nicht mehr das Zeilenende. Wurde eine Datei wie oben in einen String eingelesen, finden `^` und `$` im Multiline-Mode auch die logischen Zeilenenden und -anfänge, die sich mitten im Text befinden, und nicht nur den Dateianfang und ihr Ende wie im normalen Modus. Dazu legen wir eine Datei mit drei Zeilen an:

```
Zeile1
Zeile2
Zeile3
```

Diese Datei wird als langer String in einem Skalar eingelesen. Benutzt man nun den Multiline-Ausdruck `m/(\w+)$/m`, findet Perl „Zeile1“, weil `$` auch logische Zeilenenden erkennt. Verwendet man dagegen den Modifizierer `s`, wird „Zeile3“ ausgegeben:

```
undef $/;

open (DAT, "<", "data") || die "No data: $!\n";
$string = <DAT>;
close(DAT);
print $string, "\n\n";

# Ohne Modifizierer
$string =~ m/(\w+)$/;
$zeile = $1;
print "O: ", $zeile, "\n";

# Multiline-Modus
$string =~ m/(\w+)$/m;
$zeile = $1;
print "M: ", $zeile, "\n";

# Singleline-Modus
$string =~ m/(\w+)$/s;
$zeile = $1;
print "S: ", $zeile, "\n";
```

Die Programmausgabe liefert wie erwartet:

```
Zeile1
Zeile2
Zeile3

O: Zeile3
M: Zeile1
S: Zeile3
```

Abschließend zwei Beispiele für häufig verwendete reguläre Ausdrücke: Der erste Ausdruck beschreibt eine E-Mail-Adresse (zumindest oberflächlich), um testen zu können, ob sich ein Sendeversuch überhaupt lohnt:

```
([\w\-\.]+)@([\w\-\.]+)
```

Die beiden Zeichenklassen (`[\w\-\+\._]`) vor und hinter dem „@“ beschreiben einen Namen aus Buchstaben, Ziffern, dem Minuszeichen, dem Underline und dem Punkt. Das deckt sich ungefähr mit Usernamen bzw. Domainangaben. Eine Gültigkeit der Adresse ist jedoch nicht feststellbar.

Der zweite Ausdruck zeigt die Zerlegung einer URL in ihre Teilkomponenten:

```
($host,$port,$file) = ($url =~ m|http://([^/:]+):{0,1}(\d*)(.*)$|);
```

`http://` steht für sich selbst, dann kommt eine Zeichenfolge, die entweder mit „/“ oder „:“ endet, also die Hostangabe. „:0,1“ deckt sich mit einem oder keinem Doppelpunkt. Die folgende Ziffernfolge (die auch leer sein kann) wird in $port gespeichert und der Rest der URL in $file. Das Ganze geht schief, wenn der Dateiname nur aus Ziffern besteht.

Der Modifizierer „g“ hat noch eine besondere Eigenschaft. Wird er innerhalb einer Schleife verwendet, kann man nacheinander jedes Auftreten eines bestimmten Suchmusters bearbeiten. Zum Beispiel kann man im folgenden Listing zählen, wie oft „Fisch“ auftaucht:

```
use strict;
use warnings;

my $text = 'Ein Fisch, zwei Fische, drei Fische, ' .
           'vier Fische, Goldfische, Silberfische';

my $count = 0;
$count++ while ($text =~ m/fisch/gi);
print $count, " Fische gefunden\n";
```

Als Ergebnis erhalten wir „6 Fische gefunden“, weil ja auch die Groß- und Kleinschreibung ignoriert wurden.

