

weiteren Zeilen des Skripts liest und ausführt. Möglich wird diese Schreibweise durch die Unix-Shell selbst, die beim Starten eines Skripts dessen erste Zeile auf das Vorkommen des oben beschriebenen Pseudokommentars untersucht.

Um das neue Skript `umfang1` ausführen zu können, muß die Datei lediglich noch mit Hilfe des Kommandos `chmod` ausführbar gemacht werden (dieser Schritt wird bei allen folgenden Beispielen nicht mehr gesondert beschrieben):

```
mr@hamlet> chmod 755 umfang1
mr@hamlet> ls -l umfang1
mr@hamlet> -rwxr-xr-x 1 mr staff 360 Dec 5 08:24 umfang1
mr@hamlet> umfang1
Bitte geben Sie den Radius des Kreises ein:
2.5
Der Durchmesser des Kreises ist: 15.70796327
```

2.3 VARIABLEN

Wie jede gebräuchliche Programmiersprache kennt auch Perl Variablen zur Speicherung von Werten (Zahlen, Text, ...). Eine Variable ist nichts anderes als die Zuordnung eines Namens zu einem Wert. Um anschließend den Wert z.B. in einer Berechnung verwenden zu können, kann einfach der diesem Wert zugeordnete Name, also die Variable, an dieser Stelle eingesetzt werden. Die einmal gemachte Zuordnung kann jederzeit (in einer Zuweisung) wieder verändert werden. Die Zuweisung eines Werts an eine Variable erfolgt mit Hilfe des `=`-Operators in der Form `variable=wert;`.

Variablen haben in Perl keinen Typ, d.h. eine Variable darf beispielsweise sowohl Zeichenketten, ganze Zahlen als auch Gleitpunktzahlen enthalten. Darüber hinaus müssen Variablen in Perl nicht deklariert oder definiert werden, wie es in allen höheren Programmiersprachen z.B. in C/C++ oder Pascal notwendig ist. Eine Variable ist automatisch ab ihrer ersten Verwendung definiert. Perl unterscheidet drei Arten von Variablen, die durch verschiedene Symbole vor dem Variablennamen unterschieden werden: skalare Variablen (z.B. `$var`), Arrays (z.B. `@var`) und Hashs (z.B. `%var`), auch assoziative Arrays genannt.

SKALARE VARIABLEN

Eine skalare Variable wird in Perl mit dem `$`-Zeichen vor dem Variablennamen gekennzeichnet. Sie darf wie jede der in Perl bekannten Variablenarten sowohl Zahlen als auch Zeichenketten enthalten, wie folgendes Beispiel demonstriert:

```
#!/usr/bin/perl
# Datei: vartest1
$x=10; # Ganzzahl
$hallo="Hallo hier bin ich"; # Zeichenkette
$y=12.34567; # Gleitkommazahl

print "$x $hallo $y ->$neu<- \n";

mr@hamlet> vartest
10 Hallo hier bin ich 12.34567 -><<-
```

Wie in diesem Beispiel zu sehen ist, erhält eine Variable wie `$neu`, die verwendet wird, ohne das zuvor ein Wert zugewiesen wurde, automatisch den Wert `undef` als Wert, der bei der Ausgabe mit `print` automatisch zu einer leeren Zeichenkette umgewandelt wird, ohne daß ein Fehler oder eine Warnung gemeldet wird.

SPEZIELLE FUNKTIONEN FÜR SKALARE VARIABLEN

Die Perl-Standardbibliothek verfügt über eine große Zahl an Funktionen, die auf skalare Variablen angewendet werden können. An dieser Stelle sollen die wichtigsten kurz erläutert werden.

In Perl gibt es einen Unterschied zwischen einer Variablen, die keinen Wert enthält, etwa nach einer Anweisung der Form `$x=""` und einer Variablen, die nicht existiert, also noch nicht verwendet wurde. Diese Unterscheidung kann mit der Funktion `defined $variable` getroffen werden. Diese Funktion liefert einen Wert ungleich 0, wenn eine Variable existiert (auch wenn diese keinen Wert enthält), und die leere Zeichenkette, wenn die Variable nicht existiert. Entsprechend existiert eine Funktion `undef $variable`, mit der die angegebene Variable gelöscht wird.

Insbesondere bei der Verarbeitung von Zeichenketten ist es wichtig, die Länge einer Zeichenkette bestimmen zu können und Zugriff auf einen Teil der Zeichenkette zu haben. In Perl kann die Funktion `length` dazu verwendet werden, die Zahl der Zeichen in einer Zeichenkette zu bestimmen. Die Anweisungen

```
$x="12345";
print length($x);
```

bewirken beispielsweise die Ausgabe der Zahl 5, da die Variable `$x` fünf Zeichen enthält.

Der Zugriff auf einen Teil einer Zeichenkette kann mit der Funktion `substr` durchgeführt werden. Diese Funktion hat folgende Syntax:

```
substr string, offset [, length]
```

Der Parameter `string` ist eine Variable oder eine Zeichenkette, aus der `length` Zeichen ab der Zeichenposition `offset` kopiert werden. Das erste Zeichen hat dabei die Position 0. Wird der optionale Parameter `length` weggelassen, werden ab `offset` alle Zeichen bis zum Ende von `string` kopiert und als Ergebnis geliefert. Hat `length` einen negativen Wert, werden entsprechend viele Zeichen vom Ende der Zeichenkette `string` nicht kopiert. Schließlich ist es auch möglich, eine Zuweisung an einen `substr`-Ausdruck vorzunehmen, sofern `string` keine Konstante darstellt. In diesem Fall ersetzt der zugewiesene Wert die Zeichenkette in `string`, die durch die Parameter `offset` und `length` beschrieben ist. Je nachdem, ob die ersetzte Zeichenkette länger oder kürzer als die ursprüngliche Zeichenkette ist, wächst oder schrumpft die Zeichenkette in `string`.

