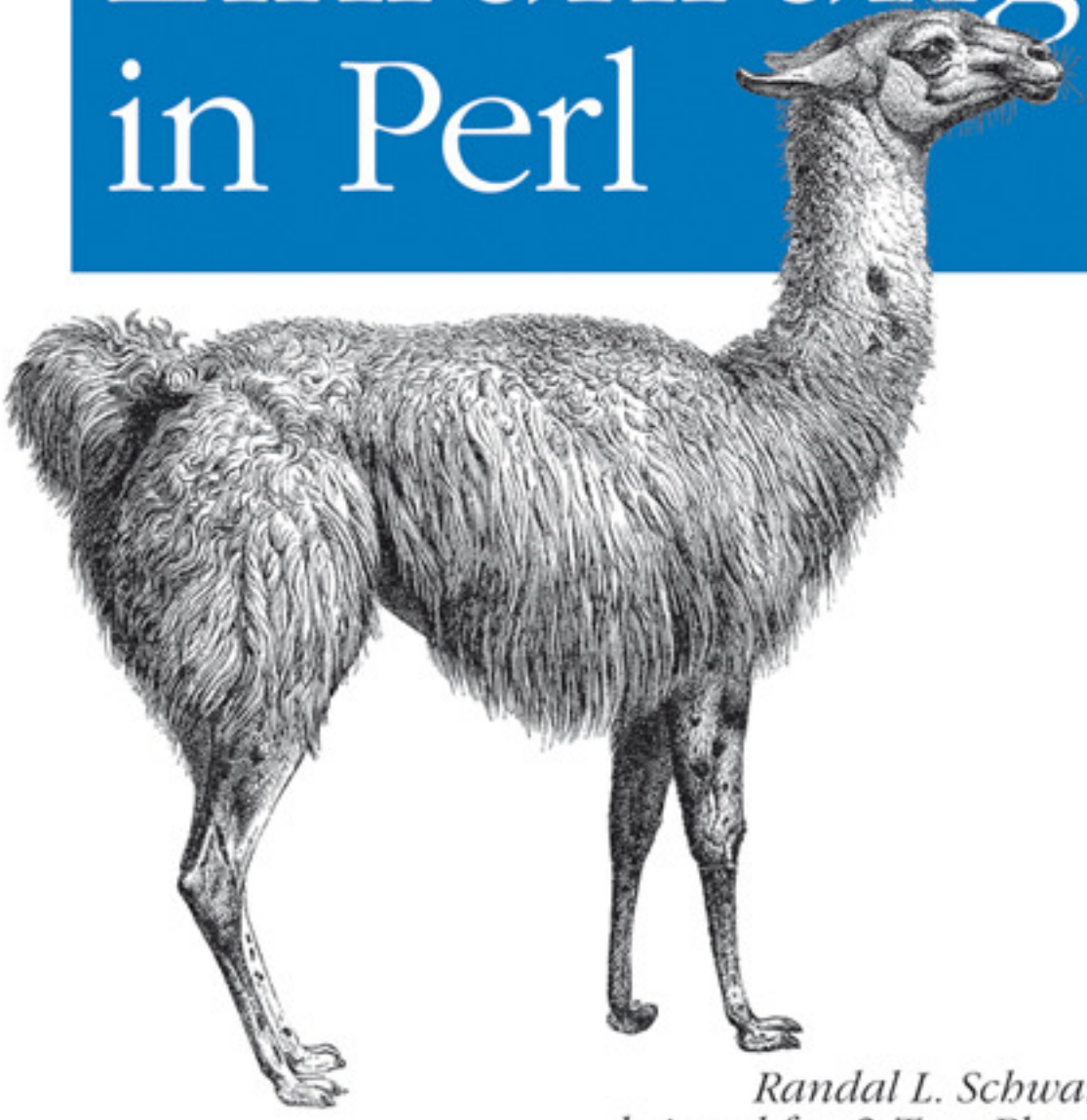
Behandelt Perl 5.14

Deutsche Ausgabe der 6. Auflage

Einführung in Perl



Randal L. Schwartz,
brian d foy & Tom Phoenix

Deutsche Übersetzung von Eike Nitz

O'REILLY®

Inhalt

KAPITEL 4

Subroutinen

Wir haben bereits einige eingebaute Systemfunktionen kennengelernt, unter anderem `chomp`, `reverse`, `print` und so weiter. Wie in anderen Sprachen ist es auch in Perl möglich, *Subroutinen*, also benutzerdefinierte Funktionen, zu erstellen.[1] Dadurch kann dasselbe Codestück in einem Programm mehrmals verwendet werden.

Der Name einer Subroutine besteht ebenfalls aus einem Perl-Identifier (Buchstaben, Zahlen und Unterstrichen, aber keine Ziffer am Anfang), dem ein Ampersand-Zeichen (&, »Kaufmanns-Und«) vorangestellt wird. Dieses Zeichen kann in bestimmten Fällen auch weggelassen werden. Die entsprechende Regel finden Sie am Ende dieses Kapitels. Bis auf Weiteres werden wir das Zeichen immer verwenden, sofern es nicht ausdrücklich verboten ist. Dadurch sind Sie immer auf der sicheren Seite. Selbstverständlich werden wir Ihnen auch sagen, an welchen Stellen es verboten ist.

Genau wie Skalare und Arrays haben auch Subroutinen ihren eigenen Namensraum, so dass Perl nicht durcheinanderkommt, wenn Sie eine Subroutine namens `&fred` und einen Skalar `$fred` nebeneinander in Ihrem Programm benutzen wollen – auch wenn es dafür unter normalen Umständen eigentlich keinen Grund gibt.

Subroutinen definieren

Um Ihre eigene Subroutine zu definieren, benutzen Sie das Schlüsselwort `sub`, gefolgt vom Namen der Subroutine (ohne das Kaufmanns-Und). Darauf folgt der eingerückte Codeblock in geschweiften Klammern, den wir den *Körper* der Subroutine nennen, zum Beispiel so:

1 Im Gegensatz zu Pascal unterscheiden wir in Perl nicht zwischen einer *Funktion*, die einen Wert zurückgibt, und einer *Prozedur*, die das nicht tut. Eine *Subroutine* in Perl ist immer benutzerdefiniert, während eine *Funktion* dies nicht unbedingt sein muss. Das bedeutet: Das Wort *Funktion* kann als Synonym für *Subroutine* benutzt werden, oder es kann eine von den Funktionen meinen, die in Perl eingebaut sind. Deshalb heißt dieses Kapitel auch *Subroutinen*, da es um die Funktionen geht, die Sie selbst definieren können, und nicht um die eingebauten. Meistens jedenfalls.

```
sub marine {
  $n += 1;  # globale Variable $n
  print "Hallo, Taucher Nummer $n!\n";
}
```

Subroutinen können Sie an jeder beliebigen Stelle in Ihrem Programm definieren. Programmierer, die sich mit Sprachen wie C oder Pascal auskennen, schreiben ihre Subroutinen vermutlich lieber an den Anfang ihrer Programme; andere wiederum schreiben sie lieber an das Ende, so dass der Hauptteil des Programms am Anfang steht. Die Wahl ist Ihnen überlassen. Sie brauchen Ihre Subroutinen jedenfalls nicht vorzudeklarieren.[2]

Subroutinendefinitionen sind global, das heißt, sie gelten in Ihrem gesamten Programm; ohne irgendwelche schlauen Tricks anzuwenden, sind private Subroutinen nicht möglich.[3] Wenn Sie also in einem Programm zwei Subroutinen mit demselben Namen definieren,[4] überschreibt die später definierte die von vorher. Allerdings sagt Perl Ihnen Bescheid, wenn das passiert, sofern Sie Warnungen angeschaltet haben. Dieses Vorgehen wird in der Regel als schlechte Form angesehen, oder auch als Zeichen von Verwirrung beim Wartungsprogrammierer.