6.8 Beliebige Begrenzer

Oft verändern auch die Begrenzer des Ausdrucks ihre Bedeutung. Teilweise kombiniert man die Modifizierer x und m, um den Schrägstrich als normales Zeichen verwenden zu können. Der Perl-Befehl `m` schaltet explizit die Wahl eines neuen Begrenzers für den Ausdruck ein, er gilt nur für die Suche. Die übliche (verkürzte) Schreibweise `/regexp/` ist nur eine Abkürzung für `m/regexp/`. Das folgende Beispiel macht dasselbe wie die Anweisung `(split("/", $path))[-1]`, nämlich den Dateinamen zu extrahieren:

```
$path =~ m#([^/]*)$#;
$filename = $1;
```

`m#` sorgt dafür, dass man den Schrägstrich beliebig im regulären Ausdruck benutzen kann. Beim Suchen und Ersetzen mit `s` kann ebenfalls ein anderes Zeichen benutzt werden (ich selbst verwende gerne mal den senkrechten Strich). Es geht aber auch mit diversen Klammern:

```
$path = "/usr/local/bin/tief/drin";

$path =~ m#([^/]*)$#;
$filename = $1;
print $filename, "\n";

$path =~ m{([^/]*)$};
$filename = $1;
print $filename, "\n";

$path =~ m[([^/]*)$];
$filename = $1;
print $filename, "\n";

$path =~ m(([^/]*)$);
$filename = $1;
print $filename, "\n";

$path =~ m<([^/]*)$>;
$filename = $1;
print $filename, "\n";
```

Besonders erstaunlich finden Sie sicher den letzten Ausdruck mit den < ... >-Klammern.

Das folgende Programmbeispiel zeigt die Mächtigkeit von Perl bei der Bearbeitung regulärer Ausdrücke. Es erzeugt eine Cross-Referenz aller Befehlsworte, Variablen und Unterprogramme eines Perl-Quelltextes (auch für ähnliche Sprachen wie C, C++ oder PHP einsetzbar). Es besteht aus zwei Unterprogrammen. Das erste, `process()`, erzeugt die Referenzliste, und das zweite, `result()`, gibt die Liste im HTML-Format aus, sodass sie mit dem Browser gelesen und ausgedruckt werden kann. Zur Speicherung dient ein Hash, dessen Keys aus den gefundenen Wörtern gebildet werden. Der Inhalt besteht aus

den Zeilennummern der Fundstellen, jeweils durch ein Leerzeichen getrennt. Ein- und Ausgabedatei werden beim Aufruf über die Kommandozeile übergeben.

```
use strict;

# Parameter von der Kommandozeile lesen,
# Abbruch falls keine angegeben wurden
die "Usage: xref <source file> <output file> ...\n" unless ($#ARGV >= 1);
my $source = shift  @ARGV;
my $outfile = shift @ARGV;

my %xref = ();

process($source);
result($outfile);

# Ergebnisse als HTML ausgeben
sub result
  {
  my $outfile = shift;
  my $old = '';
  my $num = '';
  my $key = '';
  my $first = 0;
  my @numlist = ();

  open(OUT, ">$outfile") || die "Failed to create $outfile: $!";
  print OUT qq~
  <HTML>
  <HEAD>
   <TITLE>Cross Reference Index</TITLE>
  </HEAD>
  <BODY>
  <H1>Cross Reference Index</H1>
  <DL>
  ~;

  foreach $key (sort keys %xref)
    { # Begriff ausgeben
    print OUT "\n<DT>$key<DD>\n";
    @numlist = split(/\s/, $xref{$key});
    $old = ''; $first = 1;
    foreach $num (sort {$a <=> $b} @numlist)
      { # sortierte Zeilennummern ausgeben
      if ($num ne $old)
        {
        $old = $num;
        if ($first) { print OUT "$num"; $first = 0; }
        else { print OUT ", $num"; }
        }
      }
    }
  print OUT qq~

  </DL>
  </BODY>
  </HTML>
```

```
    ~;
    close OUT;
    }

  sub process
    {
    my $source = shift;
    my $line = '';
    my @match = ();
    my @words = ();
    my $word = '';
    my $lineno = 0;

    open(SRC, $source) || die "Failed to open $source: $!\n";
    while ($line = <SRC>)
      {
      # Zeile bearbeiten (in Worte zerlegen)
      chomp($line);
      $lineno++;
      # Leerzeichen am Ende entfernen
      $line =~ s/\s*$//;
      # Sonderbehandlung von '$#': $#var --> $var
      $line =~ s/\$\#/\$/g;
      # Kommentare bearbeiten (Beginn mit '#')
      $line =~ s/\#.*$//;
      # Strings in Gaensefuesschen (") bearbeiten
      while (@match = $line =~ m|([^\"]*)(\"[^\"]*\")(.*)|)
        { $line = $match[0] . $match[2]; }
      # Strings in einfachen Anfuehrungszeichen (') bearbeiten
      while (@match = $line =~ m|([^\']*)(\'[^\']*\')(.*)|)
        { $line = $match[0] . $match[2]; }
      # Alle nicht benoetigten Zeichen entfernen
      $line =~ s/[^\$\%\@\#A-Za-z0-9]/ /g;
      # Mehrfache Leerzeichen durch eines ersetzen
      $line =~ s/ +/ /g;
      # Fuehrende Leerzeichen entfernen
      $line =~ s/^\s*//;
      # Ist noch was uebrig fuer die Indizierung?
      next unless ($line);
      # Worte einzeln bearbeiten
      @words = split(/\s/, $line);
      foreach $word (@words)
        { $xref{$word} .= $lineno . " " if ($word =~ /^[\$\%\@\#A-Za-z]/); }
      }
    close SRC;
    }
```

