vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen
nächstes Kapitel

Pattern Matching mit regulären Ausdrücken

Sie haben also den Stoff bis hierher gelesen und gelernt? Erliegen Sie jetzt allerdings nicht dem Glauben, dass Sie damit schon eine gute Basis für die Perl-Programmierung haben, nur weil Ihnen die meisten Konzepte, die auch vielen anderen Sprachen gemein sind, bereits begegnet sind. Denn wenn Sie dieses Buch vor dem heutigen Kapitel zur Seite legen und mit dem bisher Erlernten versuchen, Perl zu nutzen, entgeht Ihnen einer der leistungsfähigsten und flexibelsten Aspekte von Perl - Pattern Matching mit Hilfe regulärer Ausdrücke. Pattern Matching ist mehr als nur die Suche nach einer Zeichenfolge in Ihren Daten; es ist eine Art, Daten zu sehen und zu verarbeiten, die unglaublich effizient ist und sich erstaunlich leicht programmieren läßt. Perl zu lernen, ohne Bekanntschaft mit regulären Ausdrücken zu machen, ist wie Snowboard fahren, ohne mit Schnee in Berührung zu kommen. In anderen Worten, hören Sie hier noch nicht auf - der gute Teil kommt erst.

Heute beschäftigen wir uns eingehend mit regulären Ausdrücken, warum sie so nützlich sind, wie sie aufgebaut werden und wie sie funktionieren. Morgen setzen wir dann die Diskussion fort und kommen zu den fortschrittlicheren Anwendungen von regulären Ausdrücken. Heute jedoch lernen Sie:

- was sich hinter Pattern Matching und regulären Ausdrücken verbirgt und warum sie so nützlich sind
- wie man einfache reguläre Ausdrücke mit Einzelzeichen-Suchläufen und Pattern Matching-Operatoren aufbaut
- wie man mit Zeichengruppen sucht
- wie man mehrere Vorkommen eines Zeichens findet
- wie man Suchmuster in Bedingungen und Schleifen verwendet

Sinn und Zweck des Pattern Matching

Pattern Matching ist eine Technik, mit der ein String, bestehend aus Text oder binären Daten, nach einer Zeichenfolge durchsucht wird. Die Zeichenfolge wird dabei in Form eines speziellen Suchmusters angegeben. Wenn Sie zum Beispiel in Ihrer Textverarbeitung mit dem Suchen-Befehl nach einem Zeichenstring in einer Datei suchen oder sich im Web einer Suchmaschine bedienen, um etwas zu finden, setzen Sie bereits eine vereinfachte Form des Pattern Matching ein: Ihr Kriterium lautet »suche diese Zeichen«. In oben genannten Umgebungen können Sie Ihre Kriterien üblicherweise Ihren Bedürfnissen anpassen, zum Beispiel können Sie nach diesem oder jenem suchen oder nach dem und dem, aber nicht nach jenem, Sie können nach ganzen Wörtern suchen oder nur nach den Wörtern, die einen Schriftgrad von 12 haben und unterstrichen sind. Pattern Matching in Perl ist noch weitaus komplizierter. Hier können Sie nämlich hochspezielle Kombinationen an Suchkriterien definieren, und das mit unwahrscheinlich wenig Code, wobei Sie eine spezielle Minisprache zur Musterdefinition, die sogenannten regulären Ausdrücke, verwenden.

Programmen übernommen hat, wurden auch hier leichte Änderungen und viele zusätzliche Ergänzungen vorgenommen. Wenn Sie bereits mit regulären Ausdrücken vertraut sind, werden Sie keine Schwierigkeiten mit den regulären Ausdrücken in Perl haben, da Sie in der überwiegenden Zahl der Fälle die gleichen Regeln anwenden können (auch wenn es einige Fallen gibt, vor denen man sich hüten muss, besonders wenn Sie vorher hauptsächlich mit komplizierten regulären Ausdrücken gearbeitet haben).

> *Der Begriff **regulärer Ausdruck** mag auf den ersten Blick etwas unsinnig erscheinen. Erstens handelt es sich dabei nicht um richtige Ausdrücke, und zweitens läßt sich nur schwer erklären, was so regulär daran ist. Doch Sie sollten sich nicht allzu lange mit dem Namen aufhalten. Der Begriff **regulärer Ausdruck** entstammt der Mathematik und beschreibt die Sprache, mit der Sie Muster für das Pattern Matching in Perl schreiben.*

Die obigen Beispiele mit der Suchmaschine oder dem Suchen-Befehl sollten Ihnen zeigen, was mit Pattern Matching alles möglich ist. Sie sollten jetzt aber nicht schlußfolgern, dass man Pattern Matching nur für das Suchen nach Textstellen verwenden kann. Der Leistungsumfang der regulären Ausdrücke in Perl umfaßt:

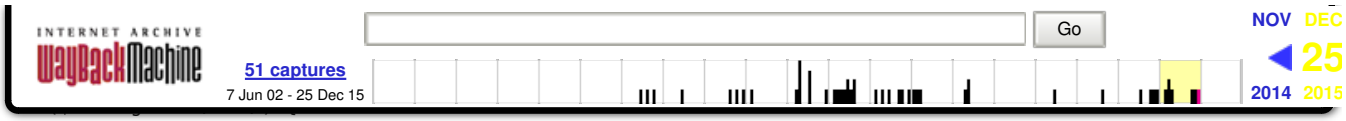
- Eingabe-Validierung - es wird sichergestellt, dass der Benutzer die von Ihnen gewünschten Daten eingegeben hat.
- Prüfung, ob die Eingabe in dem korrekten spezifischen Format erfolgt ist, zum Beispiel ob E-Mail-Adressen aus den richtigen Komponenten bestehen.
- Herausziehen bestimmter Teile einer Datei, die einem bestimmten Kriterium entsprechen (Sie könnten zum Beispiel alle Überschriften extrahieren, um ein Inhaltsverzeichnis zu erstellen, oder alle Links einer HTML-Datei).
- Zerlegung eines Strings auf der Basis bestimmter Trennzeichen-Felder (und oft komplizierter verschachtelter Trennzeichen-Felder).
- Feststellen von Unregelmäßigkeiten in einer Datenmenge - zum Beispiel mehrere Leerzeichen, wo sie nicht hingehören, doppelte Wörter, Fehler in der Formatierung.
- Das Zählen der Vorkommen eines Musters in einem String.
- Suchen&Ersetzen-Operationen - einen String suchen, der mit einem vorgegebenen Muster übereinstimmt, und ihn durch einen anderen String ersetzen.

Dies ist natürlich nur eine kleine Liste der Möglichkeiten - Sie können reguläre Ausdrücke in Perl für eine Vielzahl von Aufgaben einsetzen. Allgemein läßt sich sagen, dass Sie für eine Aufgabe, bei der Sie einen String oder einen Text durchgehen, am besten Perl mit seinen regulären Ausdrücken einsetzen. Viele der Operationen zum Suchen von Strings, die Sie gestern kennengelernt haben, lassen sich mit Mustern viel besser lösen.

Pattern-Matching-Operatoren und -Ausdrücke

Beim Pattern Matching legen Sie zuerst einmal fest, was Sie suchen wollen, dann schreiben Sie einen regulären Ausdruck für die Suche und setzen anschließend das Muster in einer Situation ein, in der das Ergebnis der Suche (gefunden oder nicht gefunden) von Nutzen ist. Wie wir es schon von Perl gewohnt sind, ist die Art und Weise, wie Sie Muster einsetzen, davon abhängig, wo und in welchem Kontext Sie das Muster verwenden.

Lassen Sie uns mit einem ziemlich einfachen Beispiel beginnen - Muster in einem Booleschen skalaren Kontext, bei dem der Ausdruck ***wahr*** zurückliefert, wenn der String das Muster enthält.




```
   # tue etwas ...
}
```

Der Test der `if`-Anweisung besagt: Liefere ***wahr*** zurück, wenn der String in der Variablen `$string` das Muster `foo` enthält. Beachten Sie, dass es sich bei dem Operator `=~` nicht um einen Zuweisungsoperator handelt, auch wenn er so aussieht. `=~` wird ausschließlich für den Mustervergleich verwendet und bedeutet: »Suche das Muster auf der rechten Seite im String auf der linken Seite.« Manchmal wird dieser Operator auch der ***Binding***-Operator genannt.

Das Suchmuster selbst steht zwischen den Slash-Zeichen von `m//`. Dieses spezielle Muster ist eines der einfachsten, das Sie erzeugen können, da es nur aus drei Sonderzeichen in Folge besteht (später werden Sie noch erfahren, was eigentlich eine Übereinstimmung (englisch ***match***) ist und was nicht). Ein Muster könnte auch folgendermaßen aussehen: `m/.*\d+/` oder `m/^[+-]?\d+\.?\d*$/` oder aus einem sonstwie unverständlichen Satz an Zeichen bestehen. Aber bitte keine Panik -- ich werde Ihnen in Bälde zeigen, wie man diese Muster dechiffriert.

Für diese Art von Muster ist das `m` optional und kann weggelassen werden (was in der Regel auch geschieht). Wenn Sie den Inhalt der Standardvariablen `$_` durchsuchen, können Sie außerdem die Variable und den Operator `=~` fortlassen. Sie werden in Perl häufig auf Kurzversionen wie die folgende stoßen:

```
if (/^\d+/) { # ...
```

Die ausgeschriebene Langversion dazu wäre:

```
if ($_ =~ m/^\d+/)  { # ...
```

Einen einfachen Fall dieser Art haben Sie bereits gestern im Zusammenhang mit der `grep`-Funktion kennengelernt, die mit Hilfe von Mustern einen String-Abschnitt innerhalb des Listenelements `$_` sucht.

```
@foodinge = grep /foo/, @strings;
```

Diese Zeile wiederum entspricht der ausgeschriebenen Form:

```
@foodinge = grep { $_ =~ /foo/ } @strings;
```

Im Laufe unserer heutigen Lektion werden Sie mehrere Möglichkeiten kennenlernen, Muster in unterschiedlichen Kontexten und aus verschiedenen Gründen einzusetzen. Dabei liegt die Hauptarbeit im Erlernen der Syntax der regulären Ausdrücke. Lassen Sie uns also mit diesem Aspekt beginnen.

