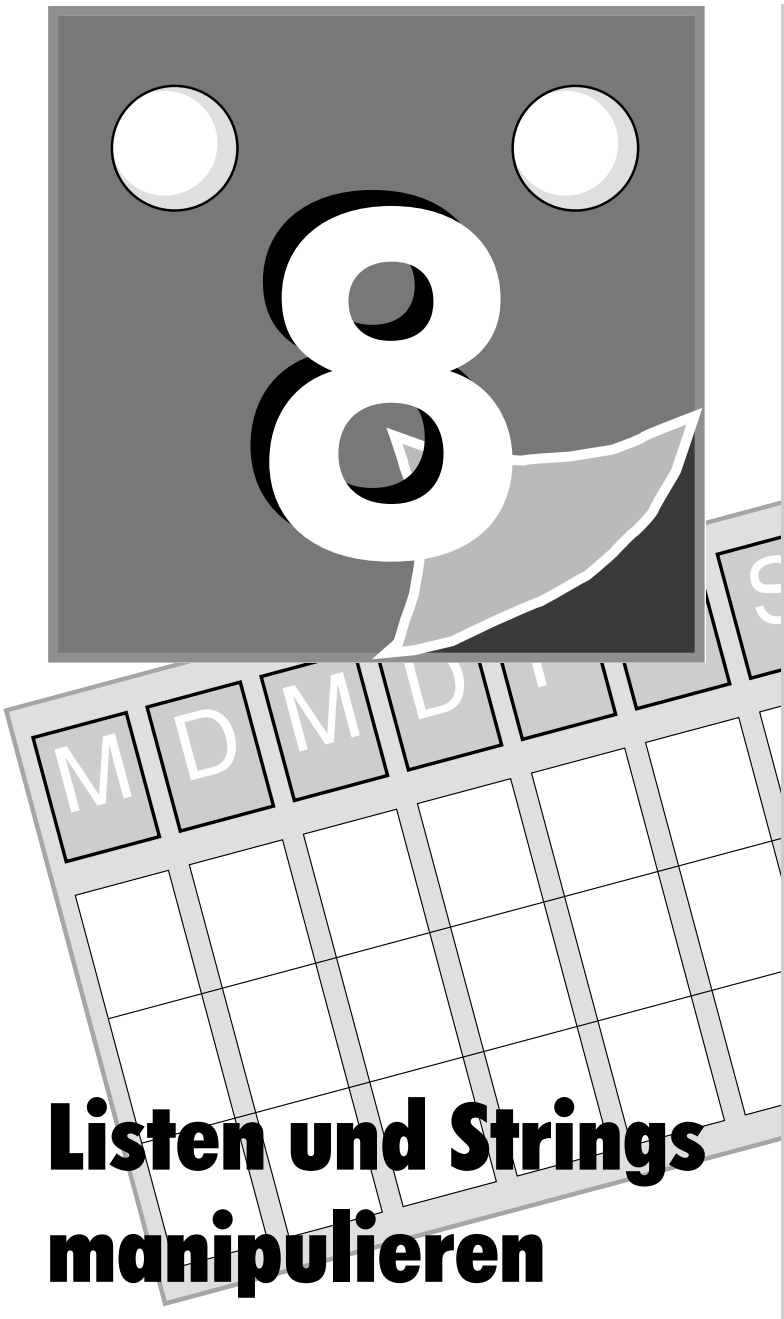

Listen und Strings manipulieren

Woche 2

Wir beginnen die zweite Woche mit der Verfeinerung Ihres Wissens über Listen, Arrays, Hashes und Strings – und wie Sie die darin enthaltenen Daten manipulieren können. Insbesondere werde ich Ihnen heute etliche Perl-Funktionen vorstellen, die es sich lohnt, im Repertoire zu haben:

Die Themen von heute sind:

- Array- und Hash-Segmente (*Slices*)
- Listen sortieren
- Listen durchsuchen
- Listenelemente hinzufügen oder entfernen
- Listenelemente verändern
- Verschiedene Stringmanipulationen: `reverse`, Substrings finden und extrahieren

Array- und Hash-Segmente

An den Tagen 4 und 5 haben Sie gelernt, wie Sie mit Arrays und Hashes arbeiten können. Perl kennt aber auch Mechanismen zur Manipulation von Teilmengen von Arrays und Hashes. Eine solche Teilmenge eines Arrays oder Hash, die selbst wieder ein Array oder Hash darstellt, heißt Segment oder *Slice* (Scheibe, Schnitte).

Der Zugriff auf ein Array-Segment ist dem auf ein einzelnes Element sehr ähnlich, auch auf ein *Slice* beziehen Sie sich mit eckigen Klammern. Es gibt jedoch zwei signifikante Unterschiede:

```
@array = (1,2,3,4,5);
$ein_wert = $array[0];   # $ein_wert ist 1
@slice = @array[0,1,2];   # @slice ist (1,2,3)
```

Sehen Sie die Unterschiede? Bei der Elementzugriffssyntax `$array[0]` nutzen Sie das Präfix `$` und eine Indexnummer in rechteckigen Klammern für den Zugriff auf ein Element in der Arrayvariablen. Das Ergebnis speichern Sie in der Skalarvariablen `$ein_wert`. In der Segmentschreibweise setzen Sie ein `@` an den Anfang des Arrayvariablennamens und eine Liste von Indizes in die Klammern. Das Ergebnis ist im Beispiel eine dreielementige Liste, die Sie im Array `@slice` speichern.

Sie können in die Klammern für ein Segment jeden beliebigen Ausdruck stellen, solange er nur zu einer Liste von gültigen Indizes evaluiert (das heißt: ausgewertet) wird. Der Bereichsoperator zum Beispiel eignet sich besonders gut für Segmente aus aufeinanderfolgenden Elementen:

```
@monate = qw (januar, februar, maerz, april, mai, juni, juli,
              august, september, oktober, november, dezember);
@viertes_quartal = @monate[9..11];
```

Beachten Sie, dass Sie nicht aus Versehen ein Array-Segment mit einem einzigen Index verwenden, wenn Sie eigentlich nur auf ein einzelnes Element zugreifen wollen:

```
$ein_wert = @array[5];
```

Diese Schreibweise ist ein häufiger Fehler. Sie ziehen hier nämlich eine einelementige Liste aus dem Array `@array` und versuchen sie dann in skalarem Kontext auszuwerten. Gegebenenfalls machen eingeschaltete Warnungen Sie darauf aufmerksam.

Hash-Segmente sind den Array-Segmenten sehr ähnlich, erfordern aber eine andere Syntax.

```
%hashslice = @hash{'dies','das','anderes'};
```

Die Schlüssel, die Sie extrahieren wollen, werden in den geschweiften Klassen aufgelistet, wie einzelne Hash-Schlüssel, aber ein Array-Variablensymbol wird vorangestellt. Der Grund ist, dass das Ergebnis eine Liste von Schlüsseln und Werten ist (in der korrekten Reihenfolge: Schlüssel, Wert, Schlüssel, Wert und so weiter). Durch Zuweisung an eine Hash-Variable kann man dieses Segment wieder in ein Hash verwandeln. Was Sie auf keinen Fall schreiben sollten, ist:

```
%hashslice = %hash{'dies','das','anderes'};
```

Diese Zeile würde (selbst ohne aktive Warnungen) zu einer Fehlermeldung führen. Auch Hash-Segmente sind Arrays. Allgemein gilt: Um aus einem Array oder Hash einen Skalar zu extrahieren, verwenden Sie die Skalarschreibweise (`$`), um ein Segment zu extrahieren, verwenden Sie die Arrayschreibweise (`@`).

