3 Der AWK-Rahmen

Ein direkter Vorläufer von Perl war die Sprache AWK, benannt nach den Initialen der Autoren Aho, Weinberger und Kernighan. AWK war ein Spezialwerkzeug, dazu gedacht, um Textdateien Zeile für Zeile zu analysieren und mit dem Ergebnis etwas anzufangen. Wegen der eingebauten Ablauflogik waren AWK-Programme oft rekordverdächtig kurz. Es gibt viele Beispiele, die aus genau einer noch dazu kurzen Zeile bestehen und leisten, wofür man sonst viele Zeilen C++ oder Java bräuchte. Perl ist ein würdiger Nachfolger für AWK; es gibt sogar ein Konvertierungsprogramm `a2p`, das AWK nach Perl konvertiert. Perl hat auch eigene Optionen, die den Interpreter sehr nach AWK aussehen lassen (`-a` und `-n`). Das alles wollen wir nicht weiter verfolgen, sondern uns auf den wesentlichen Kern von AWK konzentrieren.

Während wir im letzten Kapitel noch viele »echte« Übungsaufgaben angegeben haben, können Sie am Ende dieses Kapitels schon wirklich Brauchbares in Perl programmieren.

3.1 Der AWK-Rahmen

Die Erfahrungen mit AWK haben vor allem eines gezeigt:

Viele Probleme kann man durch folgenden Rahmen leicht behandeln:

```
FUER alle zeilen einer textdatei
MACHE FOLGENDES:
    lies eine Zeile
    zerlege die Zeile in Worte
    vergleiche die Zeile mit einem Muster.
    Falls Übereinstimmung, mache etwas.
```

Das hört sich zunächst einmal sehr speziell an. Da wir aber festlegen können, was eine Zeile und was ein Wort ist, wird der Rahmen sich als sehr flexibel erweisen. Obendrein sind die erlaubten Muster, mit denen ein Text untersucht wird, sehr leistungsfähig.

Wir werden in diesem Kapitel zeigen, wie dieser Rahmen mit Leben gefüllt wird.

3.1.1 Teile einer Zeile bearbeiten

Sie haben gelernt, wie eine Textdatei Zeile für Zeile gelesen wird, zum Beispiel mit dem Raumschiff-Operator <>. Der nächste Schritt in dem angestrebten Rahmen ist dann, die Zeile in Worte zu zerlegen. Manchmal sagen wir auch Felder, um zu betonen, dass das Ergebnis nicht einzelne Wörter sein müssen.

Es gibt viele Möglichkeiten, einen String in einzelne Teile aufzuspalten. Am einfachsten zu verstehen ist die Funktion `split`. Die Funktion braucht zwei Dinge: Sie müssen erstens angeben, was zerlegt werden soll und zweitens an welchen Zeichen getrennt wird.

Denken Sie zum Beispiel an einen Satz, den Sie in Worte zerlegen wollen. Die Worte sind dann durch Leerzeichen getrennt. Sie werden der Funktion `split` dann ein Leerzeichen als Trennzeichen mitgeben und außerdem den Satz, um den es geht:

```
...=split(/ /,"Das ist mein Satz");
...=split(/ /,"Das ist mein Satz");
```

split

Das Leerzeichen ist das erste Argument. Damit man es sieht, ist es in Divisionszeichen `/` eingeschlossen. Die Schreibweise wird häufig benutzt und wir erklären Sie gleich näher. Die beiden Argumente von `split` sind in runde Klammern eingeschlossen. Tatsächlich ist das ein Zugeständnis an Pascal-Programmierer, denn genau wie bei `print` sind die runden Klammern hier erlaubt, aber nicht notwendig. Das Ergebnis, das `split` erzeugt – in unserem Beispiel also die vier Worte des Satzes – wird mit einer Zuweisung an eine Variable übergeben.

Neuer Datentyp: Listen

Hier haben wir jetzt ein interessantes Problem. Sie können das Ergebnis nicht ohne Verluste einer skalaren Variablen zuweisen. Denn skalare Variablen waren ja gerade dadurch beschrieben worden, dass sie genau ein Ding speichern können. Und vier Worte sind nun mal drei zuviel. Tatsächlich brauchen wir hier einen neuen Datentyp, der *Liste* oder Feld genannt wird. Wenn Sie schon ein wenig Programmiererfahrung haben, werden Sie an dieser Stelle vielleicht einwenden, dass eine Liste etwas anderes ist als ein Feld. Stimmt – in anderen Programmiersprachen. In Perl gibt es nur eine Sorte von Variablen, und die kann für beides stehen. Wir benutzen das Wort Liste.

Eine vollständige Liste erkennt man dem anderen Zeichen zu Beginn: Eine Liste beginnt mit einem `@`. Das Beispiel kann jetzt fertig geschrieben werden.

```
my @wort_liste = split(/ /,"Das ist mein Satz");
```

In der neu erzeugten Wort-Liste sind jetzt vier Wörter abgelegt, sauber getrennt und jedes für sich. Wie aber kommt man an die Einzelteile heran?

Elemente

Zunächst einmal sind die Wörter – man spricht auch von den *Elementen* der Liste – ohne weiteres Zutun von Ihrer Seite mit Nummern versehen worden, und zwar hat das erste Element die Nummer 0, das zweite die Nummer 1 und so weiter, im Beispiel bis zur Nummer 3. Wenn Sie ein bestimmtes Wort haben wollen, dann ist das einfach. Sie brauchen nur die Nummer des Wortes in eckigen Klammern hinter den Listennamen zu schreiben. Um also das dritte Wort auf dem Bildschirm auszugeben, müssen Sie schreiben:

```
print "$wort_liste[2]\n";
```

Index

Haben Sie diese Zeile genau betrachtet? Dann sollten Ihnen gleiche mehrere Dinge auffallen.

- Es werden eckige Klammern für die Indizes benutzt, genau wie bei den Feldern in den meisten anderen Programmiersprachen. Das überrascht zumeist nur den, der noch unverdorben ist.
- Das dritte Element hat die Nummer 2. Das ist einfach Folge der Tatsache, dass wir mit 0 zu zählen beginnen. Merken Sie sich:
  Der Index ist immer um 1 kleiner als die Nummer des gewünschten Elements.
  Warum man das gemacht hat? Sie werden im Kapitel 4 noch sehen, dass es gute Gründe dafür gibt.
- Wirklich überraschend ist das Dollarzeichen am Beginn! Hatten wir nicht eben gesagt, dass eine Liste durch `@` eingeleitet wird? Nein, denn wenn Sie genau lesen, dann steht da: Eine vollständige Liste wird durch `@` eingeleitet. Wenn Sie nur ein einziges Element haben wollen, dann müssen Sie `$` schreiben, denn ein einziges Element ist ein skalarer Wert.