Ebenfalls interessant für die Verarbeitung von Zeichenketten ist die Funktion `index`, mit der die Position einer Zeichenkette in einer anderen festgestellt werden kann. Die Funktion hat folgende Syntax:

```
index string, substr [, offset]
```

Der Parameter `string` enthält die zu durchsuchende Zeichenkette, der Parameter `substr` ist die zu suchende Zeichenkette. Der optionale Parameter `offset` bestimmt, ab welcher Position in `string` nach `substr` gesucht werden soll. Wird der Parameter weggelassen, beginnt die Suche am Anfang von `string`.

Folgendes Beispiel verdeutlicht die Arbeitsweise der Funktionen `index` und `substr`. In einer Zeichenkette `$str` soll die Position bestimmt werden, an der das Wort "einfach" steht. Anschließend soll es durch das Wort "mehrfach" ersetzt werden:

```
#!/usr/bin/perl
# Datei: str
$str="Perl ist einfach gut ...";
$search="einfach";
print "$str\n";
$start=index $str, $search;# Best. Start des Worts "einfach"
print "einfach steht ab Position $start \n";
substr $str, $start, length($search) = "mehrfach"; # Ersetzen
print "$str \n";          # Ausgabe des veraenderten Strings

mr@hamlet> str
Perl ist einfach gut ...
einfach steht ab Position 9
Perl ist mehrfach gut ...
```

In diesem Beispiel wurde die Funktion `index`¹ verwendet, um die Position der Zeichenkette "einfach" in `$str` zu bestimmen. Anschließend wird diese Zeichenkette mit Hilfe von `substr` durch den Text "mehrfach" ersetzt.

Zwei weitere oft verwendete Funktionen sind `chomp` und `chop`. Die Funktion `chomp` entfernt das Zeilenumbruchzeichen vom Ende einer Zeichenkette. Enthält die Zeichenkette kein solches Zeichen am Ende, bewirkt `chomp` nichts. Im Gegensatz dazu entfernt `chop` *immer* das letzte Zeichen einer Zeichenkette, unabhängig davon, um welches Zeichen es sich handelt. Als Ergebnis liefern die Funktionen das entfernte Zeichen.

Mit Hilfe dieser Funktionen hätte beispielsweise nach der Eingabe des Radius-Werts im Skript-Beispiel zur Berechnung des Umfangs aus dem letzten Abschnitt das Zeichen `\n` vom Ende der Eingabe leicht entfernt werden können:

```
#!/usr/bin/perl
# Datei: str1
print "Bitte geben Sie den Radius ein: ";
$radius=<STDIN>; # Eingabe des Radius durch den Benutzer
print "Vor chomp: ", length($radius), "\n";
chomp $radius; # oder einfacher: chomp( $radius=<STDIN> );
print "Nach 1. chomp: ", length($radius), "\n";
chomp $radius; # Nochmal probieren
print "Nach 2. chomp: ", length($radius), "\n";
chop $radius; # Jetzt mal chop aufrufen
print "Nach chop:", length($radius), "\n";
print "radius: $radius\n";

mr@hamlet> str1
Bitte geben Sie den Radius ein: 12
Vor chomp: 3
Nach 1. chomp: 2
Nach 2. chomp: 2
Nach chop:1
radius: 1
```

Wie an der Ausgabe deutlich wird, bewirkt der erste `chomp`-Aufruf das Entfernen des am Ende von `$radius` stehenden `\n`-Zeichens. Ein weiterer Aufruf von `chomp` hat keine Auswirkungen mehr, im Gegensatz zu einem Aufruf von `chop`, der bewirkt, daß ein weiteres Zeichen vom Ende der Zeichenkette `$radius` entfernt wird. `$radius` enthält anschließend nur noch das Zeichen `1`.

Eine weitere Funktion aus der Perl-Standardbibliothek ist `reverse`. Mit Hilfe der Funktion `reverse` ist es möglich, eine Zeichenkette umzudrehen. Die so veränderte Zeichenkette wird als Ergebnis dieser Funktion geliefert. Die als Parameter übergebene Zeichenkette wird nicht verändert:

¹ Neben der Funktion `index` existiert auch die Funktion `rindex`, die fast identisch mit `index` ist. Der Unterschied besteht darin, daß `index` eine Zeichenkette von links nach rechts durchsucht, während `rindex` umgekehrt, also von rechts nach links sucht.

```
#!/usr/bin/perl
# Datei str2
$str="abcde";
$str=reverse $str;
print $str, "\n";

mr@hamlet> str2
edcba
```

Wie in der Ausgabe zu erkennen ist, wurde die Zeichenkette in `$str` Zeichen für Zeichen umgedreht.

ARRAYS

Arrays sind ein wesentlicher Bestandteil von Perl. Im Gegensatz zu anderen Sprachen, bei denen Arrays stets sehr statischer Natur sind, stellen Arrays in Perl einen sehr flexiblen und leistungsfähigen Datentyp dar.

Arrays sind Variablen, die nicht nur einen Wert enthalten dürfen, sondern beliebig viele, wobei zur Unterscheidung jedem in einem Array enthaltenen Wert ein Index (eine Zahl) zugeordnet wird. Ein Array kann man sich gut als Tabelle vorstellen. Jede Zeile der Tabelle enthält dabei genau einen Wert. Die Nummer der Zeile ist der Index des Arrays, der bei Null beginnt.