Listen sortieren

Sie haben schon gesehen, wie man mit der Funktion `sort` eine Liste sortiert, zum Beispiel die Schlüssel in einem Hash:

```
@keys = sort keys %hash
```

Standardmäßig sortiert `sort` eine Liste in ASCII-Reihenfolge. Um eine Liste aus Zahlen in numerischer Reihenfolge zu sortieren, müssen Sie zwischen `sort` und der zu sortierenden Liste einen zusätzlichen Ausdruck einfügen:

```
@keys = sort { $a <=> $b } keys %hash;
```

Was macht dieser Extraausdruck? Wofür stehen `$a` und `$b`? In den vorangehenden Kapiteln habe ich Ihnen gesagt, dass Sie diese Zeile einfach auswendig lernen sollten. Jetzt aber erkläre ich, was das alles bedeutet.

Der Teil in den geschweiften Klammern legt fest, wie `sort` die Liste sortieren soll. Genauer gesagt vergleicht er zwei Elemente und teilt `sort` das Ergebnis mit: –1, wenn das erste Element kleiner ist als das zweite, 0, wenn beide Elemente gleich sind, und +1, wenn das erste größer ist als das zweite ist.

Diesen Vergleich können in Perl zwei Operatoren erledigen: <=> und `cmp`. Warum zwei Operatoren? Aus dem gleichen Grund, aus dem es zwei Sätze von Gleichheits- und Vergleichsoperatoren gibt: einen für Strings (`cmp`) und einen für Zahlen (<=>).

Der Operator `cmp` arbeitet auf Strings. Er gibt –1, 0 oder 1 zurück, je nachdem, ob der erste Operand (in ASCII-Hinsicht) kleiner als, gleich oder größer als der zweite ist. Wenn Sie Zeilen schinden wollten, könnten Sie `cmp` auch auf folgende Art schreiben, in einem »warum einfach, wenn's auch kompliziert geht«-Wettbewerb hätten Sie damit gute Chancen:

```
$ergebnis = '';
if ($a lt $b ) { $ergebnis = -1; }
elsif ($a gt $b { $ergebnis = 1; }
else { $ergebnis = 0; }
```

Der Operator <=>, manchmal wegen seines Aussehens auch Raumschiffoperator (*Spaceship-Operator*) genannt, macht genau das gleiche, nur eben mit Zahlen.

Doch was sind `$a` und `$b`? Es sind temporäre Variablen von `sort` – Sie müssen sie nicht deklarieren, und sie verschwinden von selbst, sobald die Sortierung erledigt ist. Beim Sortieren der Liste enthalten `$a` und `$b` jeweils die zwei Elemente, die gerade miteinander verglichen werden.

Weil `$a` und `$b` spezielle Variablen der `sort`-Routine sind und auf zwei Elemente in der Liste verweisen, seien Sie vorsichtig mit eventuellen anderen Variablen namens `$a` und `$b`. Wenn Sie `$a` oder `$b` innerhalb Ihrer Sortierungsroutine verändern, kann das zu unerwünschten Ergebnissen führen.

Solange Sie nichts anderes festlegen, verwendet `sort` den `cmp`-Operator für den Wertevergleich. Mit `{ $a <=> $b }` werden die Werte numerisch verglichen und sortiert.

Im übrigen wird standardmäßig aufsteigend sortiert. Wenn Sie absteigend sortieren möchten, vertauschen Sie einfach `$a` und `$b`:

```
@keys = sort { $b <=> $a } keys %hash; #groesste Keys zuerst
```

Mit einer `sort`-Routine und einer `foreach`-Schleife können Sie einen Hash nach Schlüsseln sortiert ausgeben. Etwas komplizierter ist es, die Ausgabe nach den Hash-

Werten zu sortieren. Wenn Sie lediglich die Werte brauchen, erstellen Sie mit der Funktion `values` eine Liste dieser Werte und sortieren sie dann. Auf diese Weise verlieren Sie jedoch den Zugriff auf die ursprünglichen Schlüssel. Es gibt keine Möglichkeit mehr, einen Wert wieder seinem Schlüssel zuzuordnen. Was tun? Sie können die Schlüssel in Wertreihenfolge sortieren und dann beides ausgeben, die Schlüssel und ihre Werte:

```
foreach $key (sort {$zeugs{$a} cmp $zeugs{$b}} keys %zeugs) {
   print "$key, $zeugs{$key}\n";
}
```

Hier sortieren wir nicht die Schlüssel (wir vergleichen nicht einfach `$a` und `$b`), sondern die mit den Schlüsseln verbundenen Werte (`$zeugs{$a}` und `$zeugs{$b}`). Das Ergebnis ist eine nach den Werten sortierte Liste der Schlüssel im Hash, über die wir dann mit `foreach` iterieren können.

Suchen

Listen sortieren ist einfach – denn es gibt eine Funktion dafür. Eine Liste hingegen nach einem bestimmten Element oder einem Teil eines Elements zu durchsuchen, ist nicht ganz so einfach, denn – TMTOWTDI[1] – Sie haben definitiv mehr als eine Möglichkeit. Zum Beispiel könnten Sie eine Liste von Strings mit `foreach` durchlaufen und jedes Element mit dem gesuchten String vergleichen, etwa so:

```
chomp($suchwort = <STDIN>);  # wonach wir suchen
foreach $el (@strings) {
   if ($suchwort eq $el) {
      $gefunden = 1;
   }
}
```

Wenn Sie nicht nach ganzen Listenelementen, sondern in den Listenelementen nach einem Teilstring suchen, nutzt Ihnen der `eq`-Operator allerdings herzlich wenig. Hilfreicher ist hier eine der Stringfunktionen, die ich Ihnen am Tag 3 kurz vorgestellt habe – die Funktion `index` durchsucht einen String nach einem Teilstring und gibt die Position zurück, an der der Teilstring zuerst im String vorkommt. Findet sie den Teilstring nicht, liefert sie –1 (Sie erinnern sich, dass Stringpositionen, wie Arrays, bei 0 beginnen):

1. There's more than one way to do it.

```
foreach $el (@strings) {
   if ((index $el, $suchwort) >= 0) {   # -1 heisst nicht gefunden
      $gefunden = 1;
   }
}
```

Effizienter und noch leistungsfähiger ist die Suche nach Teilstrings mit Hilfe von Mustern. Pattern Matching, das Arbeiten mit Mustern, werden wir aber erst morgen kennenlernen; halten Sie sich daher nicht allzusehr mit der folgenden Syntax auf.

```
foreach $el (@strings) {
   if ($el =- /$suchwort/) {
      $gefunden = 1;
   }
}
```

Es geht sogar noch kürzer. Die Perl-Funktion `grep` (benannt nach dem gleichnamigen Unix-Tool) dient insbesondere dem Durchsuchen von Listen. Wie viele andere Perl-Features auch benimmt `grep` sich in einem Listenkontext anders als in skalarem Kontext.

Sie übergeben der Funktion `grep` ein Suchkriterium (das kann ein Block oder ein Ausdruck sein) und eine Liste. Im Listenkontext gibt `grep` eine neue Liste der Elemente zurück, für die das Suchkriterium *wahr* ergab. Hier zum Beispiel holen wir uns alle Elemente größer 100 aus der Liste `@zahlen`:

```
@hohe = grep { $_ > 100 } @zahlen;
```

Beachten Sie unsere Freundin, die `$_`-Variable. Die Funktion `grep` nimmt jeweils ein Listenelement, weist es `$_` zu und wertet dann den Ausdruck in den geschweiften Klammern in Booleschem Kontext aus. Ist das Ergebnis *wahr*, erfüllt das jeweilige Element unser Suchkriterium und wird der Ergebnisliste hinzugefügt. Ist das Ergebnis *falsch,* geht `grep` zum nächsten Element der Liste `@zahlen` – und macht so weiter, bis sie zum Ende der Liste kommt.