Wie Sie in unserem vorigen Beispiel vielleicht schon bemerkt haben, ist es möglich, globale Variablen im Körper einer Subroutine zu benutzen. Um genau zu sein, waren alle Variablen, die Sie bisher gesehen haben, global, das heißt von jedem Teil Ihres Programms aus zugänglich. Das verschreckt natürlich die Sprachpuristen, aber die hat das Perl-Entwicklungsteam schon vor Jahren aus der Stadt gejagt. Im Abschnitt »Private Variablen in Subroutinen« auf Seite 72 weiter unten in diesem Kapitel zeigen wir Ihnen, wie Sie Variablen anlegen können, die nur in Ihrer Subroutine gültig sind.

Subroutinen aufrufen

Innerhalb eines Ausdrucks rufen Sie eine Subroutine auf, indem Sie einfach ihren Namen (mit vorangestelltem Kaufmanns-Und) verwenden:[5]

```
&marine;  # sagt Hallo, Taucher Nummer 1!
&marine;  # sagt Hallo, Taucher Nummer 2!
&marine;  # sagt Hallo, Taucher Nummer 3!
&marine;  # sagt Hallo, Taucher Nummer 4!
```

2 Es sei denn, Ihre Subroutine ist besonders trickreich angelegt und deklariert einen »Prototyp«, der den Compiler anweist, die aufrufenden Argumente auf eine bestimmte Art und Weise zu parsen und zu interpretieren. In dem seltenen Fall, dass Sie so etwas brauchen, finden Sie die nötigen Informationen in der *perlsub*-Dokumentation.

3 Wenn Sie richtig schlau sein wollen, lesen Sie nach, was in der Dokumentation über Codereferenzen steht, die in privaten (lexikalischen) Variablen gespeichert werden.

4 Über gleichnamige Subroutinen in unterschiedlichen Packages werden wir erst in *Intermediate Perl* (»Alpaka-Buch«) zu sprechen kommen.

5 Und regelmäßig auch ein nachgestelltes Paar runder Klammern, selbst wenn diese leer sind. So wie wir sie hier geschrieben haben, würde unsere Subroutine den Wert von `@_` von einer aufrufenden Subroutine »erben«. Wir werden gleich darauf kommen; hören Sie also hier nicht auf zu lesen, denn Sie könnten sonst Code schreiben, der Dinge tut, die Sie nicht erwarten.

Sie werden im Verlauf dieses Kapitels noch andere Methoden des Aufrufens von Subroutinen kennenlernen.

Rückgabewerte

Subroutinen werden immer als Teil eines Ausdrucks aufgerufen, selbst wenn das Ergebnis des Ausdrucks nicht benutzt wird. Als wir vorhin die Subroutine `&marine` aufgerufen haben, haben wir den Wert des Ausdrucks berechnet, das Ergebnis aber wieder verworfen.

Oft ist es aber so, dass Sie beim Aufruf einer Subroutine auch das Ergebnis verwenden wollen. Sie verwenden also den *Rückgabewert* einer Subroutine. In Perl haben alle Subroutinen einen Rückgabewert – es macht dabei keinen Unterschied, ob explizit ein Wert zurückgegeben wird oder nicht. Dennoch hat nicht jede Subroutine auch einen *nützlichen* Rückgabewert.

Da alle Perl-Subroutinen so aufgerufen werden können, dass sie einen Wert zurückgeben, wäre es reine Verschwendung, wenn dafür jedes Mal eine spezielle Syntax benutzt werden müsste. Deshalb hat Larry die Sache vereinfacht: Wenn eine Subroutine aufgerufen wird, werden die Werte automatisch im Hintergrund mitberechnet. Der Wert, der *zuletzt* in der Subroutine berechnet wird, ist *automatisch* auch der Rückgabewert.

Lassen Sie uns zum Beispiel einmal die folgende Subroutine betrachten:

```
sub summe_von_fred_und_barney {
  print "Sie haben die summe_von_fred_und_barney-Subroutine aufgerufen.\n";
  $fred + $barney;  # dies ist der Rückgabewert
}
```

Der letzte Ausdruck, der im Körper der Subroutine ausgewertet wurde, berechnet die Summe von `$fred` und `$barney`. Also ist die Summe von `$fred` und `$barney` auch der Rückgabewert der Subroutine. Hier sehen Sie die Subroutine in Aktion:

```
$fred   = 3;
$barney = 4;
$wilma  = &summe_von_fred_und_barney;     # $wilma erhält den Wert 7
print "\$wilma hat den Wert $wilma.\n";
$betty  = 3 * &summe_von_fred_und_barney; # $betty erhält den Wert 21
print "\$betty hat den Wert $betty.\n";
```

Dieser Code erzeugt die folgenden Ausgaben:

```
Sie haben die summe_von_fred_und_barney-Subroutine aufgerufen.
$wilma hat den Wert 7.
Sie haben die summe_von_fred_und_barney-Subroutine aufgerufen.
$betty hat den Wert 21.
```

Hierbei ist die `print`-Anweisung eine Hilfe zum Debuggen, damit Sie sehen, dass Sie die Subroutine tatsächlich aufgerufen haben. Ist Ihr Programm einmal fertig gestellt, würden

Sie diese Anweisungen wieder entfernen. Aber nehmen wir einmal an, Sie hätten am Ende der Subroutine ein weiteres print eingefügt:

```
sub summe_von_fred_und_barney {
  print "Sie haben die summe_von_fred_und_barney-Subroutine aufgerufen.\n";
  $fred + $barney;  # dies ist jetzt nicht mehr der Rückgabewert
  print "Juhu! Jetzt bin ich der Rückgabewert!\n";  # Hoppla!
}
```

In diesem Beispiel ist der letzte ausgewertete Ausdruck nicht mehr die Addition, sondern die print-Anweisung. Der Rückgabewert von print ist normalerweise 1, was einfach »die Ausgabe war erfolgreich« bedeutet.[6] Das ist jedoch nicht der Rückgabewert, den wir eigentlich haben wollten. Seien Sie also vorsichtig, wenn Sie eine Subroutine um zusätzlichen Code ergänzen, da der letzte ausgewertete Ausdruck der Rückgabewert sein wird.

Und was ist jetzt mit der Summe aus $fred und $barney in der zweiten Subroutine passiert? Wir haben den Wert nirgendwo abgelegt, also hat Perl ihn wieder verworfen. Hätten Sie die Warnungen eingeschaltet, hätte Perl Sie höchstwahrscheinlich gewarnt, dass hier ein »useless use of addition in a void context« (eine unnütze Verwendung einer Addition ohne Zusammenhang) vorliegt. Das ist nur eine andere Art zu sagen, dass Sie die Summe nicht benutzen, weder durch Speichern in einer Variablen noch auf eine andere Weise.