Möchte man auf das Array als Ganzes zugreifen, also nicht auf ein einzelnes Element, wird vor dem Variablennamen ein `@` angegeben. Möchte man jedoch auf ein bestimmtes Element eines Arrays zugreifen, muß der Variablenname mit vorangestelltem `$`-Zeichen und dem Index des Elements in `[]` angegeben werden.

```
#!/usr/bin/perl
# Datei: arrayintro
$day[0]="Sonntag";
$day[1]="Montag";
$day[2]="Dienstag";
print $day[1], "\n";
print @day, "\n";

mr@hamlet> arrayintro
Montag
SonntagMontagDienstag
```

Eng mit dem Begriff des Arrays verbunden ist der Begriff der Liste. Eine Liste ist eine unbenannte Aufzählung von Werten, die durch Kommata voneinander getrennt werden. Die Liste beginnt mit einer runden öffnenden Klammer und endet mit einer runden schließenden Klammer. Beispielsweise ist `("Sonntag", "Montag", "Dienstag")` eine Liste in

Perl. () ist die leere Liste. Die Initialisierung des Arrays @day aus obigem Beispiel hätte man mit Hilfe einer Liste einfacher wie folgt schreiben dürfen:

```
#!/usr/bin/perl
@day=("Sonntag", "Montag", "Dienstag");
print $day[1], "\n"; # Ausgabe des Werts: Montag
```

Die Elemente der Liste werden durch diese Anweisung einfach von links nach rechts an das Array @day zugewiesen, wobei das erste Element in \$day[0] abgelegt wird, da Arrays in Perl normalerweise bei Null beginnen. Wichtig ist in diesem Fall, daß die Zuweisung an das Array @day erfolgt. Die Verwendung von \$day=(...) hätte zu einem anderen Ergebnis geführt, da \$day eine andere Variable als @day ist, die zudem kein Array, sondern eine einfache skalare Variable darstellt:

```
#!/usr/bin/perl
# Datei arrays
@day=("Sonntag", "Montag", "Dienstag"); # Array
$day=("Januar", "Februar", "Maerz"); # Skalare Variable

print $day[2], "\n";
print $day, "\n";

mr@hamlet> arrays
Dienstag
Maerz
```

Wie das Beispiel zeigt, sind \$day und @day verschiedene Variablen. Das Ergebnis der ersten Ausgabe ist nicht weiter überraschend. Die Ausgabe von \$day[2] bewirkt die Ausgabe der Zeichenkette Dienstag, da sie das dritte Element (von Null an gezählt) des Arrays ist. Die zweite Ausgabe zeigt, daß es sich bei @day und \$day tatsächlich um zwei verschiedene Variablen handelt. Da \$day eine skalare Variable ist und somit nur einen Wert speichern kann, wurden bei der Zuweisung der Liste an \$day zwar nacheinander alle Elemente der Liste an \$day zugewiesen, erhalten blieb aber nur das letzte Element, da eine Liste ja von links nach rechts bearbeitet wird.

Zuweisungen an Listen können nicht nur Konstanten wie die Zeichenketten "Montag", etc. enthalten, sondern auch skalare Variablen und sogar Arrays. Da diese Tatsache recht verwirrend erscheinen muß, hier einige Beispiele zur Verdeutlichung:

```
# Verwendung von skalaren Variablen in Listen
$siebenundvierzigelf=4711;
@number=(4710, $siebenundvierzigelf, 4712);
```

```
# Verwendung eines Arrays in einer Liste
@work=("Mo", "Di", "Mi", "Do", "Fri");
@week=("Sun", @work, "Sat");

# Zuweisung eines Arrays an eine Liste und umgekehrt
@zahlen=(1, 2, 3, 4, 5);
($z1, $z2, $z3)=@zahlen;
```

Im letzten Beispiel werden alle Elemente eines Arrays nacheinander an die in der Liste enthaltenen Variablen zugewiesen, so daß im Anschluß an die Zuweisung `$z1` den Wert 1, `$z2` den Wert 2 und `$z3` den Wert 3 enthält. Die überzähligen Elemente 4 und 5 in `@zahlen` werden wie erwartet bei der Zuweisung nicht beachtet. Sollen diese Werte ebenfalls zugewiesen werden, kann als letztes Element der Liste wiederum ein Array angegeben werden, das alle restlichen, noch nicht zugewiesenen Elemente erhält:

```
# Zuweisung eines Arrays an eine Liste mit Array
@zahlen=(1, 2, 3, 4, 5);
($z1, $z2, $z3, @rest)=@zahlen;
print @rest, "\n"; # Ausgabe von: 4 5
```

Die Ausgabe `print @rest;` bewirkt die Ausgabe aller Elemente des Arrays `@rest` (unformatiert und ohne Trennzeichen zwischen den Elementen), dem alle überzähligen Elemente zugewiesen wurden. Da die ersten drei Elemente aus `@zahlen` an `$z1...$z3` zugewiesen wurden, erhält `@rest` die Zahlen 4 und 5. Nach der Zuweisung enthält `$rest[0]` also den Wert 4 und `$rest[1]` den Wert 5.