Sie können als Suchkriterium jeden beliebigen Ausdruck oder Block verwenden (er wird dann in Booleschem Kontext ausgewertet), nur sollte `grep` damit etwas anfangen können (nämlich anhand dessen die neue Liste erstellen). Außerdem müssen Sie nach einfachen Ausdrücken ein Komma setzen; nach Blöcken ist das Komma nicht unbedingt notwendig

`Grep` arbeitet auch mit regulären Ausdrücken. In dieses Thema steigen wir zwar erst morgen ein, aber ich gebe Ihnen, sozusagen als kleinen Vorgeschmack, hier schon mal ein Beispiel:

```
@ixe = grep /x/, @worte;
```

Gesucht werden die Zeichen zwischen den Schrägstrichen (hier nur der Buchstabe x). In diesem Beispiel speichert grep alle Elemente des Arrays @worte, die den Buchstaben x enthalten, in das Array @ixe. Sie können zwischen die Schrägstriche des Patterns (des Suchmusters) auch eine Variable setzen:

```
print "Suchen nach? ";
chomp($suchwort = <STDIN>);
@gefunden = grep /$suchwort/, @worte;
```

Das Pattern kann beliebig viele Zeichen enthalten; grep wird genau diese Zeichenfolge in jedem Listenelement suchen. So findet zum Beispiel /der/ alle Elemente, in denen »der« enthalten ist (also »der«, »deren« und »oder«, nicht aber »Der« oder »Erde«). Wie Sie mit gewissen Sonderzeichen noch ausgeklügeltere Suchkriterien definieren, besprechen wir morgen.

In skalarem Kontext verhält sich grep ähnlich wie in einem Listenkontext, nur dass es keine Liste aller gefundenen Elemente zurückgibt, sondern die Zahl der Treffer (0 bedeutet »nichts gefunden«).

Man verwendet die grep-Funktion meistens mit Listen oder Arrays, aber nichts hindert Sie daran, sie auch auf Hashes anzuwenden, solange Sie nur Ihr Suchkriterium richtig formulieren. Die folgende Zeile findet zum Beispiel mit Hilfe von grep alle Hash-Schlüssel, bei denen entweder der Schlüssel oder der zugehörige Wert größer 100 ist:

```
@hohe_keys = grep { $_ > 100 or $numhash{$_} > 100 } keys %numhash;
```

Ein Beispiel: Mehr Namen

Am Tag 5 hatten wir ein einfaches Beispiel, das Namen (Vor- und Nachnamen) eingelesen, gesplittet und nach Nachnamen aufgeschlüsselt in einem Hash gespeichert hat. Am Tag 6 haben wir die Namen mit dem Zeileneingabeoperator <> aus einer externen Datei in ein Hash gelesen. Heute wollen wir das Namensskript weiter ausbauen und fügen eine große while-Schleife hinzu, die dem Benutzer vier Optionen anbietet:

- Namensliste nach Nachnamen sortieren und ausgeben
- Namensliste nach Vornamen sortieren und ausgeben
- Vor- oder Nachname suchen (exakte Übereinstimmung)
- Programm beenden

Wenn Sie eine der ersten drei Optionen auswählen, führt das Programm den entsprechenden Befehl aus und zeigt Ihnen danach erneut das Menü an, so dass Sie eine andere Option wählen können. Nur die vierte Option beendet das Programm. Hier ein Beispiel, wie diese neue Version, *mehrnamen.pl,* auf dem Bildschirm aussehen könnte:

```
% mehrnamen.pl namen.txt
1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen (exakte Suche)
4. Programm beenden

Geben Sie eine Zahl ein: 1
Burroughs, William S.
Salinger, J. D.
Sartre, Jean Paul
Schiller, Friedrich
Shakespeare, William
Vonnegut, Kurt

1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen
4. Beenden

Geben Sie eine Zahl ein: 2

Friedrich Schiller
J. D.  Salinger
Jean Paul  Sartre
Kurt Vonnegut
William Shakespeare
William S.  Burroughs
1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen
4. Beenden

Geben Sie eine Zahl ein: 3
Wonach suchen? Will
gefundene Namen:
   William S.  Burroughs
   William Shakespeare

1. Namensliste nach Nachnamen sortieren
2. Namensliste nach Vornamen sortieren
3. Namen suchen
```

```
4. Beenden

Geben Sie eine Zahl ein: 4
%
```

Listing 8.1 zeigt den Code für unser und Such- und Sortierskript:

Listing 8.1: mehrnamen.pl

```
1:  #!/usr/bin/perl -w
2:
3:  %names = ();                    # Hash mit Namen
4:  @raw = ();                      # die einzelnen Wörter
5:  $fn = "";                       # Vorname
6:  $exit = 1;                      # Programm verlassen?
7:  $in = '';                       # temp Input
8:  @keys = ();                     # temp Treffer-Keys
9:  @n = ();                        # temp Name
10: $search = '';                   # wonach gesucht wird
11:
12: while (<>) {
13:     chomp;
14:     @raw = split(" ", $_);
15:     if ($#raw == 1) {  # Normalfall
16:         $names{$raw[1]} = $raw[0];
17:     } else {  # erstelle den Vornamen
18:         $fn = "";
19:         for ($i = 0; $i < $#raw; $i++) {
20:             $fn .= $raw[$i] . " ";
21:         }
22:         $names{$raw[$#raw]} = $fn;
23:     }
24: }
25:
26: while ($exit) {
27:
28:     print "\n1. Namensliste nach Nachnamen sortieren\n";
29:     print "2. Namensliste nach Vornamen sortieren\n";
30:     print "3. Namen suchen\n";
31:     print "4. Beenden\n\n";
32:     print "Geben Sie eine Zahl ein: ";
33:
34:     chomp($in = <STDIN>);
35:
36:     if ($in eq '1') {       # nach Nachnamen sortieren und ausgeben
37:
```

```
38:         foreach $name (sort keys %names) {
39:             print "$name, $names{$name}\n";
40:         }
41:
42:     } elsif ($in eq '2') {   # nach Vornamen sortieren und ausgeben
43:
44:         @keys = sort { $names{$a} cmp $names{$b} } keys %names;
45:         foreach $name (@keys) {
46:             print "$names{$name} $name\n";
47:         }
48:
49:     } elsif ($in eq '3') {      # Namen finden (1 oder mehr)
50:
51:         print "Wonach suchen? ";
52:         chomp($search = <STDIN>);
53:
54:         @keys = ();  # @keys fuer Search zuruecksetzen
55:         while (@n = each %names) {
56:             if (grep /$search/, @n) {
57:                 $keys[++$#keys] = $n[0];
58:             }
59:         }
60:         if (@keys) {
61:             print "gefundene Namen: \n";
62:             foreach $name (sort @keys) {
63:                 print "   $names{$name} $name\n";
64:             }
65:         } else {
66:             print "Nichts gefunden.\n";
67:         }
68:
69:    } elsif ($in eq '4') {      # Ende
70:       $exit = 0;
71:    } else {
72:       print "Eingabefehler.  1 bis 4 bitte.\n";
73:    }
74: }
```

Den Rahmen dieses Skripts bildet eine große `while`-Schleife, die sich um die Optionen 1 bis 4 kümmert. Nachdem sie die Auswahlmöglichkeiten ausgegeben und um die Eingabe der entsprechenden Zahl gebeten hat, enthält die `while`-Schleife ein paar Überprüfungen der Eingabe. Wenn die Eingabe nicht 1, 2 , 3 oder 4 war, übernimmt das letzte `else` in Zeile 71 und beginnt von vorne (da wir auf diese Zahlen mit einem Stringvergleich überprüfen, würde ein versehentlicher Buchstabe keine Warnungen produzieren).

Viel interessanter ist, was bei jeder einzelnen Option passiert. Die beiden Sortieroptionen (Option 1 und 2 in Zeile 36 bzw. 42) verwenden jeweils eine unterschiedliche Form der `sort`-Funktion, die Sie in dieser Lektion bereits kennengelernt haben. Die dritte Option, die in den Namen nach dem eingegebenen String sucht, arbeitet mit `grep`.

Betrachten wir zuerst die beiden Sortieroptionen. Unser Namens-Hash ist nach Nachnamen aufgeschlüsselt, also ist das Sortieren nach Nachnamen ganz einfach. Tatsächlich ist die `foreach`-Schleife in Zeile 38 bis 40 genau dieselbe wie in den bisherigen Versionen dieses Beispiels.