»Der letzte ausgewertete Ausdruck« meint hier tatsächlich den letzten Ausdruck, den Perl auswertet, und nicht den letzten Ausdruck in der Subroutine. Die folgende Subroutine gibt z. B. den größeren der beiden Werte $fred und $barney zurück:

```
sub ist_fred_oder_barney_groesser {
  if ($fred > $barney) {
    $fred;
  } else {
    $barney;
  }
}
```

Der letzte ausgewertete Ausdruck ist hier entweder $fred oder $barney. Je nachdem, welcher Wert größer ist, ist also $fred oder $barney der Rückgabewert. Welcher von beiden das sein wird, können Sie erst sagen, wenn Sie wissen, welche Variablen $fred und $barney zur Laufzeit des Programms haben werden.

Dies sind alles noch recht triviale Beispiele. Das Ganze wird besser, sobald Sie in der Lage sind, die Subroutine bei jedem Aufruf mit verschiedenen Werten aufzurufen, anstatt sich auf globale Variablen verlassen zu müssen. Wie der Zufall es will, ist genau das das Thema des kommenden Abschnitts.

6 Der Rückgabewert von print ist wahr bei einer erfolgreichen Operation und falsch, wenn die Ausgabe fehlschlägt. Wie festgestellt wird, ob eine Operation fehlgeschlagen ist, erfahren Sie im nächsten Kapitel.

Argumente

Die Subroutine `ist_fred_oder_barney_groesser` wäre noch viel nützlicher, wenn Sie nicht gezwungen wären, die globalen Variablen `$fred` und `$barney` zu verwenden. Wollten Sie zum Beispiel auch den größeren Wert von `$wilma` und `$betty` ermitteln, müssten Sie diese erst in die Variablen `$fred` und `$barney` kopieren, bevor die Subroutine für diesen Fall benutzbar wäre. Stünde in diesen Variablen nun bereits etwas anderes Nützliches, müssten Sie diese Werte vor der Operation erst sichern, zum Beispiel in `$fred_sichern` und `$barney_sichern`. Am Ende der Subroutine müssten Sie dann die ursprünglichen Werte auch noch wieder nach `$fred` und `$barney` zurückkopieren.

Zum Glück gibt es in Perl Subroutinenargumente. Um eine Liste von Argumenten an die Subroutine zu übergeben, schreiben Sie diese in runden Klammern hinter den Aufruf der Subroutine, wie hier gezeigt ist:

```
$n = &max(10, 15);  # Subroutine wird mit zwei Argumenten aufgerufen
```

Die Liste wird an die Subroutine *übergeben*, das heißt, dass Perl die Argumente der Subroutine zur Verfügung stellt, die sie dann ganz nach Bedarf bearbeiten kann. Selbstverständlich muss auch die Parameterliste (ein anderer Name für die Liste der Argumente) irgendwo gespeichert werden. Perl tut das automatisch in der speziellen Arrayvariable `@_`, die für die Dauer der Subroutine die aufrufenden Argumente enthält.

Der erste Parameter, der auf diese Weise an die Subroutine übergeben wurde, ist folglich in `$_[0]` zu finden, der zweite in `$_[1]` und so weiter. An dieser Stelle ein wichtiger Hinweis: Diese Variablen haben mit der Variablen `$_` so wenig zu tun wie `$dino[3]` (ein Element aus dem Array `@dino`) etwas mit der Variablen `$dino` zu tun hat. Die Parameterliste muss für die Verwendung in der Subroutine einfach in einer Arrayvariablen gespeichert werden, und Perl verwendet dafür das Array `@_`.

Jetzt *könnten* Sie eine Subroutine namens `&max` schreiben, die so ähnlich aussieht wie `&ist_fred_groesser_als_barney`. Anstelle von `$fred` könnten Sie nun den ersten Subroutinenparameter einsetzen (`$_[0]`) und anstelle von `$barney` den zweiten Subroutinenparameter (`$_[1]`). Am Schluss *könnten* Sie also etwa Folgendes schreiben:

```
sub max {
  # vergleichen Sie dies mit &ist_fred_oder_barney_groesser
  if ($_[0] > $_[1]) {
    $_[0];
  } else {
    $_[1];
  }
}
```

Wie gesagt, Sie *könnten* das so machen. Mit den ganzen Indizes sieht das jedoch ziemlich hässlich aus; zudem ist es schwer zu lesen, zu schreiben und zu debuggen. Sie werden gleich eine bessere Methode kennenlernen.

Ein anderes Problem besteht darin, dass der Name &max zwar schön kurz ist, uns aber nicht daran erinnert, dass diese Subroutine nur mit exakt zwei Parametern korrekt funktioniert.

```
$n = &max(10, 15, 27);  # Hoppla!
```

max ignoriert weitere Parameter, da es sich das dritte Argument, $_[2], nie ansieht. Perl ist es egal, ob sich etwas darin befindet oder nicht. Im Fall unzureichender Parameter wird bei dem Versuch, hinter das Ende von @_ zu schauen, einfach undef verwendet, genau wie bei jedem anderen Array auch. Weiter hinten in diesem Kapitel zeigen wir, wie Sie die &max-Subroutine so umschreiben können, dass eine beliebige Anzahl von Parametern übergeben werden kann.

Die Variable @_ existiert in der Subroutine lokal.[7] Gibt es also auch eine globale Variable @_, so sichert Perl ihren Wert vor dem Aufruf der Subroutine. Ist die Subroutine beendet, wird der ursprüngliche Wert wiederhergestellt.[8] Das hat zur Folge, dass eine Subroutine Argumente an eine andere Subroutine übergeben kann, ohne dabei die Werte ihrer eigenen @_-Variablen zu verlieren. Bei verschachtelten Subroutinenaufrufen bekommt so jede Subroutine ihr eigenes @_-Array zugeteilt. Das funktioniert sogar dann, wenn sich die Subroutine rekursiv selbst aufruft.

Private Variablen in Subroutinen

Wenn Perl uns für jeden Aufruf ein neues @_ geben kann, ist das dann auch mit Variablen möglich? Aber sicher!

Standardmäßig sind alle Variablen in Ihrem Programm als globale Variablen angelegt; das heißt, sie sind von jedem Teil des Programms aus zugänglich. Mit dem my-Operator können Sie aber jederzeit auch private bzw. *lexikalische Variablen* anlegen, zum Beispiel:

```
sub max {
  my($m, $n);    # neue, private Variablen für diesen Block anlegen
  ($m, $n) = @_; # die Parameter benennen
  if ($m > $n) { $m } else { $n }
}
```

Diese Variablen sind nun privat beziehungsweise besitzen einen auf den umschließenden Block begrenzten Geltungsbereich; andere Variablen außerhalb des Blocks, die auch $m oder $n heißen, werden davon nicht beeinflusst. Das funktioniert ebenso in der anderen Richtung: Kein Code kann diese privaten Variablen versehentlich oder

7 Sofern vor dem Namen der aufgerufenen Subroutine ein Kaufmanns-Und steht und sie ohne nachgestellte runde Klammern (oder Argumente) benutzt wird, »erbt« die aufgerufene Subroutine das Array @_ von der aufrufenden Routine. Das ist in der Regel keine gute Idee, kann aber manchmal ganz nützlich sein.

