



Wolfgang Wiedl

# Reguläre Ausdrücke

Grundlagen, Praxislösungen, Referenz

Galileo Computing 

# Auf einen Blick

1	Einleitung .....	11
2	Zeichen: Codierung und Bezeichnung .....	75
3	Zeichenklassen .....	137
4	Anker und Vergleiche .....	199
5	Operatoren .....	247
6	PHP-Spezialitäten .....	361
7	Perl-Spezialitäten .....	375
8	Einbettung in JavaScript .....	409
9	Reguläre Ausdrücke in der Apache-Konfiguration .	433
	Index .....	443

# Inhalt

## **1 Einleitung 11**

1.1	Mustersuche .....	11
1.2	Globale Mustersuche .....	15
1.3	Mustersuche in PHP .....	19
1.4	Mustersuche in Perl .....	25
1.5	Substitution .....	30
1.6	Substitution mit Code-Auswertung .....	41
1.7	Zerlegung .....	45
1.8	Filtern .....	49
1.9	x-Modifier .....	51
1.10	Beispiele in anderen Werkzeugen .....	54
1.10.1	Python .....	54
1.10.2	JavaScript .....	57
1.10.3	vi-Editor .....	58
1.10.4	mySQL-Datenbank .....	61
1.10.5	Apache-Konfiguration .....	64
1.10.6	Suchen und Ersetzen in Dreamweaver .....	67
	Lösungen der Übungsaufgaben .....	68

## **2 Zeichen: Codierung und Bezeichnung 75**

2.1	Zeichencodierung .....	75
2.1.1	Zugriff auf Code-Punkte .....	77
2.1.2	cp1252-Code .....	79
2.1.3	Unicode .....	80
2.2	Verschiedene Codierungen in Perl .....	85
2.3	Zeilenvorschub .....	88
2.4	Literale Zeichen und Escapesequenzen .....	92
2.4.1	Escapesequenzen im interpolativen Kontext .....	94
2.5	Here-Dokumente .....	101
2.6	Variablen im interpolativen Kontext .....	102
2.7	Text-Operatoren .....	104
2.8	Auswertung im interpolativen Kontext .....	108
2.9	Reguläre Ausdrücke .....	113
2.10	Anwendung: Verarbeitung von HTML-Formularen .....	124
	Lösungen der Übungsaufgaben .....	129

## **3 Zeichenklassen 137**

3.1	Frei definierbare Zeichenklassen .....	138
3.1.1	Positive Zeichenklassen und Klasselemente .....	138
3.1.2	Negierende Zeichenklassen .....	143
3.1.3	Regeln für Metazeichen .....	149
3.1.4	Zeichenklassen und Modifier .....	152
3.1.5	Syntaxfragen .....	153
3.2	Locale und Encoding .....	154
3.2.1	Locales .....	154
3.2.2	Encoding und Zeichenklassen in Perl .....	161
3.2.3	Locale und Encoding .....	162
3.3	Vorgegebene Zeichenklassen .....	168
3.3.1	Kritische Zeichen .....	168
3.3.2	Vordefinierte Perl-Zeichenklassen .....	170
3.3.3	POSIX-Klammerausdrücke .....	175
3.3.4	Unicode-Kategorien, -Blöcke und -Properties .....	178
3.3.5	Block-Eigenschaften .....	183
3.3.6	Schriften .....	183
3.3.7	Definition von Zeichenklassen durch Subroutinen in Perl .....	188
3.3.7	Lösungen der Übungsaufgaben .....	191

## **4 Anker und Vergleiche 199**

4.1	Anker in der Mustersuche .....	199
4.2	Nulltreffer und Nullmuster .....	210
4.3	Anfang und Ende im Suchtext .....	214
4.4	Anfang und Ende in einer Zeile .....	216
4.5	Wortgrenzen .....	222
4.6	Probleme mit Ankern .....	224
4.7	Vergleich nach rechts, verbrauchte Positionen und Zeichen .....	228
4.8	Vergleich nach links .....	235
	Lösungen der Übungsaufgaben .....	239

## **5 Operatoren 247**

5.1	Alternativen .....	247
5.2	Verkettung .....	252
5.3	Quantifier .....	255
5.3.1	Maximale und minimale Quantifier .....	257
5.3.2	Schachtelung quantifizierter Elemente und Alternativen .....	259
5.3.3	Quantifizierte Teilmuster mit Nulltreffern .....	261
5.3.4	Backtracking .....	264
5.3.5	Syntax der Quantifier .....	272

5.4	Einweg-Teilmuster .....	275
5.5	Bedingte Teilmuster .....	282
5.6	Klammerung .....	289
5.6.1	Gruppierende Klammern .....	290
5.6.2	Lokale Modifier .....	291
5.6.3	Einfangende Klammern .....	302
5.6.4	Interne Rückwärtsreferenzen .....	304
5.6.5	Interne Rückwärtsreferenzen und oktale Escapes .....	307
5.6.6	Interne Rückwärtsreferenzen auf nicht definierte Treffer oder Nulltreffer .....	309
5.6.7	Interne Rückwärtsreferenzen in Vergleichen nach links .....	311
5.6.8	Externe Rückwärtsreferenzen und Treffervariablen .....	312
5.6.9	Einfangende Klammern in negierenden Konstrukten .....	314
5.6.10	Geltungsbereich der Treffervariablen \$n .....	316
5.6.11	Werte der Treffervariablen .....	323
5.6.12	Einfangende Klammern bei der Textzerlegung .....	327
5.6.13	Einfangende Klammern und e-Modifier .....	335
5.7	Anwendung: Primzahlentest mit Strichlinien-Arithmetik .....	343
5.8	Anwendung: Rekursives Löschen von Blöcken in HTML-Dateien .....	346
	Lösungen der Übungsaufgaben .....	353

## **6 PHP-Spezialitäten 361**

6.1	Reguläre Ausdrücke nach POSIX .....	361
6.1.1	Mustersuche .....	361
6.1.2	Textersetzung .....	362
6.1.3	Textzerlegung .....	362
6.1.4	Zeichenmuster .....	363
6.1.5	Zeichenklassen .....	365
6.1.6	Anker .....	365
6.1.7	Operatoren .....	367
6.1.8	Rückwärtsreferenzen .....	367
6.1.9	Nullmuster und Nulltreffer .....	367
6.2	Multi-Byte String-Funktionen .....	368
6.3	Modifier zu Perl-kompatiblen regulären Ausdrücken .....	368
6.4	Rekursive Muster in Perl-kompatiblen regulären Ausdrücken .....	369

## **7 Perl-Spezialitäten 375**

7.1	Grenzmarkierungen .....	375
7.2	Modifier .....	378
7.3	Spezielle Variablen für reguläre Ausdrücke .....	379
7.4	Bindungsoperatoren .....	381
7.5	Progressive Mustersuche .....	382

7.6	Textzerlegung .....	385
7.7	Testen von regulären Ausdrücken .....	390
7.8	Der Debugger .....	393
7.9	Ausführbarer Code .....	395
7.10	qr//-Operator .....	397
7.11	Rekursive Muster – Mustergenerierung zur Laufzeit .....	399
7.12	Benannte Musterelemente .....	404

## **8 Einbettung in JavaScript 409**

8.1	Zeichenketten und maskierte Zeichen .....	411
8.2	Musterelemente .....	412
8.3	Unicode .....	415
8.4	Reguläre Ausdrücke .....	416
8.4.1	Mustersuche mit match und search .....	417
8.4.2	Textersetzung mit replace .....	419
8.4.3	Textzerlegung mit split .....	421
8.4.4	Methoden der Klasse regular expression .....	422
8.4.5	Mustersuche mit test .....	425
8.4.6	Mustersuche mit exec .....	425
8.4.7	Optimierung eines regular-expression-Objektes mit compile .....	427
8.5	Anwendung .....	428

## **9 Reguläre Ausdrücke in der Apache-Konfiguration 433**

9.1	Rewrite Engine .....	433
9.2	RewriteRule .....	434
9.3	RewriteCond .....	436
9.4	Rückwärtsreferenzen .....	440

## **Index 443**

## 4 Anker und Vergleiche

*In den beiden vorangehenden Kapiteln wurden hauptsächlich Musterelemente betrachtet, die entweder auf ein ganz bestimmtes Zeichen oder aber auf eines aus einer Klasse von Zeichen passten. Dieses Zeichen kann irgendwo im Suchtext auftreten. Jetzt werden Musterelemente betrachtet, die Randbedingungen für eventuelle Treffer vorgeben, etwa von der Art, dass Treffer nur am Anfang des Suchtextes oder nur nach einem Punkt möglich sind, wobei der Punkt selbst nicht zum Treffer gehört. Im Gegensatz zu den vorherigen Musterelementen passen diese Musterelemente nicht unbedingt auf ein Zeichen im Suchtext, sie können auch nur auf eine Position vor oder nach einem Zeichen passen.*

### 4.1 Anker in der Mustersuche

Die Anker `^`, `\z` und `$` wurden schon in der Einleitung benutzt, um mögliche Treffer am Anfang oder Ende des Suchtextes zu verankern. Solche Anker unterscheiden sich wesentlich von jenen Musterelementen, die jeweils auf ein Zeichen passen:

- ▶ Sie passen auf Positionen vor oder nach einem Zeichen, nicht auf ein Zeichen. Wenn man von einem Treffer spricht, meint man in der Regel eine Zeichenkette oder wenigstens ein Zeichen, auf welches das Muster passt. In diesem Sinne erzielt ein Anker keinen Treffer, aber er kann auf eine Position passen oder auch nicht passen.
- ▶ Eine Verkettung oder Quantifizierung von Ankern ist in der Regel sinnlos, `^^^^` kann höchstens den Sinn haben, dass man sich viermal versichert, ob wirklich der Anfang des Suchtextes vorliegt. Eine Verkettung oder Quantifizierung von Zeichenmustern oder Zeichenklassen bewirkt dagegen, dass nach einem Treffer gesucht wird, der gerade so viele Zeichen umfasst, wie die Kette der Musterelemente vorgibt. Eine Kette `aaa` passt auf eine Folge von drei aufeinander folgenden Buchstaben `a` im Suchtext und nicht etwa dreimal auf dasselbe `a`.
- ▶ Nach erfolgreicher Überprüfung des Ankers `^` bleibt die Mustersuche auf derselben Position im Suchtext stehen, auf der sie vor der Überprüfung war. Nach erfolgreicher Überprüfung eines literalen `a` rückt die Mustersuche dagegen eine Position im Suchtext weiter.

- ▶ Da eine erfolgreiche Prüfung eines Ankers keine Veränderung der Position im Suchtext bewirkt, kann man an ein und derselben Position zwei oder auch mehrere Prüfungen miteinander verknüpfen.

`/^z/`

passt auf eine Position im Suchtext, an der dieser beginnt und endet, passt also nur, wenn als Suchtext eine leere Zeichenkette vorgegeben wird.

- ▶ Die Prüfung zu

`/ab/`

findet dagegen an zwei aufeinander folgenden Positionen statt: Wenn rechts von der aktuellen Position im Suchtext ein `a` gefunden wurde, wird dieses in den vorläufigen Treffer einverleibt. Dann wird die aktuelle Position um eine Stelle nach rechts verschoben (hinter das gerade verbrauchte `a`), und es wird geprüft, ob rechts von der jetzt aktuellen Position ein `b` folgt. Die Prüfungen dieser literalen Musterelemente sind immer mit einem Fortschreiten im Suchtext verbunden, und zwei aufeinander folgende literale Musterelemente prüfen deshalb immer unterschiedliche Zeichen im Suchtext.

- ▶ Die Prüfung mit

`/^abz/`

passt auf den Suchtext `ab`. Der Anker `^` passt nur auf eine Position, der kein Zeichen vorangeht, also auf Position 0 im Suchtext. Das Musterelement `a` passt überall im Suchtext, sofern rechts von der aktuellen Position im Suchtext ein `a` folgt. Die beiden Musterelemente `^a` zusammen passen daher am Anfang eines Suchtextes, der mit `a` beginnt. Da mit der erfolgreichen Prüfung des Musterelementes `a` das Zeichen `a` im Suchtext verbraucht ist, erfolgt die Prüfung des nächsten Musterelementes `b` an der Position hinter dem `a` im Suchtext und passt, wenn rechts davon ein Zeichen `b` folgt. Ist das der Fall, so ist auch das Zeichen `b` im Suchtext verbraucht, und die Prüfung des Musterelementes `z` erfolgt auf der Position hinter dem `b` im Suchtext. `z` passt dort nur, wenn rechts von dieser Position kein Zeichen mehr folgt. Insgesamt passt das Muster deshalb nur auf einen Suchtext, der nur `ab` umfasst.

An einer Position im Suchtext können beliebig viele Prüfungen vorgenommen werden, aber nur eine davon kann verbrauchend sein. Mit einer erfolgreichen verbrauchenden Prüfung endet immer die Folge der Prüfungen an einer Position.

Anfang und Ende eines Suchtextes sind nicht die einzigen Randbedingungen, die man üblicherweise an einen Treffer stellt. Anfang und Ende einer Zeile oder Anfang und Ende eines Satzes oder Wortes sind analoge Bedingungen. Es gibt aber auch kompliziertere Formulierungen. Im Satz `alles nach "Sommer" bis zum Ende streichen` beschreibt `alles` den möglichen Treffer, der gestrichen werden soll. `Sommer` ist eine Positionsangabe, die den Anfang dieses Treffers beschreibt, selbst aber nicht zu diesem Treffer gehört. Der entsprechende reguläre Ausdruck ist

```
(?<=Sommer) . {1, }
```

und umfasst

- ▶ den Vergleich nach links `(?<=Sommer)`, der zwar auf eine Zeichenkette `Sommer` passt, diese aber nicht als Treffer erzielt, also nicht verbraucht;
- ▶ das quantifizierte Musterelement `. {1, }`, das auf jede nichtleere Folge beliebiger Zeichen außer `\n` (linefeed) passt und diese als Treffer erzielt.

Die Prüfung eines Musterelementes findet im Suchtext immer an einer Position vor oder nach einem Zeichen statt, wir sprechen einfach von der jeweils aktuellen Position oder Zeichengrenze. Es gibt Musterelemente, welche die Überprüfung des Suchtextes rechts von dieser Position erfordern, und es gibt solche, welche die Überprüfung des Suchtextes links von dieser Position erfordern. Es gibt auch Musterelemente, die eine Überprüfung nach beiden Seiten erfordern. Wir sprechen im Folgenden von links-, rechts- und beidseitig vergleichenden Musterelementen.

Ein literales `a` in einem regulären Ausdruck erfordert die Überprüfung eines Zeichens rechts von der aktuellen Position im Suchtext. Ein Musterelement `(?<=Sommer)` erfordert die Überprüfung von sechs Zeichen links von der aktuellen Position im Suchtext. Der Anker `^` erfordert eine Prüfung nach links, dort dürfen im Suchtext keine Zeichen existieren. Anstatt von »rechts« und »links« kann man auch von »vorwärts« und »rückwärts« sprechen, was sich leichter auf Schriften übertragen lässt, die von rechts nach links oder von oben nach unten geschrieben werden. Hier werden aber nur Schriften betrachtet, die von links nach rechts geschrieben werden.

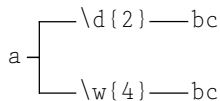
Neben der Richtung, in der die zu einem Musterelement erforderlichen Vergleiche erfolgen, ist ein wesentliches Unterscheidungsmerkmal noch, ob bei einem positiven Vergleich ein Zeichen in den vorläufigen Treffer eingegliedert wird oder nicht. Ein positives Ergebnis einer Prüfung mit `^` liefert kein Zeichen zum (vorläufigen) Treffer des regulären Ausdrucks, eine positiv verlaufene Prüfung mit dem Musterelement `a` dagegen als Beitrag das Zeichen `a`.

Von einem vorläufigen Treffer wird hier deshalb gesprochen, weil die Prüfungen Schritt für Schritt erfolgen und dabei jeweils einzelne Glieder zum Treffer eingesammelt werden. Sobald ein Schritt fehlschlägt, werden einige Schritte rückgängig gemacht, und die dabei eingesammelten Glieder des vorläufigen Treffers werden wieder freigegeben.

### Beispiel 4.1

Mit dem Muster  $a(\backslash d\{2}\mid\backslash w\{4})bc$  gibt es im Suchtext `da234bbc` auf Position 1 einen Treffer zum Musterelement `a`, und das betreffende `a` aus dem Suchtext wird Teil des vorläufigen Treffers. Danach gibt es eine Verzweigung oder einen Rückkehrpunkt. Die Suche kann mit  $\backslash d\{2}$  oder  $\backslash w\{4}$  fortgesetzt werden. Da nur ein Pfad zur gleichen Zeit verfolgt werden kann, wird die Suche zuerst mit dem Musterelement  $\backslash d\{2}$  fortgesetzt, die Verzweigung wird aber als Rückkehrpunkt markiert. Bei der Suche mit  $\backslash d\{2}$  werden auch die beiden Ziffern `23` aus dem Suchtext in den vorläufigen Treffer einverleibt. Auf Position 4 (zwischen 3 und 4) erweist sich der eingeschlagene Weg aber als Sackgasse. Jetzt kehrt die Mustersuche zum Rückkehrpunkt zurück, muss dabei aber die in der Sackgasse eingesammelten zwei Ziffern wieder freigeben. Die Suche mit dem Musterelement  $\backslash w$  ab dem Verzweigungspunkt führt dann zum Erfolg.

Ein Muster gibt verschiedene Kombinationsmöglichkeiten seiner Musterelemente vor, die wir hier als verschiedene Pfade durch das Muster kennen gelernt haben.



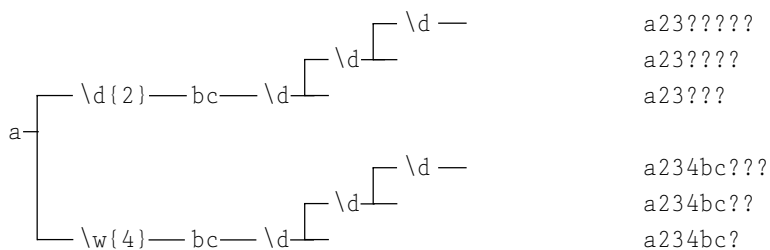
Wenn einer dieser Pfade vollständig im Suchtext passt, ist die Mustersuche erfolgreich. Wenn mehrere passen, liefert der erste gefundene erfolgreiche Pfad die Lösung.

