Einer nach dem Anderen bitte...

Es gibt etliche Operatoren in allen Programmiersprachen und zu einem Großteil sind sie auch überall gleich. Jeder setzt die Operatoren ein und macht sich wenig Gedanken darüber, wie und in welcher Reihenfolge diese abgearbeitet werden. In fast allen Fällen macht der Compiler oder Interpreter auch das was man möchte.Perl nennt das DWIM „Do what I mean“.

In einigen Fällen scheint auf den ersten Blick auch alles zu funktionieren, aber bei einem gründlicheren Test würde auffallen, dass vielleicht doch nicht alles so funktioniert wie es soll.

Für die Operatoren gibt es festgelegte Rangordnungen, wie man sie zum Beispiel auch aus der Mathmatik kennt: „Punkt-vor Strichrechnung“. So erkennen wir auf Anhieb in welcher Reihenfolge der Ausdruck `2 + 6 * 3` berechnet wird.

Damit ist schon die Rangordnung innerhalb der Operatoren `*`, `/`, `+`, `-` geklärt. Aber allein in Perl gibt es über 50 Operatoren - und da immer zu wissen welcher Operator eine höhere Wertigkeit hat, ist nicht so einfach. Wichtig ist noch zu wissen, dass alle C-Operatoren die gleiche Wertigkeit in Perl behalten haben.

Dass man leicht mal in die Falle der Wertigkeiten tappen kann, sieht man bei diesem Stück Code:

```
my $link = 'htt://foo-magazin.de';
if (! $link =~ m!^https?://!i ){
   print "Fehler!";
}
```

Die Intention des Programmierers ist, einen Fehler auszugeben wenn ein Link *nicht* mit `http://` oder `https://` anfängt. Sieht alles logisch aus. `$link =~ m!^https?://!i` überprüft, ob der Link mit `http://` anfängt und dann einfach mit `!` negiert. Alles klar, oder?
Hier durchkreuzt aber die Wertigkeit von `!` die Pläne. Dieser Operator hat eine sehr hohe Wertigkeit und so wird erst das `!` angewendet und danach erst `=~`. Also wird erst `$link` negiert und dann wird überprüft, ob die Negation von `$link` mit `http://` anfängt - und das ist nie der Fall, so dass nie „Fehler!“ ausgegeben wird.

Ich will...

... dass der Code das macht was *ich* will!

Um solche Fehler zu vermeiden, kann man mehrere Wege gehen: Entweder man verwendet andere Operatoren und verringert so die Anzahl der Operatoren, man verwendet synonyme Operatoren oder man gruppiert Code-Stücke.

Durch Gruppierung wird alles eindeutig, aber so kann es passieren, dass der Code schnell unübersichtlich wird, weil zu viele öffnende und schließende Klammern enthalten sind:

```
my $link = 'htt://foo-magazin.de';
if (!($link =~ m!^https?://!i)){
   print "Fehler!";
}
```

Duch die Gruppierung wird allerdings deutlich gemacht, dass zuerst das Matching gemacht werden soll und danach das Ergebnis negiert wird - der Code macht das was sich der Programmierer ursprünglich überlegt hat.

Die nächste Möglichkeit ist die Verwendung anderer Operatoren. Wer Perl‘s Operatoren kennt, weiß, dass es für `!(...`

=~) einen alternativen Operator gibt: !~ Damit sieht der Code folgendermaßen aus:

```
my $link = 'htt://foo-magazin.de';
if ( $link !~ m!^https?://!i ){
   print "Fehler!";
}
```

Der Vorteil dieser Möglichkeit liegt darin, dass die if-Bedingung nicht durch Klammernpaare unübersichtlich wird und durch das !~ ist auf den ersten Blick erkenntlich, dass das Matching negiert werden soll. Ein Nachteil ist allerdings, dass viele Einsteiger diesen Operator gar nicht kennen.

Für viele Operatoren gibt es auch einen synonymen Operator, der eine andere Wertigkeit hat. So hat and eine niedrigere Wertigkeit als &&, so wie or eine niedrigere Wertigkeit als || hat. Genauso gibt es für ! einen entsprechenden Operator: not

```
my $link = 'htt://foo-magazin.de';
if ( not $link =~ m!^https?://!i ){
   print "Fehler!";
}
```

Jetzt hat =~ eine höhere Wertigkeit als not und das Matching wird zuerst ausgeführt und danach die Negierung durch das not. Der Vorteil dieser Möglichkeit liegt darin, dass es sehr gut zu lesen ist - fast wie ein vollständiger englischer Satz. Außerdem wird mit =~ ein Operator verwendet, der den meisten (Perl-)Programmierern bekannt ist.

