Ausgabe 1/2008 – www.foo-magazin.de - 6,00 €

$foo

PERL MAGAZIN

Neue RegEx-Features in Perl
Balanced Matching und Grammatiken

Perl 6 Tutorial - Teil 2
Operatoren für Skalare

Encode
Charsets oder „Warum funktionieren meine Umlaute nicht?“

Nr 05

Sichern Sie Ihren nächsten Schritt in die Zukunft

Astaro steht für benutzerfreundliche und kosteneffiziente Netzwerksicherheitslösungen. Heute sind wir eines der führenden Unternehmen im Bereich der **Internet Security** mit einem weltweiten Partnernetzwerk und Büros in Karlsruhe, Boston und Hongkong. Eine Schlüsselrolle im Hinblick auf unseren Erfolg spielen unsere Mitarbeiter und hoffentlich demnächst auch Sie! Astaro bietet Ihnen mit einer unkomplizierten, kreativen Arbeitsumgebung und einem dynamischen Team beste Voraussetzungen für Ihre berufliche Karriere in einem interessanten, internationalen Umfeld.

Zur Verstärkung unseres Teams in Karlsruhe suchen wir zum nächstmöglichen Eintritt:

Perl Backend Developer (m/w)

Ihre Aufgaben sind:

- Entwicklung und Pflege von Software-Applikationen
- Durchführung eigenständiger Programmieraufgaben
- Optimierung unserer Entwicklungs-, Test- und Produktsysteme
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Unsere Anforderungen an Sie sind:

- Fundierte Kenntnisse in der Programmiersprache Perl, weitere Kenntnisse in anderen Programmier- oder Script-Sprachen wären von Vorteil
- Selbstständiges Planen, Arbeiten und Reporten
- Fließende Deutsch- und Englischkenntnisse

Software Developer (m/w)

Ihre Aufgaben sind:

- Entwicklung und Pflege von Software-Applikationen
- Durchführung eigenständiger Programmieraufgaben
- Optimierung unserer Entwicklungs-, Test- und Produktsysteme
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Unsere Anforderungen an Sie sind:

- Kenntnisse in den Programmiersprachen Perl, C und/oder C++ unter Linux, weitere Kenntnisse in anderen Programmier- oder Script-Sprachen wären von Vorteil
- Kompetenz in den Bereichen von Internet Core Protokollen wie SMTP, FTP, POP3 und HTTP
- Selbstständiges Planen, Arbeiten und Reporten
- Fließende Deutsch- und Englischkenntnisse

Astaro befindet sich in starkem Wachstum und ist gut positioniert um in den Märkten für IT-Sicherheit und Linux-Adaption auch langfristig ein führendes Unternehmen zu sein. In einer unkomplizierten und kreativen Arbeitsumgebung finden Sie bei uns sehr gute Entwicklungsmöglichkeiten und spannende Herausforderungen. Wir bieten Ihnen ein leistungsorientiertes Gehalt, freundliche Büroräume und subventionierte Sportangebote. Und nicht zuletzt offeriert der Standort Karlsruhe eine hohe Lebensqualität mit vielen Möglichkeiten zur Freizeitgestaltung in einer der sonnigsten Gegenden Deutschlands.

Interessiert?
Dann schicken Sie bitte Ihre vollständigen Unterlagen mit Angabe Ihrer Gehaltsvorstellung an careers@astaro.com. Detaillierte Informationen zu den hier beschriebenen Stellenangeboten und **weitere interessante Positionen** finden Sie unter www.astaro.de. Wir freuen uns darauf, Sie kennen zu lernen!

Astaro AG
Amalienbadstr. 36 • D-76227 Karlsruhe
Monika Heidrich • Tel.: 0721 25516 0

www.astaro.de

e-mail | web | net | security

astaro internet security

PERL'S GREATEST STRENGTH

Recently I was in Switzerland, teaching a week-long „Introduction to Perl" class. Apart from an opportunity to visit one of the most beautiful and highly civilized countries on Earth, that week also gave me an unexpected gift: the chance to see Perl through the eyes of newcomers to our favorite language...and thereby to examine and reflect upon Perl's inherent strengths and weaknesses.

In a sense, I have had a lot of practice at that kind of examination and reflection. Because that's what I've been helping Larry do for the past seven years, as we have worked to design Perl 6. But in Switzerland I found it very valuable to look at Perl from „the other end" too: to consider Perl's limitations and advantages from the novice point-of-view.

Perl's weaknesses are well-known and widely discussed. And, of course, we're addressing them directly in the development of Perl 6. The main culprits are:

Too much punctuation

Perl's punctuation variables, with their global scope and magic control of fundamental behaviours, are not only a notorious source of subtle bugs, they're also a major cause of code obfuscation. And when you add in the obscure and symbol-laden syntax of Perl's regular expressions, it's easy to loose track of the meaning of a program amidst the line noise.

Mutant ninja regexes

Not only is the syntax of regexes too noisy, it's also a mess. Important regex features (like non-capturing groups, look-ahead, and whitespace matching) have to use complex and ugly syntaxes, because all the simple and pretty syntax is already allocated to earlier (but less useful) features. Regexes are also missing important facilities like pattern naming, hierarchical captures, and full grammars (though 5.10 definitely improves regex semantics considerably -- at the cost of still more ugly syntax).

No static typing

Perl's dynamic typing is powerful and easy to use, but it also requires discipline and understanding to ensure that type mistakes don't creep in (and experience and skill to detect them when they inevitably do occur). Static typing can help detect and prevent many common kinds of errors, but Perl doesn't provide static typing. Unless you're prepared to resort to tied variables (and the performance hit they always entail), you simply can't do something and simple and useful as ensuring a variable can store only integers let alone restrict it to storing enumerated types, specific classes of object, or values satisfying arbitrary value constraints (for example: only storing prime numbers).

No parameter lists

It's rather embarrassing that a language a well-designed, mature, powerful, and popular as Perl still doesn't have something as fundamental as formal parameter lists for subroutines. Fortran had this essential feature half a century ago!

Insufficient OO

Perl's bare-bones OO mechanism lacks encapsulation, safe inheritance, powerful dispatch mechanisms, or aspect-oriented features. Recent techniques like inside-out objects and tools like the Moose framework can help, but they are not built-in, or even in the core distribution, so they also add extra layers of complexity, as well as performance overheads and inconvenience.

On the other hand, Perl's overwhelming strengths are things that we rarely think about (and the fact that we rarely need to think about them is a large part of their strength). But teaching Perl to newcomers from other languages makes those advantages stand out very clearly:

Trouble-free strings

In Perl strings and string manipulation are built into the language, fully integrated with all its other features, and therefore very easy to use. You don't have to worry about memory management or reallocation during operations, or deallocation of a string when it's no longer needed. Strings are easy to manipulate, to extract substrings from, and to search.

Powerful regexes

Speaking of search, Perl's regexes (despite the faults enumerated earlier) do provide an efficient mechanism for recognizing data. In particular, they are remarkably useful for determining the structure of input data and then deciding how to handle that data. Substitutions also provide a very powerful means of manipulating and modifying strings.

The adaptable hash

The hash is almost certainly the single most valuable data type in Perl. It can be used for everything from calculating histograms to implementing tree nodes to performing string translation to passing named subroutine arguments to building look-up tables.

Dynamic typing & automatic casting

These features work so well--and so transparently--that we often forget how valuable they are. We take for granted that it's easy to read in a series of strings, arithmetically add a number to each one, use the results as a set of hash keys, sort those keys numerically, then print them as strings. Perl's type system works very hard to make that complex series of type transformations both invisible and safe.

Flexible subroutines

Although the lack of parameter lists can be frustrating, it can also be very convenient. In Perl it's so easy to create subs that deal with dynamic data, or lists, or optional arguments. Closures (anonymous subroutines that refer to lexically scoped data) are also an extraordinarily powerful tool, which is simply not available in most other popular languages.

The CPAN

No catalogue of Perl's virtues would be complete without a mention of Perl's killer app. The CPAN provides thousands of essential „wheels" that don't have to be constantly reinvented. It has a vast collection of superb high-quality software and frameworks for database manipulation, HTML and XML parsing, networking and web interactions, event-driven programming, configuration and command-line processing, system-level programming, unit testing, and so much more.

All of these features combine to make Perl the extraordinarily usefullanguage that so many of us rely on every day. But they're *not* Perl's greatest strength. As powerful and useful as all those features are, on my recent trip I was also reminded of a feature of Perl that is vastly more important.

Even though I live in Australia, I was able to attend YAPC::Europe in Vienna this year thanks to the efforts of the conference organizers, who found several sponsors to cover my expenses, including one extraordinary individual who personally paid half my airfare. Apart from allowing me to speak at the conference, that generous support gave Larry and I some (all too rare) face-to-face time to thrash out a couple of niggling Perl 6 design issues.

The conference itself was exceptional. Larry and I had a large attendance at our Perl 6 talk, and were delighted with the audience's spirited and intelligent participation. Their questions, suggestions, and other valuable feedback...both during the talk and afterwards...also led to some important changes in the Perl 6 design.

Over the three days of the conference I had the chance to chat with dozens of individual Perl folk. Without exception they were friendly, enthusiastic, interesting, funny, and great company.

Then there was the final auction, during which a large number of individuals and organizations donated their time, expertise, goods, treasures, and money. Together we were able to raise thousands of euros to support local Perl groups, to kick-start next year's conference, to underwrite an entire Winter of Code `http://socialtext.useperl.at/woc/`, and--most significantly--to cover the expenses of a fellow conference attendee who suffered a tragic family loss during the conference.
And *that* is Perl's greatest strength: the amazing world-wide community that, together, we create and sustain and share

The generosity of our community manifests itself in obvious ways, such as the vast array of free software on the CPAN, but also in less obvious ways, as in the vast amount of time and effort donated by the remarkable team of people who are the maintainers (and continual improvers!) of the CPAN.

Likewise, we have the huge variety of social and socio-technical activities of the hundreds of Perl Mongers groups listed on perl.org, but also the tireless behind-the-scenes work of the special few who maintain those sites.

We are privileged that Larry is not only a genius at language design, but an extraordinary designer of communities; that Perl is not only an amazing language, but also an magnet for extraordinary people.

Damian Conway

Die Codebeispiele können mit dem Code

BDMO1

von der Webseite www.foo-magazin.de heruntergeladen werden!

Viel Spaß beim Lesen!

Renée Bäcker

IMPRESSUM

Herausgeber:	Smart Websolutions Windolph und Bäcker GbR Maria-Montessori-Str. 13 D-64584 Biebesheim
Redaktion:	Renée Bäcker, Katrin Blechschmidt, André Windolph
Anzeigen:	Katrin Blechschmidt
Layout:	//SEIBERT/MEDIA
Auflage:	500 Exemplare
Druck:	powerdruck Druck- & VerlagsgesmbH Wienerstraße 116 A-2483 Ebreichsdorf
ISSN Print:	1864-7537
ISSN Online:	1864-7545

INHALTSVERZEICHNIS

Hier werden kurz die Autoren vorgestellt, die zu dieser Ausgabe beigetragen haben.

Renée Bäcker

Seit 2002 begeisterter Perl-Programmierer und seit 2003 selbständig. Auf der Suche nach einem Perl-Magazin ist er nicht fündig geworden und hat so diese Zeitschrift herausgebracht. In der Perl-Community ist Renée recht aktiv - als Moderator bei Perl-Community.de, Organisator des kleinen Frankfurt Perl-Community Workshop und Mitglied im Orga-Team des deutschen Perl-Workshops.

Herbert Breunung

Ein perlbegeisteter Programmierer aus dem ruhigen Osten, der eine Zeit lang auch Computervisualistik studiert hat, aber auch schon vorher ganz passabel programmieren konnte. Er ist vor allem geistigem Wissen, den schönen Künsten, sowie elektronischer und handgemachter Tanzmusik zugetan. Seit einigen Jahren schreibt er an Kephra, einem Texteditor in Perl. Er war auch am Aufbau der Wikipedia-Kategorie: „Programmiersprache Perl“ beteiligt, versucht aber derzeit eher ein umfassendes Perl 6-Tutorial in diesem Stil zu schaffen.

brian d foy

brian ist der Herausgeber des englischsprachigen Perl-Magazins „The Perl Review“ (http://www.theperlreview.com), Autor von „Mastering Perl“ (O‘Reilly Media) und ein Perl-Berater und -trainer bei Stonehenge Consulting Services.

Mortiz Lenz

Moritz Lenz wurde 1984 in Nürnberg geboren. Schon in seiner Schulzeit entwickelte er Vorlieben für Chemie, Physik und Informatik. Inzwischen studiert er Physik mit Nebenfach Informatik in Würzburg. Seit etwa vier Jahren ist Perl seine bevorzugte Programmiersprache.
Zu seinen Lieblingsthemen gehören Kryptografie und Sicherheitsaspekte, reguläre Ausdrücke, Unicode und die Perl 6-Entwicklung.

Max Maischein

Max Maischein ist Baujahr 1973 und studierter Mathematiker. Seit 2001 ist er für die DZ BANK in Frankfurt tätig und betreut dort den Fachbereich Operations und Services im Prozess- und Informationsmanagement.

Ronnie Neumann

Ronnie Neumann nutzt Perl seit seiner Zeit als System- und Netzwerkadministrator. Seit dieser Zeit ist er als User im Forum http://board.perl-comunity.de aktiv. Ronnie nutzt Perl für administrative Tasks und Web-Anwendungsentwicklung. Seit einiger Zeit ist er als Lehrer für arbeitstechnische Fächer an einer Berufsschule in Frankfurt tätig und hat dort leider zu wenig Einsatzmöglichkeiten für Perl.

Mike Schilli

Michael Schilli kümmert sich bei Yahoo in Sunnyvale/Kalifornien um die Perl-Belange des Unternehmens. Er schreibt seit nunmehr fast 10 Jahren die monatliche Perl-Kolumne “Perl-Snapshot“ im Linux-Magazin.

Einiges zum Thema CGI-Sicherheit

Die Sicherheit bei Web-Anwendungen ist ein wichtiges Thema und vor dem Hintergrund dass es keine 100%ig sicheren Webanwendungen gibt, ist es umso wichtiger, die Anwendungen möglichst nah an die 100% zu bringen. Dieser Artikel kann nicht alles zum Thema Sicherheit bei CGI-Programmen aufzeigen, aber einige wichtige Punkte ansprechen.

Benutzereingaben

Es gibt einen Grundsatz, den man immer berücksichtigen sollte: Traue keiner Benutzereingabe!

Als Programmierer sollte man immer denken „Der User will mir schaden“. Man sollte also eine gesunde Paranoia entwickeln. Besonders wichtig ist dabei auch die Erkenntnis, dass die meisten Angriffe auf eine Anwendung nicht „von außen“, sondern „von innen“ kommen. In einer Firma bedeutet das, dass die meisten Angriffe von den eigenen Mitarbeitern ausgehen. Dies sind sehr häufig keine gewollten Angriffe, sondern der Mitarbeiter gibt in einem kleinen Moment der Unachtsamkeit etwas Falsches ein, oder er weiß gar nicht, dass er mit einer bestimmten Eingabe etwas Böses anrichten kann.

Bei der Überprüfung von Benutzereingaben sollte nach Möglichkeit das „Whitelist“-Verfahren verwendet werden. Dabei werden die Eingaben daraufhin überprüft, ob sie nur *erlaubte* Zeichen enthält. Im Gegensatz dazu wird häufig das „Blacklist“-Verfahren angewendet, bei dem die Eingaben daraufhin überprüft werden, ob sie *nicht-erlaubte* Zeichen enthalten. Auf den ersten Blick sehen diese beiden Verfahren sehr ähnlich aus, aber bei genauerer Betrachtung sieht man, dass bei dem „Blacklist“-Verfahren auch eher mal unerwünschte Eingaben durchrutschen, weil nicht *alle* Zeichen als „nicht erlaubt“ markiert sind, die eine negative Auswirkung auf das System haben können.

Bei dem „Whitelist“-Verfahren werden vielleicht „zu viel“ Eingaben abgeblockt. Aber das ist leichter zu beheben als ein kompromittiertes System nach einer unvollständigen „Blacklist“.

CGI.pm

Als allererstes sollte man das Modul CGI.pm verwenden. Es nimmt einem viel Arbeit ab und reduziert so die Fehlerwahrscheinlichkeit. Das Modul wird schon seit einigen Jahren entwickelt und hat somit einen sehr guten Stand. In vielen CGI-Skripten findet man noch eine Funktion „ReadParse“ (Listing 1), die wahrscheinlich das schlechteste Überbleibsel aus „Matt‘s Script Archive“ ist. Dort werden die CGI-Parameter geparst und in ein Hash gespeichert. Dieses Parsen ist allerdings nicht so sicher wie die Verwendung der `Vars`-Methode aus dem CGI-Modul.

Da ist

```
use CGI;
my %in = CGI::Vars();
```

viel kürzer, übersichtlicher und sicherer.

Taint-Modus

Man sollte das Skript immer im Taint-Modus laufen lassen. In diesem Modus sorgt Perl dafür, dass Benutzereingaben automatisch als „tainted“ (befleckt) markiert werden und in

kritischen Funktionsaufrufen nicht verwendet werden können. Das Tainting funktioniert allerdings nicht automatisch bei der Verwendung von Modulen mit XS- oder C-Anteil, hier ist Vorsicht geboten.

Um die Skripte im Taint-Modus laufen zu lassen wird `perl` beim Aufruf die Option `-T` mitgegeben. Unter UNIX/Linux geht das einfach in der Shebangzeile am Anfang des Skriptes:

```
#!/usr/bin/perl -T
...
```

Leider gibt es hier einen Unterschied zwischen Linux und Windows. Bei Windows bringt es nichts, wenn das `-T` im Shebang steht, weil die Shebangzeile nicht vor dem Aufruf des Interpreters ausgewertet wird. Die Option muss schon beim Starten des Interpreters bekannt sein. In der Webserverkonfiguration (je nach Webserver unterschiedlich) muss festgelegt werden, wie der Webserver mit den Dateiendungen `.pl` oder `.cgi` umgehen soll. Dort muss das `-T` an den Pfad zum Perl-Interpreter angehängt werden. Dabei ist zu beachten, dass die Einstellung dann nicht nur für ein einzelnes Skript gilt, sondern für alle Skripte im so konfigurierten Teil.

Werden „tainted“ Variablen zum Beispiel bei einem `open`-Aufruf verwendet, erscheint in den Logfiles eine Fehlermeldung, dass potentiell gefährliche Daten verwendet werden:

```
Insecure dependency in open
   while running with -T switch at skript.pl
```

XSS

Ein Schlagwort, das in den letzten Monaten immer bekannter wurde, ist „Cross-Site-Scripting“ (XSS). Beim Cross-Site Scripting wird Code auf Seite des Clients ausgeführt. Daher muss der Angreifer seinem Opfer einen präparierten Hyperlink zukommen lassen, den er zum Beispiel in eine Webseite einbindet oder in einer E-Mail versendet. Gefährlich wird es besonders dann, wenn die Quelle eigentlich vertrauenswürdig ist - zum Beispiel ein Forum, in dem man sich schon seit Jahren bewegt.

Dort könnte der Angreifer die manipulierten Links in eine Privatnachricht packen oder in einen Thread. Deshalb muss hier der Programmierer besonders aufpassen.

```
sub ReadParse {
  # Read in text
  if ($ENV{'REQUEST_METHOD'} eq "GET") {
    $in = $ENV{'QUERY_STRING'};
  } elsif ($ENV{'REQUEST_METHOD'} eq "POST") {
    for ($i = 0; $i < $ENV{'CONTENT_LENGTH'}; $i++) {
      $in .= getc;
    }
  }

  @in = split(/&/,$in);

  foreach $i (0 .. $#in) {
    # Convert plus's to spaces
    $in[$i] =~ s/\+/ /g;

    # Convert %XX from hex numbers to alphanumeric
    $in[$i] =~ s/%(..)/pack("c",hex($1))/ge;

    # Split into key and value.
    $loc = index($in[$i],"=");
    $key = substr($in[$i],0,$loc);
    $val = substr($in[$i],$loc+1);
    $in{$key} .= "\0" if (defined($in{$key})); # \0 is the multiple separator
    $in{$key} .= $val;
  }
}
```

Listing 1

Ein klassisches Beispiel für Cross-Site Scripting ist die Übergabe von Parametern an ein CGI-Skript einer Website. Ein kleines Beispiel ist in Listing 2 zu sehen, während Listing 3 zeigt, wie einfach diese Lücke geschlossen werden kann. Wer die möglichen Auswirkungen mal testen will, kann in das Eingabefeld `<script language="javascript">alert('test');</script>` eingeben und das Formular abschicken.

HTML und BBCode

Injizierter HTML- und Javascript-Code kann zu Angriffen auf die Client-Seite einer Anwendung verwendet werden. Wenn man gewisse HTML-Elemente zulassen will (z.B. in einem Forum), ist es geschickt, eines der BBCode-Module von CPAN zu benutzen. So kann man alle HTML-Elemente, die der Benutzer direkt eingibt, „unschädlich" machen, z.B. mit HTML::Entities.

HTML-Escaping

Die Template-Module HTML::Template und HTML::Template::Compiled bieten auch die Möglichkeit, ein „default_escape" zu setzen. Ist der Standard-Wert auf ‚html' eingestellt, werden automatisch bei allen Parametern die Sonderzeichen in Entities umgewandelt. Das vereinfacht das Verhindern von XSS.

```
#!/usr/bin/perl

use strict;
use warnings;
use CGI;

my $cgi = CGI->new;
print $cgi->header;
my %params = $cgi->Vars;

if( $params{action} ){
    print "Sie haben $params{input} eingegeben";
}
else{
    print qq~
        <form method="post">
            <input type="text" name="input" />
            <input type="hidden" name="action" value="1" />
            <input type="submit" value="abschicken" />
        </form>
    ~;
}
```

Listing 2

```
#!/usr/bin/perl

use strict;
use warnings;
use CGI;
use HTML::Entities;

my $cgi = CGI->new;
print $cgi->header;
my %params = $cgi->Vars;

if( $params{action} ){
    my $var = HTML::Entities::encode_entities( $params{input} );
    print "Sie haben $var eingegeben";
}
else{
    print qq~
        <form method="post">
            <input type="text" name="input" />
            <input type="hidden" name="action" value="1" />
            <input type="submit" value="abschicken" />
        </form>
    ~;
}
```

Listing 3

Datenbanken

In Datenbanken werden häufig vertrauliche Daten gespeichert - seien es Namen und Adressen von Kunden oder Zugangsdaten zu einer Plattform. Deshalb sollte hier ein besonderer Augenmerk auf der Sicherheit liegen. Angreifer versuchen mit sogenannten SQL-Injections Informationen über die Datenbank und die Inhalte zu bekommen.

Das DBI-Modul bietet die wunderbare Möglichkeit, die ?-Notation (Platzhalter) zu verwenden. Damit werden automatisch alle Sonderzeichen gequotet und die Möglichkeit des Injizierens von SQL-Code verhindert. Das Gleiche kann mit der Funktion `quote` aus dem Modul erreicht werden. Weiterhin gibt es eine (experimentelle) Möglichkeit, Ein- und Ausgabewerte als „tainted“ zu markieren (Optionen TaintIn, TaintOut, Taint).

Als Beispiel für eine mögliche Gefahr soll hier an einem Login-Vorgang gezeigt werden, wie gefährlich dies sein kann (Listing 4) und wie es sicherer (Listing 5) ist.

```
my $stmt = "SELECT count(*) FROM users
   WHERE id = '$name' and
         password = "$password'";
my $sth  = $dbh->prepare( $stmt );
$sth->execute;
```

Listing 4

Hier werden Formulareingaben nicht auf SQL-Sonderzeichen überprüft und einfach in den SQL-Befehl eingebaut. Der User soll hier als authentifiziert gelten wenn `count` größer 1 ist. Wenn ein Angreifer sowohl für `$name` als auch für `$password ' OR '1' = '1` eingibt, würde das diesen SQL-Befehl erzeugen:

```
SELECT count(*) FROM users
    WHERE id = '' OR '1' = '1' AND
    password = '' OR '1' = '1'
```

Das liefert die Anzahl der Einträge in der Tabelle users. Der Angreifer wäre eingeloggt.

```
my $stmt = "SELECT count(*) FROM users
   WHERE id = ? and
         password = ?";
my $sth  = $dbh->prepare( $stmt );
$sth->execute( $name, $password );
```

Listing 5

Durch die `?` wird das quoting hervorgerufen und DBI macht die Sonderzeichen unschädlich. Für das obige Beispiel wird so folgender SQL-Befehl ausgeführt:

```
SELECT count(*) FROM users
   WHERE id = '\' OR \'1\' = \'1' AND
   password = '\' OR \'1\' = \'1'
```

So wäre der Angreifer nicht eingeloggt.

