Ausgabe 4/2008 – www.foo-magazin.de - 6,00 €

$foo PERL MAGAZIN

CPAN::Mini
Perls Schatztruhe auf meinem eigenen Rechner!

XS Tutorial
Perl mit C erweitern

Google Summer of Code
und Perl war dabei!

Logging für Perl-Programme
Ich weiß was Du letzten Sommer getan hast...

Nr 08

Sichern Sie Ihren nächsten Schritt in die Zukunft

Astaro macht Netzwerksicherheit einfach. Als Marktführer in Deutschland für UTM (Unified Threat Management) bietet Astaro die funktionsreichste All-In-One-Appliance für den Schutz von Netzwerken. Auf Sie warten ein dynamisches Team und eine internationale Arbeitsumgebung mit einem Partnernetzwerk und Niederlassungen weltweit. Arbeiten Sie in den Zukunftsmärkten IT-Sicherheit und OpenSource. Packen Sie mit an!

Astaro wächst weiter, zur Verstärkung unserer Produktentwicklung in Karlsruhe suchen wir:

Perl Backend Developer (m/w)

Ihre Aufgaben sind:

- Aktive Mitarbeit an der Produktentwicklung
- Entwicklung und Pflege technisch anspruchsvoller Software-Systeme im Backend-Bereich
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Diese Stärken sollten Sie mitbringen:

- Fundierte Kenntnisse der Programmiersprache Perl
- Kenntnisse in anderen Programmier- oder Scriptsprachen wie C oder Bash wären von Vorteil
- Sehr gute Englischkenntnisse
- Selbstständiges Planen, Arbeiten und Reporten

Perl/C Software Developer (m/w)

Ihre Aufgaben sind:

- Aktive Mitarbeit an der Produktentwicklung
- Systemnahe Entwicklung technisch anspruchsvoller Software unter Linux
- Tatkräftige Unterstützung beim Aufbau und der Pflege des internen technischen Know-hows

Diese Stärken sollten Sie mitbringen:

- Sehr gute Kenntnisse der Programmiersprachen Perl und C
- Kenntnisse des Linux Kernels und insbesondere des Network Stacks wären von Vorteil
- Sehr gute Englischkenntnisse
- Selbstständiges Planen, Arbeiten und Reporten

Bei uns erwarten Sie ein leistungsorientiertes Gehalt, ein engagiertes Team sowie ein freundliches, modernes Arbeitsumfeld mit flexiblen Arbeitszeiten und allem, was Sie bei Ihrer täglichen Arbeit unterstützt. Begeisterte Mitarbeiter liegen uns am Herzen: Neben einer spannenden Herausforderung am attraktiven IT-Standort Karlsruhe bieten wir Ihnen das besondere Flair im sonnigen Baden und subventionierte Sportangebote am Arbeitsplatz.

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

Astaro AG • Monika Heidrich
Amalienbadstr. 36/Bau 33a
D-76227 Karlsruhe
Tel.: 0721 25516 0

www.astaro.de

e-mail | web | net | security

astaro internet security

VORWORT

I'm in your Perl

Wäre es nicht schön, von sich sagen zu können, dass man an einer Software mitgearbeitet hat, die von zig tausend Usern verwendet wird? Bei Open Source Software ist dies relativ leicht möglich. Auch bei Perl ist das so...

Die nächste Version in der 5.8.x-Reihe -- Perl 5.8.9 -- wird bald veröffentlicht. Bis dahin ist noch etwas Arbeit zu erledigen. Die Hauptarbeit dürfte wohl Nick Clark haben, der Patches und andere Änderungen in den maint-5.8-Zweig im Perforce-Repository übernehmen muss. Aber es sind noch viel mehr Personen involviert:

Paul Fenwick hat ein Projekt zur Dokumentation der Änderungen -- der perl589delta -- gestartet, damit die Arbeit auf viele Schultern verteilt wird. Jeder kann daran teilnehmen und seinen Beitrag zur Perl 5.8.9 beitragen.

Überhaupt ist jeder Entwickler willkommen. Man muss sich nicht unbedingt im Perl Core auskennen. Schon das Testen der aktuellen Sourcen kann viel helfen. Die Perl5-Porters haben nicht alle Betriebssysteme zur Verfügung und hoffen auf User, die mit "exotischen" Betriebssystemen aushelfen können. Mit den Smoke-Tests wird die Qualität von Perl sichergestellt.

Auch für Leute, die Programmieren wollen, gibt es genug zu tun. In der Dokumentation findet sich das Dokument `perltodo`, in der TODOs aufgelistet sind, für die man entweder nur Perl-Kenntnisse, Perl- und C-Kenntnisse oder nur C-Kenntnisse erforderlich sind.

Hilfe ist immer willkommen. Wer jetzt interesse bekommen hat, kann sich an die Perl5-Porters-Mailingliste wenden.

Jetzt aber viel Spaß mit der neuen Ausgabe von $foo.

Die Codebeispiele können mit dem Code

jWeru3

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

Renée Bäcker

Ab dieser Ausgabe werden alle weiterführenden Links auf del.icio.us gesammelt. Für diese Ausgabe: http://del.icio.us/foo_magazin/issue8

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

ALLGEMEINES

MODULE

PERL

TIPPS & TRICKS

USER-GRUPPEN

NEWS

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.

Ferry Bolhár-Nordenkampf

Ferry kennt Perl seit 1994, als sich sein Dienstgeber, der Wiener Magistrat, näher mit Internet-Technologien auseinanderzusetzen begann und er in die Tiefen der CGI-Programmierung eintauchte. Seither verwendet er - neben clientseitigem Javascript - Perl für so ziemlich alles, was es zu programmieren gibt; auf C weicht er nur mehr aus, wenn es gar nicht anders geht - und dann häufig auch nur, um XS-Module für Perl schreiben. Wenn er nicht gerade in Perl-Sourcen herumstöbert, schwingt er das gerne das Tanzbein oder den Tennisschläger.

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.

Marcus Holland-Moritz

Geboren 1977 in Thüringen und seit Ende der 80er Jahre dem Programmieren verfallen (damals noch auf einem C16). Die Jahrtausendwende hat ihn in den Raum Stuttgart verschlagen, wo er seitdem Software für Patientenmonitore in C und C++ entwickelt. Perl hat er dabei eher zufällig (und zuerst widerwillig) entdeckt; mittlerweile schreibt er jedoch die meisten Tools, die er zum Arbeiten braucht, in Perl. Er ist Autor mehrerer CPAN-Module und auch bei den perl5-porters aktiv.

Moritz 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.

Perl::AI - RoboCamel lebt...

Künstliche Intelligenz ist in aller Munde, aber nicht nur in der Robotik mit dem Honda-Roboter ist künstliche Intelligenz gefragt, sondern in vielen anderen Gebieten auch. So habe ich an einem Projekt mitgearbeitet, das Bereiche im menschlichen Genom finden soll, die für Medikamente wichtig sind. Dabei kann man sich das Genom selbst anschauen - was bei der Größe des menschlichen Genoms zu lange dauern würde - oder man lässt Computer entscheiden.

Perl bietet sehr vielfältige Möglichkeiten künstliche Intelligenz zu nutzen und zu erzeugen. Ich möchte an zwei Beispielen zeigen, wie 'künstliche Intelligenz' mit Perl geschaffen wird.

deklarative Programmierung

Die deklarative Programmierung bietet einige Vorteile gegenüber der Prozeduralen Programmierung. In der deklarativen Programmierung werden funktionale Beziehungen zu Prädikatenlogischen Ausdrücken erweitert und eignen sich so hervorragend zur Wissensgewinnung. Perl bietet mit den Modulen `AI::Prolog` und `Language::Prolog::Yaswi` zwei einfache Möglichkeiten mit Prolog zu arbeiten. Prolog gehört zu den logischen Programmiersprachen und wird sehr häufig in sogenannten Expertensystemen verwendet.

Support Vector Machines

Support Vector Machines (SVM) werden zur Klassifizierung verwendet. Man kann damit zum Beispiel vorhersagen, ob ein neues Auto wahrscheinlich erfolgreich sein wird, oder ob es eher ein Flop wird. Wie bei allen Vorhersagen, gibt es keine 100%-ige Sicherheit bei den Aussagen. Ich habe Support Vector Machines in der Bio-Informatik zur Klassifizierung von Genom-Daten verwendet. Da das in dem Bereich noch nicht allzu häufig gemacht wurde und unser Problem noch überhaupt nicht betrachtet wurde, wurden nach der Klassifizierung die Ergebnisse im Labor getestet und erstaunlich gute Werte hervorgebracht.

Wie gut diese Vorhersage ist, hängt von den Trainingsdaten ab. Eine SVM muss mit bekannten Daten trainiert werden und dann kann man mit unbekannten Daten die Vorhersage treffen. So wie man in der Wettervorhersage bessere Ergebnisse erzielt je mehr man das Wetter der letzten Jahre anschaut und die Trainingsdaten daraus bezieht.

Perl und künstliche Intelligenz

Perl wird bisher eher selten mit künstlicher Intelligenz in Verbindung gebracht. Es ist vielleicht nicht die Sprache der Wahl, wenn es um große KI-Anwendungen geht, aber Perl kann hier einmal mehr beweisen, dass es mehr kann als nur reine Parsing-Aufgaben.

Perl kann zum Beispiel wieder - wie so oft - als 'Glue' fungieren und verschiedene Ansätze miteinander verbinden.

Die `AI::*`-Module decken einen sehr breiten Bereich ab. Und dazu gibt es noch einige weitere Module, die für KI-Anwendungen benutzt werden können und nicht im `AI::*`-Namensraum auftauchen.

Oftmals wird gar keine große Anwendung benötigt, sondern es reicht etwas Kleines. Hier kann die künstliche Intelligenz den Perl-Code deutlich verringern und den Einsatz von 'großen' Anwendungen vermeiden.

Die Module können auch die Zusammenarbeit von verschiedenen Programmierern erleichtern, da die Programmierer unabhängig von einander arbeiten können und die Ergebnisse können durch die Module leicht zusammengefügt werden.

AI::Prolog

Prolog ('Programming in Logic', auch frankophon 'Prologue') ist eine Programmiersprache, die maßgeblich von Alain Colmerauer, einem französischen Informatiker, Anfang der 1970er Jahre entwickelt wurde und zur Familie der deklarativen Programmierung zählt. Sie ist eine Vertreterin der logischen Programmiersprachen.

Grundprinzip

Prolog-Programme bestehen aus einer Datenbasis, die Fakten und Regeln umfasst. Der Benutzer formuliert Anfragen an diese Datenbasis. Der Prolog-Interpreter benutzt die Fakten und Regeln, um systematisch eine Antwort zu finden. Ein positives Resultat bedeutet, dass die Antwort logisch ableitbar ist. Ein negatives Resultat bedeutet nur, dass aufgrund der Datenbasis keine Antwort gegeben werden kann. Dies hängt eng mit der Closed world assumption zusammen (siehe unten). Das typische erste Programm in Prolog ist nicht wie in prozeduralen Programmiersprachen ein Hallo-Welt-Beispiel, sondern eine Datenbasis mit Stammbauminformationen.

Weitere Techniken

Entscheidend für die Prolog-Programmierung sind die Techniken der Rekursion und die Nutzung von Listen. Ist die Rekursion in den meisten Programmiersprachen nur eine zusätzliche Variante zur Iteration, ist sie bei der Prolog-Programmierung die einzige Möglichkeit, 'Schleifen' zu produzieren.

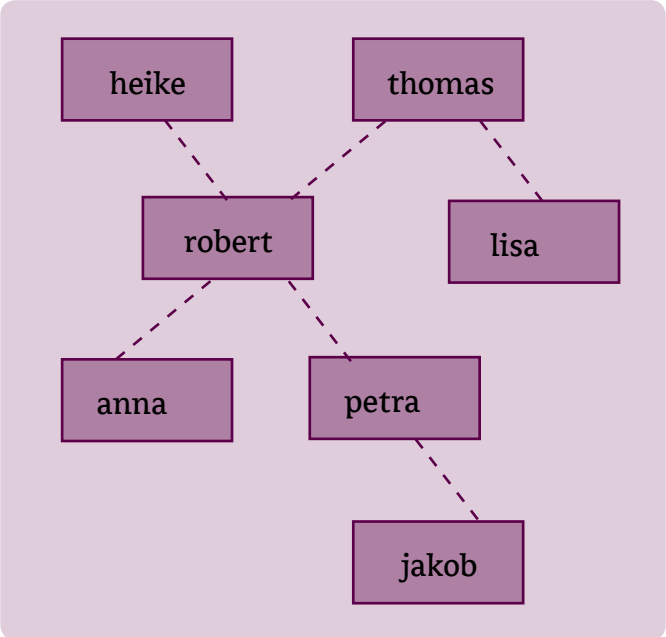


Abbildung 1: Familienstammbaum

Anwendungsgebiete

In den 1980er Jahren spielte die Sprache eine wichtige Rolle beim Bau von Expertensystemen. Die Sprache wird auch heute noch in den Bereichen Computerlinguistik und Künstliche Intelligenz verwendet. Außerdem gibt es einige kommerzielle Anwendungen im Bereich des Systemmanagements, bei denen asynchrone Ereignisse (Events) mit Hilfe von Prolog oder darauf basierenden proprietären Erweiterungen verarbeitet werden. Ein Beispiel hierzu ist das Produkt 'Tivoli Enterprise Console' von IBM, das auf BIM-Prolog basiert.

Als Beispiel nehmen wir einen kleinen Familienstammbaum, der aussieht wie in Abbildung 1 dargestellt.

Das ist die Datenbasis unserer Anwendung. Unsere Anwendung soll Verwandtschaftsverhältnisse klären bzw. die Frage nach bestimmten Verwandten klären. In unserem Fall will Jakob die Namen seiner Urgroßeltern erfahren.

Man kann diesen Stammbaum so lesen: (Ein Elter ist ein Elternteil)

heike ist ein elter von robert
robert ist ein elter von petra

Um die oben genannte Frage beantworten zu können, müssen wir aber noch gewisse Regeln festlegen. So muss ein Großelter der fragenden Person ein Elternteil der Eltern der fragenden Person sein, und so weiter.

Wie es in Code gegossen aussieht, ist in Listing 1 zu sehen.

