Kapitel 4

perl: Practical Extraction and Report Language

4.1 Einführung

perl stellt eine weitere Programmiersprache dar. Sie ist aus der Erkenntnis entstanden, daß man in awk einige Dinge nur schwer tun kann, da awk zu stark an das Konzept der Bearbeitung von Records gebunden ist. Andererseits sind z.B. in **C** Pattern-Matching Operationen schwierig. In der Shell hingegen ist richtiges Rechnen wiederum nicht effizient möglich.
Deshalb wurde perl als ein Gemisch von Konzepten aus allen diesen Programmen entworfen. Darin liegt aber, jedenfalls in meinen Augen, auch eine seiner größten Schwächen: Die Konzepte sind im Gemisch teilweise sehr unübersichtlich. Man sollte daher darauf verzichten, perl als Sprache für größere Projekte zu verwenden[1]. Auch die in der perl-Version 5 aufgestülpten Objektorientierten Features ändern an der Ungeeignetheit für größere Projekte nichts.
In diesem Kapitel werden zunächst die grundlegenden Entitäten, Variablen, Ausdrücke und Kommandos erläutert. Danach werden die von perl verwendeten Pattern, die sich stark von den awk-Pattern unterscheiden, vorgestellt. Die in perl vorhandene Möglichkeit, Unterprogramme zu definieren wird in Abschnitt 4.5 diskutiert. Im darauf folgenden Abschnitt werden einige spezielle Variablen aufgelistet. Abschnitt 4.7 stellt die Verwendung von perl zur Realisierung von CGI-Scripten vor. Zum Abschluß folgen noch einige Beispiele und Warnungen zur Verwendung von perl.

[1]"Write once - read never"

4.2 Variablen & Ausdrücke

Für Variablen `perl` kennt drei verschiedene Datentypen:

1. Skalare. Ein Skalar kann entweder ein String sein (`"Ein String"`) oder eine Zahl. Ähnlich wie `awk` konvertiert `perl` dabei automatisch Strings zu Zahlen und umgekehrt, je nachdem, was es an der jeweiligen Stelle braucht. Eine Zahl wird in `perl` immer als eine Fließkommazahl mit doppelter Genauigkeit gespeichert.

2. Vektoren von Skalaren. Vektoren sind Listen von Skalaren, also Arrays, die mit Zahlen indiziert werden.

3. Assoziative Vektoren von Skalaren. Dies sind auch Arrays, nur das sie (wie in `awk`) assoziativ sind, also durch Strings indiziert werden.

Als Typ eines Ausdruckes kann allerdings immer nur 'Skalar' oder 'Liste von Skalaren' auftreten. Vektoren und assoziative Vektoren werden dabei zu Listen konvertiert, indem ihre Werte aufgezählt werden, bzw. die Aufzählung in der Form $\mathtt{\textit{Schlüssel}_1,\ \textit{Wert}_1,\ \textit{Schlüssel}_2,\ \textit{Wert}_2,...}$ vorgenommen wird. Z.B. wertet ein assoziativer Vektor, der den Schlüssel `"a"` mit dem Wert `19` und den Schlüssel `"b"` mit dem Wert `42` verknüpft zu der Liste `"a",19,"b",42` aus. Ein normaler Vektor, welcher dem Index `1` den Wert `19` und dem Index `2` den Wert `42` zuordnet, wertet zu der Liste `19,42` aus.
Die Namen von Variablen geben Auskunft über ihren Typ:

- Skalare Variablen sind von der Form `$`*Name*

- Variablen, die Vektoren von Skalaren sind, haben die Form `@`*Name*

- Variablen, die assoziative Vektoren sind, schreibt man in der Form `%`*Name*

Ein *Name* ist dabei definiert als ein Buchstabe oder Unterstrich, gefolgt von Buchstaben, Ziffern oder Unterstrichen.
Gewöhnungsbedürftig ist, daß in `perl` z.B. Zugriffe auf ein Element eines Vektors als `$foo[1]` geschrieben werden (was das Element `1` des Vektors `@foo` referenziert)[2]. Zugriffe auf Elemente eines assoziativen Vektors werden als `$bar{"a"}` geschrieben (referenziert das Element des assoziativen Vektors `%bar%`, welches den Schlüssel `"a"` hat).
Eine weitere Kurzschreibweise in `perl` ist, daß man mehrere Indices in eine Referenz schreiben kann, z.B.:

[2]Im Unterschied zum vielleicht erwartetem `@foo[1]`

@foo[1,2,3] Dies steht als eine nützliche Abkürzung für die folgende Liste `($foo[1], $foo[2], $foo[3])`.

@bar{"a","b"} Steht für `($bar{"a"}, $bar{"b"})`. Man beachte eine weitere Inkonsistenz: `%bar{"a","b"}` ist verkehrt!

@foo[1..3] Steht für `($foo[1], $foo[2], $foo[3])`.

Den letzten gültigen Index in einem nicht-assoziativen Vektor `@foo` schreibt man als `$#foo`. Es sei bemerkt, daß `perl` per Default den Wert `0` als ersten Index in solch einem Array ansieht[3].

4.2.1 Skalare

Ein Skalar ist entweder eine Zahl, oder ein String. Zahlen können dezimal '`3.1415`', octal '`0755`' ooder hexadezimal '`0x80`' angegeben werden.
Strings kommen in `perl` in insgesamt sieben (nein, kein Tippfehler) Varianten:

- Man kann den String als Zeichenkette angeben, welche durch einfache Anführungszeichen (`'`) eingeschlossen ist: `'Ein String'`

- Dann kann man eine Zeichenkette angeben, die durch doppelte Anführungszeichen (`"`) umschlossen wird: `"Ein $wert String"`. In dieser Zeichenkette findet Variablenersetzung statt[4]. Hat z.B. die Variable `$foo` den Wert `9999`, so wertet der String `"Dies sind $foo Zeichen"` zu dem String `'Dies sind 9999 Zeichen'` aus. Dies funktioniert auch mit komplizierteren Zugriffen, so wertet etwa `"6*9 = $bar{'2'}"` zu `'6*9 = 42'` aus.

- Ähnlich wie in der Shell kann man `perl` anweisen, ein Kommando in einer Subshell auszuführen und die Ausgabe dieses Kommandos als String zu interpretieren. Dazu muß man das auszuführende Kommando in Rückanführungszeichen (`` ` ``) einschliessen: `` `ls -l` `` könnte dann z.B. zu dem String `'ccRcxIrM.i    ssh-thesing'` auswerten.

Will man viel Text in einen String packen, ggf. mit einer Menge Zeilenumbrüche, dann bietet es sich an, die[5] '<<' Syntax zu benutzen. Dabei wird der Text beginnend mit der nächsten Zeile, bis zum Auftreten eines bestimmten Begrenzers als String genommen. Z.B.

[3] Dies kann aber durch eine Zuweisung and eine spezielle Variable geändert werden: `$[=1` setzt den ersten Index eines Vektors auf 1.

[4] Ähnlich wie in der Shell.

[5] Ebenfalls von der Shell her bekannte.

```
$foo=<<EOM;
Dies ist
ein Text
EOM
```

Dies liefert dasselbe, wie `$foo='Dies ist\nein Text\n';` Man beachte, daß der Begrenzer am Anfang einer Zeile auftreten muß, damit er als Begrenzer erkannt wird.
`perl` kennt nun vier Möglichkeiten, solche Sammlungen vorzunehmen:

- Als Begrenzer (wie `EOM` in unserem Beispiel) wird ein einzelnes Wort verwandt. Dies entspricht der unten erläuterten Angabe von `"EOM"` als Begrenzer.

- Als Begrenzer nimmt man eine Zeichenfolge[6], welche man in einfachen Anführungszeichen (`'`) einschliesst, z.B. `<<'EOM'`. Dies entspricht dann den einfachen Strings, d.h. der Text wird genauso gesammelt und zu einem String gemacht, wie er folgt.

- Als Begrenzer wird eine Zeichenfolge eingesetzt, die man in doppelten Anführungszeichen einschliesst, z.B. `<<"EOM"`. Dies entspricht den Strings, welche in doppelten Anführungszeichen eingeschlossen sind, d.h. in dem gesammelten Text wird ebenfalls Variablenersetzung durchgeführt.

- Endlich kann man auch die Rückanführungszeichen (`` ` ``) verwenden, z.B. `` <<`EOM` ``. Dann wird jede Zeile des folgenden Textes als ein Kommando aufgefasst, welches dann ausgeführt wird. Die Ausgaben aller dieser Kommandos (bis zum Begrenzer) werden dann aneinandergehängt und als Strings genommen.

Man beachte, daß diese Sammlungen immer an der Stelle (und für) eines Strings stehen. Z.B. muß man im obigen Beispiel erst die Zuweisung mit einem Semikolon beenden, bevor der zu sammelnde Text beginnt. Man kann auch mehrere Sammlungen in einem Kommando haben, z.B. `print <<A <<B;`. Diese müssen dann in der Reihenfolge ihres Auftretens angegeben werden.
In `perl` kann man Steuerzeichen in Zeichenketten einsetzen, indem man die in Tabelle 4.1 aufgelisteten Escape-Sequenzen verwendet.
Eine verwirrende Besonderheit von `perl` ist, daß ein Ausdruck auch den Wert 'undefiniert' haben kann. Dieser wertet immer zu `0` (bzw. `""`) aus, und repräsentiert einen Fehler. Man kann mittels der `defined` Funktion prüfen, ob ein Ausdruck definiert ist, oder nicht:

```
if (defined(open(FH,"/tmp/xx")))....
```

[6] Welche auch Leerzeichen enthalten kann.

