3.17 Unterprogramme

3.17.1 Syntax

Definition gekennzeichnet durch **Schlüsselwort** `sub`:
`sub-marine.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
sub marine {
  $n++;  # globale Variable $n
  print "Moin, Matrose Nr. $n!\n";
}
&marine;
&marine;
&marine;
&marine;
```

▶▶ `./sub-marine.pl`

Aufruf gekennzeichnet durch **Kaufmanns-Und.** (Das Kaufmanns-Und kann man **oft auch weglassen**, aber es ist besser, es zu schreiben.)

Unterprogramme **müssen nicht vor** ihrer Benutzung **definiert** werden.

Unterprogramme haben wiederum einen **eigenen Namensraum**.

3.17.2 Rückgabewerte und der `return`-Operator

Unterprogramme haben **immer** einen **Rückgabewert**. Der **Wert** ist der **letzte Ausdruck, der berechnet** wurde:
`return-last.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
sub summe_von_hugo_und_karl {
  print "Berechnung der Summe von \$hugo und \$karl.\n";
  $hugo + $karl;
  #### print "Berechnung beendet.\n";
}
$hugo = 3;
$karl = 5;
$summe = &summe_von_hugo_und_karl;
print "Die Summe ist $summe.\n";
```

▶▶ `./return-last.pl`

Vorsicht dabei **mit Debugging**-Anweisungen!

▶▶ `./return-last.pl` ohne die Auskommentierung

Auch `print` hat einen **Rückgabewert**. (Er zeigt an, ob die Ausgabe erfolgreich war.)

Immerhin gibt `perl` eine **Warnmeldung**, wenn sie eingeschaltet sind.

Wenn ein Unterprogramm im **Listen-Kontext** berechnet wird, kann es **auch** eine **Liste zurückgeben**:

`return-list.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
sub liste_von_hugo_bis_karl {
  if ($hugo < $karl) {
    $hugo .. $karl;
  } else {
    $karl .. $hugo;
  }
}
$hugo = 17;
$karl = 9;
@liste = &liste_von_hugo_bis_karl;
print "Die Liste ist '@liste'.\n";
```

▶▶ `./return-list.pl`

Man sieht übrigens, daß der **letzte berechnete Ausdruck** zählt, **nicht** die **letzte Zeile** des Unterprogramms.

Mit der **Funktion** `wantarray` kann man **herausfinden, welcher Kontext** gerade aktuell ist:

`return-wantarray.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
sub hugo_bis_karl {
  if ($hugo < $karl) {
    if(wantarray) { $hugo .. $karl; }
    else          { $karl - $hugo; }
  } else {
    if(wantarray) { $karl .. $hugo; }
    else          { $hugo - $karl; }
  }
}
$hugo = 17;
$karl = 9;
@liste = &hugo_bis_karl;
print "Die Liste ist '@liste'.\n";
$differenz = &hugo_bis_karl;
print "Die Differenz ist $differenz.\n";
```

▶▶ `./return-wantarray.pl`

Eingebaute Funktionen wie `wantarray` werden **immer ohne Kaufmanns-Und** aufgerufen.

Mit dem **`return`-Operator** kann man sofort aus einem Unterprogramm zurückkehren: `return-operator.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub element_nummer {
  foreach (0..$#namen) {
    if ($namen[$_] eq $such_name) {
      return $_
    }
  }
  return;
}
$nummer = &element_nummer;
print "Die Nummer von $such_name ist $nummer.\n";
```

▶▶ `./return-operator.pl`

Wenn man dem `return`-Operator **kein Argument** mitgibt, liefert er `undef` zurück.

▶▶ `./return-operator.pl` mit modifiziertem `$such_name`

Im **Listen-Kontext** liefert er dann die **leere Liste**.

3.17.3 Parameter

Die **Kommunikation** von Werten **über globale Variablen** ist umständlich und **fehleranfällig**. Besser sind **Parameter** für die Funktionen.

Parameter werden **in runden Klammern** übergeben, wie bei den eingebauten Funktionen:
`return-operator2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub element_nummer {
  $mein_such_name = $_[0];
  shift @_;
  @meine_namen = @_;
  foreach $name (0..$#meine_namen) {
    if ($meine_namen[$name] eq $mein_such_name) {
      return $name
    }
  }
  return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name ist $nummer.\n";
```

▶▶ `./return-operator2.pl`

Das Unterprogramm **bekommt die Parameter im Array `@_`**.