Als erstes wird in einem String die Datenbasis für das Programm festgelegt. Natürlich kann man die Daten auch aus einer Datei einlesen. Danach wird der 'Prolog-Interpreter' erzeugt und die Datenbasis wird an den Interpreter übergeben. Danach können dann verschiedene Abfragen (Queries) an die Datenbasis gestellt werden.

Das Modul wurde von Curtis 'Ovid' Poe geschrieben und befindet sich noch in der Entwicklung. Trotzdem ist es für kleinere Sachen sehr gut einsetzbar. `AI::Prolog` ist eine PurePerl-Lösung für logische Aufgaben. Mit diesem Module kann man Prolog in Perl-Programme einbetten und die Vorteile der logischen Programmierung nutzen.

Für eine schnellere Variante bietet sich `Language::Prolog::Yaswi` an, das den SWI-Prolog-Interpreter verwendet.

```
#!/usr/bin/perl
use AI::Prolog;

my $database = <<'END_PROLOG';
elter( heike,robert ).
elter( thomas,robert ).
elter( thomas,lisa ).
elter( robert,anna ).
elter( robert,petra ).
elter( petra,jakob ).
grosselter( X,Y ) :- elter( X,Z ), elter( Z,Y ).
urgrosselter(X,Y) :- grosselter(X,Z), elter(Z,Y).
END_PROLOG

my $prolog = AI::Prolog->new($database);
$prolog->query("urgrosselter(heike,jakob).");
my @results;
while (my $result = $prolog->results) {
  push(@results,$result->[1]);
}
print join(" und ",@results)," sind die Urgrosseltern von Jakob.\n";
```

Listing 1

Der Nachteil bei beiden Modulen ist, dass man die Prolog-Syntax kennen muss und selbst auf das Backtracking achten muss. Es empfiehlt sich also, vor der Herausgabe an Kunden die Eigenschaften des Programms genau zu testen.

Die Verwendung von Prolog kann Programme um etliche Zeilen an Code verkürzen und die Wartbarkeit vereinfachen. Wenn die Urgroßmutter bestimmt werden soll, dann muss nicht die ganze Logik des Programms geändert werden, sondern es wird einfach die Datenbasis erweitert und die Abfrage angepasst. Das bedeutet im Idealfall genau eine Zeile Perl-Code, während bei einer reinen Perl-Lösung etliche Zeilen Code geändert werden müssten. Dies ist eine große Fehlerwahrscheinlichkeit.

Algorithm::SVM

Das Prinzip der Klassifikation bzw. Mustererkennung mit Support-Vector-Maschinen (SVM) beruht auf dem Finden einer optimal trennenden Hyperebene in einem hochdimensionalen Merkmalsraum - typischerweise mit wesentlich höherer Dimension als der ursprüngliche Merkmalsraum (letzterer wird auch als Eingaberaum bezeichnet) und handelt um eine überwachte Lernmethode aus dem Bereich Maschinelles Lernen. Um sich das besser vorstellen zu können kann man in einem zweidimensionalen Koordinatensystem ein paar Punkte denken, die von einer Linie voneinander getrennt werden (Abbildung 2).

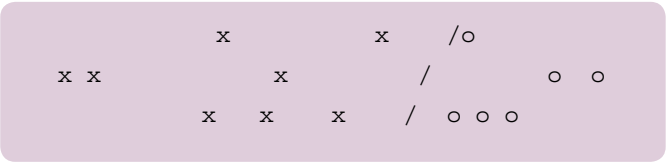


Abbildung 2: Punkte im Koordinatensystem

Eine Linie trennt die Punkte 'x' von den Punkten 'o'

Die Methodik geht auf V. Vapnik zurück, der 1974 ein Paper über die Prinzipien der Hyperebenentrennung veröffentlichte.

Mit dem Modul `Algorithm::SVM` lassen sich Support-Vector-Machines trainieren. Es wird aber das Programm `libSVM` benötigt. Das Modul eignet sich meiner Meinung nach nur, wenn die (Trainings- oder Test-)Daten im Laufe des Programms aus irgendwelchen Berechnungen heraus gewonnen werden. Wenn man die Daten schon vorher hat, sollte man die SVM doch besser 'per Hand' trainieren.

Das wohl bekannteste Einsatzgebiet von SVMs außerhalb von Forscherkreisen dürfte wohl die Spam-Bekämpfung sein. Da SVMs für Klassifizierungen benutzt werden - Einteilung in die eine oder ander Klasse - kann man damit gut Mails klassifizieren. Hier gibt es auch nur zwei Klassen: gehört die Mail zur bösen Klasse (Spam) oder zu der guten Klasse (gewünschte Mail)?

Ich zeige aber kein Beispiel aus der Spam-Bekämpfung, sondern wie man ein neu entwickeltes Fahrzeug klassifizieren

kann. In dem Beispiel werden Pseudodaten von bisher entwickelten Autos als Trainingsdaten verwendet. So wird der VW Käfer unter die Lupe genommen, genauso wie einen Lamborghini oder der Smart.

In dem Beispiel werden folgende 5 Einflussfaktoren (Dimensionen) betrachtet:

- Wartung (vhigh, high, med, low)
- Anzahl der Türen (2, 3, 4, 5more)
- Personen (2, 4, more)
- lug_boot (small, med, big)
- Sicherheit (low, med, high)

Die Dimensionen sollen immer einen Wert zwischen 0 und 1 haben. Aus diesem Grund werden die Werte in Noten zwischen 0 und 1 übersetzt.

Wir haben folgende Beispieldaten (Auszug):

```
vhigh,2,2,small,low
vhigh,2,2,small,med
high,3,4,med,low
low,2,4,big,low
low,2,4,big,med
...
```

Da die SVM nur mit Zahlen umgehen kann, müssen die Angaben dann im Programm übersetzt werden (Listing 2).

Als erstes legen wir mit dem Hash `\%mapping` fest, durch welche Werte die Ausprägung der Dimension repräsentiert wird (z.B. dass 'small' in der Dimension 4 durch '0.33' in der SVM repräsentiert wird. Danach bilden wir die Datensätze für das Training und speichern diese in einem Array.

Nachdem wir die Datensätze haben, erzeugen wir ein neues `Algorithm::SVM`-Objekt und legen gleich ein paar Optionen für die SVM fest. Das Modul bietet aber auch die jeweiligen Methoden, um die Werte nachträglich zu setzen.

Danach wird die SVM trainiert und das Modell wird für eine spätere Verwendung in einer Datei gespeichert. Damit wollen wir vermeiden, den langen Vorgang des Trainierens immer wiederholen zu müssen.

```
#!/usr/bin/perl

use strict;
use warnings;
use FindBin ();
use Algorithm::SVM;
use Algorithm::SVM::DataSet;

my $home            = $FindBin::Bin.'/';
my $trainingsfile    = $home.'car_data_main.txt';
my $predictionsfile  = $home.'car_data.txt';
my $model           = $home.'new-sample.model';

my %mapping = (
    0 => {vhigh => 1,    high    => 1,
          med   => -1,   low     => -1,   },
    1 => {vhigh => 1,    high    => 0.66,
          med   => 0.33, low     => 0,    },
    2 => {2     => 0.4,  3       => 0.6 ,
          4     => 0.8,  '5more' => 1,    },
    3 => {2     => 0.33, 4       => 0.66,
          more  => 0.99,                  },
    4 => {small => 0.33, med     => 0.66,
          big   => 0.99,                  },
    5 => {low   => 0.33, med     => 0.66,
          high  => 0.99,                  },
  );

train_svm($model,$trainingsfile,%mapping);
predict($model,$predictionsfile,%mapping);
```

Listing 2 (Fortsetzung auf nächster Seite)

```
#---------------------------------------------------#
#                 Subroutines                       #
#---------------------------------------------------#

sub train_svm{
  my ($model,$trainingsfile,%mapping) = @_;
  my @tsets = ();

  #generate datasets for training
  open(my $fh_training,"<",$trainingsfile) or die $!;
  while(my $line = <$fh_training>){
    next if($line =~ /^\s*$/);
    my @info    = split(/,/,$line);
    my $dataset = Algorithm::SVM::DataSet->new(
                      Label => $mapping{0}->{$info[0]},
                      Data  => [
                         map{$mapping{$_}->{$info[$_]}}(1..5)
                              ]);
    push(@tsets,$dataset);
  }
  close $fh_training;

  my $svm = Algorithm::SVM->new(Type   => 'C-SVC',
                                Kernel => 'radial',
                                );

  $svm->train(@tsets);
  # Perform cross validation on the training set.
  my $accuracy = $svm->validate(5);

  # Save the model to a file.
  $svm->save($model);
}# train_svm

sub predict{
  my ($model,$predictionsfile,%mapping) = @_;

  my $svm = Algorithm::SVM->new();
  $svm->load($model);

  open(my $fh_predict,"<",$predictionsfile) or die $!;
  while(my $line = <$fh_predict>){
    chomp $line;
    next if($line =~ /^\s*$/);
    my @info    = split(/,/,$line);
    my $dataset = Algorithm::SVM::DataSet->new(
                      Label => $mapping{0}->{$info[0]},
                      Data  => [
                         map{$mapping{$_}->{$info[$_]}}(1..5)
                                ]);
    my $result = $svm->predict($dataset);
    print sprintf("%-15s",$info[-1]),": ",$result,"\n";
  }
  close $fh_predict;
}# predict
```

Listing 2 (Fortsetzung)

So eine Datei für ein Modell sieht dann ungefähr so aus:

```
svm_type c_svc
kernel_type rbf
gamma 0.25
nr_class 2
total_sv 1728
rho 0
label 1 -1
nr_sv 864 864
SV
1 0:1 1:0.4 2:0.33 3:0.33 4:0.33
1 0:1 1:0.4 2:0.33 3:0.33 4:0.66
1 0:1 1:0.4 2:0.33 3:0.33 4:0.99
...
```

Um unsere neuen Fahrzeuge zu klassifizieren, erzeugen wir die Probe-Datensätze. Die Ausprägungen der Dimensionen müssen durch die gleichen Werte repräsentiert werden wie beim Training, da sonst die Ergebnisse verfälscht werden.

Danach wird die SVM gestartet und das Ergebnis ausgewertet. Wir geben einfach das neue Automodell an und geben dann an, in welche Klasse es gehört. Die -1 heißt, dass sich das Fahrzeug wahrscheinlich nicht gut verkaufen lässt, die 1 bedeutet das Gegenteil. Die Ausgabe sieht dann so aus:

```
Leopard 3    : -1
Kaefer       : 1
Golf 17      : -1
Test1        : -1
Test2        : -1
test3        : 1
Ford Geier   : 1
Opel Marina  : -1
Auto 3       : 1
Fahrzeug 6   : -1
"Perl" 16V   : 1
Testwagen 9  : -1
```

Perl wird sich also wahrscheinlich gut verkaufen, während der Leopard 3 wohl nicht so viele Abnehmer finden wird.

Dieses Modul eignet sich sehr gut für kleinere Anwendungen und wenn man die Daten im Laufe des Perl-Programms erhält. Wenn die Daten aber schon vorliegen, dann sollte man die SVM 'per Hand' trainieren.

Ein weiteres Einsatzgebiet für dieses Modul ist das Anwenden der SVM. Wenn also das Modell schon vorliegt und für unbekannte Daten angewendet werden soll. Wenn eine Integration der SVM in eine Pipeline gewünscht ist, dann ist `Algorithm::SVM` eine sehr gute Wahl.

Die zwei gezeigten Beispiele veranschaulichen, dass man mit Perl auch "künstliche Intelligenz" erschaffen kann. Für ausgedehnte Berechnungen sollten aber dann doch andere Sprachen bzw. Bibliotheken verwendet werden, die schneller sind.

Renée Bäcker

SQL-Statements erzeugen

Bei der Entwicklung von Perl-Programmen hört man immer wieder, dass es zwei Möglichkeiten gibt, Datenbankabfragen zu machen: Erstens die SQL-Statements in den Code schreiben und mit DBI direkt zu arbeiten und zweitens Objekt-Relationale-Mapper wie DBIx::Class oder Class::DBI zu verwenden. Beides hat Vor- und Nachteile. Während die Arbeit mit DBI schnell und "schlank" ist, ist DBIx::Class relativ langsam, da sehr viele Module geladen werden müssen und die Objekte groß werden. Dafür kann man mit DBIx::Class sehr gut durch die Tabellen "navigieren". Man muss sich im Perl-Code nicht mehr darum kümmern, wie die einzelnen Tabellen miteinander verknüpft sind. Bei DBI ohne zusätzliche Module muss man SQL-Statements schreiben.

Mit SQL::Abstract steht ein Mittelweg zur Verfügung. Das Modul generiert aus Perl-Datentypen ein SQL-Statement, das man dann an DBI übergeben kann (siehe Listing 1).

LIMIT, ORDER BY, ...

SQL::Abstract kann nur einfache Statements erzeugen und die Möglichkeiten für die Angabe von LIMIT oder ORDER BY fehlen komplett. Dafür muss ein weiteres Modul installiert werden: SQL::Abstract::Limit (siehe Listing2).

Das Modul erzeugt immer ein ORDER BY sobald vier Parameter übergeben werden. Ein LIMIT ohne ORDER BY ist nicht möglich - im Gegensatz zu einer Abfrage, die man selbst schreibt. Weiterhin muss noch ein limit_dialect angegeben werden, der für jedes Datenbanksystem die passende LIMIT-Syntax kennt. Wer mit MySQL arbeitet, muss 'LimitXY' wählen (siehe Listing 3).

```
use SQL::Abstract;

my $sql   = SQL::Abstract->new;
my $table = 'testtabelle';
my $cols  = [qw/col1 col2 col3/];
my $where = {
   ID => 129,
};

my ($stmt,@binds) = $sql->select(
    $table, $cols, $where
);

print $stmt, "\n   ", join(" :: ", @binds),"\n";
__END__
C:\>abstract.pl
SELECT col1, col2, col3 FROM testtabelle WHERE ( ID = ? )
   129
```

Listing 1

```
my $sql   = SQL::Abstract::Limit->new( limit_dialect => 'LimitXY' );
my ($stmt,@binds) = $sql->select(
    $table, $cols, $where, ['col1']
);
__END__
C:\>abstract_2.pl
SELECT col1, col2, col3 FROM testtabelle WHERE ( ID = ? ) ORDER BY col1
129
```

Listing 2

```
my $sql   = SQL::Abstract::Limit->new( limit_dialect => 'LimitXY' );
my ($stmt,@binds) = $sql->select(
    $table, $cols, $where, ['col1'], 10
);
__END__
C:\>abstract_2.pl
SELECT col1, col2, col3 FROM testtabelle WHERE ( ID = ? ) ORDER BY col1 LIMIT 10
129
```

Listing 3

```
my $sql   = SQL::Abstract->new;
my $where = {
   ID => 129,
   col5 => 3,
};

my ($stmt,@binds) = $sql->select(
    'test', 'col2', $where
);

print $stmt,"\n";
__END__
C:\>abstract_2.pl
SELECT col2 FROM test WHERE ( ID = ? AND col5 = ? )
```

Listing 4

```
my $sql   = SQL::Abstract->new;
my $where = [
   ID => 129,
   col5 => 3,
];

my ($stmt,@binds) = $sql->select(
    'test', 'col2', $where
);

print $stmt,"\n";
__END__
C:\>abstract_2.pl
SELECT col2 FROM test WHERE ( ( ID = ? ) OR ( col5 = ? ) )
```

Listing 5

Um mit `LIMIT` arbeiten zu können, muss man neben dem vierten Parameter für das `ORDER BY` noch einen fünften Parameter angeben, der den Offset angibt.