Sequenz	Bedeutung
`\n`	Newline
`\r`	Carriage-Return
`\t`	TAB
`\f`	Form-Feed
`\b`	Backspace
`\o`*oo*	ASCII Code, der durch die Oktalzahl *oo* gegeben ist.
`\x`xx	ASCII Code, der durch die Hexzahl xx gegeben ist.
`\u`	Das folgende Zeichen wird als Großbuchstabe ausgegeben.
`\l`	Das folgende Zeichen wird als Kleinbuchstabe ausgegeben.
`\U`	Die Zeichen bis zum `\E` werden in Großbuchstaben ausgegeben.
`\L`	Die Zeichen bis zum `\E` werden in Kleinbuchstaben ausgegeben.
`\E`	Beendet `\U` oder `\L` Bereich.

Tabelle 4.1: Escape-Sequenzen in `perl`.

4.2.2 Listen

Listen sind von der Form i_1, i_2, ... Wie oben gesagt, werten Vektoren zu Listen aus.

4.2.3 Wahrheit

`perl` kennt, ähnlich wie **C**, keinen Datentyp `bool`. An Stellen, an denen ein Wahrheitswert nötig ist (etwa in einer `if` Anweisung), wird das Argument ausgewertet und dann folgendermassen interpretiert:

- Eine Zahl, die nicht `0` ist, wird als 'wahr' interpretiert. Die `0` als 'falsch'.
- Der leere String (`""`) wird als 'falsch' interpretiert, genauso wie die Zeichenkette `"0"`. Jede andere nicht-leere Zeichenkette wird als 'wahr' interpretiert[7].
- Der undefinierte Wert wird als 'falsch' interpretiert.

[7] Die Zeichenkette `"00"` wird als 'wahr' interpretiert....`perl` halt:-)

- Ein Vektor wird als 'wahr' interpretiert, wenn er nicht leer ist (auch assoziative Vektoren), ansonsten als 'falsch'.

4.2.4 Kontexte

In perl gibt es Operatoren, die sowohl einen Skalar, als auch eine Liste zurückliefern, bzw. als Argument nehmen können. Was jetzt wirklich geliefert wird, wird durch den *Kontext* eines Ausdruckes entschieden. Verlangt dieser Kontext eine Liste, wird eine Liste geliefert, sonst ein Skalar. Etwa kann man mittels des Ausdruckes <STDIN> aus der Standardeingabe lesen; es gibt aber Unterschiede im Bezug auf den vorhandenen Kontext:

@foo=<STDIN>; Listenkontext. Liest alle Zeilen aus der Eingabe und legt sie als Elemente in dem Vektor @foo ab.

$baz=<STDIN>; Skalarkontext. Liest die nächste Zeile aus der Eingabe und legt sie in der skalaren Variablen $baz ab.

Es sei bemerkt, das Kontexte des öfteren zu Verwirrung führen können.

4.2.5 Filehandle

Eigentlich gibt es außer Variablen und Ausdrücken noch eine dritte Entität in perl: Filehandles. Diese stehen für geöffnete Files und werden etwa in print oder printf Operatoren gebraucht, um Ausgaben auf Files umzuleiten.
In perl werden Filehandles durch Namen dargestellt (also ohne $, @ oder %, wie bei Variablen). Nach der Tradition enthalten Filehandlenamen nur Großbuchstaben. In perl sind drei Filehandle vordefiniert: STDIN, STDOUT und STDERR, welche der Standardeingabe, Standardausgabe und der Fehlerausgabe eines Unix Prozesses entsprechen.
Um z.B. eine Zeile von der Standardeingabe in eine Variable zu lesen schreibt man: $foo=<STDIN>;
Die umschließenden '<','>' bedeuten dabei das Lesen aus dem eingeschlossenen Filehandle. Wie oben gesagt, ist diese Operation kontext-abhängig: In einem skalaren Kontext (wie im Beispiel) wird eine Zeile eingelesen, in einem Vektor-Kontext, wie etwa @foo=<STDIN>; werden alle Zeilen eingelesen und als Elemente des Vektors abgelegt.
perl bietet drei Möglichkeiten, um Filehandles zu manipulieren:

open(*File*, *Expr*) Öffnet ein File, dessen Name durch *Expr* gegeben ist und weist es dem Filehandle *File* zu. Wenn der Filename mit '<' beginnt, dann wird das File zur Eingabe geöffnet. Beginnt er mit '>', dann wird das File zur Ausgabe geöffnet. Bei einem vorstehenden '>>' wird das File zum Anhängen geöffnet. Man kann vor < und > noch ein '+' setzen, wenn man das File zum Lesen und Schreiben öffnen will. Der spezielle Filename '-' steht für `STDIN`, '>-' für `STDOUT`. Eine weitere nützliche Möglichkeit ist es, Filenamen der Form |*`Pipe`* zu verwenden. Dann wird das File zur Ausgabe geöffnet und die Ausgabe wird als Eingabe zu dem Programm *Pipe* gegeben, das in einer Subshell ausgeführt wird. Andererseits kann man auch *`Pipe`*| verwenden, dann wird das File zum Lesen geöffnet und die Eingaben sind die Ausgaben des Programmes *Pipe*, welches in einer Subshell gestartet wird.

Einige Beispiele:

```
open(PASSWD,"</etc/passwd");
open(OUTH,">/tmp/output");
open(INOUT,"+>/tmp/data");
open(MAILH,"|/usr/bin/mail");
```

`open` gibt `1` zurück, wenn das File geöffnet werden konnte, ansonsten den undefinierten Wert (s.o.).

close(*File*) Schließt das File, das durch das Handle *File* beschrieben wird.

<File> Dient zur Eingabe aus einem File. s.o.

4.2.6 Operatoren

`perl` kennt die in Tabelle 4.2 aufgelisteten Operatoren, von denen eine Menge der Sprache **C** entliehen sind, um aus Teilausdrücken neue Ausdrücke zu formen. Die Tabelle ist nach steigender Priorität geordnet und enthält auch die Assoziativität der Operatoren.
Wir werden die von **C** her bekannten Operatoren nicht oder nur kurz besprechen; wir beschränken uns haupsächlich auf die spezielleren Operatoren von `perl`.
Die komplexeren List-Operatoren, unären Operatoren und die Operatoren zur Dateiprüfung werden in eigenen Abschnitten besprochen. Hier beginnen wir als erstes mit dem Kommaoperatoren, z.B. `$foo[1]=99, $summe=0;` Der Kommaoperator wertet sein linkes Argument aus, verwirft den Wert (so daß nur die darin auftretenden Nebeneffekte interessant sind). Danach wird der rechte Operand ausgewertet und als Wert des gesamten Ausdruckes genommen. Diese Art von Operator ist auch in **C** vorhanden. In `perl` besteht die Gefahr der Verwechslung mit einer Liste: Je nach Kontext kann der Ausdruck `1,2` eine Liste mit zwei Einträgen darstellen, oder eine Anwendung des Kommaoperators sein[8].

[8]Ich hatte schon erwähnt, daß `perl` aus vermengten Konzepten besteht? Gut.

Operator	Assoziativität	Bemerkung
Listen-Operatoren	keine	Siehe Abschnitt 4.2.7
,	links	Kommaoperator
=, +=, -=, *=, /=, %=, **=, .=, x=, <<=, >>=, &=, |=, ^=	rechts	Zuweisungsoperatoren.
?:	recht	Conditional
..	keine	Bereichsoperator
||	links	Logisches Oder
&&	links	Logisches Und
|, ^	links	Bitweises Oder, Exklusiv-Oder
&	links	Bitweises Und
==, !=, <=>, eq, ne, cmp	keine	Gleichheit auf Zahlen und Strings
<, >, <=, >=, lt, gt, le, ge	keine	Vergleiche auf Zahlen und Strings
unäre Operatoren	keine	Siehe Abschnitt 4.2.8
Dateiprüfung	keine	Dateiprüfung, siehe in Abschnitt 4.2.9
<<, >>	links	Bit-Shift links, rechts
+, -, .	links	Addition, Subtraktion und String-konkatenation
*, /, %, x	links	Multiplikation, Division, Modulo und Stringreplizierung
=~, !~	links	Matching-Operatoren
!, ~, *unäres Minus*	rechts	Logische sowie bitweise Negation, Vorzeichen
**	rechts	Potenzierung
++, --	keine	Inkrement, Dekrement
(	links	Die öffnende Klammer eines Funktionsaufrufes

Tabelle 4.2: Operatoren in perl.