Die Pfade durch das Muster lassen sich als Baum darstellen, und durch die Formulierung des Musters wird auch vorgegeben, in welcher Reihenfolge diese Pfade überprüft werden. Eine Überprüfung wird natürlich abgebrochen, sobald feststeht, dass der betreffende Pfad bzw. die betreffenden Pfade nicht zum Ziel führen.

Wenn anstelle des Musters aus Beispiel 4.1 das Muster

$a(\backslash d\{2}\mid\backslash w\{4})bc\backslash d\{1,3\}$

gegeben ist, hat man die Pfade



Die Regeln zur Interpretation des Musters besagen, dass diese Pfade von oben nach unten überprüft werden müssen. Beim gleichen Suchtext wie in Beispiel 4.1 steht für die Suche ab Startposition 1 (beginnend mit `a` im Suchtext) aber nach dem Treffer von `\d{2}` und dem Fehlversuch mit dem Musterelement `b` schon fest, dass alle oberen Pfade nicht bis zum Ende erfolgreich durchlaufen werden können. Man kann die Suche auf diesen drei Pfaden, die ein gemeinsames Anfangsstück haben, also schon hier abbrechen. Die Suche auf den unteren Pfaden ist etwas länger erfolgreich, endet aber auch nach der erfolgreichen Überprüfung des Musterelementes `c`, da keine weiteren Zeichen mehr vorhanden sind. Die Überprüfung der Reststücke kann auch hier gemeinsam abgebrochen werden, sobald feststeht, dass keine Ziffer mehr folgt. Rechts ist jeweils angegeben, welche vorläufigen Treffer bis zum Abbruch aus dem Suchtext eingesammelt wurden. Die Fragezeichen geben an, wie viele Zeichen im Treffer noch bis zum Erfolg fehlen.

Dieses Szenario beschreibt nur die Mustersuche ab Startposition 1. Da sie hier nicht erfolgreich war, wird danach versucht, ab Startposition 2 einen Treffer zu erzielen und so fort.

Es hängt vom jeweiligen Musterelement ab, ob nach einer erfolgreichen Prüfung mit ihm die überprüften Zeichen des Suchtextes Teil des vorläufigen Treffers werden oder nicht. Falls dies geschieht, etwa bei einem literalen `a`, spricht man von verbrauchenden Musterelementen. Literale Zeichen und alle Arten von Zeichenmuster und Zeichenklassen sind Beispiele verbrauchender Musterelemente. Nicht verbrauchende Musterelemente werden häufig auch als Versicherung (*assertion*) bezeichnet. Wir verwenden hier den einprägsameren Begriff »Anker« für alle nicht verbrauchenden Musterelemente.

Vom jeweiligen Musterelement hängt es auch ab, welche Folgen das Ergebnis einer Überprüfung auf die Mustersuche hat. Wenn eine Überprüfung nicht erfolgreich war, wird die Suche auf den jeweiligen Reststücken der Pfade nicht fortgesetzt und zum nächstliegenden Rückkehrpunkt zurückgekehrt. Eventuell eingesammelte Zeichen auf diesem nun verworfenen Wegstück müssen dabei wieder zurückgegeben werden. Wenn es keinen Rückkehrpunkt gibt, war die Suche ab dem aktuellen Startpunkt erfolglos. Wenn die Überprüfung aber

erfolgreich und das Musterelement, mit dem die Überprüfung erfolgte, verbrauchend war, so wird die aktuelle Position im Suchtext hinter die dabei verbrauchten Zeichen gesetzt. War das Musterelement dagegen nicht verbrauchend, so ändert sich an der aktuellen Position nichts.

## Beispiel 4.2

Im Suchtext `Der Sommer ist schön.` soll das Wort nach `Sommer` durch das Wort `war` ersetzt werden.

Als regulärer Ausdruck wird:

```
/(?<=Sommer )\w{1,}/
```

verwendet. Anstelle von `\w` könnte man auch `[[:alpha:]]` oder wenn möglich `\pL` verwenden.

Wie verläuft nun die Mustersuche? Im Suchtext gibt es 21 Zeichengrenzen:

```
|D|e|r| |S|o|m|m|e|r| |i|s|t| |s|c|h|ö|n|.|  
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

Im regulären Ausdruck zwei (komplexe) Musterelemente:

```
(?<=Sommer ) und \w{1,}
```

Die Musterelemente bilden eine Kette, und damit gibt es nur den Pfad:

```
(?<=Sommer ) → \w{1,}
```

Die Prüfung mit beiden Musterelementen muss erfolgreich sein, mit dem zweiten Musterelement wird deshalb nur dann gesucht, wenn die Suche mit dem ersten Musterelement erfolgreich war.

Die Suche mit dem ersten Musterelement beginnt auf Position 0, nach einem erfolglosen Versuch steht schon fest, dass auf der jeweiligen Startposition kein Treffer zu erzielen ist. Diese wird deshalb um eine Stelle weiter nach rechts verschoben. Wenn die Suche jedoch erfolgreich war, wenn also vor der aktuellen Startposition im Suchtext die Zeichenfolge `Sommer` steht, ist die Suche mit diesem Vergleich nach links beendet. Die Suche mit dem nächsten Musterelement beginnt dann da, wo die mit dem vorangehenden geendet hat.

Im gegebenen Fall findet der Vergleich nach links auf Position 11 zum ersten Mal einen Treffer. Da der Vergleich nach links nicht verbrauchend ist, ändert das positive Ergebnis nichts an der Position im Suchtext und trägt auch nichts zum vorläufigen Treffer der momentanen Suche bei. Die Suche

mit dem zweiten Musterelement `\w{1,}` beginnt auf Position 11, erfolgt nach rechts und muss auf dieser Position erfolgreich sein, wenn die gesamte Mustersuche ab Position 11 erfolgreich sein soll. `\w{1,}` versucht so viele Zeichen wie möglich zu verbrauchen, die Prüfung ist aber schon dann erfolgreich, wenn nur mindestens eines verbraucht wird. Wie Sie sehen, können sogar drei Zeichen verbraucht werden. Alle Musterelemente auf dem (einzigen) Pfad wurden erfolgreich geprüft, damit ist die Mustersuche erfolgreich beendet, der Treffer ist die Zeichenkette `ist`.

Das PHP-Programm:

```
$Stext = 'Der Sommer ist schön.';
print preg_replace('/(?<=Sommer )\w{1,}/', 'war', $Stext);
```

liefert dementsprechend die Ausgabe:

```
Der Sommer war schön.
```

In Beispiel 4.2 mussten einige Zeichenpositionen vergeblich überprüft werden, bevor das erste Musterelement passte, dann hat aber auch sofort das zweite Musterelement gepasst, so dass ein Treffer gefunden wurde. Es kann aber auch passieren, dass das erste Musterelement passt, aber das zweite oder dritte nicht. Dann muss die Mustersuche wieder neu begonnen werden, wobei die schon überprüften und verworfenen Startpositionen natürlich nicht noch einmal überprüft werden müssen. Die Mustersuche verläuft dann etwas komplizierter.

### Beispiel 4.3

Wir betrachten erneut den Suchtext `Der Sommer ist schön`. Diesmal soll das Wort vor `schön` ersetzt werden durch `war`. Wir haben also dasselbe Problem mit einer anderen Randbedingung formuliert.

Ein Musterelement, das auf ein nachfolgendes Wort `schön` passt, ist `(?=schön)`. Es handelt sich um einen Vergleich nach rechts, der ebenso wie ein Vergleich nach links nicht verbrauchend ist. Als regulärer Ausdruck soll

```
/[[[:alpha:]]{1,}(?= schön)/
```

verwendet werden. Es gibt hier im Prinzip unendlich viele Pfade:

```

[[[:alpha:]] └─ (=? schön)
                └─ [[[:alpha:]] └─ (=? schön)
                        └─ [[[:alpha:]] └─ (=? schön)
                                └─ ...

```



```
[[[:alpha:]]+ (?= schön)
```

was wiederum misslingt. Damit steht nun fest, dass ab Startposition 0 kein Treffer zu erzielen ist, und die ganze Prozedur wiederholt sich ab Startposition 1, 2 ...

Wir überblicken Suchtext und Muster so weit, dass wir sagen können, dass es sinnlos ist, ab Startposition 1 zu suchen. Wenn man von 0 bis 3 keinen Treffer zu `(?= schön)` gefunden hat, wird man auch von 1 bis 3 keinen finden. Der Algorithmus der Mustersuche tut das nicht.

Erst auf Startposition 11 nimmt die Mustersuche einen anderen Verlauf. Eine Folge von drei `[[[:alpha:]]`-Elementen passt auf `ist`, und die nachfolgende Prüfung nach rechts mit `(?= schön)` ist erfolgreich, womit die gesamte Mustersuche erfolgreich ist.

Die PHP-Anweisungen:

```
$Stext = 'Der Sommer ist schön.';
print preg_replace('[[[:alpha:]]+(?= schön)/', 'war', $Stext);
```

liefern:

```
Der Sommer war schön.
```

Das hier vorgestellte Modell einer Mustersuche ist noch nicht vollständig und muss im Zusammenhang mit Quantifiern und Alternativen noch ergänzt werden. Was jedoch auf später verschoben wird.

Uns Menschen fällt es nicht schwer, eine Mustersuche in derart einfachen Suchtexten durchzuführen, und wir sind nicht auf einen solchen Algorithmus angewiesen. Wenn Sie jedoch eine ganz lange Zahlenreihe daraufhin überprüfen müssen, ob in ihr eine andere lange Zahlenreihe vorkommt, werden Sie ein ähnliches Verfahren verwenden.

Beispiel 4.3 lässt sich auch ohne Verwendung eines Vergleiches nach links formulieren, wird dadurch aber nicht einfacher.

#### Beispiel 4.4

Wir verwenden zur Mustersuche den regulären Ausdruck:

```
/(Sommer )\w{1,}/
```

in dem `Sommer` als literale Zeichenkette angegeben wird.

```
|D|e|r| |S|o|m|m|e|r| |i|s|t| |s|c|h|ö|n|. |
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

Im Suchtext wird jetzt ein Treffer ab Position 4 gefunden, der sich bis Position 14 erstreckt. Dies ist aber nicht der gesuchte Treffer, der gegen `war` ausgetauscht werden kann. Man muss deshalb beim Treffer den gewünschten Treffer von seiner an sich unerwünschten Umgebung, die aber im Treffer mitgeliefert wird, unterscheiden. Dies geschieht hier mit Hilfe der einfangenden Klammern, die dafür sorgen, dass der Treffer des eingeklammerten Teils der Musterkette abgespeichert wird. Bei der Substitution wird nun zwar der gesamte Treffer ersetzt, wir können den Ersatztext aber so formulieren, dass er den abgespeicherten Teil des Treffers enthält.

```
$Stext = 'Der Sommer ist schön.';
print preg_replace('/(Sommer) \w{1,}/', "$1 war", $Stext);
```

Im Prinzip wurden hier `Sommer` und die nachfolgende Leerstelle zuerst aus dem Suchtext entfernt und dann wieder eingefügt. Die Variable `$1` hat als Wert den Treffer der einfangenden Klammer, also `Sommer`, und wird im Ersatztext von PHP auch dann ausgewertet, wenn dieser mit `'`-Grenzen angegeben wird.

Der Algorithmus der Mustersuche in Beispiel 4.4 hört sich umständlich an und ist es in diesem speziellen Falle auch. Hier wird in komplizierter Weise zuerst eine Zeichenkette gesucht, die praktisch alles sein kann, und dahinter soll `Sommer` stehen. Wir würden in einem solchen Fall immer zuerst nach `Sommer` suchen und dann das Wort davor suchen.

## Beispiel 4.5

Kapitel 1 dieses Buches enthält als Word-Datei etwas über 500 Zeichenfolgen `\r\n`, die als Zeilenenden beim Einlesen verwendet werden können, was relativ lange Zeilen ergibt. In dieser Datei sollen nun die 50 Zeichen ermittelt werden, die vor der Zeichenfolge `Lösungen der Übungsaufgaben` stehen. Als ersten Ansatz werden die meisten wohl das Muster:

```
/{50}(?=Lösungen der Übungsaufgaben)/
```

versuchen, aber das ist genau das ineffektive Verfahren, von dem gesprochen wurde. In jeder der 500 Zeilen wird geschaut, ob es 50 Zeichen gibt und ob hinter diesen 50 Zeichen das Wort `Sommer` folgt. Ersteres passt bei langen Zeilen fast immer, Letzteres nicht. Nach einem Fehlversuch wird dann die Startposition um eine Stelle verschoben, und die ganze Prozedur beginnt von neuem. Die Mustersuche mit:

```
/(?=Lösungen der Übungsaufgaben)(?<=.{50}))/
```

ist viel effektiver. Das ist die Formulierung, mit der man der Mustersuche sagen kann, dass sie zuerst nach der Zeichenkette Lösungen der Übungsaufgaben und danach nach den 50 davor stehenden Zeichen suchen soll. In fast allen Fällen muss nur das erste Musterelement geprüft werden, und die aufwändige Prüfung des zweiten Elementes, das zudem meist sowieso passt, kann eingespart werden.

Die einfangende Klammer im zweiten Musterelement ist notwendig, um den Treffer festzuhalten, da ein Vergleich nach links seinen Treffer nicht im Treffer des gesamten Ausdruckes abliefern.

Auf einem zurzeit gängigen Laptop ergeben sich mit den beiden Mustern Durchlaufzeiten von 0.581 zu 0.012 Sekunden. Die Suche mit dem ersten Muster dauert also fast 50-mal länger als die mit dem zweiten. Im folgenden PHP-Programm finden Sie nur die Messung zum ersten Muster:

```
$IN = fopen('../Kapitell/Kapitell.doc','r');
if ( $IN )                               # Datei offen?
{ $start = microtime();                   # Anfangszeit
  while ( !feof($IN) )                   # Dateiende erreicht
  ?
  { $Zeile = fgets($IN);                  # Zeile einlesen
    if ( preg_match(
      '/.{50}(?=Lösungen der Übungsaufgaben)/',$Zeile,$Tr)
      { print "$Tr[0]\r\n"; }             # Treffer ausgeben
    }                                     # Ende Zeilen lesen
  $ziel = microtime();                    # Endzeit
  $sZeit = preg_split('/ {1,}/',$start,2); # Zerlegung
  $zZeit = preg_split('/ {1,}/',$ziel,2);  # Zerlegung
  $sec = $zZeit[1] - $sZeit[1];           # Sekunden
  $msec = $zZeit[0] - $sZeit[0];         # Microsec
  $sec += $msec;
  print "$sec Sekunden \r\n";
}
else
{ print "Eingabe nicht lesbar.\r\n"; }    # Fehlermeldung
```

Leider setzen Vergleiche nach links voraus, dass die Länge der zu prüfenden Zeichen schon bei der Kompilierung des regulären Ausdruckes feststeht. Das war hier der Fall: Es wurde nach 50 Zeichen gesucht. In praktischen Anwendungen wird diese Voraussetzung aber nicht gegeben sein.

## Übungsaufgabe 4.1

Kann man in Beispiel 4.2 anstelle von `[[:alpha:]]{1,}` auch schreiben: `[[:alpha:]]{1,}`?

## 4.2 Nulltreffer und Nullmuster

Bisher haben wir nur dann von einem Treffer gesprochen, wenn dieser auch Zeichen enthielt. Im Folgenden wollen wir auch leere Zeichenketten als Treffer zulassen und sprechen dann von Nulltreffern. Anker tragen keine Zeichen zu einem Treffer bei, weshalb wir auch davon sprechen, dass Anker, bei positiver Prüfung, einen Nulltreffer liefern. Diese Sichtweise ist im Zusammenhang mit Substitutionen von Vorteil, dort wird der Treffer eines regulären Ausdrucks gegen einen Ersatztext ausgetauscht und das kann durchaus auch ein Nulltreffer sein. Letztlich ist dies aber nur eine Frage der Ausdrucksweise.

### Beispiel 4.6

Wir betrachten einen Text, in dem der Satz `Der Sommer war schön.vor-` kommt, und wollen diesen Satz ersetzen lassen durch `Der Sommer war schön, aber trocken.` Hier muss aus dem Suchtext nichts entfernt werden, es muss lediglich ein zusätzlicher Textteil zwischen `schön` und dem Punkt eingefügt werden. Wir verwenden daher einen regulären Ausdruck, der möglichst genau auf diesen Satz passt, aber einen Nulltreffer liefert. Dieser Nulltreffer wird dann ersetzt durch `aber trocken`. Ein solcher Ausdruck ist:

```
/(?<=Der Sommer war schön)(?=\.)/
```

Ein Nulltreffer wird erzielt, weil weder der Vergleich nach links noch der nach rechts verbrauchend ist. Der Nulltreffer wird hinter `schön` und vor dem Punkt erzielt. Die ganze Mustersuche läuft darauf hinaus, dass mit Position 0 beginnend auf jeder Zeichenposition geprüft wird, ob davor `Der Sommer war schön` und dahinter ein Punkt steht, die zweite Prüfung wird nur durchgeführt, wenn die erste erfolgreich war. Sobald eine solche Position gefunden wird, ist die Mustersuche erfolgreich beendet, und die Ersetzung beginnt. Eine entsprechende PHP-Anweisung lautet:

```
$Stext = 'Es war ein seltsames Jahr. Der Sommer war schön.';
print preg_replace('/(?<=Der Sommer war schön,)(?=\.)/',
    ' aber trocken',$Stext);
```

Da beide Musterelemente an derselben Position überprüft werden, kann man sich fragen, ob man die Reihenfolge dieser Musterelemente in der Musterkette auch vertauschen kann. Das ist für den Leser vielleicht verwirrend und sollte deshalb nicht getan werden, für das Programm ist es jedoch ohne Belang (abgesehen vom eventuell notwendigen Zeitaufwand).

```
$Stext = 'Es war ein seltsames Jahr. Der Sommer war schön.';
print preg_replace('/(=?\.)(<=Der Sommer war schön)/',
                  ', aber trocken',$Stext);
```

liefert dasselbe Ergebnis wie obige PHP-Anweisung.