Öffnen einer Datei

Grundsätzlich sollte man nicht den User bestimmen lassen, wie der Dateiname lautet. Man sollte eigentlich immer selbst festlegen, wie die Datei heißt und wo sie hingespeichert wird.

Außerdem ist die Drei-Parameter-Form von open() (also `open (FILEHANDLE, ">", $dateiname)`) sicherer, da das Umlenkungszeichen damit eindeutig festgelegt wird. Beim Verwenden des Taint-Modus wird man automatisch vor der Verwendung von unsicheren Dateinamen geschützt.

Öffnen einer Pipe

Wenn z.B. mit sendmail eine E-Mail verschickt werden soll, wird gerne der typische Fehler gemacht, die E-Mail-Adresse des Empfängers direkt in die Kommandozeile zu schreiben:

```
open PIPE, "|/usr/lib/sendmail $empfaenger";
```

Was fällt auf? Genau, `$empfaenger` ist hier vermutlich eine Benutzereingabe. Der sollten wir niemals trauen. In die Kommandozeile gehören nur selbst festgelegte Parameter. Den Empfänger kann man auch mit der Option -t und einer „To:“-Zeile bestimmen.

```
# Etwas besser, aber noch nicht gut:
open PIPE, "|/usr/lib/sendmail -t" or die $!;
print PIPE <<EOM;
To: $empfaenger
Subject: Blubber

Hallo $name,
usw.
EOM
...
```

Aber auch das ist nicht sicher.
Denn $empfaenger oder $name kann mehrere Zeilen beinhalten, die noch ein paar zusätzliche Header-Zeilen wie z.B. „Bcc: boeser_bube@domain.example“ beinhalten oder auch eine komplette E-Mail hinzufügen kann. sendmail betrachtet eine E-Mail als fertig, wenn im Body der Punkt alleine auf einer Zeile vorkommt.

Also, Usereingaben gehören auch nicht ungeprüft in den Header einer E-Mail, und sendmail startet man mit der Option „-oi“, die das mit dem erwähnten Punkt verhindert.

```
# Prüfe $empfaenger auf Gültigkeit,
# z.B. mit Email::Valid
# Ersetze stumpf alle unerwünschten Zeichen:
$empfaenger =~ tr/\r\n\t\f\0//d;
open PIPE, '|-',
   '/usr/lib/sendmail -t -oi' or die $!;
print PIPE <<EOM;
From: test@test.com
To: $empfaenger
Subject: Blubber

Hallo $name,
usw.
EOM
...
```

Häufig kann man solche Pipe-Verwendungen vermeiden, indem man z.B. auf Module zurückgreift. Im Falle von „Mails versenden“ kann man z.B. das Mail::Sender-Modul benutzen.

Verwendung von eval

Man sollte im Allgemeinen auf String-evals mit Daten aus unsicheren Quellen verzichten, da der String eine „böse“ Zeichenfolge beinhalten könnte. Auch hier wird man im Taint-Modus geschützt.

Wenn doch ein String-eval mit unbekannten Inhalt verwendet werden soll, kann man auf das Modul Safe zurückgreifen, mit dem man bestimmte Perl-Funktionen (Opcodes) erlauben oder ausschalten kann. Es ist allerdings möglich, dass Safe Sicherheitslücken hat!

Neueste Perl-Version

Auch das Perl-Binary kann typische, aus der C-Welt bekannte Fehler wie Buffer Overflows enthalten. Deshalb sollte man möglichst die neueste Perl-Version verwenden. Bei kritischen Bugfixes werden auch die älteren Stränge (5.6.x, 5.003_xx) aktualisiert.

Aktuelles

Wie viele Schwachstellen es in Webanwendungen gibt, kann man nur erahnen. Auf dem 24. Chaos Computer Congress (24C3) haben Hacker etliche Webseiten „angegriffen“ und kleine Veränderungen vorgenommen. Unter http://events.ccc.de/congress/2007/Hacks sind weit über 150 verschiedene Hacks aufgeführt, die während des Kongresses gemacht wurden. Viele dieser Angriffe sind SQL-Injections und XSS-Attacken. Aber auch Fälle, in denen über die URL `/etc/passwd` geöffnet werden konnten.

Um solche Attacken durchzuführen muss man kein Spezialist sein. Es sind häufig ganz einfache Sachen, die man über jede Suchmaschine finden kann.

Renée Bäcker

TPF-TICKER

Parrot Grant Update

Es gibt wieder Neuigkeiten von Parrot (http://www.parrotcode.org):

Am 17. Oktober wurde Parrot 0.4.17 veröffentlicht. Neben einigen Bugfixes wurde auch an NQP weitergearbeitet. NQP ist eine abgespeckte Version von Perl6, die sich zum Compilerbau eignet.

Einige Design-Milestones wurden erreicht, so dass auch wieder Geld vergeben wurde.

Die Roadmap und eine Übersicht über das ausgegebene Geld ist unter http://www.perlfoundation.org/parrot_grant_from_nlnet zu finden.

Attribute in Perl

Seit Perl 5.5 gibt es in Perl sogenannte Attribute, aber vielen dürften sie erst bei `Catalyst` aufgefallen sein. Attribute sind also schon seit vielen Jahren im Perl-Core enthalten, aber erst in den letzten zwei, drei Jahren beschäftigen sich mehr und mehr Programmierer mit dem Thema - und das meist nur als „Anwender“ ohne selbst Attribute implementiert zu haben. In `Catalyst` werden Attribute für Subroutinen vergeben, so dass diese Subroutinen nur dann etwas machen, wenn beispielsweise der aufgerufene Pfad bei einem Regulären Ausdruck matcht. Wie eine solche Subroutine aussieht, ist in Listing 1 dargestellt.

```
sub foo : Local {
   # mach was
}

sub bar : Path('/test/'){
   # mach was wenn der Pfad '/test/' enthält
}
```

Listing 1

Aber Attribute können nicht nur für Subroutinen vergeben werden, sondern auch für Skalare, Hashes und Arrays. Und es können mehrere Attribute für eine Variable angewendet werden, so dass `my $x :Attr1 : Attr2 = 1;` möglich ist. An dem Beispiel ist auch zu erkennen, dass vor jedem Attribut ein Doppelpunkt steht. Attributnamen und Doppelpunkt können auch durch ein oder mehrere Leerzeichen getrennt sein.

In der Dokumentation von `attributes` ist vermerkt, dass Attribute für Variablen noch in der Entwicklung sind und sich die Anwendung noch ändern kann. Allerdings hat sich in den letzten Jahren nicht wirklich was getan und es gibt Module, die es für den Programmierer einfacher machen mit Attributen zu arbeiten.

Es gibt drei (Core-)Wege, wie Attribe in eigene Programme eingeführt werden können:

- attrs
- attributes
- Attribute::Handlers

Es gibt noch weitere Möglichkeiten, die auf diesen drei Wegen basieren. Ein weiteres Attribute-Modul, das später vorgestellt wird, ist Attribute::Method::Typeable.

attrs

Das Pragma attrs wurde in Perl 5.5 eingeführt und ist mittlerweile als „deprecated“ eingestuft. Aus diesem Grund wird dieses Pragma hier nur der Vollständigkeithalber erwähnt.

attributes

Seit Perl 5.6 ist das Pragma attributes dem Vorgänger „attrs“ vorzuziehen. attributes stellt einige Built-in Attribute zur Verfügung, wie zum Beispiel `method` oder `lvalue`.

Das `method`-Attribut spielt zum Beispiel bei mod_perl2 eine wichtige Rolle, wenn eigene mod_perl2-Handler geschrieben werden. Besitzt eine Methode das `method`-Attribut, so übergibt der Apache zwei Parameter an diese Methode. Mit nur kleinen Änderungen an der Apache-Konfiguration, ist ein eigener Response-Handler eingesetzt.

Eine weitere Einsatzmöglichkeit für `method` ist die „Unterscheidung“ zwischen Core-Funktionen und (Objekt-)Methoden. In Listing 3 ist ein Skript zu sehen, dass die Warnung „`Ambiguous call resolved as CORE::warn(), qualify as such or use & at C:\method.pl line 13.`“

ausgibt. Perl weist den User darauf hin, dass eine Funktion mit dem gleichen Namen wie eine Core-Funktion geschrieben wurde. Perl führt aber trotzdem die Core-Funktion aus.

```
#!/usr/bin/perl

use strict;
use warnings;

my %hash = ( test => 'hallo' );

sub warn{
   my ($class,$msg) = @_;
   print "Warnung: $msg\n";
}

warn( $hash{test} );
```

Listing 2

Soll die Warnung unterdrückt werden, weil die User-definierte Subroutine nur als Methode einer Klasse (`Klasse->warn('hallo')`) oder eines Objekts (`$obj->warn('hallo')`) aufgerufen wird - und nicht als Funktion -, kann die Subroutine mit dem `method`-Attribut versehen werden. Damit weiß Perl, dass bei einem Aufruf von `warn` als Funktion die Core-Funktion gemeint ist und bei einem Aufruf als Methode soll die User-definierte Subroutine ausgeführt werden.

Mit dem `lvalue`-Attribut kann eine Subroutine als `lvalue` verwendet werden. D.h. man kann ihr einen Wert zuweisen. Allerdings ist das eher selten und es ist für Programmierer nicht sofort ersichtlich was da gemacht wird. Der „Nutzen“ (Einsparen von Code) ist im Vergleich zu den „Kosten“ (Wartbarkeit, Lesbarkeit) eher gering. In Listing 3 wird gezeigt, wie eine Subroutine auch als `lvalue` verwendet werden kann.

```
#!/usr/bin/perl

use strict;
use warnings;
use attributes;

{
   package Test;

   my $name;
   sub name : lvalue {
      $name;
   }
}

print "Weise der Sub 'lvalue sub' zu...\n";
Test::name() = 'lvalue sub';
print "Returnwert der Sub: "Test::name();
```

Listing 3

Versucht man einer „normalen“ Subroutine einen Wert zuzuweisen, bricht Perl die Compilierung ab (Listing 4).

```
C:\>lvalue_test.pl
Can't modify non-lvalue subroutine call in scalar assignment at
C:\lvalue_test.pl line 18, near "'lvalue sub';"
Execution of C:\lvalue_test.pl aborted due to compilation errors.
```

Listing 4

Sollen alle Attribute einer Subroutine oder einer Variablen ausgegeben werden, kann man die Funktion `get` aus attributes verwenden (Listing 5).

```
#!/usr/bin/perl

use attributes;

sub test : method : lvalue{
   # any code
}

print $_,"\n" for attributes::get(\&test);
```

Listing 5

Eine weitere Funktion in `attributes` ist `reftype`, was ähnlich wie die Core-Funktion `ref` ist. Allerdings liefert `reftype` bei einem geblessten Hash nicht den Namen der Klasse, sondern `HASH`. Diese Funktion ist ganz nützlich, wenn man feststellen muss, für welche Art von Variable ein Attribut erstellt werden soll.

Attribute::Handlers

Dieses Modul wurde in der Version 5.7.3 in den Perl-Core aufgenommen und erleichtert die Anwendung von Attributen. Die weiteren Beispiele in diesem Artikel werden mit diesem Modul umgesetzt.

Der große Vorteil von Attributen ist, dass - je nach Anwendung - Code gespart werden kann und Änderungen an der Funktionalität sehr schnell vorgenommen werden können.

Eine Anwendung

Als Beispielanwendung soll eine kleine Pseudoshell geschrieben werden, bei der Administratoren Dateien verschieben dürfen, normale User aber nicht. Diese Pseudoshell soll diese Anweisungen nicht wirklich ausführen, sondern nur ausgeben was sie tun würde.

Diese Anwendung wird hier in drei Teilen geschrieben. Der erste Teil ist ein Modul, das die Attribute implementiert (Listing 6).

Die Attribute werden mit `Attribute::Handlers` umgesetzt. In diesem Codeausschnitt wird nur die Subroutine gezeigt, die für das Attribut „Administrator“ zuständig ist. Die Funktion für „User“ sieht sehr ähnlich aus. Der Name der Subroutine legt den Namen des Attributs fest. Diese Subroutine hat selbst Attribute - die besagen, für welchen Typ das Attribut umgesetzt werden soll. In diesem Fall soll das Attribut bei Subroutinen verwendet werden. Deswegen bekommt das Attribut noch `CODE` übergeben. Soll ein Attribut für einen Skalar implementiert werden, so muss es `ATTR(SCALAR)` heißen. Für andere Variablentypen geht das entsprechend und es können mehrere Typen kombiniert werden.

`Attribute::Handlers` übergibt mehrere Parameter an die Subroutine. Das ist zum Einen der package-Name in dem das Attribut verwendet wird. In diesem Beispiel wird ‚TestShell‘ übergeben. Als nächstes wird eine Referenz auf den Typeglob übergeben, in dem die Subroutine steht. Der dritte Parameter ist eine Referenz auf die Subroutine, die dieses Attribut verwendet. Weiterhin werden noch Attributname (zum Beispiel ‚Administrator‘), die mit dem Attribut verbundenen Daten und der Name der Phase übergeben.

Die mit dem Attribut verbundenen Daten sind entweder ein String oder eine Arrayreferenz - je nach Anzahl übergebener Daten. Bei `sub test : Test(hallo) {}` wird die ‚hallo‘ als ein String übergeben, weil es nur ein einziger übergebener Wert ist. Sind es mehrere Werte, wird eine Arrayreferenz übergeben (zum Beispiel `sub test : Test(qw(1 2)) {}`).

```
package TestShell;

use base qw(AttributeTest);

sub move : Administrator{
   my ($self,$file1,$file2) = @_;
   print "Verschiebe $file1 nach $file2\n";
}

sub ls : User{
   my ($self,$dir) = @_;
   print "Liste files in $dir auf\n";
}
```

Listing 7

```
package AttributeTest;

use strict;
use warnings;
use Attribute::Handlers;

sub Administrator : ATTR(CODE){
   my ($pkg, $sym, $code) = @_;

   my $name = *{ $sym }{NAME};

   no warnings 'redefine';

   *{ $sym } = sub {
       unless ( $_[0]->type eq 'admin' ){
          level_error( 'admin' );
          return;
       }
       my @ret = $code->( @_ );
       return @ret;
   };
}
```

Listing 6

Daran kann man schon erkennen, dass Attribute sehr nützlich sind, da in den Methoden `move` und `ls` die Überprüfung auf das notwendige User-Level nicht mehr implementiert werden muss. Wenn - wie in diesem Beispiel - nur eine einzige Methode pro User-Level gibt, sind Attribute ein gewisser „Overkill“, aber sobald mehrere Methoden geschrieben werden sollen, die eine Kontrolle benötigen, spart das viel Zeit und es ist schon auf den ersten Blick zu erkennen, welche Methode welches Mindestlevel hat.

`TestShell` muss von `AttributeTest` erben, damit die Attribute funktionieren. Wenn man auf Vererbung verzichten möchte, müssen die Funktionen von `AttributeTest` in die `TestShell` geschrieben werden.

Das Modul hat noch weitere Methoden, die aber für die Attribut-Funktionalität uninteressant sind.

Das Testskript verwendet dann das Modul `TestShell` und versucht als Administrator und dann als User die beiden Methoden auszuführen.

```
use TestShell;

my $admin = TestShell->new( 'admin' );
my $user  = TestShell->new( 'user' );

$admin->move( 'test1.txt', 'test2.txt' );
$admin->ls( '/home/user' );

$user->move( 'test1.txt', 'test2.txt' );
$user->ls( '/home/user' );
```

Listing 8

An der Ausgabe ist zu erkennen, dass immer erst die Level-Überprüfung stattfindet (siehe Listing 9)

```
C:\>example.pl
Verschiebe test1.txt nach test2.txt
Liste files in /home/user auf
Mindestlevel admin nicht erreicht
Liste files in /home/user auf
```

Listing 9

Weitere Beispiele für Attribute in Anwendungen sind `Catalyst`, `CGI::Application`, `Test::Class` und viele mehr. Bei `CGI::Application` gibt es das Plugin `AutoRunmode`, mit dem über Attribute bestimmt wird, welche Subroutinen als Runmode „registriert“ werden sollen und welche Subroutine der Startrunmode ist. Mit diesem Plugin ist es nicht mehr nötig, bei jeder neuen Subroutine das Setup von `CGI::Application` anzupassen.

Bei `Test::Class` wird über Attribute bestimmt, welche Subroutine vor einem Test laufen muss und welche Subroutine für das Aufräumen nach den Tests zuständig ist. Weiterhin kann festgelegt werden, wie viele Tests in einer Subroutine ausgeführt werden:

```
use base qw(Test::Class);

sub vor_test : Test(setup) {
    # zusichern/einhalten
    # von constraints
};

sub vier_tests : Test(4) {
    # vier tests
};

sub aufraeumen : Test(teardown) {
    # ueberreste aufraeumen
};
```

Listing 10

Wie oben angekündigt, soll auch ein Beispiel mit `Attribute::Methode::Typeable` gezeigt werden. Dieses Modul sollten sich vor allem diejenigen anschauen, denen es nicht gefällt, dass Perl nicht so wirklich typisiert ist. Mit dem Modul können Typen für Subroutinen und Variablen bestimmt werden, wie man es von Java her kennt.

Intern arbeitet das Modul mit `Attribute::Handlers` und `Hook::WrapSub`. Letzteres ist dafür da, um das `*{ $sym } = sub {}` zu sparen. Das übernimmt das `Hook::WrapSub`.

Bei dem Modul aus Listing 11 wird eine Methode als „Public“ gekennzeichnet und sie erwartet zwei Integer. Und eine weitere Methode, die „Private“ ist und einen String erwartet. `Attribute::Methode::Typeable` übernimmt dann die Kontrolle, ob die Methode überhaupt ausgeführt werden darf und ob die Parameter stimmen (Anzahl und Typ).

```
sub mal :Public( Integer Integer ) {
   my ($self,$a,$b) = @_;
   print $a * $b;
}

sub greet :Private( String ) {
   my ($self,$who) = @_;
   print "Hello $who!\n";
}
```

Listing 11

Renée Bäcker

Zehnter Deutscher

Perl-Workshop

13.02.- 15.02.2008

www.perl-workshop.de

Regionales Rechenzentrum Erlangen

Größere Datenmengen in Bilder gepresst

Ein großes Einsatzgebiet für Perl ist die Systemadministration. Es gibt auch einige Tools, die die Admins bei ihrer Arbeit unterstützen. Monitoring-Tools bieten die Informationen in verschiedensten Formaten an - unter anderem auch in verschiedenen Graphen. Solche Darstellungen eignen sich besonders gut, wenn dem User etwas präsentiert werden soll, das auf den ersten Blick zu erfassen ist. Nicht umsonst heißt es „1 Bild sagt mehr als 1000 Worte“.

Dieser Artikel schlägt zwei Fliegen mit einer Klappe: Zum Einen werden die Ergebnisse der Perl-Umfrage 2007 für den deutschsprachigen Raum (Deutschland, Österreich, Schweiz) aufbereitet und zum Anderen wird damit gleich gezeigt, wie schnell mit Perl Grafiken (hier nur Graphen) erstellt werden können. Wer selbst etwas mit den Daten der Umfrage arbeiten will, kann diese auf der Webseite http://perlsurvey.org/results/ herunterladen. Dort sind die Daten in mehreren Formaten hinterlegt.

Hier werden die Daten im CSV-Format verwendet. Aber da es auch für CSV einen DBI-Treiber gibt, können die Skripte mit nur wenigen Änderungen auch im Zusammenspiel mit -anderen- richtigen Datenbanken verwendet werden.
Nicht beachtet werden Personen, die aus dem deutschsprachigen Raum in das nicht-deutschsprachige Ausland gezogen sind, da es sonst zu unübersichtlich wird. Betrachtet werden soll die Herkunft der Personen im deutschsprachigen Raum, die Verteilung auf Branchen, die Altersverteilung, das Einkommen und Teilnahme an Perlmonger-Treffen und Konferenzen.

Als erstes soll die Herkunft der Umfrage-Teilnehmer betrachtet werden. Dabei ist es auch ganz interessant, wie viele Personen aus dem Ausland in den deutschsprachigen Raum gekommen sind.

Es gibt mehrere Module für die Generierung von Grafiken wie zum Beispiel `Image::Magick`, aber ich persönlich finde die GD-Module gerade im Bereich „Graphen“ sehr hilfreich und einfach. Es gibt zwar einige Stellen, an denen die Module den User mehr unterstützen könnten, aber für die meisten Anwenundungen sind die Module sehr gut geeignet. Es können viele unterschiedliche Arten von Graphen erzeugt werden.

Das Skript in Listing 1 erstellt ein sogenanntes Tortendiagramm für die Herkunft der einzelnen Personen. Für die Erstellung des Bildes wird das Modul `Chart::Pie` verwendet. In den Zeilen 9 - 18 werden die notwendigen Daten aus der „CSV-Datenbank“ geholt. Da der CSV-Treiber ein paar Beschränkungen hat, wurde die CSV-Datei vorher ein klein wenig bearbeitet. So wurden in den Spaltennamen alle Zeichen durch ein Unterstrich ersetzt, die nicht alphanumerisch oder ‚_‘ waren.

Die Sortierung muss der Programmierer selbst machen, da die Module die Daten in der Reihenfolge in die Grafik bringen, wie sie übergeben werden.

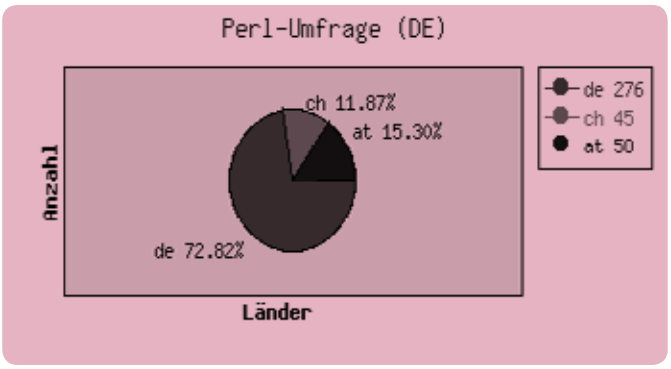

Abbildung 1: Herkunft (Tortendiagramm)

```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
use Chart::Pie;

my $png  = 'residence.png';
my $dbh  = DBI->connect("DBI:CSV:f_dir=./;csv_sep_char=\\,");

my $stmt = q~SELECT Country_of_residence FROM perlsurvey_de~;
my $sth  = $dbh->prepare( $stmt );
$sth->execute;

my %hash;
while( my ($country) = $sth->fetchrow_array ){
   $hash{$country}++;
}

my @countries = keys %hash;
my @values    = values %hash;

my @sorted    = sort{ $values[$b] <=> $values[$a] }0..$#values;

my ($width,$height) = (400,200);


my $chart = Chart::Pie->new($width,$height);

$chart->set(
    title            => 'Perl-Umfrage (DE)',
    x_label          => 'Länder',
    y_label          => 'Anzahl',
    transparent      => 'false',
    min_val          => 0,
    legend           => 'right',
);


$chart->add_dataset( @countries[@sorted] );
$chart->add_dataset( @values[@sorted] );

$chart->png( $png );
```

Listing 1

Danach wird ein neues Objekt von `Chart::Pie` erzeugt; dabei wird die Breite und die Höhe der Grafik übergeben. Werden keine Größen übergeben, wird die Standardgröße 400x300 Pixel genommen. Danach werden noch einige Zusatzoptionen gesetzt, damit die Grafik aussagekräftiger wird. Zu beachten ist, dass keine mehrzeiligen Titel verwendet werden können. Die Angabe von `min_val` empfiehlt sich in den meisten Fällen, da sonst der niedrigste Wert aus den Datensätzen als „0-Punkt" der y-Achse genommen wird.