6.9 Perl-Besonderheiten

Seit der Version 5.6 erlaubt es Perl, reguläre Ausdrücke übersichtlicher zu schreiben, indem sie auch als Variable übergeben können. Beispielsweise könnte die Stadtangabe einer Adresse folgendermaßen definiert werden:

```
my $plz   = '[0-9]{5}';
my $space = '[ \t]+';
my $ort   = '\w+';

if (m/$plz$space$ort/) ...;
```

Weil die Apostrophe bzw. `q//` nicht immer das Gewünschte erzeugen, gibt es extra für reguläre Ausdrücke `qr//`. Damit sähe das obige (etwas an den Haaren herbeigezogene) Beispiel folgendermaßen aus:

```
my $plz   = qr/[0-9]{5}/;
my $space = qr/[ \t]+/;
my $ort   = qr/\w+/;

if (m/$plz$space$ort/) ...;
```

Auf diese Weise lassen sich komplexe reguläre Ausdrücke in Stringvarablen ablegen, gut kommentieren oder sogar Perl-Konfigurationen modifizieren.

Dieser „quote regex Operator“ kann jedoch noch mehr. Er zitiert und compiliert das ihm übergebene Muster als regulären Ausdruck. Das mit `qr` erzeugte Compilat kann seinerseits wieder Teil eines regulären Ausdrucks sein, wie Sie oben gesehen haben. Die Begrenzer bei `qr` dürfen übrigens beliebige Zeichen sein. Es geht aber noch schlimmer, denn Perl akzeptiert auch einen rekursiven regulären Ausdruck der Form

```
my $muster   = qr/bla${muster}bla/;
```

Neben einigen anderen hat Perl auch noch ein paar Erweiterungen regulärer Ausdrücke auf Lager, die sehr komplexe Funktionen möglich machen:

`(?:muster)` Der Ausdruck in der Klammer wird wie gewohnt behandelt, es wird aber keine Rückwärtsreferenz (`\n` bzw. `$n`) erzeugt.

`(?=muster)` Betrachtet das Vorhandensein eines nachfolgenden Musters. So wird beispielsweise durch `/\w+(?=\t)/` ein Wort gefunden, das von einem Tabulatorzeichen gefolgt wird. Das Tab ist jedoch nicht in `$&` enthalten.

`(?!muster)` Betrachtet das Fehlen eines nachfolgenden Musters. So wird beispielsweise durch `/\w+(?!\t)/` ein Wort gefunden, das *nicht* von einem Tabulatorzeichen gefolgt wird.