Einfache Muster

Beginnen wir mit einigen der einfachsten und grundlegendsten Muster, die Sie erzeugen können: Muster, die mit bestimmten Zeichenfolgen übereinstimmen, Muster, die nur an bestimmten Positionen im String vorkommen können, und kombinierte Mustern, die mit Hilfe sogenannter Alternationen erstellt werden.

Zeichenfolgen

Zu einem der einfachsten Muster, für die Übereinstimmungen gesucht werden sollen, gehören die Zeichenfolgen:

```
/foo/
/dies oder das/
```

All diese Muster werden gefunden (match), wenn ihre Daten diese Zeichen in genau der angegebenen Reihenfolge enthalten. Dabei müssen alle Zeichen, auch die Leerzeichen deckungsgleich sein. Das Wort `oder` im zweiten Muster hat dabei keine besondere Bedeutung (es ist kein logisches `ODER`). Dieses Muster wird nur gefunden, wenn die Daten irgendwo den String `»dies oder das«` enthalten.

Beachten Sie, dass Zeichen in Mustern irgendwo in einem String gefunden werden können. Wortgrenzen werden nicht berücksichtigt - das Muster `/es/` wird sowohl in dem String `»es war einmal«` als auch in dem String `»diese Unterschiede sind keine«` gefunden. Das Muster `/es /` wird jedoch nur in dem ersten String gefunden, da es ein Leerzeichen enthält und die Zeichen `e`, `s` und das Leerzeichen nur in dem ersten String in genau der vorgegebenen Reihenfolge auftreten.

Beim Pattern Matching ist die Groß- und Kleinschreibung zu beachten: `/kazoo/` wird nur `kazoo` finden und nicht `Kazoo` oder `KAZOO`. Um bei einer Suche die Unterscheidung in Groß- und Kleinschreibung aufzuheben, müssen Sie nach dem Muster direkt die Option `i` eingeben (`i` steht für ***Groß- und Kleinschreibung ignorieren***):

```
/kazoo/i   # sucht nach allen Versionen in Groß- und Kleinschreibung
```

Sie können aber auch speziell Muster erzeugen, die entweder nach Groß- oder nach Kleinbuchstaben suchen. Darauf möchte ich aber erst im nächsten Abschnitt eingehen.

Sie können in den Mustern nahezu alle alphanumerischen Zeichen sowie die Escape-Darstellungen für binäre Daten (oktale und hexadezimale Escape-Zeichen) verwenden. Es gibt jedoch eine Reihe von Zeichen, die nur zusammen mit einem vorangehenden Escape-Zeichen in Mustern verwendet werden können. Diese Zeichen werden auch Metazeichen genannt und beziehen sich auf die Sprache des Pattern Matching und nicht auf das literale Zeichen.

Die folgenden Metazeichen sollten Sie in Ihren Mustern beachten:

```
^ $

. +

? *

{ (

) \

/ |

[
```

Wenn Sie irgendwann tatsächlich ein Metazeichen in einem String suchen wollen - zum Beispiel das Fragezeichen (?) - müssen Sie dem Metazeichen das Escape-Zeichen Backslash (`\`) voranstellen :

```
/\?/  # sucht nach einem Fragezeichen
```

Übereinstimmung an Wort- und Zeilengrenzen

Wenn Sie Muster zum Vergleichen einer Zeichenfolge erzeugen, können diese Zeichen irgendwo innerhalb des Strings vorkommen, und die Suche führt zum Erfolg. Manchmal jedoch möchten Sie einen Mustervergleich nur für Zeichen an einer bestimmten Position durchführen - zum Beispiel `/es/` nur als alleinstehendes Wort erkennen oder `/kazoo/` nur zu Beginn einer Zeile (das heißt dem Beginn eines Strings).

*einzelne Eingabezeile handelt, die aus einem einzigen String ohne Neue-Zeile-Zeichen besteht. Unter dieser Voraussetzung sind die Begriffe **String**, **Zeile** und **Daten** synonym. Morgen werden wir dann sehen, wie Muster mit mehrzeiligen Daten umgehen.*

Um Muster an einer bestimmten Position zu suchen, müssen Sie einen Musteranker verwenden. Der Musteranker für den Beginn eines Strings lautet `^`. Ein Beispiel:

```
/^kazoo/  # Übereinstimmung nur, wenn kazoo am Anfang einer Zeile steht
```

Der Musteranker für das Ende eines Strings lautet `$`:

```
/Ende$/   # Übereinstimmung nur, wenn Ende am Ende der Zeile steht
```

Auch hier sollten Sie sich das Muster als eine Folge von Elementen vorstellen, bei der jeder Teil des Musters mit den Daten, die durchsucht werden sollen, übereinstimmen muss. Die Pattern-Matching-Routinen in Perl starten den Suchvorgang an der Position, die direkt vor dem ersten Zeichen liegt und damit dem Zeichen `^` entspricht. Danach wird jedes Zeichen einzeln überprüft bis zum Ende der Zeile, das dem `$`- Zeichen entspricht. Folgt auf das Ende des Strings eine neue Zeile (angezeigt durch ein Neue-Zeile-Zeichen), befindet sich die durch `$` markierte Position direkt vor diesem Zeichen.

Anhand eines Beispiels möchte ich Ihnen zeigen, was passiert, wenn Sie versuchen, das Muster `/^foo/` in dem String »`sein oder nicht sein`« zu finden (was natürlich nicht klappt, uns aber trotzdem einen Versuch wert sein soll). Perl startet mit dem Anfang der Zeile, die dem `^`-Zeichen entspricht. Dieser Teil der Zeile ist somit ***wahr***. Danach erfolgt der Test des ersten Zeichens. Das Muster erwartet dort ein `f`, findet aber ein `s` vor. Deshalb wird der Mustervergleich abgebrochen und ***falsch*** zurückgeliefert.

Und was passiert, wenn Sie versuchen, obiges Muster auf den String »`fob`« anzuwenden? Der Mustervergleich führt etwas weiter - übereinstimmend sind der Anfang der Zeile, das `f` und das `o`, aber dann schlägt der Mustervergleich bei `b` fehl. Denken Sie auch daran, dass `/^foo/` in dem String »`-foo`« nicht gefunden wird - denn `foo` steht nicht direkt am Anfang der Zeile, wie es vom Muster erwartet wird. Übereinstimmung ist nur gegeben, wenn alle vier Elemente des Musters dem String entsprechen.

Hier einige interessante, aber knifflige Verwendungen von `^` und `$`. Können Sie raten, wann es bei diesen Beispielen zu einer Übereinstimmung kommt?

```
/^/
/^1$/
/^$/
```

Das erste Muster entspricht allen Strings, die am Anfang einer Zeile beginnen. Es wäre schon ein recht seltsamer String, der nicht mit dem Anfang einer Zeile beginnen würde. Deshalb entspricht dieses Muster allen Strings, sogar einem leeren String.

Das zweite Muster erwartet am Anfang einer Zeile die Zahl `1` und anschließend das Ende der Zeile. Eine Übereinstimmung gibt es also nur, wenn der String aus einer `1` besteht, wirklich nur aus einer `1` - dazu gehören weder »`123`« noch »`foo 1`« und auch nicht »1«.

Das dritte Muster geht davon aus, dass auf den Beginn der Zeile direkt das Ende der Zeile folgt - das heißt, dass es keine eigentlichen Daten gibt. Dieses Muster sucht nach einer leeren Zeile. Denken Sie jedoch daran, dass `$` direkt vor dem Zeichen für »Neue Zeile« steht. Deshalb wird sowohl »« als auch »`\n`« gefunden.

liegt. Eine Wortgrenze wird durch das Escape-Zeichen[1] `\b` angezeigt. Mit dem Muster `/\bes\b` finden Sie nur das ganze Wort `es` im String. Vorkommen, bei denen die Zeichen `e` und `s` in der Mitte eines Wortes stehen (wie in »Vergessenheit«) werden übergangen. Sie können mit `\b` sowohl auf den Anfang als auch auf das Ende eines Wortes Bezug nehmen. Das Muster `/ \bes/` zum Beispiel führt sowohl in dem String »`es war einmal`« als auch in »`die Situation eskaliert`« oder sogar »`so sei es!`« zu einer Übereinstimmung, aber nicht in »`dieses Beispiel`« oder »`in Vergessenheit geraten`«.

Sie können auch nach einem Muster suchen, das nicht an einer Wortgrenze steht. Dazu gibt es das `\B`. Demzufolge wird das Muster `/\Bes/` nur gefunden, wenn die Zeichen `e` und `s` innerhalb eines Wortes und nicht zu Beginn stehen.

Alternativen vergleichen

Manchmal werden Sie nach mehr als einem Muster im gleichen String suchen und dann prüfen wollen, ob alle Muster oder überhaupt eines davon gefunden wurde. Sie könnten dieses Problem dadurch lösen, dass Sie mehrere Pattern-Matching- Ausdrükken mit Hilfe von Perls logischen regulären Ausdrücken für das Boolesche UND (`&&` oder `and`) und ODER (`||` oder `or`) kombinieren, beispielsweise zu:

```
if (($in =~ /dies/) || ($in =~ /das/)) { ...
```

Wenn der durchsuchte String `/dies/` oder `/das/` enthält, liefert der ganze Test ***wahr*** zurück.

Für die ODER-Suche (vergleiche dies oder das Muster - eine Übereinstimmung reicht) gibt es AUCH ein Metazeichen, das Sie in den regulären Audrücken verwenden können: das Pipe-Zeichen (`|`). So könnte man den langen `if`-Test aus dem obigen Beispiel auch folgendermaßen schreiben:

```
if ($in =~ /dies|das/) { ...
```

Die Verwendung des `|`-Zeichens innerhalb eines Musters wird auch als ***Alternation*** bezeichnet, da es Ihnen erlaubt, verschiedene Muster zu vergleichen. Für ein solches Muster wird der Wert ***wahr*** zurückgegeben, wenn eines der Muster übereinstimmt.