Den Hash nach Vornamen zu sortieren (die zweite Option) ist da schon schwieriger. Doch im Abschnitt »Sortieren« habe ich Ihnen ja gerade gezeigt, wie man das macht – die dort gezeigte Technik können Sie hier in Zeile 44 bis 47 anwenden. Sie erstellen mit einer `sort`-Routine, die nach Werten sortiert, eine temporäre Liste der Schlüssel, durchlaufen diese mit `foreach` und geben die Namen in der richtigen Reihenfolge aus.

Womit wir bei der Suche wären. In Zeile 51 und 52 fragen wir, wonach gesucht werden soll. Diesen Suchstring speichern wir in der Variablen `$search`.

Auf den ersten Blick denken Sie vielleicht, diese Suche wäre ganz einfach – nur `grep` und das Pattern `/$search/`. Der Haken ist aber, dass wir in einem Hash sowohl die Schlüssel als auch die Werte durchsuchen müssen.

Wollten wir nur in den Nachnamen suchen, könnten wir einfach mit Hilfe der `keys`-Funktion die Schlüssel durchsuchen, etwa so:

```
@treffer = grep /$search/, keys %names;
```

Um an die Werte zu kommen, könnten wir mit einer `foreach`-Schleife die Schlüssel durchlaufen und jeden Schlüssel und seinen Wert nach dem gefragten String durchsuchen. Dieser Ansatz wäre nicht falsch. Aber in diesem besonderen Beispiel wollte ich `grep` verwenden, deswegen habe ich einen vielleicht ungewöhnlich anmutenden Weg gewählt (ich nenne es lieber einen kreativen Weg): Ich nehme die `each`-Funktion, die, wie Sie am Tag 5 gelernt haben, mir eine Liste aus Schlüssel/Wert-Paaren liefert. Dann verwende ich `grep` mit dieser Liste und speichere die Schlüssel für später.

Das mache ich in Zeile 55 bis 59. Schauen wir uns die Zeilen noch mal genauer an:

```
55:         while (@n = each %names) {
56:             if (grep /$search/, @n) {
57:                 $keys[++$#keys] = $n[0];
58:             }
59:         }
```

Von `each` erhalten wir eine zweielementige Liste mit einem Schlüssel/Wert-Paar. Rufen wir die `each`-Funktion mehrmals auf, arbeitet sie sich durch alle Schlüssel und Wer-

te im Hash. Die `while`-Schleife in Zeile 55 wird so oft durchlaufen, wie Schlüssel/Wert-Paare im Hash sind, wobei jedes Paar unserem temporären Namensarray `@n` zugewiesen wird. Wenn kein Paar mehr übrig ist, gibt `each` die leere Liste `()` zurück, `@n` evaluiert zu *falsch*, und die `while`-Schleife stoppt.

Innerhalb der `while`-Schleife überprüfen wir mit einer `if`-Bedingung, `grep` und dem Pattern `/$search/`, ob der Suchstring in einem der beiden Elemente von `@n` vorkommt. Wir verwenden `grep` hier in skalarem Kontext, deswegen liefert `grep` uns die Anzahl der Treffer. Wenn `grep` nichts finden konnte, gibt es 0 zurück, und wir fahren fort mit dem nächsten `while`-Durchlauf. Wenn es aber etwas gefunden hat, liefert es eine Zahl ungleich Null, unsere Bedingung ist erfüllt, und wir machen mit Zeile 57 weiter.

In Zeile 57 holen wir uns den in `$n[0]` gespeicherten Schlüssel und hängen ihn ans Ende des Arrays `@keys` an (wenn wir den Schlüssel haben, haben wir auch Zugriff auf den Wert, um den brauchen wir uns also keine Sorgen zu machen). Sie erinnern sich, dass `$#keys` uns den höchsten Index im Array liefert, `++$#keys` sich also auf die Position nach der letzten Position bezieht. Beachten Sie die Präfixschreibweise – so können wir den Index inkrementieren, bevor wir dem Array an dieser Stelle etwas zuweisen. Das machen wir erst danach: Wir fügen dem Array als neues letztes Element unseren Schlüssel hinzu.

Puh! Diese Zeile ist ein Beispiel dafür, wieviel Information man in eine einzige Zeile stopfen kann. Zum Glück gibt es einen viel leichteren Weg, ein Element am Ende einer Liste anzufügen: die Funktion `push`, mit der wir uns später in dieser Lektion noch befassen.

Nachdem die Liste mit den Trefferschlüsseln erstellt ist, müssen wir sie nur noch ausgeben. In diesem Fall überprüfen wir erst, ob `@keys` überhaupt etwas enthält (Zeile 60), und wenn dem so ist, sortieren wir es und geben die Namen aus. Anderenfalls teilt eine kleine Meldung mit, dass nichts gefunden wurde (Zeile 66).

Listenelemente hinzufügen oder entfernen

Sie können Listen-, Array- und Hash-Elemente immer mit den Methoden hinzufügen und entfernen, die ich an den Tagen 4 und 5 beschrieben habe. Aber in manchen Situationen sind diese Standardwege zum Hinzufügen und Entfernen von Elementen etwas unbequem, schlecht zu lesen oder uneffizient. Hilfsbereit wie immer, stellt Perl Ihnen einige Funktionen zu Verfügung, die Ihnen die Arbeit mit Listen erleichtern werden. Zu diesen Funktionen gehören:

- `push` und `pop`: ein Element am Ende der Liste hinzufügen und entfernen
- `shift` und `unshift`: ein Element am Anfang der Liste hinzufügen und entfernen
- `splice`: ein Element irgendwo in der Liste hinzufügen oder entfernen

push und *pop*

Mit den Funktionen `push` und `pop` kann man Elemente am Ende einer Liste hinzufügen oder entfernen. Das heißt, sie betreffen die höchsten Indexpositionen dieser Liste. Wenn Sie jemals mit S*tacks* (Stapeln) gearbeitet haben und Ihnen die »*Last In First Out*«-Idee ein Begriff ist,[1] werden Sie `push` und `pop` kennen. Hatten Sie noch nie mit S*tacks* zu tun, stellen Sie sich einen Stapel Teller vor. Wenn Sie einen Teller auf diesen Stapel legen, ist das auch der Teller, den Sie als erstes wieder herunternehmen (Sie werden wahrscheinlich nicht irgendeinen aus der Mitte herausziehen). *Stack* ist englisch und heißt nichts anderes als Stapel – `push` entspricht »Teller rauf« und `pop` »Teller runter«. Ganz einfach. Was beim Tellerstapel oben ist, ist bei Arrays das Ende, das heißt `push` fügt ein Element am Ende des Arrays an, und `pop` nimmt dieses Element wieder weg.

Die Funktion `push` braucht zwei Argumente: eine Liste, die verändert, und eine Liste von Elementen, die angefügt werden soll. Die Originalliste wird an Ort und Stelle verändert und `push` gibt die neue Zahl der Elemente in der veränderten Liste zurück (sozusagen, wie viele Teller jetzt in dem Stapel sind). So haben wir zum Beispiel im vorigen Abschnitt diesen häßlichen Code gebraucht, um dem Array `@keys` einen Schlüssel hinzuzufügen:

```
$keys[++$#keys] = $n[0];
```

Genau das gleiche könnten wir mit `push` so schreiben:

```
push @keys, $n[0];
```

Weniger Zeichen und leichter zu verstehen. Beachten Sie, dass das zweite Argument hier ein Skalar ist (der dann in eine einelementige Liste konvertiert wird); es könnte auch eine Liste sein. So können Sie mehrere Listen ganz einfach zusammenfügen:

```
push @ergebnis_liste, @liste1;
```

Die Funktion `pop` macht das Gegenteil von `push`: Sie entfernt das letzte Element aus der Liste oder dem Array. Der `pop`-Funktion muss man nur ein Argument übergeben, nämlich die zu verändernde Liste. `Pop` entfernt das letzte Element und gibt dieses Element zurück.