Oftmals ist es wichtig, feststellen zu können, wieviele Elemente ein Array hat. Perl stellt zu diesem Zweck die Schreibweise `$#array` zur Verfügung, wobei `array` eine Array-Variable sein muß. Das Ergebnis beschreibt den Index des letzten Elements des Arrays. Da Arrays bei Null beginnen, ist die tatsächliche Zahl der Elemente des Arrays genau um eins größer als das Ergebnis von `$#array`. Auf ein leeres Array angewendet, liefert der Operator die Zahl -1. Auch hierzu ein Beispiel:

```
#!/usr/bin/perl
# Datei: arraysize
@numbers=(0..9); # entspricht: @numbers=(0,1,2,3,4,5,6,7,8,9);
print $#numbers, "\n";

mr@hamlet> arraysize
9
```

Der erste Wert der oben verwendeten Liste wird in `$numbers[0]` abgelegt, der letzte Wert in `$numbers[9]`. Daher ist das Ergebnis des Ausdrucks `$#numbers` die Zahl 9, obwohl das Array zehn Elemente umfaßt. Zur Erzeugung der zehn Elemente wurde im obigen Beispiel der Bereichsope-

rator `..` verwendet, der einfach eine Liste mit Elementen erzeugt, die mit dem links angegebenen Element beginnt, mit dem rechts angegebenen Element beendet wird und alle dazwischen liegenden Werte beinhaltet¹.

Eine weitere interessante Möglichkeit, die Perl in Bezug auf Arrays bietet, sind sogenannte Array-Slices, also Teile eines Arrays. Folgendes Beispiel zeigt, was damit gemeint ist:

```
#!/usr/bin/perl
@array=("a", "b", "c", "d", "e");
($first, $second)=$array[2,3];
```

Die Anweisung `$array[2,3]` liefert als Ergebnis einen Array-Slice, also einen Ausschnitt aus dem Array, der aus den Elementen mit dem Index 2 und 3 besteht. Diese beiden Elemente werden den Variablen `$first` und `$second` zugewiesen. Anstelle der Aufzählung der Indexwerte zur Bestimmung des Teil-Arrays kann ebenso der Bereichsoperator verwendet werden. Um beispielsweise den Elementen mit dem Index 2, 3 und 4 einen anderen Wert zuzuweisen, könnte folgende Anweisung verwendet werden:

```
$array[2..4]=("charly", "delta", "echo");
```

Diese Zuweisung ist identisch mit folgendem Programmfragment:

```
$array[2]="charly";
$array[3]="delta";
$array[4]="echo";
```

Auf eine Liste darf wie auf ein Array der Index-Operator `[]` angewendet werden, um ein oder mehrere Elemente der Liste auszuwählen:

```
# Zuweisungs des dritten Elements einer Liste
$element=(1, 2, 3, 4)[2];    # Zugriff auf: 3
# Zuweisung von zwei Elementen einer Liste
@twoels=(1, 2, 3, 4)[2,3];  # Zugriff auf: 3 und 4
```

Wie in diesem Beispiel ersichtlich, kann der Index-Operator `[]` wie gewohnt auf eine Liste angewendet werden. Der erste Aufruf selektiert das dritte Element der Liste (mit dem Index 2) und weist den enthaltenen Wert der Variablen `$element` zu. Im zweiten Aufruf wird eine Array-Slice, bestehend aus zwei Elementen, selektiert und dem Array `@twoels` zugewiesen.

¹ Dieser Operator ist nicht auf Gleitkommazahlen anwendbar, wohl jedoch auf Zeichen. So ergibt z.B. `(a..z)` eine Liste mit allen Kleinbuchstaben.

SPEZIELLE ARRAY-FUNKTIONEN

Perl kennt eine Reihe von Funktionen, die speziell für die Bearbeitung von Arrays bestimmt sind. Die wichtigsten Array-Funktionen heißen `push`, `pop`, `shift`, `unshift`, `splice` und `reverse`.

Die Funktionen `push` und `pop` haben folgende Syntax:

```
push array, liste
pop array
```

Das Argument `array` stellt ein Perl-Array dar. Der Parameter `liste` ist eine beliebige Perl-Liste. Die Wirkung der Funktion `push` auf das Array `array` ist die, daß die angegebene Liste an das Ende des Arrays angehängt wird. Das Array wird in diesem Fall also als Stapel betrachtet, der um die Elemente der Liste erhöht wird. Die Funktion `pop` hat die gegenteilige Wirkung. Die Anwendung von `pop` auf ein Array entfernt das letzte (oberste) Element des Arrays, das wie bei `push` als Stapel betrachtet wird. Das Ergebnis der Funktion `pop` ist das Element, das entfernt wurde. Folgendes Beispiel verdeutlicht die Wirkungsweise der beiden Kommandos:

```
#!/usr/bin/perl
# Datei: pushpop
#
# Zuweisung eines Arrays an eine Liste
@monate=('Jan', 'Feb');

push @monate, ('Mar', 'Apr', 'Mai', 'Jun', 'Jul');
push @monate, ('Aug');

@rest=('Sep', 'Okt', 'Nov', 'Dez');

push @monate, @rest;

# Ausgabe des Index des letzten Elements in @monate
print '$#monate ist: ', $#monate, "\n";
print '@monate enthaelt: ', @monate[0..$#monate], "\n";

print "Ergebnis eines Aufrufs von pop:\n";
print pop @monate, "\n";
print '@monate enthaelt: ', @monate[0..$#monate], "\n";

mr@hamlet> pushpop
$#monate ist: 11
@monate enthaelt: JanFebMarAprMaiJunJulAugSepOktNovDez
Ergebnis eines Aufrufs von pop:
Dez
@monate enthaelt: JanFebMarAprMaiJunJulAugSepOktNov
```

Die erste Ausgabe des Skripts zeigt, daß das letzte Element des Arrays `@monate` den Index 11, das Array insgesamt also zwölf Einträge (0..11) hat. In der zweiten Zeile wird deutlich, daß die Aufrufe der Funktion `push` die angegebenen Elemente einfach an das Ende des Arrays anfügt. Ein Aufruf von `pop` liefert und entfernt das letzte Element dieses Arrays, wie die letzten beiden Zeilen der Beispielausgabe zeigen.