Alle Ankerzeichen, die Sie mit einem Alternationszeichen verwenden, beziehen sich nur auf das Muster, das auf der gleichen Seite wie das Pipe-Zeichen steht. So bedeutet zum Beispiel das Muster `/^dies|das/`, dass das »`dies` am Anfang der Zeile« oder das »`das` irgendwo« zu suchen ist und nicht, dass »`dies`« oder »`das`« am Anfang der Zeile zu stehen hat. Wenn Sie diese letztgenannte Mustervariante wünschen, könnten Sie `/^dies|^das/` schreiben, die elegantere Lösung wäre aber, Ihre Muster in Klammern zu setzen:

```
/^(dies|das)/
```

Bei diesem Muster überprüft Perl zuerst den Anfang der Zeile und testet dann alle einzelnen Zeichen von »`dies`«. Gibt es dabei keine Übereinstimmung, geht Perl wieder zurück zum Anfang der Zeile und versucht, für »`das`« eine Übereinstimmung zu finden. Bei dem Muster `/^dies|das/` wird zuerst versucht, eine Übereinstimmung für alles, was auf der linken Seite der Alternation steht (Anfang der Zeile gefolgt von »`dies`«), zu finden. Ist der Versuch fehlgeschlagen, geht Perl zurück und durchsucht den ganzen String nach »`das`«.

Eine noch bessere Lösung wäre es, nur die Elemente zusammenzufassen, die sich im Muster unterscheiden, das heißt nicht nur das `^`-Zeichen für den Anfang der Zeile, sondern auch das `d`-Zeichen:

```
/^d(ies|as)/
```

aufwenden muss, um so besser.

Sie können alle Arten von Alternativen in einem Muster zusammenfassen. So sucht zum Beispiel das Muster `/(1.|2.|3.|4.) Mal/` nach »`1. Mal`«, »`2. Mal`« und so weiter - solange wie die Daten eine der Alternativen innerhalb der Klammern sowie den String » `Mal`« enthalten (man beachte das Leerzeichen).

Mit Zeichengruppen vergleichen

So weit, so gut? Die bisher erstellten regulären Ausdrücke werden wohl auch Sie nicht als besonders komplex einstufen, vor allem nicht, wenn Sie jedes Muster - Zeichen um Zeichen, Alternative um Alternative unter Berücksichtigung etwaiger Zusammenfassungen - aus der Sicht von Perl betrachten. In diesem Abschnitt möchte ich auf einige der Kurzformen der regulären Ausdrücke zu sprechen kommen, die Sie zum Beschreiben und Zusammenfassen von verschiedenen Arten von Zeichen verwenden können.

Zeichenklassen

Angenommen Sie haben einen String und wollten eines von vier Wörtern in diesem String suchen: `Aal`, `Mal`, `Tal` und `Wal`. Sie könnten dazu folgendes eingeben:

```
/Aal|Mal|Tal|Wal/
```

Das funktioniert natürlich. Perl würde den ganzen String nach `Aal` durchsuchen, dann nach `Mal`, dann nach `Tal` und so weiter. Es geht aber auch kürzer - sowohl für Sie bei der Eingabe als auch für Perl bei der Ausführung. Dazu müssen Sie die Zeichen so zusammenfassen, dass Sie das `al` nicht jedesmal separat anführen:

```
/(A|M|T|W)al/
```

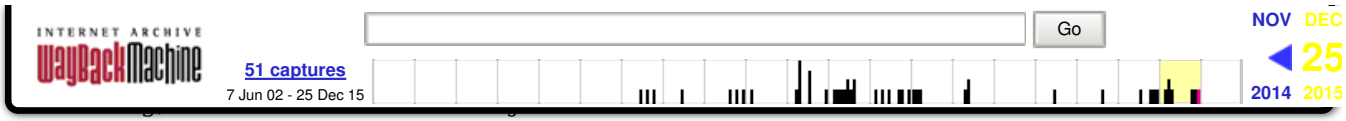
Bei diesem Muster durchsucht Perl den gesamten String nach `A`, `M`, `T` oder `W` und erst, wenn es eines dieser Zeichen findet, versucht es, eine Übereinstimmung mit `al` zu erzielen. Das ist wesentlich effizienter!

Diese Art von Muster - bei der eine große Anzahl an Alternativen einzelner Zeichen besteht - ist derart häufig, dass es hierfür in den regulären Ausdrücken eine eigene Syntax gibt. Der Satz an alternativen Zeichen wird dabei als ***Zeichenklasse*** bezeichnet, die Sie in eckige Klammern setzen. Mit Hilfe einer Zeichenklasse würde unser Beispiel von oben wie folgt geschrieben:

```
/[AMTW]al/
```

Damit sparen Sie eine Reihe von Zeichen, und es ist darüber hinaus auch noch leichter zu lesen. Dieses Beispiel wird von Perl genauso gelesen und abgearbeitet wie das mit den Alternationszeichen: Erst werden die Zeichen innerhalb der Zeichenklasse abgeglichen und dann alle Zeichen außerhalb davon.

Die Zeichen, die innerhalb einer Zeichenklasse auftauchen können, unterliegen anderen Regeln als die, die außerhalb stehen. Die meisten Metazeichen werden innerhalb einer Zeichenklasse zu einem ganz gewöhnlichen Zeichen (abgesehen von dem Zeichen für die schließende eckige Klammer, die aus verständlichen Gründen mit einem Escape-Zeichen versehen sein muss: dem ***Caret*** (^), das nicht an erster Stelle stehen darf, oder einem Bindestrich, dem innerhalb einer Zeichenklasse eine besondere Bedeutung zukommt). Ein Beispiel zur Kontrolle der Interpunktion am Ende eines Satzes (Interpunktion nach einer Wortgrenze und vor zwei Leerzeichen) könnte wie folgt aussehen:



Bereiche

Angenommen Sie wollten nach allen Kleinbuchstaben von `a` bis `f` suchen (zum Beispiel in einer Hexadezimalzahl). Ihre Eingabe könnte lauten:

```
/[abcdef]/
```

Das sieht doch sehr nach einem Bereich aus? Sie können innerhalb von Zeichenklassen Bereiche definieren, verwenden dazu allerdings nicht den aus Tag 4 bekannten Bereichsoperator (..). Reguläre Ausdrücke verwenden für Bereiche den Bindestrich (weshalb Sie ihn auch mit einem Escape-Zeichen versehen müssen, wenn Sie konkret nach einem Bindestrich suchen). Ein Bereichsmuster für `a` bis `f` würde folgendermaßen aussehen:

```
/[a-f]/
```

Sie können als Bereich eine beliebige Anzahl von Zeichen oder Zahlen eingeben: `/[0-9]/`, `/[a-z]/` oder `/[A-Z]/`. Die Bereiche lassen sich sogar kombinieren. Dabei entspricht `/ [0-9a-z]/` der Langform `/[0123456789abcdefghijklmnopqrstuvwxyz]/`.

Negierte Zeichenklassen

Eckige Klammern definieren eine Klasse von zu vergleichenden Zeichen in einem Muster. Sie können aber auch einen Satz von Zeichen definieren, der nicht verglichen werden soll, eine sogenannte negierte Zeichenklasse. Dazu müssen Sie lediglich dafür Sorge tragen, dass das erste Zeichen in der Zeichenklasse ein Caret-Zeichen (^) ist. Das Beispiel für einen Mustervergleich, bei dem alles außer `A` oder `B` verglichen wird, lautet:

```
/[^AB]/
```

Beachten Sie, dass das Caret innerhalb einer Zeichenklasse eine andere Bedeutung hat als außerhalb. Innerhalb wird damit eine negierte Zeichenklasse definiert und außerhalb der Anfang einer Zeile.

Wenn Sie irgendwann einmal über eine Zeichenklasse nach dem Caret-Zeichen suchen wollen, können Sie das problemlos - Sie müssen jedoch sicherstellen, dass es nicht das erste Zeichen ist oder es mit einem Escape-Zeichen versehen (am besten verwenden Sie für beide Fälle ein Escape-Zeichen, damit Sie sich nicht so viele Regeln merken müssen):

```
/[\^?.%]/    #  sucht nach ^, ?, ., %
```

Sie sollten sich diese Syntax merken, da Sie wahrscheinlich ziemlich häufig von negierten Zeichenklassen in regulären Ausdrücken Gebrauch machen werden. Eine Feinheit gilt es jedoch zu beachten: Negierte Zeichenklassen negieren nicht den ganzen Wert des Musters. Wenn also `/[12]/` bedeutet: »Liefere ***wahr*** zurück, wenn die Daten die Zeichen 1 oder 2 enthalten«, bedeutet `/[^12]/` nicht: »Liefere ***wahr*** zurück, wenn die Daten weder 1 oder 2 enthalten«. Wäre dies der Fall, würde es zu einer Übereinstimmung kommen, auch wenn der untersuchte String leer wäre. Was negierte Zeichenklassen eigentlich aussagen, ist: »Suche nach allen Zeichen, die in dem Muster nicht aufgeführt sind.« Es muss also zumindest ein richtiges Zeichen gefunden werden, damit eine negierte Zeichenklasse greift.

Besondere Klassen

Sollten Bereichsangaben für Zeichenklassen Ihnen immer noch zuviel Tipparbeit sein, gibt es als Erleichterung einige besondere Klassen (auch negierte Zeichenklassen), die einen eigenen

müssen, Sie können sie so, wie sie in der untenstehenden Tabelle aufgeführt sind, verwenden.

In Tabelle 9.1 finden Sie eine Liste der Codes für die besonderen Zeichenklassen.