1. Das *letzte* Element *rein* ist das *erste* Element *raus*.

```
$letztes = pop @array;
```

Wie bei `push` wird die Liste an Ort und Stelle verändert und hat nach dem `pop` ein Element weniger. Ein Weg, alle Elemente von einem Array in ein anderes zu verschieben, könnte zum Beispiel so aussehen:

```
while (@alt) {
   push @neu, pop @alt;
}
```

Beachten Sie, dass mit diesem Beispiel das neue Array genau die umgekehrte Reihenfolge hätte, denn das letzte Element im alten Array ist das erste, das ins neue Array verschoben wird. Zum Umkehren einer Liste eignet sich allerdings die Funktion `reverse` weit besser, und nach wie vor verschieben Sie eine Liste am einfachsten mit einer Zuweisung. Sie könnten obigen Code aber zum Beispiel verwenden, wenn Sie mit jedem Element noch etwas machen wollen, während sie es von einem Array in das andere schieben:

```
while (@alt) {
   push @neu, 2 * pop @alt;
}
```

shift und *unshift*

Wenn `push` und `pop` Elemente am Listenende hinzufügen oder entfernen, wäre es doch nett, auch Funktionen zu haben, die das gleiche am Anfang der Liste machen. Kein Problem! Das erledigen `shift` und `unshift` (von englisch *shift*: verschieben, wegrücken, ändern) sie schieben alle Elemente eine Position hoch oder runter (oder wie man sich das bei Arrays eher vorstellt, nach rechts oder links). Wie bei `push` und `pop` wird die Liste direkt geändert.

Die Funktion `shift` nimmt ein Argument entgegen, und zwar eine Liste. Sie entfernt das erste Element dieser Liste und schiebt alle anderen Elemente eine Position nach unten (links). Zurück gibt sie das entfernte Element. Auch das folgende Beispiel verschiebt Elemente von einem Array in ein anderes, jedoch wird die Reihenfolge hier nicht umgekehrt:

```
while (@alt) {
   push @neu, shift @alt;
}
```

Die Funktion `unshift` ist das Listenanfangspendant zu `push`. Man übergibt Ihr zwei Argumente: die zu ändernde Liste und die anzufügende Liste. Die geänderte Liste enthält dann die zugefügten Elemente, gefolgt von den alten Listenelementen, die soweit

nach oben (oder rechts) verschoben wurden, wie Elemente hinzugefügt worden sind. Zurück gibt `unshift` die Anzahl der Elemente in der neuen Liste:

```
$anzahl_user = unshift @user, @neue_user;
```

splice

`push` und `pop` verändern Elemente am Ende einer Liste, `shift` und `unshift` Elemente am Anfang. Die Funktion `splice` (spleissen, verbinden) ist ein Allzweckmittel, wenn Sie irgendwo in der Liste Elemente hinzufügen, entfernen oder ersetzen wollen. Sie übergeben `splice` vier Argumente:

- das zu ändernde Array
- den Offset, die Position im Array, ab der Elemente hinzugefügt oder entfernt werden
- die Anzahl zu entfernender oder zu ersetzender Elemente. Wenn dieses Argument nicht enthalten ist, ändert `splice` jedes Element vom Offset an
- die Listenelemente, die dem Array hinzugefügt werden, wenn vorhanden

Der Offset ist genaugenommen der Abstand zwischen dem Anfang eines Arrays oder Strings und einer bestimmten Position. Er gibt an, wie viele Elemente oder Zeichen Sie nach rechts gehen müssen, um von der ersten zur gewünschten Position zu gelangen – mit einem Offset 0 bleiben Sie am Anfang. Da Arrays und Strings ebenfalls bei 0 beginnen, bedeutet Offset letztlich nichts anderes als Index der Zielposition und umgekehrt.[1]

Betrachten wir zuerst, wie man mit `splice` Elemente aus einem Array entfernt. Bei jedem der folgenden Beispiele gehen wir von einer Liste mit zehn Elementen, den Zahlen von 0 bis 9, aus:

```
@zahlen = 0 .. 9;
```

Zum Entfernen der Elemente 5, 6 und 7 nimmt man als Offset 5 und als Länge 3. Ein Listenargument gibt es nicht, weil wir ja nichts hinzufügen:

```
splice(@zahlen, 5, 3);  # ergibt (0,1,2,3,4,8,9)
```

Um alle Elemente ab Position 5 (also bis zum Listenende) zu entfernen, lassen Sie einfach das Längenargument weg:

```
splice(@zahlen, 5);  # ergibt (0,1,2,3,4)
```

1. Dies alles gilt jedoch nur, solange Sie nicht den Wert der Spezialvariablen `$[` ändern und festlegen, dass die Indizes von nun an bei (7) anfangen (oder bei welchem Wert auch immer – tun Sie es nicht).

Wollen Sie Elemente ersetzen, übergeben Sie `splice` als viertes Argument eine Liste mit den einzufügenden Elementen. Hier zum Beispiel ersetzen Sie die Elemente 5, 6 und 7 durch die Elemente »fuenf«, »sechs« und »sieben«:

```
splice(@zahlen, 5, 3, qw(fuenf sechs sieben));
                        # ergibt (0,1,2,3,4,fuenf,sechs,sieben,8,9)
```

Ob Sie genauso viele Elemente entfernen, wie Sie hinzufügen, kümmert Perl nicht. Es fügt einfach die Elemente der übergebenen Liste an der durch den Offset festgelegten Position ein und rückt alle anderen Elemente entsprechend zurecht. Im folgenden Beispiel löscht `splice` die Elemente 5, 6 und 7 und setzt den String »nicht vorhanden« an ihre Stelle:

```
splice(@zahlen, 5, 3, "nicht vorhanden");
                          # ergibt (0,1,2,3,4,"nicht vorhanden",8,9)
```

Das Array hat dann nur noch acht Elemente.

Um dem Array Elemente hinzuzufügen, ohne welche zu entfernen, übergeben Sie als Länge 0. Hier zum Beispiel fügen wir nach Element 5 ein paar Zahlen ein:

```
splice(@zahlen, 5, 0, (5.1, 5.2, 5.3));
                     # ergibt  (0,1,2,3,4,5,5.1,5.2,5.3,6,7,8,9)
```

Wie die anderen Funktionen zur Manipulation von Listen verändert die Funktion `splice` die Liste direkt und gibt eine Liste der entfernten Elemente zurück. Diese Eigenschaft können Sie ausnutzen, um eine Liste in mehrere Teile zu zerlegen. Sagen wir zum Beispiel, Sie hätten eine Liste `@gesamtliste`, die Sie in zwei Listen `@liste1` und `@liste2` zerlegen möchten. Das erste Element in `@gesamtliste` ist die Anzahl der Elemente, die Sie in die `@liste1` verschieben wollen, alle übrigen Elemente sollen in `@liste2`. Sie könnten das mit den folgenden drei Zeilen lösen. Die erste Zeile entfernt ab Position 1 so viele Elemente aus `@gesamtliste`, wie das Element 0 festlegt, und weist sie `@liste1` zu. Die zweite entfernt das Element 0, und die dritte speichert die verbleibenden Elemente in `@liste2`:

```
@liste1 = splice(@gesamtliste, 1, $gesamtliste[0]);
shift @gesamtliste;
@liste2 = @gesamtliste;
```

Weitere Möglichkeiten zur Listenmanipulation

Warten Sie, das war noch nicht alles. Ich habe noch mehr Funktionen, die beim Manipulieren von Listen ganz nützlich sein können: `reverse`, `join` und `map`.

reverse

Die Funktion `reverse` nimmt jedes Element einer Liste und färbt es blau. Kleiner Scherz. In Wahrheit kehrt `reverse` die Reihenfolge der Listenelemente um:

```
@andersrum = reverse @liste;
```

Wie bereits gesagt können Sie die Reihenfolge statt mit `reverse` auch mit anderen Methoden wie `shift`, `unshift`, `push`, `pop` oder bloßem Umsortieren umkehren (aber versuchen Sie, Arrayelemente so wenig wie möglich hin- und herzuschieben, der Überblick geht schnell verloren).