`(?<=Muster)` Betrachtet das Vorhandensein eines vorhergehenden Musters. So wird beispielsweise durch `/(?<=\t)\w+/` ein Wort gefunden, das einem Tabulatorzeichen nachfolgt. Das Tab ist jedoch nicht in `$&` enthalten.

`(?<!muster)` Betrachtet das Fehlen eines vorhergehenden Musters. So wird beispielsweise durch `/(?<!\t)\w+/` ein Wort gefunden, ddem *kein* Tabulatorzeichen vorausgeht.

`(?-xism:muster)` Erlaubt das Setzen der Modifizierer „x“, „i“, „s“ oder „m“ im Ausdruck. Ein gegebenenfalls gesetzter Modifizierer kann durch das vorangestellte Minuszeichen gelöscht werden. So wird z. B. durch `(?i:Muster)` das Ignorieren der Groß- und Kleinschreibung eingeschaltet.

6.10 Regex-Helfer

Wer viel mit komplexen regulären Ausdrücken arbeitet, ist sicher für Hilfe dankbar. Diese kommt in Form von Modulen, die Ihnen die Zusammenstellung der Ausdrücke vereinfachen, aber auch von Modulen, die bei der Analyse von regulären Ausdrücken helfen.

Das erste Modul, `Regexp::Common` bietet eine Sammlung von gebräuchlichen regulären Ausdrücken, die Matches für IP-Adressen, Real- oder Integer-Zahlen und anderes enthalten. Der Vorteil dieser Sammlung gegenüber selbst geschriebenen Ausdrücken liegt darin, dass die Ausdrücke im Modul getestet und stabil sind. Die Verwendung von `Regexp::Common` macht keine Schwierigkeiten. Die regulären Ausdrücke sind in einem hierarchisch geordneten Hash abgelegt. So befinden sich beispielsweise unterhalb von `$RE{num}` alle Ausdrücke, die Zahlen bearbeiten. Mit `$RE{num}{int}` testet man dann ganze Zahlen. Das folgende Beispiel fischt ganze Zahlen aus den Daten heraus:

```
use strict;
use warnings;
use Regexp::Common;

my @zahlen = qw(1.05 0.534 3 15 12e12);

for my $i (@zahlen)
  {
  print "$i ist Integer\n" if( $i =~ /^$RE{num}{int}$/ );
  }
```

Mit dem Modul `Regexp::Assemble` können mehrere reguläre Ausdrück OR-verknüpft werden. Das Modul reiht dabei die übergebenen Teile nicht einfach aneinander, sondern versucht, Schnittmengen zu finden und das Resultat als möglichst kompakten Ausdruck darzustellen. Im folgenden Beispiel wird überprüft, ob ein Ausdruck die Worte „Perl“ oder „Paul“ enthält:

```
use strict;
use warnings;
use Regexp::Assemble;

my @words = qw(Perl Paul);
my $string = "Perl ist - wie so oft - ueberraschend";
my $re = Regexp::Assemble->new;
$re->add($_) for @words;
print "Gefunden!\n" if $string =~ /$re/;
```

Der reguläre Ausdruck, der von `Regexp::Assemble` im Listing oben erzeugt wird, lautet `(?-xism:P(?:er|au)l)`. Das Modul erkennt jedoch keine Zeichenbereiche. So produziert es beispielsweise `(?-xism:[123456789])` anstelle des kompakteren Ausdrucks `(?-xism:[1-9])`. Beide Module bieten relativ viele Features, die in deren Dokumentation ausführlich geschildert werden.