Man darf aber nicht vergessen, dass komplexe Abfragen auch mit `DBIx::Class` und `SQL::Abstract` schnell komplex werden. Vor allem am Anfang treten auch immer wieder Probleme damit auf, was denn jetzt Hashreferenzen für eine Auswirkung haben und was Arrayreferenzen bedeuten. Gewisse Sachen, wie zum Beispiel Aufrufe von SQL-Funktionen, müssen auch mit `SQL::Abstract` direkt geschrieben werden, so dass auch weiterhin Kenntnisse über die Funktionen und Möglichkeiten des Datenbanksystems nötig sind.

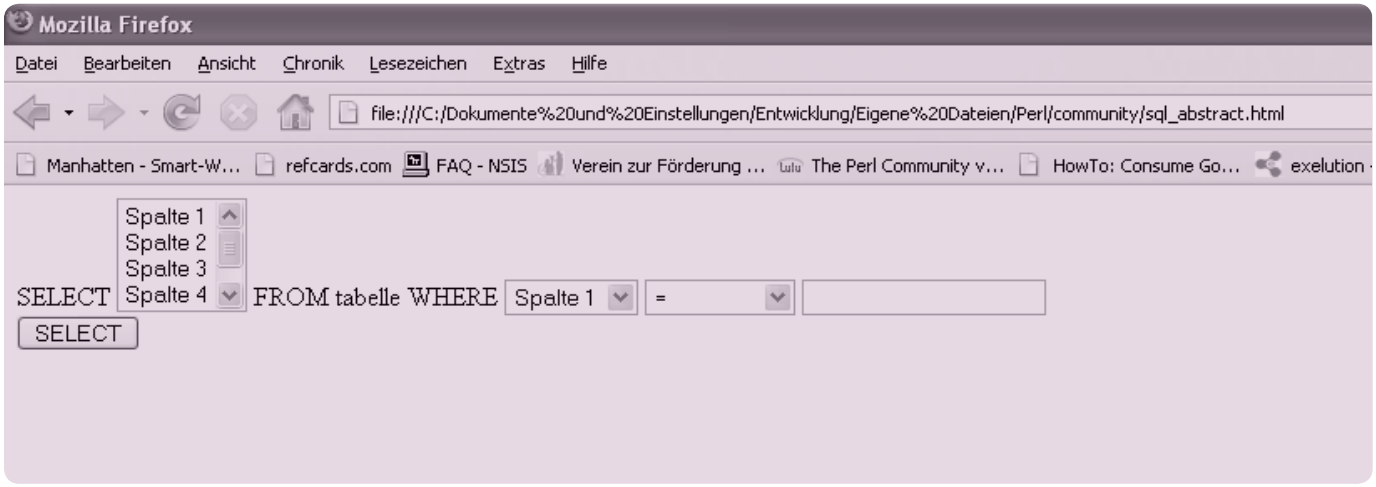
Mit Hashreferenzen werden UND-Verknüpfungen in der `WHERE`-Bedingung dargestellt (Listing 4), mit Arrayreferenzen ODER-Verknüpfungen (Listing 5).

Diese Datenstruktur kann beliebig tief strukturiert sein, damit alle möglichen `WHERE`-Bedingungen dargestellt werden können. Ein `WHERE ID = ? AND (col5 = ? OR col7 = ?)` wird zum Beispiel mit

```
my $where = {
    -and =>
      [
        ID => { '!=' => 129 },
        [
          col5 => 3,
          col7 => 19,
        ],
      ]
 };
```

Listing 6

erreicht. Gerade wenn man noch nicht viel mit Modulen wie `DBIx::Class` gearbeitet hat, ist die Formulierung der Bedingung sehr kompliziert und bedeutet häufig "Rumprobierei".



Wird ein einfacher String oder eine Zahl für die Spalte angegeben, wird Standardmäßig das '=' als Vergleichsoperator genommen. Soll ein anderer Vergleich genommen werden, muss dies in einer Hashreferenz angegeben werden (Listing 7).

```
my $where = [
   ID => { '!=' => 129 },
   col5 => 3,
];
```

Listing 7

Mit dem Beispiel sieht die Bedingung so aus:

```
WHERE ( ( ID != ? ) OR ( col5 = ? ) )
```

Es gibt Fälle, in denen Funktionen des Datenbanksystems genutzt werden sollen. Die kennt `SQL::Abstract` natürlich nicht alle. Um diese dennoch nutzen zu können, muss eine Referenz auf einen String geliefert werden. Das Beispiel in Listing 8 verwendet die `REGEXP`-Funktion von MySQL (`WHERE ( REGEXP(test) )`).

```
my $where = {
   '' => \'REGEXP(test)',
};
```

Listing 8

Das Modul bietet keine großen Vorteile, wenn die zu selektierenden Spalten schon von vornherein feststehen. Aber es bietet dann einen großen Vorteil, wenn die Spalten erst im Programmablauf festgelegt werden (können). Dann muss man nicht mit eigenen `map`- oder `grep`-Lösungen arbeiten, wenn man die ?-Notation von DBI verwenden will, da das dann alles von `SQL::Abstract` übernommen wird.

Ein Manko bei dem Modul ist, dass Tabellen- und Spaltennamen nicht gequotet werden. So treten bei Spaltennamen mit Leerzeichen Probleme auf.

Ein Beispiel für den Einsatz von `SQL::Abstract(::Limit)?` ist zum Beispiel die Suche in der Datenbank über ein Webformular. Ein User kann sich dabei aussuchen, welche Spalten selektiert werden sollen - und mit welchen Bedingungen. In Abbildung 1 ist ein Screenshot von so einem Formular zu sehen. Listing 9 zeigt das Skript, das dieses Formular auswertet und dann die Ergebnisse als einfache Tabelle ausgibt.

Renée Bäcker

```
#!/usr/bin/perl

use strict;
use warnings;
use CGI;
use CGI::Carp qw(fatalsToBrowser);
use DBI;

use lib qw(./lib);
use SQL::Abstract::Limit;

my $cgi = CGI->new;
print $cgi->header;

my %params = $cgi->Vars;

##
# der nachfolgende Code ist relevant für SQL::Abstract

my $sal   = SQL::Abstract::Limit->new( limit_dialect => 'LimitXY' );
my $where;

$where = {
     $params{wherecol} => { $params{whereop} => $params{whereval} },
} if $params{whereval};

my $limit = defined $params{limitval} ? $params{limitval} : undef;

my ($sql,@binds) = $sal->select(
  'testtabelle', [$cgi->param('columns')], $where, 'col1', $limit
);

# der relevante Code ist zu Ende
##

my $dbh = DBI->connect( "DBI:mysql:xxx:xxx","xxx","xxx" ) or die $DBI::errstr;
my $sth = $dbh->prepare( $sql ) or die $dbh->errstr;
$sth->execute( @binds )         or die $dbh->errstr;

print "<table>";
while( my @row = $sth->fetchrow_array ){
   print "<tr>",
         map{ "<td>$_</td>" }@row,
         "</tr>";
}
print "</table>";
```

Listing 9

Perls Schatztruhe... ... auf meinem eigenen Rechner!

Man kann CPAN getrost als "Schatztruhe" bezeichnen, denn dort findet man (fast) alles was man an Perl-Modulen braucht. PDF-Dateien erzeugen oder bearbeiten? Kein Problem. Auf CPAN findet man mit Sicherheit ein passendes Modul. Bildbearbeitung, Testautomatisierung? Auch dafür findet man unzählige Module auf CPAN - genau wie für zigtausend andere Fälle auch.

Zum jetzigen Zeitpunkt (30.09.2008 - 20:00) gibt es 14352 Distributionen (~ 50.000 Module) von 6877 Autoren. Und es kommen jeden Tag einige neue Module hinzu. Aber manchmal ist es so, dass man genau dann, wenn man es bräuchte, keinen Zugang zu CPAN hat oder das Netzwerk soll geschont werden. Was wenn jetzt ein Modul dringend benötigt wird?

In diesem Fall kann man nur hoffen, dass man ein eigenes CPAN dabei hat.

Ricardo Signes hat das Modul `CPAN::Mini` geschrieben, mit dem sich sehr leicht ein lokales CPAN erstellen lässt. Mit dem Modul werden immer die neuesten Versionen - wobei Entwickler-Versionen davon ausgeschlossen sind - von Modulen geladen und ältere Versionen gelöscht. So wird der Speicherbedarf reduziert. Während auf CPAN rund 5 Gigabyte an Daten versammelt sind, passt ein lokales Mini-CPAN auf eine ganz normale CD.

Mit dem Modul `CPAN::Mini` wird ein Programm mit dem Namen `minicpan` mitgeliefert, mit dem sich mit einem einfachen Befehl ein lokales Mini-CPAN erstellen lässt:

```
~reneeb>minicpan -r http://cpan.mirror -l ./localcpan
```

Jetzt wurden in `./localcpan` die Module abgelegt. In Zukunft wird mit dem gleichen Befehl das lokale CPAN aktualisiert. Neue Module werden heruntergeladen und alte werden gelöscht. Somit wird auch der Programmlauf wesentlich kürzer.

Sollen in Zukunft die Module über das `CPAN.pm`-Modul unter Verwendung des lokalen CPANs installiert werden, kann man `CPAN.pm` so konfigurieren, dass das eigene Mini-CPAN in die URL-Liste aufgenommen wird.

```
cpan> o conf urllist unshift
                    file:///homes/reneeb/localcpan/
```

Aber das `commit` nicht vergessen, wenn diese Änderung dauerhaft sein soll.

Mit Filtern kann man auch bestimmte Distributionen aus dem Mini-CPAN ausschließen. Das kann zum Beispiel gewollt sein, wenn man selbst CPAN-Autor ist und man seine eigenen Distributionen nicht im Mini-CPAN haben möchte. Oder wenn Linux-User keine Win32-Distributionen herunterladen wollen.

Es gibt zwei unterschiedliche Filter, die man bei `CPAN::Mini` einsetzen kann:

Mit den `path_filters` wird überprüft, ob der Pfad einer Distribution gewissen Bedingungen entspricht und mit `module_filters` werden die Distributionsnamen überprüft (siehe Listing 1).

Sowohl `path_filters` als auch `module_filters` können drei verschiedene Werte haben. Zum einen eine Code-Referenz, die "wahr" zurückliefern muss für den Fall, dass die Distribution herausgefiltert werden muss und "falsch", wenn die Distribution gespiegelt werden soll. Der zweite mögliche

```
use CPAN::Mini;

CPAN::Mini->update_mirror(
  remote => "http://cpan.mirrors.comintern.su",
  local  => "/usr/share/mirrors/cpan",
  path_filters   => [
    qr/RENEEB/,
    sub{ $_[0] =~ /RENEEB/ }, # äquivalent
  ],
  module_filters => [
    qr/Win32/,
    sub{ $_[0] =~ /Win32/ }, # äquivalent
  ],
);
```

Listing 1

Wert ist ein Regexp-Objekt. Die letzte Möglichkeit für die Filter ist eine Arrayreferenz, in denen die verschiedenen Filter gespeichert sind. Diese Filter werden mit "OR" verknüpft. Die einzelnen Filter können Code-Referenzen oder Regexp-Objekte sein.

Ab jetzt habe ich immer einen USB-Stick mit den aktuellen Modulen dabei...

CPAN::Mini::Webserver

Wer kennt schon alle Module, weiß welches Modul für eine Aufgabe eingesetzt werden kann oder nicht? Wer kennt die Dokumentation zu all diesen Modulen? Wohl keiner. Auf CPAN kann man aber nicht suchen wenn es keine Internetverbindung gibt. Aus diesem Grund hat Leon Brocard das Modul `CPAN::Mini::Webserver` geschrieben. Wenn das MiniCPAN vorhanden ist und `CPAN::Mini::Webserver` installiert ist, kann man das Skript `minicpan_webserver` starten und im Browser die URL `http://localhost:2963/` aufrufen. Und schon hat man eine lokale CPAN-Suche zur Verfügung.

Renée Bäcker

Logging für Perl-Programme

Es gibt viele Module, die versprechen, das Logging in Perl-Programmen zu vereinfachen. Das ausgereifteste Modul dürfte jedoch `Log::Log4perl` sein, eine Portierung des beliebten Java-Loggers log4j.

Allerdings muss man sich als Log4perl-Einsteiger erst einmal mit den Möglichkeiten des Moduls auseinandersetzen und sich anschauen wie die Konfiguration gemacht wird.

Ich weiß, was Du letzten Sommer getan hast

Logging ist für die Fälle gedacht, in denen man sich anschauen muss, was eine Anwendung getan hat. In welchem Zustand war die Anwendung als der Fehler auftrat? Was hatte der User eingegeben?

Mit dem Logging kann auch die Performanz einer Anwendung gemessen werden. Wo ist der Flaschenhals der Anwendung?

Da Log4perl mehrere Stufen von Logmeldungen kennt, eignet es sich zu Analysen auf dem Live-System und Debugging im Entwicklungssystem. Man kann sehr genau einstellen, was wie geloggt werden soll.