Die `reverse`-Funktion arbeitet auch reibungslos mit Skalaren. In diesem Fall kehrt es die Reihenfolge der Zeichen im String um. Wir kommen gleich im Abschnitt »Strings manipulieren« darauf zurück.

join

Die Funktion `split` spaltet einen String in mehrere Listenelemente auf. Die Funktion `join` (verbinden, vereinigen) tut das Gegenteil: `join` fügt Listenelemente zu einem einzigen String zusammen, wobei Sie als Trennzeichen eine beliebige Zeichenfolge festlegen können. Wenn Sie zum Beispiel eine Zahlenfolge mit `split` in eine Liste zerlegt haben:

```
@liste = split(' ', "1 2 3 4 5 6 7 8 9 10";
```

dann könnten Sie sie mit `join` wieder zurück in einen String verwandeln, in dem die Zahlen durch Leerzeichen voneinander getrennt sind:

```
$string = join(' ',@liste);
```

Oder Sie könnten die Elemente durch Pluszeichen voneinander trennen:

```
$string = join('+',@liste);
```

Oder Sie trennen sie gar nicht:

```
$string = join('',@liste);
```

map

Sie wissen bereits, wie man mit `grep` die Elemente einer Liste, die ein bestimmtes Suchkriterium erfüllen, in einer anderen Liste speichert. Die Funktion `map` arbeitet ganz ähnlich, nur dass `map` nicht bestimme Elemente aus einer Liste auswählt, sondern

einen gegebenen Ausdruck oder Ausdrucksblock auf jedes Listenelement anwendet und alle Ergebnisse in einer neuen Liste sammelt.

Sagen wir zum Beispiel, Sie haben eine Liste aus Zahlen von 1 bis 10 und Sie möchten eine Liste mit den Quadraten all dieser Zahlen erstellen. Sie könnten dafür eine `foreach`-Schleife und `push` nehmen:

```
foreach $zahl (1 .. 10) {
   push @quadrate, ($zahl*$zahl);
}
```

Derselbe Vorgang würde mit `map` so aussehen:

```
@zahlen = (1..10);
@quadrate = map { $_**2 } @zahlen;
```

Wie `grep` schnappt sich `map` nacheinander die Elemente der Liste und weist diese nacheinander `$_` zu. Anders als `grep` wertet es den Ausdruck dann nicht in Booleschem, sondern in Listenkontext aus und speichert jedes Ergebnis in der neuen Liste – auch Null, auch Werte, die anders sind als das ursprüngliche Listenelement in `$_` und sogar mehrere Elemente gleichzeitig. Stellen Sie sich `map` als einen Filter für jedes Listenelement vor, wobei das Ergebnis des Filters sein kann, was immer Sie möchten.[1]

Der »Filter«-Teil von `map` kann entweder ein einzelner Ausdruck oder ein ganzer Block sein (nach einem einfachen Ausdruck müssen Sie ein Komma setzen, nach einem Block nicht unbedingt). Bei einem einfachen Ausdruck gibt `map` für jedes Listenelement das Ergebnis dieses Ausdrucks zurück. Bei einem Block liefert es das Ergebnis des zuletzt ausgewerteten Ausdrucks im Block – sorgen Sie also dafür, dass die letzte Evaluierung im Block das gewünschte Ergebnis hat.[2] Im folgenden Beispiel ersetzt `map` alle negativen Zahlen durch 0 und jede 5 durch die Zahlen 2 und 3. Ich habe das ganze etwas leichter lesbar formatiert. Weder bei `map` noch bei `grep` muss der gesamte Code in einer Zeile stehen:

```
@neue_zahlen = map {
               if ($_ < 0) {
                   0;
               } elsif ($_ == 5) {
                   (3,2);
               } else { $_; }
} @zahlen;
```

1. Denken Sie nicht nur an das *Heraus*filtern, sondern vor allem an die Transformation. In Perl könnten Sie aus einer Tasse Wasser fünf Tassen Cappuccino machen – versuchen Sie das mal mit Ihrem Kaffeefilter.

2. Das gilt übrigens für Blöcke im allgemeinen. Mehr dazu an Tag 11.

Strings manipulieren

In dieser Lektion ging es bisher vornehmlich um die Manipulation von Listen und Listeninhalten mittels verschiedener Perl-Funktionen. Perl hat aber auch einige sehr nützliche Funktionen zum Manipulieren von Strings (ein paar davon habe ich bereits am Tag 2 vorgestellt). In diesem Abschnitt wollen wir uns einige dieser Funktionen, `reverse`, `index`, `rindex` und `substr`, genauer ansehen.

Jede dieser Funktionen dient der Veränderung von Strings. In vielen Fällen sind andere Methoden wahrscheinlich geeigneter – der Punktoperator (`.`) zum Verketten, Variablenwerte und -interpolation zum Erstellen oder Patterns zum Suchen und Durchsuchen von Strings. Doch in manchen Fällen sind die folgenden Funktionen zumindest einfacher zu handhaben, insbesondere wenn Sie ähnliche Funktionen zur Stringmanipulation von anderen Sprachen her gewohnt sind.

reverse

Sie kennen die `reverse`-Funktion schon im Zusammenhang mit Listen. In einem Listenkontext kehrt sie die Reihenfolge der Elemente in der Liste um. In skalarem Kontext verhält `reverse` sich anders: Hier wird die Reihenfolge der Zeichen in einer Zahl oder einem String umgekehrt.

Aus dem String »NIE GRUB RAMSES MARBURG EIN« macht `reverse` also »NIE GRUBRAM SESMAR BURG EIN«. Beachten Sie, dass die Reihenfolge *aller* Zeichen umgekehrt wird und ein eventueller Zeilenvorschub am Ende eines Strings in seiner Umkehrung am Anfang steht. Wenn Sie diesen Effekt vermeiden wollen, verwenden Sie vorher `chomp`.

Der Unterschied zwischen `reverse` in Skalar- und Listenkontext kann manchmal etwas verwirren. Nehmen Sie zum Beispiel folgenden Code:

```
foreach $string (@liste) {
   push @andersrum, reverse $string;
}
```

Auf den ersten Blick sieht das Beispiel so aus, als nehme es alle Stringelemente aus dem Array `@liste`, kehre jedes einzelne um und schiebe das Ergebnis in das Array `@andersrum`. Wenn Sie den Code jedoch ausführen, enthält `@andersrum` exakt dasselbe wie `@liste`. Warum? Weil das zweite Argument der Funktion `push` eine Liste zu sein hat, wird `reverse` hier im Listen- statt im skalaren Kontext aufgerufen. Der String in `$string` wird folglich als eine einelementige Liste interpretiert, die auch rückwärts nur dieses eine Element enthält. Die Zeichen innerhalb dieses Elements bleiben unberührt. Dieses »Mißverständnis« kann die `scalar`-Funktion klären:

```
foreach $string (@liste) {
   push @andersrum, scalar (reverse $string);
}
```

index und *rindex*

Mit den Funktionen `index` und `rindex` suchen Sie in Strings nach Teilstrings. Wenn Sie ihnen zwei Strings übergeben (einen zu durchsuchenden und einen zu suchenden), liefern sie die Position des zweiten Strings im ersten zurück oder, wenn er nicht gefunden wurde, –1. Positionen beginnen auch in Strings bei 0.[1] Anders als bei Arrays bezeichnen sie aber die Stellen *zwischen* den Zeichen, das heißt Position 0 ist die Stelle unmittelbar vor dem ersten Zeichen des Strings. Sie können mit `index` oder `rindex` `grep`-ähnlichen Code wie diesen hier schreiben:

```
foreach $str (@liste) {
   if ((index $str, $suchwort) != -1) {
      push @treffer, $str;
}
```

Der Unterschied zwischen `index` und `rindex` sind der Startpunkt und die Richtung der Suche. Die Funktion `index` beginnt ihre Suche am Anfang des Strings und findet die Position, an der der gesuchte Teilstring das erste Mal auftaucht. `rindex` hingegen »rollt den String von hinten auf«: Es beginnt am Ende und findet die letzte Position des Teilstrings.