8 Sie erkennen hier vielleicht den gleichen Mechanismus, der auch bei der foreach-Schleife zur Anwendung kommt. In beiden Fällen sichert Perl den Wert einer Variablen und stellt ihn automatisch wieder her.

absichtlich[9] von außen verändern. Sie könnten diese Subroutine also in ein beliebiges Perl-Programm auf dieser Welt einbauen und dabei sicher sein, dass keine anderen Variablen in diesem Programm, die zufällig auch `$m` oder `$n` heißen, davon durcheinander gebracht werden.[10]

Innerhalb eines `if`-Blocks muss nach dem Ausdruck, der den Rückgabewert definiert, kein Semikolon stehen. Auch wenn Perl es gestattet, das letzte Semikolon innerhalb eines Blocks wegzulassen,[11] lässt man es normalerweise nur dann weg, wenn der Code so einfach ist, dass der Block in einer einzigen Zeile Platz findet.

Die Subroutine aus dem vorigen Beispiel lässt sich sogar noch weiter vereinfachen. Ist Ihnen aufgefallen, dass die Liste (`$m, $n`) zweimal erwähnt wird? Dabei lässt sich der `my`-Operator auch auf eine in runden Klammern stehende Liste anwenden. Daher ist es üblich, die beiden ersten Anweisungen in der Subroutine miteinander zu kombinieren:

```
my($m, $n) = @_;  # Subroutinenparameter benennen und "privatisieren"
```

Diese Anweisung erzeugt gleichzeitig zwei private Variablen und weist ihnen Werte zu. Dadurch haben die Parameter nun die einfacheren Namen `$m` und `$n`. Fast jede Subroutine beginnt mit einer Zeile wie dieser, in der die Parameter benannt werden. Wenn Sie diese Zeile in Zukunft sehen, wissen Sie also, dass die Subroutine zwei skalare Parameter erwartet, die innerhalb der Subroutine `$m` und `$n` heißen.

Parameterlisten mit variabler Länge

In der rauen Wirklichkeit werden oft Parameterlisten an Subroutinen übergeben, die beliebige Längen haben können. Das liegt ebenfalls an der Perl eigenen Philosophie der »Grenzenlosigkeit«. Sie unterscheidet Perl von vielen traditionellen Programmiersprachen, bei denen für jede Subroutine ein striktes Typing gilt. Das bedeutet, dass immer nur eine vordefinierte Anzahl von Parametern erlaubt ist, die einem vordefinierten Typ entsprechen müssen. Es ist zwar schön, dass Perl so flexibel ist; das kann aber (wie Sie bei der `&max`-Subroutine gesehen haben) zu Problemen führen, sobald wir die Subroutine mit einer anderen Anzahl von Argumenten aufrufen, als sie erwartet.

9 Fortgeschrittene Programmierer werden erkennen, dass die lexikalische Variable auch außerhalb des Geltungsbereichs über eine Referenz zugänglich sein kann. Dies lässt sich mithilfe von Referenzen bewerkstelligen, aber niemals mit dem Variablennamen selbst. Wie das funktioniert, zeigen wir in *Intermediate Perl* (»Alpaka-Buch«).

10 Wenn das Programm natürlich bereits eine Subroutine mit dem Namen `&max` hätte, würden Sie *die* durcheinanderbringen.

11 Das Semikolon beendet nämlich eigentlich keine Anweisungen, sondern trennt sie nur voneinander.

Es lässt sich jedoch leicht prüfen, ob die korrekte Anzahl von Argumenten übergeben wurde, indem wir das Array @_ auf seine Länge überprüfen. Um die Parameterliste von &max zu testen, hätten wir die Subroutine z.B. auch wie folgt schreiben können:[12]

```
sub max {
  if (@_ != 2) {
    print "WARNUNG! Sie müssen &max genau zwei Argumente übergeben!\n";
  }
  # weitermachen wie zuvor...
  .
  .
  .
}
```

Die if-Überprüfung benutzt hier den Namen des Arrays in skalarem Kontext, um herauszufinden, wie viele Elemente es enthält. Dieses Verfahren kennen Sie bereits aus Kapitel 3.

Auch diese Art der Überprüfung wird in der Praxis so gut wie nie benutzt. Es ist besser, Subroutinen so zu schreiben, dass sie sich auf die Parameter einstellen.

Eine bessere &max-Subroutine

Lassen Sie uns &max also so umschreiben, dass sie mit einer beliebigen Anzahl von Argumenten umgehen kann, so dass folgender Aufruf möglich wird:

```
$maximum = &max(3, 5, 10, 4, 6);

sub max {
  my($max_bis_jetzt) = shift @_;  # Das erste Argument ist bis jetzt
                                  # das größte.
  foreach (@_) {                  # Die weiteren Argumente ansehen.
    if ($_ > $max_bis_jetzt) {    # Könnte dieser Wert vielleicht
      $max_bis_jetzt = $_;        # noch größer sein?
    }
  }
  $max_bis_jetzt;
}
```

Dieser Code benutzt einen sogenannten »Hochwassermarken«-Algorithmus: Wenn nach einer Flut der Wasserstand sinkt, zeigt die Hochwassermarke, an welcher Stelle das Wasser seinen höchsten Stand hatte. In unserer Subroutine benutzen wir zum Feststellen des bisher höchsten Pegelstands die Variable $max_bis_jetzt, in der jeweils die größte bisher gefundene Zahl gespeichert wird.

12 Sobald Sie im nächsten Kapitel die Funktion warn kennengelernt haben, werden Sie wissen, wie Sie eine falsche Benutzung Ihrer Subroutine in eine korrekte Warnung verwandeln können. Oder Sie überlegen sich, dass dieser Fall wichtig genug ist, um die Funktion die zu benutzen, die im selben Kapitel vorgestellt wird.

Die erste Zeile setzt $max_bis_jetzt auf den Wert 3 (den ersten Parameter in unserem Beispiel). Dieser wird per shift aus dem Parameter-Array @_ entfernt und $max_bis_jetzt zugewiesen. @_ enthält jetzt also nur noch die Werte (5, 10, 4, 6), da wir die 3 entfernt haben. Bis jetzt ist also der erste Parameter (3) die größte Zahl, da 3 die *einzige* Zahl ist, die wir bis jetzt gesehen haben.

Jetzt geht die foreach-Schleife die übrigen Werte aus der Parameterliste in @_ der Reihe nach durch. Standardmäßig wird als Schleifenkontrollvariable $_ benutzt. (Denken Sie aber daran: @_ und $_ haben prinzipiell nichts miteinander zu tun; sie haben nur zufällig ähnliche Namen.) Beim ersten Schleifendurchlauf erhält $_ den Wert 5. Die if-Überprüfung stellt fest, dass 5 größer ist als der Wert, der bisher in $max_bis_jetzt gespeichert war, also wird $max_bis_jetzt auf den Wert 5 gesetzt – die neue Hochwassermarke.