Im Gegensatz zu den meisten anderen Modulen vom CPAN wird die Konfiguration von Log4perl schon nativ in extra Dateien vorgenommen. So muss bei Änderungen in Ausgabe-Device oder Debug-Leveln nicht das Skript angepasst werden, sondern nur die Konfigurationsdatei.

Als weiteren Grund für die Verwendung von Log4perl nennt dessen Autor Mike Schilli, dass Log4perl das letzte Logging-Modul sein soll, dass man installiert, da es alle Anwendungsfälle abdeckt.

Log4perl in Aktion

Log4perl eignet sich sowohl für die Verwendung in kleinen Skripten als auch in großen Anwendungen. Für die Verwendung in kleinen Skripten eignet sich die `easy`-Funktionalität des Moduls. Beim Laden von Log4perl einfach mit dem `:easy`-Tag alles importieren was notwendig ist.

```
#!/usr/bin/perl

use Log::Log4perl qw(:easy);
```

Dann stehen einige Methoden und einige Variablen zur Verfügung:

```
#!/usr/bin/perl

use Log::Log4perl qw(:easy);

Log::Log4perl->easy_init( $INFO );
INFO "Testausgabe";
DEBUG "Debugausgabe";
```

Am Anfang des Skripts wird Log4perl initialisiert. Mit `$INFO` wird dem Modul mitgeteilt, dass das niedrigste Debug-Level, das ausgegeben werden soll, *INFO* ist. Alle niedrigen Levels werden nicht ausgegeben. Wenn das Skript ausgeführt wird, erscheint in der Kommandozeile nur die Info-Ausgabe, da das Level von `DEBUG` niedriger ist, als das von `INFO`.

Wenn man das Logging in Modulen nutzen möchte, sollte man die Initialisierung weglassen. So kann man dann einfach im Hauptprogramm festlegen ob überhaupt geloggt werden soll und wenn ja, was mitgeschrieben werden soll.

```
package MyModule;

use Log::Log4perl qw(:easy);

sub test {
  DEBUG "ich bin jetzt in ",__PACKAGE__,
        "::test\n";
}
```

Das Laden von `Log::Log4perl` schaltet das Logging noch nicht ein.

So wird nichts ausgegeben, da das Skript das Logging nicht einschaltet:

```
#!/usr/bin/perl

use strict;
use warnings;
use MyModule;

MyModule->test();
```

In diesem Fall aber wird die Meldung ausgegeben, da das Mindestlevel auf *DEBUG* gesetzt wird.

```
#!/usr/bin/perl

use strict;
use warnings;
use MyModule;
use Log::Log4perl qw(:easy);

Log::Log4perl->easy_init( $DEBUG );
MyModule->test();
```

Diese Beispiele zeigen, wie einfach das Logging bei Perl-Programmen ist. Und damit man beim Go-Live nicht versehentlich zu viel mitloggt, kann man in den Systemen eine entsprechende Umgebungsvariable einbauen und im Skript noch eine "Weiche" einbauen:

```
use MyModule;
use Log::Log4perl qw(:easy);

my $level = $DEBUG;
if( $ENV{APP_SYSTEM} eq 'live' ){
   $level = $ERROR;
}

Log::Log4perl->easy_init( $level );
MyModule->test();
```

Jetzt wird auf allen Systemen ab *DEBUG* mitgeloggt, auf dem Live-System aber nur noch die Fehlermeldungen.

Bisher wurde nur gezeigt, wie man mit `:easy` das Logging initialisieren kann. Der Standard sieht etwas anders aus. Dazu wird die Methode `get_logger` importiert.

```
use Log::Log4perl qw(get_logger);

Log::Log4perl->init( 'config.conf' );

my $logger = get_logger();
$logger->debug( 'Test' );
```

`get_logger` ohne Parameter liefert den Logger für das aktuelle `package`, mit einem Leerstring den Root-Logger und mit einem String (z.B. `get_logger('Bar.Tender')`) den Logger für die entsprechende Kategorie (siehe Abschnitt *Categories*).

Datenfluss steuern

Viele Informationen sind meistens nützlich, aber zu viele Informationen erschweren wiederum die Fehlersuche. Aus diesem Grund muss der Datenfluss mehr gesteuert werden. `Log::Log4perl` bietet drei "Mechanismen", mit denen der Datenfluss eingegrenzt oder erweitert werden kann. Die folgenden Abschnitte stellen diese Mechanismen vor.

Levels

Wie schon oben angedeutet, gibt es bei `Log::Log4perl` verschiedene Levels, mit denen geloggt werden kann. Insgesamt werden sechs dieser Level definiert. Die folgende Auflistung ist mit aufsteigender Priorität sortiert.

- trace
- debug
- info
- warn
- error
- fatal

Die eingesetzten Levels sollten auch mit Bedacht gewählt werden, damit kein großes "Rauschen" in den höheren Loglevels entsteht. Für alle diese Loglevels existieren Subroutinen, die die Nachrichten auf das entsprechende Level setzen - also `$logger->debug( 'Nachricht' )`. Es gibt auch Skalare, die den Loglevel repräsentieren. Diese sind in den Beispielen am Anfang schon gezeigt worden. Für `trace` gibt es `$TRACE`, für `debug` gibt es `$DEBUG` und so weiter.

Categories

Für jede Kategorie gibt es einen eigenen Logger, den man extra konfigurieren kann. Eine Kategorie beschreibt einen Bereich der Anwendung. Was das genau bedeutet, wird in Beispielen besser ersichtlich.

Durch die Kategorien ist es auch möglich, Logging nur für einen Teil der Anwendung durchzuführen, während in anderen Teilen nichts eingesetzt wird.

Die Bezeichnung der Kategorien erinnert an die `package`-Namen, sind jedoch hierarchisch aufgebaut. Das heißt, wir haben das `package` "Foo::Magazin", dann existiert eine Kategorie "Foo" und eine Kategorie "Foo.Magazin".

In der Konfiguration von Log4perl können dann die einzelnen Kategorien separat behandelt werden:

```
log4perl.category.Foo = ERROR, ...
...
log4perl.category.Foo.Magazin = DEBUG, ...
log4perl.additivity.Foo.Magazin = 0
```

Jetzt werden für die Kategorie "Foo" alle Meldungen mit der Priorität von *ERROR* und höher ausgegeben und für die Kategorie "Foo.Magazin" alle `DEBUG`-Nachrichten. Damit die `DEBUG`-Meldungen von "Foo.Magazin" nicht auch mit dem Appender von "Foo" angezeigt werden, ist es notwendig, mit dem `additivity`-Attribut zu arbeiten.

Appenders

Die *Appender* legen fest, wohin geloggt wird. Wie viele andere Logging-Frameworks unterstützt auch `Log::Log4perl` Ausgabemedien wie `STDERR`, eine Datei oder die Datenbank. Aber das Modul geht noch viel weiter und bietet *Appender* für RRDs oder Sockets. Weiterhin können die Ausgabemedien des Moduls `Log::Dispatch` verwendet werden und man kann noch sehr einfach eigene Module schreiben.

Standardmäßig wird nach `STDERR` geloggt. Möchte man in eine Datei loggen, so muss man folgendes tun (siehe Listing 1).

Was wird hier konfiguriert? In der ersten Zeile der Konfiguration wird für den Logger das minimale Loglevel festgelegt. In diesem Fall ist das "ERROR". Zusätzlich wird hier noch der Name für einen *Appender* festgelegt, der frei wählbar ist. Hier wählen wir einfach mal "MyLogFile". Danach wird der Appender selbst noch konfiguriert.

Da wir in ein Logfile loggen wollen, wählen wir als Modul, das das handeln soll `Log::Log4perl::Appender::File`, das schon mit Log4perl mitgeliefert wird. Die Datei, in die geloggt werden soll, heißt "test.log".

Man kann *Appender* auch dynamisch zum Logging hinzufügen. Dazu wird ein Appender-Objekt erzeugt und über `add_appender` dem Logger hinzugefügt (siehe Listing 2).

```
use Log::Log4perl qw(get_logger);

# Konfiguration für das Logging
my $config = q~
    log4perl.logger                       = ERROR, MyLogFile
    log4perl.appender.MyLogFile           = Log::Log4perl::Appender::File
    log4perl.appender.MyLogFile.filename  = test.log
    log4perl.appender.MyLogFile.layout    = SimpleLayout
~;

# mit Konfiguration initialisieren
Log::Log4perl->init( \$config );

my $logger = get_logger();
$logger->error("Eine Fehlermeldung");
$logger->warn("Eine Warnung");
```

Listing 1

```
my $database_appender = Log::Log4perl::Appender->new(
  'Log::Log4perl::Appender::DBI',
  datasource => 'DBI:mysql:databse:host',
  username   => 'myuser',
  password   => 'mypassword',
  sql        => qq~INSERT INTO log (message) values (?)~,
);

$logger->add_appender( $database_appender );
$logger->error( 'Eine Fehlermeldung' );
```

Listing 2

Layouts

Mit den Layouts wird das Aussehen der Meldungen beeinflusst. Standardmäßig wird das `StandardLayout` verwendet, das einfach den Level und die Meldung ausgibt. Möchte man ein eigenes Layout erzeugen, so kann man mit `PatternLayout` arbeiten. Bei Log4perl sind einige Formate vorbestimmt.

- %c Die Kategorie in der das Logging-Ereignis ausgelöst wurde
- %C Vollqualifizierter Package-/Klassenname des aufrufenden Codes
- %d Aktuelles Datum im yyyy/MM/dd hh:mm:ss Format
- %F Datei, in der das Logging-Ereignis aufgetreten ist
- %H Hostname
- %l aufrufende Methode + Datei + Zeile
- %L Zeilennummer innerhalb der Datei, in der die Log-Meldung ausgelöst wurde
- %m Log-Meldung
- %M Funktion bzw. Methode, in der die Log-Meldung ausgelöst wurde.
- %n Zeilenumbruch
- %p Priorität des Logging-Ereignisses
- %P Prozess-ID des aktuellen Prozesses
- %r Millisekunden seit Programmstart
- %x Elemente des NDC (Nested Diagnostic Context) stacks
- %X{key} Der Eintrag zum Schlüssel 'key' im MDC (Mapped Diagnostic Context)
- %% Das Prozentzeichen (literal)

So könnte dann in der Konfigurationsdatei folgendes stehen (siehe Listing 3).

Hier werden die Log-Meldungen auf dem Bildschirm ausgegeben und statt dem Standard-Layout wird mit eigenen Mustern ein neues Layout generiert. Jetzt wird der Level der Meldung vor dem Klassennamen ausgegeben. Danach kommen noch Informationen über die Stelle, an der die Log-Meldung ausgelöst wurde, und die Meldung an sich.

Diese Formate sind auch leicht noch zu ergänzen:

```
log4perl.PatternLayout.cspec.U =
            sub { return "UID $<" }
```

Damit kann dann noch das Format `%U` verwendet werden.

Filter

Mit Filtern kann man die Ausgabe von Meldungen einschränken. Allerdings sollte das eher die Ausnahme sein, da Filter sehr rechenintensiv sind. Mögliche filter sind boolsche Werte, String- oder Level-Vergleiche. Mit Log4perl werden schon ein paar Filter-Module mitgeliefert. Die Filter werden in bekannter Art und Weise konfiguriert (siehe Listing 4).

Mit dieser Konfiguration werden alle Meldungen ausgegeben, die den String "foo meldung" beinhalten.

Die Konfiguration auslagern

Gerade für größere Anwendungen lohnt es sich, die Konfiguration des Loggers in Dateien auszulagern. Die Konfigurationsstrings aus den Beispielen können eins zu eins in eine Konfigurationsdatei gespeichert werden. Bei der Initialisierung wird dann nicht die Referenz auf den String übergeben, sondern der Dateiname

```
Log::Log4perl->init( 'myapp.conf' );
```

Das Auslagern der Konfiguration in eine extra Datei hat den Vorteil, dass bei Änderungen von Logleveln oder Appendern nicht der Code der Anwendung angefasst werden muss, sondern nur die Konfigurationsdatei.

Eine einfache Konfigurationsdatei sieht so aus (siehe Listing 5).

```
log4perl.logger.MyTest = ERROR, MyScreen
log4perl.appender.MyScreen = Log::Log4perl::Appender::Screen
log4perl.appender.MyScreen.layout = Log::Log4perl::Layout::PatternLayout
log4perl.appender.MyScreen.layout.ConversionPattern =[%p][%C] (%l)- %m
```

Listing 3

```
log4perl.filter.M1 = Log::Log4perl::Filter::StringMatch
log4perl.filter.M1.StringToMatch = foo meldung
log4perl.filter.M1.AcceptOnMatch = true
```

Listing 4

```
log4perl.logger.MyTest = ERROR, MyScreen
log4perl.appender.MyScreen = Log::Log4perl::Appender::Screen
log4perl.appender.MyScreen.layout = PatternLayout
log4perl.appender.MyScreen.layout.ConversionPattern =[%p][%C] (%l)- %m

log4perl.logger.MyTest.Test = DEBUG, Logfile
log4perl.appender.Logfile = Log::Log4perl::Appender::File
log4perl.appender.Logfile.filename = test.log
log4perl.appender.Logfile.layout = SimpleLayout

log4perl.logger.TestCase = INFO, Logfile
```

Listing 5

Mit dieser Konfiguration wird für drei Kategorien das Logging konfiguriert: Für die Kategorie "MyTest" werden die Meldungen auf dem Bildschirm ausgegeben - aber nur wenn diese ein Level von *ERROR* oder höher haben. Die Bildschirmausgabe ist auch noch etwas angepasst, so dass man recht schnell den Klassennamen und den aufrufenden Code erkennt.

Für die Kategorie "MyTest.Test" werden die Meldungen mit einem Level von *DEBUG* und höher in eine Logdatei mit dem Namen "test.log" geschrieben. Dort wird das einfache Layout verwendet. Da "MyTest.Test" auch den Appender von "MyTest" verwendet, werden diese Meldungen auch auf dem Bildschirm ausgegeben.

Die dritte Kategorie wurde angelegt, um die Wiederverwendbarkeit von Appendern zu zeigen. Für alle Meldungen in der Kategorie "TestCase" mit einem Level von *INFO* und höher werden in die gleiche Datei geschrieben wie für die Kategorie "MyTest.Test".