Skalare in Listen

Sie sehen also, dass der Vorspann `$` oder `@` nicht zum Namen gehört. Er ist mehr ein Hinweis für Perl auf den Typ von Element, der zu erwarten ist. Wenn Sie nur ein Element zulassen wollen, dann schreiben Sie `$`, wenn Sie viele Elemente zulassen wollen, dann schreiben Sie `@` (sollte Ihnen das so weit zu einfach sein, können Sie ja mal überlegen, wie das mit einer Liste ist, die nur aus einem Element besteht. Hat die einen `$` als Vorspann? Die Auflösung gibt es in Kapitel 5).

Für Listen gibt es eine spezielle Form der `for`-Schleife, damit wir alle Elemente der Reihe nach besuchen können.

```
for my $w (@wort_liste) {
    print "$w\n";
}
```

Listen und for

Die skalare Variable `$w` durchläuft alle Variablen in der Liste. Das ist übrigens wörtlich gemeint, denn wenn Sie den Wert der Variablen ändern, wird auch das zugehörige Element in der Liste geändert. Ein praktischer Mechanismus, um Listen zu verändern.

```
my @wort_liste=(1,2,3,4);     # Liste mit 1,2,3 und 4
for my $w (@wort_liste) {
    $w = "-";                 # aendert wort_liste
}
print "@wort_liste\n";        # 4 Minuszeichen mit
                              # Leerzeichen dazwischen
                              # - - - -
```

Listenliteral

In dem Beispiel sehen Sie auch, wie eine Liste einfach mit Werten gefüllt wird. Sie schreiben einfach die gewünschten Werte in runde Klammern, mit Kommata dazwischen. Dieses Ding ist so etwas wie eine *konstante Liste* und wird darum auch *Listenliteral* genannt.

Zum Abschluss geben wir die Liste aus, und zwar interpoliert. Es werden also Listenvariablen zwischen doppelten Anführungszeichen durch die Werte ersetzt. Aha, werden Sie sich sagen, so wird das ja wohl weiter gehen.

Leider nein, denn Listen sind der letzte Typ, der vernünftig interpoliert wird. Außerdem gibt es eine Besonderheit: Beim Interpolieren werden nicht einfach die Listenwerte aneinander geklebt, sondern es erscheint ein Leerzeichen zwischen den einzelnen Werten. Deshalb sieht der interpolierte Ausdruck anders aus, als wenn wir einfach geschrieben hätten:

```
print @wort_liste;      # 4 Minuszeichen
                        # ohne Zwischenraum
                        # ----
```

In diesem Fall werden die Listenelemente wirklich unmittelbar aneinander gesetzt. Es gibt noch eine ganze Menge weiterer Funktionen für Listen. Stellvertretend zeigen wir eine Anwendung zweier wichtiger Vertreter: `sort` und `reverse`.

```
print reverse sort (4,2,5,3,1);    # 5,4,3,2,1
```

Lesen Sie das wie einen deutschen Satz, nur von links nach rechts: Nimm die ganzen Zahlen in der Liste, sortiere sie, drehe die Ergebnisliste herum und gib das Ganze dann auf dem Bildschirm aus. Das Ganze ist übrigens kein übertrieben gutes Beispiel. Lesen Sie im Listenkapitel nach, warum.

Als Letztes in der Erläuterung von `split` bleibt noch, die merkwürdige Schreibweise des ersten Parameters zu erklären, des Trennzeichens. Wir beginnen mit einer (hoffentlich) motivierenden Abschweifung.

Viele Leerzeichen

Eigentlich ist es ja ungeschickt, dass wir uns auf ein einziges Leerzeichen geeinigt haben, um den Raum zwischen zwei Worten zu beschreiben. Wie schnell sind zwei Leerzeichen geschrieben, und wer kann schon im gedruckten Text ohne weiteres zwei Leerzeichen von einem unterscheiden (Lektoren ausgenommen). Eigentlich wollen wir angeben, dass es zwischen zwei Worten eine beliebige Zahl von Leerzeichen geben kann, aber mindestens eines.

Machen Sie sich bitte klar, dass das mit einer einfachen Variablen nicht zu machen ist! Eine Variable hat einen Wert, und der ist entweder ein Leerzeichen oder aber zwei Leerzeichen und dann nicht mehr eines. Wir müssten so etwas wie eine Schleife programmieren! Das wäre dann aber doch zu umständlich.

3.1.2 Muster

Speziell für solche Situationen gibt es in Perl so genannte *Muster*. Wir geben nämlich nicht mehr genau an, was zwischen zwei Worten liegt, sondern nur ein Baumuster. Die Art, wie diese Muster geschrieben werden, heißt auch regulärer Ausdruck (obwohl die Spezialisten unter Ihnen leicht erkennen werden, dass Perls reguläre Ausdrücke viel mehr leisten, als die Mathematiker zulassen würden).

Reguläre Ausdrücke werden in Perl normalerweise zwischen zwei Trennzeichen eingeschlossen. Zumeist wird, Sie ahnen es schon, der Slash oder Divisionsstrich benutzt. Das erste Argument von `split` ist also ein regulärer Ausdruck, der zwischen zweimal `/` eingeschlossen ist.

Sie werden wohl einwenden, dass das Muster nicht besonders aufregend ist: Einfach nur ein Leerzeichen. Stimmt, Muster müssen nicht kompliziert sein. Was würden Sie denn als Muster vorschlagen, das beschreiben soll, dass hier genau ein Leerzeichen erlaubt ist?

Metazeichen

Spannend wird die Geschichte durch einige besondere Zeichen. Es gibt inzwischen eine ganze Menge davon; wir betrachten hier aber nur das Multiplikationszeichen * und das Pluszeichen. Beide Zeichen haben in einem regulären Ausdruck (und auch nur da!) eine besondere Bedeutung.

Der Stern

Ein Stern bedeutet, dass das Zeichen zu seiner Linken beliebig oft wiederholt werden darf. Beliebig oft bedeutet dabei auch: eventuell gar nicht. Wenn sie also in einem regulären Ausdruck schreiben `a*`, dann bedeutet das:

Nimm so viele a wie du willst, vielleicht aber auch gar keine.

Das Pluszeichen

Ein Pluszeichen bedeutet, dass das Zeichen links davon beliebig oft wiederholt werden darf, mindestens aber einmal. Wenn Sie in einem regulären Ausdruck `a+` schreiben, dann muss wenigstens einmal ein `a` vorkommen. Jetzt können wir beliebig viele Leerzeichen zwischen unseren Worten zulassen und mindestens eines verlangen:

```
my @wort_liste=split(/ +/,"Das ist mein Satz");
```

Sie geben vielleicht zu, dass das Leerzeichen im Druck oder am Bildschirm schwer zu erkennen ist. Außerdem würden wir gerne auch noch Tabulatoren zulassen. Dafür gibt es eine weitere Schreibweise. Wenn Sie mehrere Zeichen als Alternative erlauben wollen, dann zählen Sie diese Zeichen in eckigen Klammern auf.

```
my @wort_liste=split(/[     ]+/,"Das ist mein Satz");
```

Sehen Sie jetzt den Tabulator? Nein, das ist auch noch nichts. Wie wäre es mit:

```
my @wort_liste=split(/[ \t]+/,"Das ist mein Satz");
```

Das Fluchtsymbol

Speziell für solche Zwecke hat der Tabulator ein eigenes Fluchtsymbol bekommen. Sie erinnern sich: Fluchtsymbole waren meist Paare von Zeichen, die mit \ beginnen, gefolgt von einem Buchstaben. Das `t` steht natürlich für Tabulator und das ganze ist ein Symbol, das nur eine andere Schreibweise für einen Tabulator erlaubt: Obwohl Sie also zwei Zeichen schreiben, nämlich `\` und `t`, wirkt das Ganze wie ein einziges Zeichen. Das Gleiche gilt übrigens auch für die eckigen Klammern im regulären Ausdruck. Sie zählen hier die möglichen Alternativen von Zeichen auf. Die Klammer steht darum nur für ein Zeichen, das eines der aufgezählten Zeichen sein muss.