Beim nächsten Schleifendurchlauf hat $_ den Wert 10. Das ist ein Rekordhochwasser, also wird nun dieser Wert in $max_bis_jetzt gespeichert.

Einen Durchlauf später hat $_ den Wert 4. Der if-Test schlägt fehl, da 4 kleiner ist als der in $max_bis_jetzt gespeicherte Wert. Daraufhin wird der Körper der if-Überprüfung übersprungen.

Noch einen Schleifendurchlauf weiter hat $_ den Wert 6, also wird auch dieses Mal die von der if-Überprüfung abhängige Neuzuweisung übersprungen. Da sich keine weiteren Werte mehr in der Parameterliste befinden, wird an dieser Stelle die Schleife beendet.

Schließlich wird $max_bis_jetzt als Rückgabewert verwendet. Dies ist die größte Zahl, die wir bei unserer Überprüfung gefunden haben: 10.

Leere Parameterlisten

Der verbesserte &max-Algorithmus funktioniert jetzt schon wesentlich besser, selbst wenn mehr als zwei Parameter übergeben werden. Aber was passiert, wenn überhaupt kein Wert übergeben wird?

Auf den ersten Blick scheint dieser Fall etwas zu esoterisch zu sein, um sich darüber Sorgen zu machen. Warum sollte jemand &max aufrufen, ohne irgendwelche Parameter zu übergeben? Das kommt selten vor, aber vielleicht haben Sie ja eine Zeile wie diese in Ihrem Programm:

```
$maximum = &max(@zahlen);
```

Das Array @zahlen kann unter Umständen eine leere Liste enthalten; vielleicht wurde sie aus einer Datei gelesen, die sich als leer herausgestellt hat. Sie müssen also wissen, was &max in einem solchen Fall tut.

In der ersten Zeile der Subroutine wird versucht, der Variablen $max_bis_jetzt den mit shift aus dem (jetzt leeren) Array @_ entfernten ersten Wert zuzuweisen. Das ist völlig harmlos. Das Array bleibt leer, und shift weist $max_bis_jetzt den Wert undef zu.

Als Nächstes soll mit der foreach-Schleife über @_ iteriert werden. Da das Array jedoch leer ist, wird die Schleife nicht ausgeführt.

Der Rückgabewert unserer Subroutine (der Wert von $max_bis_jetzt) ist also ebenfalls undef. Wenn die Liste keine definierten Werte enthält, ist undef folglich die einzig richtige Antwort.

Selbstverständlich sollte derjenige, der die Subroutine aufruft, sich darüber im Klaren sein, dass der Rückgabewert undef sein kann; oder man könnte sicherstellen, dass die Parameterliste niemals leer ist.

Anmerkungen zu lexikalischen (my)-Variablen

Lexikalische Variablen können in jedem beliebigen Block benutzt werden, nicht nur innerhalb einer Subroutine, zum Beispiel in Blöcken einer Fallunterscheidung mit if oder in einer Schleife mit while oder foreach:

```
foreach (1..10) {
  my($quadrat) = $_ * $_;  # für diese Schleife private Variable $quadrat
  print "$_ zum Quadrat ist $quadrat.\n";
}
```

Die Variable $quadrat ist innerhalb des umschließenden Blocks als private (lexikalische) Variable angelegt. In diesem Fall ist das der Block der foreach-Schleife. Gibt es keinen umschließenden Block, so ist die Variable innerhalb der gesamten Quellcodedatei als private Variable angelegt. Momentan benutzen Ihre Programme zwar nur eine Datei für den Quellcode,[13] das kann sich aber ändern. Das wichtige Konzept dahinter ist, dass der *Geltungsbereich* eines lexikalischen Variablennamens auf den kleinsten umschließenden Block bzw. die Datei beschränkt ist. Der *einzige* Code, der $quadrat benutzen kann, befindet sich im selben Geltungsbereich wie die Variable. Dadurch wird die Pflege Ihres Codes erheblich leichter: Wird in $quadrat ein falscher Wert gefunden, so kann der Schuldige in einem begrenzten Codestück aufgespürt werden. Erfahrene Programmierer haben (oft auf die harte Tour) gelernt, dass die Begrenzung des Geltungsbereichs einer Variablen auf eine Seite oder auf ein paar Zeilen Code den Entwicklungs- und Testzyklus erheblich beschleunigen kann.

Beachten Sie auch, dass der my-Operator den Kontext einer Zuweisung nicht verändert:

```
my($num) = @_;  # Listenkontext, das Gleiche wie ($num) = @_;
my $num  = @_;  # skalarer Kontext, das Gleiche wie $num = @_;
```

Im ersten Beispiel erhält $num den ersten Parameter von @_ in Form einer Zuweisung im Listenkontext. Im zweiten Beispiel erhält er stattdessen die Anzahl der Elemente im skalaren Kontext. Hierbei könnte jede der beiden Codezeilen das sein, was der Programmierer wollte. Anhand nur einer Zeile lässt sich das jedoch nicht feststellen. Daher kann Perl

13 Wiederverwendbare Bibliotheken und Module behandeln wir in *Intermediate Perl* (»Alpaka-Buch«).

Sie auch nicht warnen, wenn Sie die falsche Variante benutzen. (Selbstverständlich sollten Sie nicht *beide* Zeilen in derselben Subroutine verwenden, da Sie keine zwei lexikalischen Variablen gleichen Namens innerhalb eines einzigen Geltungsbereichs deklarieren können; das ist nur ein Beispiel.) Wenn Sie Code wie diesen sehen, lässt sich der Kontext der Zuweisung immer ermitteln, indem Sie überlegen, wie dieser ohne das Wort `my` aussehen würde.

Solange wir allerdings die Verwendung von `my( )` mit runden Klammern besprechen, sollten Sie nicht vergessen, dass `my` nur eine *einzelne* lexikalische Variable deklariert:[14]

```
my $fred, $barney;        # falsch, $barney wird nicht deklariert
my($fred, $barney);       # deklariert beide Variablen
```

Selbstverständlich lassen sich mit `my` auch neue private Arrays anlegen:[15]

```
my @telefon_nummern;
```

Neu angelegte Variablen sind zu Beginn immer leer: `undef` für Skalare und eine leere Liste bei Arrays.

Bei herkömmlicher Perl-Programmierung würde man wahrscheinlich `my` verwenden, um eine neue Variable in einem Geltungsbereich einzuführen. In Kapitel 3 haben wir gesehen, dass man mithilfe der `foreach`-Struktur seine eigenen Kontrollvariablen definieren könnte. Das kann auch eine lexikalische Variable sein:

```
foreach my $stein (qw/ Feuerstein Schiefer Lava /) {
    print "Ein Stein ist $stein.\n";  # Gibt Namen von drei Steinen aus
```

Das gewinnt im folgenden Abschnitt an Bedeutung, wo wir damit anfangen werden, ein Feature zu verwenden, das Sie alle Ihre Variablen deklarieren lässt.

Das »use strict«-Pragma

Perl ist eine recht »freizügige« Sprache.[16] Aber vielleicht wollen Sie ja, dass Perl ein wenig mehr Disziplin verlangt. Dafür gibt es das Pragma `use strict`.