Code	Äquivalente Zeichenklasse	Bedeutung
\d	[0-9]	Alle Ziffern
\D	[^0-9]	Alle Zeichen außer Ziffern
\w	[0-9a-zA-z_]	Alle »Wortzeichen« (alphanumerische Zeichen und _)
\W	[^0-9a-zA-z_]	Alle Zeichen außer »Wortzeichen«
\s	[\t\n\r\f]	Whitespace-Zeichen (Leerzeichen, Tabulator, Neue Zeile, Wagenrücklauf, Blattvorschub)
\S	[^ \t\n\r\f]	Alle Zeichen außer Whitespace-Zeichen

Tabelle 9.1: ***Codes für Zeichenklassen***

Die Wortzeichen (`\w` und `\W`) bedürfen einer Erklärung. Warum sollte ein Unterstrich ein Wortzeichen sein, die anderen Interpunktionszeichen aber nicht? Um genau zu sein, haben Wortzeichen sehr wenig mit Wörtern zu tun. Es handelt sich dabei um alle zulässigen Zeichen für die Vergabe eines Variablennamens: Zahlen, Buchstaben und Unterstriche. Alle anderen Zeichen gehören nicht zu den Wortzeichen.

Sie können diese Zeichencodes überall dort verwenden, wo Sie einen besonderen Zeichentyp benötigen. Der Code `\d` bezieht sich zum Beispiel auf alle Ziffern. Mit `\d` können Sie ein Muster erzeugen, das drei Ziffern vergleicht (`/\d\d\d/`) oder drei Ziffern, ein Leerzeichen und sechs weitere Ziffern (`/\d\d\d \d\d\d\d\d\d/`) - eine Telefonnummer wie 089 121212. Wie es auch ohne die lästigen Wiederholungen geht, werden Sie bald erfahren, wenn wir zu den Quantifizierern kommen.

Mit . ein beliebiges Zeichen finden

Die am breitesten gefaßte Zeichenklasse, die Sie definieren können, vergleicht nach nur einem, dafür aber beliebigen Zeichen. Hierfür verwenden Sie das Punktzeichen (`.`). Das folgende Muster ergibt für alle Zeilen, die wirklich nur ein Zeichen enthalten, eine Übereinstimmung:

```
/^.$/
```

Eigentlich kommt der Punkt vor allem in Mustern mit Quantifizierern (die wir gleich behandeln werden) zum Einsatz. Sie können den Punkt aber auch nutzen, um zum Beispiel Felder einer bestimmten Länge anzuzeigen:

```
/^..:/
```

Dieses Muster führt nur zu einer Übereinstimmung, wenn die Zeile mit zwei Zeichen und einem Doppelpunkt beginnt.

Mehr zu dem Punktoperator nach dem folgenden kleinen Beispiel.

Ein Beispiel: Den Zahlenbuchstabierer optimieren

nachdem Sie jetzt einiges Grundwissen zu den regulären Ausdrücken erworben haben, lassen Sie uns eine Neufassung dieses Skripts erstellen, bei der die ganzen `if`-Anweisungen durch reguläre Ausdrücke ersetzt werden.

Und da wir gerade dabei sind, können wir auch den Teil des Zahlenbuchstabierers überarbeiten, der die Eingabe überprüft. Wir können, was die Eingabevalidierung betrifft, mit regulären Ausdrücken wesentlich mehr machen - bis hin zur Absurdität. Ja, wir werden in diesem Skript bei der Eingabevalidierung der Absurdität sogar sehr nahe kommen. Diese Version testet auf eine Reihe von möglichen Eingaben und gibt dabei verschiedene (manchmal recht sarkastische) Kommentare aus:

```
% zahlenbuchstabierer2.pl
Geben Sie die zu buchstabierende Zahl ein (0-9): foo
Sie können mich nicht täuschen.  Die Eingabe enthält Buchstaben.
Geben Sie die zu buchstabierende Zahl ein (0-9): 45foo
Sie können mich nicht täuschen.  Die Eingabe enthält Buchstaben.
Geben Sie die zu buchstabierende Zahl ein (0-9): ###
huh?  Das sieht *wirklich* nicht nach einer Zahl aus
Geben Sie die zu buchstabierende Zahl ein (0-9): -45
Das ist eine negative Zahl.  Bitte nur positive Zahlen!
Geben Sie die zu buchstabierende Zahl ein (0-9): 789
Zu groß!  0 bis 9 bitte.
Geben Sie die zu buchstabierende Zahl ein (0-9): 4
Danke!
Die Zahl 4 schreibt sich vier
Eine weitere Zahl versuchen (j/n)?: x
j oder n, bitte
Eine weitere Zahl versuchen (j/n)?: n
%
```

Anstatt Ihnen jetzt das Skript ganz zu zeigen und jede Zeile einzeln durchzugehen, möchte ich einen anderen Weg einschlagen: Ich zeigen Ihnen Teile der alten und der neuen `zahlenbuchstabierer`-Version, erläutere sie und bringe das Beispiel am Ende, so dass Sie sich dann einen Gesamteindruck verschaffen können.

Beginnen wir mit der Schleife, die eine Zahl als Eingabe erwartet. So sah die Schleife in der alten Version des Zahlenbuchstabierers aus:

```
while () {
       print 'Geben Sie die zu buchstabierende Zahl ein: ';
       chomp($num = <STDIN>);
       if ($num ne "0" && $num == 0) { # wenn $num ein String
            print "Keine Strings. Eine Zahl von 0 bis 9 bitte.\n";
            next;
        }
        if ($num > 9) { # wenn $num mehrstellige Zahl
            print "Zu hoch. 0 bis 9 bitte.\n";
            next;
        }
        if ($num < 0) { # wenn $num negative Zahl
            print "Keine negativen Zahlen. 0 bis 9 bitte.\n";
            next;
        }
        last;
    }
```

Wir können diese drei Tests in der Schleife ohne größere Probleme durch reguläre Ausdrücken, die sogar verständlicher sind, ersetzen - und wir können darüber hinaus auf noch schwierigere Sachen prüfen. Unsere neue Schleife prüft, ob die drei folgenden Hauptbedingungen gegeben sind:

- ob die Eingabe aus einer einzelnen Zahl, und damit meine ich nur einer Ziffer, besteht

Der zweite Test kann in mehrere kleinere Tests zerlegt werden, die dann auf alphabetische Zeichen, negative Zahlen (beginnen mit einem -), Fließkommazahlen (mit einem Dezimalpunkt) oder absolut abwegige Zeichen testen. Sehen Sie im folgenden die neue Version unserer Schleife, die zusätzlich von der Variablen `$_` Gebrauch macht, um uns etwas Tipparbeit bei den Tests des Mustervergleichs zu ersparen:[2]

```
1: while () {
2:     print 'Geben Sie die zu buchstabierende Zahl ein (0-9): ';
3:     chomp($_ = <STDIN>);
4:     if (/^\d$/) {  # korrekte Eingabe
5:         print "Danke!\n";
6:         last;
7:     } elsif (/^$/) {
8:         print "Sie haben nichts eingegeben.\n";
9:     } elsif (/\D/) { # keine Zahlen
10:        if (/[a-zA-z]/) { # Buchstaben
11:            print "Sie können mich nicht täuschen.  Die Eingabe enthält
                   Buchstaben.\n";
12:        } elsif (/^-\d/) { # negative Zahlen
13:            print "Das ist eine negative Zahl. Bitte nur positive
                   Zahlen!\n";
14:        } elsif (/\./) { # Dezimalzahlen
15:            print "Das sieht sehr nach einer Dezimalzahl aus.\n";
16:            print "Ich kann Dezimalzahlen nicht in Worten ausgeben.
                   Versuchen Sie eine neue Zahl.\n";
17:        } elsif (/[\W_]/) {  # andere Zeichen
18:            print "huh?  Das sieht *wirklich* nicht nach einer Zahl
                  aus\n";
19:        }
20:    } elsif ($_ > 9) {
21:        print "Zu groß!  0 bis 9, bitte.\n";
22:    }
23: }
```

Werfen wir jetzt, Zeile für Zeile, einen Blick auf die regulären Ausdrücke, damit Sie wissen, auf was getestet wird:

- Zeile 4: `/^\d$/`
  Dieses Muster prüft, ob die Eingabe aus einer einstelligen Zahl besteht - das heißt, ob die Eingabe genau unseren Erwartungen entspricht. Ich habe dieses Muster als erstes definiert, damit für den Fall, dass der Benutzer eine korrekte Eingabe vornimmt, nicht eine Menge Zeit damit vergeudet wird, die ganzen Optionen auf Korrektheit zu prüfen. Wird bereits hier eine Übereinstimmung festgestellt und die Eingabe war korrekt, kann die Schleife an dieser Stelle direkt mit `last` verlassen werden.
- Zeile 7: `/^$/`
  Wie Sie bereits im Abschnitt zum Mustervergleich an Grenzen gelernt haben, überprüft dieses Muster auf eine leere Zeile - die Sie immer dann erhalten, wenn bei der Eingabeaufforderung die Eingabetaste betätigt wird, ohne dass vorher etwas eingegeben wurde.
- Zeile 9: `/\D/`
  Dieser Zeichencode steht für die Zeichenklasse »alle Zeichen außer Zahlen«. Wenn bei der Eingabeaufforderung irgend etwas eingegeben wird, das keine Zahl ist, zum Beispiel eine Kombination von Zahlen und Buchstaben, nur Buchstaben oder Zeichen wie -, . oder `$` trifft dieses Muster zu. Anschließend wird in eine Reihe von Untertests verzweigt, um die Eingabe von speziellen nicht-numerischen Zeichen abzufangen.
- Zeile 10: `/[a-zA-Z]/`
  Dieser Zeichenklassenbereich sucht nach den Zeichen des Alphabets. Ich habe hier absichtlich auf die Verwendung des Codes `\w` verzichtet, da dieser den Unterstrich mit eingeschlossen hätte. Den jedoch möchte ich zusammen mit allen anderen Zeichen in einem

der Test auf negative Zahlen

- Zeile 14: `/\./`
  Bei einer Eingabe, die einen Punkt enthält, handelt es sich höchstwahrscheinlich um eine Fließkommazahl. Beachten Sie, dass der Punkt hier mit einem Escape- Zeichen versehen werden muss, um einen tatsächlich Punkt zu repräsentieren (ansonsten stellt der Punkt in Mustern ein Metazeichen dar).
- Zeile 17: `/[\W_]/`
  Hier verwenden wir eine Zeichenklasse für zwei Abfragen: alle Zeichen, die keine Wortzeichen sind (0-9, a-z, A-Z) sowie den Unterstrich. Dies ist unsere Auffanglösung für alle anderen Zeichen, die eventuell eingegeben wurden.
- Zeile 20: `hier kein Muster`
  Diese Zeile fängt alle Eingaben ab, bei denen es sich um eine Zahl handelt (und die nicht durch die meisten der vorherigen Tests abgefangen werden konnten), die jedoch aus mehr als einer Ziffer bestehen. Hier wird nur geprüft, ob die Zahl größer als 9 ist. Es gibt zwar auch für diesen Fall ein Muster, doch haben wir dieses noch nicht kennengelernt.