Muster suchen

Muster haben noch eine Eigenschaft, die Ihre Anwendung sehr angenehm macht: Sie können benutzt werden, um zu entscheiden, ob ein Muster in einem String vorkommt. Dazu werden Sie formal wie ein Wahrheitswert verwendet. Natürlich müssen wir zuerst festlegen, welcher String untersucht werden soll. Die Schreibweise dafür ist leider ziemlich missverständlich, denn der gewünschte String wird mit dem regulären Ausdruck durch das Symbol =~ verknüpft. Mit der nächsten Zeile prüfen Sie, ob in der Variablen $string das Wort Hallo vorkommt:

```
if ($string=~ /Hallo/) { print "Hier kommt Hallo vor!";
 }
```

Der Operator =~

Dieser Operator erinnert zwar an eine Zuweisung, hat aber überhaupt nichts damit zu tun! Er weist Perl wirklich nur an, den Inhalt der Variablen `$string` für den Vergleich zu benutzen.

Mit dieser Schreibweise können wir eine Textdatei nach einem Wort oder einem allgemeinen Muster durchsuchen! Das Muster darf dabei auch in einer Variablen enthalten sein. Im nächsten Beispiel soll der Benutzer erst das gesuchte Wort eingeben, und wir geben dann alle Zeilen aus, die das gesuchte Wort enthalten.

```
#!/usr/bin/perl -w
use strict;
print "Geben Sie das gesucht Wort ein: ";
my $gesucht = <STDIN>;                # Wort lesen
chomp($gesucht);                      # Zeilenende
                                      # abschneiden
while(defined (my $zeile = <>)) {
    if ($zeile =~ /$gesucht/) { print "$zeile"; }
}
```

Wenn Sie das Beispiel einmal selbst ausprobieren, haben Sie wahrscheinlich die merkwürdige Zeile mit dem `chomp` vergessen und Ihr Programm hat nicht richtig funktioniert. Was ist hier passiert?

chomp

Nun, die ganze Verarbeitung einer Datei ist zunächst einmal ausgerichtet auf Zeilen einer Textdatei. Wir lesen Zeile für Zeile. Das Ende einer Zeile ist normalerweise durch (mindestens) ein eigenes Zeichen gekennzeichnet. Das Zeichen

ist natürlich ein ganz Besonderes und wird mit dem Fluchtsymbol `\n` notiert. Über die Tastatur erzeugen Sie das Zeichen jedes Mal, wenn Sie die Enter-Taste eingeben.

Wenn Sie nichts dagegen tun, wird dieses Zeichen mitgelesen und steht in der Variablen `$eingabe`. Sobald Sie das Wort Hallo eingeben, wird also nicht einfach nach Hallo gesucht, sondern nach Hallo mit einem nachfolgenden Zeilenende.

Die Funktion `chomp` leistet nun genau, was wir hier brauchen; sie schneidet nämlich ein Zeilenende am Schluss einer Zeile ab, falls es eines gibt. Ist keines da, macht `chomp` gar nichts. Können Sie jetzt vorhersagen, was das Programm ohne `chomp` macht?

Ohne `chomp` finden Sie das gesuchte Wort genau dann, wenn es in der Textdatei am Ende einer Zeile steht. Probieren Sie das einmal aus!

Wozu sind Muster gut?

Jetzt haben Sie also auch einen ersten Eindruck von den regulären Mustern gewonnen und fragen sich vielleicht: Na und? Zwar werden wir noch ein ganzes Kapitel nur mit regulären Ausdrücken füllen, aber um Ihre Anschauung gleich richtig zu orientieren, sind hier einige Beispiele, die Sie zwar noch nicht vollständig verstehen, aber doch schon anwenden können. Unter anderem können reguläre Muster zwei Dinge besonders gut:

- einen String analysieren und in Teile zerlegen, um dann eventuell
- Teile eines Strings durch etwas anderes zu ersetzen.

Hier sind zwei konkrete Beispiele:

Suchen

Wir nehmen an, dass in der Variable `$eingabe` irgendwo ein Datum enthalten ist. Sie wollen nur das Datum haben, der Text interessiert Sie nicht. Das geht so:

```
my ($tag, $monat, $jahr) =
       ($eingabe =~
           /([0-9][0-9]).([0-9][0-9]).([0-9][0-9])/);
```

Eine wirkliche Erklärung gibt es, wie gesagt, später in einem eigenen Kapitel. Hier nur eine Orientierung. Links von der Zuweisung steht ein Mustervergleich: Es wird die Variable `$eingabe` mit dem Muster in den Slashes verglichen. Die Zeichenklassen in den eckigen Klammern kennen Sie schon. Sie lernen jetzt, dass man auch Bereiche angeben kann, also `[0-9]` statt `[0123456789]`. Ziemlich praktisch so weit. Ebenfalls neu ist, dass Sie beim Mustervergleich ein Ergebnis bekommen, nämlich eine Liste aller Werte in runden Klammern. Diese Liste können Sie zuweisen, zum Beispiel einer Liste von Variablen.

Das Muster ist nicht besonders praktisch, denn die Jahresangabe darf nur zweistellig sein, und außerdem wird auch von jeder Monatsangabe eine füh-

rende Null verlangt. Das ist aber leicht zu ändern. Es gibt einen subtileren Fehler: Die Punkte in einem regulären Muster werden nicht als Punkte interpretiert, sondern als besonderes Zeichen, als Metazeichen, das für *irgendein Zeichen* steht. Zusammengefasst heißt das: Unser Muster akzeptiert 48A123B57 als Datum, nicht dagegen den 1.1.2002.

Die Lösung für das letzte Problem ist einfach: Entwerten Sie das Metazeichen, indem Sie einen Backslash davor schreiben. Ein insgesamt besseres Muster könnte so aussehen:

```
/([0-9]+)\.([0-9]+)\.([0-9]+)/
```

Es geht aber noch erheblich besser – siehe das Kapitel über reguläre Ausdrücke.