Als nächstes werden die Daten für die x-Achse übergeben und danach die dazugehörigen Werte. Das Ergebnis ist in Abbildung 1 zu sehen.

In Listing 2 ist ein Ausschnitt aus dem Skript zu sehen, das die Grafik für die Geburtsländer erstellt (siehe Abbildung 2). Es gibt ein paar geringe Unterschiede zu dem Skript vorher: Zum Einen wird ein anderes Modul verwendet, nämlich `Chart::StackedBars`, das ein Balkendiagramm erstellt, und zum anderen wurde hier ein Label für die Legende festgelegt. Weiterhin wird `skip_int_ticks` verwendet, um die y-Achse übersichtlicher zu gestalten. Ohne diese Option würde bei jeder Integer-Zahl ein Strich sein. Mit dieser Option wird nur bei jeder 20. Zahl ein solcher Strich gemacht.

häufige Fehler

Die häufigsten Fehlermeldungen, die bei den Tests aufgetaucht ist, sind

```
The number of legend labels and datasets
     doesn't match at C:\birth.pl line 48
Unknown named color () for dataset-1
  at C:\birth.pl line 48
```

```
my $chart = Chart::StackedBars->new($width,$height);

$chart->set(
    title                   => "Perl-Umfrage (DE) - Geburtsland",
    x_label                 => 'Länder',
    y_label                 => 'Anzahl',
    transparent             => 'false',
    legend_labels           => [ 'Anzahl der Personen' ],#$legend,
    y_ticks                 => 10,
    min_val                 => 0,
    legend                  => 'right',
    integer_ticks_only      => 1,
);

$chart->set( skip_int_ticks => 20 );
```

Listing 2

Die erste Fehlermeldung weist daraufhin, dass man mehr Label als Datensätze hat. Die Fehlerursache ist dabei aber nicht immer einfach - auch weil das Modul nicht auf die wirkliche Ursache des Problems hinweist. Hier muss man immer ein wenig „spielen", bis es passt. Die nächste Fehlermeldung betrifft die Legende. Hier teilt uns das Modul mit, dass nicht genügend Farben für die Legende zur Verfügung stehen. Also muss man selbst die Farben festlegen und an das Modul übergeben.

Mit Graphen können größere Datenmengen leicht aufbereitet werden und Perl zusammen mit GD bieten gute Tools, um solche Graphen schnell und einfach zu generieren. Durch die GDGraph-Module haben viele Formen von Graphen die gleiche API, so dass die Darstellungsart sehr schnell geändert werden kann. Wie schon weiter oben erwähnt gibt es allerdings auch Stellen, an denen GD den User mehr unterstützen könnte. So wäre es wünschenswert, wenn die Module (optional) eine Warnung ausgeben würden, falls die Balken

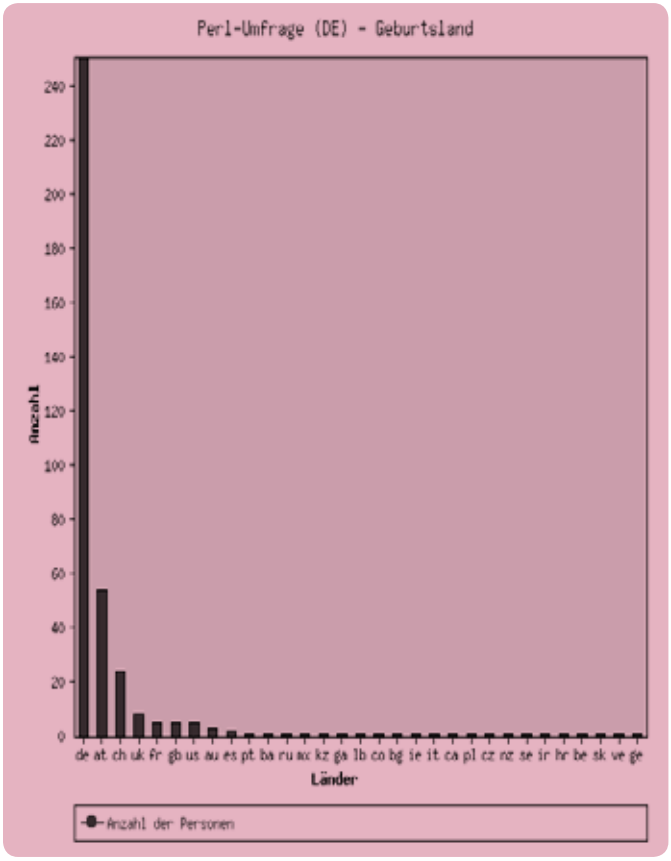

Abbildung 2: Geburtsländer (Balkendiagramm)

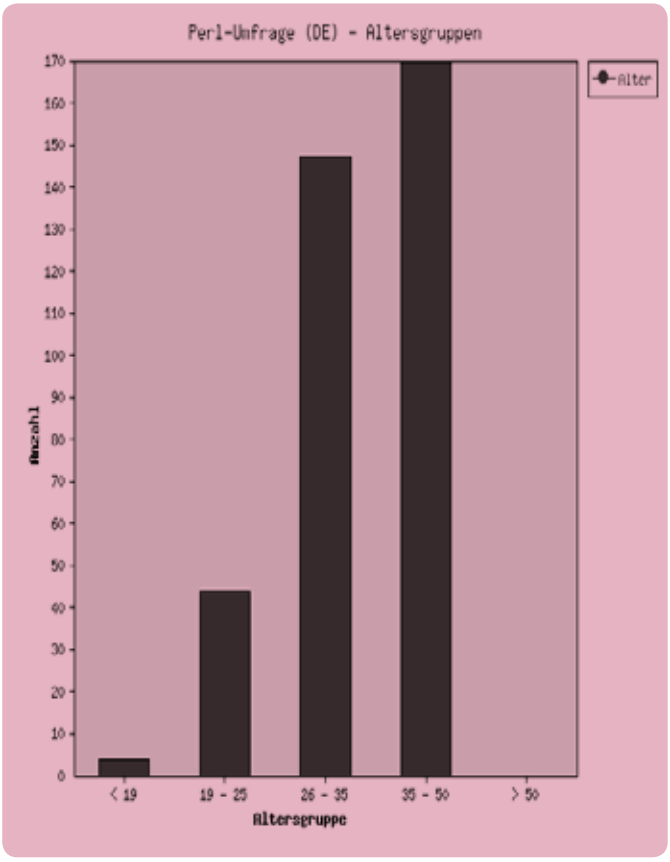

Abbildung 3: Alterstruktur (Balkendiagramm)

zu schmal werden um noch irgendwas zu erkennen. Auch dass nach knapp über 60 Farben einfach Schluss ist und nicht „von vorne“ angefangen wird, ist eher schlecht. Man kann zwar selbst Farben bestimmen, aber als default wäre es gut, wenn das Modul dem Programmierer möglichst viel Arbeit abnimmt.

Auch wenn die Legende nicht komplett angezeigt werden kann, gibt das Modul keine Warnung aus. Die Legende wird einfach abgeschnitten.

Die Umfrage-Ergebnisse

Insgesamt haben 379 Personen aus Deutschland, Österreich und der Schweiz (Wohnort) an der Perl-Umfrage 2007 teilgenommen. Aus den Abbildungen 1 und 2 geht hervor, dass die meisten Teilnehmer im deutschsprachigen Raum bei der Perl-Umfrage in Deutschland wohnen. Die meisten sind auch in Deutschland geboren (86.81%), aber es sind auch 13.19% aus anderen Ländern in den deutschsprachigen Raum gezogen.

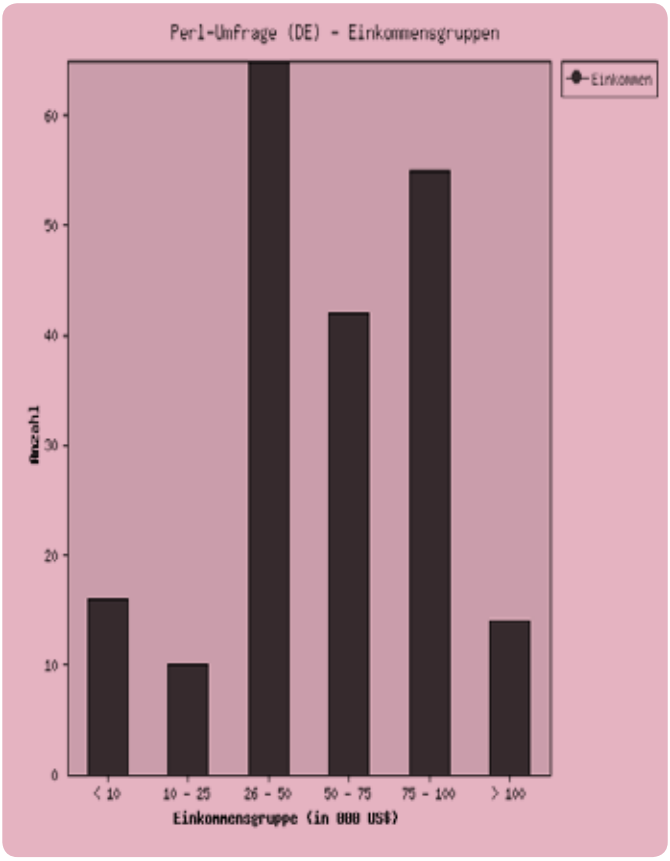

Abbildung 4: Einkommen (Balkendiagramm)

Ganz interessant ist die Betrachtung der Altersgruppen (Abbildung 3). Diese Grafik kann man auf mehrere Arten deuten. Zum Einen, dass Perl kaum ganz junge Nachwuchsprogrammierer hat, was ein Grund zur Sorge ist. Aber eine andere Lesart wäre zu sagen, dass Perl selten als Einsteigersprache dient, sondern erst im Laufe der Zeit entdeckt wird. Um sagen zu können, welche Deutung „richtig“ ist, müsste die Perl-Umfrage in den nächsten Jahren immer wiederholt werden. Wenn man sich Foren u.s.w. anschaut, kann man erkennen, dass beides irgendwie richtig ist. Es gibt sicherlich einige Sprachen, die „hipper“ und „cooler“ sind - auf den ersten Blick. Viele kommen erst im Berufsleben zu Perl: dann wenn etwas automatisiert werden soll.

Und es scheint, als könne man mit Perl-Programmierung ganz gut leben (Abbildung 4), denn immerhin verdient über die Hälfte (54.95%) derjenigen, die eine Angabe dazu gemacht haben, über 50.000 US$ pro Jahr.

Es bleibt zu hoffen, dass so eine Umfrage in jedem Jahr gemacht wird und sie auch immer mehr Personen erreicht - auch diejenigen, die nicht jeden Tag in den einschlägigen Foren unterwegs sind.

Renée Bäcker

Rezension - Mastering Perl

Perl-Programmierer auf dem Weg vom Novizen zum Meister - das ist die Reihe „Learning Perl“, „Intermediate Perl“ und „Mastering Perl“. Diese Rezension beleuchtet das letzte Buch der Reihe - „Mastering Perl“. Dieses Buch soll den Leser auf den „richtigen Weg zum Perl-Meister“ bringen; dazu werden Themen behandelt, die schon Perl-Erfahrung voraussetzen.

Auf rund 300 Seiten werden in 18 Kapiteln Themen wie Perl-Debugger, erweitertes RegEx-Wissen und vieles mehr besprochen. Den Anfang machen dabei die Regulären Ausdrücke. Sicherheitsregeln sollten meiner Meinung nach aber schon früher besprochen werden, da das gerade bei Anfängern ein häufiger Fehler ist. Mit den Themen Benchmarking und Profiling werden jedoch zwei Bereiche besprochen, die für den „letzten Schliff“ von Programmen wichtig ist und wirklich etwas für Profis ist. Dabei geht der Autor auf Perl-Internas ein und erklärt deren Bedeutung.

Größtenteils wird hier tatsächlich Wissen vermittelt, mit dem nur „ernsthafte“ Perl-Programmierer in Berührung kommen. Der Einsteiger wird sich erstmal wenig um den Perl-Debugger kümmern oder um Benchmarking. Für Programmierer auf dem Weg zum sehr guten Perl-Programmierer ist solches Wissen aber sehr nützlich.

brian d foy schafft es mit vielen Code-Beispielen und Abbildungen, das Thema gut zu verdeutlichen, wobei manche Programm-Ausgaben sehr unübersichtlich wirken. Zu jedem Thema zeigt der Autor verschiedene Möglichkeiten auf, wie eine bestimmte Problematik angegangen werden kann. Dabei wird meistens auf Wege aus dem Perl-CORE und dann auf zusätzliche Module von CPAN eingegangen.

Im Anhang beschreibt der Autor noch seinen eigenen Weg bei der Suche nach einer Lösung von Problemen mit Perl-Programmen. Dies ist eine ganz nützliche „Checkliste“, sollte ein Problem zum richtigen Problem werden.

Das Buch ist zur Zeit nur in englischer Sprache erhältlich. Da die einzelnen Abschnitte relativ klein gehalten sind und brian d foy keine komplexen Satzstrukturen verwendet, ist das Buch auch für jeden mit durchschnittlichen Englischkenntnissen gut lesbar.

Renée Bäcker

brian d foy
O‘Reilly Media, 2007
0-596-52724-1

INTERVIEW

Richard Dice über “The Perl Foundation”

FM: Hallo Richard. Vielen Dank, dass Du Dir die Zeit für dieses Interview genommen hast. Möchtest Du uns etwas über Dich erzählen?

RD: Hallo Renée. Schön, wieder mit Dir zu reden (auch wenn wieder nur über das Internet).

Zuerst das Allgemeine über mich: Wie Du bereits erwähnt hast, heiße ich Richard Dice. Sicherlich sind viele deiner Leser Europäer. Deshalb sollte ich vielleicht erwähnen, dass mein Nachname nicht wie im italienischen „Dee-che“, sondern wie im Englischen „Dyse“ ausgesprochen wird. Ich bin Kanadier, geboren in Montreal und lebe seit 11 Jahren in Toronto. Davor lebte ich in verschiedenen Städten wie Ontario und Quebec. Anfang der 90er Jahre studierte ich Astronomie und Angewandte Mathematik. Kürzlich beendete ich mein Studium zum MBA an der Universität in Toronto. Ich lebe mit meiner Ehefrau und einer sehr außergewöhnlichen Katze in unserem Haus, ungefähr 10km von Toronto entfernt. Ich arbeite als IT Director bei Raybec Communications, einem Marketing & IT Startup-Unternehmen.

FM: Wie und wann kamst Du zu Perl?

RD: Ich programmiere seit 1994 in Perl. 1998 schrieb ich eine Reihe von Artikeln für Webmonkey, einem Geschäftsbereich von Wired/Hotwired. Diese Artikel über Web- und Datenbankprogrammierung mit Perl, Mysql, Apache und Linux wurden zu dieser Zeit sehr bekannt. Seit 1999 bin ich Mitglied der Montreal Perl Mongers, seit 2001 auch Mitglied der Toronto Perl Mongers (seitdem ich wieder in Toronto lebe, nachdem ich 2 Jahre in Montreal verbracht habe).

Ich würde sagen, das ist auch die Zeit, in der meine Perl Community Laufbahn begann. Während dieser Zeit arbeitete Damian Conway im Zuge eines Perl Development Grants der Perl Foundation Vollzeit an Perl, das dann neu strukturiert wurde. Ich bemerkte, dass eine Lücke in seinem Nordamerika Tourplan war und fragte ihn, ob er eine Rede in Toronto halten würde. Dies tat er.

Damian’s Reden in diesem Sommer hatten einen guten Erfolg. Wir planten, dass er im Sommer 2002 nach Toronto zurückkommen sollte. Mit einem Jahr Planungszeit wurde die Veranstaltung zu einem „Damian Festival“. Ungefähr 750 Leute kamen in einer Woche, um ihn zu sehen. Nach diesem Erfolg schlug er vor, dass ich zur YAPC::NA nach Toronto einladen sollte. Dies war schließlich im Sommer 2005 der Fall. Nicht lange Zeit danach fragten mich Allison Randal, die frühere Präsidentin der TPF, und der damals aktuelle TPF Präsident Bill Odom, ob ich Interesse daran hätte, der TPF beizutreten. Ich entschied mich dafür. Im Oktober 2005 wurde ich zum „Steering Committee Chairman“ gewählt.

FM: Du bist seit August 2007 der Präsident von The Perl Foundation (TPF). Wie hat diese neue Position Deine Sicht verändert?

RD: Aufgrund meiner großen Verantwortung in der TPF, habe ich es mir zu meinem persönlichen Ziel gemacht, mein Wissen über Perl zu vergrößern und meine Sicht zu erweitern. Ich versuche, möglichst viele Informationen über Perl zu sammeln, mit denen ich besser verstehen kann, wohin Perl in der immer größer werdenden IT-Welt passt. Meine These ist, dass ein besseres Verständnis uns helfen kann, uns darauf zu fokussieren, was am meisten Verbesserung(en) benötigt.

Was ich bis jetzt gelernt habe ist, dass die Programmiersprache Perl stark ist und die Perl-Community unglaublich dynamisch ist (hierbei sind die Eintrittszahlen in die Community sehr wichtig; setzen Arbeitskräfte Perl ein? lernen Studenten Perl und setzen es ein?)

FM: Wo gibt es Potential zur Verbesserung?

RD: Ich denke, dass in zwei großen Bereichen Verbesserungen notwendig sind.

Ein Bereich ist der technische. Die Perl-Umgebung besteht aus Erfahrungen von Leuten, die in Perl programmieren, aber in der Struktur ihrer größeren IT- (und Geschäfts-) Infrastruktur arbeiten. Zwei Probleme für diese Leute fallen mir hier auf: Eines ist die Verteilung von Anwendungen, die in Perl geschrieben sind. Dies können Anwendungen sein, die von außerhalb bezogen oder intern entwickelt wurden. Man sollte immer daran denken, dass die Verteilung nicht von den gleichen Leuten gemacht werden muss wie die Entwicklung. Wenn während der Verteilung etwas schief läuft, ist der eigentliche Entwickler nicht erreichbar um Fehler zu erkennen und zu beheben. Selbst wenn er erreichbar ist, ist es für ihn höchstwahrscheinlich eine Zeitverschwendung dies zu tun. So oder so sind Schwierigkeiten bei der Verteilung von Perl-basierten Systemen ein Hindernis für die Verwendung, den Einsatz und die Verbreitung von Perl.

Ein ähnliches Problem ist die Verteilung von Perl Modulen. CPAN funktioniert sehr gut für Leute, die (a) darauf Zugriff haben (dies grenzt viele interne Entwicklergemeinschaften in großen Firmen aus) und (b), die kein Problem damit haben, Modulkonflikte und/oder Abhängigkeiten an C-Bibliotheken oder anderen Formen von nicht-Perl Quellen zu lösen. Um diese Probleme zu umgehen, sollte es eine Option geben, die es ermöglicht, eine Art von erweiterten Kernmodulen zu installieren: z.B. so etwas wie Perl SDK, eine Zusammenfassung von getesteten „best of CPAN“. Strawberry Perl ist ein spannender Versuch, das Problem von C Modul Abhängigkeiten auf Windows Plattformen anzugehen. Ich habe gehört, dass es beginnt ein Set empfohlener Perl Module zu identifizieren. Es wird interessant sein zu beobachten, wie der Rest der Welt Strawberry Perl annimmt.

Es gibt ein ganz grundlegendes Problem, das in dieser Analyse immer wieder auftaucht. Dies ist, dass alles, was mit Perl passiert, ein Ergebnis von „Bottom-Up“ Community Bemühungen ist. Durch das gewachsene Verhalten von zehntausenden individuellen Programmierern, die ihre eigenen „local itches“ haben, entstand etwas wirklich Großartiges in Perl. Aber ich bin mir nicht sicher, ob dies das Verhalten ist, das wir brauchen, um die oben erwähnten Lücken schließen. Vielleicht ist es es nicht (offensichtlich ist es noch nicht so weit, ansonsten wären diese Probleme bereits adressiert worden). Vielleicht ist es es doch, aber nur, wenn einige ergänzenden magischen Bestandteile zu dem Mix hinzugefügt werden. Nachdem, wie wir in den vergangenen 20 Jahren gehandelt haben, müssten diese bestimmten Dinge beides sein: „some specific person‘s itch and an itch that that particular person has the capability to scratch.“ Wir könnten schon jetzt in der Lage sein, diese Probleme durch diesen Mechanismus zu lösen. Womöglich müssen wir aber einen neuen Mechanismus entwickeln. Möglicherweise können wir diese Probleme aber auch gar nicht lösen, damit riskieren wir aber Ausgrenzung und Veralterung.

Ein anderer Bereich, der aus meiner Sicht Verbesserung benötigt, ist Perl als Organisation. Wenn jemand außerhalb der Perlwelt „zu Perl sprechen will“, wen soll er kontaktieren? Wenn jemand eine Änderung benötigt, mit wem kann man zusammenarbeiten? Wer kann die Interessen der Perl Community in Bereichen vertreten, in denen diese angesprochen werden müssen? Wer kann den Wert von Perl denen erklären, die über die Möglichkeiten von Zukunftswachstum und –entwicklung Ihres IT-Betriebs nachdenken?

Zurzeit machen wir in diesem Bereich viel zu wenig, jedenfalls wenn man es daran misst, was gemacht werden könnte. Mir ist klar, dass aus meiner Sicht die Antwort „The Perl Foundation“ sein müsste. Allerdings sind wir eine freiwillige Organisation. Jeder, der hier mitarbeitet, macht das freiwillig. Ich denke, dass alle Leute, die in der TPF mitarbeiten, Leute mit großartigem Charakter, Intelligenz und hohem Können sind. Aber wir sind alle auch (gelinde gesagt) sehr stark mit unserer Arbeit und unserem Privatleben beschäftigt. Es kann sehr frustrierend sein zu wissen, was man alles für die Perl Community tun könnte, aber nicht genug Ressourcen hat, diese Dinge auch zu tun. Wir machen mit der Zeit, die wir zur Verfügung haben, das, was uns möglich ist.

FM: Ich denke viele unserer Leser wissen nicht viel über TPF. Was sind die Ziele von TPF und was macht sie?

RD: Ein Zitat von unserer Webseite http://www.perlfoundation.org/ -

„The Perl Foundation coordinates the efforts of numerous grass-roots Perl-based groups, including:
- International Yet Another Perl Conferences
- Carries the legal responsibility for Perl 5, Perl 6 and Parrot
- perl.org
- Perl Mongers
- PerlMonks“

Diese Dinge sind nicht unerheblich. Eine wichtige Pflicht für uns ist die internationalen Copyrights und Markenzeichen an Perl im Namen der Perl Community zu besitzen. TPF steht außerdem an der Spitze der Arbeit an der Artistic 2.0 Lizenz. Allison Randal gewann kürzlich ein 2007 White Camel für Ihre Arbeit an Artistic 2.0.

Die Wurzeln der TPF liegen in den YAPC Konferenzen in Carnegie Mellon University in Pittsburgh in 1999 und 2000. Nach der Konferenz im Jahr 2000 entschied sich das Organisationsteam, dass es nicht daran interessiert war, weiterhin die Konferenz an der CMU durchzuführen. Aber sie wollten, dass die YAPC weiterhin existiert. Deshalb wurde die TPF ausgegliedert. Der erste Job der neu entstandenen TPF war es, die treibende Kraft zu werden, um die künftigen YAPC (letztendlich YAPC:NA) Austragungsorte zu finden.