Unter einem *Pragma* verstehen wir einen Hinweis an den Compiler, der etwas über den Code aussagt. In diesem Fall teilt das Pragma `use strict` dem internen Compiler von Perl mit, für den Rest des Blocks oder der Quellcodedatei einige gute Programmierregeln anzuwenden.

14 Wie üblich wird die Aktivierung der Warnungen Ihnen bei einem Missbrauch von `my` einen entsprechenden Hinweis ausgeben, oder aber Sie rufen die Nummer 0180-LEXIKALISCHER-MISSBRAUCH an und melden ihn selbst. Die Verwendung des `strict`-Pragmas, auf das wir in Kürze kommen, sollte diesen Missbrauch von vornherein verbieten.

15 Oder auch Hashes, wie wir in Kapitel 6 sehen werden.

16 Das ist Ihnen mit Sicherheit noch gar nicht aufgefallen.

Wozu könnte das gut sein? Stellen Sie sich einmal vor, Sie schreiben in Ihrem Programm eine Zeile wie diese:

```
$bam_bam = 3;  # Perl erzeugt diese Variable automatisch
```

Danach schreiben Sie eine Zeit lang weiter. Nachdem die Zeile vom Bildschirm verschwunden ist, geben Sie nun die folgende Zeile ein, um die Variable zu erhöhen:

```
$bambam += 1;  # Hoppla!
```

Da Perl hier einen neuen Variablennamen findet (der Unterstrich in einem Variablennamen *ist* wichtig), wird eine neue Variable erzeugt und diese erhöht. Wenn Sie schlau sind und Glück haben, haben Sie die Warnungen eingeschaltet, und Perl kann Sie darüber informieren, dass einer oder beide globalen Variablennamen jeweils nur einmal in Ihrem Programm vorkommen. Wenn Sie allerdings nur schlau sind, kommen beide Namen nicht nur einmal in Ihrem Programm vor, und Perl kann Sie folglich auch nicht warnen.

Um Perl mitzuteilen, dass Sie ab jetzt etwas restriktiver vorgehen wollen, schreiben Sie das Pragma `use strict` an den Beginn Ihres Programms (oder in einen beliebigen Block oder eine Datei, in der Sie diese Regeln anwenden wollen):

```
use strict;  # einige vernünftige Programmierregeln anwenden
```

Ab Perl 5.12 verwenden Sie dieses Pragma implizit, wenn Sie eine »Mindestversion« angeben:

```
use 5.012;  # lädt strict für Sie
```

Jetzt wird Perl neben anderen Einschränkungen[17] darauf bestehen, dass Sie Ihre Variablen deklarieren, was üblicherweise mit `my` getan wird:[18]

```
my $bam_bam = 3;  # neue lexikalische Variable
```

Jetzt kann Perl sich beschweren, wenn Sie versehentlich eine Variable mit dem Namen `$bambam` deklarieren, wodurch der Fehler schon zur Kompilierzeit abgefangen wird.

```
$bambam += 1;   # es gibt keine Variable mit diesem Namen:
                # fataler Fehler zur Kompilierzeit
```

Diese Einschränkung gilt natürlich nur für neue Variablen; die in Perl eingebauten Variablen wie `$_` und `@_` müssen nicht extra deklariert werden.[19]

17 Wenn Sie mehr über die Beschränkungen wissen wollen, lesen Sie die Dokumentation für `strict`. Die Dokumentation für ein Pragma finden Sie unter dem Namen des jeweiligen Pragmas; die Eingabe von `perldoc strict` (oder die auf Ihrem System funktionierende Dokumentationsmethode) sollte die nötigen Dokumente für Sie finden. Kurz gesagt: Die Einschränkungen sorgen dafür, dass Strings in den meisten Fällen in Anführungszeichen stehen müssen und dass Referenzen echte (harte) Referenzen sein müssen. (Mit Referenzen, ob weich oder hart, befassen wir uns erst in *Intermediate Perl* (»Alpaka-Buch«). Für Perl-Anfänger bedeuten diese Einschränkungen jedoch kaum eine Änderung.

18 Es gibt nämlich auch noch andere Möglichkeiten, Variablen zu deklarieren.

19 Unter gewissen Umständen ist es nicht sinnvoll, `$a` und `$b` gesondert zu deklarieren, da sie intern von `sort` benutzt werden. Wenn Sie dieses Feature testen, sollten Sie also andere Variablennamen benutzen als diese zwei. Übrigens ist die Tatsache, dass `use strict` diese zwei Namen nicht verbietet, der am häufigsten gemeldete Fehler in Perl, der gar keiner ist.

Wenn Sie das `use strict`-Pragma in ein zuvor geschriebenes Programm einbauen, werden Sie in der Regel eine wahre Flut von Fehlermeldungen bekommen. Daher ist es besser, das Pragma dann zu benutzen, wenn es gebraucht wird: zu Beginn der Programmierarbeiten.

Die meisten Leute empfehlen bei allen Programmen, deren Quellcode nicht mehr auf einen Bildschirm passt, `use strict` zu benutzen. Ganz unsere Meinung.

Von jetzt an werden die meisten (aber nicht alle) unserer Beispiele so geschrieben sein, als wäre `use strict` benutzt worden, auch wenn wir die Anweisung nicht jedes Mal ausdrücklich zeigen. Das heißt: Wo es angemessen ist, deklarieren wir die Variablen in der Regel mit `my`. Obwohl wir es hier nicht immer tun, möchten wir Sie dazu ermuntern, `use strict` in möglichst jedem Ihrer Programme zu benutzen. Irgendwann werden Sie es uns danken.

Der return-Operator

Was ist, wenn Sie Ihre Subroutine direkt anhalten wollen? Der Operator `return` gibt sofort bei seinem Aufruf einen Wert aus einer Subroutine zurück.

```
my @namen    = qw/ Fred Barney Betty Dino Wilma Pebbles Bam-Bam /;
my $ergebnis = &welches_element_ist("Dino", @namen);

sub welches_element_ist {
  my($was, @array) = @_;
  foreach (0..$#array) {  # Indizes der Elemente von @array
    if ($was eq $array[$_]) {
      return $_;   # richtiges Element gefunden, Wert sofort zurückgeben
    }
  }
  -1;              # Element nicht gefunden (return ist hier optional)
}
```

Sie fordern diese Subroutine auf, im Array `@namen` den Index von `Dino` zu finden. Zuerst wird mit `my` eine Parameterliste deklariert: Es gibt das `$was`, nach dem wir suchen, und ein `@array` von Werten, in dem gesucht wird. Dies ist in unserem Fall eine Kopie des Arrays `@namen`. Die `foreach`-Schleife geht die Indizes der Werte von `@array` der Reihe nach durch (wie in Kapitel 3 gezeigt, ist der erste Index `0` und der letzte `$#array`).

Für jeden Schleifendurchlauf testen wir, ob `$was` dem Element aus unserem `@array` gleicht, das den aktuellen Index trägt.[20] Ist dies der Fall, wird der aktuelle Index sofort zurückgegeben. Das ist in Perl die häufigste Verwendung des Schlüsselworts `return` – einen Wert sofort zurückzugeben, ohne den Rest einer Subroutine auszuführen.