Ersetzen

Es kommt aber noch besser. Sie können Muster nicht nur suchen, Sie können ein Muster auch durch etwas ersetzen, und dabei muss der Ersatztext nicht einfach fest sein. Wenn Sie etwa ein Datum aus dem deutschen Format (Tag.Monat.Jahr) in das amerikanische übersetzen wollen, dann sieht das so aus:

```
$eingabe =~ s/([0-9]+)\.([0-9]+)\.([0-9]+)/$2.$1.$3/;
```

Sie erkennen unser verbessertes Muster aus dem vorhergehenden Absatz. Die erste Änderung ist der Buchstabe `s` vor dem Muster. Das s steht für substitute, also etwa »Ersetze das«. Beim Ersetzen braucht man immer zwei Dinge: Das, was ersetzt werden soll, und zweitens, wodurch es ersetzt werden soll. Diese zweite Angabe schreiben wir einfach hinter das Muster und beenden Sie auch durch einen Backslash. Sie sehen, dass der Buchstabe `s` eine ganz handfeste Operation auslöst. Wie immer, wenn eine Operation verlangt werden kann, ohne dass runde Klammern im Spiel sind, sprechen wir von einem Operator.

So weit war das einfach und ist für die meisten Programmiersprachen erhältlich, als mehr oder weniger komfortable Stringfunktion. Spannend wird die Sache durch die Form unseres Ersatztexts. Wir dürfen uns nämlich beim Ersetzen auch wieder auf die Klammern im vorderen Teil beziehen. Dazu werden Nummern verwendet, die mit 1 beginnen – aus den beliebten historischen Gründen ist der Nummer 0 etwas ganz anderes zugeordnet. Die Fundstellen sind Teile des vorgegebenen Strings, also skalar, und haben darum einen Dollar als Präfix. Die Teile entsprechen gerade den geklammerten Teilen des regulären Ausdrucks. Unser Befehl vertauscht den Tag `$1` mit dem Monat `$2`. Lassen Sie sich aber nicht verwirren. Der Ersatztext selbst ist kein Muster. Das würde keinen Sinn ergeben, denn Sie müssen eindeutig sagen, was eingefügt werden soll. Deshalb haben die Punkte im zweiten Teil auch keine besondere Bedeutung, sondern sind einfach das: Punkte.

Sollte Sie diese Zeile nicht beeindrucken, dann programmieren Sie diese Lösung einfach einmal in einer anderen Programmiersprache Ihrer Wahl. Sie werden schon sehen, was Sie davon haben.

Eine Anmerkung noch: Der Operator `s` ersetzt immer die erste Fundstelle des Musters und hört dann auf. Wenn das nicht Ihren Wünschen entspricht, können Sie das Verhalten des Operators durch Optionen steuern, die hinten, vor dem Strichpunkt, eingefügt werden. Die wichtigste dieser Optionen ist wohl das `g` wie globales Ersetzen.

```
$zeile =~ s/;/./g;    # Alle Strichpunkte
                      # durch Punkte ersetzen
```

Durch die Anweisung im Beispiel werden alle Strichpunkte durch Punkte ersetzt. Wenn Sie das `g` entfernen, gilt das nur für den ersten Strichpunkt.

Anker

Jetzt haben Sie einen ersten Eindruck davon, was reguläre Ausdrücke sind und was sie leisten können. Natürlich gibt es noch viele weitere Metazeichen, zum Beispiel das Symbol | um alternative Möglichkeiten auszuzählen. Wichtig sind vor allem noch die Positionsangaben, die oft auch Anker genannt werden. So bezeichnet der Dollar `$` innerhalb eines regulären Ausdrucks das Ende einer Zeile. Das Gegenstück sieht noch etwas merkwürdiger aus: Der Zeilenanfang wird durch das Karet ^ repräsentiert. Wir beenden unsere Blitztour durch die Welt der regulären Ausdrücke mit zwei Beispielen.

Welches Muster beschreibt einen Strichpunkt am Ende eine Zeile? Ganz einfach, Sie müssen nur daran denken, dass *Zeilenende* gleichbedeutend mit einem Dollar ist!

```
;$
```

Und was ist eine leere Zeile? Nun, in einer leeren Zeile folgt auf den Anfang unmittelbar das Zeilenende.

```
^$
```

Wie sieht dann ein Programm aus, das die leeren Zeilen in einer Textdatei zählt? So:

```
my $anzahl = 0;                  # Zaehler
while(my $zeile = <>) {
    if ($zeile =~ /$^/) {        # falls gefunden
        ++$anzahl;               # zaehlen
    }
}
print " $anzahl leere Zeilen"
```

3.2 Probleme und Lösungen

In diesem Abschnitt haben wir einige Probleme versammelt, die Sie mit Ihrem jetzigen Wissen in Perl schon bearbeiten können.

3.2.1 Die Zeilen einer Textdatei einen Tabulator einrücken

Die Aufgabe besteht darin, alle Zeilen einer Datei mit einem Vorspann zu versehen, der hier ein Tabulator sein soll. Eine ausführliche Lösung für dieses Problem sieht so aus:

```
#!/usr/bin/perl -w
use strict;
$^I='.bck';                          # Inplace-Edietieren
while(defined(my $zeile = <>)) {     # Solange was da ist
    print "\t$zeile";                # Schreiben
}
```

Wir benutzen den Raumschiff-Operator <>, um die Zeilen der Textdateien zu lesen. Die Zeilen werden in der Variablen `$zeile` gespeichert und dann mit `print` ausgegeben. Dazu benutzen wir die Interpolation, das heißt, wir haben die Variable `$zeile` einfach in doppelte Anführungszeichen eingeschlossen und den Rest Perl überlassen, das zuverlässig an Stelle von `$zeile` den Inhalt der Variablen einfügt. Den Tabulator vor der Variablen haben wir als Fluchtsymbol `\t` geschrieben. Das ist zwar nicht zwingend, ist aber beim Lesen besser zu erkennen als ein echter Tabulator. Die neue Datei wird mit dem Perl-Trick *Inplace-Editieren* geschrieben. Sie erinnern sich: Dazu wird in der Variablen `$^I` die Endung einer Backup-Datei angegeben. Jede Ausgabe mit `print` wird dabei direkt in die Zieldatei umgeleitet.

Die Kurzvariante

So weit sollten Sie die Lösung verstanden haben. Tatsächlich wir ein hartgesottener Perl-Programmierer das alles in einer Zeile schreiben, die so aussieht:

```
I^I='.bck';
while(<>) { print "\t$_"; }
```

Wir werden die Details im Kapitel über Anweisungen besprechen; hier wird nur so viel angemerkt: In vielen Situationen können Sie die Variable, mit der Sie arbeiten wollen, einfach weglassen. So entfällt in unserem Beispiel die Variable `$zeile`. Perl benutzt dann eine Art universelle Standardvariable, die leider auf den unschönen Namen `$_` hört. Die Prüfung, ob `$_` definiert ist, wird von Perl für die Variable `$_` auch automatisch vorgenommen.