Eine weitere Bestrebung der TPF war die Unterstützung von Damian Conway 2001. Ähnliche Förderungen folgten im Jahr 2002 für Larry Wall und Dan Sugalski. Der moderne Nachfolger dieser Bemühungen, ist das Perl Development Grant Programm. Vor kurzem wurde ein neuer Zuschuss verkündet, der Patrick Michaud und seine Arbeit am Parrot Compiler Toolchain unterstützt.

Wenn man für die YAPC und an Perl Development Grants arbeitet, spricht es dafür, dass man Perl unterstützt. Wir werben auch außerhalb der Perl Community für Perl. Weitere große Projekte, die TPF im letzten Jahr übernommen hat, sind die Teilnahme an der Forrester Research Studie über „dynamic languages“ (z.B. Perl, Python, PHP, Ruby and ECMAscript) und die Zusammenarbeit mit The Linux Foundation, um Perl 5.8.8 in die LSB 3.2 Spezifikationen zu bringen. Außerdem bemüht sich die TPF, Vertreter von Perl auf IT-Konferenzen zu schicken.

FM: Was kann jeder einzelne von uns tun, um TPF zu unterstützen?

RD: TPF ist eine Organisation, die auf einer Mission beruht: Perl und die Perlkultur zu unterstützen. Wenn Du das machst, in welchem Bereich Du es kannst, machst Du die gute Arbeit, die wir als wichtig ansehen und bist ein Freund der TPF!

Ihr könnt Eure Ideen der TPF mitteilen. Meine Email-Adresse ist rdice@perlfoundation.org. Solltest Du eine Idee haben, von der wir wissen sollten, gib sie uns weiter. Solltest Du etwas bemerken, das bei Perl und der TPF anders gemacht werden sollte, bitte sag’ es uns. (Ein wirklich gutes Beispiel ist, das mir jemand geschrieben hat, dass Oracle 10.2.0 eine Perl-Distribution mitliefert! Dies zu wissen, macht es für mich viel einfacher Oracle näher zu kommen, um die TPF zu unterstützen.)

Ihr könnt Euch freiwillig melden, die treibende Kraft hinter einem TPF Projekt zu sein. Zum Beispiel haben wir im vergangenen Jahr eigenständige Hackathons gesponsert – bedeutend waren hier die Hackathons in Chicago und Amsterdam. Wir würden gerne mehr von diesen Veranstaltungen machen, aber wir brauchen mehr lokale Organisatoren. Es ist jede Menge Arbeit, aber diese Arbeit wird belohnt und ist sehr wertvoll durch das was Hackathons erreichen.

Ich wurde von Perl 5 Porters und der Perl 6 Workgroup angesprochen, weil sie Hilfe benötigen, um neue Mailing Listen Zusammenfasser zu finden. Dies ist eine sehr wichtige und sichtbare Arbeit. Falls jemand von Euch Interesse daran hat, bitte setzt Euch mit mir in Verbindung.

Ihr könnte uns gerne ansprechen, um im Namen von Perl zu sprechen. Wir müssen an mehr Konferenzen teilnehmen und wir müssen mehr Meetings mit Firmen machen, die Perl nutzen (oder nutzen könnten). Falls Ihr von Konferenzen wisst, an denen TPF teilnehmen sollte, kontaktiert mich und lasst es mich wissen. Bitte nehmt auch zur Kenntnis, dass wir auch finanzielle Unterstützung benötigen, um einen Delegierten auf eine Konferenz zu schicken. Nur durch finanzielle Unterstützung können wir Hackathons sponsern, YAPC Konferenzen und Perl Workshops unterstützen und sie sind das A und O einer Organisation, um sie am Leben zu halten (wie ein Steuerberater, der uns mit unseren Steuererklärung hilft).

NEUE REGEX-FEATURES IN PERL 5.10: Balanced Matching und Grammatiken

Perl 5.10 unterstützt rekursive Reguläre Ausdrücke. Das heißt mein Regulärer Ausdruck kann auf einen Teil von sich selbst verweisen, selbst wenn dieser Teil ein Teil des Ausdrucks ist, auf den verwiesen wird. Dies ermöglicht es, „symmetrische" Muster zu finden.
Zusammen mit ein paar weiteren neuen Features der Perl 5.10 RegEx kann ich auch eine komplette Grammatik innerhalb meines Regulären Ausdrucks definieren und Sachen machen, die weit über die traditionelle Definition von „Regulärer Ausdruck" hinaus geht.

„Symmetrische" Texte haben eine Startsequenz und eine Stopsequenz. Diese Muster zu finden ist einfach, wenn es eine Startsequenz und Stopsequenz gibt. Dieser String hat eine öffnende Klammer, ein eingeschlossener Text gefolgt von einer schließenden Klammer:

```
$ _ = "( simple to grab this )";
```

Ich erzeuge einen Treffer mit einer „nicht-gierigen" Suche, der den Text zwischen der öffnenden und der schließenden Klammer matcht.

```
my @matches = /
    \( \s*
    (.*?)      # interessanter Bereich
    \s* \)
/x;
```

Es ist sogar einfach mehrere Gruppen von „symmetrischen" Texten zu finden - so lange sie nicht verschachtelt sind

```
$ _ = "(simple to grab this) and (this too)";


my @matches = /
    \( \s*
    (.*?)      # interessanter Bereich
    \s* \)
/xg;
```

Was ich bisher nicht machen konnte war, einen „symmetrischen" Text zu matchen in dem eine Gruppe eine andere Gruppe enthält. Hier habe ich einen Satz Klammern, der einen weiteren Satz enthält:

```
$ _ = "(outer group (inner group) out again)";
```

Mein „nicht-gieriger" Ausdruck funktioniert hier nicht, da er nicht bei der ersten schließenden Klammer aufhört und ich bekomme schließlich Teile aus beiden Gruppen: sowohl der äußeren wie auch der inneren Gruppe:

```
outer group (inner group
```

Wenn ich mehr Zeit für diesen Regulären Ausdruck verwenden wollte, könnte ich erreichen, dass ich eine Ebene der Verschachtelung matchen könnte, aber was wäre mit zwei Ebenen? Ich könnte dafür ebenfalls einen Ausdruck machen. Aber für jede Ebene der Verschachtelung brauche ich einen unterschiedlichen Regulären Ausdruck. Ich müsste zu einer Lösung wechseln, die etwas wie `Text::Balanced` verwendet. Oder nicht?

Ich brauche nicht wirklich unterschiedliche RegEx für die einzelnen Levels der Verschachtelung. Ich benötige nur einen Weg für den RegEx um sich in sich selbst umzusehen. Ich kann rekursive RegEx in Perl 5.8 nutzen, aber ich muss einige Tricks anwenden um das zum Laufen zu bringen (Listing 1).

Ich muss diese Dinge sehr vorsichtig angehen, damit dies alles funktioniert. Als erstes muss ich `$pattern` vor dem Ganzen deklarieren, weil ich es innerhalb des Musters verwenden will und ich möchte der Variablen wieder etwas zuweisen. Zweitens - obwohl ich `$pattern` nicht direkt im Regulären Ausdruck evaluiere - evaluiere ich es indirekt wenn ich den Match-Operator verwende. Um dies zu erreichen verwende ich `(??{...})` für einen „verzögerten Subausdruck" was zur Laufzeit passiert. Es ist keine Interpolation; es wird dann

```
#!/usr/local/bin/perl5.8.8
use strict;

my $string   = "(x (x) y (x) x)";

my $pattern;

$pattern = qr[
    \(
        (?>
            (?>[^()]+)
                |
            (??{$pattern})
        )*
    \)
]x;

if( $string =~ /^($pattern)$/)
    {
    print "I matched!\n";
    }
```

Listing 1

```
#!/usr/local/bin/perl5.9.5
use strict;

my $string   = "(x (x) y (x) x)";

my $pattern = qr[
    ^
    (
    \(
        (?>
            (?>[^()]+)
                |
            (?1)
        )*
    \)
    )
    $
]x;

if( $string =~ /$pattern/)
    {
    print "I matched!\n";
    }
```

Listing 2

eingefügt wenn es gebraucht wird. Ich habe diese Warnung in `perlre` für Perl 5.8.8 gefunden (und sie ist auch noch in Perl 5.9.5):

```
"(??{ code })"
 WARNING: This extended regular expression
 feature is considered highly experimental,
 and may be changed or deleted without
 notice.  A simplified version of the syntax
 may be introduced for commonly used idioms.
```

Das ist nicht gut. Ich musste verschiedene Befehle und ein experimentelles Feature verwenden um es zum Laufen zu bringen. Jetzt funktioniert es, aber was ist mit der Wartung wenn der Programmierer beginnt Dinge hin und her zu schieben? Was passiert wenn das `(?{...})`-Feature verschwindet?

In Perl 5.10 habe ich Zugriff auf built-in Rekursions-Features. In meinem vorherigen Programm habe ich `(??{...})` verwendet um den Muster-String erneut zu evaluieren. Jetzt kann ich das Gleiche machen indem ich auf eine bestehende Gruppierung verweise. Ich schreibe das frühere Programm neu (Listing 2).

Ich muss `$pattern` nicht vorher deklarieren, weil ich es nicht in sich selbst verwenden werde. Da ich `$pattern` nicht als Subausdruck verwenden muss, kann ich in `$pattern` den gesamten Regulären Ausdruck, durch das Verschieben der „einfangenden“ Klammern und der Anfangs- und End-Anker in das Muster, halten - anstatt es im Match-Operator zu haben. Schließlich habe ich anstelle von `(?{$pattern})` ein `(?1)`, eines der neuen Features in Perl 5.10.

Die `(?n)`-Sequenz teilt der RegEx-Engine mit, dass der Subausdruck der n-ten Gruppierung verwendet werden soll (das ist der Grund, aus dem die „einfangenden“ Klammernn in das Muster geschrieben werden musste). Es stellte sich heraus, dass das viel schneller ist, da die RegEx-Engine nicht auf externe Variablen zugreifen muss und das komplette Muster auf einmal kennt.

Ich werde für einen Moment abschweifen, um auf zwei andere neue Features zu schauen, die ich in meiner Grammatik verwenden will. Ich kann mit `(?<NAME>)` eine Gruppierung benennen. Alles innerhalb von `(?...)` und den spitzen Klammern ist der Name zwischen spitzen Klammern. Ich habe dies in „Named Captures in Perl 5.9.5“ in The Perl Review 4.0 (Herbst 2007) beleuchtet. In Listing 3 ist ein kurzes Beispiel zu sehen.

Das gibt eine Nachricht aus, die den Text enthält, den ich mit dem benannten Treffer gefunden habe:

```
I matched foo!
```

Im Regulären Ausdruck habe ich die erste Gruppierung „first“ genannt und verweise später mit `\k<first>` darauf.

```
#!/usr/local/bin/perl5.9.5
use strict;

my $string   = "foo bar foo";

if( $string =~ /(?<first>\w+) \w+ \k<first>/)
    {
    print "I matched ${+{first}}!\n";
    }
```

Listing 3

```
#!/usr/local/bin/perl5.9.5
use strict;

my $string   = "foo bar quux";

if( $string =~ /(?<first>\w+) \w+ ((?&first))/)
    {
    print "I matched $1 and $2!\n";
    }
```

Listing 4

Nach dem Match, tauchen alles benannten Matches im neuen Spezial-Hash `%+` auf, der als Schlüssel den Namen des Treffers hat und als Wert den Text, der gefunden wurde. In diesem Beispiel hätte ich `\1` verwenden können um das Gleiche zu erreichen, aber später werde ich sehen, dass die nummerierten Rückreferenzen nicht geeignet sind, weil ich mir schlecht merken kann, welche Nummer zu welcher Gruppierung gehört.

Mit dem gleichen Feature kann ich weitere spezielle Dinge tun. I habe die `\k<first>`-Syntax für Rückreferenzen verwendet, so dass dieser Teil des RegEx exakt den gleichen Text matchen muss wie die Gruppierung mit dem Namen „first“. Wie auch immer, jetzt wo ich die Gruppierung benannt habe, kann ich auch dieses Muster wiederverwenden (anstelle des Textes, das das Muster gefunden hat). Ich verwende die `(?&name)`-Syntax um auf einen bennanten Unterausdruck zu verweisen (Listing 4).

Das gibt den ersten und den zweiten gespeicherten Text aus:

```
I matched foo and quux!
```

Bemerke, dass ich weiterhin die Nummernvariablen `$1` und `$2` verwenden kann auch wenn ich der „einfangenden“ Gruppe einen Namen gegeben habe. Der Text in `$2` war nicht der gleiche literale Text, weil das `(?&first)` die RegEx-Engine angewiesen hat, das *Muster* mit diesem Namen verwenden soll, nicht den gefundenen Text.

Bis jetzt habe ich rekursive Reguläre Ausdrücke, benannte Unterausdrücke und Referenzen auf Unterausdrücke. Ich benötige ein weiteres neues Feature um das alles zu einger Grammatik zusammenzufügen.

Ich kann Ausdrücke mit Bedingungen vewenden, in dem ich die Syntax `(?(CONDITION)yes-part|no-part)` benutze. Wenn der Teil der Syntax in `CONDITION` wahr ist, verwendet die RegEx-Engine den `yes-part` für den nächsten Teil des Pattern matchs, andernfalls den `no-part`. Diese Bedingung

```
#!/usr/local/bin/perl5.9.5
use strict;

my $string   = "foo bar quux";

if( $string =~ /
    ((?&word))
      \s+
    (?&word)
      \s+
    ((?&word))

    (?(DEFINE)  (?<word>\w+))
    /x
    )
    {
    print "I matched $1 and $2!\n";
    }
```

Listing 5

ist nicht beliebiger Perl-Code und es gibt verschiedene Dinge, wie ich auf andere Teile des RegEx verweisen kann. Du kannst das alles in `perlre` aus Perl 5.10.0 sehen, so dass ich Dich das selbst lesen lasse. Ich möchte eine bestimmte Bedingung verwenden: Wenn `CONDITION` der literale Wert ‚DEFINE‘ ist, soll die RegEx-Engin nur den `yes-part` betrachten (und ich gebe keinen `no-part` an). Weiterhin führt die RegEx-Engine den `yes-part` nicht direkt aus. Das heißt, dass es nicht direkt als Muster auf den String angewendet wird.

Ich schreibe mein letztes Programm neu und verwende dieses neue Feature. Ich füge das `(?(DEFINE)(?<word>\w+))` am Ende ein: Der Reguläre Ausdruck sieht die Definition bevor es mit dem Pattern Matching beginnt, da dass ich am Anfang darauf Zugriff habe. Innerhalb der Bedingung habe ich einen benannten Unterausdruck `(?<word>\w+)`. Ich kann diesen Unterausdruck überall im Muster verwenden, genauso wie ich es vorher getan habe. Für diesen Match-Operator habe ich den /x-Modifikator verwenden, so dass ich den RegEx besser lesbar gestalten kann (Listing 5).

Dieses Codestück erzeugt die gleiche Ausgabe wie das vorherige Programm:

```
I matched foo and quux!
```

Jetzt möchte ich all diese Features kombinieren. Als erstes wiederhole ich das „symmetrische Klammern“-Problem mit dem ich begonnen habe. Ich definiere eine kleine Grammatik am Ende meines Regulären Ausdrucks (und vielleicht bedeutet der Ausdruck „RegEx“ zu diesem Zeitpunkt auch wirklich nicht mehr); und ich rufe die Grammatik mit `((?&group))` auf. Bemerke, dass die Definition von `(?<group>...)` auf sich selbst verweist (siehe Listing 6).

```
#!/usr/local/bin/perl5.9.5
use strict;

my $string   = "(x (x) y (x) x)";

my $pattern;

$pattern = qr[
    ((?&group))

    (?(DEFINE)
        (?<other>    [^()]                                    )
        (?<open>     \( (?&other)*                            )
        (?<close>    (?&other)* \)                            )
        (?<group>    (?&open)((?&group)(?&other)*)*(?&close)  )
      )
]x;

if( $string =~ /$pattern/)
    {
    print "I matched $1!\n";
    }
```

Listing 6

Das ist alles. Ich habe mit einem Regulären Ausdruck begonnen und bin bei einer Grammatik gelandet. Das bedeutet, dass Fragen wie „Kann Perl eine RFC 2822 EMail-Adresse parsen?“ jetzt eine gute Antwort haben, da ich einfach die Grammatik definiere. Besser: Abigail definiert diese Grammatik für uns, wie er es in seinem Perl 5.10 RegEx Vortrag gemacht hat (siehe http://perl.abigail.be/Talks/RE5.10/HTML/email-1.html).

Brian d foy

Rätselhaftes Open

Neulich brachte ein Kollege die Frage auf, warum Perls open() im Pipe-Modus unterschiedliche Ergebnisse zurückliefert, je nachdem, ob ein Kommando seine STDERR-Ausgaben umleitet oder nicht.

In der Tat ist es verwunderlich, warum

```
open PIPE, "asdf |" or die "Fehler!";
```

mit einer Fehlermeldung abbricht und

```
open PIPE, "asdf 2>/dev/null |"
   or die "Fehler!";
```

so tut, als wäre alles in bester Ordnung. Nun ist dieser Test keine Garantie dafür, dass ein Kommando tatsächlich korrekt abgelaufen ist und seine Ausgabe über das File-Handle PIPE bereitsteht. Hierzu muss der sorgfältige Programmierer auch noch das Schließen der Pipe überprüfen:

```
   close PIPE or die "Fehler: $!";
```

Aber in beiden Fällen ist asdf kein gültiges Kommando (wenigstens auf meinem System) und open() sollte sofort einen Fehler zurückliefern. Was ist also der Unterschied zwischen den beiden? Warum verursacht die Umleitung von STDERR diese Anomalie?

Eine Nachfrage bei den Perlmönchen auf perlmonks.com brachte einige wertvolle Hinweise: Auch die Funktion system() reagiert unterschiedlich: Je nachdem, ob sie ein ihr übergebenes Kommando direkt über ein fork() mit anschließendem exec() im Prozesskind ausführen kann, oder wegen auftretenden Umleitungen die Shell des Systems bemühen muss.

Das Ganze wollte ich verifizieren, also warf ich den Debugger an. Nicht den Perl-Debugger, sondern gdb, denn ich wollte Einsicht in die internen Abläufe des Perl-Interpreters perl gewinnen. Meine Perlinstallation war mit dem Compilerflag -g übersetzt, also bereit für's Debuggen mit sichtbarem Sourcecode:

```
$ gdb /path/to/perl
(gdb) dir /path/to/perl/source
(gdb) b Perl_my_popen
(gdb) run test.pl
```

Das Kommando `dir` hilft gdb, die Perlsource zu finden, um sie zeilenweise beim Durchschreiten des Programms anzuzeigen. Durch Herumschnüffeln im Sourcecode hatte ich die Funktion Perl_my_popen() gefunden, die mir verdächtig danach aussah, für open() externe Kommandos anzuzapfen. Mit `run test.pl` startet die letzte Zeile das Testskript, das nur das oben gezeigte `open()`-Kommando mit Umleitung ausführt. Und siehe da, gdb stoppt sogleich an dem in Perl_my_popen gesetzten Breakpoint. Nach einigen Einzelschritten (Kommando `n` in gdb) wird die Zeile

```
  while ((pid = PerlProc_fork()) < 0) {
```

sichtbar. Ein fork() spaltet den Programmablauf ja bekanntlich in Prozessvater und Kind und der Debugger muss sich entscheiden, welchem Verlauf er folgen will. In gdb bestimmt

```
(gdb) set follow-fork-mode child
```

dass der Kindprozess der momentan wichtigere ist (`parent` wäre die Alternative). Einige Einzelschritte weiter kommt gdb an der Funktion `do_exec3` an, die verdächtige Untersuchungen mit der ihr übergebenen Kommandozeile anstellt:

```
strchr("$&*(){}[]'\";\\|?<>~`\n",*s)
```

Falls perl irgendeines dieser Zeichen im open() übergebenen Kommando findet, ruft es statt einem exec() folgendes auf:

```
PerlProc_execl(PL_sh_path, "sh", "-c",
  cmd, (char*)0);
```

Dies bemüht die Shell /bin/sh und übergibt ihr das Kommando zur Ausführung. Hätte perl hingegen keine Sonderzeichen gefunden, hätte es das Kommando an Leerzeichen getrennt und das auszuführende Programm samt Argumenten mit PerlProc_execvp() ausgeführt.

Geht alles klar, führen exec()-Funktionen das ihnen übergebene Kommando aus und kehren nie wieder zurück. Der Vaterprozess schnappt danach einfach die Ausgabe des Kindes auf. Wird das Kommando hingegen nicht gefunden, kehren sie zurück und es liegt ein Fehler vor, den perl an das Perl-Skript zurückmeldet.

Nun wird auch klar, warum ein nicht-existierendes Kommando ohne Umleitung einen Fehler liefert, eines mit Umleitung aber nicht: Im ersten Fall scheitert exec(), im zweiten Fall aber ruft es erfolgreich die Shell auf. Was später passiert, dass nämlich die Shell vor dem Kommando kapituliert, kommt erst später ans Licht.

Damit wäre der Fall aufgeklärt, allerdings scheiden sich die Geister, ob es sich um einen Bug oder ein Feature handelt ...

Mike Schilli

PERL-NACHRICHTEN.DE

Ich hatte der TPF mal vorgeschlagen, sogenannte „local representatives“ einzuführen. Damit könnte man einen besseren Kontakt zur „Basis“ bekommen und Neuigkeiten besser weiterverbreiten. Das würde der Wahrnehmung von Perl in der Öffentlichkeit ganz gut tun. Wir wissen, dass Perl noch lange nicht tot ist und eine tolle und mächtige Sprache ist. Leider ist das Bild von Perl in der Öffentlichkeit nicht ganz so gut.

TPF möchte meinen Vorschlag mal überdenken und auf die Frage, wie die „local representatives“ an Neuigkeiten etc. kommen würden und wie es weiterverbreitet wird, habe ich gesagt, dass es eine Plattform geben müsste, bei der jeder Neuigkeiten einreichen kann und wo sich jeder Interessierte über einen RSS-Feed über die neuesten Sachen informieren kann.

Die Plattform ist soweit fertig und ist unter http://www.perl-nachrichten.de erreichbar. Natürlich bin ich über jeden Eintrag - der mit Perl zu tun hat - dankbar. Man braucht kein Login oder so etwas. Es kann nur einige Minuten dauern, bis ich die Nachricht freigeschaltet habe.

Das ganze ist mit CGI::Application, HTML::Template::Compiled und weiteren Modulen aus dem CPAN umgesetzt.

Wenn die TPF auf meinen Vorschlag eingeht, wird die Plattform an eine Mailingliste angeschlossen, so dass Nachrichten automatisiert weitergereicht werden können. Und es sind noch einige andere Sachen daran zu verbessern...

Renée Bäcker

TPF-TICKER

Mozilla unterstützt Perl6-Entwicklung

Zusammen mit der Perl-Foundation hat die Mozilla Foundation einen Grant für die Perl6-Entwicklung genehmigt. Patrick Michaud - der Perl6-Compiler-Pumpking - bekommt für die nächsten vier Monate 15.000 US$ (10.000 US$ von Mozilla und 5.000 US$ von TPF).

Während dieser Zeit soll Perl6 auf Parrot laufen, die Testsuite soll erweitert werden, das Parrot Compiler Toolkit soll weitgehend fertiggestellt werden (inkl. Dokumentation) und noch weiteres.

Grant Manager wird Jesse Vincent, der selbst die Perl6-Entwicklung stark unterstützt.

Heise Zeitschriften Verlag

Der Heise Zeitschriften Verlag steht für qualitativ hochwertigen Journalismus. Wir verlegen mit c't und iX zwei auflagenstarke Computertitel, das Technologiemagazin Technology Review sowie das Online-Magazin Telepolis. Unsere Website heise online für Computer- und Internet-Interessierte zählt zu den meistbesuchten deutschen Special-Interest-Angeboten.

Zur Weiterentwicklung von heise online suchen wir zum nächstmöglichen Termin

Perl-Profis (m/w)

für die Entwicklung datenbankbasierter Web-Anwendungen.