Neben `push` und `pop` stellen `unshift` und `shift` zwei weitere wichtige Funktionen zur Bearbeitung von Arrays dar. Die Funktionen `unshift` und `shift` arbeiten ähnlich wie `push` und `pop`, mit dem Unterschied, daß Elemente nicht am Ende des Arrays eingefügt/gelöscht werden, sondern an dessen Anfang. Die Syntax dieser Kommandos lautet wie folgt:

```
unshift array, liste
shift array
shift
```

Die Parameter `array` und `liste` stehen wiederum für ein beliebiges Perl-Array und eine Perl-Liste. Die Funktion `unshift` fügt die angegebene Liste `liste` am Beginn des Arrays `array` ein. Alle bisherigen Elemente von `array` werden einfach nach hinten verschoben.

Die Funktion `shift` bewirkt, daß das erste Element (mit dem Index 0) aus dem angegebenen Array entfernt wird und alle anderen Elemente um eine Position nach vorne rutschen. Das Element, das bisher z.B. unter dem Index 1 im Array zu finden war, liegt im Anschluß an die `shift`-Operation auf Index 0. Die Funktion liefert das entfernte Element als Ergebnis. Wird `shift` auf ein leeres Array angewendet, ist das Ergebnis der Wert `undef`, also „nichts“. Wird kein Array angegeben, wird automatisch das spezielle Array `@ARGV`¹ bearbeitet, das alle Parameter enthält, die auf der Kommandozeile an das Perl-Skript übergeben wurden.

Ein kleines Beispiel zur Verdeutlichung:

```
#!/usr/bin/perl
#
# Datei shiftunshift

print '$#ARGV ist: ', $#ARGV, "\n";
print '@ARGV enthaelt: ', @ARGV[0..$#ARGV], "\n";

shift;
print "Nach shift Operation:\n";
print '$#ARGV ist: ', $#ARGV, "\n";
print '@ARGV enthaelt: ', @ARGV[0..$#ARGV], "\n";

unshift @ARGV, (1, 2, 3);
```

¹ Im Gegensatz zu anderen Programmiersprachen enthält `$ARGV[0]` nicht den Programmnamen. Auf diesen kann über die Variable `$0` zugegriffen werden.

```

print "Nach unshift \@ARGV, (1, 2, 3) Operation:\n";
print '$#ARGV ist: ', $#ARGV, "\n";
print '@ARGV enthaelt: ', @ARGV[0..$#ARGV], "\n";

mr@hamlet> shiftunshift a b c
$#ARGV ist: 2
@ARGV enthaelt: abc
Nach shift Operation:
$#ARGV ist: 1
@ARGV enthaelt: bc
Nach unshift @ARGV, (1, 2, 3) Operation:
$#ARGV ist: 4
@ARGV enthaelt: 123bc

```

Beim Start des Skripts werden drei Parameter auf der Kommandozeile übergeben, die von Perl automatisch in das Array `@ARGV[0..2]` abgelegt werden. Daher ist der Index des letzten Elements, der mit `$#ARGV` ausgegeben wird, 2. Das Ausführen der Funktion `shift` ohne Parameter bewirkt das Entfernen des ersten Elements. Eine anschließende Ausführung der `unshift`-Funktion mit der Liste `(1, 2, 3)` als Parameter bewirkt, daß diese drei Elemente am Anfang des Arrays eingefügt werden.

Obwohl `shift` speziell zur Bearbeitung des Arrays `ARGV` ohne Parameter aufgerufen werden darf, gilt das nicht für die Funktion `unshift`. Hier muß immer das zu bearbeitende Array sowie eine Liste angegeben werden.

Aufmerksamen Lesern wird aufgefallen sein, daß im letzten Beispiel die auszugebenden Zeichenketten bis auf eine Ausnahme in einfache Apostrophe (') gefaßt wurden. Diese Apostrophe verhindern eine Auswertung der Zeichenkette durch Perl. Daher wird beispielsweise in der Anweisung

```
print '$#ARGV ist: ', $#ARGV, "\n";
```

der Text `$#ARGV ist: 4` ausgegeben und nicht etwa `4 ist 4`, d.h. das links in einfachen Apostrophen geschriebene `$#ARGV` wird als reine Zeichenkette betrachtet, und nicht als Anweisung zur Ermittlung des Index des letzten Elements des Arrays `@ARGV`. Auch Escape-Sequenzen wie z.B. `\n` (siehe Tabelle 1 auf Seite 9) werden nicht ausgewertet, wenn sie in einfachen Apostrophen stehen. In der drittletzten Zeile des vorangegangenen Perl-Skripts wurden doppelte Anführungszeichen verwendet, da hier am Ende der Zeile ein `Neue-Zeile-Zeichen(\n)` ausgegeben werden sollte und nicht die beiden Zeichen `\` und `n`. Da der Text aber die Variable `@ARGV` enthält, deren Name als Text ausgegeben werden sollte (im Gegensatz zur Ausgabe des Werts der Variablen), wurde einfach das `@`-Zeichen des Variablennamens mit einem `\` versehen (d.h. gequotet). Dadurch erkennt Perl diesen Namen in der `print`-Anweisung nicht mehr als Variablennamen und ersetzt die Zeichenkette `@ARGV` nicht durch den Wert der Variablen `@ARGV`. Diese Vorgehensweise wird als Quoten bezeichnet. Die genauen Regeln des Quotens in Perl werden in Kapitel 2.6 auf Seite 46 besprochen.