Sie haben jetzt gelernt, wie ein Tabulator einer Zeile vorangestellt wird. Der Tabulator war natürlich nur ein Beispiel für viele Möglichkeiten. Sie können zum Beispiel auch die Datei mit Zeilenummern versehen.

3.2.2 Zählen, wie oft ein Wort in einer Textdatei vorkommt

Wie oft kam in Ihrem letzten Perl-Programm das Wort `if` vor? Sie werden jetzt doch hoffentlich nicht zu zählen anfangen, sondern ein Perl-Programm schreiben. Gerade dann, wenn die Anzahl noch leicht zu überblicken ist, kann sich das lohnen.

Sie können zum Suchen ein reguläres Muster verwenden, das Sie in einem `if` auswerten. Wird das Muster gefunden, dann wird die Variable `$anzahl` hochgezählt.

```
my $gesucht = "MEINWORT";
my $anzahl = 0;                          # Zaehler
while(defined (my $zeile = <>)) {
    if ($zeile =~ /$gesucht/) {          # falls gefunden
        ++$anzahl;                       # zaehlen
    }
}
print "Das Wort wurde $anzahl Mal gefunden"
```

Wie lange wäre ein entsprechendes Programm in Ihrer (bisherigen) Lieblingsprogrammiersprache? Falls Ihnen das Programm immer noch zu lang ist, kommt hier eine Kurzfassung, die aber auch schon Anweisungen aus den nächsten Kapiteln benutzt.

```
my $gesucht = "MEINWORT";
my $anzahl = 0;
while(<>)) { ++$anzahl if /$gesucht/ }
print "Das Wort wurde $anzahl Mal gefunden"
```

Die Variable `$zeile` haben wir durch die automatische Variable `$_` ersetzt. Das »umgekehrte `if`« (siehe Folgekapitel) erlässt Ihnen auch noch die Klammern. Viel kürzer geht es dann (fast)[1] nicht mehr.

3.2.3 Die dritte und zweite Spalte einer Textdatei ausgeben

Angenommen, Sie haben eine Textdatei, in der einzelne Spalten durch Tabulatoren getrennt sind. Wir nehmen an, dass jede Zeile mindestens drei solche Spalten enthält und im eigentlichen Text keine Tabulatoren vorkommen.

Die Aufgabe ist jetzt, die dritte und die zweite Spalte auszugeben – also in falscher Reihenfolge. Außerdem soll ein Doppelpunkt zwischen den beiden Spalten eingefügt werden. Hier haben Sie die Lösung:

1 Wenn Sie es unbedingt wissen wollen: Mit den Optionen –p und –a können Sie auch noch das `while` und das `split` weglassen. Bleibt das `if`. Das ist dann AWK pur.

```
while(my $zeile = <>) {
    my @worte = split /\t/, $zeile;
    print "$worte[2]:$worte[1]\n"
}
```

Wir zerlegen die Zeile in Worte mit der Funktion `split`. Als Trennzeichen haben wir ein einzelnes Tabulatorzeichen `\t` angegeben. Da es sich beim dem ersten Argument von `split` um einen regulären Ausdruck handelt, darf auch dem Tabulator ein Pluszeichen folgen. Dann wäre auch eine Folge von Tabulatoren erlaubt. Es ist dann aber andererseits nicht mehr möglich, eine leere Spalte anzugeben.

```
while(defined (my $zeile = <>)) {
    my @worte = split /\t/, $zeile;
    print "$worte[2]:$worte[1]"
}
```

Achten Sie bei der Ausgabe darauf, dass das erste Wort die Nummer 0 hat, das dritte Wort also die Nummer 2. Die einzelnen Worte sollten mit dem Zeichen `$` angekündigt werden.

3.2.4 Die Zeilen vor und nach einem gesuchten Wort ausgeben

Es gibt inzwischen eine ganze Reihe von Programmen – auch unter Windows – mit denen man die Zeilen einer Textdatei finden kann, die ein gesuchtes Wort enthalten. Oft ist aber die Umgebung der Suchstelle wichtig. Deshalb geben wir in diesem Abschnitt ein Programm an, das zusätzlich die Zeile vor und nach der Suchstelle ausgibt.

Die Lösung arbeitet ganz klassisch – Zeile für Zeile – und merkt sich jeweils die beiden zurückliegenden Zeilen in eigenen Variablen.

```
#!/usr/bin/perl -w
use strict;
my $eingabe = <STDIN>;
chomp($eingabe);
my $vorher='';
my $vor_vorher='';
while (defined(my $zeile=<>)) {
    if ($vorher =~ /$eingabe/) {
        print "$vor_vorher$vorher$zeile";
    }
    $vor_vorher = $vorher; # die letzte Zeile ist
                           # jetzt vorletzte
    $vorher = $zeile;      # die aktuelle Zeile
                           # ist jetzt die
                           # letzte
}
```

Die Hauptschwierigkeit ist es, den richtigen Zeitpunkt zu finden, zu dem die letzte und vorletzte Zeile aktualisiert werden sollen. Wir haben uns entschieden, das ganz am Ende der Schleife zu tun. Die letzten beiden Zuweisungen sollen die richtigen Werte für einen nächsten Durchlauf der Schleife erzwingen – falls es einen solchen gibt. Beim nächsten Durchlauf ist die jetzt aktuelle Zeile die zuletzt gelesene, die aktuell letzte wird zur vorletzten.

Details

Betrachten wir noch einige Implementierungsdetails. Damit nichts durcheinander geht, lesen wir zunächst das gesuchte Wort von der Tastatur und geben dabei ausdrücklich `STDIN` als Quelle an. Die Datei lesen wir mit dem Raumschiff-Operator.

Das Programm hat in seiner jetzigen Form einige Macken. Sie können zwar mehrere Dateien durchsuchen. Angenommen aber, das gesuchte Wort ist in der ersten Zeile der zweiten Datei. Dann wird die letzte Zeile der ersten Datei ausgegeben, was verwirrend ist. Außerdem hätte man in diesem Fall schon gerne die Information, in welcher Datei etwas gefunden wurde.

eof

Für die erste Verbesserung verwenden wir die Funktion `eof`, die prüft, ob das Programm am Ende einer Datei angekommen ist. In diesem Fall weisen wir den Variablen einfach einen leeren String zu. Den Namen der aktuellen Datei können Sie einer so genannten vordefinierten Variable entnehmen. Perl kennt eine ganze Reihe von solchen vordefinierten Variablen; sie haben zum Beispiel schon `@ARGV` angewendet, um die Argumente in der Kommandozeile zu lesen.

Es gibt zum Beispiel auch eine Variable, in der die aktuelle Zeilennummer enthalten ist – wir müssten also gar nicht selbst zählen. Die Variable mit dem aktuellen Dateinamen des Raumschiff-Operators heißt `$ARGV`. Wahrscheinlich stutzen Sie jetzt – eben bezeichnete `@ARGV` noch die Argumente auf der Kommandozeile. Ja, aber das ist auch eine Liste, der Dateiname dagegen ist ein Skalar. Sie können für Listen und Skalare durchaus die gleichen Namen verwenden, ohne dass Perl durcheinander kommt. Ob das auch für den Programmierer gilt, das müssen Sie entscheiden. Diese Eigenheit ist jedenfalls gemeint, wenn Sie hören:

Listen und Skalare leben in verschiedenen Namensräumen.