Ihre Qualifikationen:

- mehrjährige Programmiererfahrung
- intensiver Kontakt mit DBI bzw. SQL sowie CPAN
- Erfahrungen mit einem Anwendungs-Framework (z. B. CGI::Application) und einem Template-System

Folgende Kenntnisse runden Ihr Profil ab:

- Ausgeprägte Erfahrung mit der Linux-Kommandozeile
- Sicherheitsaspekte von Web-Anwendungen (SQL-Injection, XSS)
- JavaScript-, CSS- und XHTML-Erfahrung
- Verarbeitung von XML und Unicode
- Objektorientiertes Programmieren
- Persistente Laufzeit-Umgebungen (FastCGI oder mod_perl)

Wir bieten Ihnen ein inspirierendes Arbeitsumfeld in einem erfolgreichen Team. Wenn es Sie reizt, am Erfolg von heise online mitzuwirken, freuen wir uns auf Ihre aussagekräftigen Bewerbungsunterlagen unter Angabe Ihrer Gehaltsvorstellung und des frühesten Eintrittstermins.

Für weitere Auskünfte steht Ihnen Herr Wolfgang Schemmel unter E-Mail ws@heise.de zur Verfügung.

Bewerbungen richten Sie bitte an:

Ein Unternehmen der Heise Medien Gruppe

Heise Zeitschriften Verlag GmbH & Co. KG
– Personalservice –
Aurelia Quignon

Helstorfer Straße 7
30625 Hannover
Telefon: 05 11/53 52-2 63
Aurelia.Quignon@heise-medien.de

Probleme mit base.pm

Das Modul `base.pm` wird in Perl Modulen verwendet, um die Vererbung zwischen Klassen festzulegen. Das Modul hat sich aber im Laufe der Zeit an seinem eigentlichen Ziel, dem deklarative Setzen von `@ISA`, vorbeientwickelt und hat dadurch die Fehlersuche in Programmen erschwert.

Was macht base.pm?

Die Vererbung von Methoden zwischen Klassen wird über die Package-Variable `@ISA` geregelt. Wenn die Klasse `Bar` Methoden der Klasse `Foo` erben soll, so wird das Perl über `@Bar::ISA` mitgeteilt:

```
@Bar::ISA = qw(Foo);
```

Dabei wird das Modul `Foo.pm` nicht automatisch von Perl geladen.

Gemäß seiner Dokumentation verbindet `base.pm` das Laden der Datei und die Modifikation von `@ISA`:

```
use base qw(Mama Papa);
```

ist im Wesentlichen äquivalent zu

```
BEGIN {
    require Mama;
    require Papa;
    push @ISA, qw(Mama Papa);
};
```

Seiteneffekte von base.pm

Über den dokumentierten Zweck hinaus hat `base.pm` einige Seiteneffekte, die unvermutet zu Fallstricken werden können.

base.pm setzt $VERSION, falls vorher keine definiert war.
Perl stellt für jedes Package die Subroutine `&VERSION` zur Verfügung, die normalerweise den Wert von `$VERSION` zurückliefert. Es ist aber auch möglich, eine eigene Implementation von `&VERSION` zu schreiben und `$VERSION` undefiniert zu lassen. Idealerweise sollten `$VERSION` und `&VERSION` immer den selben Wert zurückliefern.

```
package Blurgh;
# keine $VERSION
package Foo;
sub VERSION { 1 };
package Baz;
$VERSION = 2;

package Bar;
use base qw'Foo Baz Blurgh';

package main;
for $p (qw(Foo Baz Blurgh)) {
    printf "%s: VERSION(): %s\n\t\$VERSION: %s\n",
         $p, $p->VERSION,
         ${$p\::VERSION};
}
```

Listing 1

Das Programm 1 in Listing 1 zeigt, dass `base.pm` für jedes an `base.pm` genannte Modul die Variable `$VERSION` setzt, falls diese nicht schon gesetzt ist.

`$VERSION` in den einzelnen Modulen:

```
package Blurgh; # keine $VERSION
sub schlomp { 1 };
package Foo;
sub VERSION { 1 }; sub schlomp { 42 };
package Baz; $VERSION = 2;
```

Ausgabe Programm 1 (Listing 1), `&Foo::VERSION` stimmt nicht mit `$Foo::VERSION` überein. In der Package `Blurgh` wurde die Variable `$VERSION` von `base.pm` als Seiteneffekt erzeugt und gesetzt.

```
Foo: VERSION(): 1
     $VERSION: -1, set by base.pm
Baz: VERSION(): 2
     $VERSION: 2
Blurgh: VERSION(): -1, set by base.pm
     $VERSION: -1, set by base.pm
```

base.pm lädt nicht alle Module unter Perl 5.6

Unter Perl 5.6 lädt `base.pm` kein Modul, dessen `$VERSION` schon mal angefasst wurde. Das hat im allgemeinen keine Folgen, aber gerade im folgenden Fall einer Ausgabe zur Fehlersuche verschlimmert `base.pm` die Situation:

```
# Das Modul Foo ist nicht geladen
print "Debug: Foo VERSION: $Foo::VERSION";

require Bar;
Bar->schlomp;
# Cannot locate method „schlomp"
# via package ‚Bar' at ...
```

Unter Perl 5.6 legt die Ausgabe von `$Foo::VERSION` die Variable `$Foo::VERSION` an, wenn diese vorher nicht existierte. `base.pm` verwendet eine eigene Methode, die Existenz von `$Foo::VERSION` um zu schauem, ob das Modul `Foo` geladen wurde. Die übliche Methode, das Nachschlagen in `%INC`, wird nicht verwendet und daraus ergibt sich der Fehler, dass ein Modul nicht geladen wird, obwohl die Datei existiert und es noch nicht von Perl geladen wurde.

Auch wenn Perl 5.6 mit dem Erscheinen von Perl 5.10 von machen schon zum alten Eisen gezählt wird, ist 5.6.x immer noch eine weit verbreitete Version, zum Beispiel als das systemeigene Perl von Debian 3.x.

Der Fehler „Datei nicht gefunden" wird unterdrückt

Das schlimmste aller Vergehen von `base.pm` ist ein grundsätzlicher Verstoß gegen den Informationserhalt bei auftretenden Fehlern. `base.pm` unterdrückt Fehlermeldungen, wenn ein Modul nicht geladen werden kann.

Der Code

```
use base 'ExistiertNicht';
```

liefert die Fehlermeldung

```
Base class package „ExistiertNicht" is empty.
    (Perhaps you need to 'use' the module
        which defines that package first.)
```

Diese Meldung versteckt die eigentliche Fehlerursache, nämlich, dass keine Datei namens `ExistiertNicht.pm` in den Verzeichnissen in `@INC` gefunden wurde. Möglicherweise liegt ein Netzwerkfehler, ein Berechtigungsproblem oder eine fehlende Datei als Ursache vor, aber die Fehlermeldung von `base.pm` gibt uns keine Hilfestellung für die Fehlersuche. `base.pm` unterdrückt Ladefehler von Modulen einzig damit der folgende Code funktioniert:

```
use Tie::Hash;
use base 'Tie::StdHash';
```

Dies ist der einzige Code in der Perl Distribution, der diese Besonderheit benötigt - es gibt keine Datei `Tie/StdHash.pm` sondern `Tie::StdHash` lebt in `Tie/Hash.pm`.

Der Preis, nämlich der Verlust der hilfreichen Fehlerdiagnostik ist hoch für diese Besonderheit.

base.pm hat viel zu viel Code

`base.pm` hat mindestens 100 Zeilen Code, nur um einen Effekt wie der Code in Listing 2 zu erreichen. Zuerst werden die Module zu allen ererbten Klassen geladen, dann wird `@ISA` der aufrufenden Package gesetzt.

In Zeile 4 wird die erbende Klasse ermittelt. in Zeile 6 wird geprüft, ob das Laden des Elternmoduls versucht werden soll. Für den Fall, dass die Elternmodule geladen werden sollen, werden in Zeilen 9 bis 16 die Module geladen. In Zeile 10 wird geprüft, ob die erbende Klasse von sich selber erben

soll - das ist nicht sinnvoll, aber wir geben nur eine Warnung aus. Der Programmierer weiss hoffentlich, was er da tut. In den Zeilen 14 und 15 wird aus dem Klassennamen, zum Beispiel `Mein::Modul`, der Dateiname `Mein/Modul.pm` erzeugt und diese Datei dann geladen. Wenn ein Fehler beim Laden auftritt, bricht Perl mit einer informativen Fehlermeldung ab, die angibt, welche Datei warum nicht geladen werden konnte.

In den Zeilen 19 bis 22 wird dann die Variable `@ISA` in der erbenden Klasse auf die Elternklassen gesetzt.

parent.pm, ein Ersatz für base.pm

Die vielen Sonderteile von `base.pm` lassen sich leider nicht mehr zurücknehmen, ohne möglicherweise viel Code anzupassen, der `base.pm` verwendet und auch weiterhin unverändert laufen soll.

Um trotzdem bei der Fehlersuche nicht unterdrückten Informationen hinterherzulaufen, habe ich `parent.pm` geschrieben, ein Modul, das im Wesentlichen der Code in Listingg 2 ist. In allen meinen Modulen konnte ich die Verwendung von `base.pm` direkt ersetzen durch `parent.pm` ohne weitere Änderungen vornehmen zu müssen.

Ist die Verwendung von base.pm gefährlich?

Nein! Solange ein Programm läuft und man nicht nach Fehlern suchen muss, ist `base.pm` ungefährlich. Die Verwendung von `parent.pm` ist während der Entwicklung und bei der Fehlersuche informativer als `base.pm`. `parent.pm` verwendet Programmiertechniken, die bis zu Perl Version 5.004 funktionieren, aber ohne konkreten Anlass gibt es keinen Grund, bestehende Module von `base.pm` auf `parent.pm` umzustellen.

`parent.pm` ist über das CPAN verfügbar und läuft auf Perl Version ab 5.004.

Max Maischein

```
sub import {
    my $class = shift;

    my $inheritor = caller(0);

    if ( @_ and $_[0] eq '-norequire' ) {
        shift @_;
    } else {
        for ( my @filename = @_ ) {
            if ( $_ eq $inheritor ) {
                warn "Class '$inheritor' tried to inherit from itself\n";
            };

            s{::|'}{/}g;
            require "$_.pm"; # dies if the file is not found
        }
    }

    {
        no strict 'refs';
        # This is more efficient than push for the new MRO
        # at least until the new MRO is fixed
        @{"$inheritor\::ISA"} = (@{"$inheritor\::ISA"} , @_);
    };
};

"All your base are belong to us"
```

Listing 2

Perl 6 Tutorial - Teil 2 - Operatoren für Skalare

Fast keine Einleitung

Herzlich willkommen zur zweiten Folge dieses umfassenden Perl 6-Tutorials. Pugs und ein Editor der Wahl sind hoffentlich griffbereit und wir beginnen sofort. Wer noch weitere einleitende Worte möchte, findet diese im ersten Teil in der vorigen Ausgabe.

Perl ist eine operatorbasierende Programmiersprache

Unlängst konnte ich herzlich lachen, als ich diesen Satz in der p6l-Mailingliste sah. Der wahre Kern dieser Pointe besteht darin, dass Perl 6 tatsächlich auffallend viele Operatoren besitzt und das Operatoren einen bedeutenden Anteil an Perls Ausdrucksstärke haben. Und weil Operatoren bereits mit wenig Quellcode angewendet werden können, eignen sie sich vortrefflich als Einstieg in die Tiefen der Sprache.

Letztlich ist ihre Kenntnis auch eine wichtige Vorraussetzung für spätere, umfangreichere Beispiele.

Ein wenig Numerik zum Aufwärmen

Beginnen wir ganz einfach mit den geläufigsten Operatoren, den Grundrechenarten (‚+', ‚-', ‚*'), die jeder Insaße dieser Zivilisation vom Grundschullehrer nahe gebracht bekommt.

Perl 6:

```
say 3  + 4;    #  7
say 3  - 4;    #  -1
say 3  * 4;    #  12
```

Ebenso zeigte euch der freundliche Mathelehrer auch die Division, Modulo (Rest einer ganzzahligen Division) und Potenzierung, nur werden die in Computersprachen etwas anders geschrieben als im Unterricht, da man hier in den alten Tagen auf den ASCII-Zeichensatz beschränkt war.

Perl 6:

```
say 3  / 4;    #   0.75
say 3  % 4;    #   3     ( 3 div 4 = 0 Rest 3 )
say 3 ** 4;    #  81     ( = 3 * 3 * 3 * 3    )
```

Algebra unter erschwerten Bedingungen

So weit erfüllen die Ergebnisse die Erwartungen eines Mathematikers und zeigen, daß @larry noch weiß, welche Sachen man besser nicht ändert. Was Mathematiker jedoch nicht erwarten, aber sehr wohl Perl-Programmierer kennen, die mit Daten zweifelhafter Herkunft hantieren, sind Zahlenwerte, die Buchstaben enthalten können. Diese übergeht Perl großzügig, indem es alles ab dem ersten Zeichen ignoriert, dass nicht als Zahl interpretiert werden kann. Das wird Überführung eines Wertes in den numerischen Kontext genannt und geschieht immer dann, wenn Perl anhand der verwendeten Operatoren erkennt, dass hier mit Zahlen gerechnet wird. Folgende Beispiele geben gleiche Ergebnisse unter Perl 5 und 6 aus. Die Kommanotation von ‚print' hab ich deswegen gewählt, damit die Beispiele wirklich unter Perl 5 und 6 laufen, da Strings in beiden Sprachen unterschiedlich zusammengefügt werden.

Perl 5 & 6:

```
# 6.6, da 3 * 2.2
print 3 * '2.2j4',"\n";
#   0, da 3 * 0
print 3 * 'b2.2j4',"\n";
# -2.2, einfache Negierung
print   - '2.2ah',"\n";
```

Die Unterschiede beginnen, wenn man das ‚+' als Präfixoperator benutzt.

Perl 5 & 6:

```
   print   + '2.3ah',"\n";
```

Während Perl 5 dieses ‚+' ignoriert, überführt es in Perl 6 den folgenden String in den numerischen Kontext, wie das ‚-' in Perl 5 und 6, nur ohne zu negieren. Zwar gab es bereits in Perl 5 ‚int', welches ganze Zahlen zurückgibt und ‚abs' das positive Zahlenwerte liefert, aber eine einfache und direkte Umwandlung in den numerischen Kontext gab es erstaunlicherweise bisher nicht.

Perl 5 & 6:

```
print int '2.3ah';        # 2
print abs '-2.4ah';       # 2.4
```

Noch mehr Numerik zum Vertiefen

Perl 6:

```
say   + '2.3ah';          #   2.3
say   - '2.0ah';          # - 2
```

Diese Beispiele sind es wert, längere Zeit meditativ betrachtet zu werden. Denn sie beschreiben eine grundlegende Arbeitsweise vieler Operatoren. Perl 6 kennt nämlich nicht nur Skalar, Array und Hashkontext, sondern noch viele mehr, wie zum Beispiel den numerischen Kontext. Dessen Symbol ist quasi das ‚+', welches diesen nicht nur erzwingt sondern auch Sigil für eine Reihe von Operatoren ist, die ebenfalls diesen Kontext forcieren. Sie sind bereits als bitverändernde Ops bekannt, aber äußerlich kaum wiederzuerkennen. Die guten alten Shiftoperatoren werden jetzt z.B.‚+<' und ‚+>' geschrieben, wie das nächste Beispiel zeigt. Das sieht zuerst ungewohnt aus, ist aber logisch, weil meist nur der numerische Wert, den eine Zeichenkette ausdrückt, für Bitoperationen gebraucht wird. Außerdem sind ‚<<' und ‚>>' jetzt interpolierende Schwestern der ‚<' und ‚>'-Klammern geworden. So wie „" interpolierende Schwestern von ‚' sind. Analog zum Erklärten sind die bitweisen ‚und'(and), ‚oder'(or) und ‚entweder-oder'(xor) nun ‚+&', ‚+|' und ‚+^'. Die einfachen logischen Operatoren ‚&', ‚|' und ‚^' sind jetzt junktiv. Das werde ich genauer im Teil 3 erklären, wo es um alle Operatoren geht die entfernt mit Listen (Arrays) zu tun haben.

Perl 6:

```
say   5 +< 2;   # 20; 10100 = 101 << 2
say   5 +> 2;   # 1;     001 = 101 >> 2
say   5 +& 3;   # 1;     001 = 101 & 011
say   5 +| 3;   # 7;     111 = 101 | 011
say   5 +^ 3;   # 6;     110 = 101 ^ 011
```

Die meisten sonstigen numerischen Operatoren bleiben wie bekannt, lediglich das Autoincrement (++) und Autodecrement (--) sind nun keine reinen numerischen Op's mehr. Bevor ihr einen Herzschlag bekommt: klar verändern sie Zahlenwerte um 1. Aber sie erzwingen keinen Kontext, sondern passen sich ihm an. Denn sie sind allgemeine Operatoren geworden, die den Vorgänger oder Nachfolger eines Wertes rufen und bedienen sich daher des gleichen Mechnismus wie einige Vergleichsops. Ähnlich berechnen auch ‚div' und ‚mod' (auf Zahlen angewendet) Division und Modulo, können sich aber einem anderen Kontext auch anpassen.

Perl 6:

```
my  $a = 5;
say   ++$a;     # 6 (5 + 1)
say   $a++;     # 6 (erhöht wird danach)
say   --$a;     # 6 (6 + 1 - 1)
say   $a--;     # 6 (verringert wird danach)
say   $a --;    # Error
```

Routinierte Perlprogrammierer sollten nicht vergessen, daß jetzt zwischen Variable und ihrem Präfix- oder Suffixoperator kein Leerzeichen stehen darf. Die Gründe hab ich im vorigen Teil des Tutorials dargelegt.

Operatoren des Stringkontext

Analog zum numerischen gibt es, wie erwartet, auch einen Stringkontext, dessen Symbol ‚~' ist. Warum die Tilde? Nun, weil der ultimative Textoperator ‚=~' hieß, also Tilde schon immer etwas mit Strings zu tun hatte und der Punkt, wie letzes mal schon gezeigt, nun die Objektorientierung aus-

drückt. Die Tilde wandelt also in den Stringkontext um und kann auch Strings verknüpfen, wie der Unixbefehl ‚cat', den es jetzt auch innerhalb von Perl gibt.

Perl 6:

```
say            ~ '2.3ah';   # '2.3ah'
say            ~ '2.0';     # '2.0'
say            ~ 0xff;      #  255
say  'rot '    ~ 'blau'     # 'rot blau'
say  cat('rot ', 'blau');   # dito
say  'Farbe:' ~ color();    # eine Testausgabe
```

Die Tilde kann aber auch genauso mit den bitveränderndem Operatoren kombiniert werden, wie im numerischen Paralleluniversum das Plus. Auch wenn ich bis heute keinen praktischen Nutzen, für die daraus entstehenden Operatoren gefunden habe, will ich nicht ausschließen das es einen gibt. Die Tilde signalisiert, dass die Werte zeichenweise in Ordinalzahlen umgewandelt werden (mit ‚ord'). Auf die so-entstandenen Reihen numerischer Werten, können nun die bitweisen Rechenarten problemlos angewendet werden. Lediglich beim shiften muss der rechte Operand eine Zahl sein. Die Ergebnisse werden dann selbstverständlich vor der Ausgabe wieder mit ‚chr' in Textzeichen umgewandelt.

Perl 6:

```
'-' ~<  1 ;  #'Z'= chr  90 (    45 << 1      )
'z' ~>  1 ;  #'='= chr  61 (   122 >> 1      )
'a' ~& 'D';  #'@'= chr  64 (1100001 & 1000100)
'a' ~| 'D';  #'e'= chr 101 (1100001 | 1000100)
'a' ~^ 'D';  #'%'= chr  37 (1100001 ^ 1000100)
```

Perl 6 kennt aber erfreulicherweise auch die bekannten Befehle zur Bearbeitung von Zeichenketten wie ‚index', ‚substr', ‚uc', ‚lc' und so fort und sogar rubysche Spielereien wie reverse, welches die Reihenfolge der Zeichenkette umkehrt. Genau genommen sind das aber eingebaute Funktionen und keine Operatoren, um die es hier geht. Die bestehen aus wenigen, nicht alphanumerischen Zeichen. Obwohl, das stimmt eigentlich auch nicht, denn Perl 6 kennt ebenfalls noch das kleine ‚x', mit dem Perlschreiber schon früher Strings vervielfältigten.

Perl 6:

```
'a'  x 5;    # 'aaaaa'
'so' x 3;    # 'sososo'
```

Bei wirklich komplexen Fällen, führt jedoch kein Weg an den regulären Ausdrücken vorbei! Die nennen sich jetzt rules und bieten derart viele Möglichkeiten, dass sie hier mindestens einen eigenen Teil benötigen. Deswegen folgen jetzt nur wenige Beispiele um den in Perl 5.10 eingeführten Smartmatchoperator zu demonstrieren. Dieser ersetzt in Perl 6 das ‚=~' vollständig.

Perl 6:

```
"Ich hab getanzt ..."  ~~ m/tanz/;       # 1
"Ich hab getanzt ..." !~~ m/wanze/;      # 1
# Error
"Ich hab getanzt ..."  ~~ s/tan/schwit/;
```

Im Gegensatz zu Perl 5 liefert er aber nur einen boolschen Wert zurück, der den Erfolg der Suche meldet. (Auch beim Ersetzen.) ‚!~~' negiert am Ende der Suche lediglich diesen Ergebniswert. Jede weitere Information zum Suchvorgang können Programmierer wie gewohnt den Spezialvariablen entehmen, deren Syntax sich auch geändert hat. Die Fehlermeldung im Beispiel ist der Tatsache geschuldet, dass nur in Variablen, aber nicht in konstanten Ausdrücken ersetzt werden kann.

Vergleichsoperatoren

Die neue Arbeitsweise des ‚~~' erklärt sich einfach mit dem Umstand, dass es ein Vergleichoperator wie ‚==' ist. Der gibt ja auch nur wahr oder falsch zurück. (im perlschen Sinne) Ohne zu übertreiben kann man sogar sagen, das ‚~~' der Endgegner unter den Vergleichsoperatoren ist, noch weit mächtiger als bereits in Perl 5.10. Je nach Typ und Inhalt seiner Operanden versucht er in jeder Lage etwas Sinnvolles abzuliefern und hat dazu eine große Tabelle, die Vorrang und Kompatibilität aller internen Perlobjekte zueinander angibt. Das hat von Natur aus immer etwas mit Unschärfe zu tun und tatsächlich sieht ‚~~' ein wenig aus, wie das Symbol für das Wort circacirca.

Das genaue Gegenteil dazu ist der Operator ‚===', der die Identität, also Inhalt und Typ zweier Werte vergleicht. Wem das immer noch nicht genau genug ist, kann noch auf ‚eqv' zurückgreifen, welches zusätzliche Laufzeiteigenschaften beachtet.

Perl 6:

```
[1,2]  ===  [1,2];     # 0
'3.0'  ===   3;        # 0
```

Das erste Beispiel ergab negativ, weil es Referenzen (heißen jetzt Captures) auf 2 unterschiedliche Arrays sind. Das zweite ist ebenfalls negativ, weil für Perl offensichtlich ist, dass der linke Operand ein String ist und der rechte numerisch.

Gewöhnlich entgehen jedoch Perl-Programmierer solchen Problemen, in dem sie auch beim Vergleich den bevorzugten Kontext erzwingen. Wie in Perl 5 zeigen Ops aus mathematischen Symbolen (‚<‘,‘>‘,‘<=‘,‘>=‘,‘==‘) den numerischen Kontext an. Vergleichsops aus Buchstaben (‚lt‘,‘gt‘,‘le‘,‘ge‘,‘eq‘) vergleichen zeichenweise.

Perl 6:
```
[1,2] ==   [1,2];     # 1
'3.d' ==    3;        # 1
[1,2] eq   [1,2];     # 1
 3.0  eq    3;        # 1
```

Mit anderen Worten: ‚$a eq $b‘ ist eine Kurzschreibweise für ‚~$a === ~$b‘ und ‚$a == $b‘ kann auch ‚+$a === +$b‘ formuliert werden. Was aber tun, wenn nicht bekannt ist, ob man Zeichen oder Zahlen vergleichen wird? Dann nimmt man ‚~~‘, was aber nur gut gehen kann, wenn beide Vergleichswerte immer noch den gleichen Typ haben.