Eine weitere Array-Funktion, die hier besprochen wird, ist `splice`. Mit Hilfe von `splice` ist es möglich, beliebige Ausschnitte eines Arrays auszuschneiden oder durch eine Liste zu ersetzen. Die Syntax dieser Funktion lautet wie folgt:

```
splice array, index, anzahl, liste
splice array, index, anzahl
splice array, index
```

Der Parameter `array` steht für ein Perl-Array, `index` für einen Index eines Elements in einem Array, `anzahl` für die Zahl der zu bearbeitenden Elemente beginnend ab `index` und `liste` für eine beliebige Perl-Liste.

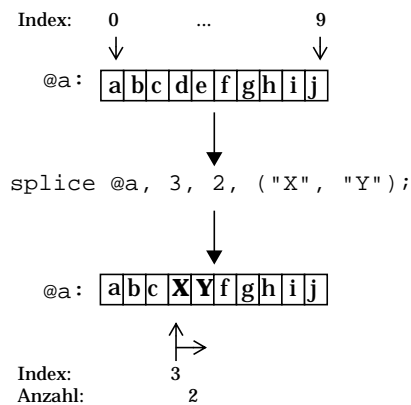


Abb. 2: Die Arbeitsweise von `splice`

Die Wirkungsweise von `splice` in der ersten oben genannten Form ist die, daß `anzahl` Elemente von `array` beginnend mit `index` durch die Elemente aus `liste` ersetzt werden. Das Array wird dabei automatisch kleiner oder größer, je nachdem, ob mehr Elemente eingefügt als ausgeschnitten werden. In Abb. 2 auf Seite 24 ist die Wirkungsweise der Funktion `splice` auf ein Array `@a` noch einmal graphisch dargestellt.

Wird beim Aufruf von `splice` der Parameter `liste` weggelassen, so werden einfach die durch `index` und `anzahl` bestimmten Elemente aus dem Array gelöscht (durch die leere Liste ersetzt). Fehlt auch die Angabe von `anzahl`, werden alle Elemente ab `index` des Arrays entfernt. Das Funktionsergebnis besteht aus einer Liste der entfernten Elemente. Auch zu dieser Funktion ein Beispiel:

```
#!/usr/bin/perl
#
# Datei: splice
#
```

```

# Definition des Arrays @x
@x=("a", "b", "c", "d", "e", "f", "g", "h", "i", "j");
print "x:      ", @x, "\n";

# splice mit vier Parametern
splice @x, 3, 2, ("X", "Y");
print 'Nach: splice @x, 3, 2, ("X", "Y")', "\n";
print "x:      ", @x, "\n";

# splice mit drei Parametern
splice @x, 6, 3;
print 'Nach: splice @x, 6, 3', "\n";
print "x:      ", @x, "\n";

# splice mit zwei Parametern
@ergeb=splice @x, 2;
print 'Nach: @ergeb=splice @x, 2', "\n";
print "x:      ", @x, "\n";
print "ergeb: ", @ergeb, "\n";

mr@hamlet> splice
x:      abcdefghij
Nach: splice @x, 3, 2, ("X", "Y")
x:      abcXYfghij
Nach: splice @x, 6, 3
x:      abcXYfj
Nach: @ergeb=splice @x, 2
x:      ab
ergeb: cXYfj

```

Der erste `splice`-Aufruf bewirkt, daß die Elemente mit dem Index 3 und 4 aus dem Array `@x` durch die Zeichen "X" und "Y" ersetzt werden. Die Größe des Arrays bleibt in diesem Fall unverändert. Im zweiten Aufruf wird keine Liste von einzusetzenden Werten angegeben. In diesem Fall werden die ausgewählten Elemente (Indexwerte: 6, 7 und 8) des Arrays einfach gelöscht, wodurch es um drei Elemente kürzer wird. Der letzte Aufruf von `splice` erfolgt mit nur zwei Parametern, wodurch alle Elemente ab dem angegebenen Index (2) gelöscht werden. Das letzte Beispiel zeigt darüber hinaus, daß die `splice`-Funktion als Ergebnis die ersetzten bzw. gelöschten Zeichen des Arrays liefert auf das die Funktion angewendet wird.

Die letzte hier zu besprechende Funktion ist `reverse`. Bei der Anwendung der Funktion auf eine skalare Variable bewirkt sie die Umkehrung der als Parameter angegebenen Zeichenkette. Wird diese Funktion auf ein Array angewendet, hat sie eine vergleichbare Wirkung. Ein Aufruf der Funktion `reverse` mit einem Array als Parameter bewirkt, daß alle Elemente dieses Arrays in umgekehrter Reihenfolge als Ergebnis geliefert werden, so daß das zuvor an letzter Stelle des Arrays stehende Element

anschließend am Anfang steht und umgekehrt. Folgendes Beispiel zeigt die Wirkung:

```
#!/usr/bin/perl
# Datei: reverse
#
@a=("123", "456", "789"); # Array mit drei Elementen
print @a, "\n";          # Vor dem Aufruf
@b=reverse @a;           # Array umdrehen
print @b, "\n";          # Nach dem Aufruf

mr@hamlet> reverse
123456789
789456123
```

Wie zu erkennen ist, wurden nicht die Elemente an sich umgekehrt, sondern deren Position im Array. Daher steht das letzte Element anschließend an erster Stelle und das zuvor an erster Stelle stehende Element anschließend am Ende des Arrays. Das Array `@a` selbst wird nicht verändert.

Bei der Arbeit mit einem Array ist es oftmals notwendig, einige bzw. alle Elemente z.B. nach einem bestimmten Wert zu durchsuchen. Eine Möglichkeit hierzu ist die Verwendung einer Schleife über alle Elemente des Arrays. Dieser Ansatz wird in Abschnitt *Die foreach-Anweisung* auf Seite 82 dargestellt. Eine weitere Möglichkeit ist die Verwendung der Perl-Funktionen `grep` und `map`.