Gleiche Namen, verschiedene Dinge. Damit sieht unser Programm so aus:

```
#!/usr/bin/perl -w
use strict;
my $eingabe = <STDIN>;
chomp($eingabe);
my $vorher='';
my $vor_vorher='';
while (my $zeile=<>) {
    if ($vorher =~ /$eingabe/) {
        print "Datei: $ARGV\n$vor_vorher$vorher$zeile";
    }
```

```
    $vor_vorher = $vorher;
    $vorher = $zeile;
    if (eof) {
        $vor_vorher = '';
        $vorher = '';
    }
}
```

3.2.5 Die Ausgabe meines Programms in einer Datei speichern

Wahrscheinlich sind Sie es sehr schnell leid, dass die Ausgaben mit `print` so flüchtig sind. Sie würden gerne die Ergebnisse nicht nur kurz an sich vorbeiziehen sehen, sondern diese als Datei speichern.

Dafür braucht man entweder einige Perl-Anweisungen (`open`, `print`, `close`), die Sie in Kapitel 7 nachlesen können, oder spezielle Optionen beim Aufruf von Perl. Sie können aber jetzt schon einiges erreichen, indem Sie das Inplace-Editieren verwenden und der Variablen `$^I` eine Endung zuweisen.

Sofern Ihr Programm keine weiteren Eingaben verlangt, können Sie auch etwas anderes machen: Sie leiten einfach die Ausgabe Ihres Programms in eine Datei um:

```
perl mein_programm.pl > Ergebnis.txt
```

Seien Sie aber vorsichtig: Sollte es eine Datei Ergebnis.txt in Ihrem Arbeitsverzeichnis geben, wird sie ohne weiteres Aufhebens gelöscht.

3.2.6 Reguläre Muster und Bedingungen

Es lohnt sich, die Schreibweisen für reguläre Muster zu sammeln.

Sie prüfen, ob ein Muster vorhanden ist, auf diese Art:

```
if ($zeile =~ m/MUSTER/) {...}
```

Das `m` steht für `match` und ist optional; wir haben es die ganze Zeit weggelassen. In einem Muster können Sie Zeichenklassen verwenden, wie

`[0-9]` für eine Ziffer oder

`[A-Z]` für einen Großbuchstaben oder auch

einen Punkt `.` für ein beliebiges Zeichen (außer einem Zeilenende).

Sie können beliebige Wiederholungen eines Zeichens zulassen, indem Sie einen Stern hinter das Zeichen schreiben. Ist ein Zeichen zu wenig, verwenden Sie runde Klammern: `(ab)*` sind beliebig viele Wiederholungen von `ab`. Beliebig schließt keinmal ein. Ist das zu wenig, verwenden Sie das Pluszeichen.

Sie können auch einen String analysieren. Dabei werden die Teile eines Musters gespeichert, die Sie in runde Klammern verpackt haben. Sie bekommen die Teile als Liste zurück. So bekommen Sie die ersten drei Zeichen der Variablen `$eingabe`.

```
my ($a, $b, $c) = ($eingabe =~ /(.)(.)(.)/);
```

Sie können ersetzen, indem Sie den Operator `s` vor das Muster schreiben und den Ersatztext dahinter.

```
$zeile =~ s/Hallo/Hello/;  # Hallo zu Hello
```

Dabei können Sie sich sogar auf runde Klammern im ersten Teil beziehen.

```
$zeile =~ s/(.)/$1$1/;     # Zeichen verdoppeln
```

Was immer das erste Zeichen von `$zeile` auch war – es steht jetzt doppelt da.

Reguläre Muster tauchen in Perl an vielen unvermuteten Stellen auf. Sie können zum Beispiel `chomp` mit einem regulären Ausdruck füttern, der festlegt, was ein Zeilenende ist. Vor allem aber können Sie `split` mit einem regulären Ausdruck versehen, um Trennzeichen zu definieren. Das Beispiel erlaubt beliebig viele Leerzeichen als Trenner.

```
my @felder = split/[ ]+/,$zeile;
```

Kombiniert mit dem Operator <> können Sie schon interessante Dinge machen.

3.3 Übungen

Sie können jetzt schon viele nützliche Dinge machen; die Übungen sollen Ihnen einige Ideen geben. Um die Arbeit speziell mit den AWK-Rahmen zu trainieren, können Sie sich in den Übungen darauf beschränken, Ihr Ergebnis auf dem Bildschirm auszugeben.

Übung 1: Die erste Spalte

Angenommen, Sie haben eine Textdatei, die in Spalten mit einzelnen Worten gegliedert ist. Schreiben Sie ein Programm, das die erste Spalte allein ausgibt.

Übung 2: Alle Zeilen, die das Wort head enthalten

Schreiben Sie ein Programm, das alle Zeilen einer Textdatei ausgibt, die das Wort »head« enthalten.

Übung 3: Reguläre Ausdrücke

Hier haben Sie einen »Regulären-Ausdruck-Trainer«.

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

```
use strict;
my $beispiel = ' ';
while ($beispiel ne '') {
    print "Geben Sie den Text ein: ";
    my $beispiel = <STDIN>;
    chomp($beispiel);
    print "Geben Sie den regulaeren Ausdruck ein: ";
    my $regex = <STDIN>;
    chomp($regex);
    if ($beispiel =~ /($regex)/) {
        print "Das passt, und zwar: =>$1<=\n";
    } else {
        print "Passt leider nicht!\n";
    }
}
```

Bringen Sie das Ding zum Laufen und vervollständigen Sie die Tabelle. In der letzten Spalte sollen Sie eintragen, auf welches Zeichen des Beispieltexts das Muster passt. Sie können das mit dem »Trainer« ausprobieren

Tabelle 3.1: Reguläre Ausdrücke

Muster	Bedeutung	Beispieltext	passt auf
	das erste Zeichen	`Hallo`	H
`.*`		`Hallo`	
`^.`		`Hallo`	
`.$`		`Hallo`	
	Zeichen in runden Klammern	`(Das sind (viele) Klammern)`	
	Whitespaces (Leerzeichen, Tabs)	`Das sind 4 Leerzeichen`	
`[a-z]`	Hallo		
	5 Ziffern	`b1234567`	
	Ein kleines a und der Buchstabe danach	`Hallo`	
`.*H`		`Hallo`	

Übung 4: Zähle, wenn ...

Schreiben Sie ein Programm, das alle Zeilen einer Textdatei zählt, die das Wort `table` enthalten.

Übung 5: Ersetzen intelligent genutzt

Schreiben Sie ein Programm, das den Inhalt einer Textdatei auf dem Bildschirm unverändert ausgibt, mit einer Ausnahme: Alle Worte sollen in runde Klammern eingeschlossen sein.