Nulltreffer werden nicht nur von Ankern erzielt, sondern auch von quantifizierten Musterelementen, wenn dort der Wert 0 nicht ausgeschlossen wird. Man muss diese beiden Arten von Nulltreffern auseinander halten. Anker passen nur an bestimmten Positionen im Suchtext und prüfen Positionen ab. Quantifizierte Zeichenmuster oder Zeichenklassen stellen keine Bedingungen an die Position und prüfen Zeichen (nach rechts oder aber auf der aktuellen Zeichenposition) ab. Das Besondere an einem Musterelement wie `a{0}` oder auch `a{0,}`, was viel öfter auftreten wird als Ersteres, ist nur, dass die Prüfung auch dann positiv ausfällt, wenn rechts von der aktuellen Zeichenposition im Suchtext gar kein `a` steht. Solche Musterelemente werfen eine Reihe von Fragen auf. Viele davon erinnern zwar an die Frage, wie viele Engel auf einer Nadelspitze Platz haben, aber auf einige muss man dennoch eingehen.

Zunächst einmal findet eine Mustersuche mit einem Element wie `\d{0,}` im Suchtext

```
Hölderlin wurde am 20.3.1770 in Lauffen am Neckar geboren.
```

nicht etwa 20, sondern den Anfang des Suchtextes, da Perl-kompatible reguläre Ausdrücke von allen möglichen Treffern immer den am weitesten links stehenden finden.

Die PHP-Funktion `preg_replace` ersetzt alle Treffer eines Musters im Suchtext durch einen Ersatztext. Die Frage ist nun, was sind alle Treffer des Musters `//` oder alle Treffer des Musters `/1{0}/`. Natürlich passen solche »Nullmuster« auf kein Zeichen, sondern nur auf die Positionen vor und nach den Zeichen. Aber im Prinzip kann man »nichts« an derselben Position beliebig oft finden. Wie oft wird nun ein solcher Nulltreffer an derselben Position ersetzt?

### Beispiel 4.7

Die beiden print-Anweisungen:

```
$Stext = '123456';  
print preg_replace('/1{0}/','+', $Stext)."\r\n";  
print preg_replace('//','+', $Stext)."\r\n";
```

liefern jeweils die Ausgabe:

```
+1+2+3+4+5+6+
```

Man hat per Konvention festgelegt, dass eine leere Zeichenkette an ein und derselben Position nur einmal als Treffer erzielt werden kann und dass danach ein Positionsvorschub erfolgen muss.

Beachten Sie, dass diese Konvention für Nulltreffer des gesamten Ausdrucks gilt und bei einer einzelnen Mustersuche auch ohne Bedeutung ist. Anker können an derselben Position, innerhalb derselben Mustersuche, mehrfach geprüft werden, wie Sie in Beispiel 4.1 schon gesehen haben.

Bei der Zerlegung einer Zeichenkette in einzelne Zeichen ist die Verwendung von Nullmustern für die Bruchstellen nahe liegend.

### Beispiel 4.8

Die folgende Anweisungssequenz in Perl

```
$Stext = '123456';  
$" = ',';  
@Ziffern = split //, $Stext;  
print "@Ziffern\n";
```

liefert die Ausgabe:

```
1,2,3,4,5,6
```

`split` prüft, wo überall in der Zeichenkette eine leere Zeichenkette passt, das ist überall zwischen zwei Zeichen der Fall und nach der erwähnten Konvention auch nur einmal möglich. Das Ergebnis ist eine Liste mit den einzelnen Zeichen des zerlegten Textes. Diese Liste wird mit der nachfolgenden print-Anweisung ausgegeben. Der Wert der speziellen Perl-Variablen `$` gibt vor, welches Trennzeichen zwischen je zwei Listenelementen zu verwenden ist.

Wenn Sie das Ergebnis dieser Zerlegung mit dem der Substitution in Beispiel 4.2 vergleichen, fällt Ihnen sicher auf, dass bei der Substitution auch vor dem ersten und nach dem letzten Zeichen ein Nulltreffer gefunden wurde und dass

dementsprechend bei der Zerlegung auch acht Bruchstücke möglich wären, jeweils ein leeres vor der 1 und nach der 6 im Suchtext. Das ist tatsächlich eine Möglichkeit, die hier durch den speziellen Aufruf der `split`-Operation verhindert wurde. Eine Beschreibung der verschiedenen Aufrufarten von `split` finden Sie im Abschnitt 7.6.

Der Fall eines leeren regulären Ausdrucks (Nullmuster) wie `//`, der aber auch dann gegeben ist, wenn bei der Variablen-Interpolation von `/$Variable/` die Variable einen undefinierten oder leeren Wert hat, wird in den verschiedenen Sprachen unterschiedlich behandelt. Perl verwendet in diesem Falle zunächst den zuletzt erfolgreich ausgeführten regulären Ausdruck, der nicht in einem inneren Block versteckt ist; nur wenn ein solcher nicht vorliegt, wird ein leeres Muster verwendet. Das ist eine der vielen Ausnahmeregelungen in Perl, an die man im Ernstfall nicht denkt. PHP kennt diese Regelung glücklicherweise nicht.

### Beispiel 4.9

```
$St = 'abc1def2';
($Test = $St) =~ s/\d/#/; print "$Test\n";
($Test = $St) =~ s//#/; print "$Test\n";
```

liefert mit Perl 5.8.7 zweimal dieselbe Ausgabe

```
abc#def
```

weil zweimal dasselbe Muster verwendet wird. Dagegen liefert das PHP 5.0.4-Programm

```
$St = 'abc1def';
print preg_replace('/\d/', '#', $St)."\r\n";
print preg_replace('//', '#', $St)."\r\n";
```

die Ausgaben

```
abc#def
#a#b#c#1#d#e#f#
```

Die genannte Regelung für Nullmuster in Perl, also der eventuelle Ersatz durch ein zuvor verwendetes Muster, gilt für Mustersuche und Substitution, nicht aber für die Textzerlegung mit `split`, was es nicht einfacher macht, den Überblick zu behalten.

### Beispiel 4.10

```
$St = 'abcdef';           # Suchtext
($Test = $St) =~ s/\d/#/; # erfolgreiche Mustersuche
print "1) $Test\n";      # Ergebnis ausgeben
@Feld = split //, $St;   # Zerlegung mit leerem Muster
print "2) @Feld\n";     # Ergebnis ausgeben
```

liefert als Ausgabe:

```
1) abc#def
2) a b c d e f
```

Das Nullmuster wird bei `split` nicht ersetzt und führt zu einer Zerlegung in die einzelnen Zeichen.

Beachten Sie bitte, dass ein Nullmuster etwas ganz anderes ist als ein Muster, das einen Nulltreffer erzielt. Ein Nullmuster liegt vor, wenn nach der Variablen-Interpolation der reguläre Ausdruck aus einer leeren Zeichenkette besteht. Ein Muster, das einen Nulltreffer erzielt, kann dagegen mit beliebig vielen Zeichen formuliert werden.

Die Schachtelung von Quantifiern, die den Wert 0 zulassen, soll hier nur aus zwei Gründen erwähnt werden: Zum einen kann eine solche Schachtelung zu langen Suchzeiten führen, zum anderen hatten einige frühere Perl-Versionen in manchen Situationen Probleme mit solchen Konstrukten. So passte in Perl 5.8.4

```
/(\d{0}){1}/
```

nicht in 123. Ein Musterelement wie das angegebene wird man wohl nie brauchen.

### Übungsaufgabe 4.2

Welche Treffer erzielt `(\d{0,}\s{0,}){1,}` im Suchtext `12 3a b57 c`, und wie wirkt sich der äußere Quantifier aus? Formulieren Sie ein Muster mit einer Zeichenklasse, das dieselben Treffer in diesem Suchtext findet (jeweils mit globaler Mustersuche).

### 4.3 Anfang und Ende im Suchtext

Bei der Frage, wo man einen Treffer im Suchtext verankern kann, fallen einem wohl zuallererst Anfang bzw. Ende des Suchtextes ein. Es ist klar, was unter Anfang und Ende des Suchtextes jeweils zu verstehen ist: Eine Position, vor der

keine Zeichen stehen, ist der Anfang und eine solche, nach der keine Zeichen mehr folgen, das Ende des Suchtextes. Diese Formulierung passt auch auf leere Zeichenketten. Eine leere Zeichenkette umfasst zwar keine Zeichen, aber eine 0. Zeichenposition. Anfang und Ende einer leeren Zeichenkette fallen daher zusammen.

Ein `\A`-Anker passt auf den Anfang des Suchtextes und muss nach links prüfen, ein `\z`-Anker passt auf das Ende des Suchtextes und muss nach rechts prüfen. Für beide gibt es in jedem Suchtext eine eindeutig bestimmte Position, auf der sie passen.

Wenn ein Suchtext mit einem Zeilenvorschub endet, will man in vielen Fällen nicht am Ende des Suchtextes arbeiten, sondern vor dem abschließenden Zeilenvorschub. Wenn z.B. am Ende des Suchtextes ein Punkt vergessen wurde und nachträglich eingefügt werden soll, soll dies vor einem eventuellen Zeilenvorschub geschehen und nicht in der neuen Zeile.

Der Anker `\Z` passt auf das Ende des Suchtextes, und wenn dieser mit einem `\n`-Zeichen endet, auch auf die Position vor diesem `\n`-Zeichen.

Das ist ein bequemes Element in Perl und auch unter Unix, aber nicht unter PHP und Windows gemeinsam, da hier ein Zeilenende mit `\r\n` angegeben wird und `\Z` mitten zwischen diese beiden Zeichen passt. Wenn Sie unter PHP und Windows oder einem anderen Betriebssystem arbeiten, das nicht `\n` als Zeilenende verwendet, sind einige der Anker in den Perl-kompatiblen regulären Ausdrücken von wenig Nutzen.

Das betrifft auch den Anker `$`, der schon mehrfach verwendet wurde. Er passt ohne Einwirkung eines Modifiers, wie der `\Z`-Anker auch, auf das Ende des Suchtextes, aber auch vor ein eventuell am Ende stehendes `\n`. Das Verhalten des `$`-Ankers kann aber mit Hilfe des `m`-Modifiers gesteuert werden. Unter Einfluss eines `m`-Modifiers passt `$` vor jedem `\n` im Suchtext und unterscheidet sich dadurch von `\Z`.

PHP kennt einen `D`-Modifier. Unter Einfluss eines `D`-Modifiers passt ein `$`-Anker, wenn keine anderen Modifier auf ihn einwirken, nur auf das Ende des Suchtextes.

Wie der `$`-Anker kann auch das Verhalten des Ankers `^` durch einem `m`-Modifier gesteuert werden. Ohne `m`-Modifier passt `^` nur auf den Anfang des Suchtextes, mit `m`-Modifier dagegen hinter jedem `\n` im Suchtext. Beim `^`-Anker muss man unter PHP keinen Unterschied zwischen Unix und Windows berücksichtigen.

## 4.4 Anfang und Ende in einer Zeile

Wenn ein Suchtext Zeilen enthält, entsteht die Frage, wo eine Zeile beginnt und wo sie endet. Das ist nicht so einfach zu beantworten wie die Frage nach Anfang und Ende des Suchtextes, denn die Frage ist, ob man die jeweilige Zeichenfolge für den Zeilenumbruch, also z.B. `\n` oder `\r\n`, zur vorangehenden oder zur nachfolgenden Zeile rechnen soll oder eventuell zu keiner von beiden.

Betrachten wir dazu den Suchtext

```
Mit gelben Birnen hanget
Und voll mit wilden Rosen
Das Land in den See
```

und das Problem, an jedem Zeilenanfang zwei Unterstriche einzufugen. Wir erwarten dann als Resultat:

```
__Mit gelben Birnen hanget
__Und voll mit wilden Rosen
__Das Land in den See
```

Als Zeilenanfang sehen wir also an:

- ▶ den Anfang des Suchtextes
- ▶ jede Position nach einem Zeilenvorschub

Wenn der Suchtext nun mit einem Zeilenvorschub endet, soll die Position danach ein Zeilenanfang sein oder nicht? Tatsachlich liefern hier die Substitutionen in Perl 5.8.7 und PHP 5.0.4 unterschiedliche Antworten.

Als Anker, der auf einen Zeilenanfang passt wird `^` mit `m`-Modifizier verwendet. Mit `m`-Modifizier passt `^` hinter jedem `\n` im Suchtext.

### Beispiel 4.11

Das PHP-Programm

```
$St =
"Mit gelben Birnen hanget\r\n".
"und voll mit wilden Rosen\r\n".
"das Land in den See.\r\n";
print preg_replace('/^/m', '__',$St);
```

liefert die Ausgabe

```
__Mit gelben Birnen hanget
__und voll mit wilden Rosen
```

```
__das Land in den See.
```

```
—
```

### Das Perl-Programm

```
$St =
"Mit gelben Birnen hanget\n".
"und voll mit wilden Rosen\n".
"das Land in den See.\n";
$St =~ s/^\_/mg;
print "$St";
```

### liefert dagegen

```
__Mit gelben Birnen hanget
__und voll mit wilden Rosen
__das Land in den See.
```

Hier wird eine Position nach einem Zeilenvorschub nicht als Zeilenanfang betrachtet, wenn sie mit dem Ende des Suchtextes zusammenfallt.

Als Zeilenende sieht man jede Position vor einem Zeilenvorschub des jeweiligen Betriebssystems an. Nun kommt es nicht selten vor, dass ein Zeilenvorschub am Ende des Suchtextes fehlt. Ein Anker, der auf die Zeilenenden passt, sollte also auch am Ende des Suchtextes passen. Wobei dabei wieder die Frage ist, ob er auch dann passen soll, wenn am Ende des Suchtextes ein Zeilenvorschub steht.

Als Anker wird zu diesem Zwecke \$ zusammen mit dem m-Modifier verwendet. Allerdings setzt der voraus, dass Zeilenvorschube durch \n angegeben werden. Unter PHP und Windows ist das nicht gegeben. Dieser Anker passt bei Verwendung des m-Modifiers vor jedem \n und am Ende des Suchtextes.

### Beispiel 4.12

Unter Unix liefert sowohl das PHP-Programm

```
$St =
"Mit gelben Birnen hanget\n".
"und voll mit wilden Rosen\n".
"das Land in den See.\n";
print preg_replace('/$/m', '__', $St);
```

als auch das Perl-Programm

```

$St =
"Mit gelben Birnen hanget\n".
"und voll mit wilden Rosen\n".
"das Land in den See.\n";
$St =~ s/$/___/mg;
print "$St";

```

die Ausgabe

```

Mit gelben Birnen hanget___
und voll mit wilden Rosen___
das Land in den See.___
___

```

Wenn man die uberflussige vierte Zeile nicht ausgeben mochte, die von einer Ersetzung am Ende des Suchtextes stammt, kann man z.B. den abschließenden Zeilenvorschub am Ende des Suchtextes eliminieren, was sowohl in PHP als auch in Perl mit der `chop`-Anweisung moglich ist. Das hat allerdings den Nachteil, dass der Suchtext dadurch verandert wird. Besser ist es da, ein anderes Muster fur die Substitution vorzugeben:

```
/(?=\n)/
```

ist in Perl ein Muster, welches das gewunschte Ergebnis liefert. In PHP liefert es dieses Ergebnis nur unter Unix, da unter Windows die Zeichenfolge `\r\n` als Zeilenvorschub dient.

Ein Vergleich wie der oben angegebene lasst sich in einem regularen Ausdruck auch verneinen. Das Ergebnis von `(?=\n)` ist wahr, wenn rechts von der aktuellen Position ein `\n` folgt. Das Ergebnis von `(?!n)` ist dagegen wahr, wenn rechts von der aktuellen Position kein `\n` folgt. Solche Vergleiche mussen nicht unbedingt mit Zeichenmustern oder Zeichenklassen erfolgen, es ist auch moglich, dass sie mit Ankern erfolgen. Fur ein Konstrukt wie `(?=\z)` gibt es keinen Bedarf, da es zu `\z` aquivalent ist, ein Element `(?!z)` kann man dagegen anders nicht formulieren, es passt auf jeder Position auer am Ende des Suchtextes.

Damit hat man ein weiteres Muster, um in Beispiel 4.11 die uberflussige Zeile am Ende des Suchtextes zu vermeiden:

```
/$(?!z)/
```

ist eine Musterkette mit zwei Gliedern, die aber beide an derselben Zeichenposition prufen. Wenn die Prufung mit `$` positiv ausfallt, wird zusatzlich gepruft, ob nicht etwa an dieser Position das Ende des Suchtextes vor-

liegt. Nur wenn diese Prüfung ebenfalls positiv ausfällt, das Suchtextende also nicht vorliegt, fällt die gesamte Mustersuche an dieser Position positiv aus.

Ohne jegliche Einwirkung eines Modifiers passt ein `$`-Anker auf das Ende des Suchtextes oder vor ein `\n`, das am Ende des Suchtextes steht.

Die Perl-Anweisungen

```
$st = "Mit gelben Birnen hnget\nUnd voll mit wilden Rosen\n";
$st =~ s/$!/g;
print "$st\n";
```

liefern dementsprechend die Ausgabe

```
Mit gelben Birnen hnget
Und voll mit wilden Rosen!
!
```

Mit `m`-Modifier passt der `$`-Anker auch vor `\n`-Zeichen im Inneren des Suchtextes. Wenn im obigen Beispiel die Substitution durch

```
$st =~ s/$!/mg;
```

ersetzt wird, wird daher auch nach der ersten Zeile ein Ausrufungszeichen ausgegeben.

Sie hatten im letzten Abschnitt den `D`-Modifier von PHP kennen gelernt. `m`- und `D`-Modifier widersprechen sich offensichtlich, geben Sie also jeweils nur einen von beiden an.

Ein `^`-Anker passt ohne jegliche Einwirkung eines Modifiers nur auf den Anfang des Suchtextes, unter der Wirkung eines `m`-Modifiers auch auf Zeilenanfnge (hinter `\n`-Zeichen) im Inneren des Suchtextes.

Eng mit der Zeilenstruktur verbunden ist die mit einem Punkt bezeichnete Zeichenklasse, da ein Punkt auf alle Zeichen auer `\n` passt, sofern das Verhalten dieser Zeichenklasse nicht durch einen `s`-Modifier abgewandelt wird. Auch hier muss man aufpassen, wenn PHP unter Windows benutzt wird.

## Zeilenenden unter PHP und Windows

Es wurde schon darauf hingewiesen, dass der Zeilenvorschub unter verschiedenen Plattformen mit unterschiedlichen Zeichenfolgen beschrieben wird. Perl lost dieses Problem in seinem Textmodus, indem es bei der Eingabe die plattformabhngigen Zeilenvorschube in `\n` und bei der Ausgabe jedes `\n` in die plattformspezifischen Zeilenvorschube umsetzt.