Die in Anlehnung an das Unix-Kommando `grep` benannte Perl-Funktion `grep` wendet einen Ausdruck oder einen Block von Anweisungen nacheinander auf jedes Element eines Arrays an und liefert als Ergebnis eine Liste der Elemente des Arrays, für die der Ausdruck TRUE war. Man kann sagen, `grep` durchsucht das Array nach Werten, für die der angegebene Ausdruck wahr ist. In dem verwendeten Ausdruck wird an `$_` nacheinander jeder der im Array enthaltenen Werte zugewiesen. Eine Veränderung an `$_` bewirkt eine Veränderung des entsprechenden Array-Werts. Im folgenden ein erstes Beispiel, bei dem die Aufgabe darin besteht, genau die Werte in `@a` auszugeben, die größer als die Zahl 10 sind:

```
#!/usr/bin/perl
# Datei: grepl
#
@a=(12, 5, 99, 4);
@b=grep( $_ > 10, @a);
print @b, "\n";

mr@hamlet> grepl
1299
```

Der in diesem Beispiel angegebene Ausdruck (`$_ > 10`) ist nur dann wahr, wenn der numerische Wert von `$_` größer als 10 ist (siehe Abschnitt *Vergleichsoperatoren* auf Seite 60). Die `grep`-Funktion wendet nun diesen Ausdruck nacheinander auf alle in `@a` enthaltenen Werte an und liefert genau die Werte als Ergebnis, die größer als 10 sind, also die Zahlen 12 und 99. Die so ermittelten Zahlen werden in `@b` gespeichert, das anschließend ausgegeben wird. Da die Ausgabe unformatiert erfolgt, stehen die beiden Zahlen in der Ausgabezeile ohne Trennzeichen direkt nebeneinander.

Anstelle des im letzten Beispiel verwendeten Ausdrucks hätte auch ein Block von Anweisungen stehen dürfen, der in geschweifte Klammern gefaßt wird. Die `grep`-Anweisung hätte dann wie folgt ausgesehen:

```
@b=grep {$_ > 10} @a;
```

Der Vorteil eines solchen Blocks in einer `grep`-Anweisung ist, daß er nicht nur eine, sondern mehrere Perl-Anweisungen enthalten darf, die zur Bearbeitung der Array-Werte verwendet werden können.

In der Ausgabe des letzten Beispiel stört ein wenig, daß die Zahlen direkt nebeneinander, also ohne Trennzeichen ausgegeben werden. Mit Hilfe von `grep` ist es leicht möglich, alle Werte eines Arrays zeilenweise auszugeben:

```
#!/usr/bin/perl
# Datei: grep2
#
@a=(12, 5, 99, 4);
@b=grep {$_ > 10} @a;      # Suche nach Werte > 10 in @a
grep {print $_, "\n"} @b; # Ausgabe aller Werte in @b

mr@hamlet> grep2
12
99
```

In diesem Beispiel werden mit dem ersten `grep`-Aufruf alle Elemente aus `@a` gesucht, die größer als 10 sind. Die gefundenen Zahlen werden in `@b` gespeichert. Mit Hilfe des zweiten `grep`-Aufrufs werden nacheinander alle Werte aus `@b` ausgegeben, wobei die Ausgabe mit einem Zeilenumbruch abgeschlossen wird. Der zweite `grep`-Aufruf demonstriert zugleich die Möglichkeit, anstelle einer Bedingung zur Selektion bestimmter Array-Elemente eine einfache Perl-Anweisung zu setzen, um beispielsweise die Ausgabe der Elemente zu ermöglichen.

Neben der Möglichkeit, die Elemente eines Arrays durch einen Test auszuwerten, ist es durch Zuweisung an `$_` ebenfalls möglich, die Werte des Arrays direkt zu modifizieren, wie das folgende Beispiel zeigt:

```
#!/usr/bin/perl
# Datei: grep3
```

```

#
@a=(1, 2, 3);
@b=grep {$_ = $_*2} @a; # Multipliziere jeden Wert in @a * 2
print $a[0], "\n", $a[1], "\n", $a[2], "\n";
print @b, "\n";

mr@hamlet> grep3
2
4
6
246

```

In diesem Beispiel werden alle Werte des Arrays `@a` mit 2 multipliziert. Dies erfolgt durch Zuweisung des Ergebnisses der Multiplikation im `grep`-Aufruf an `$_`. Da das Ergebnis dieses Multiplikationsausdrucks ungleich 0 ist, also `TRUE`, enthält nach der Ausführung der `grep`-Anweisung auch `@b` die neuen Werte.

In den vorangegangenen Beispielen wurde das Ergebnis von `grep` einer Array-Variablen wie `@b` zugewiesen, d.h. das Ergebnis wurde als Liste behandelt. Wird das Ergebnis eines `grep`-Aufrufs in einem skalaren Kontext behandelt, also z.B. einer skalaren Variable zugewiesen, liefert `grep` eine Zahl, die ausdrückt, wie oft der angegebene Ausdruck `TRUE` war:

```

#!/usr/bin/perl
# Datei: grep4
#
@a=(12, 5, 99, 4, 2);
$b=grep {$_ > 10} @a;
print $b, "\n";

mr@hamlet> grep4
2

```

Die Ausgabe 2 besagt, daß der Ausdruck `$_ > 10` insgesamt zweimal `TRUE` war, also das insgesamt zwei der Elemente in `@a` größer als 10 sind.