Der nächste Teil des alten `zahlenbuchstabierer`-Skripts bestand aus einem Satz von `if...elsif`-Schleifen, die den Eingabewert mit einem Zahlenstring verglichen. Mit Hilfe von regulären Ausdrücken, der Standardvariablen `$_` und logischen Ausdrücken in der Funktion als Bedingung, können wir folgende verschachtelte `if`-Anweisungen:

```
if ($num == 1) { print 'eins'; }
    elsif ($num == 2) { print 'zwei'; }
    elsif ($num == 3) { print 'drei'; }
    elsif ($num == 4) { print 'vier'; }
    # ... andere Zahlen aus Platzgründen fortgelassen
}
```

auf einen Satz von folgenden logischen Anweisungen reduzieren:

```
/1/ && print 'eins';
/2/ && print 'zwei';
/3/ && print 'drei';
/4/ && print 'vier';
# ... und so weiter
```

Cool, nicht wahr? Es erinnert stark an `switch`-Anweisungen und ist nachweislich leichter zu lesen.

Abschließend schreiben wir unsere kleine Ja-oder-nein-Schleife neu, die das gesamte Skript bei Bedarf wiederholt. Die alte Version lautete:

```
while () {
    print 'Eine weitere Zahl versuchen (j/n)?: ';
    chomp ($exit = <STDIN>);
    $exit = lc $exit;
    if ($exit ne 'j' && $exit ne 'n') {
        print "j oder n, bitte\n";
    }
    else { last; }
}
```

Eigentlich ist an dieser Version nicht allzuviel falsch, aber da wir uns in diesem Kapitel mit Pattern Matching beschäftigen, wollen wir auch hier Muster einsetzen:

```
while () {
        print 'Eine weitere Zahl versuchen (j/n)?: ';
        chomp ($exit = <STDIN>);
        $exit = lc $exit;
        if ($exit =~ /^[jn]/) {
            last;
        }
```

Beachten Sie die Unterschiede zwischen dieser Schleife und der Eingabeschleife. In der Eingabeschleife haben wir die Eingabe in der Variablen `$_` abgelegt, so dass wir in der `if`-Bedingung nur das Muster unterzubringen brauchten. Hier testen wir auf den String in der Variablen `$exit`, so dass wir den Operator `=~` verwenden müssen. Im Muster selbst testen wir, ob die Eingabe entweder `j` oder `n` lautete (J und N werden mit der Funktion `lc` zu Kleinbuchstaben konvertiert). Wenn ja, verlassen wir die Schleife und kehren zu der äußeren Schleife zurück, die das Skript, wenn nötig, erneut durchläuft.

In diesem Beispiel habe ich eine ganze Menge regulärer Ausdrücke verwendet und viele davon eigentlich unbegründet. An dieser Stelle möchte ich Sie darauf hinweisen, dass Sie nicht um jeden Preis reguläre Ausdrücke verwenden sollten, nur weil sie cool sind. Die Perl-Maschinerie der regulären Ausdrücke ist besonders leistungsfähig, wenn es um komplizierte Sachverhalte geht. Sie erzeugen damit jedoch nicht den effizientesten Code, wenn Sie sie für einfache Probleme einsetzen. In einem solchen Fall sind einfache Tests und `if`-Anweisungen oft schneller in der Ausführung als reguläre Ausdrücke. Wenn Sie also bei Ihrem Code auch auf die Effizienz achten wollen, sollten Sie sich das merken.

Listing 9.1 enthält den vollständigen Code für die neue Version von `zahlenbuchstabierer.pl`:

Listing 9.1: Das Skript zahlenbuchstabierer2.pl[3]

```
#!/usr/bin/perl -w
# Zahlenbuchstabierer:  gibt Zahlen in Worten aus
# einfache Version für einstellige Zahlen

$exit = "";  # ob das Skript verlassen werden soll oder nicht.

while ($exit ne "n") {

    while () {
        print 'Geben Sie die zu buchstabierende Zahl ein (0-9): ';
        chomp($_ = <STDIN>);
        if (/^\d$/) {
            print "Danke!\n";
            last;
        } elsif (/^$/) {
            print "Sie haben nichts eingegeben.\n";
        } elsif (/\D/) {        # Keine Zahlen
            if (/[a-zA-z]/) { # Buchstaben
                print "Sie können mich nicht täuschen. Die Eingabe enthält
                       Buchstaben.\n";
            } elsif (/^-\d/) { # negative Zahlen
                print "Das ist eine negative Zahl.  Bitte nur positive
                       Zahlen!\n";
            } elsif (/\./) { # Dezimalzahlen
                print "Das sieht sehr nach einer Dezimalzahl aus.\n";
                print "Ich kann Dezimalzahlen nicht in Worten ausgeben.
                       Versuchen Sie eine neue Zahl.\n";
            } elsif (/[\W_]/) {  # andere Zeichen
                print "huh?  Das sieht *wirklich* nicht nach einer Zahl
                       aus\n";
            }
        } elsif ($_ > 9) {
            print "Zu groß!  0 bis 9, bitte.\n";
        }
    }

    print "Die Zahl $_ schreibt sich ";
```

```
    /5/ && print 'fünf';
    /6/ && print 'sechs';
    /7/ && print 'sieben';
    /8/ && print 'acht';
    /9/ && print 'neun';
    /0/ && print 'null';
    print "\n";

    while () {
        print 'Eine weitere Zahl versuchen (j/n)?: ';
        chomp ($exit = <STDIN>);
        $exit = lc $exit;
        if ($exit =~ /^[yn]/) {
            last;
        }
        else {
            print "j oder n, bitte\n";
        }
    }
}
```

Mehrere Übereinstimmungen von Zeichen finden

Sind Sie bereit, weiterzumachen? Dann komme ich zum zweiten Teil der Syntax für reguläre Ausdrücke: den Quantifizierern. Im Gegensatz zu den Mustern, die Sie bisher kennengelernt haben und die sich auf einzelne Dinge beziehungsweise Gruppen von einzelnen Dingen beziehen, können Quantifizierer für mehrere Vorkommen - oder keine Vorkommen - stehen. Als Quantifizierer bezeichnet man bestimmte Metazeichen in regulären Ausdrücken, die angeben, wie oft ein Zeichen oder eine Zeichenfolge in dem Suchmuster vorkommen soll.

Die regulären Ausdrücke von Perl kennen drei Metazeichen für Quantifizierer: `?`, `*` und `+`. Die Metazeichen beziehen sich auf die Häufigkeit des Musters (ein Zeichen oder eine Zeichenfolge) auf das sie direkt folgen.

Optionale Zeichen mit ?

Lassen Sie uns mit dem `?` anfangen. Damit können Sie Sequenzen finden, in denen das direkt vor dem Fragezeichen stehende Zeichen enthalten sein oder fehlen kann. Betrachten wir zum Beispiel folgendes Muster:

```
/wah?r/
```

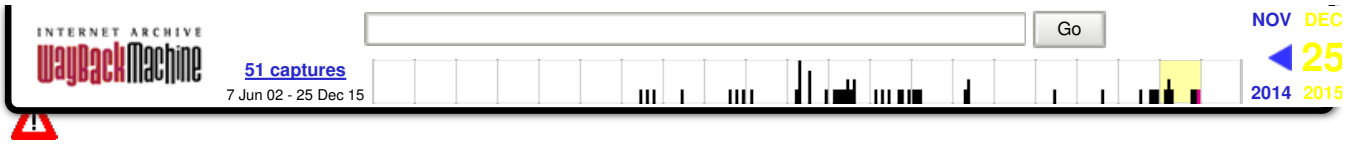
Das Fragezeichen in diesem Muster bezieht sich auf das direkt davorstehende Zeichen (`h`). Dieses Muster würde sowohl in dem String »`das ist nicht wahr`« als auch in dem String »`es war einmal`« eine Übereinstimmung finden, da sowohl `wahr` als auch `war` diesem Muster entsprechen. Der String, den Sie durchsuchen, muss ein `w`, ein `a` und ein `r` enthalten, das `h` hingegen ist optional.

Auch hier sollten Sie sich vergegenwärtigen, wie der String verarbeitet wird. Zuerst wird das `w` und dann das `a` überprüft. Dann folgt das dritte Zeichen. Ist es ein `h`, dann gibt es keine Probleme, und wir gehen im String und im Muster zum nächsten Zeichen über (das `r`). Ist es kein `h`, ist das auch kein Problem. Dann gehen wir im Muster ein Zeichen weiter und schauen, ob es hiermit eine Übereinstimmung gibt.

Mit Hilfe von Klammern können Sie Sequenzen von optionalen Zeichen zusammenstellen:

```
/Milch(bar)?/
```

Durch die Klammern wird die ganze Zeichensequenz (`bar`) optional, so dass dieses Muster sowohl



> *Da stellt sich die Frage, wozu man ein Muster wie dieses erzeugen sollte? Es scheint doch, als ob der (`bar`)-Teil dieses Musters ohne Bedeutung ist und der `/Milch/`-Teil - mit weniger Zeichen - genausogut zum Ziel führt. In diesen einfachen Fällen, in denen wir nur herauszufinden versuchen, ob etwas gefunden wird oder nicht, spielt es keine große Rolle. Morgen jedoch, wenn Sie lernen, wie Sie das gefundene Objekt extrahieren und damit komplexere Muster erstellen, wird die Unterscheidung deutlicher.*

Sie können das Fragezeichen auch zusammen mit Zeichenklassen verwenden:

```
/Punkt \d?/
```

Dieses Muster findet für die Strings »`Punkt 1`«, »`Punkt 9`« und so weiter genauso wie für den String »`Punkt `« (man beachte das Leerzeichen) eine Übereinstimmung. Jedes Zeichen der Zeichenklasse kann entweder einmal oder keinmal vorkommen, damit das Muster eine Übereinstimmung findet.