Sollen die Meldungen von "TestCase" in eine andere Datei geschrieben werden, so muss nur ein neuer Appender eingetragen werden. Es ist also kein Anpassen von Perl-Code notwendig.

```
#!/usr/bin/perl

use DBIx::Log4perl;
use DBI;

DBIx::Log4perl->init( 'config.datei' );
my $dbh = DBI->connect(
      'DBI:mysql:db:host','user','passwd' )
    or die $DBI::errstr;

my $sth = $dbh->prepare(
     "SELECT * FROM tabelle WHERE ID = ?"
);
$sth->execute(8);
```

Listing 6

```
DEBUG - prepare(0.1):
       'SELECT * FROM tabelle WHERE ID = ?'
DEBUG - $execute(0.1) = [8];
```

Listing 7

Best Practices

Vermeide Skriptnamen in Log-Meldungen

Informationen wie Skriptnamen oder aufrufende Subroutinen sollten nicht fest in den Aufruf der Logging-Funktionen geschrieben werden. Vielmehr sollten dazu die Layouts angepasst werden (siehe Abschnitt *Layouts*). Wenn etwas direkt im Code steht, ist der Wartungsaufwand viel größer wenn Funktionen in Module ausgelagert werden oder Programmnamen sich ändern.

Logge viel

Speicherplatz sollte nicht die limitierende Größe sein, vielmehr sollte man dann "rotating Logfiles" verwendet werden. Es empfiehlt sich aber, auf dem Bildschirm nur die wichtigsten Einträge auszugeben. Also *fatal*e Meldungen direkt auf STDERR und alles andere in Logfiles. So hat man größere Fehler direkt im Blick und die genaueren Informationen in Dateien abrufbar.

mod_perl

Unter mod_perl sollte die Initialisierung des Loggings in den Startup-Handler der Anwendung. Derzeit gibt es nur eine `init`-Methode und ein erneuter Aufruf überschreibt die vorherige Initialisierung. Wenn man sich nicht ganz sicher ist, wo initialisiert wird, sollte man `init_once` verwenden.

Noch mehr Logging...

Mittlerweile gibt es auf CPAN etliche Module, die das Logging mit Log4perl in andere Bereiche bringt. So kann mit `DBIx::Log4perl` alles geloggt werden, was mit `DBI` zu tun hat. Im Grunde genommen ist es einfach nur ein Wrapper um `DBI`, erleichtert das Logging aber enorm (Siehe Listing 6).

Im Log taucht dann auf, was in Listing 7 dargestellt ist

Renée Bäcker

Spezialklassen

"besondere" Klassen

Im Objektorientierten Perl gibt es zwei Spezialklassen, von denen man auf jeden Fall gehört haben sollte. Je nach Anwendungsgebiet, werden sie mehr oder weniger häufig verwendet.

UNIVERSAL

Jede Klasse in Perl erbt von `UNIVERSAL`. Diese Spezialklasse stellt schon einige Methoden zur Verfügung, die nützlich sind:

isa

Alle ge"bless"ten Referenzen können mit `isa` gefragt werden, ob sie von einem bestimmten Typ sind. So kann man typenabhängig Aktionen ausführen - wie es auch in Listing 1 dargestellt ist.

Mit `ref` bekommt man nur zurückgeliefert, von welcher Klasse das Objekt ein Objekt ist. Man kann allerdings nicht überprüfen, ob eine Klasse irgendwo im Vererbungsbaum der Klasse ist, zu der das Objekt gehört (siehe Listing 2).

Hier sieht man, dass bei `obj2` auch bei der Überprüfung von 'Pod::Simple' ein 'yes' ausgibt. Das zeigt, dass `Pod::Simple::HTML` eine Subklasse von `Pod::Simple` ist.

Kommt es also vor, dass in einer Anwendung eine Subklasse durch eine andere Subklasse ersetzt wird, dann ist es immer sinnvoll, mit `isa` zu arbeiten.

```
my $c = bless( {}, 'CGI' );
if( $c->isa( 'CGI' ) ){
   print "\$c ist vom Typ 'CGI'\n";
}

unless( $c->isa( 'Foo' ) ){
   print "\$c ist *nicht* vom Typ 'Foo'\n";
}
```

Listing 1

can

Eine sehr nützliche Methode ist `can`. Damit kann man überprüfen, ob ein Objekt oder eine Klasse eine Methode kennt. Als Parameter wird der Methodenname übergeben. Kann das Objekt bzw. die Klasse die Methode, so wird eine Referenz auf diese gesuchte Methode zurückgeliefert.

`can` eignet sich z.B. sehr gut in Systemen mit Plugins, bei denen man nicht weiß, ob sie eine erforderliche Methode auch tatsächlich implementiert hat.

```
#!/usr/bin/perl

{
   package MyTest2;
}

MyTest2->testmethode();
```

```
#!/usr/bin/perl

use Pod::Simple;
use Pod::Simple::HTML;

my $obj1 = Pod::Simple->new;
my $obj2 = Pod::Simple::HTML->new;

print "obj1:\n",
      "  ref: ", ref( $obj1 ), "\n";
for ( 'Pod::Simple', 'Pod::Simple::HTML' ) {
  print " isa $_ ?",
          $obj1->isa( $_ ) ? "yes\n" : "no";
}

print "\n\nobj2:\n",
      "  ref: ", ref( $obj2 ), "\n";
for ( 'Pod::Simple', 'Pod::Simple::HTML' ) {
  print " isa $_ ? ",
          $obj2->isa( $_ ) ? "yes\n" : "no";
}
```

Listing 2

Ruft man eine nicht-existierende Methode auf, so bricht Perl das Programm ab und meldet `"Can't locate object method "testmethode" via package "MyTest2" at ..."`.

Benutzt man `can`, kann man so einen ungewollten Abbruch vermeiden:

```
#!/usr/bin/perl

{
  package MyTest;
  sub testmethode { print "hallo Welt" }
}

{
   package MyTest2;
}

my $sub = MyTest->can( 'testmethode' );
if( $sub ){
   $sub->();
}

my $sub2 = MyTest2->can( 'testmethode' );
if( $sub2 ){
   $sub2->();
}
```

Ausgabe:

```
C:\>can.pl
hallo Welt
```

Man muss allerdings darauf achten, dass man das Objekt oder den Klassennamen explizit übergeben muss, wie das bei Subroutinen-Referenzen üblich ist (siehe auch Abschnitt "Referenzen").

VERSION

Diese Klassenmethode liefert die Version einer Klasse (die in der Variablen `$VERSION` definiert ist). Die Methode wird sehr häufig verwendet, wenn man in einem Perl-Einzeiler überprüfen möchte, welche Version installiert ist:

```
perl -MModul -e 'print Modul->VERSION'
```

Der Methode kann man eine Versionsnummer als Parameter übergeben. Dann wird überprüft, ob die Klasse mindestens die übergebene Version hat. Ist dies nicht der Fall, bricht Perl das Programm ab. Soll das also im Produktivcode eingesetzt werden, sollte ein Block-`eval` verwendet werden:

```
eval {
   require Modul;
   Modul->VERSION( 2.0 );
   1;
} or warn "Modul hat nicht die
      benötigte Version: ",$@;
```

SUPER

Mit der Pseudoklasse `SUPER` kann auf Methoden der Superklassen zugegriffen werden. Das ist dann wichtig, wenn die Subklasse eine Methode der Superklasse überschreibt, man dann aber die Methode der Superklasse verwenden will. Dies kommt häufiger bei der Methode `new` vor, da die Superklasse häufig die Methode `new` zur Verfügung stellt, diese dann aber von der Subklasse überschrieben wird. Das Listing 3 zeigt ein Beispiel, in dem genau das gemacht wird.

```
#!/usr/bin/perl

{
   package Superklasse;

   sub new {
      my ($class) = @_;
      my $self = bless {}, $class;
      $self->farbe( 'blau' );
      return $self;
   }

   sub farbe {
      my ($obj, $val) = @_;
      $obj->{farbe} = $val if @_ == 2;
      return $obj->{farbe};
   }
}

{
   package Subklasse;

   our @ISA = qw(Superklasse);

   sub new {
      my ($class) = @_;
      my $self = $class->SUPER::new();
      return $self;
   }
}

my $obj = Subklasse->new;
print $obj->farbe;
```

Listing 3

Hier hat die Klasse `Superklasse` einen Konstruktor, der noch einen Standardwert für die Farbe setzt. Die Subklasse erbt von `Superklasse` und überschreibt den Konstruktor der Superklasse. Würde hier nicht der Konstruktor der Superklasse aufgerufen werden, würde kein Standardwert für die Farbe gespeichert werden.

Der Konstruktor der Subklasse sieht auch etwas anders aus als gewohnt: Es wird kein `bless` verwendet. Vielmehr wird das Objekt, das vom Konstruktor der Superklasse geliefert wird, weiterverwendet. Hier besteht gibt es auch eine Gefahr, der man sich bewusst sein sollte: Da in Perl die Objekte nicht geschützt werden und jeder alle Attribute verändern kann, kann es passieren, dass man Attribute der Superklasse überschreibt. Ein Objekt ist ja nichts anderes als eine besondere Hashreferenz. Und Hashes können einen Schlüssel immer nur genau einmal haben. Verwendet die Subklasse jetzt `$self->{farbe}`, wird der Wert der Superklasse überschrieben.

Renée Bäcker

TPF-TICKER

Google Summer of Code

Nachdem die Perl Foundation beim letztjährigen "Google Summer of Code" nicht teilgenommen hat, gab es in diesem Jahr wieder einige Perl-Projekte. Die Perl Foundation wurde dabei hauptsächlich von Eric Wilhelm vertreten.

Insgesamt wurden 5 Projekte für Perl abgeschlossen:

* Flesh out the Perl 6 Test Suite
  Student: Adrian Kreher
  Mentor: Moritz Lenz

* wxCPANPLUS
  Student: Samuel Tyler
  Mentor: Herbert Breunung and Jos Boumans

* Native Call Interface Signatures and Stubs Generation for Parrot
  Student: Kevin Tew
  Mentor: Jerry Gay

* Incremental Tricolor Garbage Collector
  Student: Andrew Whitworth
  Mentor: chromatic

* Math::GSL
  Student: Thierry Moisan
  Mentor: Jonathan Leto

Typeglobs - Teil II

*In der vorigen Ausgabe wurden Eigenschaften von Typeglobs und das **Wie** beim Zuweisen und Auslesen ihrer Werte behandelt. Heute geht es um das **Was**: Was kann man mit Typeglobs alles anstellen? Im folgenden werden einige Beispiele aus der Praxis beschrieben.*

Bearbeiten von Symboltabellen

Das Bearbeiten von Symboltabellen kann vor allem für Test- und Debuggingzwecke hilfreich sein. Da Symboltabellen als Hashes implementiert sind, kann man die mit Perl hierfür mitgelieferten Funktionen und Operatoren wie mit jedem anderen Hash verwenden. Dadurch wird es möglich, Symboltabellen zu durchsuchen und ihre Einträge weiterzuverarbeiten. Von dieser Möglichkeit macht unter anderem auch der Perl-Debugger in seinen `V` und `X` Befehlen Gebrauch.

Das Beispiel in Listing 1 zeigt, wie Symboltabellen rekursiv durchsucht werden. Dabei macht es sich das in Abschnitt Stashes und Packages (siehe $foo Nr. 7) beschriebene Konzept *verschachtelter* Packages zunutze (die Zeilennummern dienen nur zu Referenzzwecken und sind kein Bestandteil des Codes).

Dieser Code listet vom angegebenen Stash (oder von `%main::`, falls keine Angabe erfolgt ist) alle Typeglobs und die in ihnen belegten Slots in der Notation `*package::name{slot}` auf. Falls das zweite Argument auf `true` gesetzt ist, werden auch alle Stashes unterhalb des angegebenen rekursiv durchlaufen. An diesem Code ist einiges bemerkenswert.

```
 1  use warnings;
 2  use strict;
 3  my @slots = qw[SCALAR ARRAY HASH CODE IO FORMAT];
 4  my $stash = shift || 'main::';
 5  $stash = 'main::' if $stash eq '::';
 6  $stash .= '::' unless $stash =~ /::$/;
 7  my $recflg = shift;
 8  show_globs($stash, $recflg);
 9
10  sub show_globs {
11    my ($stash, $recflg, $index) = @_;
12    $index ||= 0;
13    no strict 'refs';
14    foreach my $glob (values %{$stash}) {
15      my $name = *{$glob}{NAME};
16      next if $stash eq 'main::' && $name eq 'main::';
17      my $fullname = $stash . $name;
18      foreach my $slot (@slots) {
19        my $text = ' ' x $index . '*' . $fullname . "{$slot}\n";
20        if ($slot eq 'SCALAR') {
21          print $text if defined ${$glob};
22        }
23        else {
24          print $text if defined *{$glob}{$slot};
25        }
26      }
27      show_globs($fullname, 1, $index+1) if $name =~ /::$/ && $recflg;
28    }
29  }
```

Listing 1

Zunächst werden in Zeile 3 die bekannten Wertetypen von Perl, die in Slots von Typeglobs vorkommen können, aufgelistet. Wenn man nicht an allen Wertetypen interessiert ist, kann man die nicht Benötigten auch weglassen.

Weiters fällt die `no strict 'refs'` Anweisung innerhalb der Funktion in Zeile 13 auf; sie ist notwendig, da in Zeile 14 eine symbolische Referenz zum Ansprechen des jeweiligen Stashes (`%{$stash}`) verwendet wird. Dasselbe Problem stellt sich auch, wenn auf diese Art Typeglobs angesprochen werden sollen:

```
my $name = 'ENV';
# dasselbe wie print %ENV
print %{*{$name}{HASH}};
```

Dieser Code liefert mit `use strict 'refs'` den Laufzeitfehler

```
Can't use string ("ENV") as a symbol
       ref while "strict refs" in use
```

zurück. Wie weiter unten noch gezeigt wird, kann es im Umgang mit Symboltabellen und Stashes häufig notwendig werden, `strict 'refs'` temporär auszuschalten.