Übung 6: Die erste Spalte

Angenommen, Sie haben eine Textdatei, die in Spalten mit einzelnen Worten gegliedert ist. Schreiben Sie ein Programm, das die erste Spalte allein ausgibt.

Übung 7: Die erste Spalte, wenn...

Angenommen Sie haben eine Textdatei, die in Spalten mit einzelnen Worten gegliedert ist. Schreiben Sie ein Programm, das die erste Spalte allein ausgibt, vorausgesetzt, in dieser Zeile ist das Wort *Statistik* enthalten.

Übung 8: Mehr als zwei Felder

Sie haben eine Textdatei, die in Spalten mit einzelnen Worten gegliedert ist. Diesmal nehmen wir aber an, dass in jeder Zeile unterschiedlich viele Worte stehen. Schreiben Sie ein Programm, das alle Zeilen ausgibt, die mehr als zwei Worte enthalten.

Übung 9: Zeilen als Felder

In dieser Übung sollen Sie Ihr Verständnis von einer Zeile etwas erweitern. Sie können so ziemlich frei definieren, was als Ende der Zeile angesehen wird, indem Sie der Variablen $\ einen Wert zuweisen. Diese Variable ist mit `\n` vorbelegt; Sie dürfen einen beliebigen, regulären Ausdruck zuweisen.

Schreiben Sie ein Programm, das jede Zeile einer Textdatei als Feld interpretiert und nur völlig leere Zeilen als Trennzeichen. Geben Sie alle Felder verbunden mit einem Pluszeichen in einer Zeile aus. Die leeren Zeilen sollen verschwinden.

Übung 10: Mehr Spalten

Sie haben eine Textdatei, die in Spalten mit einzelnen Worten gegliedert ist. Schreiben Sie ein Programm, das die zweite und dritte Spalte ausgibt, vorausgesetzt, in dieser Zeile ist das Word *Dringend* (mit großem D) enthalten.

Übung 11: Wie oft?

Schreiben Sie ein Programm, das ausgibt, in wie vielen Zeilen einer Textdatei das Wort *Dringend* vorkommt.

Übung 12: Spaltensumme

Angenommen Sie haben eine Textdatei, die in Spalten mit einzelnen Worten gegliedert ist. Die Worte sind diesmal aber von besonderer Art: Es sind ganze Zahlen. Schreiben Sie ein Programm, das alle Zahlen in der zweiten Spalte addiert und die Summe ausgibt.

Übung 13: Umlaute

Schreiben Sie ein Programm, das alle Umlaute einer Textdatei in Ihre HTML-konformen Darstellungen umsetzt. Dabei wird dann `ä` zu `&auml;` und `Ä` zu `&Auml;`. Den Rest können Sie sich wohl erschließen.

Die Lösung soll mit dem Substitutionsoperator `s` arbeiten.

Übung 14: Mehr als 20 Zeichen

Schreiben Sie ein Programm, das alle Zeilen einer Textdatei ausgibt, die mehr als 20 Zeichen enthalten.

Übung 15: Sieben Wörter

Schreiben Sie ein Programm, das jede Zeile einer Textdatei ausgibt, die mit Name beginnt und aus genau sieben Wörtern besteht.

Übung 16: Umgekehrte Zeilen

Schreiben Sie ein Programm, das die Reihenfolge der Wörter in einer Zeile umkehrt.

Übung 17: Von hinten nach vorn

Schreiben Sie ein Programm, das die Reihenfolge der Zeilen einer Datei umkehrt.

Das ist etwas anderes, als bei der vorangegangenen Aufgabe! Diesmal soll aus:

```
1. Zeile
2. Zeile
```

werden

```
2. Zeile
1. Zeile
```

Übung 18: Umleitung der Ausgabe

Wenn Sie sich fragen, was man mit dem Programm aus der vorangegangenen Aufgabe machen kann, probieren Sie die *Umlenkung der Standardausgabe* aus.

3.4 Tipps

Tipp zu Übung 1

Verwenden Sie die Funktion `split`, um eine Textzeile in Worte zu zerlegen.

Tipp zu Übung 2

Verwenden Sie ein Muster, um die Zeilen herauszufinden.

Tipp zu Übung 3

Sie beenden den »Trainer« durch Eingabe eines leeren Textes.

Es gibt auch ein paar fertige Zeichenklassen wie `\w` (für Whitespace) und `\s` für alphanumerische Zeichen oder `\d` für Ziffern. Es ist halt bequemer, `\d` zu schreiben statt `[0-9]`.

Tipp zu Übung 4

Verwenden Sie den Operator <> zum Lesen der Datei und eine Mustersuche mit dem Operator m.

Tipp zu Übung 5

Denken Sie sich einen regulären Ausdruck aus, der ein Wort beschreibt. Ersetzen Sie jedes Wort durch sich selbst in runden Klammern.

Tipp zu Übung 6

Verwenden Sie `split`, um die Zeile in Spalten zu zerlegen.

Tipp zu Übung 7

Prüfen Sie mit einem regulären Ausdruck, ob das gesuchte Wort in der Zeile enthalten ist. Falls ja, verfahren Sie wie in der vorangegangenen Aufgabe.

Tipp zu Übung 8

Verwenden Sie die Funktion `split`.

Tipp zu Übung 9

Verwenden Sie `split`, um die Zeile in Spalten zu zerlegen.

Tipp zu Übung 10

Verwenden Sie einen Mustervergleich, um genau diese Zeilen herauszusuchen.

Tipp zu Übung 11

Sie brauchen eine zusätzliche skalare Variable als Zähler.

Tipp zu Übung 12

Sie brauchen eine zusätzliche skalare Variable für die Summe.

Tipp zu Übung 14

Die Länge eines Strings können Sie mit der Funktion `length` ermitteln.

Tipp zu Übung 16

Verwenden Sie die Funktion `reverse`, um die Liste umzudrehen, die `split` liefert.

Tipp zu Übung 17

Lesen Sie alle Zeilen in eine Liste, und verwenden Sie die Funktion `reverse`, um die Liste umzudrehen.

Tipp zu Übung 18

Vielleicht haben Sie noch nie mit einer *Umleitung der Ausgabe* gearbeitet. Starten Sie irgendein Programm in der DOS-Box, zum Beispiel dir. Die Ausgabe von dir können Sie jetzt in einer Datei speichern, indem Sie ein Größer-Zeichen > anhängen:

```
dir > Ergebnis.txt
```

Unter Unix können Sie das Gleiche mit dem Befehl `ls` machen:

```
ls > Ergebnis.txt
dir > Ergebnis.txt
```

Seien Sie vorsichtig beim Experimentieren. Sollte es die Zieldatei schon geben, wird diese gnadenlos gelöscht.

3.5 Lösungen

Lösung zu Übung 1

```
while(my $line = <>) {
    my @list = split / /, $line;
    print $list[0];
}
```

Die Lösung hat den Nachteil, dass Sie nur ein Leerzeichen als Trennzeichen zwischen den Spalten zulassen. Sie werden bald lernen, wie man das verbessert.