Mehrere optionale Zeichen mit *

Der zweite Quantifizierer ist der Multiplikator, das `*`-Zeichen, das ähnlich dem `?` funktioniert. Im Gegensatz zum Fragezeichen, das nur eine oder keine Instanz des vorangehenden Zeichens erlaubt, ist bei dem `*` keine oder eine beliebige Anzahl des Zeichens möglich. Betrachten wir folgendes Muster:

```
/xy?z/
```

In diesem Beispiel sind das x und das z erforderlich, das y aber kann beliebig oft oder auch gar nicht enthalten sein. Dieses Muster findet `xyz`, `xyyz`, `xyyyyyyyyyyz` oder auch nur `xz` ohne das y.

Wie bei dem `?` können Sie auch das `*` zusammen mit Zeichenfolgen oder Zeichenklassen verwenden. Eines der Hauptanwendungsbereiche ist die Kombination des `*`-Zeichens mit dem Punktzeichen. Damit drücken Sie aus, dass an dieser Position eine beliebige Anzahl von beliebigen Zeichen stehen kann:

```
/dies.*/
```

Mit diesem Muster finden Sie die Strings »`dieses`«, »`dies ist nicht mein Problem, kümmere Du dich drum`« oder einfache nur »`dies`« - zur Erinnerung, es muss kein Zeichen am Ende stehen, damit eine Übereinstimmung gefunden wird.

Ein häufiger Fehler ist, zu vergessen, dass `*` für »keine oder mehrere Instanzen« steht, und es wie folgt zu verwenden:

```
if (/^[0-9]*$/) {
    # soll Zahlen enthalten
}
```

Dieses Muster soll nur zu einer Übereinstimmung führen, wenn die Eingabe Zahlen enthält, und zwar nur Zahlen. Erreicht wird damit jedoch, dass dieses Muster zwar Zahlen wie »`7`«, »`1540`« und »`15443`« findet, aber auch leere Strings akzeptiert, da das `*`- Zeichen bedeutet, dass eine Übereinstimmung auch dann gegeben ist, wenn keine Zahl vorhanden ist. Normalerweise würden Sie das `+` statt dem `*` benutzen, wenn irgend etwas zumindest einmal vorhanden sein soll.

Beachten Sie, dass dieses Beispiel »finde keine oder mehrere Zahlen« nicht bedeutet, dass jeder String, der keine Zahlen aufweist, gefunden wird. So wird zum Beispiel der String »`Lederhosen`« zu

müssen Sie sehr genau bei der Angabe Ihrer Musterdefinitionen sein.

Mindestens eine Instanz mit +

Das Metazeichen `+` ist in seiner Funktionsweise mit dem Zeichen `*` identisch - bis auf einen kleinen Unterschied: Anstatt keine oder mehrere Instanzen des gegebenen Zeichens oder der Zeichenfolge zu prüfen, verlangt `+`, dass das Zeichen oder die Zeichenfolge mindestens einmal zu finden ist (eine oder mehrere Instanzen). Betrachten wir das obige Beispiel:

```
/xy+z/
```

Dieses Muster führt für folgende Strings zu einer Übereinstimmung: `xyz`, `xyyz`, `xyyyyyyz`, aber nicht für `xz`. Das y muss mindestens einmal vorhanden sein.

Wie schon bei `*` und `?` können Sie auch das `+`-Zeichen mit Zeichenfolgen und Zeichenklassen verwenden.

Die Anzahl der Wiederholungen beschränken

Bei den Metazeichen `+` und `*` kann das gegebene Zeichen oder die Zeichenfolge beliebig oft vorhanden sein, es gibt nach oben keine Begrenzung (Zeichen mit `?` können maximal nur einmal vorhanden sein). Wie aber gehen Sie vor, wenn Sie eine Übereinstimmung nur für eine spezielle Anzahl von Wiederholungen suchen? Was, wenn Sie für das gesuchte Muster eine Ober- beziehungsweise Untergrenze angeben wollen, so dass eine Wiederholung zuviel oder zuwenig nicht zu einer Übereinstimmung führt? Um die Anzahl der zu findenden Wiederholungen festzulegen, können Sie den optionalen Metazeichen geschweifte Klammern wie folgt einsetzen:

```
/\d{1,4} /
```

Dieses Muster wird gefunden, wenn die durchsuchten Daten eine Ziffer, zwei Ziffern, drei Ziffern oder vier Ziffern, jeweils gefolgt von einem Leerzeichen, enthalten. Weder mehr Ziffern noch keine Ziffer sind als Eingabe zulässig. Die erste Zahl in den geschweiften Klammern gibt die minimale Anzahl und die zweite die maximale Anzahl der Wiederholungen an. Durch Angabe nur einer Zahl können Sie eine feste Zahl von Wiederholungen angeben:

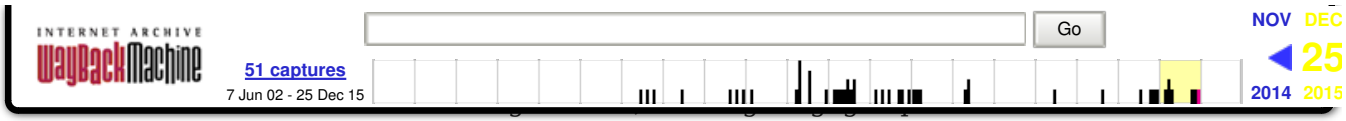
```
/a{5}b/
```

Ein Muster wie dieses sucht nach genau fünf `a`'s in einer Reihe, gefolgt von einem `b` - nicht mehr und nicht weniger. Es entspricht damit dem Muster `/aaaaab/`. Ein weniger spezifischer Einsatzbereich von `{}` wäre beispielsweise:

```
/DM \d+\.\d{2}/
```

Können Sie nach Studium dieses Musters selbst herausfinden, wie eine Übereinstimmung aussehen müßte? Ich habe hier eine Reihe von Escape-Zeichen verwendet, so dass Sie vielleicht etwas verwirrt sind. Zuerst wird die Zeichenfolge »`DM «` gesucht, dann ein oder mehrere Dezimalzahlen `\d+`, gefolgt von einem Punkt (`.`). Abschließend dürfen dann nur noch zwei und nicht mehr Dezimalzahlen stehen. Setzen Sie diese Aussagen zusammen, und Sie werden feststellen, dass dieses Muster einem Geldbetrag entspricht - `DM 45.23` wären demnach korrekt wie auch `DM 0.45` oder `DM 15.00`. Die Angabe von `DM .45` oder `DM 34.2` hingegen würden nicht erkannt, da dieses Muster mindestens eine Zahl links des Dezimalpunktes erwartet und genau zwei Zahlen rechts davon.

Kommen wir zurück zu den geschweiften Klammern. Sie können damit auch nur eine Untergrenze festlegen und die Anzahl der Wiederholungen nach oben offenlassen. Lassen Sie dazu die Zahlenangabe für das Maximum fort, und behalten Sie das Komma bei:



Zu Ihrer Information: Sie können auch `+`, `*` und `?` mit Hilfe der geschweiften Klammern ausdrücken:

```
/x{0,1}/   # entspricht /x?/
/x{0,}/    # entspricht /x*/
/x{1,}/    # entspricht /x+/
```

Mehr zum Erstellen von Mustern

Wir haben dieses Kapitel mit einem allgemeinen Überblick über die Verwendung von Mustern in Perl-Skripts begonnen, wobei wir einen `if`-Test und den Operator `=~` verwendet haben (den Sie beim Durchsuchen von `$_` fortlassen können). Und da Sie jetzt eine Vorstellung davon haben, wie Sie mit der Syntax der regulären Ausdrücke Muster erstellen, möchte ich zu Perl zurückkehren und beschreiben, wie Sie Muster in Ihren Perl-Skripts einsetzen können. Außerdem zeige ich Ihnen, wie Sie Variablen in den Mustern interpolieren und Muster in Schleifen verwenden.

Muster und Variablen

In allen bisher betrachteten Beispielen haben wir Muster als hartcodierten Satz von Zeichen in `if`-Bedingungen verwendet. Was aber, wenn Sie je nach Eingabe verschiedene Dinge vergleichen wollen? Wie können Sie das Suchmuster bei Ausführung des Skripts erzeugen oder ändern?

Problemlos! Muster können wie Anführungszeichen Variablen enthalten, und der Wert der Variablen wird im Muster ersetzt:

```
$muster = "^\d{3}$";
if (/$muster/) {...
```

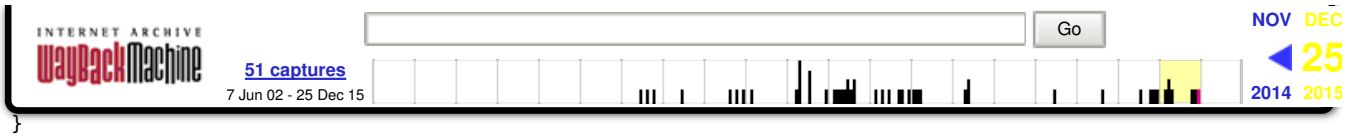
Die betreffende Variable kann einen String eines beliebigen Musters enthalten, einschließlich Metazeichen. Sie können diese Technik verwenden, um Muster auf verschiedene Art und Weise zu kombinieren oder um nach Mustern auf der Basis einer Eingabe zu suchen. Sehen Sie im folgenden ein Beispiel für ein einfaches Skript, das Sie auffordert, ein Muster und die zu durchsuchenden Daten einzugeben. Je nachdem, ob es zu einer Übereinstimmung kommt, liefert das Skript ***wahr*** oder ***falsch*** zurück.

```
#!/usr/bin/perl -w

print 'Geben Sie ein Muster ein: ';
chomp($muster = <STDIN>);

print 'Geben Sie den String ein: ';
chomp($in = <STDIN>);

if ($in =~ /$muster/) { print "wahr\n"; }
else { print "falsch\n"; }
```

Sie werden dieses Skript (oder ein ähnliches) mit zunehmenden Kenntnissen über reguläre Ausdrücke sehr nützlich finden.