Erwähnenswert ist auch Zeile 16 - sie verhindert einen Endloslauf des Skripts. Die Implementierung von Stashes, besonders der Algorithmus zur deren Verschachtelung sieht vor, dass jeder Stash einem Elternstash entstammt. Wem entstammt dann aber `%main::`? Nun, er stammt von sich selbst ab. Anders gesagt, enthält `%main::` ein Element, dessen Name `main` lautet und das wiederum auf `%main::` zeigt [1]. Dadurch entsteht eine (gewollte) Zirkularität, die - in diesem Beispiel - in Zeile 16 unterbrochen wird. Wenn man also in Skripts Stashes auf diese Weise durchläuft und auch `%main::` verarbeiten will, muss man dessen besondere Eigenschaft (*selbstreferenzierender Stash*) in der gezeigten oder in einer ähnlichen Weise berücksichtigen.

Die Prüfung beider Variable ist notwendig, da ansonsten Stashes wie z.B. `MyPackage::main::` unberücksichtigt bleiben würden.

Schließlich soll noch auf die unterschiedliche Behandlung von Skalarslots und anderen Slots in Zeile 20 ff. hingewiesen werden. Wie bereits erwähnt, ist der Skalarslot eines Typeglobs *immer* belegt (auch wenn der Skalarwert nur `undef` ist),. Daher muss die im Slot abgelegte Referenz dereferenziert werden, um feststellen zu können, ob tatsächlich ein Wert vorhanden ist. Bei allen anderen Wertetypen genügt das Testen ihrer Slots.

Auf die Ausgabe der einzelnen *Werte* der Variablen wurde hier verzichtet, da dies im Falle von Codewerten und Handles ohnehin nicht möglich ist und die Ausgabe bei Array- und Hashwerten in Abhängigkeit von deren Elementanzahl sehr umfangreich werden kann. Auch ist nicht gesagt, dass jeder Wert nur darstellbare Zeichen enthält, so dass diese u.U. vor der Ausgabe noch entsprechend bearbeitet werden müssten. Dies sei dem Leser bei Interesse als Aufgabe überlassen.

Da übrigens auch die Namen einzelner Perl-Variable teilweise mit nicht darstellenbaren Zeichen abgespeichert werden (so wird z.B. das `^U` der Variable `${^UNICODE}` mit seinem tatsächlichen ASCII-Wert 0x15 (octal 025) abgespeichert), erhält man beim Starten des obigen Skripts scheinbar nicht immer vollständige Namen (z.B. nur `NICODE`). Ob und wie solche Namen dargestellt werden, hängt von den Fähigkeiten des jeweiligen Ausgabegerätes ab. Guter Code stellt daher vor der Ausgabe von Stashnamen sicher, dass diese nur darstellbare Zeichen enthalten.

Ein weiteres Beispiel (siehe Listing 2), das das Löschen von Symboltabellen zeigt, ist (in etwas gekürzter Form) dem Modul `Symbol.pm` entnommen.

```
 1  sub delete_package {
 2    my $pkg = shift;
 3    $pkg = "main::$pkg" unless $pkg =~ /^main::/;
 4    $pkg .= '::' unless $pkg =~ /::$/;
 5
 6    my ($stem, $leaf) = $pkg =~ /(.*::)(\w+::)$/;
 7    my $stem_symtab = *{$stem}{HASH};
 8    return unless exists $stem_symtab->{$leaf};
 9
10    my $leaf_symtab = *{$stem_symtab->{$leaf}}{HASH};
11    foreach my $name (keys %$leaf_symtab) {
12      undef *{$pkg . $name};
13    }
14    %$leaf_symtab = ();
15    delete $stem_symtab->{$leaf};
16  }
```

Listing 2

Dieses Beispiel zeigt das Ansprechen von Stashes über Hashreferenzen, die direkt den jeweiligen Slots der Typeglobs entnommen werden (Zeilen 7 und 10). Der Code verdient eine nähere Betrachtung:

```
my $stem_symtab = *{$stem}{HASH};
```

Hier wird der Hashslot der Stashes, dessen Name in `$stem` abgelegt ist, geholt; das Ergebnis in `$stem_symtab` ist eine Hashreferenz. In Zeile 8 wird mittels Dereferenzierung geprüft, ob das angegebene Element (d.h., das in diesem Stash enthaltene Package) existiert:

```
return unless exists $stem_symtab->{$leaf};
```

Falls nicht, wird an dieser Stelle abgebrochen (was nicht existiert, braucht auch nicht gelöscht zu werden). Ansonsten wird nun der Stash, auf den dieses Element zeigt, geholt:

```
my $leaf_symtab =
   *{$stem_symtab->{$leaf}}{HASH};
```

Schreibt man das ohne die Zwischenvariable `$stem_symtab`, so ergibt sich:

```
my $leaf_symtab =
   *{*{$stem}{HASH}->{$leaf}}{HASH};
```

Das lässt sich - von innen nach außen - wie folgt auflösen:

- `*{$stem}` liefert den Typeglob des Stashes, dessen Name in `$stem` steht.
- `*{$stem}{HASH}` spricht den Hash-Slot dieses Typeglobs an, das ergibt daher eine Hashreferenz.
- `*{$stem}{HASH}->{$leaf}` spricht durch Dereferenzierung das Hashelement an, dessen Name in `$leaf` steht. Der *Wert* dieses Elements ist wiederum ein Typeglob, der mit
- `*{*{$stem}{HASH}->{$leaf}}` angesprochen wird, und somit wird mit
- `*{*{$stem}{HASH}->{$leaf}}{HASH}` dessen Hashslot adressiert, wodurch man wiederum eine Hashreferenz - diesmal auf den gesuchten Stash - erhält. In der Schleife ab Zeile 11 wird dieser Hash dann durch erneutes Dereferenzieren und Auslesen der Elementnamen durchlaufen.

Setzt man für `$stem` den Wert `"main::"` und für `$leaf` den Wert `"Alpha::"` ein, so ergibt sich für diese Zeilen Listing 3, woraus die Funktionsweise nun klar erkennbar sein sollte.
In Zeile 12 werden nun nacheinander alle Typeglobs in `$leaf_symtab` auf `undef` gesetzt und damit alle Werte mit diesen Namen gelöscht. In Zeile 14 wird der Stash noch einmal explizit geleert, bevor in Zeile 15 das Stashelement aus der übergeordneten Symboltabelle entfernt wird.

Die Funktion ist kurz, aber gründlich - sie löscht das angegebene Package *und alle darunter liegenden Packages* komplett. Daher ist bei der Anwendung Vorsicht geboten; auf die Probleme, die sich aus schreibendem (bzw. löschendem) Zugriff auf Stashes ergeben können, wurde schon im letzten Heft hingewiesen.

Zum Abschluss sei noch kurz die Funktion `gensym` des genannten Moduls erwähnt - sie liefert eine Referenz auf einen Typeglob zurück, die später z.B. anstelle eines Filehandles verwendet werden kann:

```
package Symbol;
my $genseq = 0;
my $genpkg = "Symbol::";

sub gensym () {
  my $name = "GEN" . $genseq++;
  my $ref = \*{$genpkg . $name};
  delete $$genpkg{$name};
  $ref;
}
```

Hier wird ein temporärer Name für den Typeglob (`Symbol::GEN<n>`) erzeugt und anschließend auf diesen Typeglob eine Referenz gelegt (wodurch ein Anlegen des Typeglobs *zur Laufzeit* erzwungen wird). Das entsprechende Element wird sodann wieder aus dem Stash gelöscht, der Typeglob bleibt aber durch die auf ihn zeigende Referenz erhalten und wird dadurch *anonym*. Die Referenz wird dann an den Aufrufer zurückgegeben.

Erwähnenswert ist noch das Konstrukt

```
*{$genpck . $name}
```

Wie bereits erwähnt, handelt es sich dabei um eine *symbolische Referenz*. Der Name des Typeglobs wird erst zur Laufzeit aus dem Inhalt der beiden Variablen gebildet, wobei die

```
 7    my $stem_symtab = *main::{HASH};
 8    return unless exists $stem_symtab->{'Alpha::'};
 9
10    my $leaf_symtab = *{$stem_symtab->{'Alpha::'}}{HASH};
```

Listing 3

erste den Package- oder Stashnamen und zweite den Symbolnamen enthält. Konstrukte dieser Art findet man häufig in Modulcode, wenn das Modul (typischerweise seine `import`-Funktion) die Symboltabelle durch *Exportieren* manipuliert. Im obigen Beispiel sind die zwei Doppelpunkte, die Package- und Symbolnamen trennen, schon in einer der Variablen enthalten, ansonsten trifft man oft auch auf Code wie diesen:

```
my @symbols = qw(alpha beta gamma);

# Package Namen des Aufrufers holen
my $pckg = caller;

# Symbol. Referenzen sind jetzt erlaubt
no strict 'refs';

foreach my $symbol (@symbols) {
  # Exportieren des Symbols
  *{$pckg.'::'.$symbol} = *$symbol;

  # Andere Schreibweise
  *{"${pckg}::$symbol"} = *$symbol;
}
```

Beide Schreibweisen bewirken dasselbe - durch das Duplizieren des Typeglobs wird das Symbol in den Namensraum des Aufrufers (also des Codes, der das Modul mit `use` oder `require` geladen hat, meistens `'main'`) exportiert. Die beiden Ausdrücke ergeben jeweils einen vollqualifizierten Symbolnamen.

Beachte, dass im zweiten Fall der Name der ersten Variable in geschwungene Klammern gesetzt werden muss, sonst würde der Parser den Ausdruck als

```
*{$pckg:: . $symbol}
```

(d.h., den Wert der Variable `$pckg::`, verbunden mit dem Wert von `$symbol`) verstehen, was wohl nicht das wäre, was man haben wollte.

Aliasing und Importing

Unter *Aliasing* versteht man, wie schon kurz erwähnt, das Ansprechen ein- und derselben Daten unter verschiedenen Namen. So kann nach Ausführen der Typeglob-Zuweisung

```
*beta = *alpha;
```

alles in `alpha` auch als `beta` angesprochen werden. Das ist noch nicht besonders interessant, aber sobald unterschiedliche Packages in Spiel kommen, wird die Bedeutung von Aliasing erkennbar:

```
package Alpha;

sub test {
  ...
}

*main::test = *test;
```

Nun kann auch im Hauptprogramm die Funktion mit `test()` (statt `Alpha::test()`) angesprochen werden.

Diese Methode hat allerdings einen nicht ganz unwesentlichen Nebeneffekt - nicht nur die Funktion `test` ist nun in beiden Packages unter diesem Namen sichtbar, sondern auch alle gleichnamigen Variablen und Handles. Wenn also im Package `Alpha` und im Hauptprogramm jeweils eine globale Variable `$test` verwendet wird, kann es mit diesem Ansatz Probleme geben, weil dann aus beiden Packages heraus unabsichtlich ein und dieselbe Variable angesprochen wird.

Daher existiert auch eine abgeschwächte Version des Aliasing, die man als *partielles* Aliasing bezeichnet. Die Idee dahinter ist nicht neu - anstatt den gesamten Typeglob zuzuweisen, wird nur der Slot des gewünschten Wertetyps - in der Regel der Codeslot - zugewiesen, also auf das obige Beispiel bezogen:

```
*main::test = *test{CODE};
```

oder die häufigere Schreibweise mit dem Referenzoperator :

```
*main::test = \&test;
```

Hier wird nur der Codeslot von `*main::test` auf den Wert von `*Alpha::test` gesetzt. Diese beiden Slots referenzieren also dieselbe Funktion. Die übrigen Slots beider Typeglobs, so sie verwendet werden, zeigen auf unterschiedliche Werte und daher sind nun `$main::test` und `$Alpha::test` wieder unterschiedliche Variable.

Und damit wird das Funktionsprinzip des bekannten `Exporter`-Moduls deutlich, dessen `import()` Methode mit allen Symbolnamen in `@EXPORT` und - sofern beim Aufruf angegeben, auch in `@EXPORT_OK` - partielles Aliasing durchführt. Der folgende Code ist aus `Exporter/Heavy.pm` entnommen (siehe Listing 4).

```
 1  foreach $sym (@imports) {
 2    (*{"${callpkg}::$sym"} = \&{"${pkg}::$sym"}, next)
 3      unless $sym =~ s/^(\W)//;
 4    $type = $1;
 5    *{"${callpkg}::$sym"} =
 6      $type eq '&' ? \&{"${pkg}::$sym"} :
 7      $type eq '$' ? \${"${pkg}::$sym"} :
 8      $type eq '@' ? \@{"${pkg}::$sym"} :
 9      $type eq '%' ? \%{"${pkg}::$sym"} :
10      $type eq '*' ?  *{"${pkg}::$sym"} :
11      do {require Carp; Carp::croak("Can't export symbol: $type$sym")};
12  }
```

Listing 4

```
 5    *{$callpkg.'::'.$sym} =
 6      $type eq '&' ? *{$pkg.'::'.$sym}{CODE}   :
 7      $type eq '$' ? *{$pkg.'::'.$sym}{SCALAR} :
 8      $type eq '@' ? *{$pkg.'::'.$sym}{ARRAY}  :
 9      $type eq '%' ? *{$pkg.'::'.$sym}{HASH}   :
10      $type eq '*' ? *{$pkg.'::'.$sym}         :
11      do {require Carp; Carp::croak("Can't export symbol: $type$sym")};
```

Listing 5

Ja, diese Zeilen implementieren die Kernfunktionalität von `Exporter`. Das Modul stellt noch einige weitergehende Möglichkeiten (Export-Tags, Exportieren in ein anderes als das aufrufende Modul, Versionsprüfungen, Symbole zum Exportieren sperren, u.a.m) zur Verfügung, aber letztlich basiert das alles auf obigem Code.

Was passiert hier? Nun, das Array `@imports` enthält die Liste aller Namen, die exportiert werden sollen (Funktions- oder Variablenamen). Funktionen können mit oder ohne führendem Sigil `&` angegeben werden. Die Variable `$callpkg` enthält den Namen des Packages, *in das*, `$pkg` das Package, *aus dem* exportiert werden soll. Durch die `foreach` Schleife wird `@imports` Element für Element abgearbeitet.

In den Zeilen 2 und 3 wird festgestellt, ob dem Namen ein Sigil vorangeht. Wie bereits erwähnt, muss dies bei Funktionsnamen nicht der Fall sein (und ist auch meistens nicht). Falls das Sigil fehlt, wird ein partielles Aliasing mit einer Funktionsreferenz durchgeführt. Damit werden solche Fälle wie

```
use MyModule qw(funcx funcy funcz);
```

in denen Funktionsnamen ohne `&` angegeben werden, abgedeckt.