Den Bereichsoperator `..` hatten wir schon kennengelernt. Dieser Operator ist ein Beispiel für einen Operator, der vom Kontext abhängend entweder eine Liste liefert, oder eine Zahl. Wenn der Operator (wie im obigen Beispiel) in einem Listen-Kontext verwandt wird, liefert er die Liste, die aus den aufgezählten Werten von seinem linken bis zu seinem rechten Operand reicht, z.B.: `(1..4)` wertet aus zu `(1,2,3,4)`. Dies funktioniert auch, wenn die Operanden Zeichen sind, etwa `("a".."c")=("a","b","c")`.
Wird der Operator aber in einem skalaren Kontext verwandt (etwa wie in `if (1..10)`), dann verhält er sich äußerst merkwürdig. Er simuliert nämlich die von `sed` und `awk` her bekannten Adressbereiche. Dies geschieht dadurch, daß für jedes Auftreten dieses Operators ein internes Flag gespeichert wird. Bei jeder Auswertung dieses Operators geschieht nun folgendes: Ist das Flag 'falsch', dann bleibt es falsch, wenn der linke Operand zu 'falsch' auswertet. Wertet er zu 'wahr' aus, dann wird das Flag auf wahr gesetzt. Der Wert des Flags wird zurückgeliefert. Ist das Flag 'wahr', dann wird bei jeder Auswertung des Operators geprüft, ob der rechte Operand zu 'wahr' auswertet. Ist dies der Fall, wird das Flag auf 'falsch' gesetzt. Man vergleiche dies mit dem Adressbereichsalgorithmus im Kapitel über `sed`[9]. NB: Besteht einer der Operanden lediglich aus einer Zahl, so wird in der Auswertung der Bedingung dieser Seite diese Zahl gegen die interne Variable `$.` verglichen, in welcher die aktuelle Zeilennummer abgelegt ist.
Der Vergleichsoperator <=> vergleicht zwei Zahlen und liefert als Ergebnis `-1`, falls der linke Operand kleiner ist, als der rechte. Sind beide Operanden gleich, ist das Resultat `0`; ist der linke Operand größer, als der rechte, ist das Ergebnis `+1`. Z.B.: `1 <=> 2`
In `perl` muß man Vergleiche zwischen Strings und solche zwischen Zahlen unterscheiden. Um Strings zu vergleichen, benutzt man die Operatoren `eq` für Gleichheit (oder `ne` für Ungleichheit), z.B. `"aber" eq "nicht"`, `"nix" ne "nada"`. Der `cmp` Operator entspricht dem <=> Operator, arbeitet aber auf Strings. Um auf lexikographisch kleiner, größer, kleiner-gleich, größer-gleich zu testen, benutzt man bei Strings die `lt`, `gt`, `le` und `ge` Operatoren: `"nix" le "aber"`,`"Linux" gt "SunOS"`.
Der Stringreplizierungsoperator wiederholt einen String: Der String, der durch Auswertung des linken Operanden entsteht wird sooft wiederholt, wie die Zahl angibt, zu der der rechte Operand auswertet. Z.B. wertet `"a" x 4` zu `"aaaa"` aus.
Die Matching-Operatoren `=~` und `!~` entsprechen denen von `awk` (siehe Abschnitt 3.3). Wir werden auf sie etwas näher in Abschnitt 4.4 eingehen.
Als letztes bleibt noch der Potenzierungsoperator `**`; dieser entspricht dem `awk`-Operator `^` und muß somit nicht näher erläutert werden.

[9]Ich hatte doch schon erwähnt, daß `perl` eine merkwürdige Semantik hat?

4.2.7 Listen-Operatoren

Die Bezeichnung Listen-Operator ist für die hier vorgestellten Konstrukte zunächst verwirrend, sehen sie doch auf den ersten Blick aus, wie Funktionsaufrufe. In der Tat sind sie dies jedoch nicht, sondern stellen fest verdrahtete Bestandteile von `perl` dar.
Der Grund dafür liegt darin, daß sie in einigen verschiedenen Formen auftreten, die eine gesonderte Behandlung nötig machen.
Die Listen-Operatoren treten i.A. in einer (oder mehrerer) der folgenden Formen auf:

op Als einfacher Operator.

op Liste Als Operator, gefolgt von einer Liste.

op NameVal Liste Als Operator, gefolgt von einem Namen oder einem Ausdruck, gefolgt von einer Liste.

Nicht alle Operatoren unterstützen alle Formen, wir geben die unterstützten Formen bei den einzelnen Operatoren an.
Es sei noch angemerkt, daß man die *Liste* in den obigen Formen nicht in Klammern einschliessen *muß*, es jedoch darf. Z.B. ist in dem Ausdruck `chmod 0755, "/etc/passwd", "/etc/ping";` die Liste ungeklammert.
Hier nun die Operatoren:

chmod *Liste* Bietet die Funktionalität des *chmod(1)* Kommandos direkt als Operator. Das erste Element der Liste ist der neue Modus[10] der den Dateien, welches die zweiten, dritten, etc. Elemente der Liste sind, versucht wird zuzuweisen. Ein Beispiel hatten wir schon in der Einleitung zu diesem Abschnitt gesehen. Dieser Operator gibt die Anzahl der Files zurück, für die die Operation erfolgreich war.

chown *Liste* Wechselt den Eigentümer und die Gruppe von Dateien (äquivalent zum *chown(1)* Kommando). Hierbei stellt das erste Element der angegebenen Liste die neue numerische User-Id, das zweite die neue numerische Gruppen-Id dar. Die restlichen Elemente sind Filenamen. Z.B.: `chown (0,0,"/etc/sendmail.cf","/etc/passwd")`[11]

Auch dieser Operator gibt die Anzahl der Files zurück, für die die Operation erfolgreich war.

[10] Als Zahl, nicht symbolisch (wie bei *chmod(1)* möglich).
[11] Hier sieht man, daß die *Liste* auch geklammert werden darf.

die [*Liste*] Dieser Operator gibt die aneinandergehängten Werte der *Liste* (oder den String `"Died"`, falls die Liste fortgelassen wurde) auf dem Filehandle `STDERR` aus und beendet das Programm. In einem `eval` Ausdruck allerdings wird nur die Auswertung des `eval` abgebrochen (und in der Variablen `$@` der Fehlertext gespeichert), siehe dort. Beispiel: `if ($a!=1) { die("Fehler: a nicht 1");}` Angemerkt sei noch, daß die Zeilennummer und das `perl`-File, in welchem das `die` aufgerufen wurde, an die Fehlermeldung angehängt werden, wenn diese nicht mit `\n` endete.

exec ***Liste*** Dieser Operator entspricht in etwa dem *exec(2)* Systemaufruf und führt ein anderes Programm an Stelle des `perl`-Programmes aus. Beispiel: `exec "cat", "/tmp/xx" , "-"` Dieser Operator kehrt nur zurück, wenn die Ausführung des Programmes fehlschlug.

kill ***Liste*** Sendet ein Signal an eine Liste von Prozessen. Das erste Element der Argument-Liste ist das Signal, welches gesendet werden soll. Es kann entweder numerisch angegeben werden, oder als String, der den Signalnamen darstellt. Die Nummern bzw. Namen der Signale findet man u.a. in *signal(3)* (wobei die Namen ohne das in der Manual-Page auftretende `SIG` zu nehmen sind). Beispiel: `kill "HUP", 1`

Dieser Operator gibt ebenfalls die Anzahl der Prozesse zurück, denen das Signal erfolgreich zugestellt werden konnte.

print ***File Liste***

print ***Liste***

print Der `print` Operator ist für die Ausgabe von Werten zuständig. Per Default wird die Ausgabe auf `STDOUT` geleitet, man kann die aber mittels `select` (siehe Abschnitt 4.2.8) umleiten. In der ersten Form kann man aber auch explizit ein Filehandle angeben, durch welches die Ausgabe erfolgen soll. Der Operator gibt die zusammengehängten Werte der *Liste* aus. Fehlt die Liste, wird der Inhalt der Variablen `$_` ausgegeben. Ein oft gemachter Fehler ist es, zwischen dem Filehandle und der Liste ein Komma zu setzen. Der Operator gibt `0` zurück, wenn bei der Ausgabe ein Fehler auftrat, sonst `1`.

Beispiel: `print FOO "Schoen ist das Leben", "\n"`

printf ***File Format*****,** ***Liste***

printf ***Format*****,** ***Liste*** Dieser Operator ähnelt der *printf(3)* Funktion und dem Operator gleichen Namens in `awk`. Wie beim `print` Operator kann man ein

Filehandle angeben, in welches die Ausgabe erfolgen soll. Default ist wiederum `STDOUT`. In jedem Fall muß man diesem Operator als *Format* einen Format-String übergeben, der den gleichen Aufbau hat, wie der in *printf(3)* beschriebene. Die Elemente der *Liste* werden dann an den Argumentpositionen im Format-String eingesetzt. Man beachte, daß wie beim `print` Operator *kein* Komma zwischen *File* und *Format* gesetzt werden darf. Sehr wohl aber *muß* man *Format* und *Liste* durch ein Komma trennen. Diese Funktion gibt ebenfalls `0` zurück, falls bei der Ausgabe Fehler auftraten, sonst `1`.

Beispiel: `printf "%d: %d\n", $line, $numchar;`

return *Liste* Kehrt aus einer Subroutine zurück (siehe Abschnitt 4.5) und gibt dabei die ausgewertete *Liste* zurück.

reverse *Liste* Dieser Operator liefert in einem Vektor-Kontext eine Liste zurück, die aus der umgedrehten übergebenen *Liste* besteht. In einem skalaren Kontext wird nur das erste Element der Liste genommen, als String interpretiert, dieser umgedreht und zurückgegeben.

Beispiel: `reverse (1,2,3)=(3,2,1),reverse @foo`

sort *SubName Liste*

sort *Liste* Sortiert eine Liste. Wenn *SubName* angegeben ist, stellt es den Namen einer Subroutine dar, die beim Vergleich von Elementen aufgerufen wird. Ansonsten wird die *Liste* in der kanonischen Reihenfolge sortiert. Die Subroutine muß die Werte `-1`, `0` oder `1` zurückgeben, je nachdem ob die zwei zu vergleichenden Elemente in der Relation 'kleiner', 'gleich' oder 'größer' stehen (vgl. `cmp` und `<=>` Operatoren). Die beiden Elemente werden der Subroutine dabei in den Variablen `$a` und `$b` übergeben.

Beispiel:

```
sub canonically { $a <=> $b; }
@sorted=sort canonically @unsorted;
```

system *Liste* Führt das in der *Liste* angegebene Programm in einer Subshell aus und gibt den Exit-Code des Programmes zurück.