Man kann auch eine **variable Anzahl von Parametern** übergeben, wie im obigen Beispiel. `@_` ist eine Liste, daher ist das kein Problem.

Beim Aufruf wird **der alte Inhalt** von **`@_` wegkopiert** und **beim Rücksprung zurückkopiert**. Analog wie bei `foreach`:
`return-operator3.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub string_ist_gleich {
  return($_[0] eq $_[1]);
}
sub element_nummer {
  $mein_such_name = $_[0];
  shift;  # arbeitet auf @_
  foreach $name (0..$#_) {
    if (&string_ist_gleich($_[$name], $mein_such_name)) {
      return $name
    }
  }
  return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name ist $nummer.\n";
```

►► `./return-operator3.pl`

Nach Ende des verschachtelten Unterprogramms wird die Variable `@_` restauriert, so daß sie **in der Schleife weiter benutzt** werden kann.

3.17.4 Private Variablen

Per Default sind **alle Variablen global**.

Mit dem **`my`-Operator** kann man **lexikalische** Variablen erzeugen, die **lokal zum aktuellen Block** sind, also z.B. zum aktuellen Unterprogramm:
`return-operator4.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
my @namen;
my $such_name;
my $nummer;
@namen = qw/ meyer mueller schulze /;
$such_name = 'mueller';
sub element_nummer {
  my $such_name;
  my @namen;
  my $name;
  $such_name = $_[0];
  shift @_;
  @namen = @_;
  $name = 0;
  while(@namen) {
    if($such_name eq pop @namen) {
      return $name;
    }
    $name++;
  }
  return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name in '@namen' ist $nummer.\n";
```

▶▶ `./return-operator4.pl`

Die lexikalischen Variablen des Unterprogramms **gelten** hier **bis zur schließenden Klammer**.

Die **am Anfang definierten lexikalischen Variablen** gelten für den **Rest der Datei**, da vorher kein Ende des Blocks kommt.

Wie man sieht, wird das **Array** `@namen` **der obersten Ebene nicht** durch das Unterprogramm **modifiziert**.

Man kann die **Definition** und die **erste Wertzuweisung zusammenfassen**. Man kann außerdem **mehrere Variablen gleichzeitig** deklarieren:
`return-operator5.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
my @namen = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub element_nummer {
  my ($such_name, @namen) = @_;
  foreach $name (0..$#namen) {
    if ($namen[$name] eq $such_name) {
      return $name
    }
  }
  return;
}
my $nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name ist $nummer.\n";
```

►► `./return-operator5.pl`

Jetzt muß die Liste der Variablen in **runden Klammern** stehen.

Fast **alle Unterprogramme fangen so an**.

Ein paar **Details zum Scoping**:
`my-scope.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
while(defined(my $zeile = <>)) {
  print $zeile;
  if((my $antwort = <STDIN>) =~ /^good/i) {
    print "    Great!\n";
  } elsif ($antwort =~ /^bad/i) {
    print "    Oh, well..!\n";
  } else {
    chomp $antwort;
    print "**** '$antwort' is neither 'good' nor 'bad'.\n";
    exit 1;
  }
}
```

►► `./my-scope.pl my-scope.pl`

Der **Scope** von `$zeile` reicht **von** der **Schleifenbedingung** über die **ganze while-Schleife**.

Der **Scope** von `$antwort` reicht **von** der **if-Bedingung über alle Blöcke** des if-elsif-else-Konstruktes.

Das **if-elsif-else-Konstrukt** ist hier **neu**. Es tut, was man denkt, daß es tut. Man beachte das **fehlende „e“**!.