Neben `grep` existiert eine weitere Funktion mit dem Namen `map`, die es wie `grep` erlaubt, alle Elemente eines Arrays zu bearbeiten. Der wesentliche Unterschied zwischen `grep` und `map` besteht darin, daß `grep` nur die Werte des behandelten Arrays als Ergebnis liefert, für die der gewählte Ausdruck `TRUE` ist. `grep` liefert also i.d.R. nur einige der Elemente eines Arrays als Ergebnis. Der Rückgabewert eines Aufrufs von `map` hingegen besteht aus einer Liste *aller* Werte, die das Ergebnis der Anwendung des verwendeten Ausdrucks auf die Array-Elemente sind:

```

#!/usr/bin/perl
# Datei: map1
#
@a=(a, b, c, d);

```

```

@b=map {($_,$_)} @a;
#
print @b, "\n";

mr@hamlet> map1
aabbccdd

```

In diesem Beispiel werden nacheinander alle Werte des Arrays `@a` durch den gezeigten Ausdruck `($_,$_)` bearbeitet. Das Ergebnis dieses Ausdrucks, das in diesem Fall aus einer Liste mit zwei Elementen besteht, wird an `@b` zugewiesen. Wie bei `grep` dient `$_` als Alias für das gerade betrachtete Array-Element. Der Ausdruck `($_,$_)` bewirkt daher die Duplikation des jeweiligen Elements. Da genau diese Liste für jedes Array-Element als Ergebnis geliefert wird, bewirkt die obige `map`-Anweisung eine Verdopplung aller Elemente des Arrays.

Während `grep` also eher für das Suchen in einem Array von Elementen mit bestimmten Eigenschaften dient, eignet sich `map` eher für die Bearbeitung aller Array-Elemente. Ein letztes Beispiel zeigt eine weitere Anwendung von `map`:

```

#!/usr/bin/perl
# Datei: map2
#
@a=("a", "b", "c");
@b=map {ord($_)} @a;

print $b[0], "\n", $b[1], "\n", $b[2], "\n";

mr@hamlet> map2
97
98
99

```

Die Perl-Funktion `ord` liefert für ein Zeichen dessen ASCII (bzw. Uni)-Code. Die Wirkung des oben dargestellten `map`-Kommandos besteht also in der Umwandlung der in `@a` gespeicherten Zeichen in deren Code. Das Ergebnis stellt dementsprechend den ASCII-Code für die Zeichen `a`, `b` und `c` dar.

HASHS (ASSOZIATIVE ARRAYS)

Neben dem im vorangegangenen Abschnitt eingeführten Typ des Arrays verfügt Perl über einen weiteren Datentyp, dem sogenannten Hash. Oftmals wird dieser Typ auch als assoziatives Array bezeichnet.

Wie schon der Name assoziatives Array sagt, handelt es sich bei diesem Typ um eine Form des Arrays. Eine Variable dieses Typs kann also nicht nur einen Wert enthalten, sondern (nur durch den zur Verfügung stehen-

den Speicherplatz begrenzt) beliebig viele. Bei einem einfachen Array erfolgt der Zugriff über den Namen einer Array-Variablen und der Angabe des Index des gewünschten Elements. Der Index muß in diesem Fall immer eine Zahl sein, die in `[]` gefaßt hinter dem Variablennamen steht, wie z.B. `$x[3]`.

Ein Hash stellt eine Erweiterung des Array-Begriffs dar, indem der Index, der zur Auswahl eines der in dem Array enthaltenen Elemente dient, nicht unbedingt eine Zahl sein muß, sondern eine beliebige Zeichenkette sein kann. Der Zugriff auf ein bestimmtes Element in einem assoziativen Array erfolgt daher durch die Angabe des Variablennamens (mit vorangestelltem `$`) und der Indexzeichenkette, die in geschweiften Klammern hinter dem Variablennamen steht. Der Index wird auf diese Weise dem Wert zugeordnet, es entsteht eine Assoziation zwischen Index und Wert, daher die Namensgebung *assoziatives Array*. Der Index wird bei assoziativen Arrays üblicherweise als *key* (Schlüssel) bezeichnet.

Der Zugriff auf das Array als Ganzes erfolgt durch Voranstellen des `%`-Zeichens vor den Variablennamen. Ein einfaches Beispiel soll dies verdeutlichen:

```
#!/usr/bin/perl
# Datei: hashintro
$months{"Januar"}=31;
$months{"Februar"}=28;
$months{"Maerz"}=31;
#
print $months{"Februar"}, "\n";
print $months{"xyz"}, "\n";
print %months, "\n";
```

```
mr@hamlet> hashintro
28
```

```
Januar31Maerz31Februar28
```

Wie in dem Beispiel zu sehen ist, dienen die Monatsnamen als Index für das Hash `%months`. Jedem Monat kann auf diese Weise die Anzahl der Tage in diesem Monat zugeordnet werden. Verwendet man bei der Ausgabe eines Werts aus dem Array einen Index, dem kein Wert zugeordnet ist, wie in der Anweisung `print $months{"xyz"}, "\n";`, wird als Wert die leere Zeichenkette ausgegeben.

Wird versucht, das gesamte Hash auszugeben, werden nacheinander alle Werte mit den zugehörigen Indizes ausgegeben. Dabei ist die Reihenfolge jedoch nicht unbedingt identisch mit der Reihenfolge der Initialisierung, was an der internen Darstellung eines Hashs durch Perl liegt. Daher sind Ausgaben dieser Art in der Regel nicht zweckmäßig, insbesondere weil es leistungsfähigere Funktionen gibt, um alle Elemente eines Hashs zu verarbeiten (siehe Seite 32).