Z.B.: `system "ls", "-l", "/";`

unlink *Liste* Löscht die angegebenen Files und gibt die Anzahl erfolgreich gelöschter Files zurück. Man kann Directories nur löschen, wenn `perl` das `-U` Flag mitgegeben wurde.

Z.B.: `unlink "/tmp/xx.$$", "/tmp/yy.$$";`

utime *Liste* Setzt die Zugriffs- und Modifikationszeiten für Files. Die ersten beiden Elemente der *Liste* sind die Zugriffs- und Modifikationszeiten, als Sekunden seit dem 1. Januar 1970 (GMT), die restlichen Filenamen. Gibt die Anzahl erfolgreicher Änderungen zurück.

Z.B.: `$jetzt=time; utime $jetzt, $jetzt, "xx.c";`

warn *Liste* Hängt die Werte der gegebenen Liste aneinander und gibt sie auf `STDERR` aus. Anders, als der `die` Operator wird aber die Ausführung des Programmes fortgesetzt.

Z.B.: `if ($a!=$b) {warn "Parameter stimmen nicht"}`

4.2.8 Unäre Operatoren

Auch die unären Operatoren sehen aus, wie Funktionasufrufe. Sie werden nur deshalb als Operatoren betrachtet, weil man ihr Argument nicht einklammern muß. So sind z.B. `cos(3.0)` und `cos 3.0` identisch.
Hier die Operatoren:

alarm *Expr* Sendet nach *Expr* Sekunden ein `SIGALARM` an den Prozess (siehe *signal(3)*), etwa `alarm 4`.

chdir *Expr* Wechselt in das Verzeichnis, das durch den String *Expr* gegeben wird. Bei Erfolg gibt der Operator `1` zurück, sonst `0`. Etwa: `chdir ("/tmp");`

chop *Liste*

chop *Variable*

chop Der `chop` Operator trennt das letzte Zeichen eines Strings ab. Wird er auf eine Liste angewandt, trennt er das letzte Zeichen jedes Elementes ab. Wird kein Argument gegeben, trennt er das letzte Zeichen von `$_` ab. Der Operator kann auf auch L-Werte[12] angewandt werden, so führt `chop($a=<STDIN>);` dazu, daß in `$a` eine Zeile eingelesen und danach das letzte Zeichen entfernt wird. Der Operator gibt das Zeichen zurück, das er abgetrennt hat.

Beispiele:

[12] Ein L-Wert ist etwas, was auf der linken Seite einer Zuweisung stehen kann.

```
chop @a;
chop $a;
chop $a[3];
```

close *File* Schliesst das Filehandle *File*.

cos *Expr* Berechnet den Cosinus des Argumentes.

defined *Expr* Gibt `0` zurück, falls das Argument den undefinierten Wert besitzt, `1` sonst.

each *AVektor* Gibt eine Liste aus (Schüssel, Wert) zurück. Bei jeder Auswertung wird ein weiteres Paar aus dem assoziativen Vektor *AVektor* geliefert, so daß die Elemente aufgezählt werden.

Das folgende Beispiel gibt alle (Schlüssel, Wert) Paare des assoziativen Vektors `%bar` aus:

```
while ( ($s,$w)= each %bar) {
  printf("Schluessel %s, Wert %s\n",$s,$w);
}
```

Sind alle Werte aufgezählt, wird die leere Liste zurückgegeben, die als Bedingung zu 'falsch' auswertet (siehe Abschnitt 4.2.3). Man kann eine Aufzählung nicht in der Mitte zurücksetzen. Will man wieder von vorne beginnen, muß man erst alle Paare aus dem Vektor gelesen haben.

eval *Expr*

eval Der String *Expr* wird als `perl`-Programm interpretiert und ausgeführt. Dies hat den gleichen Effekt, als stünde dieser Programmtext an der Stelle des `eval`. Tritt in dem Programm ein Fehler auf, wird die Fehlermeldung in `$@` geschrieben. Wird *Expr* weggelassen, dann wird der Wert von `$_` benutzt.

exit *Expr*

exit Berechnet den (numerischen) Wert von *Expr* und bricht dann das Programm mit diesem Exit-Code ab. Wenn man das Argument forlässt, wird `0` als Exit-Code genommen.

exp *Expr*

exp Berechnet e^{Expr}. Wird das Argument weggelassen, wird `$_` genommen.

fileno *File* Gibt die Filenummer des Files zurück, das durch *File* referenziert wird.

getc *File*

getc Liest ein Zeichen aus dem durch *File* referenzierten File und gibt es zurück. Lässt man das Argument weg, wird aus `STDIN` gelesen. Ist kein Zeichen mehr verfügbar, wird `""` zurückgegeben.

gmtime *Expr*

gmtime Wandelt eine Zeit, die als Sekunden seit dem 1. Januar 1970 übergeben wird, in eine Liste mit neun Elementen um. Z.B.:

```
($sek,$min,$std,$mtag,$mon,$jahr,$wtag,$jtag,$sz)=
    gmtime (time);
```

Dir Liste besteht aus Sekunden, Minuten, Stunden, Tag (des Monats), Monat (0..11), Jahr (zweistellig), Wochentag (0..6, Sonntag=0), Tag im Jahr und ob Sommerzeit herrscht (`$sz=1` im obigen Beispiel). Lässt man *Expr* weg, dann wird `gmtime (time)` ausgeführt. Die zurückgegebene Zeit ist die, welche in der UTC (GMT) Zeitzone herrscht.

hex *Expr*

hex Wandelt das Argument (oder `$_`, falls kein Argument gegeben wurde), welches als hexadezimale Zeichenkette angesehen wird in eine Zahl um. Z.B.: `hex "0x80"=128`.

int *Expr*

int Wandelt das Argument (oder `$_`) in eine Ganzzahl um, indem die Nachkommastellen abgeschnitten werden.

keys *AVektor* Gibt eine Liste mit den Schlüsseln des assoziativen Vektors *AVektor* zurück. Z.B. ergibt `keys %bar` die Liste `"a","b"`, falls vorher `%bar` mit `$bar{"a"}=9; $bar{"b"}=42;` initialisiert wurde.

length *Expr*

length Gibt die Länge des Argumentes (bzw. `$_`) zurück: Dieses wird als String aufgefasst und dann dessen Länge genommen.

localtime ***Expr***

localtime Wandelt das Argument, welches als Sekunden seit dem 1. Januar 1970 gegeben ist, in eine Liste derselben Form wie bei `gmtime` um. Die Zeit wird allerdings relativ zur lokalen Zeitzone genommen. Ohne Argument wird `localtime(time)` ausgeführt.

log ***Expr***

log Berechnet den natürlichen Logarithmus des Argumentes (oder `$_`).

lstat ***File***

lstat ***Expr*** Führt die zu `stat` äquivalente Operation aus. Der einzige Unterschied ist, daß Daten über einen symbolischen Link selbst und nicht über die Datei, auf die er zeigt, zurückgegeben werden.

oct ***Expr***

oct Wandelt den Argumentstring (oder `$_`), welcher als Octalzahl angesehen wird, in eine Zahl um. Z.B. `oct "010"`=8.

ord ***Expr***

ord Gibt den ASCII-Code des ersten Zeichens des Argument-Strings (oder `$_`) zurück. Das Ergebnis ist immer positiv.

pop ***Vektor*** Kürzt den Vektor *Vektor* um dessen letztes Element und gibt dieses zurück. So enthält nach

```
@foo=(1,2,3);
$el=pop @foo;
```

der Vektor `@foo` die Elemente `1,2` und `$el` den Wert `3`.

rand ***Expr***

rand Gibt einen 'Zufalls'-Wert zwischen 0 und dem Wert von *Expr* zurück (bzw. zwischen 0 und 1). Das Ergebnis ist eine Fließkommazahl.

readlink ***Expr***

readlink Gibt den Filenamen, auf welchen der symbolische Link, dessen Name als Argument übergeben wird, (bzw. `$_`) zeigt. Ist der übergebene Name kein Link, oder tritt sonst ein Fehler auf, wird der undefinierte Wert zurückgegeben und die Variable `$!` auf den Fehlercode gesetzt.

Beispiel: `$realname=readlink "/etc/localtime";`

require *Expr*

require Dieser Operator lädt ein externes `perl`-Programm (Package, etc.) zum aktuellen Programm dazu. Das Argument (bzw. `$_`) ist der Name des Programms. Dieser wird in allen Directories gesucht, die in dem Vektor `@INC` abgelegt sind. Wird das Programm nicht gefunden, wird das komplette Programm abgebrochen.

reset *Expr*

reset Wird benutzt, um Variablen bzw. Suchmuster zu löschen. Falls ein Argument (String) gegeben wird, dann werden alle Variablen, deren Namen mit den dort spezifizierten Buchstaben anfangen, gelöscht. Wird kein Argument gegegeben, werden nur alle `?`*`Pattern`*`?` Suchmuster (siehe Abschnitt 4.4) gelöscht. Z.B. löscht `reset "a"` alle Variablen, die mit dem Buchstaben ’`a`’ beginnen. `reset "a-z"` löscht alle Variablen, die mit einem Kleinbuchstaben beginnen.

rmdir *Expr*

rmdir Entfernt das angegebene Directory (oder das in `$_` abgelegte). Liefert `1` bei erfolgreicher Durchführung, sonst `0`. Z.B.: `rmdir "/tmp/data"`

shift *Vektor*

shift Dieser Operator kürzt den angebenen Vektor (oder `@ARGV`) um dessen ersten Wert und liefert diesen Wert zurück. Falls der Vektor kein Element enthält, wird der undefinierte Wert zurückgegeben. Wird `shift` ohne Argument innerhalb einer Subroutine benutzt, wird statt `@ARGV` der Vektor `@_` gekürzt. Ansonsten vgl. mit `pop`, etc.