PHP tut dies nicht. Das bedeutet, dass man unter Windows mit der Zeichenfolge `\r\n` arbeiten muss.

Perl-kompatible reguläre Ausdrücke sind in Bezug auf die Zeilenstruktur mit dem Textmodus von Perl verbunden. So passt die mit einem Punkt bezeichnete Zeichenklasse ohne `s`-Modifier auf alle Zeichen außer `\n`. Unter Perl heißt dies, auf alle Zeichen einer Zeile, aber nicht auf den Zeilenvorschub. Unter PHP und Windows wäre das Äquivalent, dass der Punkt auf alle Zeichen außer der Zeichenfolge `\r\n` passt. Das ist aber nicht so. Der Punkt passt auf `\r`, so dass diese Zeichenklasse unter PHP und Windows wenig mit der Zeilenstruktur des Suchtextes zu tun hat.

### Beispiel 4.13

Dieses Problem begegnet uns auch bei der Verwendung des `$`-Ankers wieder:

```
$St = "1a\r\n2b\r\n";
$out=preg_replace('/$/m','!',$St); # Substitution
print "$out\r\n\r\n";           # Ausgabe: veränderte Kette
for($i=0;$i<strlen($out);$i++)
{ $sign = substr($out,$i,1);     # einzelne Zeichen in $out
  $cp   = ord($sign);           # entsprechende Codepunkte
  if ( $cp == 10 ) { $sign = '\n'; } # für Ausgabe ersetzen
  elseif ( $cp==13 ) { $sign = '\r'; } # für Ausgabe ersetzen
  printf("%2d %4d %s\r\n",$i,$cp,$sign);
}
```

Die Absicht bei der Substitution in Zeile 2 ist wohl, vor jedem Zeilenende ein `!`-Zeichen einzufügen. Man würde dann als Ausgabe erhalten:

```
1a!
2b!
```

Tatsächlich sieht man unter Windows auf dem Terminal aber die Ausgabe:

```
!a
!b
!

0  49  1
1  97  a
2  13  \r
3  33  !
4  10  \n
```

```

5  50  2
6  98  b
7  13  \r
8  33  !
9  10  \n
10 33  !

```

Wie man am zweiten Teil der Ausgabe erkennen kann, in dem die ausgegebenen Zeichen und ihre Codepunkte aufgelistet werden, wird das Ausrufezeichen zwischen `\r` und `\n` eingeschoben. Dadurch wird `\r` bei der Ausgabe als normales »Carriage Return« behandelt, die Ausgabe setzt also an den Zeilenanfang zurück, und das nachfolgend ausgegebene Ausrufezeichen überschreibt das erste Zeichen der schon ausgegebenen Zeile. Richtet man die Ausgabe in eine Datei, so erscheint:

```
1a!b!
```

Und natürlich passt auch der `\Z`-Anker vor einem abschließenden `\n` im Suchtext, aber nicht vor einem `\r`, wenn der Suchtext mit `\r\n` endet. Wenn man ein unter Windows entwickeltes PHP-Skript auf einen Unix-Server lädt, muss man die unterschiedlichen Zeilenenden berücksichtigen.

#### Perl Beispiel 4.14

Mit diesem PHP-Programm soll aus allen Zeilen einer Datei, die mit einer Zahl enden, diese Zahl extrahiert und ausgegeben werden. Die folgende Lösung funktioniert unter Unix, nicht jedoch unter Windows.

```

$in = fopen('in.txt','r');           # Datei zum Lesen öffnen
while ( $Zeile = fgets($in) )       # Einlese-Schleife
{ if (preg_match('/\d{1,}$/',$Zeile,$Treffer))
  { print "$Treffer[0]\n" }
}

```

Verwendet man `/\d{1,}\r\n/` anstelle des Musters `/\d{1,}$/`, so funktioniert es unter Windows, aber nicht unter Unix. Eine Lösung, die auf beiden Plattformen läuft, wäre `/\d{1,}\r{0,1}\n/`. Hier wird nach einer Zahl gesucht, die entweder vor einer Zeichenfolge `\r\n` oder aber vor einem `\n` steht.

Eine andere Lösung wäre, zunächst alle »weißen Zeichen« am Ende einer Zeile abzuschneiden, das kann man mit einer Substitution machen oder auch mit der Anweisung:

```
$Zeile = rtrim($Zeile);
```

Man kann solche Ersatzlösungen aber nicht mechanisch anwenden, schauen Sie sich dazu die Übungsaufgabe 4.2 an.

### **Übungsaufgabe 4.3**

In einer Variablen seien mehrere Zeilen eines Textes abgespeichert, wobei offen ist, ob die letzte Zeile mit einem Zeilenvorschub abgeschlossen ist oder nicht.

Aus dieser Variablen soll die letzte Zeile extrahiert werden. Mit welchem Muster kann dies erfolgen?

### **Übungsaufgabe 4.4**

In einer Datei sollen alle Zeilen, die nicht mit einem Punkt oder Doppelpunkt enden, durch eine Leerstelle und nachfolgendes `\` verlängert werden. Welche Substitution kann man verwenden, die sowohl unter Unix als auch unter Windows für PHP und Perl geeignet ist? Die Substitution soll auch dann durchgeführt werden, wenn die letzte Zeile nicht mit einem Zeilenvorschub endet.

### **Übungsaufgabe 4.5**

Mit einem Programm soll unter Unix eine Datei eingelesen werden, die mehrere Zeilen enthält, darunter auch Leerzeilen. Mit Hilfe einer Substitution soll jede Zeile, die nicht leer ist und nicht schon mit einem Punkt endet, mit einem solchen abgeschlossen werden.

## **4.5 Wortgrenzen**

Neben den Zeichen und Zeilen kennen wir innerhalb von Texten auch Wörter. Ein Wort umfasst keine Leerstellen, beginnt in der Regel nach einer Leerstelle und endet vor einer solchen, diese Positionen bezeichnen wir als Wortgrenzen. Allerdings kennen wir auch andere Arten von Wortgrenzen, ein Wort kann z.B. vor einem Punkt oder Komma enden oder direkt am Anfang eines Textes beginnen. Allgemein bezeichnen wir jede Position zwischen einem Wort- und Nicht-Wortzeichen als Wortgrenze, dabei ist es egal, ob das Wortzeichen vor oder nach dem Nicht-Wortzeichen steht. Das bedeutet z.B. dass auch in Wörtern mit Bindestrich vor und nach dem Bindestrich eine Wortgrenze vorliegt und dass Wortgrenzen keinen Unterschied zwischen Wortanfang und Wortende machen.

Jede Position in einem Suchtext, einschließlich der ersten und letzten, ist entweder eine Wortgrenze oder eine Nicht-Wortgrenze. Anfang und Ende des Suchtextes werden so behandelt, als ob dieser von Nicht-Wortzeichen umgeben wäre.

In Perl-kompatiblen regulären Ausdrücken passt das Musterelement `\b` auf eine Wortgrenze, das Musterelement `\B` auf eine Zeichengrenze, an der keine Wortgrenze vorliegt. Sowohl `\b` als auch `\B` vergleichen nach rechts und links von der aktuellen Zeichenposition und ändern bei positiv verlaufener Prüfung nicht die aktuelle Zeichenposition.

Das Problem mit Wortgrenzen ist, dass sie über die Perl-Zeichenklasse `\w` definiert sind, die zum einen auf das Zeichen `_` passt, zum anderen aber in der locale `C` nicht auf die Zeichen ab Codepunkt 128 im ISO 8859-1-Zeichenvorrat.

Natürlich kann man auf Wortgrenzen in regulären Ausdrücken verzichten. Es ist nicht schwer, sie durch andere geeignete Musterelemente zu ersetzen, aber meistens werden dann Zeichenmuster oder Zeichenklassen verwendet, die ungewünschte Treffer erzielen und die man z. B. bei einer Substitution wieder herausfiltern muss. Auch wenn man nicht in der locale `C` arbeitet, bleiben bei der Suche nach Wortgrenzen Probleme bestehen. Eines davon ist die Klassifizierung des versteckten Trennstriches (ISO 8859-1: `0xAAC`) als Nicht-Wortzeichen. Sobald ein solches Zeichen in einem Wort auftaucht, erkennen die regulären Ausdrücke dort Wortgrenzen. Die Positionen, auf die `\b` jeweils passt, sind nicht immer die, die man als Wortgrenzen gerne hätte. Wünschenswert wären hier Steuerungsmöglichkeiten durch die Anwender, entsprechende Modifier gibt es zurzeit aber nicht.

Die Escape-Sequenz `\b` hat nur innerhalb eines regulären Ausdrucks und da nur außerhalb frei definierbarer Zeichenklassen die Bedeutung einer Wortgrenze. In `"`-begrenzten Zeichenketten und in frei definierbaren Zeichenklassen steht `\b` für ein Backspace-Zeichen. Man muss deshalb nicht nur aufpassen, wo man diese Escape-Sequenz verwendet, sondern auch darauf, wann sie ausgewertet wird. Die folgende Mustersuche findet keinen Treffer:

```
$Suchtext = 'Mit gelben Birnen hängt ' ;
$Muster = "\b[A-Z]\w{0,}\b";
@Treffer = $Suchtext = m/$Muster/g;
```

da die Escape-Sequenzen `\b` schon bei der Auswertung der `"`-begrenzten Zeichenkette in Backspace-Zeichen bzw. deren Codierung umgesetzt werden. Die Mustersuche

```
$Suchtext = 'Mit gelben Birnen hanget ';
@Treffer = $Suchtext = m/\b[A-Z]\w{0,}\b/g;
```

findet dagegen die beiden Treffer »Mit« und »Birnen«.

### ubungsaufgabe 4.6

Was wird mit den folgenden Mustersuchen abgepruft, welche Werte liefern die Ausdrucke, und wo passen die Muster?

```
'\b' =~ m/\b/           # Perl
"\b" =~ m/\b/           # Perl
preg_match('/\b/', '\b') # PHP
preg_match('/\b/', "\b") # PHP
preg_match("/\b/", '\b') # PHP
preg_match("/\b/", "\b") # PHP
```

### ubungsaufgabe 4.7

Eine Wortgrenze kann ein Wortanfang oder ein Wortende sein. Mit welchem Muster kann man diese beiden Wortgrenzen unterscheiden?

## 4.6 Probleme mit Ankern

Anker, die auf den Anfang des Suchtextes passen sollen, durfen in einer Musterkette offensichtlich keine verbrauchenden Musterelemente vor sich haben, da diese die aktuelle Position nach rechts verschieben wurden. Nicht verbrauchende Musterelemente sind vor einem entsprechenden Anker aber moglich. In Frage kommen hauptsachlich Vergleiche nach rechts oder aber Tests auf Wortgrenzen, obwohl man zu solchen Konstrukten meist bessere Alternativen findet.

```
/(?=[AEIOU])^\w{1,}/
```

ist jedenfalls ein zulassiger regularer Ausdruck, der auf das erste Wort eines Suchtextes passt, sofern dieses mit einem grogeschriebenen (deutschen) Vokal anfangt. Die Alternative dazu:

```
/[AEIOU]\w{0,}/
```

werden manche Leser aber vorziehen. Ein regularer Ausdruck wie

```
/.^\w{0,}/
```

ist zwar in PHP und Perl auch zulässig, kann aber niemals passen, weil `^` frühestens ab Position 1 im Suchtext geprüft wird. Analog kann ein `\z`-Anker sinnvollerweise nur am Ende einer Musterkette stehen, eventuell noch gefolgt von anderen nicht verbrauchenden Musterelementen.

```
/\z\b/
```

passt auf das Ende des Suchtextes, sofern davor ein Wortzeichen steht. Eine Folge von nichtverbrauchenden Musterelementen führt eine mit »und« verknüpfte Prüfung auf der aktuellen Position durch, und dabei ist die Reihenfolge egal.

```
/\zende/
```

ist ein in PHP und Perl zulässiges Muster, wird jedoch nie passen, da hinter dem Ende des Suchtextes keine Zeichenfolge `ende` im Suchtext folgen kann. Was für `\z` gilt, gilt auch für `$`, wenn dieser Anker mit `D`-Modifier in PHP benutzt wird. `\Z`- und `$`-Anker passen ohne Modifier auf das Ende des Suchtextes, und wenn dieser mit einem `\n` endet, auch auf die Position vor diesem Zeichen. Muster wie:

```
/\Z\n/, /\Z\s/
```

sind daher durchaus sinnvoll. Aufpassen muss man beim `$`-Anker allerdings, dass er nicht in einem Konstrukt erscheint, das mit einem Variablennamen verwechselt werden kann. Das gilt insbesondere für Perl mit seinen vielen speziellen Variablennamen, die nur aus einem `$` und einem weiteren Sonderzeichen (Satzzeichen, Punctuation) bestehen. `^` und `\b` können ebenso mitten in einer Musterkette vorkommen, bereiten aber keine Probleme.

Die Mehrdeutigkeit von Zeichenketten, die mit einem `$`-Zeichen beginnen, macht ihre Interpretation in regulären Ausdrücken für Menschen schwierig. Das Muster:

```
/\$. /s
```

würden Sie vermutlich als `$`-Anker interpretieren, gefolgt von einer Punkt-Zeichenklasse, die wegen des `s`-Modifiers auch auf ein `\n` passt. Im folgenden Perl-Beispiel ist dies allerdings nicht der Fall.

#### Beispiel 4.15

```
open(IN, 'in.txt');
$Zeile = <IN>;
```

```
if ( $Zeile =~ m/$./s ) { print "passt\r\n"; }
else { print "$.\n"; }
```

liefert irgendeine Ausgabe, da in Perl eine Variable `$.` definiert ist, die jeweils die Eingabezeilennummer des letzten Dateihandles, aus dem gelesen wurde, als Wert hat. Mit Mustern und regulären Ausdrücken hat dies eigentlich nichts zu tun, da die Variablen-Interpolation stattfindet, bevor der Ausdruck als regulärer Ausdruck interpretiert wird, aber bei der Formulierung eines regulären Ausdrucks muss man einfach darauf achten.

Falls Sie auf die Idee kommen, in Beispiel 4.15 anstelle des Punktes direkt `\n` anzugeben, tappen Sie in Perl in die nächste Falle, sie haben dann den regulären Ausdruck:

```
/$\n/
```

Und da es in Perl aber eine Variable mit der Bezeichnung `$$` gibt, deren Wert das Trennzeichen für Ausgabesätze ist, wird auch dieser Ausdruck bei der Variablensubstitution abgeändert, was zu ganz unerwarteten Treffern führen kann.

#### Beispiel 4.16

```
$Zeile = "Versammelt waren die Todeshelden
Itzt, da er scheidend
Noch einmal ihnen erschien.\n";
$Zeile =~ s/$\n//;
print "$Zeile\n";
```

entfernt nicht etwa den Zeilenvorschub am Ende des Suchtextes, sondern das erste `n` im Suchtext. Der Wert der Variablen `$$` ist in diesem Falle undefiniert, weshalb aus `/$\n/` einfach `/n/` wird.

Wenn man einen Zeilenvorschub am Ende des Suchtextes mit einer Substitution entfernen möchte (es gibt auch andere Möglichkeiten), kann man das natürlich mit dem Muster `\n\z` machen. Man kann in Perl die Substitutionsanweisung auch mit einfachen Apostrophen als Grenzmarkierungen für das Muster und den Ersatztext angeben, dann unterbleibt die Variablen-Interpolation. Die Perl-Substitution

```
$Zeile =~ s'$$\n'';
```

entspricht daher der PHP-Substitution

```
preg_replace('/$$\n/', '', $Zeile, 1);
```

Beide führen zum gewünschten Ergebnis.

Die vielen speziellen Variablennamen in Perl bilden für die Arbeit mit regulären Ausdrücken einfach Fallen. Daran ändert auch die Tatsache nichts, dass man einige Zeichenfolgen, die auch spezielle Variablennamen sind, in regulären Ausdrücken explizit nicht als solche interpretiert, weil sie viel häufiger in anderer Bedeutung vorkommen.

### Beispiel 4.17

```
$Zeile = "Versammelt waren die Todeshelden
Itzt, da er scheidend
Noch einmal ihnen erschien.\n";
$Zeile =~ s/(\w{1,}\. $)//;
print "$Zeile\n";
```

entfernt das letzte Wort und den anschließenden Punkt aus dem Suchtext, dies ist auch das, was man erwartet. Aber es gibt in Perl auch eine Variable \$), die hier aber nicht interpoliert wird, ihr Wert ist die effektive Gruppen-ID des laufenden Prozesses.

\$) wird ebenso wie \$| in regulären Ausdrücken nicht interpoliert, weil solche Zeichenfolgen relativ häufig in regulären Ausdrücken in anderer Bedeutung auftreten, sei es als Test auf ein Zeilenende in einer Klammer oder in einer Alternative. Man soll sich daher merken: \$) und \$| kommen oft genug in anderer Bedeutung vor, so dass sie nicht als Variablennamen angesehen werden, \$\ und viele andere kommen nicht oft genug vor und unterliegen deshalb der Variablen-Interpolation. Ob solche Regeln das Erlernen von Perl erleichtern, soll hier nicht diskutiert werden.

Anker, die auf einen Zeilenanfang oder ein Zeilenende passen, können mitten in einer Musterkette auftreten. Im folgenden Beispiel werden aus einem vierzeiligen Suchtext die Zeilen zwei und drei ausgegeben.

### Beispiel 4.18

```
$Suchtext =
"neue Zeile 1\nneue Zeile 2\nneue Zeile 3\nneue Zeile 4\n";
$Muster = '$\n(.{1,})$\n.{1,})$';
$Suchtext =~ m/$Muster/m;
print "$1\n";
```

Das Problem mit der Mehrdeutigkeit von Konstrukten wie \$\n wurde hier dadurch vermieden, dass das ganze Muster als Wert einer '-begrenzten Perl-Zeichenkette angegeben wurde, was in vielen Situationen ein brauchbarer Ausweg ist.