Ein paar **Details zur Aufrufsemantik**: Die Listen-Elemente von `@_` werden per Referenz übergeben. Man kann also die Werte der aufrufenden Funktion verändern:
`param-ref.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
my ($name1, $name2) = qw/ meyer mueller /;
sub upcase_in {
  for (@_) { tr/a-z/A-Z/ }
}
upcase_in($name1, $name2);
print "Die Namen sind '$name1' und '$name2'.\n";
```

▶▶ `./param-ref.pl`

Die Funktion `upcase_in` ändert also alle **übergebenen Variablen in Großbuchstaben**.

Es wird die Funktion `tr///` benutzt, auch `y///` genannt, die eine **zeichenweise Ersetzung** durchführt. Sie ist mit `s///` verwandt, aber ersetzt das erste Zeichen des ersten Musters durch das erste Zeichen des zweiten Musters usw.

Aber Achtung, man kann sich auch unerwünschte Effekte einhandeln:
`param-ref2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
my @namen = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub string_ist_gleich {
  return($_[0] eq $_[1]);
}
sub element_nummer {
  my $such_name = $_[0];
  shift;  # arbeitet auf @_
  foreach $name (0..$#_) {
    if (&string_ist_gleich($_[$name], $such_name)) {
      $_[0] = 'YYYYYYYYYYY';
      return $name
    }
    $_[$name] = 'XXXXXXXXX';
  }
  return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name in '@namen' ist $nummer.\n";
```

▶▶ `./param-ref2.pl`

Die **Aufrufparameter** werden hier unerwartet **zerstört**.

Daher ist es gut, die **Parameter mit** `my` zu **kopieren**. Dann hat man **„call by value“**, nicht „call by reference“, und es kann nichts passieren:
`param-ref3.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
my @namen = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub string_ist_gleich {
  return($_[0] eq $_[1]);
}
sub element_nummer {
  my $such_name = $_[0];
  shift;
  my @namen;
  foreach (@_) {
    @namen = (@namen, $_);
  }
  foreach $name (0..$#namen) {
    if (&string_ist_gleich($namen[$name], $such_name)) {
      $such_name = 'YYYYYYYYYYY';
      return $name
    }
    $namen[$name] = 'XXXXXXXXX';
  }
  return;
}
$nummer = &element_nummer($such_name, @namen);
print "Die Nummer von $such_name in '@namen' ist $nummer.\n";
```

▶▶ `./param-ref3.pl`

Man beachte, daß wir `@_` **elementweise kopieren mußten**. Hätten wir die ganze Liste kopiert, wären nur die Referenzen auf die Elemente kopiert worden.

Auch wenn man `@_` einen **ganz neuen Wert zuweist**, passiert **keine Modifikation**, weil nur die Elemente der Liste Referenzen sind, nicht die Liste selbst.

`perl` erlaubt **allgemein** auch die Verwendung von **Referenzen auf Variablen**, was eine Art **sicherer Ersatz für Pointer** ist. Aber das ist ein **fortgeschrittenes Thema**, auf das ich nicht eingehen werde.

3.17.5 Das Pragma `use strict`

Es ist **guter Programmierstil**, **jede Variable explizit** zu definieren.

Das schützt vor Schreibfehlern. Ein einzelner Tippfehler wird durch die `-w`-**Warnungen** erkannt. Aber wenn man die **falsche Schreibweise mehrfach** verwendet, **hilft** das **nicht**. Das folgende ist falsch:

`return-operator6-wrong.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
my @namen_liste = qw/ meyer mueller schulze /;
my $such_name = 'mueller';
sub element_nummer {
  my ($such_name, @namen) = @_;
  foreach $name (0..$#namen) {
    if ($namen[$name] eq $such_name) {
      return $name
    }
  }
  return -1;
}
my $nummer = &element_nummer($such_name, @namenliste);
print "Die Nummer von $such_name in '@namenliste' ist $nummer.\n";
```

▶▶ `./return-operator6-wrong.pl`

Die falsche Schreibweise wird zweimal verwendet, dehalb erkennt `-w` das nicht. Die Deklaration der **richtigen Schreibweise** mit `my` wird **auch nicht beanstandet**, weil die Variable gleichzeitig auch **initialisiert** wird.

▶▶ `./return-operator6-wrong.pl` mit `use strict;` vorneweg

Erst jetzt wird die Fehlerursache gemeldet.

▶▶ `./return-operator6-wrong.pl` mit eingefügten Underscores

Faustregel: Für Programme, die länger als ein Schirm voll sind, sollte man **immer** `use strict` **verwenden**.

3.17.6 Parameter-Prototypen

Es ist auch möglich, **Prototypen für die Parameter** von Unterprogrammen zu definieren. Man kann damit z.B. festlegen, ob ein Skalar oder eine Liste erwartet wird (**skalarer oder Listen-Kontext**).