Das Gleiche geht auch wieder kürzer (einfacher?):

```
while (<>) { print ((split / /)[0]); }
```

Mit den geeigneten Optionen geht es noch kürzer:

```
#!/usr/bin/perl -a -n
print $F[0];
```

Wir haben dabei die notwendigen Optionen in der so genannten »She-Bang«-Zeile angegeben. Das ist die erste Zeile, die wie ein Kommentar beginnt und den Pfad für Perl enthält. Diese Zeile wird unter Unix benutzt, um der Shell die benutzte Programmiersprache anzugeben; unter Windows werden immerhin die Optionen teilweise berücksichtigt. Sie können sich schnell einen Überblick verschaffen, was an Optionen möglich ist. Geben Sie ein:

```
perl --help | less
```

Die Angabe `--help` bewirkt, dass eine Erklärung angezeigt wird. Da diese normalerweise zu lang ist, haben wir die Ausgabe in das Anzeigeprogramm `less` umgeleitet. Sie beenden dieses Programm durch die Eingabe von [Q].

Lesen Sie mal nach, was die Optionen `-a` und `-n` bei Perl bewirken.

Lösung zu Übung 2

```
while(my $line = <>) {
    if ($line =~ /head/) { print $line; }
}
```

Falls Ihnen das zu lang ist, haben Sie einige Möglichkeiten. Sie können sich der »nachklappernden Form« der Bedingung bedienen (die wir streng genommen noch nicht besprochen haben) und in der Schleife auch schreiben:

```
print $line if $line =~ /head/;
```

Sie sparen so die Klammern. Schließlich können sie `$line` auch noch weglassen. Perl denkt sich, dass Sie so was wollen ...

```
while (<>) { print if /head/; }
```

Toll, nicht?

Lösung zu Übung 3

Muster	Bedeutung	Beispieltext	passt auf
`.`	ein beliebiges Zeichen, also das erste Zeichen, das auftritt	`Hallo`	`H`
`.*`	beliebig viele beliebige Zeichen	`Hallo`	`Hallo`
`^.`	Auch das erste Zeichen	`Hallo`	`H`
`.$`	das letzte Zeichen	`Hallo`	`o`
`\(.*\)`	Zeichen in runden Klammern	`(Das sind (viele) Klammern)`	`(Das sind (viele) Klammern)`
`[ \t\n]`	Whitespaces (Leerzeichen, Tabs)	`Das sind 4 Leerzeichen`	(man sieht nichts: ein Leerzeichen)
`[a-z]`	Kleinbuchstaben	`Hallo`	`allo`
`[0-9]{5}`	5 Ziffern	`b1234567`	`12345`
`a[a-zA-Z]`	Ein kleines a und der Buchstabe danach	`Hallo`	`al`
`.*H`	Beliebige Zeichen und dann ein H	`Hallo`	`H`

Tabelle 3.2: Reguläre Ausdrücke

Lösung zu Übung 4

```
my $anz=0;
while (my $zeile = <>) {
    if ($zeile =~ m/table/) {
        ++$anz;
    }
}
```

Sie können auch mit der »Allerweltsvariablen« `$_` arbeiten an Stelle von `$zeile`; Wenn Sie außerdem noch das `if` umdrehen, wird die Sache viel kürzer, weil Sie dann die Klammern weglassen können:

```
while(<>) { ++$anz if /table/; }
```

Lösung zu Übung 5

```
while (my $zeile = <>) {
    $zeile =~ s/(\w+)/($1)/;
    print $zeile;
}
```

Beachten Sie die völlig unterschiedliche Bedeutung der runden Klammern im regulären Ausdruck und im Ersetzungstext.

Lösung zu Übung 6

```
use strict;
while (my $line = <>) {
    my @worte = split / /, $line;
    print $worte[0];
}
```

Das geht auch wieder kürzer (einfacher?):

```
while (<>) { print ((split / /)[0]); }
```

Lösung zu Übung 7

```
use strict;
while (my $line = <>) {
    if ( $line =~ /Statistik/) {
         my @worte = split / /, $line;
         print $worte[0];
    }
}
```

Das geht auch wieder kürzer (einfacher?):

```
while (<>) {
    print (split / /)[0] if (/Statistik/);
}
```

Hier wird das nachgestellte `if` benutzt, um Klammern zu sparen.

Lösung zu Übung 8

```
use strict;
while (my $zeile = <>) {
    print $zeile if (split /\s+/,$zeile) > 0;
}
```

Lösung zu Übung 9

```
use strict;
$\='^$';
while (my $zeile=<>) {
    print join '+', split /\n/, $zeile;
    print "\n";
}
```

Lösung zu Übung 10

```
use strict;
while (my $zeile=<>) {
    if ($zeile =~ /DRINGEND/) {
            my @f = split /\s+/, $zeile;
            print "$f[1] $f[2]\n";
    }
}
```

Lösung zu Übung 11

```
use strict;
my $anzahl=0;
while (my $zeile=<>) {
    if ($zeile =~ /DRINGEND/) {
        ++$anzahl;
    }
}
print "Das Wort DRINGEND kam $anzahl mal vor\n";
```

Lösung zu Übung 12

```
use strict;
my $total=0;
while (my $zeile=<>) {
    if ($zeile =~ /DRINGEND/) {
            my @f = split /\s+/, $zeile;
            $total += $f[1];
    }
}
print "Summe ist: $total";
```

Lösung zu Übung 13

```
@ARGV="test.dat";
$^I = ".bak";
while(my $zeile = <>) {
    $zeile =~ s/ä/&auml;/g;
    $zeile =~ s/ö/&ouml;/g;
    $zeile =~ s/ü/&uuml;/g;
    $zeile =~ s/Ä/&Auml;/g;
    $zeile =~ s/Ö/&Ouml;/g;
    $zeile =~ s/Ü/&Uuml;/g;
    print $zeile;
}
```

Lösung zu Übung 14

```
use strict;
while (my $zeile=<>) {
    if (length($zeile) > 20) {
        print $zeile;
    }
}
```

Lösung zu Übung 15

```
use strict;
while (my $zeile=<>) {
    if ($zeile =~ m/^Name/ and split(/\s+/,$zeile) == 7) {
        print $zeile;
    }
}
```

Lösung zu Übung 16

```
use strict;
while (my $zeile=<>) {
    print reverse join ' ', split / /, $zeile;
    print "\n";
}
```

Lösung zu Übung 17

```
use strict;
my @zeilen = <>;
print reverse @zeilen;
```

Lösung zu Übung 18

Hängen Sie hinter den Aufruf von Perl ein Größer-Zeichen > und einen Dateinamen.

Sie sollen also so etwas eingeben:

```
perl -w nummer.pl > Meine_nummerierte_datei.txt
```

Als Ergebnis bekommen Sie dann eine Textdatei mit nummerierten Zeilen.