sin *Expr*

sin Liefert den Sinus von *Expr* (bzw. `$_`).

sleep *Expr*

sleep Legt den Prozess für die angegebene Anzahl Sekunden schlafen. Ist kein Argument gegeben, wird der Prozess für immer schlafen geschickt. Der Schlaf kann durch ein `SIGALARM` (siehe `alarm`) unterbrochen werden. Der Operator gibt die Anzahl Sekunden, die der Prozess geschlafen hat, zurück.

sqrt *Expr*

sqrt Gibt die Quadratwurzel des Argumentes (oder von `$_`) zurück.

srand ***Expr***

srand Setzt den Startwert für die Generierung von Zufallszahlen (siehe `rand`). Ohne Argument wird `srand(time)` ausgeführt.

stat ***File***

stat ***Expr*** Gibt einen aus dreizehn Elementen bestehenden Vektor zurück, welcher Informationen über das File enthält, welches entweder über ein Filehandle oder seinen Namen übergeben wird. Die Elemente des Vektors sind: Device-Nummer, I-Node-Nummer, Modus-Bits, Anzahl Links auf das File, User-ID des Besitzers, Gruppen-ID des Besitzers, File-Typ, Größe, Zugriffszeit, Modifizierungszeit, Erstellungszeit, Blockgrösse und Anzahl Blocks in dem File. Die Einträge entsprechen den Feldern des Records, der bei dem *stat(2)* Systemaufruf benutzt wird.

study ***Skalar***

study Dieser Operator untersucht den angegebenen String (oder `$_`) und merkt sich Optimierungen, so daß nachfolgende Mustervergleiche in diesem String schneller gehen (können).

Z.B.: `study $foo; print "ja" if $foo ˜ /ˆnix/;`

tell ***File***

tell Gibt die Position des Filepointers in dem angegeben *File* zurück. Wird das Argument weggelassen, wird das zuletzt referenzierte File verwandt.

Z.B.: `$pos=tell STDIN;`

umask ***Expr***

umask Setzt die umask für den Prozess, siehe *umask(2)*. Das Argument ist eine Zahl, die die Maske angibt. Die alte Maske wird als Resultat zurückgegeben. Ohne Argument wird keine neue Maske gesetzt, sondern nur die alte returniert.

Beispiel: `$old=umask 077;`

undef ***Expr***

undef Dieser Operator löscht die Definition seines Argumentes, das ein L-Wert sein muß und gibt selbst den undefinierten Wert zurück. Ohne Argument wird nur der undefinierte Wert geliefert. Z.B.: `undef $foo[2];`

values ***AVektor*** Ähnlich zu `keys`, nur das hier alle Werte des assoziativen Vektors *AVektor* als Liste geliefert werden.

4.2.9 Dateiprüfung

`perl` erlaubt es, mit primitiven Operatoren vielfältige Tests auf Eigenschaften von Files durchzuführen. Das Ergebnis ist entweder `1`, falls der Test positiv ausfällt, oder `0`. Ausnahmen: Die Operatoren `-A`, `-C` und `-M` liefern eine Zeit zurück (es wird die Anzahl Tage seit dem Ereignis zurückgegeben). Alle Operatoren zur Dateiprüfung nehmen ein Argument, welches den Namen der Datei (des Verzeichnisses) angibt, oder ein Filehandle. Auf dem so refernziertem File wird dann der Test durchgeführt. Man kann das Argument auch fortlassen. Dann wird als Ersatz der Inhalt von `$_` als Filename benutzt. Ausnahme: Der `-t` Operator benutzt als Default das Filehandle `STDIN`.
Alle Operatoren haben die Form −*Buchstabe File*. Dabei muß der Buchstabe direkt im Klartext folgen; folgendes wird also nicht funktionieren:
`$a="f"; if (-$a "/tmp/xx")....`
Hier nun also die Operatoren, in alphabetischer Reihenfolge:

-A Gibt die Zeit zurück, die seit dem letzten Zugriff auf die Datei vergangen ist, etwa `if (-A "/tmp/xx" > 1)....`

-B Die Datei ist eine Binärdatei (Gegenstück zu `-T`). Der Test wird durchgeführt, indem ein Block der Datei gelesen wird und die Anzahl von Steuerzeichen in ihm überprüft wird. Sind mehr als 10% davon enthalten, ist die Datei eine Binärdatei.

-b Testet, ob die Datei ein Block-Device (z.B. `/dev/sd0a`) ist.

-C Zeit, die seit Erstellung der Datei vergangen ist.

-c Testet, ob die Datei ein Character-Special Device ist (etwa `/dev/rsd0a`).

-d Testet, ob das Argument ein Directory ist.

-e Testet, ob die Datei existiert.

-f Testet, ob das Argument eine einfache Datei ist.

-g Testet, ob die Datei das Set-Group-ID Bit gesetzt hat (siehe *chmod(2)*).

-k Testet, ob die Datei das Sticky Bit gesetzt hat (siehe *chmod(2)*).

-l Prüft, ob das Argument ein symbolischer Link ist.

-M Gibt die Zeit seit der letzten Änderung der Datei zurück.

-O Prüft, ob die User-Id der Datei dieselbe wie die reale User-Id des Prozesses ist (also die des Benutzers, der `perl` gestartet hatte).

-o Prüft, ob die User-Id der Datei dieselbe ist, wie die effektive User-Id des Prozesses ist (ist verschieden von der realen User-Id für Programme, die das Set-User-Id Bit gesetzt hatten).

-p Prüft, ob die Datei eine Pipe (FIFO) ist.

-R Prüft, ob die Datei mit der realen User-Id des Prozesses lesbar ist.

-r Prüft, ob die Datei mit der effektiven User-Id des Prozesses lesbar ist.

-S Testet, ob die Datei ein Socket ist (siehe *socket(2)*).

-s Testet, ob die Datei nicht leer ist.

-T Testet, ob die Datei eine Textdatei ist (siehe `-B`).

-t Prüft, ob die Datei ein Terminal ist.

-u Liefert `1`, wenn das Set-User-Id der Datei gesetzt ist.

-W Prüft, ob die Datei mit der realen User-Id schreibbar wäre.

-w Prüft, ob die Datei mit der effektiven User-Id schreibbar wäre.

-X Prüft, ob die Datei mit der realen User-Id ausführbar wäre.

-x Testet, ob die Datei mit der effektiven User-Id ausführbar wäre.

-z Prüft, ob die Datei die Größe 0 hat.

4.2.10 Sonstige Funktionen

Dieser Abschnitt präsentiert einige weitere Funktionen, die nicht zu den Listen Operatoren oder unären Operatoren gehören.

atan2(x,y) Gibt den arcus tangens von x/y zurück.

delete ***$Array{Key }*** Löscht den angegebenen Schlüssel aus dem assoziativen Vektor *Array*. Z.B.: `delete $foo{"nix"};`

eof ***File***

eof Gibt `1` zurück, falls das angegebene Filehandle am Ende der Datei angekommen ist, sonst `0`. Ohne Argument wird das zuletzt benutzte Filehandle verwandt. Z.B.: `if (eof(STDIN))...`

fork Stellt den *fork(2)* Systemaufruf in `perl` zur ”Verfügung. Es wird eine Kopie des Prozesses angelegt. Im Vaterprozess liefert diese Funktion die Prozess-Id des Kindprozesses zurück. Im Kindprozess eine 0. Tritt bei dem Anlegen des neuen Prozesses ein Fehler auf, ist das Ergebnis der Funktion der undefinierte Wert.

grep(*Expr*,*Liste*) Diese Funktion wertet den Ausdruck *Expr* nacheinander für die Elemente der Liste aus. Der Rückgabewert ist dann die Liste der Elemente, für die der Ausdruck 'wahr' ergab. Innerhalb des Ausdruckes steht das jeweilige Element der Liste als `$_` bereit.

Z.B. stehen in diesem Beispiel am Schluß in `@foo` alle Zeilen aus `@bar`, die keine Leerzeilen sind:

```
@foo=grep(! (length($_)==0),@bar);
```

index(s,t,p)

index(s,t) Gibt die Position des ersten Auftretens des Substrings t im String s zurück. Die Suche beginnt dabei an der Position p (oder am ersten Zeichen, wenn p fehlt). Die Positionen zählen dabei ab 0[13]. Wird der Substring nicht gefunden, wird `-1` zurückgegeben (bzw. `$[-1`).

Z.B. liefert `index("laberaber","aber")` den Wert 1 zurück, wohingegen der Ausdruck `index("laberaber","aber",2)` den Wert 5 liefert.

join(*Expr*,*Liste*) Diese Funktion verbindet die Strings, welche in der *Liste* angegeben sind, mit dem String, welcher durch *Expr* dargestellt wird und gibt die resultierende Zeichenkette zurück.

Z.B:

```
$s=join(":",$name,$nachname,$telefon,$login);
$t=join(" ",@foo);
```

link(o,n) Erstellt einen Link von der Datei mit Namen o zu dem neuen Namen n. Z.B. `link("./lib/libfoo.a", "/usr/lib/libfoo.a");` Das Ergebnis der FUnktion ist `1`, im Erfolgsfall, sonst `0`.

[13]Genauer, ab dem Wert der in der Variablen `$[` abgelegt ist.

mkdir(f,m) Erstellt ein Directory mit Namen n und Zugriffsmodus m. Gibt `1` zurück, falls das Directory erstellt werden konntze, sonst `0`. Z.B. `mkdir("/tmp/data", 0700)`.

push(*Array*,*Liste*) Fügt die Elemente der *Liste* am Ende des Vektors *Array* an. Z.B. `push(@foo, 1, 2, 3)`.

rename(o,n) Bennent das File mit dem Namen n in o um.