Die Syntax und Semantik sind aber **komplex,** und das ganze lohnt sich **nur, wenn** man **eigene** `perl`**-Module** schreibt. Daher lasse ich das Thema hier aus.

3.18 Weitere Kontrollstrukturen

Es gibt in `perl` eine Reihe von **alternativen Schreibweisen**, die sich oft besser lesen lassen.

3.18.1 Die `unless`-Anweisung

Manchmal ist es natürlicher, nicht `if(!...)` zu schreiben, sondern `unless`(...):
`unless.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
unless ($nenner == 0) {
  print "$zaehler / $nenner = ", ($zaehler / $nenner), "\n"
}
```

▶▶ `./unless.pl`

Man darf dabei auch `else` verwenden:
`unless-else.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
unless ($nenner == 0) {
  print "$zaehler / $nenner = ", ($zaehler / $nenner), "\n"
} else {
  print "Division durch Null abgefangen!\n";
}
```

▶▶ `./unless-else.pl`

Aber **Vorsicht**: unless-else **kann** auch **verwirrend** sein.

3.18.2 Die `until`-Anweisung

Manchmal ist es natürlicher, nicht `while(!...)` zu schreiben, sondern `until(...)`:
`until.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
my $verhaeltnis = 0;
until ($zaehler < $nenner) {
  $zaehler -= $nenner;
  $verhaeltnis++;
}
print "Das ganzzahlige Verhaeltnis ist $verhaeltnis\n"
```

▶▶ `./until.pl`

3.18.3 Ausdruck-Modifikatoren

Man darf an das Ende eines einzelnen Ausdrucks einen Modifikator schreiben, was die Notation kompakter macht. Beispiel:
`unless-mod.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
chomp (my $zaehler = <STDIN>);
print "Nenner = ";
chomp (my $nenner = <STDIN>);
print "$zaehler / $nenner = ", ($zaehler / $nenner), "\n"
  unless ($nenner == 0);
```

▶▶ `./unless-mod.pl`

Hier **sieht** die Modifikator-Version auch **natürlicher aus** als die längere Variante mit geschweiften Klammern.

Die **Bedeutung** ist genau die **gleiche** wie vorher.

Ausnahme: Es beginnt **kein neuer Scope**.

Und es gibt hier **kein** `else`.

Das ganze geht auch für andere Konstrukte. Beispiel:
`if-mod.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $tracing = 1;
my $x = 5;
print "\$x ist gleich '$x'\n"          if $tracing;
```

▶▶ ./if-mod.pl

Weiterhin gibt es es als **Modifikatoren**: `until`, `while` und `foreach`:
`mod-misc.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $n = 2;
$n *= 2 until $n > 10;
print " ", ($n -= 2) while $n > 0;
print "\n";
print " $_" foreach (1..5);
print "\n";
```

▶▶ ./mod-misc.pl

Der **Operator** `*=` multipliziert die Variable auf der linken mit dem Wert auf der Rechten. Wir suchen also die kleinste Zweierpotenz größer zehn.

Obwohl die **Schleifenbedingungen** jetzt **hinten** stehen, werden sie **trotzdem vor der Schleife** ausgeführt.

Man beachte, daß **vor dem Modifikator kein Semikolon** steht.

Damit es nicht gar zu kompliziert wird, ist es **nicht erlaubt, an einen Modifikator** noch einen **weiteren Modifikator hinten dran** zu schreiben. Ebenso kann man den Modifikator **nur an einen einzelnen Ausdruck** dran schreiben. Sonst muß man die normale Schreibweise nehmen.

Bei dem `foreach`-Modifikator kann man **keine andere Laufvariable** angeben, man muß mit `$_` arbeiten.

3.18.4 „Nackte“ Blöcke

Wenn man von einer normalen **while-Schleife** das Schlüsselwort und die Bedingung **wegnimmt**, bekommt man einen **nackten Block**. Er wird genau **einmal ausgeführt**.