Perl 6:
```
'5' ~~  5.0;          # 1
'du' ~~  ,du';        # 1
3.0 ~~  '3f';         # 0
```

Die oberhalb aufgezählten Vergleichsops brauchen wohl kaum eine Vorstellung, auch nicht ihre negierte Form (mit einem ‚!‘ davor). Selbst die verkürzten Formen der ‚ungleich‘-Operatoren (numerisch ‚!=‘ für ‚!==‘) und (im string Kontext ‚ne‘ für ‚neq‘), haben bereits vor langer Zeit ihren Platz in der Perl-Folklore gefunden (jeweils beide Schreibweisen sind in Perl 6 erlaubt). Und doch steckt auch hier eine neue verallgemeinerte Logik hinter diesen Operatoren, denn es gibt weit mehr sortierbare Dinge als Zahlen und Buchstaben, die man praktischerweise mit einem Operator befragen kann, ob sie sich in der gewünschten Reihenfolge befinden. Die Ops dafür sind ‚before‘ und ‚after‘, die leider derzeit noch nicht von Pugs verstanden werden.

Perl 6:
```
say  3 before 5;          # 1
say 'b' after 'a';        # 1
```

Somit ist klar, dass ‚$a > $b‘ eine andere Schreibart für ‚+$a after +$b‘ ist und ‚$a lt $b‘ auch als ‚~$a before ~$b‘ ausgedrückt werden kann. All die bisherigen Vergleichsop haben jedoch gemeinsam, daß sie ein wahr oder falsch liefern. Nun, eigentlich ist es ein ‚Bool::True‘ oder ‚Bool::False‘, die im normalen, numerischen Kontext zu 1 oder 0 evaluiert werden, doch heben wir uns diese Spitzfindigkeiten für später auf, wenn das interne Objektsystem ansteht. Selbst wenn ich mehrere Vergleiche kombiniere, wie in:

Perl 6:
```
# wird in 1 Zug ausgewertet
say 3 < $b < 7;

# in 2 Zügen ausgewertet
say 3 < $b == $b < 7;
```

erhalte ich einen boolschen Wert, da nur zwei verschiedene Ergebnisse möglich sind. Im allgemeinen Vergleich sind es jedoch drei: kleiner als, gleich und größer als. Was sich in den Rückgabewerten, wie in P5, als -1, 0 und 1 ausdrückt. Aber vor der Umwandlung in den Ergebniskontext ist es ein ‚Order::Increase‘, ‚Order::Same‘ oder ‚Order::Decrease‘. Diese und weitere internen Methoden sind auch dafür sehr nützlich, dass ein Modul wie `DateTime` sie mit eigenen Routinen überladen kann. (So etwas wie „Tie“ gibt es hier nicht mehr.) Stellt dann der Befehl ‚cmp‘ fest, dass er hier zwei Objekte vom Typ `DateTime` vergleichen soll und die Parameterlisten der bereitgestellten Methoden auch zu den Operanden passen, werden sie verwendet. Und voila, ‚cmp‘ kann auch Zeitpunkte vergleichen. Ähnlich arbeiten auch ‚++‘ und ‚--‘, die ebenfalls dank mitgelieferter Methoden, Vorgänger- und Nachfolger eines Objektinhaltes ermitteln können. Jetzt hat mancher Perlkundiger sicher gerufen: „Aber ‚cmp‘ forciert doch den Stringkontext?“. Richtig, in Perl 5, aber nicht in Perl 6. Hier steht ‚cmp‘ wirklich für ‚compare‘ (vergleiche). Soll beim Vergleich auf numerischen Kontext geachtet werden, so benutze man das bekannte ‚<=>‘, im Strinkontext allerdings ‚leg‘. Das hört sich ausgesprochen ‚LowerEqualGreater‘ an und ist parallel zu <=>, an die Anfangsbuchstaben der Namen der verwandten Ops ‚lt‘, ‚eq‘, ‚gt‘ angelehnt.

Der dritte Skalarkontext

In diesem Text wurde es schon einige Male unterschwellig erwähnt, Perl 6 kennt auch einen boolschen Kontext. Und er wird mit dem ‚?‘ gefordert, da einfache Fragen auch meist mit ja oder nein beantwortbar sind. Wie der numerische, so kennt der boolsche auch einen verneinenden Operator: das Ausrufungszeichen. Eine Umwandlung in den boolschen Kontext tat man bisher eher implizit, z.B. beim Einsetzen einer Variable in eine if-Bedingung oder in Verbindung mit

logischen Operatoren. Jetzt geht es überall und explizit, auch mit ‚true' oder ‚not':

Perl 6:
```
say   ? '2.3ah';     #  1
say true 0      ;    #  0
say   ! '2.3ah';     #  ''
say  not      '';    #  1
```

Und sicher gibt es hier ebenfalls die kontexterzeugenden Bitoperatoren. Hier macht es auch ein wenig mehr Sinn als bei Texten. Nur nicht für die Shiftoperatoren, denn bei einem Bit Registerbreiter, gibt es nichts zu shiften.

Perl 6:
```
  say  0 ?& 'aha';  #  0
  say  3 ?| '2.3';  #  1
  say '' ?^  4    ; #  1
```

Steht das ‚?^' vor einem einzelnen Operanden, entspricht es einem ‚!' oder not.

Schlussendlich .. die Auswahloperatoren

Bei den logischen Operatoren gibt es noch eine wohlbekannte Gattung, die den Kontext der Werte nicht verändert, auch wenn sie intern mit den Umwandlungen der Werte in den boolschen Kontext rechnet. Ich nenne sie Auswahloperatoren, da sie den Wert eines Operanden zurückliefern. Manche nennen sie Kurzschlussoperatoren, da sie vorzeitig abbrechen, wenn das Ergebnis absehbar ist. Das kann man dazu nutzen, Befehle bedingt ausführen zu lassen. Ich halte meine Bezeichnung für treffender, da einer aus dieser Gattung (xor), diese Eigenschaft nicht besitzt. Ihre Syntax wird durch die Wiederholung des logischen Operatorzeichens gebildet.

Perl 6:
```
  say  0 && 'aha';  #   0
  say  4 and  0   ; #   0
  say  3 ||   0   ; #   3
  say  0 or   5   ; #   5
  say '' ^^   4   ; #   4
  say  7 xor  4   ; #   ''
```

Der mittlere Op hat eine recht neue Variante, die erst Perl 5.10 kennt. Sie nennt sich defined-or und liefert nur dann den rechten Wert, wenn der linke undefniert ist. Die drei Operatoren des letzten Listings haben noch eine Schreibweise, in der sie mit niedriger Priorität (in einem Ausdruck als Letzte) ausgeführt werden. (‚and' statt ‚&&', or statt ‚||' und xor statt ‚^^'). Diese wird gebraucht um Terme logisch zu verknüpfen, ohne allzuviel Klammern benutzen zu müssen, die die Lesbarkeit erschweren würden.

Wirklich neu ist in dem ganzen nur das ‚orelse'. Es funktioniert exakt wie ein ‚or', nur das die Spezialvariable ‚$!' des linken Blocks, auch im rechten Block weiterbesteht, falls dieser ausgeführt werden sollte. Dadurch wird dieser Befehl nützlich, um in knapper Form einen Aufruf oder kleinen Block mit einem zweiten zu verknüpfen, der im Falle eines Problems, die Fehlermeldung des Ersten auswerten kann.

Perl 6:
```
  do {...} orelse { say $! }
```

Der wiederum vertraute, ternäre Operator ist nicht direkt ein Auswahloperator, funktioniert aber ähnlich und wurde auch optisch den Auswahloperatoren angepasst. Das war anders kaum möglich, da Fragezeichen, wie bekannt, jetzt den boolschen Kontext erzwingen.

Perl 6:
```
  $a = defined $b ?? $b !! $c;
  $a = $b // $c;       # kürzer
```

Die kaum noch bekannten Flipflopops wurden umbenannt, um Verwechslungen mit den gleichnamigen Bereichsoperator zu vermeiden (‚..' zu ‚ff' und ‚...' zu ‚fff'). Dateitestoperatoren wurden den Perl 6-Konventionen angepasst und werden mit ‚:' statt ‚-' am Anfang geschrieben. Sie haben nun auch negierte Versionen (‚:!') und können gestapelt aber auch sonstig als Junctions logisch verknüpft werden.

Perl 6:
```
  if $datei ~~ :e     { say 'gibts' }
  if $datei.:!e       { say 'gabs'  }
  if $datei :r :w :x { ... }
  if $datei :r | :x  { ... }
```

Ich glaube das reicht als Indiz dafür, dass Perl 6 eine operatorbasierende Sprache ist. Mehr wäre auf einmal auch schwer aufzunehmen. Deswegen wird der nächste Teil die bisher nicht erwähnten behandeln, die vor allem Arrays betreffen.

Herbert Breunung

Winter of Code

Ein paar Monate nach der YAPC::EU 2007 hat Vienna.pm - der Ausrichter im letzten Jahr - nachgerechnet und festgestellt, dass von der YAPC noch viel Geld „übrig geblieben“ ist. Im Gegensatz zu den meisten anderen Perl Monger-Gruppen ist Vienna.pm ein eingetragener Verein, der auch einen entsprechenden Vereinszweck hat. Im Falle der Wiener ist es die Unterstützung von Perl; also musste eine Verwendung des Geldes gefunden werden, der zum Vereinszweck passt.

Als erstes werden die Veranstalter der YAPC::EU 2008 großzügig unterstützt. Die YAPC::EU 2008 findet vom 13. - 15. August 2008 in Kopenhagen statt und die Copenhagen.pm ist für die Anschubfinanzierung sicherlich dankbar. Aber es bleibt dennoch eine Menge Geld übrig und neben der Unterstützung der YEF (YAPC Europe Foundation) haben sich die Wiener dazu entschlossen, einen „Winter of Code“ durchzuführen.

Viele kennen sicherlich den „Google Summer of Code“, an den sich dieser „Winter of Code“ anlehnt. Nur ist hier die Teilnahme für alle offen und nicht nur Studenten vorbehalten. Insgesamt 20.000 € stehen für die Projekte zur Verfügung.

Die ersten Gelder sind schon vergeben:

David Landgren wird für ein Jahr bezahlt, eine wöchentliche Zusammenfassung der Perl5Porters-Mailingliste zu schreiben. Solche Zusammenfassungen gab es früher schon und ist irgendwann eingeschlafen, dank dem „Winter of Code“ kann jetzt wieder jeder Interessiert auf use.perl.org nachlesen, was die Kernentwickler von Perl machen.

Das zweite Projekt, das schon vergeben ist, ist eine Software zur Verwaltung der anderen Projekte - die erst nach Abschluss dieses Projektes anfangen. Hier hat sich Matt S. Trout von Shadowcat Systems Ltd gegen einige andere Bewerber durchgesetzt. Hier spielte die Tatsache, dass er das Geld wieder für andere Catalyst/DBIx::Class-Projekte zur Verfügung stellt und es eine Beispiel-Anwendung werden soll, eine große Rolle.

Alle weiteren Projekte werden über TODO-Tests beschrieben. Jeder Projektleiter eines Perl-Projekts kann dann über die von Matt geschriebene Software TODO-Tests für sein Projekt einreichen. Vienna.pm bewertet dieses Projekt und wenn es angenommen wird, kann sich jeder darum bewerben, diesen Test abzuarbeiten. Sowohl der Projektleiter als auch Vienna.pm werden sich die Bewerbungen anschauen und dann einen Bewerber aussuchen und einen Geldbetrag ausschreiben.

Jeder, der beim „Winter of Code“ mitmachen möchte, sollte das Wiki unter http://socialtext.useperl.at/woc/ im Auge behalten.

Diese Art der Verwendung von Geld finde ich sehr gut, da hier das Geld Perl und der Community zu Gute kommt. Einige erfolgreiche oder erfolgsversprechende Projekte können hier sowohl „Public Relations“ als auch Weiterentwicklungen erfahren. Die Perl-Foundation bietet zwar auch ein Förderprogramm (Grants) an, aber der „Winter of Code“ ist nicht wirklich eine Konkurrenz dazu, sondern vielmehr als ergänzendes Angebot zu sehen. Vienna.pm kann hier flexibler agieren als die TPF und es ist eine „einmalige“ Veranstaltung, wobei es natürlich wünschenswert wäre, wenn auch in Zukunft so etwas gemacht wird.

Renée Bäcker

Charsets oder „Warum funktionieren meine Umlaute nicht?“

Einführung

Jeder hat es schon mindestens einmal erlebt: Ein Programm, das mit Text arbeitet, funktioniert wunderbar, solange man keine Umlaute eingibt. Sonst kommt nur noch Zeichenmüll heraus und ein bis zwei nicht korrekt dargestellte Zeichen pro Umlaut.

ASCII

Um zu verstehen, warum es dazu kommt, muss man sich anschauen, wie „normaler“ Text und wie Umlaute binär abgespeichert werden.

Angefangen hat es 1963 mit ASCII, einem Standard, der 128 Zeichen je eine Zahl von 0 bis 127 zuweist, die mit 7 bit kodiert werden können.

Festgelegt sind die Zahlenwerte für lateinische Buchstaben, Ziffern, Satzzeichen und Kontrollzeichen wie „Carriage Return“ und „Line Feed“, also Zeilenumbrüche. Zeichen, die im Alltag eines Amerikaners nicht vorkommen, wie die deutschen Umlaute, kyrillische Zeichen und vieles mehr, wurden ausser Acht gelassen. Da ein Byte aus 8 Bits besteht, ist bei ASCII das erste, „most significant“ Bit immer 0.

Andere Zeichenkodierungen

Als man in Europa anfing Computer zu benutzen, mussten die benötigten Zeichen irgendwie im Computer gespeichert werden und dazu benutzte man die verbleibenden 128 Zeichen pro Byte. So entstanden die Kodierungen Latin-1 für den westeuropäischen Raum, Latin-2 für Mitteleuropa und so weiter, auch bekannt als ISO-8859-1 und ISO-8859-2.

Diese Zeichensätze stimmen in den ersten 128 Zeichen mit ASCII überein, die zweiten 128 Zeichen, also die mit 1 als erstem Bit, unterscheiden sich untereinander.

Die Grenzen dieser Zeichensätze werden einem schnell anhand eines gar nicht so alten Beispieles klar: Mit der Einführung des Euros hatten viele Länder eine neue Währung und damit ein Währungssymbol, das sich nicht in den traditionellen Zeichensätzen ausdrücken ließ! (Dieses Problem wurde durch das Einführen des Zeichensatzes ISO-8859-15 behoben, der sich nur wenig von Latin-1 unterscheidet und das €-Zeichen enthält).

Unicode

Die bisherigen Zeichenkodierungen konnten jeweils nur einen kleinen, lokal sinnvollen Bereich aller möglicher Zeichen darstellen - sobald man Texte mit gemischten Zeichensätzen verfassen wollte, ging das heillose Chaos los.

Um etwas Ordnung in das Chaos zu bekommen, hat das Unicode-Konsortium damit angefangen, jedem Zeichen, das in irgend einer Schrift in irgend einer Sprache vorkommt, eine eindeutige, ganze Zahl und einen Namen zuzuordnen.

Die Zahl heißt „Codepoint“ und wird üblicherweise als vier- oder sechsstellige, hexadezimale Zahl in der Form `U+0041` notiert; der dazugehörige Name wäre `LATIN SMALL LETTER A`.

Neben Buchstaben und anderen „Basiszeichen“ gibt es auch Akzentuierungen wie den z.B. `ACCENT, COMBINING ACUTE`, die auf den vorherigen Buchstaben einen Akzent setzen.

Wenn auf ein Basiszeichen eine Akzentuierung oder andere kombinierende Zeichen folgen, bilden mehrer Codepoints ein logischen Buchstaben, ein sogenanntes Grapheme.

Unicode Transformation Formats

Die bisher vorgestellten Unicode-Konzepte stehen vollständig unabhängig davon, wie die Unicode-Zeichen kodiert werden.

Dafür wurden die „Unicode Transformation Formats“ definiert, Zeichenkodierungen, die alle möglichen Unicode-Zeichen darstellen können. Der bekannteste Vertreter ist UTF-8, das für die bisher vergebenen Codepoints 1 bis 4 Bytes benötigt.

Auch in UTF-8 stimmen die ersten 128 Zeichen mit denen von ASCII überein.

Von UTF-8 gibt es auch eine laxe Variante, UTF8 (ohne Bindestrich geschrieben), die mehrere mögliche Kodierungen für ein Zeichen zulässt. Das Perl-Modul `Encode` unterscheidet diese Varianten.

UTF-16 dagegen benutzt für jedes Zeichen mindestens zwei Byte, für sehr hohe Unicode-Codepoints werden auch hier mehr Bytes benötigt.

UTF-32 kodiert jedes mögliche Zeichen mit vier Bytes.

Beispiele für Zeichenkodierungen (siehe Tabelle 1).

Perl und Zeichenkodierungen

Strings können in Perl entweder als Bytestrings oder als Textstrings vorliegen. Wenn man z.B. eine Zeile aus `STDIN` liest, ist sie per Default ein Bytestring.

Mit der Funktion `decode` des Moduls `Encode` kann man sie in einen Textstring umwandeln, wenn man die Zeichenkodierung kennt. In Gegenrichtung, also von Textstring nach Bytestring konvertiert man mit `encode` aus dem gleichen Modul.

Alle Textoperationen sollte man auf Textstrings durchführen, weil so auch Nicht-ASCII-Zeichen korrekt behandelt werden: `lc` und `uc` funktionieren wie erwartet, und `\w` in regulären Ausdrücken passt auf jeden Buchstaben, auch auf Umlaute, ß und allen möglichen Zeichen in allen möglichen Sprachen, die dort als Bestandteil eines Wortes angesehen werden.

`cmp` vergleicht Nicht-ASCII-Zeichen allerdings nur dann so, wie man das erwartet (also `"ä" lt „b"`), wenn `use locale` aktiv ist. Da das Verhalten von `sort` durch `cmp` definiert ist, gilt dies auch für das Sortieren von Listen (siehe Listing 1).

```
#!/usr/bin/perl
use warnings;
use strict;
use Encode qw(encode decode);

my $enc = 'utf-8';
# in dieser Kodierung ist das
# Script gespeicher
my $byte_str = "Ä\n";

# Bytestrings:
print lc $byte_str;
# gibt 'Ä' aus, lc hat nichts verändert

# Textstrings:
my $text_str = decode($enc, $byte_str);
   $text_str = lc $text_str;
# gibt 'ä' aus, lc hat gewirkt
   print encode($enc, $text_str);
```

Listing 1

Es empfiehlt sich, alle Eingaben direkt in Textstrings umzuwandeln, dann mit den Textstrings zu arbeiten, und sie erst bei der Ausgabe (oder beim Speichern) wieder in Bytestrings umzuwandeln. Wenn man sich nicht an diese Regel hält, verliert man im Programm schnell den Überblick welcher String ein Textstring und welcher ein Bytestring ist.

Perl bietet mit den IO-Layern Mechanismen, mit denen man die Kodierungen per Dateihandle oder global automatisch umwandeln lassen kann (siehe Listing 2).

Codepoint	Zeichen	ASCII	UTF-8	Latin-1	ISO-8859-15	UTF-16
U+0041	A	0x41	0x41	0x41	0x41	0x00 0x41
U+00c4	Ä	-	0xc4 0x84	0xc4	0xc4	0x00 0xc4
U+20AC	€	-	0xe3 0x82 0xac	-	0xa4	0x20 0xac
U+c218	수	-	0xec 0x88 0x98	-	-	0xc2 0x18

Tabelle 1: Beispiele Zeichenkodierung

```
# IO-Layer: $handle liefert beim Lesen jetzt Textstrings:
open my $handle, '<:encoding(UTF-8)', $datei;

# das gleiche:
open my $handle, '<', $datei;
binmode $handle, ':encoding(UTF-8)';

# Jedes open() soll automatich :encoding(iso-8859-1) benutzen:
use open ':encoding(iso-8859-1)';

# Alle Stringkonstanten werden als utf-8 interpretiert
# und in Textstrings umgewandelt:
use utf8;

# Schreibe Text mit der aktuellen locale nach STDOUT:
use PerlIO::locale;
binmode STDOUT, ':locale';
# alle Lese-/Schreibeoperation mit aktueller locale:
use open ':locale';
```

Listing 2

```
#!/usr/bin/perl
use warnings;
use strict;
use Encode;

my @charsets = qw(utf-8 latin1 iso-8859-15 utf-16);

my $test = "Ue: " . chr(220) . "; Euro: " . chr(8364) . "\n";

for (@charsets){
       print "$_ : " . encode($_, $test);
}
```

Listing 3

```
use Devel::Peek;
use Encode;
my $str = "ä";
Dump $str;
$str = decode("utf-8", $str);
Dump $str;
Dump encode('latin1', $str);
__END__
SV = PV(0x814fb00) at 0x814f678
      REFCNT = 1
      FLAGS = (PADBUSY,PADMY,POK,pPOK)
      PV = 0x81654f8 "\303\244"\0
      CUR = 2
      LEN = 4
SV = PV(0x814fb00) at 0x814f678
      REFCNT = 1
      FLAGS = (PADBUSY,PADMY,POK,pPOK,UTF8)
      PV = 0x817fcf8 "\303\244"\0 [UTF8 "\x{e4}"]
      CUR = 2
      LEN = 4
SV = PV(0x814fb00) at 0x81b7f94
      REFCNT = 1
      FLAGS = (TEMP,POK,pPOK)
      PV = 0x8203868 "\344"\0
      CUR = 1
      LEN = 4
```

Listing 4

Mit Vorsicht sollte man den Input Layer `:utf8` genießen, der annimmt, dass die Eingabedatei gültiges UTF-8 ist. Sollte sie das nicht sein, ist das eine potentielle Quelle für Sicherheitslücken (siehe `http://www.perlmonks.org/?node_id=644786` für Details).

Als Output Layer dagegen ist `:utf8` anstelle von `:encoding(UTF-8)` problemlos verwendebar.

Das Modul und Pragma `utf8` erlaubt es auch, Nicht-ASCII-Zeichen in Variablennamen zu verwenden. Da das jedoch kaum getestet ist und zum Teil mit Modulenamen und Namespaces nicht gut funktioniert, sollte man Variablennamen weiterhin nur aus lateinischen Buchstaben, dem Unterstrich _ und Ziffern zusammensetzen.

Die Arbeitsumgebung testen

Ausgestattet mit diesem Wissen kann man testen, ob das Terminal und locales auf die gleiche Kodierung eingestellt sind, und auf welche (siehe Listing 3)

Wenn man dieses Programm in einem Terminal ausführt, wird nur eine Textzeile korrekt angezeigt werden, die erste Spalte darin ist dann die Zeichenkodierung des Terminals.

Wie vorher gesagt ist das Eurozeichen € nicht in Latin-1 vorhanden, das `Ü` sollte in einem Latin-1-Terminal trotzdem richtig angezeigt werden.

Troubleshooting

Die Art, wie perl Textstrings intern speichert, führt zu etwas ungewöhnlichem Verhalten wenn diese ausgegeben werden. Wenn alle Codepoints in dem String kleiner als 256 sind, wird der String intern als Latin-1 gespeichert, und als solcher ausgegeben.

Sind Codepoints größer gleich 256 in einem String vorhanden, speichert perl den String als UTF-8, und wenn man sie mit `print` ausgibt (und wenn `warnings` aktiv sind), kommt die Warnung `Wide character in print`.

Daher empfiehlt es sich, eigene Scripte mit Zeichen zu testen, die nicht in Latin-1 vorhanden sind, z.B. das Euro-Zeichen €. Dann sieht man immer, wenn man ein `Encode::encode` an enstprechender Stelle vergessen hat, die oben genannte Warnung.