## 4.7 Vergleich nach rechts, verbrauchte Positionen und Zeichen

Eine Vergleich nach rechts (*lookahead assertion* oder eine Versicherung nach rechts) ist ein nicht verbrauchendes Musterelement, das 0 oder mehr Zeichen rechts von der aktuellen Position im Suchtext prüfen kann, diese aber nicht verbraucht. Die Formulierung kann positiv oder negativ erfolgen:

```
(?=Vergleichsmuster)
(?!Vergleichsmuster)
```

Ein Vergleich (`?=Vergleichsmuster`) ist wahr, wenn das angegebene Muster rechts von der aktuellen Position passt, sonst nicht. Dagegen ist ein Vergleich (`?!Vergleichsmuster`) wahr, wenn das angegebene Muster rechts von der aktuellen Position nicht passt. Die einleitende Zeichenfolge (`?=` bzw. `?!`) muss direkt nacheinander geschrieben werden, also ohne eingeschobene weiße Zeichen. In einer Formulierung:

```
/...( ?.....) .../
```

wird die Klammer immer als einfangende Klammer und das Fragezeichen als Quantifier (äquivalent zu `{0,1}`) interpretiert. Das gilt auch bei der Verwendung eines `x`-Modifiers. In der Formulierung:

```
/...( ?.....)...../x
```

fehlt der Operand zum Quantifier, und die Formulierung ist falsch. PHP gibt hier mit `preg_match` und `preg_match_all` eine Warnung aus und findet keinen Treffer, `preg_replace` behandelt diese Formulierung so, als wäre kein `x`-Modifikator angegeben.

Perl behandelt diesen Fall aber so, als ob die Leerstelle zwischen öffnender Klammer und Fragezeichen nicht existieren würde. Solche Formulierungen, auf welche die verschiedenen Werkzeuge irgendwie reagieren, nicht unbedingt aber mit einer Fehlermeldung, sollte man als Fehler ansehen.

Eine fehlerhafte Formulierung ist es auch, wenn nach der einleitenden Zeichenfolge (`?` ein weißes Zeichen eingefügt wird, dies wird aber glücklicherweise sowohl von PHP als auch von Perl immer als Fehler erkannt. Die Zeichenfolge (`?`) tritt bei regulären Ausdrücken recht häufig auf und ist nur mit bestimmten Zeichen auf der nachfolgenden Zeichenposition erlaubt. Das können bestimmte Modifier sein, ein Doppelpunkt, bestimmte Sonderzeichen usw.

Als Vergleichsmuster sind beliebige Muster möglich, auch solche mit unbestimmter Trefferlänge, also Muster mit Quantifiern wie {1,} oder {2,4} und auch solche mit weiteren Vergleichen nach rechts oder links. Vergleiche nach rechts und links können also geschachtelt auftreten.

In einem Vergleich nach rechts können einfangende Klammern auftreten. Je nach Werkzeug werden deren Treffer außerhalb der Mustersuche bzw. Substitution zugänglich gemacht oder nicht.

### Beispiel 4.19

```
$St = "Die geheimnisvolle Legende von König Arthur.";
preg_match('/
    [A-Z]           # passt auf: D
    [a-z]{0,}      # passt auf: ie
    (?=           # Anfang Vergleich nach rechts
    \             # maskierte Leerstelle
    [a-z]{0,}      # passt auf: geheimnisvolle
    \             # maskierte Leerstelle
    (             # Anfang einfangende Klammer
    [A-Z][a-z]{0,} # passt auf: Legende
    )             # Ende einfangende Klammer
    )             # Ende Vergleich nach rechts
/x', $St, $Treffer);
foreach ( $Treffer as $treffer )
{ print "$treffer "; }
```

Die Mustersuche erzielt mit den beiden ersten Elementen, die verbrauchend sind, den Treffer *Die*. Der Vergleich nach rechts passt dann auf *geheimnisvolle Legende*. `preg_match` gibt zuerst den Treffer des gesamten Ausdrucks, danach der Reihe nach die Treffer der einfangenden Klammern an die Trefferliste, ausgegeben wird also »Die« und »Legende«.

Die Treffer eines Vergleiches werden nicht an den Treffer des Ausdrucks angehängt. Sie werden in der Regel auch nicht abgespeichert, es sei denn, sie stehen in einfangenden Klammern.

Wie Sie sehen, kann eine Mustersuche neben ihrem Ergebnis, das *wahr* oder *falsch* ist, auch unterschiedliche Treffer liefern:

- ▶ den Treffer des regulären Ausdrucks, der immer eine zusammenhängende Teilkette des Suchtextes ist, eventuell auch eine leere,
- ▶ die Treffer der einzelnen einfangenden Klammern im Muster. Diese können im Treffer des gesamten Ausdruckes enthalten sein, müssen dies aber nicht. Die Treffer jeder einfangenden Klammer sind ebenfalls zusammenhängende Teilketten des Suchtextes. Die Treffer aufeinander folgender einfangender Klammern müssen jedoch nicht aufeinander folgende Teilketten des Suchtextes sein.

Die Mustersuche von Perl liefert nur die Treffer der einfangenden Klammern ab; wenn man auch den Treffer des gesamten Ausdruckes haben möchte, muss man diesen in der Regel mit einfangenden Klammern umschließen.

Bei einer Mustersuche nach allen Treffern im Suchtext oder bei einer globalen Substitution ist jeweils nur der Treffer des regulären Ausdruckes für die nachfolgende Suche verbraucht, nicht unbedingt aber die Treffer der einfangenden Klammern.

#### Beispiel 4.20

```
$St = "Die geheimnisvolle Legende vom König Arthur.";
print preg_replace(
    '/
    [A-Z]                # Großbuchstabe
    [a-z]{0,}           # Kleinbuchstabe
    (?=\ [a-z]{1,}\ )  # nachfolgendes Wort
/x', 'DIE', $St);
```

`preg_replace` führt eine globale Substitution durch, besser gesagt, eine Folge von Substitutionen, bis alle möglichen Treffer im (ursprünglichen) Suchtext ersetzt sind. Wir sprechen hier vom ursprünglichen Suchtext, weil nur da nach möglichen Treffern gesucht wird, nicht etwa auch in dem durch Substitution entstandenen veränderten Suchtext.

Die erste Mustersuche findet mit dem verbrauchenden Teil des Musters die Zeichenkette `Die` im Suchtext. Der Vergleich nach rechts ist eine Randbedingung, die z.B. ausschließt, dass auch `Arthur` gefunden werden kann. Die einfangende Klammer im Vergleich nach rechts hat für diese Substitution keine Bedeutung, man könnte allerdings ihren Wert, also das, was mit ihr eingefangen wurde, im Ersatztext verwenden. Nach der ersten Mustersuche:

- ▶ ist die Teilkette `Die` des Suchtextes verbraucht,
- ▶ wird der von der Mustersuche verbrauchte Teil des Suchtextes gegen den Ersatztext ausgetauscht,
- ▶ wird die Startposition für die nächste Mustersuche direkt hinter den verbrauchten Teil des Suchtextes verschoben.

Die zweite Mustersuche passt mit ihrem verbrauchenden Musterteil auf `Legende` und mit dem Vergleich nach rechts auf `vom König`, findet also einen Treffer und verbraucht `Legende`. Dieser verbrauchte Treffer wird ausgetauscht gegen `DIE`, und die dritte Mustersuche beginnt direkt hinter `Legende`, findet aber keinen Treffer. Das Ergebnis der globalen Substitution ist also:

```
DIE geheimnisvolle DIE vom König Arthur.
```

### Beispiel 4.21

Im folgenden Beispiel wird an jeder aufgefundenen Wortgrenze ein `|`-Zeichen eingesetzt. Bei globalen Mustersuchen und Substitutionen kann per Konvention an ein und derselben Position nur ein Nulltreffer erzielt werden.

```
$St = "Die geheimnisvolle Legende vom König Arthur.";
print preg_replace('/\b/', '|', $St);
```

liefert, sofern man nicht gerade in der locale `C` arbeitet, die Ausgabe:

```
|Die| |geheimnisvolle| |Legende| |vom| |König| |Arthur|.
```

Die erste Mustersuche passt auf den Anfang des Suchtextes und erzielt dort einen Nulltreffer, der diese Position verbraucht. Die nachfolgende Mustersuche muss also auf Position 1 beginnen.

Beachten Sie aber, dass diese Konvention nur für die nachfolgende Mustersuche in einer Folge von Mustersuchen gilt. Innerhalb einer einzelnen Mustersuche, etwa mit dem Muster:

```
/\b\w/
```

gilt diese Konvention nicht. `\b` passt auch hier auf den Anfang des Suchtextes, verbraucht diese Position aber nicht, die Prüfung zu `\w` beginnt an derselben Position, an der `\b` geprüft wurde.

Bei der Suche nach allen Treffern eines regulären Ausdrucks können sich die verbrauchten Treffer nicht überlappen, da ein Zeichen des Suchtextes nur für einen Treffer verbraucht werden kann, die Treffer einfangender Klammern können sich dagegen sehr wohl überlappen.

## Beispiel 4.22

Im Suchtext 123456789 sollen alle Treffern des Musters `(?=(\d{3}))` durch den Treffer der einfangenden Klammer im Vergleich nach rechts ersetzt werden.

```
$Suchtext = '123456789';
$Muster   = '(?=(\d{3}))';
print preg_replace("/$Muster/", "$1", $Suchtext);
```

Das Muster passt auf alle Zeichengrenzen, hinter denen noch vier Zeichen, davon drei Ziffern, folgen, also auf die Zeichengrenzen vor 1 bis 6.

```
 1 2 3 4 5 6 7 8 9
↑ ↑ ↑ ↑ ↑ ↑
```

und erzielt dort jeweils Nulltreffer. Die einfangende Klammer im Vergleich nach rechts passt immer auf die drei Ziffern hinter den Positionen 1 bis 3 relativ zur aktuellen Zeichenposition. Die erste Mustersuche passt auf Position 0, die einfangende Klammer auf die Ziffern 234.

```
 1 2 3 4 5 6 7 8 9
↑  ▲ ▲ ▲
```

Hinter Position 0, also vor der 1, wird daher 234 eingesetzt. Als Ergebniskette hat man nach dieser ersten Mustersuche also:

```
234123456789
▲▲▲
```

Nach dieser Mustersuche ist Position 0 im Original-Ersatztext nach Konvention verbraucht, da dort ein Nulltreffer erzielt wurde, die Startposition für die zweite Mustersuche ist Position 1.

```
 1 2 3 4 5 6 7 8 9
   ↑  ▲ ▲ ▲
```

Die einfangende Klammer passt auf 345, und das Ergebnis der beiden ersten Ersetzungen ist:

```
234134523456789
      ▲▲▲
```

In dieser Weise wird läuft die Prozedur weiter, bis die aktuelle Position auf der Position vor der 7 angelangt ist, hier passt das Muster nicht mehr. Das Ergebnis der durchgeführten Substitutionen ist:

234134524563567467857896789

▲▲▲ ▲▲▲ ▲▲▲ ▲▲▲ ▲▲▲ ▲▲▲

Die eingesetzten Zeichen sind markiert. Die Bezeichnung »nach rechts vergleichen« darf nicht missverstanden werden, der Vergleich nach rechts beginnt immer an der aktuellen Position und nicht etwa eine Position rechts davon.

`/$(?!\\z)/`

prüft zweimal an derselben Position, einmal ob \$ passt und einmal ob \\z nicht passt. Da hier keine Modifier verwendet werden, passt dieses Muster nur vor einem \\n am Ende des Suchstrings. Vergleiche nach rechts bieten die Möglichkeit, den Durchschnitt zweier Zeichenklassen abzufragen. Das Muster

`/(?=\p{InDevanagari})\pL/`

passt nur, wenn rechts von der Abfrageposition im Suchtext ein Zeichen steht, das sowohl im Unicode-Block Devanagari liegt als auch ein Buchstabe ist.

Von solchen mehrfachen Prüfungen rechts folgender Teile im Suchtext abgesehen, findet man positive Vergleiche nach rechts, wie die Anker \$, \\Z und \\z vorwiegend am Ende von Musterketten als Bedingungen für den rechten Rand eines Treffers. \$, \\Z und \\z sind ja spezielle positive Vergleiche nach rechts. Ein regulärer Ausdruck wie

`(S(?:[taeiou]{2})\\w{2,})`

ist syntaktisch zwar korrekt, der Vergleich nach rechts in diesem Muster ist aber kaum hilfreich. Wenn man damit

Es gibt große Stunden im Leben.

durchsucht, wird zunächst S gefunden. Die aktuelle Position im Suchtext steht dann zwischen diesem S und dem nachfolgenden t. Ab dieser Position wird dann geprüft, ob zwei Zeichen aus [taeiou] folgen, was der Fall ist. Die aktuelle Position im Suchtext hat sich dadurch aber nicht verändert. Die Mustersuche fährt dann ab dieser Position mit der Suche nach Treffern zu \\w fort. Die zuvor schon geprüften, aber nicht verbrauchten Zeichen werden jetzt noch einmal geprüft, diesmal aber auch verbraucht. Ein regulärer Ausdruck

`/S[taeiou]{2}\\w{0,}/`

ist für diese Mustersuche sicher besser geeignet. Die Überlappung der überprüften Teile von negativem Vergleich nach rechts und verbrauchenden Musterelementen kommt schon öfters vor. Wenn man in einem Suchtext alle Wörter, die mit `S` beginnen, aber von `Stunden` verschieden sind, herausfinden möchte, kann man

```
/(?!Stunden)S[[:alpha:]]{0,}/
```

verwenden. Hier wird auf jeder Position im Suchtext geprüft, ob rechts davon das Wort `Stunden` folgt; ist das der Fall, wird die Mustersuche abgebrochen und die Startposition um eins nach rechts verschoben. Ist das nicht der Fall, wird ab der Startposition der Treffer zu `S[[:alpha:]]{0,}` ermittelt.

Da Vergleiche nach rechts keine Zeichen verbrauchen, ist es sinnlos, sie zu quantifizieren. Perl beanstandet eine solche Quantifizierung nicht, sie hat aber auch keine Auswirkungen. Insbesondere ist eine »äußere« Quantifizierung eines Vergleiches nach rechts nicht äquivalent zu einer Quantifizierung innerhalb des Vergleiches nach rechts, die ja durchaus sinnvoll sein kann

```
$St = 'xaxa011xa011011b';  
$St =~ s/a(?:011){2}/#/; # innere Quantifizierung
```

liefert als Ergebnis `xaxa011x#b`, gesucht wird hier nach einem `a`, dem die Zeichenfolge `011011` folgt. Dagegen liefert:

```
$St = 'xaxa011xa011011b';  
$St =~ s/a(?:011){2}/#/; # sinnlose äußere Quantifizierung
```

das Ergebnis `xax#xa011011b`. Gesucht wird hier nach einem `a`, dem die Zeichenfolge `011` folgt, wobei Letzteres zweimal geprüft werden soll, das ist jedenfalls die einzig mögliche Interpretation des Quantifiers. PHP beanstandet solche Quantifier als Fehler, was sinnvoll ist, denn vermutlich wurde etwas anderes beabsichtigt als eine doppelte Prüfung.

Die Konstrukte mit Fragezeichen nach öffnenden runden Klammern treten in unterschiedlicher Bedeutung auf und sind nicht einfach auseinander zu halten. Zwei Fehler, die am Anfang häufiger auftreten, sind:

```
/(?!=Vergleichsmuster)/
```

Das `=`-Zeichen gehört hier zum Vergleichsmuster, ein negierter Vergleich nach rechts hat die Form:

```
/(?!Vergleichsmuster)/
```

In einem Vergleich nach rechts kommt auch kein >-Zeichen vor:

```
/(?>=Vergleichsmuster)/
```

ist kein Vergleich nach rechts.

### Übungsaufgabe 4.8

Wie viele Substitutionen werden mit der folgenden Substitutionsanweisung ausgeführt, wie lautet die Ausgabe?

```
print preg_replace('/(?=[a-z])/',' #',"abc\r\ndef\r\fe\r\n");
```

### Übungsaufgabe 4.9

Welche Treffer werden durch:

```
$St = "abc\r\nde\r\nfg ?=xxx";
preg_match_all('/( ?=[a-z])/',$St,$Tr);
erzielt?
```

## 4.8 Vergleich nach links

Vergleiche nach links (look behind assertion, Versicherung nach links) unterscheiden sich von Vergleichen nach rechts, abgesehen von einer anderen Formulierung und der Vergleichsrichtung, dadurch, dass die Zahl der Zeichen, die überprüft werden müssen, fest vorgegeben sein muss, also schon bekannt sein muss, wenn der reguläre Ausdruck kompiliert wird. Ein Vergleich nach links kann positiv oder negativ formuliert werden:

```
(?<=Vergleichsmuster)
(?<!Vergleichsmuster)
```

Da in einem Vergleich nach links die Trefferlänge fest vorgegeben sein muss, können dort keine quantifizierten Musterelemente auftreten, die unterschiedliche Trefferlängen zulassen. Ebenso sind interne Rückwärtsreferenzen nicht möglich, die im nächsten Kapitel behandelt werden.

```
(?<=\n{0,})(.{1,})
```

liefert in Perl 5.8.4 bzw. PHP 5.0.4 z.B. die Fehlermeldungen:

```
Variable length lookbehind not implemented in regex
lookbehind assertion is not fixed length
```

Unterschiedlich gehandhabt wird diese Regel bei Alternativen im Muster. Perl verlangt, dass alle Alternativen gleiche feste Länge aufweisen müssen, PHP lässt dagegen Alternativen mit unterschiedlichen Längen zu, sofern diese nur schon aus dem Quelltext hervorgehen. Das Muster:

```
(?<=\d{4}|\d{2})
```

ist in PHP gültig, in Perl dagegen nicht. Allerdings ist der Gewinn dadurch nicht sonderlich groß, denn in Perl können Sie stattdessen schreiben:

```
(?: (?<=\d{4}) | (?<=\d{2}))
```

Da die unterschiedlichen Kettenlängen in unterschiedlichen Vergleichen nach links auftreten, sind diese korrekt. In der Wirkung sind beide Konstrukte äquivalent. Das PHP-Konstrukt ist allerdings übersichtlicher.

Analog zum Beispiel mit einem Vergleich nach rechts, kann man auch einen Vergleich nach links dazu benutzen, einen mit einem recht allgemeinen Musterelement erzielten Treffer noch einmal zu überprüfen, weil man bestimmte Treffer ausschließen möchte.

Um im Suchtext nach einem beliebigen vollständigen Wort zu suchen, können wir den regulären Ausdruck `\b\w{1,}\b` verwenden, wir setzen dabei voraus, dass es keine Schwierigkeiten mit Buchstaben gibt, die nicht als solche erkannt werden (deutsche Umlaute, locale C). Wenn wir nun nach Worten suchen, die nicht mit `der` enden, können wir verwenden:

```
\b\w{1,}(?!der)\b
```

Ein Treffer von `\w{1,}` beginnt hier an einer Wortgrenze und endet, sobald `\w{1,}` kein Wortzeichen mehr verbrauchen kann. An dieser Position, also hinter dem letzten Wortzeichen, wird dann geprüft, ob das von `\w{1,}` gefundene Wort nicht mit `der` endet. An dieser Position wird auch geprüft, ob eine Wortgrenze vorliegt, was an sich überflüssig ist, da anderenfalls `\w{1,}` ein weiteres Zeichen verbraucht hätte.