Sinnvoll ist das z.B., um einen **beschränkten Scope** einzuführen:
`naked-block.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $bytenum;
{
  print "Anzahl der Bits = ";
  my $bitnum = <STDIN>;
  $bytenum = $bitnum >> 3;
  if ($bitnum % 8) {
    $bytenum++;
  }
}
print "Die Anzahl der noetigen Bytes ist $bytenum.\n";
```

▶▶ `./naked-block.pl`

Der **Scope** der Variablen `$bitnum` ist hier **sofort wieder zuende**, sobald sie nicht mehr gebraucht wird.

Der **Scope** einer Variablen sollte immer **so klein wie möglich** sein, um Programmierfehler zu vermeiden.

3.18.5 Die `elsif`-Anweisung

Die if-elsif-else-Anweisung hatten wir **bereits oben** bei den Details zum Scoping schon **eingeführt**.

3.18.6 Autoinkrement und Autodekrement

Das Post-Autoinkrement haben wir im Zusammenhang mit **Hashes schon gesehen**.

Es gibt **vier Operatoren**, die genau **wie in C** arbeiten:
`auto-inc.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $var = 1;
print "\$var == $var.\n";
print "++\$var == ", ++$var, ".\n";
print "\$var == $var.\n";
print "\$var++ == ", $var++, ".\n";
print "\$var == $var.\n";
#
print "--\$var == ", --$var, ".\n";
print "\$var == $var.\n";
print "\$var-- == ", $var--, ".\n";
print "\$var == $var.\n";
```

▶▶ `./auto-inc.pl`

Der **Auto-Inkrement-Operator** hat etwas zusätzliche **Magie**: Dies gilt nur, wenn die Variable **niemals als Zahl** verwendet worden ist. Wenn sie also ein String ist, und wenn sie nicht der leere String ist, und wenn sie sie auf das Muster

```
/^[a-zA-Z]*[0-9]*$/
```

paßt, dann wird zeichenweise, mit Übertrag, als **String inkrementiert**:
`auto-inc2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my @strings = qw/ aa Az a0 99 zz /;
foreach my $var (@strings) {
  print "\$var = $var\n";
  print "++\$var == ", ++$var, "\n";
}
```

▶▶ `./auto-inc2.pl`

Dies ist **nützlich**, wenn man **fortlaufend neue Dateierweiterungen** generieren möchte.

Die **Dekrementoperatoren** sind **nicht magisch**.

3.18.7 Schleifensteuerung

Die `foreach`-Anweisung zum **Iterieren über eine Liste** ist bereits **bekannt**. Eine andere, **äquivalente Schreibweise** ist `for`.

Mit beiden kann man wie mit der `for`-Schleife von C iterieren:
`for-loop.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
for(my $idx = 1; $idx <=5; $idx++) {
  print "$idx\n";
}
#
my $idx = 1;
while($idx <= 5) {
  print "$idx\n";
  $idx++;
}
```

▶▶ `./for-loop.pl`

Die for-Schleife läßt sich wie gezeigt **in** eine **while-Schleife auflösen**.

Ein kleiner **Unterschied** bleibt: Bei der for-Schleife bleibt eine **my-Deklaration** in der Initialisierung **lokal**.

Man kann an eine while-Schleife auch einen `continue`-Block anhängen. Er hat die gleiche **Wirkung** wie der **dritte Teil einer for-Schleife**:
`continue-loop.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
for(my $idx = 1; $idx <=5; $idx++) {
  print "$idx\n";
}
#
my $idx = 1;
while($idx <= 5) {
  print "$idx\n";
} continue {
  $idx++;
}
```

▶▶ `./continue-loop.pl`

Das ist **wichtig**, wenn man die **Schleife vorzeitig neu starten** will, was mit den folgenden Konstrukten geht.

Der `next`-Operator erlaubt, **vorzeitig** zur **nächsten Iteration** überzugehen:
`next-loop.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
for(my $idx = 1; $idx <=10; $idx++) {
  if($idx % 2) { next; }
  print "$idx\n";
}
#
my $idx = 1;
while($idx <= 10) {
  if($idx % 2) { next; }
  print "$idx\n";
} continue {
  $idx++;
}
```

▶▶ `./next-loop.pl`

Bei einer **for-Schleife** wird sofort der **dritte Teil der Klammer** ausgeführt. Bei einer **while-Schleife** wird sofort der **continue-Block** ausgeführt, sofern vorhanden. In beiden Fällen kommt **danach** die **Schleifenbedingung**. Dies **entspricht** dem **continue-Konstrukt von C**.