Muster und Schleifen

Eine Möglichkeit, Muster in Perl-Skripts einzusetzen, besteht darin, sie wie bisher in `if`-Bedingungen zu verwenden. In skalarem Booleschen Kontext evaluieren die Tests zu ***wahr*** oder ***falsch***, je nachdem ob in den Daten eine Übereinstimmung zu dem Muster gefunden wird oder nicht. Eine andere Möglichkeit, ein Muster zu verwenden, wäre als Bedingung in einer




```
}
```

Die Option `/g` wird verwendet, um alle Muster in dem gegebenen String zu prüfen (hier `$_`, Sie können aber auch den Operator `=~` verwenden, um in anderen Daten eine Übereinstimmung zu finden). Im Falle eines `if`-Tests ist die Option `/g` nicht von Belang. Der Test wird bei der ersten gefundenen Übereinstimmung ***wahr*** zurückliefern. Im Falle einer `while`- (oder `for`-) Schleife bewirkt das `/g`, dass die Bedingung bei jedem Vorkommen des Musters im String ***wahr*** ist - und die Anweisungen im Block werden ebensooft ausgeführt.

> *Wir sprechen immer noch davon, Muster in einem skalaren Kontext zu verwenden. Hier bewirkt das `/g` lediglich interessante Varianten in Schleifen. Zu den Mustern im Listenkontext kommen wir morgen.*

Ein weiteres Beispiel: Zählen

Im folgenden sehen Sie ein Skript, das von der eben besprochenen Möglichkeit, Muster in Schleifen zu verwenden, Gebrauch macht. Das Skript verarbeitet eine oder mehrere Dateien und zählt, wie oft das gesuchte Muster in der Datei vorkommt. Mit diesem Skript könnten Sie zum Beispiel zählen, wie oft Ihr Name in einer Datei erwähnt wird, oder herausfinden, wie viele Besucher Ihrer Website von America Online (`aol.com`) kamen. Ich habe dieses Skript mit einem Entwurf dieses Kapitels getestet und festgestellt, dass ich das Wort Muster bisher über 180mal verwendet habe.

Listing 9.2 zeigt dieses einfache Skript:

Listing 9.2: zaehlen.pl

```
1:  #!/usr/bin/perl -w
2:
3:  $pat = ""; # wonach gesucht wird
4:  $count = 0; # Anzahl der Vorkommen
5:
6:  print 'Nach was soll gesucht werden? ';
7:  chomp($pat = <STDIN>);
8:  while (<>) {
9:      while (/$pat/g) {
10:         $count++;
11:     }
12: }
13:
14: print "/$pat/ $count mal gefunden.\n";
```

Wie bei allen bisher erstellten Skripts, die Dateien mit `<>` durchlaufen, müssen Sie auch dieses von der Befehlszeile mit dem Namen einer Datei aufrufen:

```
% zaehlen.pl logdatei
Nach was soll gesucht werden? aol.com
/aol.com/ 3456 mal gefunden.
%
```

Nichts in Listing 9.2 sollte für Sie absolut neu sein, auch wenn es einige Punkte gibt, die Beachtung verdienen. Zur Erinnerung: `while` zusammen mit dem Zeileneingabeoperator (`<>`) weist die einzelnen Zeilen der Eingabe der Standardvariablen `$_` zu. Da Muster standardmäßig diesen Wert vergleichen, benötigen wir keine temporäre Variable, die jede Eingabezeile aufnimmt. Die erste `while`-Schleife (Zeile 8) liest die Zeilen der Eingabedateien. Die zweite `while`-Schleife

Eine wichtige Sache zu diesem Skript möchte ich noch anmerken: Wenn Sie eine Suche nach einer Textpassage statt nach einem einzigen Wort durchführen - zum Beispiel alle Vorkommen Ihres Vor- und Nachnamens suchen - kann es passieren, dass die Textpassage über zwei Zeilen geht. Ein Skript wie dieses wird solche Vorkommen nicht finden, da keine der Zeilen das komplette Suchmuster enthält. Morgen lernen Sie, wie Sie nach einem Muster suchen, dass sich über mehrere Zeilen erstreckt.

Musterpriorität

Vielleicht erinnern Sie sich noch an unsere kleine Tabelle in Kapitel 3, »Weitere Skalare und Operatoren«, die die Prioritäten der verschiedenen Operatoren verdeutlichte und es Ihnen erlaubte, herauszufinden, welche Teile eines größeren Ausdrucks zuerst ausgewertet werden. Metazeichen in Mustern unterliegen ebenfalls Prioritätsregeln, die festlegen, auf welche Zeichen oder Zeichengruppen sich die Metazeichen beziehen. In Tabelle 9.2 sind diese Prioritäten aufgeführt, wobei die Zeichen weiter oben in der Tabelle enger binden als die Zeichen weiter unten.

Zeichen	Bedeutung
( )	Gruppe und Speicher
? + * { }	Quantifizierer
x \x $ ^ (?=) (?!)	Zeichen, Anker, Vorausschau
|	Alternativen

Tabelle 9.2: Prioritäten der Metazeichen für Muster

Wie schon bei den Ausdrücken können Sie mit `()` gruppieren, damit diese als Folge ausgewertet werden.

Noch sind Ihnen nicht alle Metazeichen bekannt. Morgen werden wir mehr davon kennenlernen.

Vertiefung

In diesem Kapitel haben Sie die Grundlagen der regulären Ausdrücke kennengelernt. Morgen stelle ich Ihnen weitere Einsatzbereiche für reguläre Ausdrücke vor. Für den Fall, dass Sie noch mehr Informationen zu dem bisher Gelernten benötigen, möchte ich Ihnen die Dokumentation ***perlre-***Manpage ans Herz legen.

Weitere Einsatzbereiche für Muster

Zu Beginn dieses Kapitels haben Sie gelernt, dass der Operator `=~` Muster mit skalaren Variablen vergleicht. Neben `=~` können Sie auch `!~` verwenden:

```
$Sache !~ = /muster/;
```

`!~` ist die logische Negation von `=~`. Mit anderen Worten, es wird nur ***wahr*** zurückgeliefert, wenn

verwenden, um die exakte Position zu bestimmen, an der eine Übereinstimmung gefunden wurde (mit Hilfe von `m//g`), oder um eine Mustersuche an einer bestimmten Position in einem String zu starten. Die `pos`-Funktion übernimmt einen skalaren Wert (oft eine Variable) als Argument und liefert den Offset des Zeichens ***nach*** dem letzten Zeichen der Übereinstimmung zurück.
Ein Beispiel:

```
$finde = "123 345 456 346";
while ($finde =~ /3/g) {
   print pos $finde,"\n";
}
```

Dieses Codefragment gibt alle Positionen des Strings `$finde` aus, an denen die Zahl 3 erscheint (3, 5, 13).

Weitere Informationen zu der `pos`-Funktion finden Sie in der Hilfsdokumentation ***perlre-***Manpage.

Musterbegrenzer und Escape-Zeichen

Alle Muster, die wir bisher betrachtet haben, begannen und endeten mit einem Slash und enthielten in der Mitte die Zeichen oder Metazeichen, die überprüft werden sollten. Die Slash-Zeichen selbst sind ebenfalls Metazeichen. Das bedeutet, dass, wenn Sie nach einem Slash suchen wollen, Sie ihm einen Backslash voranstellen müssen. Dies ist für Muster mit vielen Slash-Zeichen recht mühselig - zum Beispiel für Unix- Pfadnamen, die alle von Slash-Zeichen getrennt werden. Da kann es schnell passieren, dass Sie mit einem Muster arbeiten, das folgendermaßen aussieht:

```
/\/usr(\/local)*\/bin\//;
```

Das liest sich nur sehr schwer (schwerer als so mancher andere reguläre Ausdruck). Zum Glück hat Perl dafür eine Lösung: Sie müssen ein Muster nicht unbedingt in `//` einschließen. Sie können dafür jedes beliebige nichtalphanumerische Zeichen verwenden. Der einzige Haken dabei ist, dass Sie bei Verwendung eines anderen Zeichens das `m` im Ausdruck `m//` nicht weglassen dürfen (Sie können die Begrenzer auch substituieren, müssen dann aber `s///` verwenden). Auch für die neuen Begrenzer gilt, dass Sie diesen Escape-Zeichen voranstellen müssen, wenn Sie sie innerhalb des Musters verwenden wollen. So könnte zum Beispiel der obige Ausdruck auch wie folgt geschrieben werden:

```
m%/usr(/local)*/bin/%;
```

Es kann Ihnen auch passieren, dass ihr Suchmuster eine Reihe von nichtalphanumerischen Zeichen enthält, die gleichzeitig Metazeichen für Muster sind. Dies bedingt eine ganze Menge von Backslash-Zeichen für die Metazeichen, so dass das Muster nur schwer zu lesen ist. Mit dem Escape-Zeichen `\Q` können Sie die Musterverarbeitung für eine bestimmte Anzahl an Zeichen ausschalten und mit `\E` wieder einschalten. Wenn Sie zum Beispiel nach einem Muster suchen, das die folgenden Zeichen `{(^*)}` (aus was für Gründen auch immer) enthält, würde das folgende Muster genau nach diesen literalen Zeichen suchen:

```
/\Q{(^*)}\E/;
```

Die Musterverarbeitung mit `\Q` auszuschalten, ist auch für die Variableninterpolation innerhalb von Mustern nützlich, um ungewöhnliche Ergebnisse bei der Eingabe von Suchmustern zu verhindern:

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

Andere Sprachen mögen zwar über Bibliotheken und Funktionen für reguläre Ausdrücken verfügen, aber nur Perl allein arbeitet so intensiv mit Pattern Matching, einer Technik, die eng mit vielen anderen Aspekten der Sprache verbunden ist. Perl ohne reguläre Ausdrücke ist lediglich eine weitere seltsam anmutende Sprache. Perl mit regulären Ausdrücken hingegen ist unglaublich leistungsstark.