Bevor die Vergleiche nach links als Musterelement in regulären Ausdrücken verfügbar waren, wurde das Problem häufig diskutiert, eine durch `"`-Zeichen begrenzte Zeichenkette zu finden, die selbst durch `\`-Zeichen maskierte `"`-Zeichen enthalten kann, also z.B.:

```
"Als Grenzzzeichen sind \"- und '-Zeichen möglich."
```

Mit einem Vergleich nach links ist das Problem einfach zu lösen, sofern man voraussetzen kann, dass im Suchtext keine ungepaarten, nicht maskierten `"`-

und keine gepaarten \-Zeichen auftreten. Man kann in PHP u. a. den regulären Ausdruck:

```

/
"                # einleitendes "
(                # Klammer auf
  ["^"] | (?<=\\\)  # Alternative
){0,}           # Klammer zu, Quantifier
"                # abschließendes "
/x

```

benutzen. Die Alternative passt hier auf jedes von " verschiedene Zeichen und auf ein ", dem ein Backslash vorausgeht. Die Anzahl der anzugebenden Backslashes hängt davon ab, wie die Mustersuche in die Sprache eingebettet ist. Obiges Muster ist z. B. in einer Mustersuche von PHP geeignet.

### Beispiel 4.23

In einem Dokument sollen alle Zeichenfolgen der Form ä, Ä, ... ü durch ae, Ae, ... ue ersetzt werden. Die Lösungen hängen davon ab, welche Hilfsmittel die jeweilige Sprache zur Verfügung stellt.

In Perl gibt es die Operation `$Suchtext =~ tr/Urliste/Ersatzliste/`, wobei in beiden Listen eine Zeichenfolge ohne Trennzeichen erwartet wird. Diese Operation hat mit regulären Ausdrücken nichts zu tun, obwohl auch hier ein Bindungsoperator verwendet wird.

Wenn beide Listen gleich lang sind, gibt es eine Zuordnung des n. Zeichens in der Urliste zum n. Zeichen in der Ersatzliste. Bei:

```
$Suchtext =~ tr/äöü/aou/;
```

also die Zuordnungen:

ä → a

ö → o

ü → u

Im Suchtext wird dann jedes Vorkommen eines Zeichens aus der Urliste durch das zugeordnete Zeichen aus der Ersatzliste ersetzt. Hier sollen aber nicht ein Zeichen durch ein anderes, sondern ein Zeichen durch zwei andere ersetzt werden. Natürlich kann man auf den Suchtext der Reihe nach Substitutionen wie:

```
$Suchtext =~ s/ä/ae/g;
```

```
$Suchtext =~ s/Ä/Ae/g;
```

...

```
$Suchtext =~ s/ü/ue/g;
```

anwenden und erhält das gewünschte Ergebnis. Gibt es eine kürzere Lösung? Man kann zunächst hinter jedem Umlaut ein `e` einsetzen und ersetzt dann die Umlaute durch die entsprechenden Vokale, dazu genügt:

```
$Suchtext =~ s/(?<=[äöüÄÖÜ])/e/g;
```

```
$Suchtext =~ tr/äöüÄÖÜ/aouAOU/;
```

PHP lässt bei `preg_replace` zu, dass als Muster und Ersatztext jeweils Listen angegeben werden, im Falle der Muster müssen die Listenelemente Zeichenketten mit Mustergrenzen und gültigen Mustern sein, im Falle der Ersatztexte eben Zeichenketten. Hier liegt es nahe, eine Lösung mit solchen Listen zu suchen.

```
$Muster = array('/ä/', '/ö/', '/ü/', '/Ä/', '/Ö/', '/Ü/');
```

```
$Ersatz = array('ae', 'oe', 'ue', 'Ae', 'Oe', 'Ue');
```

```
print preg_replace($Muster, $Ersatz, $St). "\r\n";
```

Da hier beide Listen gleich lang sind, wird jeweils ein Treffer des `n`. Elementes der Musterliste durch das `n`. Element der Ersatzliste ersetzt. Zu beachten ist, dass bei einer wiederholten Mustersuche Zeichen, die in einer vorangegangenen Mustersuche verbraucht wurden, durch einen Vergleich nach links in einer nachfolgenden Mustersuche trotzdem erkannt werden, sie können nur nicht ein zweites Mal verbraucht werden. Im Suchtext

```
1ab2cd3ef4
```

liefert eine globale Substitution mit `/(?<=\d)[a-z]{2}\d/` und Ersatztext `#` daher `1###`. Die erste Mustersuche findet:

```
1ab2cd3ef4
```

```
▲▲▲
```

und verbraucht die markierten Zeichen, die zweite Mustersuche startet dann auf der Position hinter den markierten Zeichen, und der Vergleich nach links erkennt die schon verbrauchte Ziffer 2. Analog arbeiten die folgenden Mustersuchen.

Im nächsten Kapitel werden einfangende Klammern in Vergleichen nach rechts und links noch einmal genauer behandelt.

## Lösungen der Übungsaufgaben

### Übungsaufgabe 4.1

Durch `[[:alpha:]]{1,}` wird eine frei definierbare Zeichenklasse angegeben, die äquivalent zu `[{}1,[:alpha:]]` ist und nur ein Zeichen des Suchtextes verbraucht. `[[:alpha:]]{1,}` ist dagegen eine quantifizierte Zeichenklasse, die einerseits nur auf Buchstaben, andererseits aber auf beliebig viele davon passt.

### Übungsaufgabe 4.2

Das Muster `(\d{0,}\s{0,}){1,}` und seine Treffer in:

```
12 3a b57 c
```

sind nicht einfach zu überschauen. Zunächst findet `\d{0,}` die beiden Ziffern 12, danach findet `\s{0,}` die Leerstelle zwischen der 2 und 3. Wegen des äußeren Quantifiers findet `\d{0,}` auch noch die 3. Damit ist die erste Mustersuche beendet, und die Startposition im Suchtext rückt für die nächste Mustersuche auf die Position hinter der 3 vor.

```
12 3a b57 c
```

```
▲▲▲▲
```

An dieser Position wurde noch kein Nulltreffer gefunden, und Nulltreffer sind mit dem angegebenen Muster möglich, also wird hier bei der nächsten Mustersuche ein solcher gefunden. Die Startposition für die nächste Mustersuche rückt danach – nach Konvention – hinter das nachfolgende `a` vor.

Ab dieser Position wird eine Leerstelle als Treffer gefunden, die Startposition rückt hinter diese Leerstelle vor. Auch hier wird ein Nulltreffer gefunden, und die Startposition für die nächste Mustersuche wird nach Konvention hinter das `b` verschoben.

Diese Mustersuche erzielt den Treffer 57, danach rückt die Startposition auf die Position vor dem `c`, wo ein weiterer Nulltreffer gefunden wird. Die Startposition rückt jetzt auf das Ende des Suchtextes, dort wird noch einmal ein Nulltreffer gefunden. Gefunden werden also der Reihe nach:

- ▶ die Zeichenkette 12 3 ab Position 0
- ▶ ein Nulltreffer auf Position 4
- ▶ eine Leerstelle ab Position 5
- ▶ ein Nulltreffer auf Position 6,
- ▶ die Zeichenfolge 57 ab Position 7

- ▶ ein Nulltreffer auf Position 10
- ▶ ein Nulltreffer auf Position 11

Dieselben Treffer ließen sich auch mit dem Muster `[\d\s]{0,}` erzielen, ein Muster, das sicherlich einfacher zu überschauen ist. Ohne den äußeren Quantifier würde der erste Treffer nur 12 umfassen, und die nachfolgende 3 würde mit der folgenden Mustersuche gefunden werden, man hätte also einen Treffer mehr.

### Übungsaufgabe 4.3

Da die Variable mehrere Zeilen enthält, braucht man einen Anker, der auf einen Zeilenanfang innerhalb des gesamten Suchtextes passt. Geeignet ist `^` mit dem `m`-Modifier. Man braucht ferner ein Musterelement, das auf den gesamten Zeileninhalt passt, dazu ist `.{1,}` geeignet, wenn kein `s`-Modifier benutzt wird. Ein Muster `/^{1,}/m` passt aber auf jede Zeile im Suchtext, man braucht noch einen Anker, der den Treffer am Ende des Suchtextes verankert. Da bei der letzten Zeile zugelassen wird, dass sie mit oder ohne Zeilenvorschub endet, ist der `\Z`-Anker geeignet. Der `$`-Anker ist wegen des schon gesetzten `m`-Modifiers für diesen Zweck ungeeignet. Eine Lösung ist demnach:

```
/.{1,}\Z/m
```

Die Frage ist nun, ob diese Lösung auch unter PHP und Windows funktioniert? Wenn die letzte Zeile nicht mit einem Zeilenvorschub abgeschlossen ist, entsteht kein Problem, ist sie jedoch mit einem Zeilenvorschub abgeschlossen, hat man folgendes Ende des Suchtextes:

```
xxxxxxxxxxx\r\n
```

Das Musterelement `.{1,}` schluckt alle Zeichen der letzten Zeile einschließlich des `\r`. `\Z` passt nach Perl-Manier sowohl auf das Ende des Suchtextes als auch auf die Position davor, sofern dieser mit `\n` endet. Man findet also auch unter PHP und Windows die letzte Zeile, aber einschließlich des `\r`.

### Übungsaufgabe 4.4

Eine Lösung, die auf alle möglichen Zeilenenden passt, ohne diese zu verbrauchen, ist:

```
(?=\r\n|\n|\z)
```

Die Verwendung der verschiedenen Zeilenende-Anker ist nicht möglich, da auch der Fall Windows und PHP berücksichtigt werden soll.

Nun sollen noch die Zeilen ausgeschlossen werden, die einen Punkt oder Doppelpunkt vor diesem Muster haben, das führt zu:

```
(?<![.:])(?=\r\n|\n|\z)
```

Ist das richtig? Wenn vor dem Zeilenende kein Punkt oder Doppelpunkt steht, dann schon. Aber was ist, wenn dort einer steht? Dann passt dieses Muster nicht auf die Position nach diesem Punkt oder Doppelpunkt, und die Muster-suche sucht auf der Folgeposition. Im Falle eines Zeilenendes der Form `\r\n` also zwischen `\r` und `\n`. Hier passt das Muster, und folglich wird hier die Substitution ausgeführt, was nicht der Aufgabenstellung entspricht. Wenn man sicher sein kann, dass ein `\r` im Suchtext nur zusammen mit einem nachfolgenden `\n` auftritt, kann man das Problem mit dem Muster

```
(?<![.: \r\n])(?=\r\n|\n|\z)
```

lösen. Wenn man nicht sicher ist, wird man vielleicht

```
(?<![.:])(?=\r\n)|(?<![.: \r])(?=\n)|(?<![.: \n])(?=\z)
```

wählen. Die erste Alternative passt auf jede Zeile mit einem Windows-Zeilenende, die nicht mit einem Punkt oder Doppelpunkt endet. Die zweite Alternative passt auf jede Zeile mit einem Unix-Zeilenende, die nicht mit einem Punkt, Doppelpunkt oder `\r` endet. Letzteres schließt aus, dass ein Zeilenende wie etwa `:\r\n`, auf das die erste Alternative nicht passt, zur zweiten Alternative passt. Die dritte Alternative passt immer am Ende des Suchtextes, sofern davor kein Punkt, Doppelpunkt oder `\n` steht. Letzteres schließt aus, dass etwa `:\n`, das nicht zur zweiten Alternative passt, zur dritten passt.

### Übungsaufgabe 4.5

Die Aufgabe ist wesentlich einfacher zu lösen als 4.4, da jede Zeile durch `\n` beendet wird. Wir verwenden zunächst den `$`-Anker mit `m`-Modifier, der auf jedes Zeilenende, also vor jedem `\n` passt. Wir müssen nun als zusätzliche Bedingung fordern, dass diesem Zeilenende kein Punkt vorausgeht, was wir mit:

```
/(?<![.\n])$/m
```

erreichen. Es ist notwendig, das `\n` in die Zeichenklasse aufzunehmen, da sonst dieses Muster am Ende des Suchtextes passen würde. Damit ist auch der Fall der Leerzeilen erledigt, denn diese sind in der Regel durch zwei aufeinander folgende `\n` gekennzeichnet. Dies gilt jedoch nicht für eine eventuelle Leerzeile

am Anfang. Wir können in die Zeichenklasse kein dementsprechendes Zeichen aufnehmen, können aber noch einen weiteren Vergleich nach links in das Muster einbauen:

### Übungsaufgabe 4.6

In den Suchtexten ist die Escape-Sequenz `\b` nur in der `"`-begrenzten Zeichenkette in Perl definiert, in allen anderen Fällen hat `\b` literale Bedeutung. In den regulären Ausdrücken in Perl hat `\b` immer die Bedeutung einer Wortgrenze, in PHP gilt dies ebenfalls, da `\b` weder in den `'`- noch in den `"`-begrenzten Zeichenketten verändert wird. Wir haben damit nur zwei wirklich verschiedene Fälle:

```
"\b" =~ m/\b/ # Perl
```

und alle restlichen, die wir durch:

```
'\b' =~ m/\b/ # Perl
```

repräsentieren können. Im oberen Fall besteht der Suchtext aus einem Backspace-Zeichen, das ist kein Wortzeichen, und folglich gibt es weder davor noch danach eine Wortgrenze. Das Muster passt nicht, der Wert des Ausdruckes ist `falsch`. In allen anderen Fällen gibt es zwischen `\` und `b`, sowie nach `b` eine Wortgrenze, das Muster passt immer und der Wert des Ausdruckes ist `wahr`.

### Übungsaufgabe 4.7

Am einfachsten ist es wohl, wenn man `\b` ergänzt durch einen zusätzlichen Vergleich, `(?<=\w)\b` passt auf einen Wortanfang, `\b(?:=\w)` auf ein Wortende. Man kann auch `(?<=\w)(?=\w)` verwenden, um einen Wortanfang festzustellen, und `(?<=\w)(?=\w)` für ein Wortende.

### Übungsaufgabe 4.8

Das Muster:

```
(?=[a-z])
```

ist nicht verbrauchend und erzielt daher nur Nulltreffer, es passt vor jedem Kleinbuchstaben im Suchtext. Die Klammerung um die Zeichenklasse wird nicht benötigt. Im Suchtext ergeben sich Treffer an den folgenden Positionen:

```
a b c \r \n d e f \r \f e \r \n
↑ ↑ ↑      ↑ ↑ ↑      ↑
```

Als Ergebnis der Substitution erhält man demnach:

```
#a#b#c\r\n#d#e#f\r\n#e\r\n
```

Wie die Ausgabe aussieht, hängt davon ab, wie das Ausgabemedium die Steuerzeichen `\r`, `\n` und `\f` behandelt.

### Übungsaufgabe 4.9

Welche Treffer erzielt:

```
$St = "abc\r\nde\r\nfg ?=xxx";
preg_match_all('/( ?=[a-z])/',$St,$Tr);
```

Das Muster besteht wegen der Leerstelle nach der öffnenden Klammer aus einer einfangenden Klammer und nicht aus einem Vergleich nach rechts. Das Fragezeichen ist ein Quantifier mit der Leerstelle als Operanden. Zwingend vorgeschrieben wird durch das Muster ein `=`-Zeichen und ein nachfolgender Buchstabe in jedem Treffer. Wenn möglich, soll eine davor stehende Leerstelle in den Treffer einbezogen werden. Beim angegebenen Suchtext ist dies nicht möglich, denn vor dem `=`-Zeichen steht dort ein Fragezeichen, gefunden wird also `=x`.

## 6 PHP-Spezialitäten

### 6.1 Reguläre Ausdrücke nach POSIX

Die regulären Ausdrücke nach POSIX werden als Zeichenketten angegeben. Anders als bei den Perl-kompatiblen regulären Ausdrücken gibt es hier keine weiteren Grenzmarkierungen für den regulären Ausdruck. Variableninterpolation in "-begrenzten Zeichenketten ist möglich. Funktionen stehen zur Muster-suche, Textersetzung und Textzerlegung zur Verfügung. Das Konzept der Modif-ier wird nicht unterstützt, anstelle des *i*-Modifiers gibt es jeweils spezielle Funktionen. Es gibt keine Konstrukte zur globalen Mustersuche oder Substitu-tion. Ein Zeichen mit Code-Punkt 0x00 wird von allen Funktionen zur Muster-suche als Ende des Suchtextes interpretiert.

#### 6.1.1 Mustersuche

Zur Mustersuche können `ereg` und `eregi` verwendet werden, die sich dadurch unterscheiden, dass `eregi` nicht zwischen Groß- und Kleinbuchstaben unter-scheidet.

`ereg` erwartet das Muster im ersten Argument. Der Suchtext wird im zweiten Argument erwartet. In einem optionalen dritten Argument kann eine Treffer-liste angegeben werden. `ereg` speichert dann im 0. Element den Treffer des gesamten regulären Ausdrucks, im 1. Element den letzten Treffer der ersten einfangenden Klammer usw. Zurückgegeben wird *wahr*, wenn ein Treffer gefunden wurde, anderenfalls *falsch*.

```
$St = 'abc, die Katze liegt im Schnee';  
ereg('( [A-Za-z]+ ){3}', $St, $Tr);  
$n=0; foreach ($Tr as $tr){print "$n  $tr\r\n"; $n++;}
```

liefert daher

```
0  die Katze liegt  
1  liegt
```

`eregi` unterscheidet sich von `ereg` nur dadurch, dass zwischen Groß- und Kleinschreibung nicht unterschieden wird. Im obigen Beispiel kann man mit `eregi` die Zeichenklasse kürzer schreiben:

```
eregi('( [a-z]+ ){3}', $St, $Tr);
```

und erhält mit demselben Suchtext trotzdem dieselben Ergebnisse.