Ein **Schleifenkonstrukt,** und auch ein nackter Block, kann **mit** einem **Label benannt** werden. Damit kann man **mehrere Blöcke auf einmal verlassen**:

`last-loop2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
ZEILE:
for(my $zeile = 1; $zeile <= 9; ((print "\n"), $zeile++)) {
SPALTE:
  foreach my $spalte (1..9) {
    next ZEILE if $spalte > $zeile;
    print "$spalte ";
  }
}
```

▶▶ `./last-loop2.pl`

Ein Label wird per Konvention in **Großbuchstaben** geschrieben. Es wird von einem **Doppelpunkt** gefolgt.

Man beachte, wie wir **zwei Anweisungen in** den **dritten Teil** der for-Schleife gepackt haben. Eine Alternative wäre eine while-continue-Schleife gewesen.

Der `last`-Operator dient dazu, eine **Schleife vorzeitig zu verlassen**:

last-loop.pl:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $idx;
for($idx = 1; $idx <=10; $idx++) {
  print "$idx\n";
  if($idx == 5) { last; }
}
print "Index nach der Schleife: $idx\n";
#
$idx = 1;
while($idx <= 10) {
  print "$idx\n";
  if($idx == 5) { last; }
} continue {
  $idx++;
}
print "Index nach der Schleife: $idx\n";
```

▶▶ ./last-loop.pl

Der **dritte Teil der for-Schleife** und der **continue-Block** werden dann **nicht** mehr ausgeführt.

Auch dies geht **über mehrere Ebenen**:
last-loop3.pl:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my @namen1 = qw/ meyer mueller schulze /;
my @namen2 = qw/ lehmann mueller feldmann /;
NAMEN1:
foreach my $name1 (@namen1) {
  NAMEN2:
  foreach my $name2 (@namen2) {
    if($name1 eq $name2) {
      print "Der Name '$name1' kommt in beiden Listen vor.\n";
      last NAMEN1;
    }
  }
}
```

▶▶ ./last-loop3.pl

Dies kann man **in C** mit continue **nicht mehr** machen!

Der `redo`-Operator dient dazu, an den **Anfang der aktuellen Iteration zurückzugehen**. Der dritte Iterations- bzw. der continue-Block wird dabei **nicht ausgewertet**. Auch die **Schleifenbedingung** wird **nicht nochmal ausgewertet**. Man braucht den Operator, wenn man die selbstgeschriebene Schleife belügen will:
`redo-loop.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
while (<>) {
   chomp;
   if (s/\\$//) {
       $_ .= <>;
       redo;
   }
   print "Die aktuelle Zeile ist: '$_'\n";
}
```

`redo-loop.txt`:

```
Dies ist die erste Zeile.
Zweite ... \
dritte ... \
und dies ist die vierte Zeile.
```

▶▶ `./redo-loop.pl redo-loop.txt`

Hier wird **zeilenweise eingelesen** und wieder ausgedruckt. **Ausnahme** ist, wenn eine Zeile auf **Backslash** endet. dann wird diese Zeile **mit der nächsten vereinigt**, ggf. auch mehrfach.

3.18.8 Der `do`-Block

Der `do`-Block verhält sich zunächst **fast** genau **wie** ein **nackter Block**. Wenn er durch einen **Schleifen-Modifikator** verändert wird, wird der **Block immer erst einmal ausgeführt**, bevor die **Schleifenbedingung** geprüft wird:
`do-block.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $var;
do {
  print "Eingabe = ";
  chomp ($var = <>);
} while($var != 0);
print "Ende der Schleife.\n";
do {
  print "Eingabe = ";
  chomp ($var = <>);
} until($var == 0);
print "Ende der Schleife.\n";
```

▶▶ `./do-block.pl`

Der `do`-Block **zählt nicht als Schleife**, so daß man `next`, `last` und `redo` **nicht** darin **anwenden** kann. Man kann **in ihn** (für `next`) oder um ihn (für `last`) aber einen **nackten Block** schreiben, der als Schleife zählt.

3.18.9 Logische Operatoren

Es gibt die logischen Operatoren `&&` (und), `||` (oder) und `!` (nicht):
`log-op.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
for $a (0, 1) {
  for $b (0, 1) {
    print "$a && $b == '", ($a && $b), "'\n";
  }
}
print "\n";
for $a (0, 1) {
  for $b (0, 1) {
    print "$a || $b == '", ($a || $b), "'\n";
  }
}
print "\n";
for $a (0, 1) {
  print "! $a == '", (! $a ), "'\n";
}
```

▶▶ `./log-op.pl`

Die **Werte** sind die **erwarteten**. Allerdings liefert 1! den **leeren String** zurück, was aber auch False ist.