Das dritte Modul, das ich Ihnen vorstelle, hilft beim Analysieren dessen, was ein regulärer Ausdruck eigentlich bewirkt. `YAPE::Regexp::Explain` hat nur nur eine Funktion bzw. Methode. Es zerlegt den regulären Ausdruck und erläutert jeden Teil ausführlich. Sehen Sie sich dazu das folgende Programm an:

```
use strict;
use YAPE::Regex::Explain;
my $re = '\(([^%(]*?)\)';
print YAPE::Regex::Explain->new($re)->explain;
```

Es produziert folgende Ausgabe. Durch die Einrückung der Erklärungen ist gut zu sehen, zu welchem Klammerpaar jeder Teil des Ausdrucks gehört:

```
The regular expression:
(?-imsx:\(([^%(]*?)\))

matches as follows:

NODE             EXPLANATION
----------------------------------------------------------------
(?-imsx:    group, but do not capture (case-sensitive)
            (with ^ and $ matching normally) (with . not
            matching \n) (matching whitespace and  # normally):
----------------------------------------------------------------
   \(          '('
----------------------------------------------------------------
                ( group and capture to \1:
----------------------------------------------------------------
     [^%(]*?    any character except: '%(' (0 or more
                 times (matching the least amount
                 possible))
----------------------------------------------------------------
    )           end of \1
----------------------------------------------------------------
    \)          ')'
----------------------------------------------------------------
    )           end of grouping
```

Eine grafische Ausgabe kann mit dem Modul `GraphViz::Regex` von Leon Brocard erzeugt werden, auf das ich im Grafikkapitel 18 näher eingehen werde.

Abschließend noch einige oft verwendete reguläre Ausdrücke:

1. Die ersten beiden Worte vertauschen:
   ```
   s/(\S+)(\s+)(\S+)/$3$2$1/
   ```
2. Schlüssel = Wert:
   ```
   m/(\w+)\s*=\s*(.*)\s*$/ # $1 $2
   ```
3. Datum der Form TT/MM/JJ HH:MM:SS:
   ```
   m|(\d+)/(\d+)/(\d+) (\d+):(\d+):(\d+)|
   ```
4. Hexadezimalwerte der Form %xx in Zeichen umsetzen:
   ```
   s/%([0-9A-Fa-f][0-9A-Fa-f])/chr(hex($1))/ge
   ```
5. Leerzeichen am Anfang und Ende löschen:
   ```
   s/^\s+//;
   s/\s+$//;
   ```
6. Zahlen aus einem String extrahieren:
   ```
   @nums = m/(\d+\.?\d*|\.\d+)/g;
   ```
7. Links in HTML finden:
   ```
   @links = m/<A[^>]+?HREF\s*=\s*["']?([^'" >]+?)[ '"]?>/sig;
   ```

Tipps zum Erstellen regulärer Ausdrücke

- Untersuchen Sie die zu bearbeitenden Daten. Versuchen Sie, ein Gefühl für die möglichen Muster zu bekommen, und versuchen Sie, die Regeln erst einmal tabellarisch zu erfassen, bevor Sie sich an den regulären Ausdruck machen.
- Verwenden Sie notfalls mehrere reguläre Ausdrücke nacheinander. Manchmal ist es einfacher, die Aufgabe in mehreren Teilaufgaben zu zerlegen.
- Vergessen Sie nicht die Funktion `split()`. Manches lässt sich leichter damit lösen als mit einfachen Mustervergleichen. Das Gleiche gilt für `substr()`.
- Verwenden Sie Klammern, um benötigte Inhalte herauszuziehen.
- Verwenden Sie negierte Zeichenklassen anstelle von Quantifizierern. Denken Sie an die Gierigkeit der Quantifizierer.
- Denken Sie daran, dass * für „Null oder mehr“ steht. Selbst für fehlende Elemente liefert der Ausdruck gegebenenfalls „wahr“.
- Denken Sie an das Alternationszeichen (|). Es kann für komplexe Muster mit mehreren Alternativen praktisch sein.
- Eventuell ist es besser, die Aufgabe ohne einen regulären Ausdruck zu lösen. Reguläre Ausdrücke sind sehr leistungsstark, aber auf Kosten der Performance.
- Berücksichtigen Sie auch den Fall, dass der bearbeitete String leer oder undefiniert ist.
- Tasten und testen Sie sich langsam an die Lösung heran.