Dieses Beispiel zeigt, wie leicht man in solche Fallen tappen kann. Hier ist es immer wichtig zu testen, ob der Code mit allen möglichen Eingaben auch das macht was er soll. Hier wäre mit wenigen Tests schon klar geworden, dass mit dem Ausdruck etwas nicht stimmt.

Auch das noch...

Ein weiteres - klassisches - Beispiel für so einen Fehler ist der folgende Code:

```
open FH, '<', $file || die $!;
```

open() scheint zwar eine Funktion zu sein, ist aber in Wirklichkeit ein Listenoperator (siehe auch perldoc perlop).

Während der Programmierer in 99,9% der Fälle will, dass die Datei geöffnet wird und in einem Fehlerfall das Programm mit einer Fehlermeldung abgebrochen wird, ist es hier so, dass das open eine niedrigere Wertigkeit als das || hat und somit wird erst $file || die $! gemacht und danach erst das open. Das open ist sogar eines der niedrigsten Operatoren überhaupt (wie alle Listenoperatoren, die rechts stehen), so dass nur noch and, not und or niedriger stehen.

Der obige Code müsste also so aussehen:

```
open FH, '<', $file or die $!
```

Periodensystem der Operatoren

Es gibt ein „Periodensystem der Operatoren“ von Mark Lentczner, das allerdings von 2004 ist und auf den (voraussichtlichen) Operatoren von Perl6 basiert. Allerdings stimmt die Rangordnung auch für die bestehenden Operatoren in Perl 5.

http://www.ozonehouse.com/mark/blog/code/PeriodicTable.pdf

Was macht der Compiler?

Mit dem Modul B::Concise kann man sich ganz gut anschauen, was der Compiler bei ! beziehungsweise not macht. Zur Veranschaulichung wird folgender Code genommen:

```
my $link = 'htt://foo-magazin.de';
if (! $link =~ m!^https?://!i ){}
```

Aufgerufen wird es mit perl -MO=Concise,-exec link.pl. -M ist der Kommandozeilenswitch anstelle eines use im Programm. So kann man statt perl -e „use CGI;“ einfach perl -MCGI schreiben. Das Modul O ist ein Modul, das sogenannte „Compilerbackend“-Module einbindet. Davon gibt es mehrere und alle bieten einen gewissen Blick in Perl-Internas.

Das (Teil-)Ergebnis des Aufrufs ist in Listing 1 zu sehen.

```
C:\>perl -MO=Concise,-exec link.pl
...
7  <0> padsv[$link:1,4] s
8  <1> not sK/1
9  </> match(/"^https?://"/) sKS/RTIME
...
link.pl syntax OK
```

Listing 1

Im Gegensatz dazu der Code, bei dem das `!` durch `not` ersetzt wurde (Listing 2).

```
C:\>perl -MO=Concise,-exec link.pl
...
7  <0> padsv[$link:1,4] s
8  </> match(/"^https?://"/) sKS/RTIME
9  <1> not sK/1
...
link.pl syntax OK
```

Listing 2

Im ersten Beispiel erkennt man, dass der Skalar genommen wird `padsv` als nächstes negiert wird `not` und dann erst der Match gemacht wird (`match`). Mit dem `not` statt dem `!` ist die Reihenfolge von `match` und `not` getauscht.

Renée Bäcker

TPF-TICKER

Act!-Updates

Act! ist ein Konferenz-Verwaltungs-Tool, das in Perl geschrieben ist. Viele der Perl-Workshops und YAPCs werden darüber administriert. Mittlerweile gibt es Act! in vielen Sprachen - unter anderem in Russisch, Portugiesisch, Italienisch, Englisch und Hebräisch.

CRM gesucht, das in Perl geschrieben ist

Die Perl-Foundation ist/war auf der Suche nach einem CRM-System, das in Perl geschrieben ist. Scheinbar gibt es nicht allzu viele davon. Ob Jim Brandt schon etwas gefunden hat, ist nicht klar. Wer also ein solches CRM-System kennt, kann sich an die Perl-Foundation wenden.

Einige Hinweise zu Perl-CRMs sind im Blog gegeben worden: http://news.perlfoundation.org/2008/01/i_need_a_crm_package.html