Die logischen Operatoren `&&` und `||` werden **von links nach rechts ausgewertet**. Was passiert, wenn das **Gesamtergebnis schon nach dem ersten Ausdruck klar** ist?

Die Auswertung erfolgt **nur soweit notwendig**, also **teilweise**:
`log-op2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Bitte sag 'ja': ";
if( defined ($_ = <>) && /^ja/i ) {
  print "\nDanke!\n";
} else {
  print "\nDann nicht.\n";
}
```

▶▶ `./log-op2.pl`

Hier wird der **rechte Teil nur** ausgewertet, **wenn** `$_` auch **definiert** ist.

Der logische Ausdruck hat **nicht nur** einen **logischen Wert**, sondern einen ganz **konkreten**: Den des **letzten ausgewerteten Ausdrucks**. Das ist **nützlich**, um z.B. einen **Default-Wert** zu wählen:
`log-op3.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Nachname: ";
my $zeile = <>;
defined $zeile && chomp $zeile;
my $nachname = $zeile || '(Kein Nachname angegeben)';
print "Eingegebener Nachname: $nachname\n";
```

▶▶ `./log-op3.pl`

Man beachte, **daß jeder Wert**, der **False ergibt**, durch den **Default-Wert ersetzt** wird, auch z.B. `0`.

Manchmal muß man **viele Klammern** verwenden, weil die **Präzedenz** der logischen Operatoren zu hoch ist:
`log-op4.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $a = '';
my $result;
($result = $a) || ($result = '(nichts)');
print "Ergebnis = $result\n";
```

▶▶ ./log-op4.pl

Anmerkung: Hier hätten wir natürlich auch `$result` nach links herausziehen können.

Für dieses Problem gibt es die logischen Operatoren `and`, `or` und `not`, die genauso funktionieren, aber eine ganz niedrige Präzedenz haben. Außerdem sind sie besser zu lesen:
`log-op5.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $a = '';
my $result;
$result = $a or $result = '(nichts)'; # ausserdem gibt's: and, not
print "Ergebnis = $result\n";
```

▶▶ ./log-op5.pl

3.18.10 Der Bedingungsausdruck ?:

Aus C übernommen wurde der **Bedingungsausdruck**.
`cond-op.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
my $a = 5;
my $b = 7;
my $max = $a > $b ? $a : $b;
print "Das Maximum von $a und $b ist $max.\n";
```

▶▶ ./cond-op.pl

Auch er wird nur **teilweise ausgewertet**:
`cond-op2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Zaehler = ";
my $zaehler = <>;
print "Nenner = ";
my $nenner = <>;
my $verhaeltnis = $nenner != 0 ? $zaehler / $nenner : 0;
print "Das Verhaeltnis ist $verhaeltnis.\n";
```

►► `./cond-op2.pl`

3.18.11 Die `case/switch`-Anweisung

In `perl` **gibt** es **keine** `case/switch`-Anweisung. Aber es gibt verschiedene Wege, sie **elegant zu simulieren**. Man nimmt dafür z.B. einen nackten Block:
`switch.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Eingabe [Ja/Nein/Weiss nicht] = ";
$_ = <>;
SWITCH: {
  /^ja/i && do { print "Fein!\n"; last SWITCH;};
  /^Nein/i && do { print "Schade!\n"; last SWITCH;};
  /^Weiss nicht/i && do { print "Merkwuerdig!\n"; last SWITCH;};
  print "Illegale Eingabe!\n"; exit 1;
}
print "Normales Programmende.\n";
```

►► `./switch.pl`

Die **do-Blöcke** sind **hier notwendig**, weil sie einen Wert liefern, während ein nackter Block eine separate Anweisung gewesen wäre.

Man kann **auch** den **Bedingungsausdruck** verwenden:
`switch2.pl`:

```
#! /usr/bin/perl -w
use diagnostics;
use strict;
print "Groesse als Zahl = ";
my $zahl = <>;
my $groesse =
  $zahl < 10 ? 'klein' :
  $zahl < 20 ? 'mittel' :
  $zahl < 50 ? 'gross' :
               'uebergross';
print "Die Groesse ist $groesse.\n";
```

►► ./switch2.pl