20 Es ist Ihnen doch aufgefallen, dass wir hier `eq` (equal, engl. »gleich«), den Vergleichsoperator für Strings, anstelle des numerischen Gegenstücks == benutzt haben, oder?

Und wenn wir nun überhaupt kein Element gefunden hätten? In diesem Fall hat sich der Autor der Subroutine entschieden, als Code für »kein Wert gefunden« `-1` zurückzugeben. Vermutlich wäre es etwas »perliger«, in diesem Fall `undef` zurückzugeben. Da dies die letzte in der Subroutine ausgeführte Anweisung ist, wurde auf die Verwendung von `return` verzichtet, obwohl `return -1` auch nicht falsch gewesen wäre.

Manche Programmierer bevorzugen es, jedes Mal, wenn es einen Rückgabewert gibt, `return` zu benutzen, da sich auf diese Weise der Code selbst dokumentiert. Das könnte zum Beispiel sinnvoll sein, wenn der Rückgabewert nicht in der letzten Codezeile der Subroutine steht, wie es bei `&ist_fred_oder_barney_groesser` weiter vorn in diesem Kapitel der Fall ist. Hier wird `return` nicht benötigt, tut aber auch niemandem weh. Manche Perl-Programmierer sehen darin allerdings nur sieben Zeichen, die man zusätzlich eingeben muss. Sie sollten also in der Lage sein, beide Arten von Code zu verstehen.

Das Ampersand-Zeichen weglassen

Wie versprochen, erklären wir Ihnen jetzt die Regeln, nach denen Sie bei einem Subroutinenaufruf das Ampersand-Zeichen (`&`, Kaufmanns-Und) weglassen können. Das ist immer dann der Fall, wenn die Subroutine vor ihrem Aufruf deklariert wird oder aus der Syntax klar erkennbar ist, dass es sich um einen Funktionsaufruf[21] handelt. So erkennt Perl eine Subroutine beispielsweise an einer in runden Klammern stehenden Parameterliste, die dem Namen der Routine nachgestellt ist. Ist mindestens eine dieser Bedingungen erfüllt, kann das &-Zeichen weggelassen werden. Die Subroutine kann nun wie eine eingebaute Funktion aufgerufen werden. (In diesen Regeln versteckt sich jedoch eine kleine Falle, wie Sie gleich sehen werden.)

```
my @karten = mischen(@kartenspiel); # &mischen kann ohne & aufgerufen werden
```

Wird die Subroutine in Ihrem Programm bereits vor ihrem Aufruf definiert, können Sie die runden Klammern um die Parameterliste weglassen:

```
sub division {
  $_[0] / $_[1];   # ersten Parameter durch zweiten dividieren
}

my $quotient = division 355, 113;  # &division benutzen
```

Das funktioniert, da runde Klammern weggelassen werden dürfen, solange es die Bedeutung des Codes nicht verändert.

Falls Sie die Deklaration der Subroutine jedoch *hinter* ihren Aufruf stellen, hat der Compiler keine Ahnung, was es mit dem versuchten Aufruf von `division` auf sich hat. Der Compiler muss die Definition vor dem Aufruf zu sehen bekommen, um einen Subroutinenaufruf wie eine eingebaute Funktion behandeln zu können. Sonst weiß der Compiler nicht, was er mit diesem Ausdruck anfangen soll.

21 In diesem Fall ist die Funktion die Subroutine `&mischen`. Wie wir gleich erläutern werden, könnte es sich dabei auch um eine eingebaute Funktion handeln.

Die Falle lauert jedoch woanders. Sie besteht nämlich darin, dass eine Subroutine denselben Namen wie eine eingebaute Funktion haben kann. Um Perl vor Verwechslungen zu bewahren, *muss* Ihre Subroutine in einem solchen Fall mit einem Ampersand-Zeichen aufgerufen werden. Dadurch stellen Sie sicher, dass tatsächlich die Subroutine aufgerufen wird und keine interne Funktion. Ohne Kaufmanns-Und können Sie die Subroutine *nur dann* aufrufen, wenn es keine eingebaute Funktion gibt, die denselben Namen trägt:

```
sub chomp {
  print "Mampf, mampf!\n";
}

&chomp;  # hier MUSS ein &-Zeichen benutzt werden
```

Ohne das Kaufmanns-Und hätten wir das eingebaute `chomp` aufgerufen, obwohl wir die Subroutine `&chomp` definiert haben. Die eigentliche Regel, nach der Sie vorgehen sollten, lautet also: Solange Sie nicht die Namen aller in Perl eingebauten Funktionen kennen, sollten Sie das &-Zeichen beim Aufruf Ihrer eigenen Funktionen *immer* benutzen (also ungefähr bei Ihren ersten hundert Programmen). Wenn Sie sehen, dass andere Leute das Ampersand-Zeichen in ihrem Code weggelassen haben, muss das kein Fehler sein, sondern kann bedeuten, dass es in Perl keine eingebaute Funktion mit demselben Namen gibt.[22]

Oft benutzen Programmierer *Prototypen*, wenn sie planen, ihre Subroutinen auf die gleiche Art wie Perls eingebaute Funktionen aufzurufen. Das geschieht oft beim Schreiben von *Modulen*. Prototypen sagen Perl, welche Parameter zu erwarten sind. Das Schreiben von Modulen ist ein Thema für Fortgeschrittene; Sie können sich aber, wenn Sie so weit sind, die Perl-Dokumentation (insbesondere die Dokumente *perlmod* und *perlsub*) zu den Themen Subroutinen, Prototypen und Module ansehen.[23]

Nicht-skalare Rückgabewerte

Als Rückgabewerte von Subroutinen können übrigens nicht nur Skalare verwendet werden. Wenn Sie eine Subroutine im Listenkontext[24] aufrufen, kann diese auch eine Werteliste zurückgeben.

Stellen Sie sich vor, Sie wollten einen Zahlenbereich ausgeben (wie der Bereichsoperator `..`), nur soll herauf- und heruntergezählt werden. Der Bereichsoperator kann nur aufwärts zählen, aber das lässt sich leicht beheben:

22 Oder es handelt sich *doch* um einen Fehler. Um herauszufinden, ob es sich um eine eingebaute Funktion handelt, können Sie die Dokumentationen zu *perlfunc* und *perlop* durchsuchen. Außerdem wird Perl Sie bei eingeschalteten Warnungen darauf hinweisen.

23 Oder Sie setzen Ihre Studien mit *Intermediate Perl* fort.

24 Mithilfe der Funktion `wantarray` können Sie feststellen, ob eine Subroutine im skalaren oder im Listenkontext ausgewertet wird. Auf diese Weise können Sie ohne Schwierigkeiten Subroutinen schreiben, die vom Kontext abhängige skalare oder Listenwerte zurück geben.

```
sub liste_von_fred_bis_barney {
  if ($fred < $barney) {
    # von $fred bis $barney aufwärts zählen
    $fred..$barney;
  } else {
    # von $fred bis $barney abwärts zählen
    reverse $barney..$fred;
  }
}
$fred = 11;
$barney = 6;
@c = &liste_von_fred_bis_barney;  # @c enthält (11, 10, 9, 8, 7, 6)
```

In diesem Fall gibt uns der Bereichsoperator eine Liste der Zahlen zwischen 6 und 11. Mit reverse wird diese Liste dann umgekehrt, so dass sie von $fred (11) bis $barney (6) geht – genau das, was wir wollten.