Wenn Daten aus externen Module kommen, z.B. aus `DBI`, kann man mit `utf8::is_utf8(STRING)` überprüfen, ob ein String als Textstring abgespeichert ist. Diese Abfrage liefert aber nur ein sinnvolles Ergebnis, wenn Zeichen außerhalb des ASCII-Zeichensatzes in dem String enthalten sind.

Auch `Devel::Peek` zeigt an, ob ein String intern als Textstring oder als Bytestring vorliegt (siehe Listing 4).

Der String `UTF8` in der Zeile `FLAGS =` zeigt, dass der String als Texttring vorliegt. In der Zeile `PV =` sieht man bei Textstrings die Bytes und in Klammer eckigen Klammern die Codepoints.

Weitere Probleme können durch fehlerhafte Module entstehen. So ist die Funktionalität des Pragmas `encode` sehr verlockend:

```
# automatische Konvertierungen:
use encoding ':locale';
```

Allerdings funktionieren unter dem Einfluss von `use encoding` AUTOLOAD-Funktionen nicht mehr, und das Modul funktioniert nicht im Zusammenspiel mit Threads.

Kodierungen im WWW

Beim Schreiben von CGI-Scripten muss man sich überlegen in welcher Kodierung die Daten ausgegeben werden sollen und das enstprechend im HTTP-Header vermerken.

Für die meisten Anwendungen empfiehlt sich UTF-8, da man damit einerseits beliebge Unicode-Zeichen kodieren kann, andererseits auch deutschen Text platzsparend darstellen kann.

HTTP bietet zwar mit dem `Accept-Charset`-Header eine Möglichkeit herauszufinden, ob ein Browser mit einer Zeichenkodierung etwas anfangen kann, aber wenn man sich an die gängigen Kodierungen hält, ist es in der Praxis nicht nötig, diesen Header zu prüfen.

Für HTML-Dateien sieht ein Header typischerweise so aus: `Content-Type: text/html; charset=UTF-8`. Wenn man einen solchen Header sendet, muss man im HTML-Code nur die Zeichen escapen, die in HTML eine Sonderbedeutung haben (`<`, `>`, `&` und innerhalb von Attributen auch `"`).

Beim Einlesen von POST- oder GET-Parametern mit dem Modul `CGI` muss man darauf achten, welche Version man benutzt: In älteren Versionen liefert die `param`-Methode immer Bytestrings zurück, in neueren Versionen (ab 3.29) werden Textstrings zurückgegeben, wenn vorher mit `charset` die Zeichenkodierung UTF-8 eingestellt wurde - andere Kodierungen werden von `CGI` nicht unterstützt.

Damit Formularinhalte vom Browser mit bekanntem Zeichensatz abgeschickt werden, gibt man im Formular die `accept-charset`-Entity mit an:

```
<form method="post" accept-charset="utf-8"
      action="/script.pl">
```

Bei Verwendung eines Template-Systems sollte man darauf achten, dass es mit charsets umgehen kann. Beispiel sind `Template::Alloy`, `HTML::Template::Compiled` (seit Version 0.90) oder Template Toolkit in Verbindung mit `Template::Provider::Encoding`.

Weiterführende Themen

Mit den Grundlagen zu den Themen Zeichenkodierungen und Perl kommt man schon sehr weit, zum Beispiel kann man Webanwendunen „Unicode-Safe“ machen, also dafür sorgen, dass alle möglichen Zeichen vom Benutzer eingegeben und dargestellt werden können.

Damit ist aber noch längst nicht alles auf diesem Gebiet gesagt. Der Unicode-Standard erlaubt es beispielsweise, bestimmte Zeichen auf verschiedene Arten zu kodieren. Um Strings korrekt miteinander zu vergleichen, muss man sie vorher „normalisieren“. Mehr dazu gibt es in der Normalisierungs-FAQ: `http://unicode.org/faq/normalization.html`.

Um landesspezifisches Verhalten für Programme zu implementieren, lohnt es, die locales genauer anzusehen, ein guter Einstiegspunkt ist das Dokument `perllocale`.

Moritz Lenz

TPF-TICKER

Perl 5.10 veröffentlicht

Pünktlich zum 20. Geburtstag von Perl (18. Dezember 2007) hat Pumpking Rafael Garcia-Suarez die neue Perl-Version auf CPAN veröffentlicht. Mit Perl 5.10 gibt es nach über 5 Jahren wieder ein „großes“ Update von Perl. Dabei wurden viele neue Features implementiert, die zum Teil aus der Perl6-Entwicklung stammen. Auch an der Engine für die Regulären Ausdrücke wurde komplett neu gestaltet, so dass jetzt auch ziemlich einfach Grammatiken definiert werden können.

Chris Dolan beendet Perl::Critic-Grant

Im Laufe der Zeit wurden 20 neue Policies für Perl::Critic umgesetzt. Dabei hat sich Chris Dolan an „Perl Best Practices“ orientiert. Während der Entwicklung wurden einige Bugs in PPI gefixt und neue Features eingeführt. Der Grant lief seit April 2007 und hatte einen Wert von 2000 US$.

Während der Zeit hat Chris Dolan Perl::Critic an unterschiedlichen Stellen promoted. Unter anderem war er Sprecher auf der WebGUI Users Conference.

Nützliche Variablen: $/

In Newsgroups und Foren tauchen immer wieder ähnliche Fragen auf, bei denen es sich die Schreiber unnötig schwer machen. Da sollen Dateien eingelesen werden, die verschiedene „Datenblöcke“ beinhalten. Unter „Datenblöcke“ sind hier mehrzeilige zusammengehörige Informationen zu verstehen, die sehr häufig in der Bio-Informatik bei Dateien im sogenannten FASTA-Format vorkommen.

Bei solchen „Datenblöcken“ bestehen die zusammengehörigen Informationen nicht nur aus einer einzigen Zeile, so dass bei einem zeilenweise Einlesen der Datei nicht immer klar ist, in welchem Datenblock sich das Programm gerade befindet.

FASTA-Dateien sind Dateien in einem bestimmten Format, in denen Sequenzdaten gespeichert sind - also mit Bioinformatik zu tun haben. Perl ist in der Bioinformatik eine weit verbreitete Programmiersprache und wenn man sich der Mächtigkeit von Perl bedient, versteht man auch warum...

Diese FASTA-Dateien werden häufig eingelesen. Dabei möchte man meistens komplette Blöcke in ein Array schreiben, anstatt wie üblicherweise die Datei zeilenweise einzulesen. (domm)

Eine FASTA-Beispieldatei (siehe Listing 1).

```
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIAGX
IENY
>gi|1234567|gb|AAD44133.8| cytochrome a [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAHTHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLLALLSPDMLGDPDNHMPGLMITADSHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIAGX
IENY
```

Listing 1

In vielen Programmen sieht man dann so etwas, wie in Listing 2 dargestellt.

Das ist aber reichlich umständlich. Es gibt bereits Module wie etwa `BioPerl` oder `Bio::FASTASequence`, die einfache Interfaces zum Lesen dieser Dateien anbieten. Wenn man das Parsen aber lieber selbst machen will, sollte man auch die Möglichkeiten von (idiomatischem) Perl nützen. (domm)

Es gibt eine Spezialvariable`$/` (siehe auch perldoc perlvar), die das „Ende des einzulesenden Blocks“ festlegt. Im Normalfall - wenn man also nichts verändert hat - ist die Variable auf „\n“ gesetzt. Damit wird festgelegt, dass immer bis zu einem „\n“ eingelesen wird.

Im Falle der FASTA-Dateien ist das zeilenweise Einlesen aber eher ungeschickt. Wenn man sich die FASTA-Datei anschaut, sieht man, dass jeder Eintrag mit „>“ an einem Zeilenanfang beginnt.

Also kann man `$/` auf „\n>“ setzen. Damit ist *ein* einzulesender Block durch ein Zeilenende-Zeichen und einem folgenden „>“ begrenzt.

Somit ergibt sich die Möglichkeit, wie in Listing 3 dargestellt.

```
#!/usr/bin/perl

my $fasta_file = ,/pfad/zur/datei.fasta‘;
my @sequences;
my $sequence;
my $counter = 0;

open(FILEHANDLE, „<$fasta_file“) or die $!;
while( <FILEHANDLE> ){
    if( $_ =~ /^>/ and $counter > 1 ){
        push(@sequences, $sequence);
        $sequence = $_;
    }
    $sequence .= $_;
    $counter++;
}
push @sequences, $sequence if $sequence;
close(FILEHANDLE);

foreach(@sequences){
    print „Sequenz: $_\n\n“;
}
```

Listing 2

Noch deutlicher wird es bei Dateien, bei denen die interessanten Blöcke durch eine Leerzeile begrenzt sind (Listing 4).

```
Header1
Info1 zu Header1
Zeile2 der Info1

Header2
Info1 zu Header2
Zeile2 der Info1
```

Listing 4

Hier kann man `$/` auf „\n\n“ setzen und schon bekommt man bei jedem Durchlauf der `while`-Schleife beim Einlesen einen kompletten interessanten Block geliefert.

Eine oft anzutreffende Aufgabe ist, eine Datei komplett in einen Skalar einzulesen. Bei Einsteigern findet man dann häufig ein Konstrukt wie in Listing 5.

```
my $content = "";
open( FH, "<datei.txt") or die $!;
while( <FH> ){
    $content .= $_;
}
close FH;
```

Listing 5

Dabei reicht es schon `$/` auf `undef` zu setzen. Damit kann man dann die ganze Datei auf einmal einlesen (Listing 6 und Listing 7).

```
my $content;
{
    local $/;
    open( FH, "<datei.txt") or die $!;
    $content = <FH>;
    close FH;
}
```

Listing 6

```
#!/usr/bin/perl

my $fasta_file = '/pfad/zur/datei.fasta';
my @sequences;

{
    local $/ = "\n>";
    open my $fh, '<', $fasta_file or die $!;
    while(my $sequence = <$fh> ){
        chomp $sequence;
        $sequence = '>' . $sequence unless
$sequence =~ /^>/;
        push @sequences, $sequence;
    }
    close $fh;
}

foreach(@sequences){
    print "Sequenz: $_\n\n";
}
```

Listing 3

```
my $content = do{
local (@ARGV, $/) = "datei.txt"; <> };
```

Listing 7

Aber man muss aufpassen: Die Änderung von `$/` sollte man auf einen möglichst kleinen Block beschränken. Deswegen habe ich oben auch das Einlesen in die geschweiften Klammern gepackt. Wenn man *kein* local verwendet und innerhalb eines Programms erst eine FASTA-Datei dann eine andere Datei einlesen will, gibt das interessante Probleme...

`chomp` arbeitet übrigens auch mit `$/`, deswegen macht es in meinem Beispielcode auch das Richtige und entfernt das „\n>“ bei jedem Eintrag.

Soll eine Datei byteweise eingelesen werden, dann kann man das auch mit einer Veränderung von `$/` erreichen: Durch eine Zuweisung einer Referenz auf einen Integer-Wert wird angegeben, wieviele Bytes jedesmal eingelesen werden sollen. Mit `$/ = \2` wird Perl mitgeteilt, dass immer 2 Bytes eingelesen werden sollen (Listing 8).

Die Ausgabe ist dann

```
gelesen: Te
gelesen: st
```

Listing 8

Allerdings muss dabei beachtet werden, dass zum Beispiel bei UTF-8 1 Zeichen 2 Bytes haben kann.

Renée Bäcker

Date::Calc

Wenn morgen, gestern schon heute war ...

Wann ist eigentlich Freitag?

Vor einigen Wochen sprach mich ein Bekannter an, der an seiner Schule mit einem Linux-basierten System arbeitet, ob es denn möglich wäre Voll-Backups, als `cron`-job, nur am letzten Freitag des Monats auszuführen. cron selbst ist nicht dazu in der Lage und die Jobs per at-daemon einzurichten geht wahrscheinlich zu häufig vergessen, als das es eine gute Lösung wäre.

Die Idee ein Skript einfach jeden Freitag zu starten, um dann in dem Skript festzustellen ob gerade dieser Freitag der letzte Freitag des Monats ist, erschien uns als legitimer Lösungsansatz um weiterhin `cron` verwenden zu können. Jetzt sind wir schon näher an dem eigentlichen Problem: Der letzte Freitag eines Monats ist frühestens der 24., spätestens der 31., außer im Februar - und bei diesem „Problem“-Monat kommen auch noch die Schaltjahre dazu.

```
   December 2007
 S M Tu  W Th  F  S
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

Listing 1

Die Problemstellung sollte per klassischem Shell-Scipting zu lösen sein, auf Perl wollte ich zu diesem Zeitpunkt noch verzichten. Es kommen also die üblichen Verdächtigen ins Spiel, wie `date`, `cal`, `grep`, `tail`, `awk` oder `cut`. `cal` gibt einen Kalender des aktuellen Monats aus (siehe Listing 1).

Die Überlegungen führte zu folgendem Lösungsansatz: Mit `grep` filtert man die Zeilen an deren Ende eine zweistellige Zahl steht. `cut` isoliert die Freitags-Spalte der Kalenders, `tail` erlaubt uns die letzte Zeile zu ermitteln und damit auch das Datum des letzten Freitags im Monat. Zum Vergleich wird das aktuelle Datum mit `date` herangezogen, wobei man durch die Formatierungsoptionen von `date` erreicht, dass die Ausgabe einzig dem aktuellen Tage entspricht.

Als Shell-Skript ausformuliert, sieht das Ganze dann aus, wie in Listing 2 dargestellt.

Zu meiner Schande muss ich eingestehen, dass ich wahrscheinlich einer der miserabelsten Shell-Programmierer bin, der rumläuft. Aus meiner Zeit als System- und Netzwerkadministrator, hat sich mir aber der Spruch eingeprägt, dass jede Lösung die gut genug ist, nicht gefeuert zu werden eine gute Lösung ist. Zu meiner Freude durfte ich feststellen, dass dieser Spruch sinngemäß aus dem Vorwort des allseits beliebten Kamel-Buchs stammt. Womit wir bei unserem eigentlichen Thema angelangt sind, wie arbeite ich mit Daten (als Plural von Datum) und Zeiten in Perl?

```
#!/bin/bash

let when=`cal | cut -d , , -f6 | egrep "[0-9]{2}" | tail -n1`
let now=`date "+%d"`

if [ $when -eq $now ]; then
    echo "doing backup ..."
    # do whatever is necessary to backup
fi
```

Listing 2

Und jetzt in Perl!

Ziel dieses Artikels ist die Vorstellung des Moduls `Date::Calc`, das eines der am häufigsten benutzten Module zum Zweck der Datumsberechnung ist und als ausgereift betrachtet werden darf. Auf das Eingangsbeispiel, der Frage nach dem letzten Freitag im Monat werde ich immer mal wieder zurückkommen, aber die Darstellung der Verwendung des Moduls `Date::Calc` steht im Vordergrund.

`Date::Calc` ist ein ausgereiftes Modul, das über eine hervorragende Dokumentation verfügt. Etwas gewöhnungsbedürftig ist sicher, dass es sich um kein objektorientiertes Modul handelt, auch wenn es mit `Date::Calc::Object` eine entsprechende Alternative gibt. Das Arbeiten mit `Date::Calc` funktioniert aber, nach kurzer Eingewöhnung, auch so reibungslos weshalb ich die klassische Variante darstellen möchte.

Als ersten Schritt beginnen wir mit einem ganz kleinen Skript (siehe Listing 3), das einige der Grundfunktionen von `Date::Calc` nutzt.

Mit `use Date::Calc qw/:all/;` wird `Date::Calc` sowie alle Funktionen die es exportiert in das Skript eingebunden. `Data::Dumper` wird in dem Beispiel genutzt um die Rückgabewerte der Funktionen `Now()` und `Today()` auszugeben. Wie man feststellen kann, handelt es sich bei den Rückgabewerten um einfache Listen, die einem Array zugewiesen werden können:

```
$VAR1 = [ 23, 30, 59 ];
$VAR1 = [ 22, 30, 59 ];
$VAR1 = [ 2007, 12, 23 ];
```

Der Unterschied zwischen der ersten und der zweiten Ausgabe besteht darin, dass bei dem zweiten Aufruf der Funktion `Now` als Parameter die Variable `$gmt` mit übergeben wurde. `Date::Calc` erlaubt bei einigen der Funktionen die Übergabe eines Wahrheitswertes um festzulegen ob mit der lokalen Zeit gearbeitet werden soll, oder sich auf die GMT, die greenwich mean time, bezogen werden soll.

Ebenfalls wichtig ist die Reihenfolge der zurückgegebenen Werte. Bei der Funktion `Now()` erhalten wir die Zeit im Format Stunde, Minute, Sekunde, bei der Funktion `Today()` im Format Jahr, Monat, Tag. Ein Quell steten Ärgernisses sind die unterschiedlichen Datumsformate mit denen man als Programmierer konfrontiert wird. Gerade deshalb ist es wichtig zu wissen in welchem Format ein Modul, das man nutzen möchte, intern arbeitet. Wichtig ist aber auch, dass man unterschiedliche Formate ausgeben kann, oder auch einlesen.

Verrückte Zeiten

`Date::Calc` erlaubt die Ausgabe des Datums in den verschiedenen landesüblichen Formaten und Sprachen. Voreinstellung ist dabei aber Englisch (siehe Listing 4)

Das Skript führt zu folgender Ausgabe:

```
Mon 24-Dec-2007
Monday, December 24th 2007
English
Mon 24-Dez-2007
Montag, den 24. Dezember 2007
Deutsch
```

Die Funktion `Language()` kann genutzt werden um den Wert der aktuellen Sprache abzufragen, oder neu zusetzen. Dabei wird intern ein numerischer Wert verwendet, weshalb die Funktion `Language_to_Text()`, bzw. die Funktion `Decode_Language()` benötigt wird um eine lesbare Repräsentation dieses Wertes zu erhalten, oder diese in die numerische Form umzuwandeln.

`Date_to_Text()` und `Date_to_Text_Long()` geben das Datum, das z.B. die Funktion `Today()` liefert in der landestypischen Darstellung aus.

Externe Eingaben

Glücklich ist jener, der nicht gezwungen ist externe Daten abzufragen und zu verarbeiten. Meistens kommen wir aber nicht drum herum, Daten (ihr wisst schon, der Plural von Datum) aus Eingabefeldern, .csv-Dateien oder sonstigen Quellen zu verwenden. Diese Daten sehen meistens genau so aus, wie wir es gerade nicht brauchen können, aber wie sehen denn Daten aus die wir haben möchten? Und welche Möglichkeiten haben wir diese Daten mit `Date::Calc` zu verarbeiten?

`Date::Calc` kennt drei Funktionen um Daten aus Text zu parsen: `Decode_Date_EU()`, `Decode_Date_US()` und `Parse_Date`, wobei letzteres dafür gedacht ist die Ausgabe des Unix-Befehls `date` zu parsen. Zu den ersten beiden gibt es nochmal Alternativen die in reinem Perl implementiert sind, aber funktional keine großen Unterschiede bieten (das ist jetzt der Zeitpunkt mal in die Doku zu schauen und Widerspruch zu erheben).

```
#!/usr/bin/perl

use strict;
use warnings;

use Date::Calc qw/:all/;
use Data::Dumper;

my $gmt = 1;                # true for using GMT instead of localtime

print Dumper [Now()];       # now in localtime
print Dumper [Now($gmt)];   # now in greenwich
print Dumper [Today()];     #
```

Listing 3

```
#!/usr/bin/perl

use strict;
use warnings;

use Date::Calc qw/:all/;

print Date_to_Text(Today()), "\n";             # print human readable
print Date_to_Text_Long(Today()), "\n";        # long format
print Language_to_Text(Language()), "\n";      # in english (default)

Language(Decode_Language("DE"));               # setting format to german
print Date_to_Text(Today()), "\n";             # print human readable but german
print Date_to_Text_Long(Today()), "\n";        # also in long format
print Language_to_Text(Language()), "\n";      # language is now ‚Deutsch'
```

Listing 4

```
#!/usr/bin/perl

use strict;
use warnings;

use Date::Calc qw/:all/;
use Data::Dumper;

Language(Decode_Language("DE"));               # setting format to german

while (<DATA>) {                               # get each line from DATA section
    my @date = Decode_Date_EU($_);             # decode current EU style

    if (@date && check_date(@date)) {          # if result is a valid date:
        print Date_to_Text(@date), "\n";       # print human readable to STDOUT
    } else {
        print Dumper [Decode_Date_US($_)];     # try US style:
                                               # will also fail for 30.02.2008
    }
}

__DATA__
16.12.07
16.12.2007
1.1.2008
29.2.2008
29.02.2008
30.02.2008
Abgabetermin am 28. 12. 2007!
12/13/2007
```

Listing 5

Die Ausgabe des obigen Skriptes sieht wie folgt aus:

```
Son 16-Dez-2007
Son 16-Dez-2007
Die 1-Jan-2008
Fre 29-Feb-2008
Fre 29-Feb-2008
$VAR1 = [];
Fre 28-Dez-2007
$VAR1 = [ 2007, 12, 13 ];
```

Deutlich zu erkennen ist, dass fast jedes Datum erkannt wird, mit Ausnahme des 30. 02. 2008, den es nun wirklich nicht gibt. Die Funktion `Decode _ Date _ EU()` liefert ein Liste im Format Jahr, Monat, Tag wie wir es schon von der Funktion `Today()` kennen. Wurde kein gültiges Datum erkannt, wird auch kein Wert zurückgegeben. Aus diesem Grund könnte man sich an dieser Stelle auch die Überprüfung der Liste mit der Funktion `check _ date()` sparen. Denoch wollte ich diese Funktion an dieser Stelle vorstellen, weil sie erlaubt zu überprüfen ob ein Datum ein gültiges Datum ist.

Wie alle Funktionen in `Date::Calc` die einen Wahrheitswert zurückliefern wird der Funktionsname `check _ date()` in Kleinbuchstaben geschrieben, im Gegensatz zu allen anderen Funktionen die `Date::Calc` zur Verfügung stellt.

Im obigen Skript wird bei Zeilen in denen die Datumserkennung mit `Decode _ Date _ EU()` fehlschlug probiert das Datum mit der Funktion `Decode _ Date _ US()` zu erkennen. In der Praxis ist dies sicherlich keine gute Idee, man sollte wissen in welchem der beiden Formate die Daten vorliegen, oder dies an einem Attribut erkennen können. Das US-Format unterscheidet sich durch das EU-Format dadurch, das es in der Reihenfolge Monat/Tag/Jahr dargestellt wird, was bei Tagen vor dem 13. zu Mehrdeutigkeiten führt.

Unix-Zeitstempel

Unix-Zeitstempel werden in Sekunden seit dem 1. Januar 1970 gerechnet (als Epoch bezeichnet). Sie kommen u.U. in Logdateien vor, oder in Datenbanken. Der Vorteil von Unix-Zeitstempeln ist sicherlich, dass man mit Sekunden vergleichbar gut rechnen kann. Ein großer Nachteil ist aber, das fast niemand mit diesen sehr großen Zahlenwerten direkt etwas anfangen kann. `Date::Calc` stellt auch hierfür Funktionen zur Verfügung: `Time _ to _ Date` und `Date _ to _ Time`. Bei der Verwendung der Funktionen ist darauf zu achten, dass als Argumente, bzw. Rückgabewerte eine komplette Liste verwendet wird, mit Jahr, Monat, Tag, Stunde, Minute, Sekunde wie folgendes Beispiel darstellt:

```
my $unixtimestamp = qx/date +%s/;
 my @date = Time _ to _ Date($unixtimestamp);

print Dumper \@date;
print Date _ to _ Text(@date[0..2]), "\n";
```

Da die Funktion `Date _ to _ Text()` aber nur drei Werte erwartet, ist es notwendig ein Array-Slice zu verwenden, oder alternativ und übersichtlicher:

```
my ($y, $m, $d) = Time _ to _ Date(
     $unixtimestamp);
print Date _ to _ Text($y, $m, $d), "\n";
```

Bei der obigen Listenzuweisung, werden die drei Werte, die nicht genutzt werden einfach verworfen.

Freitags-Skripte

`Date::Calc` stellt eine Menge weiterer nützlicher Funktionen zur Verfügung. `Day _ of _ Year()` liefert zu einem gegebenen Datum die Information, der wievielte Tage des Jahres es ist. Nicht verwechseln sollte man das mit der Funktion `Date _ to _ Days()`, die einen ähnlichen Namen hat, aber die Tage seit dem 01.01.0001 zurückliefert. Mit der Funktion `Day _ of _ Week()` erhält man einen numerischen Wert für den Wochentag, der mit der Funktion `Day _ of _ Week _ to _ Text()` in eine lesbare Form umgewandelt werden kann. Wer den numerischen Wert des Wochentages für Berechnungen nutzen möchte, sollte beachten, dass die Zählweise mit dem Montag mit 1 beginnt. `Week _ Number()` liefert die Kalenderwoche und mit `Monday _ of _ Week()` kann man für eine beliebige Woche das Datum des Montags bestimmen. Die Verwendung der Funktionen wird nachfolgend in Listing 6 dargestellt.

Am Ende des Skriptes befindet sich eine Funktion `Nth _ Weekday _ of _ Month _ Year()` die den n-ten Wochentag eines Monats liefert. Diese Funktion ist äußerst nützlich um wiederkehrende Termine zu bearbeiten, wie das Perlmonger-Treffen an jedem dritten Donnerstag im Monat. Der aufmerksame Leser wird jetzt sagen: Hey, das ist doch genau die Funktion die man benötigt um den letzten Freitag des Monats (der 5. Tag der Woche) zu bestimmen! Ja, eine Perl-Version des Backup-Skriptes ist mit dieser Funktion möglich, wobei man noch überprüfen muss, ob es sich um den 5. oder 4. Freitag im Monat dreht (siehe Listing 7).