Beiden Funktionen können Sie ein optionales drittes Argument übergeben, das die Position festlegt, an der die Suche starten soll. Wenn Sie zum Beispiel mit `index` bereits etwas gefunden haben, können Sie `index` noch einmal aufrufen und da anfangen, wo Sie aufgehört haben.

substr

Die Funktion `substr` (von Substring) extrahiert einen Teilstring aus einem String und gibt ihn zurück. Sie rufen `substr` mit folgenden drei Argumenten auf:

- Dem String, auf den Sie sich beziehen
- Dem Offset, ab dem die Extraktion beginnen soll. Wenn Sie hier eine negative Zahl übergeben, zählt `substr` die Positionen vom Ende ab.
- Der Länge des zu extrahierenden Teilstrings. Wenn Sie keine Länge angeben, werden alle Zeichen vom Offset bis zum Ende des Strings extrahiert.

1. Wenn Sie nicht an `$[` herumgespielt haben.

Der Originalstring wird von `substr` nicht verändert.

Sie verstehen nur Bahnhof? Dann extrahieren wir die ersten vier Zeichen aus dem String »Bahnhof« und speichern sie in `$teilstring`:

```
$string = "bahnhof";
$teilstring= substr($string, 0, 4);  #$teilstring ist "bahn"
```

Oder die Zeichen 5 und 6:

```
$teilstring= substr($string, 4, 2);  #$teilstring ist "ho"
$teilstring= substr($string, -3, 2);  #$teilstring ist "ho"
```

Das Besondere an `substr` ist aber, dass Sie es auch auf die linke Seite einer Zuweisung stellen können. Dann extrahiert es keine Teilstrings, sondern macht geradezu das Gegenteil: Auf der linken Seite einer Zuweisung beziehen sich die Argumente (String, Offset, Länge des Teilstrings) nicht auf das Extrahieren, sondern das Ersetzen von Teilstrings. Der String auf der rechten Seite kann größer oder kleiner als der zu ersetzende sein, das ist Perl ganz egal:

```
substr($string, 4, 3) = "fahrkarten"; #$string ist "bahnfahrkarten"
substr($string, -1, 1) = "";          #$string ist "bahnfahrkarte"
substr($string, 0, 13) = "$teilstring "; #$string ist "ho "
```

Möchten Sie einen Teilstring überall in einem String suchen und ersetzen, könnten Sie zum Beispiel eine `while`-Schleife, `index` und `substr` verwenden:

```
$str = "Dies ist ein Test-String. Den wir veraendern wollen.";
$pos = 0;
$teilstr = "i";
$ersatz = "*";

while ($pos < (length $str) and $pos != -1) {
    $pos = index($str, $teilstr, $pos);

    if ($pos != -1 ) {
        substr($str,$pos,length $teilstr) = $ersatz;
        $pos++;
    }
}
```

Klammern Sie sich aber nicht an diesen Code: Für solche Operationen gibt es bessere und kürzere Wege. Insbesondere reguläre Ausdrücke komprimieren diese gesamte Schleife in einzige Zeile:

```
$str =~ s/$teilstr/$ersatz/g;
```

Vertiefung

In dieser Lektion habe ich Ihnen viele recht gebräuchliche Funktionen zur String- und Listenmanipulation vorgestellt. Von diesen wiederum habe ich nur die gebräuchlicheren Anwendungen beschrieben. Wenn Sie Interesse an den Details dieser (oder der in diesem Buch nicht beschriebenen) Funktionen haben, schauen Sie auch in Anhang A oder die *perlfunc*-Manpage.

Einige Eigenschaften der Funktionen habe ich vor allem deshalb nicht angesprochen, weil ich Sie dann noch öfter auf später vertrösten müßte, als ich es ohnehin schon tue. Zum Beispiel kann die Sortierroutine für die `sort`-Funktion (der Teil in den geschweiften Klammern) durch den Namen einer Subroutine ersetzt werden. So könnten Sie einmal eine ganz ausgefuchste `sort`-Routine schreiben und diese an verschiedenen Stellen immer wieder verwenden.

Die Funktionen `pop` und `shift` können auch ohne Argumente verwendet werden. Auf welche Liste sie sich dann beziehen, hängt davon ab, wo im Skript sie zum Einsatz kommen: Im Hauptkörper des Skripts verändern `pop` und `shift` ohne Argumente das `@ARGV`-Array (das die für das Skript gedachten Argumente enthält). Innerhalb einer Subroutine wirken `pop` und `shift` sich auf `@_` aus, einer Liste, in der die Argumente der Subroutine landen (und hier mein Verweis auf später: Mit Subroutinen befassen wir uns an Tag 11).

Zusammenfassung

Perl lernen ist mehr als nur die Syntax des Sprachkerns zu lernen. Die Perl-Funktionen tragen viel zur Funktionalität von Perl-Skripts bei. Oft gibt es bessere Wege als diese Funktionen, aber in bestimmten Fällen sind sie wiederum sehr nützlich. Das gilt für alle Funktionen, die Sie heute kennengelernt haben. Das meiste können Sie auch ganz anders machen, aber vielleicht nicht so schnell, effizient oder übersichtlich.

Wir haben heute verschiedene Funktionen zum Manipulieren von Listen und Strings besprochen und wie man sie zur Lösung einfacher Aufgaben einsetzt. Sie haben außerdem etwas über Array- und Hash-Segmente gelernt, mit denen Sie mehrere Elemente auf einmal aus einer Liste oder aus einem Hash herausziehen können.

Die neuen Funktionen waren heute:

- `sort`, sortiert Listen
- `grep`, extrahiert Listenelemente, auf die ein bestimmtes Kriterium zutrifft
- `push` und `pop`, entfernt Elemente vom Ende einer Liste bzw. fügt sie dort hinzu

- `shift` und `unshift`, entfernt Elemente vom Anfang einer Liste bzw. fügt sie dort hinzu
- `splice`, Hinzufügen, Entfernen oder Ersetzen von Elementen an jeder Position einer Liste
- `reverse`, kehrt die Reihenfolge der Elemente in der Liste oder in skalarem Kontext, der Zeichen im String, um
- `join`, das Gegenteil von `split`, fügt Listenelemente zu einem String zusammen
- `map`, führt auf jedem Listenelement eine oder mehrere Anweisungen aus und erstellt eine neue Liste mit den Ergebnissen
- `index` und `rindex`, finden die Position eines Strings in einem anderen String
- `substr`, zum Entfernen, Hinzufügen oder Ersetzen eines Strings durch einen anderen

Fragen und Antworten

F Die meisten heute beschriebenen Funktionen scheinen nur mit Listen reibungslos zu arbeiten. Kann ich sie auch mit Hashes einsetzen?

A *Hashes und Listen sind insoweit austauschbar, als ein Hash in seine einzelnen Komponenten aufgeschlüsselt wird, wenn man ihn als Liste verwendet. Die Elemente werden wieder zu Schlüssel-Wert-Paaren, wenn aus der Liste wieder ein Hash wird. Ganz technisch gesehen können Sie also viele der Funktionen auch mit Hashes benutzen. Allerdings kommt dabei vielleicht nicht gerade das heraus, was Sie sich davon versprechen. Zum Beispiel wird `pop` mit einem Hash diesen Hash in eine Liste umwandeln und dann das letzte Element dieser Liste entfernen (den letzten Key im Hash) – nur wissen Sie ja gar nicht, in welcher Reihenfolge die Keys im Hash stehen; deswegen ist das Ergebnis nicht vorherzusagen. Hashes manipulieren Sie wohl besser mit Hash-Segmenten oder der `delete`-Funktion, oder Sie wenden die Funktionen von heute auf Listen aus den Hash-Schlüsseln und -werten an.*

F Ich will mit `reverse` eine Liste aus Strings umkehren. Aber die Reihenfolge ändert sich nicht; es ist, als hätte ich die Funktion nie aufgerufen.