Z.B. `rename("/tmp/xx", "/tmp/yy")` Rückgabewerte sind wieder `1` für Erfolg und `0` sonst.

rindex(s,t,p)

rindex(s,t) Sucht den Substring t im String s. Einziger Unterschied zu `index` ist, daß das letzte Auftreten des Substrings gesucht wird.

scalar(*Expr*) Erzwingt einen skalaren Kontext zur Auswertung des Ausdruckes *Expr*. Wird verwendet, um auch in Vektor Kontexten die skalare Version einer Funktion zu erzwingen.

select(*File*)

select Setzt das angebene Filhandle als Standardausgabe und liefert das alte Filehandle zurück. Wird kein Filehandle angegeben, wird nur das alte zurückgeliefert. Z.B. `$oldfh=select(OFH);`

splice(*Array*, o, l, *Liste*)

splice(*Array*, o, l)

splice(*Array*, o) Diese Funktion entfernt aus dem Vektor *Array* l Elemente, beginnend an der Position o. D.h. es werden die Elemente `$`*`Array`*`[`o`]` ... `$`*`Array`*`[`o+l−`1]` entfernt. Dann werden an der Stelle der entfernten Elemente die Elemente der *Liste* eingefügt. Wird die Liste weggelassen, werden keine Elemente eingefügt. Wird l weggelassen, werden alle Element ab o bis zum Ende des Vektors entfernt. Die Funktion gibt die entfernten Elemente als Liste zurück.

Z.B.:

```
@foo=(1,2,3,4);
@removed=splice(@foo,1,2,"a","b");
```

`@removed` enthält `(2,3)` und `@foo` enthält am Ende `(1,"a","b",4)`

split(/*Pattern*/,*Expr*,l)

split(/*Pattern*/,*Expr*)

split(/*Pattern*/)

split Spaltet den String, zu welchem *Expr* auswertet in Subfelder auf und gibt diese in einem Vektor zurück. Das *Pattern* beschreibt die Begrenzer der Felder. l gibt die maximale Anzahl Felder an, die in dem Vektor abgelegt werden sollen. Wird l weggelassen, werden so viele Felder wie möglich abgelegt. Wird *Expr* weggelassen, wird statt dessen `$_` verwandt. Wird auch noch das *Pattern* fortgelassen, werden als Begrenzer Leerzeichen/TABs verwandt.

In einem sklaren Kontext gibt diese Funktion die Anzahl Felder zurück, die abgespalten wurden. Das Array `@_` enthält in diesem Fall die Felder.

Z.B.

```
@bar=split(/:/,
    "thesing:x:1001:200:S. Thesing:/home/thesing",
    5);
($login,$passwd,$uid,$gid,$rest)=split(/:/,
    "thesing:x:1001:200:S. Thesing:/home/thesing",
    5);
```

Nachher stehen in `@bar` fünf Elemente. Das letzte enthält den Substring des Argumentes, beginnend mit '`S. Thesing`'.

In der zweiten Zeile enthält `$login` nachher den String `"thesing"`, `$uid` enthält `"1001"`, etc.

sprintf(f,*Liste*) Diese Funktion ähnelt `printf`, nur daß hier das Ergebnis nicht ausgegeben wird, sondern als String zurückgeliefert. Ansonsten identisch zu `printf`.

substr(s,o,l)

substr(s,o) Diese Funktion gibt den Substring des Strings s zurück, der an Position o startet und l Zeichen lang ist (oder bis zum Ende des Strings s reicht). Dabei zählt als erste Position der Wert von `$[` (Default: `0`). Ist o negativ, beginnt die Zeichenkette soviele Zeichen vom Ende von s entfernt, wie der Betrag von o angibt. Wird l weggelassen, wird immer der Substring bis zum Ende von s genommen.

Eine Besonderheit von `substr` ist, daß es auch als L-Wert, also auf der linken Seite einer Zuweisung auftreten kann. Dies ersetzt dann den durch es beschriebenen Substring durch die linke Seite der Zuweisung (s kann hier also nur eine Variable oder ein Vektorelement sein).

Zum Beispiel:

```
$t=substr("laber",1);
substr($t,0,1)="ue";
```

Nach der ersten Zuweisung enthält `$t` den String `"aber"`. Nach der zweiten den String `"ueber"`.

symlink(*o*,*n*) Legt einen symbolichen Link von dem File mit dem Namen o auf das File n an. Gibt `1` bei Erfolg, sonst `0` zurück.

time Gibt die Anzahl Sekunden seit dem 1. Januar 1970 GMT zurück.

times Gibt einen Vektor aus vier Elementen zurück, die Zeiten (in Sekunden) angeben: die Zeit, die der Prozess im User-Modus gelaufen ist, die Zeit, die er im System-Modus lief, die Zeit, die seine Kinder[14] im User-Modus liefen und die Zeit, die seine Kinder im System-Modus liefen. Z.B.: `($u,$s,$cu,$cs)=times;`

unshift(*Array*,*Liste*) Stellt das Gegenstück zu `shift` dar: Diese Funktion fügt die Elemente der übergebenen *Liste* am Anfang des Vektors *Array* ein und gibt die Anzahl der Elemente des resultierenden Vektors zurück. Z.B.: `unshift(@bar,1,2)`

wait Wartet, ähnlich *wait(2)*, auf die Beendigung eines Kindprozesses und liefert dann die Prozess-Id dieses Prozesses. Das Resultat ist `-1`, falls kein Kindprozess existiert.

wantarray Diese Funktion liefert `1`, falls der Kontext, in dem sie ausgewertet wird, ein Vektor-Kontext ist. Andernfalls wird `0` zurückgegeben. Nützlich, um in eigenen Funktionen abhängig vom Kontext eine Liste oder eine Liste zurückzugeben.

[14] Also mittels `fork` oder Subshells erzeugte Prozesse.

4.3 Kommandos

In diesem Abschnitt werden nun (endlich) die Kommandos von `perl` vorgestellt. Wir beginnen mit den einfachen Kommandos und werden danach zu den Conditionalen und Schleifen von `perl` kommen.
Zu beachten ist, daß in `perl` Kommandos immer mit einem Semikolon beendet werden müssen (ausser Blöcken). Dies steht im Gegensatz zu Pascal oder `awk`, aber in der Tradition von **C**.

4.3.1 Einfache Kommandos

Einfache Kommandos sind einmal Ausdrücke, denen optional noch ein *Guard* folgen kann. Ein Guard ist dabei eine Bedingung, die vor Auswertung des Ausdruckes geprüft wird[15]. Trifft diese Bedingung nicht zu, wird der Ausdruck nicht ausgewertet. `perl` kennt vier Typen von Guards:

if *Expr* Prüft, ob der mitgegebene Ausdruck[16] zu 'wahr' auswertet. Wenn dies der Fall ist, wird der Rumpfausdruck ausgewertet.

Z.B. `$foo=99 if $bar!=1;`

unless *Expr* Ist die Negation des `if` Guards. Es wird also geprüft, ob der Ausdruck zu 'falsch' auswertet. Wenn dies der Fall ist, wird der Rumpfausdruck ausgewertet. Z.B. `$foo=99 unles $bar=1;`

Diese beiden Guards wurden nur einmal ausgewertet; auch der Rumpfausdruck wird höchstens einmal ausgewertet. Die folgenden beiden Guards werten den Rumpfausdruck wiederholt aus, solange die Bedingung erfüllt ist. Wiederum wird die Bedingung vor dem Rumpfausdruck ausgewertet:

while *Expr* Die Bedingung wird ausgewertet; ist das Ergebnis dabei 'wahr', wird der Rumpfausdruck ausgewertet. Dann beginnt das Spiel wieder von vorne mit der Auswertung der Bedingung.

Z.B.: `$foo++ while 2*$foo<$bar;`

until *Expr* Ist die Negation des `while` Guards. Die Bedingung wird ausgewertet, ergibt dies 'falsch', wird der Rumpfausdruck ausgewertet und danach wiederum die Bedingung, etc.

Z.B. `$foo++ until 2*$foo>=$bar;`

---

[15] Mit einer Ausnahme: `do`-Blöcke werden manchmal vor Prüfung der Bedingung ausgeführt.
[16] Nicht mit dem Rumpfausdruck verwechseln! Dieser Ausdruck ist die Bedingung.

Wie in **C** kann man Kommandos zu einem *Block* zusammenfassen, indem man sie in geschweifte Klammern einschließt. Ein Block kann in `perl` an Stelle eines Kommandos eingesetzt werden. Allerdings sollte man beachten, daß vor allem die zusammengesetzten Kommandos *nur* Blöcke, keine einzelnen Kommandos, als Bestandteile nehmen. Dies reduziert allerdings eine Quelle von Mehrdeutigkeiten und Fehlern[17].
Als letztes einfaches Kommando muß nun noch der `do`-Block besprochen werden. Er stellt eine weitere Eigenart von `perl` dar und führt auch zu einigen semantischen Verrenkungen. Das `do`-Konstrukt tritt in drei Formen auf:

do *Block* Dieses Konstrukt fÜhrt nacheinander alle Kommandos in dem *Block* aus und gibt den Wert des letzten Kommandos zurück.

do *Subroutine*(*Liste*) Die Subroutine mit dem Namen *Subroutine* wird aufgerufen. In den Klammern werden die Argumente zur Routine mitgegeben. Auch wenn die Subroutine keine Argumente benutzt, müssen die Klammern angegeben werden, um diese Variante von der letzten zu unterscheiden. Das Ergebnis dieses Aufrufes ist der Wert, der von der Subroutine zurückgegeben wird. Siehe Abschnitt 4.5.

do *Expr* Nimmt den String, der durch den Ausdruck gegeben ist und interpretiert ihn als einen Filenamen eines `perl`-Programmes, das dazu geladen und ausgeführt wird. Ähnelt also dem `require`.