Heute haben Sie alles über Muster für Pattern-Matching-Operationen kennengelernt: wie man sie erstellt, sie verwendet, Teile davon speichert und sie mit anderen Teilen von Perl zusammen verwendet. Ich habe Ihnen die verschiedenen Metazeichen vorgestellt, die Sie innerhalb der regulären Ausdrücke verwenden können: Metazeichen zum Verankern eines Musters (`^`, `$`, `\B`, `\b`), zum Erzeugen einer Zeichenklasse (`[]` und `[^]`), zum Wechseln zwischen verschiedenen Mustern (`|`) und zum Suchen von mehrfachen Vorkommen eines Zeichens (`+`, `*`, `?`).

Mit Hilfe des Ausdrucks `m//` können Sie Muster auf Strings anwenden. Standardmäßig greifen Muster dabei auf den in der Variablen `$_` gespeicherten String zurück, es sei denn, Sie verwenden den Operator `=~`, um das Muster auf eine andere Variable anzuwenden.

Morgen werden wir das hier Besprochene weiter vertiefen, die bekannten Muster um weitere Muster ergänzen und neue und bessere Möglichkeiten kennenlernen, die Muster einzusetzen.

Fragen und Antworten

Frage:
Was ist der Unterschied zwischen `m//` und nur `//`?

Antwort:
Im Grunde genommen gibt es keinen Unterschied. Das `m` ist optional, es sei denn Sie verwenden ein anderes Zeichen als Musterbegrenzer. Ansonsten bewirken beide Formen dasselbe.

Frage:
Alternation bewirkt eine logische ODER-Verknüpfung in einem Muster. Wie erreiche ich ein logisches UND?

Antwort:
Am einfachsten verwenden Sie dazu mehrere Muster und verbinden sie mit dem `&&`- oder `and`-Operator:

```
/muster1/ && /muster2/;
```

Wenn Sie wissen, in welcher Reihenfolge die zwei Muster erscheinen, können Sie auch folgendes eingeben:

```
/muster1.*muster2/
```

Frage:
Ich habe ein Muster, das nach Zahlen sucht: `/\d*/`. Es sucht nach Zahlen, aber darüber hinaus auch nach allen anderen Strings. Was mache ich falsch?

Antwort:
Sie verwenden `*`, wo Sie eigentlich `+` meinen. Zur Erinnerung: `*` bedeutet »keine oder mehrere Vorkommen«. Das heißt, dass auch wenn Ihr String keine Zahl aufweist, immer noch eine Übereinstimmung gegeben ist - Sie haben dann eben »keine« Vorkommen. `+` wird verwendet, wenn das Muster zumindest einmal vorkommen soll.

die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Definieren Sie die Begriffe »Pattern Matching« und »reguläre Ausdrücke«.
2. Für welche Art von Aufgaben wird Pattern Matching sinnvollerweise eingesetzt? Nennen Sie drei!
3. Was bewirken die folgenden drei Muster?

```
  /ice\s*cream/
/\d\d\d/
/^\d+$/
/ab?c[,.:]d/
/xy|yz+/
/[\d\s]{2,3}/
/"[^"]"/
```

4. Gehen Sie davon aus, dass `$_` den Wert `123 kazoo kazoo 456` enthält. Wie lautet das Ergebnis der folgenden Ausdrücke?

```
  if (/kaz/) {  # wahr oder falsch?
while (/kaz/g) {  # was passiert?
if (/^\d+/) {  # wahr oder falsch?
if (/^\d?\s/) {  # wahr oder falsch?
if (/\d{4}/) {  # wahr oder falsch?
```

Übungen

1. Schreiben Sie Muster für die folgenden Aufgaben:

- Die ersten Wörter in einem Satz finden (das sind die Wörter, die mit einem Großbuchstaben beginnen und auf die Kombination Leerzeichen-Punkt folgen)
- Prozentangaben finden (alle Dezimalzahlen gefolgt von einem Prozentzeichen)
- Alle Zahlen finden (mit oder ohne Dezimalpunkt, positiv oder negativ)

2. FEHLERSUCHE: Was ist an folgendem Code falsch?

```
 print 'Geben Sie einen String ein: ';
chomp($input = <STDIN>);
print 'Wonach soll gesucht werden? ';
chomp($pat = <STDIN>);

if (/$pat/) {
   # Muster gefunden, bearbeiten!
}
```

3. FEHLERSUCHE: Und was ist an folgendem Code falsch?

```
 print 'Wonach soll gesucht werden? ';
chomp($pat = <STDIN>);
while (<>) {
    while (/$pat/) {
        $count++;
    }
}
```

4. Gestern haben wir ein Skript namens `mehrnamen.pl` erstellt, mit dem Sie eine Liste von Namen sortieren und nach verschiedenen Teilen durchsuchen konnten. Der Suchteil bestand aus einem verworrenen Mechanismus aus `each` und `grep`, um das Muster zu finden. Schreiben Sie

Hier die Antworten auf die Workshop-Fragen aus dem vorigen Abschnitt.

Antworten zum Quiz

1. ***Pattern Matchin***g ist ein Perl-Konzept, bei dem Muster formuliert werden, die dann auf einen String oder einen anderen Satz von Daten angewendet werden. ***Reguläre Ausdrücke*** sind eine Sprache, mit der diese Muster geschrieben werden.
2. Es gibt viele Anwendungsbereiche für Pattern Matching - sie werden nur durch Ihre Vorstellungskraft begrenzt. Hier eine kleine Auswahl:
3. a. Eingabeüberprüfung
4. b. Vorkommen in einem String zählen
5. c. Daten aus einem String auf der Basis bestimmter Kriterien extrahieren
6. d. Einen String in verschiedene Elemente zerlegen
7. e. Ein spezielles Muster durch einen anderen String ersetzen
8. f. Regelmäßige (oder unregelmäßige) Muster in einem Datensatz suchen
3. Die Antworten lauten wie folgt:
4. a. Dieses Muster findet die Zeichen `ice` und `cream`, wenn diese durch kein oder mehrere Leerzeichen getrennt sind.
5. b. Dieses Muster findet drei Ziffern in einer Reihe.
6. c. Dieses Muster findet eine oder mehrere Ziffern, die allein in einer Zeile stehen.
7. d. Dieses Muster findet ein `a`, ein optionales `b`, ein `c`, ein Komma, Punkt oder Doppelpunkt und ein `d`. Gefunden wird also `ac.d` oder auch `abc,d`, aber nicht `abcd`.
8. e. Dieses Muster findet entweder `xy` oder `y` mit einem oder mehreren `z`.
9. f. Dieses Muster findet entweder eine Ziffer oder ein Leerzeichen, das mindestens zweimal, aber nicht öfter als dreimal auftritt.
10. g. Dieses Muster findet alle Zeichen zwischen dem öffnenden und dem schließenden Anführungszeichen.
4. Die Antworten lauten:
5. a. ***wahr***
6. b. Die Schleife wird für jedes Vorkommen von `kaz` in dem String durchlaufen (hier zweimal).
7. *c.* ***wahr***. Das Muster findet eine oder mehrere Ziffern zu Beginn der Zeile.
8. *d.* ***falsch***. Dieses Muster findet 0 oder eine Ziffer zu Beginn der Zeile gefolgt von einem Leerzeichen. Die drei Ziffern in unserem String werden nicht gefunden.
9. *e.* ***falsch***. Dieses Muster findet vier Ziffern in einer Reihe; wir haben jedoch hier nur drei Ziffern.

Lösungen zu den Übungen

1. Wie meistens bei Perl gibt es mehrere Lösungswege zu einem Problem. Hier einige der möglichen Lösungen:

```
  /[.!?"]\s+[A-Z]\w+\b/
/d+%/
/[+-]\d+\.?\d+/
```

2. Die Variable, auf die das Muster angewendet wird, und die Variable, in der sich die eigentlichen Daten befinden, sind nicht identisch. Das Muster in der `if`- Anweisung versucht das Muster mit `$_` abzugleichen, aber wegen der zweiten Zeile befindet sich die eigentliche Eingabe in `$input`. Verwenden Sie statt dessen diesen `if`-Test:

```
if ($input =~ /$spat/)  {
```

3. Diese Übung war gemein, da der Code syntaktisch korrekt ist. Die zweite `while`- Schleife - das ist die mit dem Muster - sucht nach dem Muster in `$_`, was korrekt ist. Aber der Test ist ein

immer und immer wieder. Es gibt nichts, was die Iteration der Schleife stoppt.

4. Mit der Option `/g` hinter dem Muster in der `while`-Schleife erreichen Sie den besonderen Fall, dass die `while`-Schleife nur so oft durchlaufen wird, wie das Muster im String gefunden wurde, und dann abbricht. Wenn Sie Muster innerhalb von Schleifen verwenden, dürfen Sie das `/g` nicht vergessen.
4. Der einzige Teil, der geändert werden muss, sind die Zeilen, die das `@keys`-Array erzeugen (und in denen mit `grep` nach dem Muster gesucht wird). Hier werden wir eine `foreach`-Schleife verwenden und in dieser Schlüssel und Wert testen. Wir fügen noch die Option `/i` hinzu, um nicht zwischen Groß- und Kleinschreibung unterscheiden zu müssen, und setzen dann die `@keys`-Liste auf die leere Liste zurück, so dass sie nicht von einem Suchlauf zum nächsten immer länger wird. Hier sehen Sie die neue Version:

```
    } elsif ($in eq '3') {      # sucht einen Namen (1 oder mehr)

   print "Wonach soll gesucht werden? ";
   chomp($search = <STDIN>);

   @keys = ();
   foreach (keys %names) {
       if (/$search/i or $names{$_} =~ /$search/i) {
           push @keys, $_;
       }
   }

   if (@keys) {
       print "Gefundene Namen: \n";
       foreach $name (sort @keys) {
           print "   $names{$name} $name\n";
       }
   } else {
       print "Keine gefunden.\n";
   }
}
```

---

vorheriges Kapitel Inhalt Stichwortverzeichnis Suchen
nächstes Kapitel