Der kleinste mögliche Rückgabewert hat keinen Inhalt. Eine return-Anweisung ohne Argumente gibt im skalaren Kontext undef zurück und im Listenkontext eine leere Liste. Das kann praktisch sein, wenn es einen Fehler in der Subroutine gibt. Auf diese Weise kann angezeigt werden, dass kein bedeutungsvollerer Rückgabewert zur Verfügung steht.

Persistente private Variablen (Zustandsvariablen)

Mit my haben wir private Variablen für Subroutinen definiert. Allerdings wurden diese bei jedem Aufruf der Subroutine erneut definiert. Mit der Funktion state lassen sich nun für Subroutinen private Variablen deklarieren, die zwischen den Aufrufen ihre Werte behalten.

Zu Beginn dieses Kapitels hatten wie eine Subroutine namens marine, die einen Wert um eins erhöht:

```
sub marine {
  $n += 1;  # globale Variable $n
  print "Hallo, Taucher Nummer $n!\n";
}
```

Da Sie mittlerweile wissen, wofür strict gut ist, können Sie es in der neuen Version der Subroutine direkt verwenden und stellen erst einmal fest, dass $n jetzt nicht mehr erlaubt ist. Allerdings können wir $n auch nicht einfach mit my als lexikalische Variable deklarieren, weil ihr Wert zwischen den Aufrufen verfallen würde.

Indem wir die Variable mit state deklarieren, teilen wir Perl mit, dass es sich um eine private Variable handelt, deren Wert Perl sich zwischen den Aufrufen merken soll. Dieses Feature gibt es seit Perl 5.10:

```
use 5.010;

sub marine {
  state $n = 0; # private persistente Variable $n
  $n += 1;
  print "Hallo, Taucher Nummer $n!\n";
}
```

Jetzt können Sie auch bei der Verwendung von `strict` die gleichen Ausgaben erzeugen, ohne dafür eine globale Variable verwenden zu müssen. Beim ersten Aufruf der Subroutine deklariert und initialisiert Perl `$n`. In den nachfolgenden Aufrufen der Subroutine wird diese Deklaration von Perl ignoriert. Zwischen den Aufrufen merkt sich Perl den Wert von `$n` für den nächsten Aufruf.

Mit `state` lassen sich beliebige Variablentypen deklarieren, nicht nur Skalare. Hier sehen Sie eine Subroutine, die eine Gesamtsumme aller Zahlen, die in einem mit `state` deklarierten Array enthalten sind, errechnet und ausgibt:

```
use 5.010;
gesamtsumme( 5, 6 );
gesamtsumme( 1..3 );
gesamtsumme( 4 );
sub gesamtsumme {
  state $summe = 0;
  state @zahlen;
  foreach my $zahl ( @_ ) {
    push @zahlen, $zahl;
    $summe += $zahl;
    }

  say "Die Gesamtsumme von (@zahlen) ist $summe";
  }
```

Bei jedem Aufruf wird eine neue Summe ausgegeben, indem die neuen Argumente zu den bereits vorhandenen hinzugezählt werden.

```
Die Gesamtsumme von (5 6) ist 11
Die Gesamtsumme von (5 6 1 2 3) ist 17
Die Gesamtsumme von (5 6 1 2 3 4) ist 21
```

Bei der Deklaration von Arrays und Hashes mit `state` gibt es in Perl 5.10 allerdings eine kleine Einschränkung. Beide Variablentypen können (noch) nicht im Listenkontext initialisiert werden:

```
state @array = qw(a b c); # Fehler!
```

Es wird eine Fehlermeldung angezeigt, die Ihnen einen Hinweis darauf gibt, was in zukünftigen Perl-Versionen vielleicht möglich sein wird:

```
Initialization of state variables in list context currently forbidden ...(Übersetzung:
Initialisierung von Zustandsvariablen ist im Moment noch verboten.)
```

Übungen

Die Antworten zu diesen Übungen finden Sie in Anhang A. Übungsantworten auf Seite 327

1. [12] Schreiben Sie eine Subroutine `&gesamt`, die die Summe einer Liste von Zahlen zurückgibt. Hinweis: Die Subroutine sollte dabei keine I/O-Operationen durchführen, sondern ihre Parameter abarbeiten und das Ergebnis an den Aufrufer zurückgeben. Probieren Sie es mit dem folgenden Beispielprogramm, das die Subroutine

aufruft, um die Funktionsweise zu testen. Die erste Gruppe von Zahlen sollte zusammen 25 ergeben.

```
my @fred        = qw{ 1 3 5 7 9 };
my $fred_gesamt = &gesamt(@fred);
print "Die Summe von \@fred ist $fred_gesamt.\n";
print "Geben Sie einige Zahlen jeweils auf einer eigenen Zeile ein: ";
my $benutzer_gesamt = &gesamt(<STDIN>);
print "Die Summe der von Ihnen eingegebenen Zahlen ist
       $benutzer_gesamt.\n";
```

2. [5] Benutzen Sie die Subroutine aus der vorigen Übung, um ein Programm zu schreiben, das die Summe der Zahlen von 1 bis 1.000 berechnet.
3. [18] Übung für Zusatzpunkte: Schreiben Sie eine Subroutine `&ueber_durchschnitt`, die eine Liste mit Zahlen übernimmt und diejenigen zurückgibt, die größer sind als der Durchschnittswert (arithmetisches Mittel). (Tipp: Schreiben Sie eine zusätzliche Subroutine zum Ermitteln des Durchschnittswertes, indem die Gesamtsumme durch die Anzahl der Elemente dividiert wird.) Testen Sie Ihre Subroutine mit diesem Testprogramm.

```
my @fred = &ueber_durchschnitt(1..10);
print "\@fred ist @fred\n";
print "(Sollte 6 7 8 9 10 ergeben)\n";
my @barney = &ueber_durchschnitt(100, 1..10);
print "\@barney ist @barney\n";
print "(Sollte nur 100 ausgeben)\n";
```

4. [10] Schreiben Sie eine Subroutine mit dem Namen `begruesse`, die jede Person persönlich begrüßt und ihr dabei den Namen des zuletzt Begrüßten mitteilt:

```
begruesse( "Fred" );
begruesse( "Barney" );
```

   Die Ausgaben sollten aussehen wie folgt:

```
Hallo Fred! Du bist der Erste hier!
Hallo Barney! Fred war auch gerade da!
```

5. [10] Ändern Sie das vorige Programm so ab, dass jeder neuen Person sämtliche Namen der bereits Begrüßten mitgeteilt werden:

```
begruesse( "Fred" );
begruesse( "Barney" );
begruesse( "Wilma" );
begruesse( "Betty" );
```

   Die Ausgaben sollten aussehen wir folgt:

```
Hallo Fred! Du bist der Erste hier!
Hallo Barney! Die folgenden Personen waren vor Dir hier: Fred
Hallo Wilma! Die folgenden Personen waren vor Dir hier: Fred Barney
Hallo Betty! Die folgenden Personen waren vor Dir hier: Fred Barney Wilma
```