Für die erste und zweite Variante des `do`-Blockes ändert sich die Semantik der `while` und `until` Guards. Folgen sie nämlich einem `do`-Block, dann wird *zuerst* der Block ausgeführt, bevor die Bedingung der Guards getestet wird. Dies gilt aber nicht für die `if` und `unless` Guards, die immer noch vor dem Block ausgewertet werden.
Beispiele:

```
do {$foo+=$bar; $bar--;} while $bar!=0;
do canonical(1,2);
do "cgi.pl";
```

4.3.2 Zusammengesetzte Kommandos

Die zusammengesetzten Kommandos in `perl` bestehen aus sechs Varianten des Conditionals, sechs Schleifen und einem Block-Konstrukt.
Die Conditionale:

if (*Expr*) *Block*$_1$

[17] Ganz im Gegensatz zur sonstigen `perl`-Tradition, wie ich bemerken möchte:-)

if (*Expr*) *Block*$_1$ else *Block*$_2$

if (*Expr*$_1$) *Block*$_1$ elsif (*Expr*$_2$) *Block*$_2$... else *Block*$_N$ Die Bedingung *Expr* wird ausgewertet. Ergibt dies 'wahr', wird *Block*$_1$ ausgeführt. In der zweiten Variante wird *Block*$_2$ ausgeführt, wenn die Bedingung 'falsch' ergibt. Die dritte Variante prüft in diesem Fall die zweite Bedingung *Expr*$_2$ und führt, wenn dies 'wahr' ergibt den zweiten Block aus, etc.

Die anderen drei Conditionale erhält man, indem man `if` durch `unless` ersetzt, was einfach den Sinn der Bedingung umkehrt (siehe voriger Abschnitt).
Die sechs Schleifenkonstrukte von `perl` haben die Gemeinsamkeit, daß man die Schleifen mit Labeln versehen kann. Diese können dann benutzt werden, um gezielt aus bestimmten Schleifen wieder auszusteigen (mittels `last`). Ein Label ist dabei ein Name, gefolgt von einem Doppelpunkt. Das Label muß direkt vor dem Schleifenkopf stehen (siehe unten stehende Beispiele).

Label*: while (*Expr*) *Block

Label*: while (*Expr*) *Block* continue *Block Die Bedingung *Expr* wird ausgewertet. Ergibt sie 'falsch', wird die Schleife verlassen. Sonst wird der *Block* ausgeführt. Ist ein mittels `continue` eingeleiteter Block gegeben, wird dieser nach Ausführung des Schleifenrumpfes und vor der nächsten Prüfung des Schleifenbedingung ausgewertet. Dies geschieht auch, wenn im Schleifenrumpf ein `next` Kommando den nächsten Schleifendurchlauf erzwingt.

Wenn man `while` durch `until` ersetzt erhält man die nächsten beiden Schleifenvarianten. Hierbei wird dann einfach der Sinn der Bedingung umgekehrt, analog zu den `while` und `until` Guards des vorigen Abschnittes.

Label*: for (*i-Expr*; *c-Expr*; *r-Expr*) *Block Dies entspricht der von **C** oder `awk` bekannten Schleife gleichen Namens. In der Tat ist sie in `perl` äquivalent zu folgendem Konstrukt:

```
i-Expr;
Label:
while (c-Expr) Block continue {r-Expr;}
```

Label*: foreach *Variable* (*Array*) *Block Diese Schleife dient, ähnlich wie ihr Gegenstück in `awk`, dazu, Schlüssel/Wert Paare eines Vektors aufzuzählen. Dabei werden die Schlüssel des Vektors *Array* nacheinander in der variablen *Variable* abgelegt und stehen dann im Rumpf der Schleife zur Verfügung.

Z.B.

```
foreach $v (%nam) { print "n[$v]=$nam{$v}\n"; }
```

Als letztes bleibt noch das folgende Konstrukt:

Block* continue *Block Bei dieser Konstruktion wird der Block der auf das Schlüsselwort `continue` folgt dann ausgeführt, wenn in dem ersten Block eine `next` Anweisung erfolgt.

In jedem Block, auch wenn dieser nicht Rumpf einer Schleife ist, kann man mit den folgenden Kommandos den Durchlauf steuern:

next *Label* Erzwingt den nächsten Durchlauf der Schleife mit dem angegebenen *Label*. Wird es weggelassen, ist die innerste Schleife gemeint. In einem normalen Block (keine Schleife), wird ein folgender `continue` Block durchlaufen und der Block nochmal durchgeführt. In einer Schleife wird ebenfalls der `continue` Block durchlaufen, bzw. die *r-expr* einer `for` Schleife ausgewertet. Danach wird wieder zur Auswertung der Schleifenbedingung gesprungen.

last *Label* Verlässt den Rumpf der durch das *Label* bezeichneten Schleife (bzw. die innerste), ohne `continue` Blöcke auszuführen. In einem normalen Block wird einfach nur der Block sofort verlassen.

redo *Label* Ähnelt der `next` Anweisung. Einziger Unterschied ist, daß keine `continue` Blöcke ausgewertet werden und auch in einer Schleife wird die Schleifenbedingung nicht erneut ausgewertet, sondern sofort der Rumpf neu ausgeführt.

Beispiele:

```
while ($n>0) {
 $s+=$n;
 $n--;
}

for ($s=0; $n>0; $n--) {
 $s+=$n;
}

A:
while ($n>1) {
 B:
  while ($m>9) {
   ...
   last A, if $n+$m>21;
```

```
  }
}

FOO: {
 if ($a == 1) { $x=2; last FOO;}
 if ($a == 2) { $x=4; last FOO;}
}
```

Man beachte, daß das letzte Beispiel das Äquivalent einer **C** `switch` Anweisung ist:

```
 switch (a) { case 1: .... case 2: ....}
```

4.4 Pattern

`perl` kennt eine Vielzahl von Pattern, die wir im folgenden vorstellen. Dabei wird das Matching immer gegen eine Zeichenkette betrachtet.

^ Passt auf den Anfang der Zeichenkette.

$ Passt auf das Ende der Zeichenkette.

***** Passt auf beliebig (auch keine) Wiederholungen des vorstehenden Patterns.

+ Passt auf ein- oder mehrmalige Wiederholung des vorstehenden Patterns.

? Passt auf null- oder einmaliges Auftreten des vorstehenden Patterns.

{n,m} Passt auf n- bis m-malige Wiederholung des vorstehenden Patterns. Spezialformen: `{`n`,}` für mindestens n-malige Wiederholung, `{`n`}` für genau n-malig.

| Alternative z.B. `a|b`

[...] Eine Klasse, wie von `sed` und `awk` her bekannt.

(...) In `perl` können Pattern geklammert werden. `perl` merkt sich bei einem Matching, welche Teile des Strings gegen geklammerte Patternteile matchten. Diese sind dann mittels `\`n wieder in referenzierbar.

. Der Punkt matcht jedes Zeichen, ausser `\n`.

b Matcht Wortübergänge, also Stellen zwischen `\w` und `\W`, siehe unten.

B Matcht Stellen, die keine Worübergänge sind.

Die folgenden speziellen Sequencen können in Pattern eingesetzt werden, um bestimmte Zeichen zu matchen:

- Die Sequenzen `\n`, `\r`, `\t`, `\f` stehen für die gewohnten Steuerzeichen Newline, Carriage Return, TAB, Form-Feed.
- `\d` steht für eine Ziffer, ist also äquivalent zu `[0-9]`.
- `\D` steht für etwas, das keine Ziffer ist, also `[^0-9]`.
- `\w` steht für ein Wortzeichen, äquivalent zu `[0-9a-zA-Z_]`.
- `\W` steht für etwas, das kein Wortzeichen ist.
- `\s` steht für eine Leerzeichen, also `[\t\n\r\f ]`
- `\S` steht für die Negation von `\s`.
- `\0`*oo* matcht das Zeichen, dessen ASCII-Code durch die Okatalzahl *oo* gegeben ist.
- `\x`xx matcht das Zeichen, das den Hexadezimalcode xx hat.
- `\`n matcht den Teilstring, der von dem n-ten Klammerausdruck (s.o.) schon gematcht wurde. Zählung beginnt mit 1.