A *Sind Sie sicher, dass Sie `reverse` auf einen String anwenden und nicht auf eine einelementige Liste? Eine Liste mit einem einzigen Element ist auch eine Liste, deren Reihenfolge `reverse` Ihnen artig umkehrt – was den String aber nicht verändert. Achten Sie darauf, dass Sie `reverse` in skalarem Kontext aufrufen, oder erzwingen Sie ihn mit der `scalar`-Funktion.*

F In einer der letzten Lektionen haben Sie die Variable `$"` beschrieben, mit der man das Trennzeichen zwischen Listenelementen festlegt, beispielsweise beim Interpolieren einer Liste in einen String. Was ist der Unterschied zwischen dieser Technik und der `join`-Funktion?

A *Beim Ergebnis gibt es keinen Unterschied. Beide machen aus einer Liste einen String mit Trennzeichen zwischen den Listenelementen. Allerdings ist `join` hier effizienter, weil der fragliche String nicht erst interpoliert werden muss.*

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist ein Segment (oder *Slice*)? Wie wirkt sich ein Segment auf die Originalliste aus?
2. Wofür werden die Variablen `$a` und `$b` in `sort`-Routinen verwendet?
3. Was machen die Operatoren <=> und `cmp`? Was ist der Unterschied zwischen den beiden?
4. Das erste Argument der Funktion `grep` ist ein Ausdruck oder Block. Wozu dient dieser Block?
5. Wie fügt man mit `splice` einem Array Elemente hinzu? Wie ersetzt man zwei Elemente durch eine Liste aus vier Elementen?
6. Wie gibt man aus dem Block in `map` einen Wert zurück?

Übungen

1. Schreiben Sie folgende Ausdrücke neu, und zwar mit `splice`:

```
push @liste, 1;
push @liste, (2,3,4);
$list[5] = "foo";
shift @liste;
```

2. FEHLERSUCHE: Was ist falsch an diesem Code (Tipp: Es könnten mehrere Fehler sein)?

```
while ($i <= $#liste) {
   $str = @novel[$i++];
   push @ergebnis, reverse $str;
}
```

3. FEHLERSUCHE: Und was ist hiermit?

```
while ($pos < (length $str) and $pos != -1) {
    $pos = index($str, $such, $pos);

    if ($pos != -1 ) {
        $count++;
    }
}
```

4. Schreiben Sie eine neue Version des folgenden Ausdrucks. Verwenden Sie `foreach` und `push`.

```
@liste2 = grep {$_ < 5 } @liste;
```

5. Schreiben Sie ein Skript, das um die Eingabe eines Strings und eines einzelnen Zeichens bittet und dann ausgibt, wie oft dieses Zeichen in dem String vorkommt. Verwenden Sie die Funktion `index`.
6. Schreiben Sie das Skript aus Übung 5 noch einmal, diesmal ohne `index` (oder Patterns, falls Sie sich damit schon auskennen). Tipp: Probieren Sie's mit `grep`.

Antworten

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

Antworten zum Quiz

1. Ein *Slice* oder Segment ist eine Art Teilarray von Elementen aus einem Array oder Hash. Segmente haben keine Auswirkung auf die Originalliste; sie kopieren die ausgewählten Elemente in eine neue Liste, anders als bei Verwendung von `splice`, das die Originalliste (bzw. das Array) dauerhaft verändert.
2. Die Variablen `$a` und `$b` in einer `sort`-Routine sind lokale Variablen dieser Routine und enthalten die Werte der beiden jeweils zu vergleichenden Elemente.
3. Die Operatoren <=> und `cmp` haben zwei Operanden und geben –1 zurück, wenn der erste kleiner ist als der zweite, 0, wenn die beiden gleich sind, und 1, wenn der

erste größer ist als der zweite. Wenn dieses Verhalten auch nicht allzu sinnvoll für den normalen »Hausgebrauch« scheinen mag, so sind diese Operatoren für `sort`-Routinen doch von großem Nutzen, denn dort werden genau diese Angaben für die Sortierung gebraucht.

Der Unterschied zwischen <=> und `cmp` ist der gleiche wie der zwischen == und `eq`: <=> nimmt man für Zahlen, `cmp` für Strings.

4. Der Ausdruck oder Block, den man als Argument an `grep` übergibt, ist ein Kriterium, ob ein bestimmtes Listenelement (wie in `$_` gespeichert) in der Ergebnisliste gespeichert werden soll: Evaluiert der Ausdruck (oder Block) zu *wahr*, wird das Element gespeichert.
5. Man fügt einem Array mit `splice` Elemente hinzu, indem man als Längenargument (das dritte Argument) 0 angibt. So werden keine Elemente entfernt und die neuen Elemente an der angegebenen Startposition eingefügt:

```
splice(@array, 0, 0, (1,2,3,4);
# fügt (1,2,3,4) am Array-Anfang ein
```

6. Damit ein Ausdrucksblock in `map` einen bestimmten Wert zurückgibt, muss der im Block zuletzt ausgewertete Ausdruck diesen Wert liefern. Das Ergebnis der letzten Auswertung ist auch der Rückgabewert des Blocks und wird dann an die neue Liste weitergegeben.

Lösungen zu den Übungen

1. Hier die Antworten:

```
splice(@liste, $#liste+1, 0, 1);
splice(@liste, $#liste+1, 0, (2,3,4));
splice(@liste, 5, 1,"foo");
splice(@liste, 0, 1);
```

2. Dieser Code hat wirklich mehrere Fehler, den ersten in der zweiten Zeile: Der Array-Zugriffsausdruck ist hier in Wahrheit ein Array-Segment. Sie erhalten hier daher keinen String, sondern ein Array mit einem einzigen Element. Perl-Warnungen fangen diesen Fehler ab.

   Doch selbst wenn Sie diesen Fehler beheben, wird der String nicht umgekehrt. Denn in der dritten Zeile wird `reverse` in einem Listenkontext aufgerufen und der String dadurch als einelementige Liste interpretiert. Stellen Sie `reverse` mit der `scalar`-Funktion in skalaren Kontext.
3. Irgendwo in der Schleife müssen Sie die aktuelle Position erhöhen, sonst findet die Schleife immer denselben Substring an immer derselben Stelle, wieder und wieder – endlos.

4. Hier eine Antwort:

```
foreach (@liste) {
   if ($_ < 5) {
       push @liste2, $_;
   }
}
```

5. Hier eine Antwort:

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

$str = ''; #String
$key = ''; #gesuchtes Zeichen
$pos = 0;   #temp Position
$count = 0; #Zaehler

print "Geben Sie den zu durchsuchenden String ein: ";
chomp($str = <STDIN>);
print "Geben Sie das zu zählende Zeichen ein: ";
chomp($key = <STDIN>);

while ($pos < (length $str) and $pos != -1) {
    $pos = index($str, $key, $pos);

    if ($pos != -1 ) {
        $count++;
        $pos++;
    }
}

print "$key kommt im String $count mal vor.\n";
```

6. Der knifflige Teil an dieser Übung ist die Zeile, in der `split` den String in eine Liste mit den Zeichen umwandelt. Haben Sie erst einmal eine Liste, ist auch der Einsatz von `grep` ganz einfach.

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

$str = ''; #String
$key = ''; #gesuchtes Zeichen
$count = 0; #Zaehler

print "Geben Sie den zu durchsuchenden String ein: ";
chomp($str = <STDIN>);
print "Geben Sie das zu zählende Zeichen ein: ";
chomp($key = <STDIN>);
```

```
@chars = split('',$str);
$count = grep {$_ eq $key} @chars;

print "$key kommt im String $count mal vor.\n";
```