Ansonsten wurde durch den regulären Ausdruck das Sigil ermittelt und mittels Substitution vom Symbolnamen entfernt, so dass nun ab Zeile 4 je nach Sigil ein partielles Aliasing mit der entsprechenden Referenz erfolgen kann.

Es sollte nun schon klar sein, dass man statt Code in Listing 4 auch den in Listing 5 hätte schreiben können. Hier hält es Perl wie so oft: TIMTOWTDI.

Doch halt! Warum wurde in Zeile 10 von Listing 5 das `{GLOB}` weggelassen? Der Leser möge versuchen, es herauszufinden, bevor er in [2] nachsieht.

Wird mittels des Exporters ein Skalar exportiert, so wird

```
*{"${callpkg}::$sym"} = \${"${pkg}::$sym"};
```

also z.B. für die nach `main` zu exportierende Variable `$hugo` des Moduls `My_Module`

```
*main::hugo = \$My_Module::hugo;
```

ausgeführt. Dabei wird der im Zielpackage neu angelegte Typeglob als **importiert** markiert. Diese Importierung ist typspezifisch, d.h., sie gilt (in diesem Beispiel) nur für den Skalarwert und nicht für die anderen möglichen Wertetypen. Natürlich können auch Arrays, Hashes oder Funktionen importiert werden, aber das muss für jeden Wertetyp explizit geschehen.

Stößt der Parser später (im Package `main`) auf den Ausdruck

```
$hugo
```

so findet er im Typeglob den Skalarwert als importiert gekennzeichnet und erlaubt daraufhin das Verwenden des unqualifizierten Namens. Daher kann man Variable und Funktionen, sobald sie durch den Exporter (oder anderwer-

tiges partielles Aliasing) derart markiert wurden, auch bei aktivem `strict 'vars'` ohne Packagenamen ansprechen.

Wird ein Programm mit `use strict` bzw. `use strict 'vars'` kompiliert, so muss *jede* globale Variable zuerst deklariert werden, erst dann ist ein Ansprechen ohne Packagenamen möglich. Für das Deklarieren gibt es die Compilerpragmas `vars` und `subs` [3]. Sieht man sich die `import()` Methode von `vars` an, so wird der darin enthaltene, auszugsweise wiedergegebene Code

```
$sym =
   "${callpack}::$sym" unless $sym =~ /::/;
*$sym = ( $ch eq "\$" ? \$$sym
        : $ch eq "\@" ? \@$sym
        : $ch eq "\%" ? \%$sym
        : $ch eq "\*" ? \*$sym
        : $ch eq "\&" ? \&$sym);
```

hoffentlich schon etwas vertrauter wirken. `$ch` enthält das Sigil, `$sym` den Namen des zu importierenden Symbols, der vor der Typeglob-Zuweisung ggf. noch mit dem Packagenamen des Aufrufers ergänzt wird, um einen voll qualifizierten Namen zu erhalten. Somit wird daher für das Importieren eines Skalars `$hugo` in das Package `main` durch

```
use vars '$hugo';
```

nur mehr

```
$sym = "main::hugo";
*{$sym} = \$$sym;
```

ausgeführt, was sich (hier) zu

```
*main::hugo = \$main::hugo;
```

reduzieren lässt. Und das entspricht vollkommen dem obigen Code des Exporters, nur mit dem Unterschied, dass hier Ziel- und Quellpackage *dasselbe* ist. Anders gesagt: das Modul `Exporter` und die Pragmas `vars` und `subs` basieren auf derselben Idee: sie importieren Symbole - nur sind die Quellpackages andere [4].

`subs` ist eine vereinfachte, auf das Importieren von Funktionen (die ohne Sigil angegeben werden) optimierte Version von `vars`. `subs` wird benötigt, um Funktionen *vorzudeklarieren*, um sie im Code vor der eigentlichen Definition verwenden zu können:

```
func;              # Funktionsaufruf
...
sub func {print "!"}
```

Hier weiß der Parser beim Auffinden des Ausdrucks `"func"` nicht, was gemeint ist - in der Symboltabelle gibt es noch keinen passenden Eintrag. Daher wird er den Ausdruck als nicht gequotetes Stringliteral interpretieren und sich nach Ausgabe der Meldung

```
Unquoted string "func" may clash
    with future reserved word
```

nicht weiter darum kümmern. Durch das Einsetzen von

```
use subs 'func';
```

am Programmbeginn hingegen wird durch partielles Aliasing der Ausdruck `func` als Funktionsname bekannt gemacht und der Parser wird ihn ab dann als Funktionsaufruf und nicht mehr als Stringliteral interpretieren.

Kompiliert man folgenden Code

```
use strict 'vars';
use vars '$alpha';
@alpha = (1,2,3);
```

so sollte nun auch die Bedeutung der erhaltenen Fehlermeldung

```
Variable "@alpha" is not imported
```

verständlicher werden - der Compiler ist in der letzten Zeile auf den Array-Namen `@alpha` gestoßen. Durch die Deklaration in der vorhergehenden Zeile wurde zwar bereits der entsprechende Typeglob angelegt, aber als importiert wurde nur der Skalar markiert, nicht das Array. Hier muss daher durch

```
use vars qw($alpha @alpha);
```

auch das Array vor dem ersten Ansprechen entsprechend markiert werden.

Fehlt hingegen im obigen Beispiel die zweite Zeile (d.h., es wird überhaupt nichts mit `vars` deklariert), so existiert auch noch kein Typeglob und der Compiler bricht mit den wohlbekannten Meldungen

```
Global symbol "$alpha" requires explicit
     package name
Global symbol "@alpha" requires explicit
     package name
```

seine Arbeit ab. Wer Verständnis für das Entstehen von Fehlermeldungen mitbringt und diese richtig interpretieren kann, ist klar im Vorteil!

Beachte, dass folgender Code (im Package `main`)

```
use strict 'vars';
$main::alpha = 2;
print $alpha;
```

nicht funktioniert und der Compiler seine Arbeit ebenfalls mit der `not imported` Meldung beendet. Durch die Angabe des vollqualifizierten Variablennamens in der zweiten Zeile wurde der Typeglob `*alpha` zwar angelegt, *aber der Skalar wird nicht als importiert markiert*. Der Effekt ist also derselbe wie vorher, als der Skalar importiert, aber das Array angesprochen wurde. Das Importieren funktioniert *nur* durch Zuweisung einer Referenz an den jeweiligen Typeglob [5].

Aus diesem Umstand erklärt sich auch die globale Auswirkung von `vars` und `subs` - sie manipulieren Typeglobs, und die sind (einschließlich eventueller Markierungen) immer im gesamten Programm sichtbar und nicht nur innerhalb bestimmter Blöcke. Und daher existiert auch kein `no vars` bzw. `no subs` - es macht keinen Sinn, einmal erfolgte Importe (deren Existenz ohnehin nur während des Kompilierens mit `strict 'vars'` von Bedeutung ist) wieder rückgängig zu machen.

Handles dürfen übrigens aufgrund fehlender Deklarationsmöglichkeit *immer* mit unqualifizierten Namen angesprochen werden, daher ist kein Importieren notwendig und es existieren auch keine derartigen Markierungen.

In der nächsten Ausgabe werden noch Wrapper und Handles ein Thema sein.

Ferry Bolhár-Nordenkampf

[1] Genauer gesagt, enthält der Hash-Slot des Typeglobs von `$main::{main::}` eine Referenz auf `main::`. Anders wäre es nicht möglich, wie im Beispiel gezeigt, mit

```
*main::{HASH}
```

auf die oberste Symboltabelle zuzugreifen. Da ein Typeglob immer in einem *Element* eines Stashes abgelegt ist, muss es natürlich auch für das Element `main::` einen Hash - eben `%main::` - geben. Somit könnte man den Typeglob `*main` auch als Hashelement

```
$main::{'main::'}
```

ansprechen. Puh!

(Beachte, dass hier der Keyname gequoted werden muss, um den Parser nicht durcheinander zu bringen, weil dieser ansonsten die Doppelpunkte "verschlucken" würde. Auch ein Parser hat irgendwann einmal genug!).

[2] Der Ausdruck

```
*{$pkg.'::'.$sym}{GLOB}
```

liefert eine *Referenz* auf den Typeglob. Das ist hier aber nicht verlangt, vielmehr soll der Exporter bei einer Angabe von `*symnam` tatsächlich ein *vollständiges* Aliasing durchführen, und das geschieht bekanntlich durch Zuweisung eines Typeglobs direkt an einen anderen. Daher fehlt im ursprünglichen Code auch der Referenzoperator vor dem `'*'`.

[3] Auf die seit Perl 5.6 bestehende Möglichkeit, die Benutzung globaler Variable mit `our` zu deklarieren, wird in `perlsub` eingegangen.

[4] Es gibt noch einen subtilen Unterschied zwischen `Exporter.pm` und `vars.pm`. Letzterer weist auch im Fall eines Typeglobs (z.B. `use vars '*hugo'`) eine *Referenz* an den Zieltypeglob zu. Der Exporter führt stattdessen ein *direktes Aliasing* durch. Aber obwohl hier intern unterschiedliche Prozesse ablaufen, ist das Ergebnis von

```
*alpha =  *beta;
*alpha = \*beta;
```

nach außen hin dasselbe - zumindest in den aktuellen Perl-Versionen. Gemeinsam hingegen ist beiden Modulen, dass ihr Code als `BEGIN`-Block, d.h. während der Compilerphase abläuft und durch Aliasing bzw. Importieren das Verhalten des Parsers beeinflusst.

[5] Außerdem muss tatsächlich etwas *importiert* werden, d.h., das gerade eingestellte Package muss *unterschiedlich* sein zu dem, in das importiert wird. Folgender Code würde daher nicht funktionieren:

```
use strict 'vars';

BEGIN {
  *main::hugo = \$main::hugo;
}
$hugo = 2;
```

Erst wenn in obigem Code während der Zuweisung ein anderes Packages eingestellt ist, wird das Importieren wie erwartet ablaufen:

```
use strict 'vars';

BEGIN {
  package Dummy;
  *main::hugo = \$main::hugo;
}
$hugo = 2;
```

Im Falle der Pragmas ist das kein Problem, da ja während der Abarbeitung ihres Codes ohnehin gerade die Packages `vars` bzw. `subs` eingestellt sind.

TPF-TICKER

News zu laufenden Grants

Alberto Simões veröffentlicht im Zwei-Wochen-Rhythmus Neuigkeiten zu den laufenden Grants.

- Mango - Christopher Laco
  Dieser Grant ist mittlerweile abgeschlossen.

- Porting PyYAML to Perl - Ingy döt Net
  YAML::XS ist vollständig und Ingy arbeitet jetzt an einer reinen Perl-Lösung. Eine neue Deadline wurde vereinbart: Weihnachten 2008

- Improving Smolder - Michael Peters
  Die Verbesserungen an Smolder sind abgeschlossen. Ein ausführlicher Bericht folgt noch.

- Perl on a Stick - Adam Kennedy
  Adam arbeitet weiterhin an "Perl on a Stick". Bei der OSCON gab es schon ein paar USB-Sticks zu kaufen. Weitere sollen bald folgen.

- SMOP - Daniel Ruoso
  Arbeiten haben noch nicht begonnen.

- Fixing Bugs in the Archive::Zip Perl Module - Alan Haggai Alavi
  Die Arbeiten wurden nach einer Erkrankung von Alavi wieder aufgenommen

- Make localtime() and gmtime() work past 2038 - Michael Schwern
  Arbeiten wurden begonnen, sonst keine Neuigkeiten

*•Test::Builder 2 - Michael Schwern
  Arbeiten wurden begonnen, sonst keine Neuigkeiten

XS – Perl mit C erweitern

Teil 1: Grundlagen

Du hast ein tolles Perl-Modul geschrieben, aber irgendwie ist es viel zu langsam? Es gibt da eine unheimlich nützliche Bibliothek, aber nur ein Python- und C-Interface? Dein Betriebssystem kennt diese Systemfunktion, die Du unbedingt brauchst, aber Du weisst nicht, wie Du sie aus Perl aufrufen sollst? Die Antwort auf alle diese Fragen heißt kurz: XS.

Die Abkürzung XS steht für eXtension Subroutines, und diese stellen das Bindeglied zwischen Perl und C dar. Mit ihrer Hilfe kann man systemnahe Funktionen für Perl verfügbar machen, existierende rechenintensive Algorithmen oftmals um ein Vielfaches beschleunigen oder einfach auf bestehende Bibliotheken zugreifen, die ein C-Interface besitzen.

Dies ist der erste Teil einer Serie von drei Artikeln zum diesem Thema. Er gibt eine kurze Einführung in C und geht anschließend auf die Grundlagen der XS-Programmierung ein.

C-Crashkurs

Die erste Hürde, die es für viele Perl-Entwickler auf dem Weg zum ersten XS-Modul zu nehmen gibt, ist die Sprache C. Der in C bereits bewanderte Leser kann diesen Abschnitt gerne überspringen. Allen anderen soll er helfen, sich dieser Sprache ein wenig zu nähern.

Im Grunde sind Perl und C gar nicht so verschieden. Für einen guten Teil der Perl-Syntax stand die Sprache C Pate. Etwas gewöhnungsbedürftiger als die Syntax ist für die meisten sicher, dass es in C wesentlich weniger "built-ins" gibt, und dass es sich um eine streng typisierte Sprache handelt. Hat man sich damit erst einmal abgefunden, stellt man fest, dass es kein Äquivalent zum CPAN gibt. Trotzdem hat die Sprache C unbestreitbar ihre Vorteile, die sie manchmal eben unentbehrlich machen. Aber dazu später mehr.

Beginnen wir einfach mit einem kleinen Beispiel, dem unvermeidlichen "Hello World":

```
#include <stdio.h>

int main(void)
{
  printf("Hello World\n");
  return 0;
}
```

Man sieht auf den ersten Blick, dass man mehr Code als in Perl schreiben muss. Bei der ersten Zeile handelt es sich um eine `#include`-Direktive des C-Präprozessors. Im Grunde das gleiche wie ein `use` oder `require` in Perl. Hiermit wird das Interface der Bibliothek für die Ein- und Ausgabe eingebunden. Diese Bibliothek brauchen wir, da im Sprachumfang von C keine Funktion zur Ausgabe von Text (`printf`) enthalten ist. Immerhin gehört diese Funktion aber zur ANSI-C-Standardbibliothek und ist somit überall verfügbar.