Z.B.:

```
/^\d+/
/(\d+) and \1/
```

4.4.1 Operatoren

Im Zusammenhang mit Pattern sind folgende Operatoren nützlich.
Wir hatten schon in Abschnitt 4.2.6 die beiden Matching-Operatoren `=~` und `!~` erwähnt. Diese dienen dazu, Zeichenketten an ein Pattern zu *binden*. Z.B.: `$foo =~ /^\d/+`. Dabei negiert `!~` noch das Ergebnis des Matches. Der eigentliche Operator, der das Matching vornimmt ist jedoch das `/`*`Pattern`*`/`: Er liefert in einem skalaren Kontext entweder 'wahr' oder 'falsch' zurück, je nachdem, ob der Match gelang oder nicht. Dabei wird die durch die Matching-Operatoren angegebene Zeichenkette gematcht, bzw. `$_`, wenn diese nicht verwandt wurden. Gelingt ein Match, so werden in den Variablen `$1`, `$2`, etc. die Teile des Strings abgelegt dir jeweils durch den ersten, zweiten, etc. Klammerausdruck im *Pattern* gematcht wurden. So steht nach

`"123 456" =~ /(\d+) (\d+)/` in `$1` der String `"123"` und in `$2` der String `"456"`.
In einem Vektor-Kontext liefert dieser Operator einen Vektor zurück, der aus den gematchten Teilstrings besteht (also dem, was sonst in `$1`, etc. steht). Misslang der Match, wird der Nullvektor zurückgegeben. In diesem Kontext werden die Variablen `$`*n* *nicht* gesetzt.
Beispiel: `($first,$last) = ("123 456" =~ /(\d+) (\d+)/)`
Der `?`*Pattern*`?` Operator ist nahezu identisch zu `/`*Pattern*`/`, nur daß er nur einmal den Match ausführt und diesen dann immer wider zurückgibt, solange bis die `reset` Funktion ausgeführt wird.
Ähnlich zum Substitute-Kommando von `sed` kennt auch `perl` einen Operator, mit dem man Ersetzungen surchführen kann. Genau wie die beiden vorigen Operatoren, benutzt er dazu einen Ausdruck, der an ihn (via z.B. `=~`) gebunden ist (bzw. `$_`). Allerdings muß dieser Ausdruck in diesem Fall ein L-Wert sein, da ja die Zeichenkette verändert wird. Die Form des Operators ist:

`s/`*Pattern*`/`*Text*`/`*Flags*

Dabei wird in der gebundenen Zeichenkette das erste Vorkommen des *Pattern*s durch den *Text* ersetzt. Im Text können, wie bei `sed`, gemathcte Subpattern via `\`*n* eingesetzt werden (s.o.). Will man alle Vorkommen ersetzen, muß man das Flag `g` angeben. Soll der Match Groß-/Kleinschreibung ignorieren, gibt man `i` an. Will man statt eines einfachen Textes den Wert eines Ausdruckes einsetzen, gibt man diesen an der Stele von *Text* an und hängt das `e` Flag an.
Dieser Operator gibt die Anzahl der gemachten Ersetzungen zurück. Als Seiteneffekt werden in dem gebundenen L-Wert die Ersetzungen gemacht.
Beispiele:

```
$foo =~ s/^(\d+) (\d+).*$/\1-\2/;
if (s/#//) { print "# enthalten";}
```

Als letzter wird nun noch der Translate-Operator vorgestellt. Dieser hat die analoge Funktion zum Translate-Kommando von `sed`: Er ersetzt Vorkommen bestimmter Zeichen durch bestimmte andere Zeichen. Seine Form:

`tr/`s`/`r`/`

Hierbei sind s und r Zeichenfolgen. Das erste Zeichen aus s wird durch das erste Zeichen aus r ersetzt, etc. Für die Bindung gilt das gleiche, wie bei den anderen Operatoren dieses Abschnittes. Siehe *perl(1)* für eine ausführlichere Diskussion dieses Operators.

4.5 Subroutinen

In `perl` kann man Subroutinen definieren, indem man das Schlüsselwort `sub`, den Namen der Subroutine und dann einen Block, der den Rumpf der Routine darstellt, angibt:

```
sub unsinn {
 print "hallo\n";
 return "42=6*9";
}
```

Subroutinen in `perl` halten einige Überraschungen bereit. So werden die Argumente, die man an sie übergeben kann, nicht etwa im Kopf der Routine nach dem Namen aufgelistet, sondern in dem Vektor `@_` übergeben. So enthält dieser nach dem Aufruf `do unsinn(1,2)` etwa die Werte `1,2`.
Auch sind Variablen in `perl` erstmal global. In Subroutinen muss man mittels des `local` Konstruktes explizit lokale Variablen definieren, z.B.

```
 sub unsinn {
 local ($foo,$bar)=@_;

 print $foo+$bar . "\n";
}
```

Dies deklariert die lokalen Variablen `$foo` und `$bar` und weisst ihnen auch gleich den Wert des ersten bzw. zweiten Parameters der Routine zu.
Es sei noch bemerkt, daß man in `perl` lokale Variablen in jedem Block einfÜhren kann, also nicht nur in Subroutinen.
`perl` übergibt Werte an Subroutinen immer per Werte-übergabe ('Call-by-value'). Will man z.B. eine Array in einer Subroutine verändern, muß man es explizit per Referenz ('call-by-name') übergeben. Dies geschieht, indem man bestimmte Namen per Referenz übergibt.
Eine der am meisten kranken Konstrukte in `perl` ist nun, daß diese Übergabe dadurch realisiert ist, daß beim Aufruf *alle* Objekte, mit einem bestimmten Namen, übergeben werden[18]. In der Subroutine kann man dann auf alle diese Objekte wieder per Referenz Bezug nehmen, z.B.:

```
sub unsinn {
 local (*anarray)=@_;

 $anarray{"Nix"}="bar";
```

[18] Also z.B. die skalare Variable `$foo`, der Vektor `@foo` und der assoziative Vektor `%foo`.

```
}

 do anarray(*foo);
 do anarray(*baz);
```

Dieses Beispiel übergibt in den beiden Aufrufen einmal alle Objekte, deren Namen `foo` ist, bzw. alle, deren Namen `baz` ist. In der Routine wird dann mittels des `*anarray` der Name `anarray` per Referenz an diese Objekte angehängt. Die Zuweisung im Rumpf verändert also einmal den Vektor `%foo`, einmal den Vektor `%baz`.
Zum Schluß noch die Bemerkung, daß man eine Subroutine neben einem Konstrukt wie `do unsinn();` noch als `&unsinn;`, bzw. `&unsinn(1,2);` aufrufen kann.

4.6 Spezielle Variablen

Dieser Abschnitt listet einige der vielen speziellen Variablen von `perl` auf:

$| Wird diese Variable auf einen Wert !=0 gesetzt, so wird bei Ausgaben via `print` der Puffer der Datei sofort nach jeder Ausgabe auch weggeschrieben. Sonst sind die Ausgaben gepuffert und werden erst später bei Füllung des Puffers geschrieben.

$& Enthält, ähnlich wie bei `sed`, nach einem Matching den String, der auf das gesamte Pattern passte.

$_ Der Inhalt dieser Variablen wird oft als Default benutzt, wenn bei Operatoren ein Argument weggelassen wird.

$$ Die Prozessnummer des `perl`-Programmes.

$? Der letzte Exit-Code eines in einer Subshell ausgefÜhrten Kommandos.

$[Der Index des ersten Elements eines Vektors. Default: 0

$! Der Fehlercode nach einem Fehler bei z.B. einem `print` Enthält den Wert, der auch in der **C** Variablen `errno` verfügbar wäre, siehe *errno(2)*.

$@ Die Fehlermeldung im Klartext.

@ARGV Dieser Vektor enthält die Kommandozeilenparamter, mit denen das Programm aufgerufen wurde.

@INC Dieser Vektor enthält Namen von Directories, in denen nach `perl`-Programmen gesucht wird. Siehe `require`.

%ENV Dieser assoziative Vektor enthält die Umgebungsvariablen. Der Name der Variablen ist dabei der Schlüssel.

4.7 CGI-Scripte

4.8 Beispiele

Dieses Beispiel[19] bennent die Bezeichner in einem GTK-Sourcefile um[20] um. Nützlich, um eine neue Klasse auf dem Code einer alten zu basieren. Der Aufruf geschieht mittels `gtkrn` *Old-Name New-Name Files*

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

my $old= shift;
my $new= shift;

sub capitalize {
   my $str= shift;
   $str=~ s/^([a-z])/uc($1)/ei;
   $str=~ s/_([a-z])/uc($1)/gei;
   return $str;
}

sub replacestr {
  my ($head,$tail)= @_;
  if ($tail =~ /^(.*)$old(\w*)/i) {
   my $head2= $1;
   my $tail2= $2;
   if ($head eq "gtk_")
     { return "gtk_".$head2.$new.$tail2; }
   if ($head eq "GTK_")
     { return "GTK_".$head2.uc ($new).$tail2; }
   if ($head eq "Gtk")
     { return "Gtk".$head2.capitalize ($new).$tail2; }
```

[19] Beigestuert von Henrik Theiling

[20] GTK ist ein Toolkit, mit welchem man graphische Komponenten, wie Fenster, etc., konstruieren kann.

```
  }
  return $head.$tail;
}

for $file (@ARGV) {
   open (F, $file) || die $!;
   open (G, ">$file.tmp") || die $!;

   while (<F>) {
       s/(gtk_?)(\w+)/&replacestr($1,$2)/eig;
       print G;
   }
   close G;
   close F;
   rename ("$file.tmp", "$file");
}
```

4.9 Falltüren

`perl` stellt eine einzige Sammlung von Falltüren dar. Die ziemlich verwinkelte Semantik und der merkwürdige Aufbau einiger Konstrukte, zusammen mit vielen Default-Annahmen und Ersetzungen macht das Programmieren darin nicht gerade einfach.
Besonders dumm ist die Tatsache, daß Variablen durch ihre Benutzung eingeführt werden und somit Tippfehler fatale Folgen haben können. Durch das globale Scoping sind größere Projekte nur mit heftigsten Kopfschmerzen zu entwickeln.
Im Rest dieses Kapitels wurden schon einige Falltüren erwähnt, so daß dieser Abschnitt hiermit bereits enden kann.

4.10 Literatur

Es gibt vielfältige Literatur über `perl`. Besonders im O'Reilley Verlag sind eine ganze Reihe von Büchern zu `perl` erschienen. Diesem Kapitel lag [WS93] zu Grunde. Angesichts der grausigen Übersetzung dieses Werkes würde ich aber eher das Original empfehlen.