### 6.1.2 Textersetzung

Textersetzungen können mit den Funktionen `ereg_replace` und `eregi_replace` durchgeführt werden. Im ersten Aufrufargument erwartet `ereg_replace` das Muster, im zweiten den Ersatztext und im dritten den Suchtext. Durchgeführt wird eine globale Textersetzung, d.h. eine Folge von Textersetzungen, mit der alle Treffer im angelieferten Suchtext ersetzt werden. Zurückgegeben wird die Zeichenkette, die aus den Ersetzungen resultiert. Der angelieferte Suchtext wird nicht verändert.

```
$St = 'abc, die Katze liegt im Schnee';  
print ereg_replace(' [A-Za-z]+', '* ', $St)."\r\n";
```

liefert dementsprechend:

```
abc, * * * * *
```

`eregi_replace` unterscheidet sich von `ereg_replace` wieder nur dadurch, dass bei der Mustersuche nicht zwischen Groß- und Kleinbuchstaben unterschieden wird.

### 6.1.3 Textzerlegung

Zur Textzerlegung stehen die Funktionen `split` und `spliti` zur Verfügung. `split` erwartet im ersten Aufrufargument das Muster, im zweiten wird der zu zerlegende Suchtext und in einem optionalen dritten Aufrufargument die Anzahl der zurückzuliefernden Bruchstücke erwartet. Zurückgeliefert werden die Bruchstücke, also die Teilketten des Suchtextes, die zwischen den Treffern des Musters liegen.

```
$St = 'abc, die Katze liegt im Schnee';  
$Bs = split(' ', $St);  
$n=0; foreach ($Bs as $tr){print "$n  $tr\r\n"; $n++;}
```

liefert:

```
0  abc,  
1  die  
2  Katze  
3  liegt  
4  im  
5.  Schnee
```

Wenn bei der Zerlegung Bruchstücke der Länge 0 entstehen, werden diese auch an die Ergebnisliste ausgeliefert. Mit

```
$Bs = split('[aei'],$St);
```

erhält man aus dem obigen Suchtext die Bruchstücke:

```
0  ''          4  'tz'          8  'm chn'
1  'bc, d'    5  'l'           9  ''
2  ''         6  ''           10 ''
3  'K'        7  'gt'
```

Gibt man eine Grenze *n* für die Zahl der Bruchstücke vor, so enthält das letzte Element das Reststück des Suchtextes nach *n-1* Mustersuchen.

```
$Bs = split('[aei'],$St,3);
```

liefert also:

```
0  ''          1  'bc, d'       2  'e Katze liegt im Schnee'
```

`spliti` unterscheidet sich von `split` wieder nur dadurch, dass bei der Muster-suche kein Unterschied zwischen Groß- und Kleinbuchstaben gemacht wird.

### Beispiel 6.1

```
$Klasse = '[a-z]';
$St = 'abc, die Katze liegt im Schnee';
$Bs = spliti("$Klasse",$St);
$n=0; foreach ($Bs as $tr){print "$n  $tr\r\n";$n++;}
```

Die Zeichenkette mit dem Muster besteht nach der Variablen-Interpolation aus einer literalen Leerstelle und der Zeichenklasse `[a-z]`, Bruchstellen sind daher `d`, `K`, `l`, `i` und `S`:

#### 6.1.4 Zeichenmuster

Die meisten Zeichen können in POSIX-konformen regulären Ausdrücken literal genutzt werden. Maskieren muss man lediglich Zeichen, die entweder in der jeweils verwendeten Zeichenkette, also `"`- oder `'`-Zeichenkette, oder aber im regulären Ausdruck Metazeichen sind. Die Metazeichen der POSIX-konformen regulären Ausdrücke sind unten aufgelistet.

Beachten Sie, dass eine Reihe von Escape-Sequenzen im interpolativen Kontext ausgewertet werden.

## Beispiel 6.2

```
$St = 'abc, die (Katze) {liegt} +im $Schnee$.';  
ereg('\(.\+\) \{.\+\} \+.\+ \$.\+\$\'', $St, $Tr);  
foreach ($TR as $tr) { print "$tr\r\n"; }
```

Das Muster enthält die folgenden Musterelemente:

- ▶ \`(` passt auf eine `(` im Suchtext
- ▶ `.\+` passt auf mindestens ein beliebiges Zeichen
- ▶ \`)` passt auf eine `)`
- ▶ literale Leerstelle
- ▶ \`{` passt auf eine `{`
- ▶ `.\+` passt auf mindestens ein beliebiges Zeichen
- ▶ \`}` passt auf eine `}`
- ▶ literale Leerstelle
- ▶ \`+` passt auf `+`
- ▶ `.\+` passt auf mindestens ein beliebiges Zeichen
- ▶ literale Leerstelle
- ▶ \`$` passt auf `$`
- ▶ `.\+` passt auf mindestens ein beliebiges Zeichen
- ▶ \`$` passt auf einen `$`
- ▶ \`.` passt auf einen `.`
- ▶ `$` verankert den Treffer am Ende des Suchtextes

Metazeichen in POSIX-konformen regulären Ausdrücken sind lediglich:

- ▶ `.` wird als Punkt-Zeichenklasse benutzt
- ▶ `[` öffnende Klammer einer frei definierbaren Zeichenklasse
- ▶ \`\` dient zur Maskierung
- ▶ `*` dient in der Regel als Quantifier, wird in bestimmten Situationen aber literal interpretiert
- ▶ `^` dient in der Regel als Anker
- ▶ `$` dient in der Regel als Anker
- ▶ `(` (und `)`) dienen als einfangende und gruppierende Zeichen
- ▶ `{` (und `}`) dienen zur Klammerung von Quantifiern

### 6.1.5 Zeichenklassen

Ein Punkt in einem POSIX-konformen regulären Ausdruck passt auf jedes Zeichen, auch auf `\n`. Der Punkt wird hier also so behandelt wie in den Perl-kompatiblen regulären Ausdrücken, wenn dort ein `s`-Modifier benutzt wird. In:

```
$St = "abc,  
die Katze liegt im Schnee";  
ereg("(.)+", $St, $Tr);
```

passt das Muster `(.)+` auf den gesamten Suchtext.

Frei definierbare Zeichenklassen können Sie wie in Perl-kompatiblen regulären Ausdrücken verwenden, das gilt auch für die negierende Form. Innerhalb der eckigen Klammern können allerdings nur solche Elemente aufgelistet werden, die in POSIX-konformen regulären Ausdrücken definiert sind. Sie können dort keine Escape-Sequenzen oder Unicode-Zeichenklassen auflisten. Zeichenbereiche und POSIX-Zeichenklassen sind allerdings möglich.

Unterstützt werden, wie zu erwarten, die POSIX-Zeichenklassen. POSIX definiert außer Zeichenklassen auch einige andere Klammerausdrücke. In einer locale können z.B. bestimmte Äquivalenzklassen von Zeichen definiert sein, so können in einer Äquivalenzklasse mit Namen `a` etwa die Zeichen `a`, `â`, `á` und `à` stehen. In Posix-konformen regulären Ausdrücken kann man in Zeichenklassen dann den Namen der Äquivalenzklasse in der Form `[=a=]` angeben. Die Zeichenklasse `[=a=]x` würde in diesem Falle also z.B. auf `ax`, aber auch auf `âx` passen. Unterstützt werden auch Klammerausdrücke der Form `[.name.]`, auch hier muss `name` durch die jeweils benutzte locale definiert sein. Er bezeichnet hier eine Kollationssequenz. Solche Kollationssequenzen machen an sich Vorgaben zur Sortierung, so wird im Spanischen `ll` beim Sortieren wie ein einziges Zeichen behandelt, das zwischen `l` und `m` einsortiert wird. Man kann die Namen solcher Kollationssequenzen in Zeichenklassen POSIX-konformer regulärer Ausdrücke verwenden.

Bei den Perl-kompatiblen regulären Ausdrücken sind diese Konstrukte noch nicht implementiert, man erhält bei ihrer Verwendung zumeist eine Warnung, dass derartige Zeichenfolgen für spätere Erweiterungen reserviert seien.

### 6.1.6 Anker

Reguläre Ausdrücke nach POSIX unterstützen die folgenden Anker:

- ▶ `^` passt auf den Anfang des Suchtextes
- ▶ `$` passt auf das Ende des Suchtextes

- ▶ `[[[:<:]]` passt auf ein Wortanfang, also die Position zwischen einem Nicht-Wortzeichen und einem Wortzeichen, wie in Perl wird der gesamte Suchtext so behandelt, als sei er von Nicht-Wortzeichen umgeben.
- ▶ `[[[:>:]]` passt auf ein Wortende

### Beispiel 6.3

```
$St = "abc, die Katze liegt im Schnee\r\n";
ereg('([[[:<:]] [a-z]+[[[:>:]] ]^a-zA-Z]{1,2})+', $St, $Tr);
$n=0; foreach ($Tr as $tr) { print "$n $tr\r\n"; $n++;}
```

liefert:

```
0 abc, die Katze liegt im Schnee
1 Schnee
```

Im 0 Element findet man den gesamten Treffer der Mustersuche, im ersten Element den letzten Treffer der einfangenden Klammer. Diese passt der Reihe nach auf die einzelnen Wörter im Suchtext und auf ein oder zwei nachfolgende Zeichen, die keine Buchstaben sind.

### Beispiel 6.4

```
$St = "abc_ die Katze liegt im Schnee\r\n";
if (ereg('([[[:<:]] [a-z_]+[[[:>:]] ]+)', $St, $Tr) )
$n=0; foreach ($Tr as $tr) { print "$n $tr\r\n"; $n++;}
```

liefert:

```
0 abc_
1 abc_
```

Der Unterstrich gehört auch bei den regulären Ausdrücken nach POSIX zu den Wortzeichen.

### Beispiel 6.5

```
$St = "abc, die Katze liegt im Schnee";
if (ereg('([[[:<:]]S[a-z_]+$)', $St, $Tr) )
{ $n = 0; foreach ($Tr as $tr) { print "$n $tr\r\n"; $n++; } }
```

liefert:

```
0 Schnee
1 Schnee
```

Der Anker `$` passt nur am Ende des Suchtextes, nicht vor `\n` oder `\r\n`. Ersetzt man hier der Anker `$` durch einen Wortendeanker `[:>:]`, so erhält man dasselbe Ergebnis. Verwendet man jedoch `[:>: ]$`, so findet `ereg` keinen Treffer.

### 6.1.7 Operatoren

Alternativen werden wie üblich mit dem Operator `|` gebildet. Definiert ist auch die Verkettung und Quantifizierung. Definiert sind die folgenden Quantifier:

`?`, `*`, `+`, `{n}`, `{n,m}` und `{n,}`

Diese haben jeweils dieselbe Bedeutung wie in Perl-kompatiblen regulären Ausdrücken. Es gibt keine minimalen Quantifier. Es gibt nur eine Art der Klammerung, die zur Gruppierung und als einfangende Klammer dienen muss. Alle Konstrukte, die in Perl-kompatiblen regulären Ausdrücken mit `(?..)` formuliert werden, fehlen. Die Regelung der Prioritäten ist dieselbe wie bei den Perl-kompatiblen regulären Ausdrücken.

### 6.1.8 Rückwärtsreferenzen

Externe Rückwärtsreferenzen auf die Treffer einfangender Klammern sind bei der Substitution möglich und werden in der Form `\n` angegeben, also wie die internen Rückwärtsreferenzen bei Perl-kompatiblen regulären Ausdrücken.

#### Beispiel 6.6

Im folgenden Suchtext werden die `'`- und `"`-Zeichen miteinander vertauscht.

```
$St = 'in \'- und "-begrenzten Zeichenketten.';
echo ereg_replace('([\'])(.+)([\'"])', '\3\2\1', $St);
```

### 6.1.9 Nullmuster und Nulltreffer

Nullmuster im regulären Ausdruck führen zu Fehlermeldungen, auch bei `split`. Quantifizierte Musterelemente, die einen Nulltreffer zulassen, führen bei `split` zu Fehlermeldungen. So liefert

```
$Bs = split('[:space:]*', $St);
```

die Meldung

```
Invalid Regular Expression to split().
```

Die `ereg-` und `ereg_replace-`Funktionen akzeptieren dagegen solche Muster und finden gegebenenfalls auch die entsprechenden Nulltreffer.

## 6.2 Multi-Byte String-Funktionen

PHP kann optional mit einem Paket zur Bearbeitung von Zeichenketten installiert werden, bei denen die einzelnen Zeichen nicht in einem einzigen Byte abgespeichert sind. Beispiele solcher Zeichen haben Sie im Zusammenhang mit Unicode schon kennen gelernt. Da dieses Paket in den Standardinstallationen nicht enthalten ist und für PHP eine Revision der Unicode-Unterstützung angekündigt ist, soll hier nur erwähnt werden, dass dieses Paket ebenfalls Funktionen zur POSIX-konformen Mustersuche anbietet, die allerdings als experimentell gekennzeichnet sind.

## 6.3 Modifier zu Perl-kompatiblen regulären Ausdrücken

Ob bzw. welche Modifier unterstützt werden, hängt immer vom jeweiligen Werkzeug ab. PHP unterstützt die Modifier `i`, `m`, `s`, `x`. Das sind die Modifier, die in Perl-kompatiblen regulären Ausdrücken auch intern benutzt werden können; dazu bei der Substitution den `e`-Modifier. Von den üblichen Modifiern wird der `g`-Modifier nicht benutzt, stattdessen bietet PHP hier spezielle Funktionsaufrufe an. Weitgehend PHP-spezifisch sind die folgenden externen Modifier:

- ▶ **A** Mit diesem Modifier wird das Muster am Anfang des Suchtextes verankert, durch einen `A`-Modifier kann man also die entsprechenden internen Anker `\A` bzw. `^` ersetzen.
- ▶ **D** Dieser Modifier beeinflusst das Verhalten des `$`-Ankers, dieser passt dann nur auf das Ende des Suchtextes und nicht auf die Position vor einem davor stehenden `\n`. `$` wird dadurch äquivalent zu `\z`. Es gibt offensichtlich einen Widerspruch zwischen `m`- und `D`-Modifier, deshalb wird ein `D`-Modifier ignoriert, wenn gleichzeitig ein `m`-Modifier gesetzt ist.
- ▶ **S** Reguläre Ausdrücke werden vor einer Mustersuche analysiert und in eine interne Form transformiert (kompiliert), die für die Mustersuche besser geeignet ist als die externe Form, die im Programm angegeben wird. Mit Hilfe des `S`-Modifiers kann man PHP mitteilen, dass das betreffende Muster besonders gründlich analysiert und optimiert wird.
- ▶ **U** invertiert das Verhalten maximaler und minimaler Quantifier, `+` wird dadurch z.B. zu einem minimalen und `+`? zu einem maximalen Quantifier. Dieser Modifier kann auch intern gesetzt werden.

- `X` modifiziert das Verhalten von PHP bei undefinierten Escape-Sequenzen. Jeder maskierte Buchstabe, der keine definierte Escape-Sequenz einleitet, wird als Fehler gewertet. Das Musterelement

`\y`

passt normalerweise auf ein `y` im Suchtext, mit `X`-Modifier erhält man jedoch die Warnung:

```
Compilation failed: unrecognized character follows \ at ...
```

- `u` bewirkt, dass der Suchtext als UTF-8-Zeichenkette interpretiert wird. Bei einfachen Vergleichen wie:

```
$St = 'अनुपयुक्त अक्षर एनकोडिंग' ;
preg_match('/\sअक्षर\s/u', $St, $Tra);
preg_match('/(?:<=अक्षर|abc){5}/u', $St, $Trb);
```

spielt es keine Rolle, ob Muster und Suchtext nun als Byte-Kette oder utf-8-Zeichenkette miteinander verglichen werden, anders ist dies, wenn z.B. Zeichen gezählt werden müssen. Im obigen Vergleich nach links ist es auch egal, ob die Byte-Kette nun als solche oder utf-8-Zeichenkette interpretiert wird, da PHP unterschiedlich lange Alternativen in Vergleichen nach links zulässt. In den beiden folgenden Anweisungen ist das aber nicht mehr egal, weil festgestellt werden muss, wie viele Bytes zur Darstellung der fünf Zeichen benutzt werden müssen. Das erste Muster passt im oben angegebenen Suchtext `$St`, das zweite nicht:

```
preg_match('/\s.{5}\s/u', $St, $Tr);
preg_match('/\s.{5}\s/', $St, $Tr);
```

Leider endet die Unterstützung von utf-8 mit der Erkennung der utf-8-Zeichen, eine Unterstützung mit vorgegebenen Zeichenklassen ist nicht vorhanden, in frei definierbaren Zeichenklassen sind utf-8-Zeichen dagegen verwendbar.

## 6.4 Rekursive Muster in Perl-kompatiblen regulären Ausdrücken

Es gibt Muster, die sich in der Umgangssprache recht einfach formulieren lassen, mit regulären Ausdrücken aber sehr schwer. Eines dieser Probleme ist, eine korrekt verschachtelte Klammerung beliebiger Tiefe zu erkennen, also z.B. im folgenden Suchtext den unterstrichenen Teil:

als Horner-Schema:  $((((x+1) * x+2) * x+1.5) * x+1.25) * x+2$

Es ist kein Problem, einen regulären Ausdruck zu formulieren, der eine korrekte vierfache Verschachtelung von einer inkorrekten Verschachtelung unterscheiden kann, aber ein regulärer Ausdruck, der dies für beliebig tief verschachtelte Zeichenketten vermag, bedarf anderer Hilfsmittel. Ein solches Hilfsmittel ist das Musterelement  $(?R)$ , das eine Wiederholung des gesamten Musters an der angegebenen Stelle bewirkt.