```
#!/usr/bin/perl

use strict;
use warnings;

use Date::Calc qw/:all/;
use Data::Dumper;

my @date = Today();                         # 26.12.2007

print Day_of_Year(@date),    "\n",          # since 01.01.
      Date_to_Days(@date),   "\n",          # since 01.01.0001
      Day_of_Week(@date),    "\n",          # starts on monday with 1
      Day_of_Week_to_Text(
          Day_of_Week(@date)                # human readable
      ),                       "\n",
      Week_Number(@date),    "\n",          # german: Kalenderwoche
      Dumper ( [ Monday_of_Week(            # dump date of ...
          Week_Number(@date), $date[0]      # ... this weeks monday
      ) ] ),                   "\n";        #

print Dumper [Nth_Weekday_of_Month_Year(2007,11,5,5)]; # 30.11.2007
print Dumper [Nth_Weekday_of_Month_Year(2007,12,5,5)]; # undef
```

Listing 6

```
#!/usr/bin/perl

use strict;
use warnings;

use Date::Calc qw/:all/;

my @today         = Today();
my @this_month    = @today[0,1];

my @last_friday = Nth_Weekday_of_Month_Year(@this_month,5,5) ?
                   Nth_Weekday_of_Month_Year(@this_month,5,5) :
                   Nth_Weekday_of_Month_Year(@this_month,5,4) ;

my $dD = Delta_Days(@today, @last_friday); # chronologically order

if ($dD == 0) {
    # do backup ...
} else {
    print "$dD day(s) to go ...\n";
}
```

Listing 7

Die Funktion `Nth_Weekday_of_Month_Year()` erwartet vier Parameter: Jahr, Monat, Wochentag (in numerischer Form) und der wievielte Wochentag des Monats gesucht wird. Im obigen Skript wird auch eine Funktion genutzt, die noch nicht vorgestellt wurde, aber zu einer ganzen Reihe von Funktionen gehört die genutzt werden können um mit Daten zu rechnen. Die Funktion `Delta_Days()` liefert den Abstand in Tagen zwischen zwei Daten, die in chronologischer Reihenfolge angegeben werden sollen.

Berechnungen

Eine große Gruppe an Berechnungsfunktionen des Moduls `Date::Calc` beschäftigt sich mit der Addition von Zeitabständen, bzw. deren Feststellung, zwischen zwei Daten. Eine Handvoll dieser Funktionen möchte ich abschließend vorstellen. Prinzipiell funktionieren alle Funktionen dieser Gruppe ähnlich, nur die Anzahl der benötigten Parameter unterscheidet sich. Bedarfsweise ist ein Blick in die hervorragende Dokumentation des Moduls ist empfehlenswert (siehe Listing 8).

Die Funktion `Delta_Days()` wurde bereits kurz im vorherigen Abschnitt vorgestellt. Sie hat die Aufgabe den Abstand in Tagen, zwischen zwei Daten zu ermitteln. Das Gegenstück dieser Funktion, ist `Add_Delta_Days()` die auf ein gegebenes Datum eine Anzahl von Tagen addiert und das Ergebnis im gewohnten Format zurückgibt. Wo genauer gerechnet werden muss, findet die Funktion `Add_Delta_YMDHMS()`

```
#!/usr/bin/perl

use strict;
use warnings;

use Date::Calc qw/:all/;
use Data::Dumper;

my @today = Today();                              # 26.12.2007
my @then  = (2007,12,31);

my $days_to_go = Delta_Days(@today, @then);
print $days_to_go, "\n";

my @future_day = Add_Delta_Days(@today, 100);    # 100 days in the future ...
print Date_to_Text(@future_day), "\n";           # print human readable

my ($y, $m, $d, $h, $min, $sec) = (1,2,3,4,5,6); # one year, two months, ...
@future_day = Add_Delta_YMDHMS(@today, 0, 0, 0, $y, $m, $d, $h, $min, $sec);

print Date_to_Text(@future_day[0..2]), "\n";     # print human readable

@future_day = (2009,12,31);
my ($dy, $dm, $dd) = Delta_YMD(@today, @future_day);
print "Last year of the decade in $dy year(s), $dm month(s) and $dd day(s)\n";
```

Listing 8

Anwendung, die sowohl das aktuelle Datum und Uhrzeit, als auch die Anzahl an Jahren, Monaten, Tagen, Stunden, Minuten und Sekunden als Parameter erwartet. Auch diese Funktion hat ein passendes Gegenstück in der Funktion `Delta_YMDHMS()`. Etwas weniger Parameter erwartet die letzte vorgestellte Funktion `Delta_YMD()`, die ähnlich der Funktion `Delta_Days()` arbeitet, den Abstand zwischen zwei Daten aber in Jahren, Monaten und Tagen zurückliefert. `Date::Calc` beinhaltet selbstverständlich noch mehr dieser Funktions-Paare, die aber immer dem selben Schema folgen und in der Dokumentation ausführlich beschrieben sind.

Fazit

Eine vollständige Darstellung aller Methoden des Moduls `Date::Calc` wäre im Rahmen dieses Artikels nicht möglich und wahrscheinlich auch wenig sinnvoll gewesen. Ziel war es dem geneigten Leser einen Überblick über häufig genutzte Funktionen zu bieten und zu ermutigen bei Aufgabenstellungen, die Datumsberechnungen benötigen, ein Modul wie `Date::Calc` zu verwenden. Natürlich finden sich im CPAN andere, ähnlich ausgereifte Module, deren Nutzung genauso empfehlenswert ist.

Der nächste große Datumsfehler-Hype, nach dem Y2K, steht uns dann übrigens 2038 ins Haus, wenn die Unix-Timestamps überlaufen. Bis dahin bleibt aber noch ein wenig Zeit und knapp 30 Jahre spannender Technologie-Entwicklung.

\# Ronnie Neumann

TPF-TICKER

Neuer Grant Committee Vorsitzender

Alberto Simões ist neuer Vorsitzender des Grant Committees. Der Vorsitzende des „Portuguese Association for Perl Programmers“ setzte sich gegen drei andere Kandidaten durch. Simões wird vor allem für die „Call for Proposals“ und alle anderen Sachen rund um das Thema „Grant“ verantwortlich sein.

Die Tätigkeit eines Vorsitzenden ist Alberto Simões nicht unbekannt: Sowohl bei der „Portuguese Association for Perl Programmers“ und bei Braga.pm ist er der Vorsitzende.

Die Perlmongersgruppe in Erlangen

Bierliebhaber kennen Franken als ein Gebiet mit einer der höchsten Brauereidichten in Deutschland. Perl-Freunde kennen Franken als den Ausrichtungsort des 10. Deutschen Perl-Workshops vom 13. bis 15. Februar 2008. Dieser findet im Herzen des schönen Frankenlandes statt - in Erlangen. Und Erlangen wiederum hat eine sehr aktive Perlmongers-Gruppe, die Erlangen.pm!

Seit Anfang 2008 wird an jedem dritten Montag im Monat ein ruhiges italienisches Restaurant in Erlangen zum Austragungsort hitziger Diskussionen, zur Bühne interessanter Vorträge und zum kulinarischen Rahmen eines entspannten Treffens unter Gleichgesinnten. Dann ist das Erlangen.pm-Treffen!
Der frühere Zeitpunkt und Ort der Treffen (jeder dritte Donnerstag) resultierte dabei aus der zufälligen Wahl beim allerersten Treffen.

Damals, vor gut zwei Jahren musste der sächsische Zuwanderer „steffenw“ enttäuscht das Fehlen einer Perlmongers-Gruppe in Franken feststellen. Perlmongers-Gruppen kannte „steffenw“ bereits von Frankfurt.pm und Dresden.pm. So machte er sich auf die Suche nach anderen Perl-Programmierern in der Region. Dem Perl-Programmierer „horshack“ kam die Idee einer Perlmongersgruppe in seiner Region ebenfalls sehr entgegen. Die Leidenschaft, mit der die beiden in Perl programmieren, verbindet über Bundesländergrenzen hinweg. „horshack“ und „steffenw“ kannten sich darüber hinaus bereits flüchtig von diversen Perl-Workshops. Wie sich dann aber herausstellte, waren „horshack“ und „steffenw“ nicht die einzigen, die Perl nicht nur als ihren Beruf ansahen.

Seit damals treffen sich regelmäßig zwischen 2 und 15 Perl-Programmierer einmal im Monat. Die Programmierer kommen dabei aus einem Umkreis von bis zu 50 km. Der neueste Zugang scheut noch nicht einmal den Weg von 8500 km aus Shanghai, um dem Treffen beizuwohnen. Zugegeben, das Treffen der Erlanger PM-Gruppe ist nicht der ausschließliche Grund der Anreise.

Neben den regelmäßigen Treffen halten die Erlanger Perlmongers in unregelmäßigen Abständen sogenannte Freaktreffen ab. Diese Freaktreffen finden Wochenends am Dutzendteich in Nürnberg statt. Es gibt Themenschwerpunkte die in der realen Computerwelt angegangen werden. Möglich werden die Freaktreffen durch die guten Kontakte zum Kommunikationsnetz Franken, einem Internet- und Computerverein in Nordbayern mit mehr als 500 Mitgliedern.

Die Freaktreffen der Erlangen-PM behandeln immer ein zentrales Thema. Dieses Thema wird von den Teilnehmern gemeinsam angegangen. So wurde beim letzten Freaktreffen das Thema Jifty unter die Lupe genommen. Dazu haben die Mitglieder gemeinsame Testserver aufgestellt und erarbeiteten gemeinsam eine Weblösung auf Basis von Jifty.

Der bislang größte Coup der Erlanger Perlmongers ist jedoch die lokale Ausrichtung des 10. Deutschen Perl-Workshops 2008. Fieberhaft wird bereits seit einer Weile auf dieses Großevent der deutschsprachigen Perlkultur hin gearbeitet. Es werden von der Erlanger Gruppe keine Kosten und Mühen gescheut, um Erlangen als würdigen Veranstaltungsort für diesen geschichtsträchtigen Workshop vorzubereiten. Jedes Treffen der Erlangen-Perlmongers steht im Zeichen der Vorbereitungen.

Gute Kontakte zum Regionalen Rechenzentrum Erlangen der Friedrich-Alexander-Universität Erlangen-Nürnberg helfen hier. Diese Kontakte sind besonders notwendig, da der 10. Deutsche Perl-Workshop in den Räumen des Regionales Rechenzentrums Erlangen stattfindet.

Steffen Winkler

CPAN News V

JavaScript::Minifier

Viele Webseiten verwenden JavaScript - gerade in AJAX-überladenen Seiten kommen da schnell einige hundert Kilobyte nur an JavaScript zusammen. Das vergrößert die Ladezeiten, was meistens nicht gewollt ist. Wenn man sich diese JavaScript-Elemente anschaut, fällt auf, dass viele Whitespaces das Element unnötig groß machen. Da kommt JavaScript::Minifier ins Spiel. Das Modul minimiert JavaScript soweit es geht - ohne die Funktionalität einzuschränken.

```
#!/usr/bin/perl

use strict;
use warnings;
use JavaScript::Minifier qw(minify);

my $js = <<'EOM';
    function test(){
      var id        = 1;
      var myobject = document.forms[0];

      if( id == 1 ){
          alert( 'test' );
      }
    }
EOM

my $minified = minify( input => $js );
print $minified;
```

File::Tee

Die gleiche Ausgabe wie auf der Konsole auch in einer Datei zu haben, ist häufig aus Debugging-Gründen wichtig. Bevor man sich eine eigene Routine schreibt, kann man auch das Modul File::Tee verwenden. Die Verwendung ist sehr simpel, aber leider funktioniert das auf Windows nicht.

```
#!/usr/bin/perl

use strict;
use warnings;
use File::Tee qw(tee);

tee( STDOUT, '>>teed.txt' );
print "Eine Testausgabe\n";
```

MySQL::Backup

Automatisierte Backups haben einen ganz wichtigen Vorteil: man vergisst nicht Backups zu machen. Hier hilft das Modul MySQL::Backup, das sowohl die Struktur als auch die Daten in einen Dump schreibt. Mit dem Skript ist es auch möglich, die Daten wieder einzuspielen - und dabei entweder alle bestehenden Daten löscht oder einen Abgleich macht und nur nicht-vorhandene Daten in die Datenbank schreibt.

```
#!/usr/bin/perl

use strict;
use warnings;
use MySQL::Backup;

my $backupper = MySQL::Backup->new(
      'datenbank',
      'localhost',
      'user',
      'passwort',
      { SHOW _ TABLE _ NAMES => 1 },
);

my $output = './mysql _ dump.sql';
if( open my $out, ,>', $output ){
      print $out $backupper
         ->create _ structure,"\n",
       $backupper->data _ backup;
     }
```

HTML::ReportWriter

Bei vielen Web-Anwendungen werden Datenbank-Daten in tabellarischer Form angezeigt, ohne dass sie groß verändert oder angepasst werden. Mit HTML::ReportWriter kann man sich das Leben einfacher machen und auch sortierbare HTML-Tabellen erzeugen. Allerdings hat das Modul den Nachteil, dass das Aussehen des HTML-Outputs nur sehr umständlich angepasst werden kann. Auch den SQL-Befehl muss man fast komplett selbst schreiben.

```
#!/usr/bin/perl

use HTML::ReportWriter;
use CGI;
use DBI;

my $dbh = DBI->connect(
   $dsn, $user, $password )
                  or die $DBI::errst;
my $report = HTML::ReportWriter->new({
    DBH           => $dbh,
    CGI _ OBJECT    => CGI->new,
    DEFAULT _ SORT => 'Sortierende Spalte',
    SQL _ FRAGMENT =>
       'FROM tabelle WHERE spalte = 1',
    COLUMNS       =>
       [qw/spalten die angezeigt werden/],
  });
```

WebService::Audioscrobbler

Gibt es zu einem Künstler ein Lied? Welche Tags sind mit dem Künstler verbunden? Gerade für Nutzer von last.fm ist WebService::Audioscrobbler interessant. Damit lassen sich alle möglichen Zusammenhänge herausfinden: Von einem Lied über den Künstler zu „ähnliche Künstler“. Vielleicht findet man so auch seine neue Lieblingsband.

```
#!/usr/bin/perl

use strict;
use warnings;
use WebService::Audioscrobbler;

my $artist_name = 'Phil Collins';

my $ws = WebService::Audioscrobbler->new;
my $artist = $ws->artist( $artist_name );

print $_->url,"\n" for $artist->tracks;
```

Module::Cloud

Wer sich dafür interessiert, welches seine meistbenutzten Module sind, sollte sich mal Module::Cloud anschauen. Dem Modul übergibt man ein Verzeichnis und dann werden alle Dateien, die nach Perl-Code „aussehen“ überprüft. Danach kann man einen Tag-Cloud erstellen - dazu wird HTML::Tag-Cloud verwendet. Interessant wird das vor allem, wenn man bestimmte Projekte einmal damit auswertet.

```
#!/usr/bin/perl

use strict;
use warnings;
use CGI;
use Dir::Self;
use Module::Cloud;

print CGI::header();

my $mc  = Module::Cloud->new(
    dir => __DIR__,
);

my $cloud = $mc->get_cloud;
print $cloud->html_and_css;
```

Termine

Februar 2008

04. Treffen Vienna.pm
05. Treffen Frankfurt.pm
07. Treffen Dresden.pm
13. 10. Deutscher Perl-Workshop in Erlangen
14. 10. Deutscher Perl-Workshop in Erlangen
15. 10. Deutscher Perl-Workshop in Erlangen
18. Treffen Erlangen.pm
21. Treffen Darmstadt.pm
23. FROSDEM 2008 in Brüssel
    1. Ukrainischer Perl-Workshop in Kiew
24. FROSDEM 2008 in Brüssel
27. Treffen Bielefeld.pm
29. Niederländischwer Perl-Workshop

März 2008

03. Treffen Vienna.pm
04. Treffen Frankfurt.pm
06. Treffen Dresden.pm
17. Treffen Erlangen.pm
20. Treffen Darmstadt.pm
26. Treffen Bielefeld.pm

April 2008

03. Treffen Dresden.pm
07. Treffen Vienna.pm
08. Treffen Frankfurt.pm
17. Treffen Darmstadt.pm
21. Treffen Erlangen.pm
30. Treffen Bielefeld.pm

Hier finden Sie alle Termine rund um Perl.

Natürlich kann sich der ein oder andere Termin noch ändern. Darauf haben wir (die Redaktion) jedoch keinen Einfluss.

Uhrzeiten und weiterführende Links zu den einzelnen Veranstaltungen finden Sie unter

http://www.perlmongers.de

Kennen Sie weitere Termine, die in der nächsten Ausgabe von $foo veröffentlicht werden sollen? Dann schicken Sie bitte eine EMail an:

termine@foo-magazin.de

LINKS

http://www.perl-nachrichten.de

Unter Perl-Nachrichten.de sind deutschsprachige News rund um die Programmiersprache Perl zu finden. Jeder ist dazu eingeladen, solche Nachrichten auf der Webseite einzureichen.

http://www.Perl-community.de

Perl-Community.de ist eine der größten deutschsprachigen Perl-Foren. Hier ist aber nicht nur ein Forum zu finden, sondern auch ein Wiki mit vielen Tipps und Tricks. Einige Teile der Perl-Dokumentation wurden ins Deutsche übersetzt. Auf der Linkseite von Perl-Community.de findet man viele Verweise auf nützliche Seiten.

http://www.perlmongers.de/
http://www.pm.org/

Das Online-Zuhause der Perl-Mongers. Hier findet man eine Übersicht mit allen Perlmonger-Gruppen weltweit. Außerdem kann man auch neue Gruppen gründen und bekommt Hilfe...

http://www.perl-workshop.de

Der Deutsche Perl-Workshop hat sich zum Ziel gesetzt, den Austausch zwischen Perl-Programmierern zu fördern. Der 10. Deutsche Perl-Workshop findet im Februar 2008 in Erlangen statt.

http://www.perl-foundation.org

Die Perl-Foundation nimmt eine führende Funktion in der Perl-Gemeinde ein: Es werden Zuwendungen für Leistungen zugunsten von Perl gegeben. So wird z.B. die Bezahlung der Perl6-Entwicklung über die Perl-Foundationen geleistet. Jedes Jahr werden Studenten beim „Google Summer of Code“ betreut.

http://www.Perl.org

Auf Perl.org kann man die aktuelle Perl-Version downloaden. Ein Verzeichnis mit allen möglichen Mailinglisten, die mit Perl oder einem Modul zu tun haben, ist dort zu finden. Auch Links zu anderen Perl-bezogenen Seiten sind vorhanden.

// SEIBERT / MEDIA

„Statt mit blumigen Worten umschreiben unsere Programmierer den Job so:
Apache, Catalyst, CGI, DBI, JSON, Log::Log4Perl, mod_perl, SOAP::Lite, XML::LibXML, YAML“

Professionell Perl programmieren lernen?

Wir suchen Einsteiger als Perl-Programmierer/innen (Vollzeit/Praktikum)

//SEIBERT/MEDIA besteht aus den drei Kompetenzfeldern Technologie, Consulting und Design und gehört zu den erfahrenen und professionellen Internet-Agenturen in Deutschland. Wir entwickeln seit 1996 mit heute knapp 50 Mitarbeitern Intranets, Extranet-Systeme, Web-Portale aber auch klassische Internet-Seiten. Seit 2005 konzipiert unsere Designabteilung hochwertige Unternehmensauftritte und kommunikative Konzepte. Beratung im Bereich Online-Marketing und Usability runden das Leistungsportfolio ab.

Ihre Aufgabe wird sein, in unserer Entwicklungsabteilung im Team komplexe E-Business Applikationen zu entwickeln. Dabei ist objektorientiertes Denken genauso wichtig, wie das Auffinden individueller und innovativer Lösungsansätze, die gemeinsam realisiert werden.

Wir freuen uns auf Ihre Bewerbung unter http://www.seibert-media.net/jobs/.

//SEIBERT/MEDIA GMBH
RHEINGAU PALAIS / SÖHNLEINSTRASSE 8
65201 WIESBADEN
T. +49 611 20570-0 / F. +49 611 20570-70
BEWERBUNG@SEIBERT-MEDIA.NET
HTTP://WWW.SEIBERT-MEDIA.NET/JOBS/

FREAKZEIT im Linuxhotel

DAS Wochenend-Reiseziel für alle, die Spaß an Computern haben

Wo andere nach Hamburg ins Musical oder in den Schwarzwald zur Wellness fahren, treffen sich Computerfreaks nun jedes Wochenende im Linuxhotel, um zusammen mit Gleichgesinnten am PC zu arbeiten.

Es erwartet Sie eine ruhige, konzentrierte Umgebung mit Leihnotebooks zum Testen kritischer Installationen, Seminarraum, Technik, Bibliothek, Probeservern, WLAN, 230V-Steckdosen selbst im 25.000qm großen Privatpark und vielem mehr. Auch Vollpension und Getränke sind enthalten, bei schönem Wetter wird gegrillt, wenn's kalt ist, geht es in die Sauna. Fahrräder stehen bereit, kleine Modellflieger, Fitnessraum, eine Wii, und vor allem all' der Kleinkram, den man für erfolgreiche Computerarbeiten braucht.

Warum nicht 'mal ein Wochenende unter Computerfreaks? Jeder arbeitet an seinem Thema, aber man schaut sich zwanglos über die Schultern und hilft sich gegenseitig!

Die Freakzeit-Wochenenden sind ein Experiment, denn eigentlich lebt das Linuxhotel von sehr zufriedenen Unternehmen und Organisationen, die ihre Mitarbeiter bei uns schulen lassen (wir haben eines der breitesten Kursprogramme in Deutschland, siehe www.Linuxhotel.de).

Doch die IT-Branche ist etwas Besonderes! Viele Profis haben wirklich Spaß an ihrer Arbeit, und es ist nicht einzusehen, warum z.B. Musical-Freunde oft mehrfach pro Jahr von weit her zu ihren Theatern fahren, doch für Computerfreaks gibt es nichts Vergleichbares!

Das Linuxhotel soll der Ort werden, an dem man jedes Wochenende interessante Leute aus unserer Branche zwanglos treffen kann. Machen Sie mit, schnappen Sie sich Ihr Notebook, und melden sich für irgendeines der kommenden Wochenenden an! Siehe www.Linuxhotel.de und auf „Freakzeit“ klicken.