### Beispiel 6.7

Ein Muster, das auf jede korrekte Klammerung der Tiefe 1 passt, ist:

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

Korrekte Klammerung heißt, es gibt eine öffnende runde Klammer, dann kommen beliebig viele Zeichen, unter denen weder eine öffnende noch eine schließende Klammer vorkommt, schließlich folgt eine schließende Klammer. Im Suchtext

```
(1+(2+(3*(x+3)*x+2)*x+4)*x+5)+3*(2*x+4)
```

findet man damit  $(x+3)$  und  $(2*x+4)$ . Ein Muster, das korrekte Klammern der Tiefe 2 findet, ist:

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

Im gegebenen Suchtext findet man damit:  $(3*(x+3)*x+2)$ , allerdings findet dieses Muster nicht alle korrekten Klammern der Tiefe 2. Auf den Suchtext

```
((a+b)*5*(c+d))
```

passt es offensichtlich nicht. Um solche Klammern zu erfassen, muss man beliebig viele Klammern der Tiefe 1 bzw. ungeklammerte Zeichenketten innerhalb der äußeren Klammerung zulassen, man kommt dann zu:

```
/
  \( (?: [^()]* | \( [^()]* \) )* \)
/x
```

Dieses Muster passt:

- ▶ mit dem Quantifier-Wert 0 auf `()`
- ▶ mit dem Quantifier-Wert 1 auf `(...)` oder `((...))`
- ▶ mit dem Quantifier-Wert 2 auf `(...(...))` oder `((...)...)` oder `((...)(...))`
- ▶ mit dem Quantifier Wert 3 auf `(...(...)...)` oder `(...(...)(...))` oder `((...)...(...))` oder `((...)(...)...)` oder `((...)(...)(...))`

Es bleibt aber immer bei der Verschachtelungstiefe 2. Wenn man auch Klammerungen der Tiefe 3 erfassen will, muss man in der inneren Klammerung noch einmal beliebig viele ungeklammerte oder einfach geklammerte Zeichenketten zulassen. Man kommt dann zum Muster:

```

/
  \(
    (? : [^()]*)
      | \(
          (? : [^()]* | \( [^()]* \) ) *
        \)
      ) *
  \)
/x
# Anfang Ebene 1
# ungeklammert
# Anfang Ebene 2
# Ebene 3
# Ende Ebene 2
# Ebene 1
# Ende Ebene 1

```

Man kann das Schema sicher schon erkennen, für jede weitere Klammer-tiefe muss das Musterelement `[^()]*`, das den Inhalt der innersten Klammer erfasst, ersetzt werden durch eine Alternative `(?: [^()]* | \( [^()]* \) )*`, und genau das nimmt einem das Musterelement `(?R)` ab. Dieses Element hat gewisse Ähnlichkeit mit einer internen Rückwärtsreferenz, nur verweist es nicht auf den Treffer eines anderen Musterelementes, sondern auf das gesamte bisherige Muster, das an seiner Stelle eingesetzt wird. Im Muster

```

/
  \( (?: [^()]* | \( [^()]* \) ) * \)
/x

```

das auf korrekte Klammerungen der Tiefe 2 passt, sehen Sie, dass die zweite Alternative einfach eine Wiederholung des Musters ist, das man ohne diese zweite Alternative schon hat. Wenn man diese Alternative nun ersetzt durch `(?R)`, hat man ein Muster, das auf beliebig tief verschachtelte korrekte Klammerungen passt.

# Index

! 147  
2-letter code 155  
3-letter code 155  
!-Operator 50, 316, 437  
" 12, 14, 26, 44, 60, 95, 96, 105, 106, 109, 169, 172, 173, 215, 223, 303, 384  
#(Perl) 26  
\$ 40, 109, 215  
\$& 379  
\$+ 380  
\$\_ 377  
\$\_ (Perl) 33  
\$' 380  
\$1 36  
\$n 304  
\$this (PHP) 20  
\$-Zeichen 20, 26, 27  
% (SQL) 62  
&&-Operator 50  
'-Grenzen 19  
(? 43, 43  
(n) 303  
(Perl) 29  
(Python) 55  
\* 257  
+ 257, 412  
. 14  
(. (PHP) 35  
=~ (Perl) 27, 30  
> 38  
? 257  
@-Zeichen 26, 27  
    143  
^ Anker 50, 215  
^M 59  
\_ Zeichen 76  
{ Klammerung 19, 25  
{1,2} 12  
{Code-Punkt} 97  
{min,} Quantifier 256  
{min,max} 15  
{min,max} Quantifier 255, 272  
{min,max}' 14  
{Nd} 180  
{NI} 180

| 14, 137  
|-Operator 247

## A

abschließende Grenzmarkierung 149  
accept-charset 124  
aktuelle Position 201  
alarm 95  
alphanumerisches Zeichen 94  
Alternative 14, 144, 247  
A-Modifier 368  
Anker 40, 56, 199, 203  
Anweisung (Perl) 25  
Anweisung (PHP) 19  
Anweisungsblock 19, 25  
Anweisungsblock (Python) 55  
Apache 64, 433  
Apache-Konfiguration 65  
Apache-Webserver 145  
Äquivalenzklasse 365  
arabische Ziffern 172  
array (PHP) 35  
ASCII-Codierung 76  
assertion 203  
aufbereiteter regulärer Ausdruck 19  
Aufbereitungsschritt 19  
Ausdruck 335  
Aussagenlogik 147  
Auswertungsschritt 115  
AUT 155

## B

b 95, 114, 223  
Backslash 93, 95  
backspace 95  
Backtracking 264, 275  
Bedingte Teilmuster 290  
Bedingtes Teilmuster 282  
Begrenzerzeichen 31  
    begrenzteZeichenkette 29, 41  
    begrenzteZeichenkette (PHP) 24  
beidseitig vergleichend 201  
benutzerdefinierte Variable 26  
Bereichsoperator 150  
Bezeichnung für locale 154

- Binärmodus 59, 89
- Bindungsoperator 27, 30
- binmode 85
- BOM 85, 126
- BOM (Byte order mark) 87
- boolescher Kontext 28
- Bruchstelle 45, 48
- Bruchstück 46
- Byte 75, 125
- byte order mark 85
- Byte orientiert Codierung 77
- Byte-Kette 125

## C

- C locale 154
- c Modifier 383
- C++-artiger Kommentar 20
- Carriage Return 59
- carriage return 95
- C-artiger Kommentar 20
- CGI-Methode 185
- CGI-Modul 127
- CGI-Skript 126
- charset=utf-8 124
- chop 42, 44, 345
- chr 78
- class (PHP) 20
- c-Modifier 379
- Cntl-Taste 96
- Code-Punkte 76
- codierter Zeichensatz 76
- Codierung 36
- Codierungsangabe 271
- Command-Line-Version (PHP) 19
- compile 427
- Control-Zeichen 60, 96, 110
- count (PHP) 24
- cp1252 Codierung 76
- cp850 76
- Cut 276

## D

- Datei-Handle 39
- Datei-Handle (Perl) 38
- de 155
- de Morgan 147, 194
- decode 127
- Default-Operand (Perl) 33

- definierte Escape-Sequenz 25
- DEU 155
- deu 155
- Devanagari Ziffern 172
- Dezimalziffer 172
- diophantische Gleichung 260
- D-Modifier 215, 368
- Dreamweaver 67
- Durchschnittsmenge 147
- dynamischer Geltungsbereich 321

## E

- ECMA-262 409
- Einbettung von JavaScript Code 428
- Ein-Byte Codierung 77
- einfangende Klammer 12, 23, 24, 36, 208, 256, 290, 302, 315, 413
- Eingabebereichsschema von Windows XP 81
- Einweg-Teilmuster 267, 275, 290
- e-Modifier 42, 45, 330, 335, 368
- Encode-Modul 87
- encoding 164
- encoding pragma 85
- Ersatztext 30
- ersetzen 30
- Escape-Sequenz 24, 29, 41, 139
- Escapesequenz 92
- eval 99
- except-Block 56
- exec 425
- exit 43
- externe Modifier 292
- externe Rückwärtsreferenz 36, 304, 312, 419
- externer Modifier 53, 67, 152

## F

- f 95
- feof 40
- feste Länge (Vergleich nach links) 236
- fgets 40
- Firefox 1.0.7 410
- fopen 40
- Formal-Parameter 21
- formfeed 95
- form-Tag 124
- Formularparameter 127, 185

Fortsetzungszeile (Python) 55  
frei definierbare Zeichenklasse 137, 365  
ftp 59  
Funktionswert 22

## G

gefiltertes Web-Dokument 67  
genügsamer Quantifier 257, 412  
ger 155  
Gesamt-Treffer 17  
gieriger Quantifier 15, 256  
global 423  
globale Mustersuche 15  
globale Substitution 35, 41  
Glyph 75  
g-Modifizier 28, 32, 58, 368, 378, 382, 416,  
418, 419, 423  
grafisch darstellbares Zeichen 178  
graphisch darstellbares Zeichen 75  
Grenzen 19, 27  
Grenzmarkierung 12, 108  
Grenzmarkierung (Python) 55  
grep 51  
grep (Perl) 49  
Groß- und Kleinschreibung 20  
Gruppierung 12, 290

## H

Here-Dokument 101, 128  
Hexadezimalzahl 78  
hidden 127  
horizontaler Tabulator 95

## I

-i (Perl) 92  
IBM 1252 76  
Identifizier (PHP) 20  
if-Bedingung (PHP) 23  
ignoreCase 423  
i-Modifizier 152, 181, 190, 339, 416, 418, 423  
implode 330  
innere Quantifier 259  
Instanz 20, 21  
Internationalisierung 154  
Interne Rückwärtsreferenz 306  
interne Rückwärtsreferenz 303  
interner Modifizier 152, 290, 292  
Internet Explorer 421

Internet Explorer 6.0 410  
Interpolation 24, 94  
interpolativer Kontext 94, 102, 108  
ISO 10646 77, 80  
ISO 3166 155  
ISO 639 155  
ISO 8859-1 76  
ISO 8859-5 86

## J

Ja-Muster 283  
JavaScript 57, 409  
join 330

## K

Kette 12  
Kettenglied 12  
Klammern 31  
Klasse (PHP) 20  
Klassenelement 138  
Klassenname 20  
Kollationssequenz 365  
Kommentar 51, 290  
Kommentar (Perl) 26  
Kommentar (PHP) 20  
Kommentar (Python) 55  
Komplementärmenge 147  
Konstruktor 21  
Konstruktor-Aufruf 423  
Konstruktor-Funktion 417  
Kontext 375  
Kontext-Abhängigkeit 48

## L

lastIndex 423  
lc 107  
LC\_ALL 156  
LC\_COLLATE 156  
LC\_CTYPE 156  
LC\_MESSAGES 156  
LC\_MONETARY 156  
LC\_NUMERIC 156  
LC\_TIME 156  
lcfirst 107  
leere Restkette 329  
leere Zeichenkette 212  
leere Zeichenklassen 149  
leerer regulärer Ausdruck 157

Leerstelle 79, 94, 178  
lexikalisch enthalten 320  
Limit 329  
Linefeed 59  
links assoziativ 247  
links vergleichend 201  
Liste 20, 26  
Liste (Python) 55  
Liste filtern 49  
Listenelement 20  
Listenindex 20  
Listenkontext 28, 386  
literale Notation 423  
literale Zeichenkette 92  
literales Zeichen 60, 92  
literales Zeichenmuster 15, 52, 92  
local-Anweisung 321  
locale 105, 154, 164  
logische Verneinung 50  
logisches Und 50  
lokale Definition 320  
lokale Variable 321  
lokaler Modifier 292  
lookahead assertion 228

**M**  
m (Perl) 27  
map-Operator 398  
Maske 94  
maskieren 149  
maskierte Oktalzahlen 98  
maskierte Zeichenfolge 24  
maskierter Buchstabe 75  
maskiertes Zeichen 94  
match 417  
mathematisches Symbol 181  
maximaler Quantifier 15, 256  
Mengenlehre 147  
Mengenoperation 190  
Metazeichen 15, 32, 94  
minimaler Quantifier 257, 412  
m-Modifier 215, 416, 420, 423  
mnemotechnische Kurzbezeichnung 27  
Modifier 30, 152  
Modifikations Symbole 181  
modifizierter interpolativer Kontext 116  
Mozilla Browser 421  
multiline 423

Muster 11  
Muster ohne Rückkehrpunkte 275  
Musterelement 12  
Musterkette 12, 14  
Mustersuche 11, 22  
mySQL 61

**N**  
-n (Perl) 91  
Name (PHP) 20  
Name (Python) 55  
navigator 429  
negierende Zeichenklasse 36  
negierender Bindungsoperator 315  
negierte Zeichenklasse 143  
Nein-Muster 283  
new 20, 21  
newline 95  
New-Line-Zeichen 59  
nicht definierte Escape-Sequenz 25  
nicht verbrauchendes Musterelement 14,  
16, 40, 43  
Nicht-Wortzeichen 222  
nn 308  
NO-BREAK SPACE 178  
Nullmuster 211, 213, 214  
Nulltreffer 13, 18, 40, 210, 386, 422  
Number, Decimal Digit 180  
Nummerierung (einfangende Klammern)  
303  
nur gruppierende Klammern 290

**O**  
Obergrenze (Quantifier) 274  
Objekt 21  
Objekt (PHP) 20  
Oder-Operation 14  
oder-Operation 137  
Oder-Verknüpfung 27, 51  
oder-Verknüpfung 247, 438  
Oktalzahl 78, 98  
Oktalziffer 78  
o-Modifier 350, 378  
once-only Subpattern 276  
o-Option 397  
open (Perl) 38, 43, 166  
Operation 12  
ord 78

## P

- p 92
- Parentheses 289
- parseFloat 429
- passen 13
- PCRE-Bibliothek 313
- Perl 25
- Perl 5.8.4 263
- Perl Aufruf 61
- Perl kompatibler regulärer Ausdruck 19
- Perl-Skript 25
- PHP 19
- pos 382
- Position im Suchtext 199, 382
- positive Zeichenklasse 138
- POSIX-Klammerausdruck 137, 168, 175
- POSIX-Zeichenklasse 137
- pos-Operator 271
- Pragma 81
- preg\_grep 50
- preg\_match 22, 23
- preg\_match\_all 24
- preg\_replace 34
- PREG\_SPLIT\_DELIM\_CAPTURE 333
- PREG\_SPLIT\_NO\_EMPTY 333
- Primzahl 343
- print (Perl) 29
- print (PHP) 23
- print-Anweisung (PHP) 24
- Priorität 14, 28, 247
- progressive Mustersuche 379, 382, 423
- Python 54

## Q

- qq/ ... / 109
- qr//-Operator 397
- Quantifier 12, 15, 27, 51, 53, 255
- quantifizierte Zeichenklasse 27
- quantifiziertes Musterelement 12
- Quantifizierung 315
- quotemeta 107, 348

## R

- rechts vergleichend 201
- redundante weiße Zeichen 51
- Referenz 349
- Regular Expression 416, 422
- Regulärer Ausdruck 11, 22, 55

- Reihenfolge (Alternativen) 247
- re-Modul (Python) 55
- replace 419
- reset 376
- Restkette 388
- rewrite Modul 433
- RewriteEngine 66
- RewriteRule 65
- Round Brackets 289
- rtrim 172
- rtrim (PHP) 221
- Rückkehrpunkt 202
- rückwärts prüfendes Musterelement 43
- Rückwärtsreferenz 98, 115, 307
- runde Klammern 12, 289

## S

- S 181
- Sc 181
- Schachtelung 229
- Schachtelung (Klammern) 290, 303
- Schrittweite 259
- schwarzes Zeichen 178
- search 417
- setlocale 154
- Shell-artiger Kommentar 20
- shift 348
- Sk 181
- skalare Variable 20, 24, 26
- skalärer Kontext 28, 29, 382
- Sm 181
- s-Modifizier 171, 291, 368, 412
- So 181
- s-Operator 30
- Sortierreihenfolge 154
- source 423
- space 79
- spezielle Variable 26
- splice 329
- split 157, 421
- split (Perl) 46
- Sprachumgebung 105
- SQL-Abfragesprache 62
- Standard-SQL-Muster 62
- Startposition 203
- stdClass (PHP) 20
- STDOUT 85
- Steuer-Zeichen 96

Steuerzeichen 60, 75, 79  
strcoll 157  
Strg-Taste 96  
Strichliste 343  
String 416  
study 399  
Subpattern 290  
Substitution 30, 41, 330  
Substitution (Perl) 30  
Suchbegriff 11  
Suchfunktion 11  
Suchtext 11, 22, 30, 434

## T

Teilausdruck 290  
Teilmuster 12  
test 425  
Testkette 125  
Teststring 127  
Text-Modifikation 104  
Text-Modus 59, 89, 219  
Textmodus 171  
Text-Operator 104  
tr/// 237  
Transferformat 77, 84  
Treffer 12, 13, 22, 199  
Trefferliste 312  
Treffervariable 36, 304, 312, 335  
Trennmuster 385  
Triple Quotes 55  
try-Block 56

## U

U Modifier 259  
Überladung 28  
uc 107  
ucfirst 107  
U-Modifier 368, 369  
Unicode 77, 80  
Unicode Ebene 178  
Unicode Eigenschaft 179  
Unicode Kategorie 179  
unicode plane 179  
Unicode-Codierung 77  
Unicode-Ebene 77  
Unicode-plane 77  
Unicode-Property 137, 168, 178, 179, 188  
use re 396

usort 157  
utf-8 77, 80

## V

Variable 24, 29, 35, 41, 102  
Variablen-Interpolation 102  
Variableninterpolation 376, 419  
Variableninterpolation (Python) 55  
Variablenname (Perl) 26  
Variablenname (PHP) 20  
Variablennamen 103  
verbrauchen 200  
verbrauchendes Musterelement 13, 32, 203  
Verbundanweisung 19, 25  
Verbundanweisung (Python) 55  
Vereinigungsmenge 147  
Vergleich nach links 235  
Vergleich nach rechts 205, 228  
verketten 12  
Verkettung 12, 13, 35  
Verkettungsoperator 337, 343, 412  
Versicherung 203  
Versicherung nach rechts 228  
vertikaler Tabulator 169  
Verzweigung 202  
vi-Editor 58  
vordefinierte Perl Zeichenklasse 168  
vordefinierte Zeichenklasse 137  
vorläufiger Treffer 201  
vorwärts prüfendes Musterelement 43

## W

Wagenrücklauf 59, 95  
Währungssymbol 181  
weiße Zeichen 44  
weißes Zeichen 44, 169  
Wertzuweisung 28, 345  
Wildcard 62  
Windows 24  
Windows 1252 76  
Word-Dokument 190  
Wortgrenze 114, 173, 222  
Wort-Zeichen 222  
Wortzeichen 173

## X

X-Modifier 369  
x-Modifier 51, 142, 152, 272

## **Z**

Zeichen 75  
Zeichenbereich 141  
Zeichencodierung 75  
Zeichengrenze 201  
Zeichenkette 19, 411  
Zeichenkette (Python) 55  
Zeichenklasse 26, 36, 137, 138, 170  
Zeichenklasse . 170  
Zeichenmenge 138  
Zeichenmuster 27, 113  
Zeichenvorrat 36, 75  
Zeilenende 42, 55  
Zeilenvorschub 14, 24, 58, 94  
Zeilenvorschub (Perl) 29  
Zerlegung 45  
Zugriffsmodus 38, 39