Weiterhin dürfen Anweisungen in C nicht außerhalb von Funktionen stehen und jedes C-Programm muß mindestens die Funktion `main` implementieren. Diese Funktion wird beim Start des Programms aufgerufen. Der Rückgabewert der Funktion `main` ist vom Typ `int` – also ein ganzzahliger Wert – und wird als Exit-Code des Programms verwendet. Die Funktion `printf` macht schließlich weitgehend das gleiche wie in Perl auch.

Dieses Programm gilt es nun zu übersetzen, was auf praktisch jedem *nix-System (z.B. auch unter cygwin) folgendermaßen funktioniert, sofern ein C-Compiler installiert ist:

```
$ cc -o hello hello.c
```

Idealerweise hat der C-Compiler (`cc`) an unserem Programm nichts auszusetzen und wir können es direkt starten:

```
$ ./hello
Hello World
```

Im nächsten Beispiel schauen wir uns Funktionen etwas genauer an. Wie in Perl auch, können C-Funktionen natürlich Argumente und Rückgabewerte haben. Allerdings kann man so ohne Weiteres nicht mehr als einen Wert von einer Funktion zurückgeben, und die Argumente und deren Typen müssen genau spezifiziert werden.

```
#include <stdio.h>

double multiply(double a, double b)
{
  double product = a*b;
  return product;
}

int main(void)
{
  double a = 3.14, b = 5.55;
  printf("%f*%f = %f\n",
          a, b,   multiply(a, b));
  return 0;
}
```

Die Funktion `multiply` gibt das Produkt der beiden Argumente `a` und `b` zurück:

```
$ cc -o function function.c
$ ./function
3.140000*5.550000 = 17.427000
```

Möchte man mehr als einen Wert von einer Funktion zurückgeben, kommt man nicht umhin, sich Zeiger etwas näher anzuschauen.

Zeiger

Glücklicherweise sind Zeiger in C fast dasselbe wie Referenzen in Perl. Nur etwas gefährlicher. Genau wie eine Referenz in Perl verweist ein Zeiger in C auf ein Datum, beispielsweise einen Integerwert oder eine Zeichenkette. Das ist dann auch gleich einer der Gründe, warum Zeiger so gefährlich sind: Während Perl im Normalfall *weiß*, auf welches Objekt eine Referenz verweist, ist C das völlig egal. Man kann problemlos einen Zeiger auf einen Integerwert als Zeiger auf eine Zeichenkette verwenden. Das kann nützlich sein, ist aber in den meisten Fällen schlicht falsch.

```
#include <stdio.h>

void test_value(int val, int *is_neg,
                int *is_odd, int *is_pow2)
{
  *is_neg  = val < 0;
  *is_odd  = val % 2 != 0;
  *is_pow2 = *is_neg
             ? 0 : !(val & (val - 1));
}

int main(void)
{
  int val[] = { -13, 27, 1024 };
  unsigned i;
  unsigned n = sizeof val/sizeof val[0];

  for (i = 0; i < n; i++)
  {
    int neg, odd, pow2;

    test_value(val[i], &neg, &odd, &pow2);

    printf(
      "%+8d : %s, %s, %sa power of 2\n",
       val[i],
              neg ? "negative" : "positive",
                  odd ? "odd" : "even",
                      pow2 ? "" : "not ");
  }

  return 0;
}
```

Die Funktion `test_value` bekommt neben dem zu überprüfenden Integerwert `val` drei Zeiger auf Intergerwerte übergeben. Diese werden dann in der Funktion zum Speichern der Rückgabewerte benutzt. Der Zugriff auf das von einem Zeiger referenzierte Objekt erfolgt mittels des Dereferenzierungsoperators `*`. Den Zeiger auf ein Objekt erhält man, wie man beim Aufruf der Funktion sehen kann, über den Adressoperator `&`.

Wir können den Code jetzt übersetzen und ausführen:

```
$ cc -o pointer pointer.c
$ ./pointer
     -13 : negative, odd, not a power of 2
     +27 : positive, odd, not a power of 2
   +1024 : positive, even, a power of 2
```

Der äquivalente Perl-Code kann z.B. so aussehen:

```
sub test_value
{
  my($val, $is_neg, $is_odd, $is_pow2) = @_;

  $$is_neg  = $val < 0;
  $$is_odd  = $val % 2 != 0;
  $$is_pow2 = $$is_neg
              ? 0 : !($val & ($val - 1));
}
```

```
my @val = ( -13, 27, 1024 );

for my $v (@val) {
  my($neg, $odd, $pow2);

  test_value($v, \$neg, \$odd, \$pow2);

  printf "%+8d : %s, %s, %sa power of 2\n",
         $v,
         $neg ? "negative" : "positive",
         $odd ? "odd" : "even",
         $pow2 ? "" : "not ";
}
```

Der Adressoperator `&` in C entspricht in etwa dem Referenzoperator `\` in Perl. Der Dereferenzierungsoperator `*` hat, je nach Typ der Referenz, mehrere Entsprechungen, z.B. `$` für Skalarreferenzen.

Selbstverständlich liefert der Perl-Code das gleiche Ergebnis:

```
$ perl pointer.pl
     -13 : negative, odd, not a power of 2
     +27 : positive, odd, not a power of 2
   +1024 : positive, even, a power of 2
```

Strings

Im Gegensatz zu Perl kennt C keine Strings. Ein String wird in C als Array von Zeichen dargestellt. Und da ein Array in C nichts weiter ist, als ein Zeiger auf das erste Element des Arrays, ist ein String in C ein Zeiger auf das erste Zeichen des Strings. Die Längeninformation muss entweder explizit gespeichert werden, oder sie ist implizit durch ein `'\0'`-Zeichen am Ende des Strings gegeben. Auch dazu ein kleines Beispiel:

```
#include <stdio.h>

static void greet(const char *how,
                  const char *who)
{
  printf("%s %s!\n", how, who);
}

int main(void)
{
  char w[6] = { 'W','o','r','l','d','\0' };
  char p[5] = "Perl";
  const char *h = "Hello";

  printf("w @ %p\np @ %p\nh @ %p\n",
             w,      p,      h);

  greet(h, p);
  greet(h, w);

  return 0;
}
```

Die Funktion `greet` bekommt zwei Zeiger auf `char` (Zeichen) übergeben. Das `const` bedeutet, dass die Zeiger potenziell in einen "konstanten" Speicherbereich zeigen, die Daten von dort also nur gelesen und nicht manipuliert werden dürfen. Das `static` im Funktionskopf sorgt übrigens dafür, dass diese Funktion nur lokal in der aktuellen Übersetzungseinheit sichtbar ist. Funktionen (und natürlich auch globale Variablen), die man nur in einer Datei verwendet, sollte man nach Möglichkeit `static` deklarieren. C unterstützt nämlich im Gegensatz zu Perl keine Namensräume ("Packages"), so dass es beim Linken leicht zu Kollisionen kommen kann.

Aber zurück zu den Strings! In `main` werden nun drei Stringvariablen angelegt. `w` zeigt den eigentlichen Aufbau eines Strings in C. Da diese Schreibweise aber nicht wirklich benutzerfreundlich ist, zeigt die Initialisierung von `p` eine äquivalente Alternative. Das `'\0'`-Zeichen wird in diesem Fall implizit erzeugt, man muss darauf achten, auch für dieses Zeichen Platz im Array zu reservieren.

Sowohl bei `w` als auch bei `p` wird der String auf dem Stack angelegt.Bei `h` dagegen liegt lediglich der Zeiger auf dem Stack. Dieser zeigt auf einen konstanten Speicherbereich, in dem die Zeichenkette `"Hello\0"` steht. Um dies zu verdeutlichen gibt das Programm in der nächsten Zeile die Adressen der drei Strings aus:

```
$ cc -o string string.c
$ ./string
w @ 0xbfaaa82a
p @ 0xbfaaa825
h @ 0x8048550
Hello Perl!
Hello World!
```

Der Cast-Operator

Der Cast-Operator dient in C zur expliziten Umwandlung eines Typs in einen anderen. Er wird in C relativ oft verwendet (manchmal leider viel zu oft), so dass wir ihn uns kurz anschauen sollten:

```
#include <stdio.h>

int main(void)
{
  unsigned short int u = 0xffff;
  unsigned int val = 0x00215358;
  char *str = (char *) &val;
  unsigned int adr = (unsigned int) str;
```

```
  if ((short int) u < 0)
    (void) printf("u = %hd\n",
                  (short int) u);

  (void) printf("str = \"%s\" (addr: %u)\n",
                  str,        adr);

  return 0;
}
```

Der Cast-Operator besteht einfach aus einem in runde Klammern geschriebenen Typ. Mit ihm können integrale Typen, zu denen auch die Zeiger gehören, konvertiert werden. In den beiden ersten Zeilen der `main`-Funktion werden die Variablen `u` und `val` deklariert und initialisiert. Das Schlüsselwort `unsigned` macht einen Integertyp vorzeichenlos; das Schlüsselwort `short` macht ihn "kurz", was typischerweise auf einem 32-Bit-System bedeutet, dass er statt 32 Bit nur 16 Bit breit ist. Es gibt auch das Schlüsselwort `signed`, dieses ist aber bei Integertypen redundant.

In der folgenden Zeile deklarieren wir die Variable `str` als Zeiger auf eine Zeichenkette. Die Variable wird mit einem Zeiger auf `val`, eine Integervariable, initialisiert. Danach initialisieren wir die Integervariable `adr` mit einem Zeiger. Ohne die expliziten Casts `(char *)` und `(unsigned int)` wäre der C-Compiler unzufrieden:

```
$ cc -c cast2.c
cast2.c: In function 'main':
cast2.c:7: warning: initialization from
incompatible pointer type
cast2.c:8: warning: initialization makes
integer from pointer without a cast
```

Mit dem Cast können wir dem Compiler klar machen, dass wir uns sicher sind, was wir an der Stelle tun. Allerdings begibt man sich damit gerne auf dünnes Eis: ändert man etwas am Code, so kann einen der Compiler an dieser Stelle nicht mehr darauf hinweisen, dass man etwas potenziell Gefährliches macht. Man sollte es daher mit den Casts nicht übertreiben, sondern sie nur dort einsetzen, wo sie unbedingt notwendig oder ungefährlich sind.

In der `if`-Bedingung *casten* wir nun die Variable `u` auf den Typ `short int`, wir wandeln also den vorzeichenlosen Wert in `u` in einen vorzeichenbehafteten Integer gleicher Breite um. Der Wert `0xffff` ist im Zweierkomplement gleich `-1`, die Zahl ist somit negativ und sollte demzufolge ausgegeben werden.

Bei den beiden Ausgaben mit `printf` fällt noch ein weiterer Cast direkt vor dem Funktionsaufruf auf. Man muss dazu wissen, dass in C die Funktion `printf` die Anzahl der geschriebenen Zeichen (oder einen negativen Wert im Fehlerfall) zurückgibt. Möchte man dem Compiler mitteilen, dass man absichtlich den Rückgabewert einer Funktion, in diesem Fall `printf`, ignoriert, kann man den Rückgabewert einfach auf `(void)` casten. `void` bedeutet so viel wie "nichts konkretes". Man kann keine `void`-Variable instanziieren, und man kann `void`-Zeiger nicht direkt dereferenzieren.

```
$ cc -o cast cast.c
$ ./cast
u = -1
str = "XS!" (addr: 3217047940)
```

Das Programm gibt übrigens nur auf einer Little-Endian-Architektur den String `"XS!"` aus, auf einem Big-Endian-System würde der Leerstring ausgegeben.

Der GNU-Debugger

Während bei Perl das `print`-Debugging meist einem `perl -d` vorgezogen wird, nimmt man bei C statt einem `printf` gerne auch den GNU-Debugger `gdb`. Um ein C-Programm sinnvoll mit dem `gdb` benutzen zu können, sollte man das Programm mit Debug-Symbolen übersetzen. Das geht bei den meisten Compilern mit der Option `-g`. Erinnern wir uns an das Beispiel mit der Funktion `multiply`:

```
$ cc -g -o function function.c
```

Starten wir nun den Debugger:

```
$ gdb -q ./function
Using host libthread_db library
"/lib/libthread_db.so.1".
(gdb) b multiply
Breakpoint 1 at 0x80483ee: file function.c,
line 5.
(gdb) r
Starting program:
/home/mhx/src/perl/xstutorial/2008/code/func
tion

Breakpoint 1, multiply
(a=3.1400000000000001, b=5.5499999999999998)
at function.c:5
5       double product = a*b;
(gdb) n
6       return product;
(gdb) p product
$1 = 17.427
(gdb) c
Continuing.
3.140000*5.550000 = 17.427000

Program exited normally.
(gdb) q
```

Als erstes setzen wir mit `b` (`break`) einen Breakpoint auf die Funktion `multiply` und starten danach mit `r` (`run`) das Programm. Es läuft dann in den von uns gesetzten Breakpoint, und der `gdb` zeigt uns die Funktion und die Zeile, in der wir gerade stehen. Mit `n` (`next`) wird dann nur die angezeigte Zeile ausgeführt und wir stehen beim nächsten Statement. Jetzt können wir uns das Ergebnis der Multiplikation mit `p` (`print`) ansehen. Mit `c` (`continue`) läuft das Programm weiter und wird beendet, da wir in keinen weiteren Breakpoint laufen. Mit `q` (`quit`) können wir den `gdb` dann verlassen.

Inline::C

Wenn es darum geht, Perl- und C-Code miteinander zu verbinden, stolpert man früher oder später über das Modul `Inline::C`, das gerne als einfache Alternative zu XS propagiert wird. Im Grunde klingt es ja auch recht verlockend, wenn man – wie es in der Dokumentation dieses Moduls steht – einfach Perl-Funktionen in C schreiben kann. Und tatsächlich, genauso einfach scheint es in der Praxis auch zu sein:

```
use strict;
use warnings;
use 5.010;
use Inline 'C';

say add(9, 16);

__END__
__C__

int add(int x, int y)
{
  return x + y;
}
```

Führt man dieses Skript aus, so erhält man das erwartete Ergebnis:

```
$ perl inline.pl
25
```

Ganz anders sieht es leider aus, wenn die Beispiele etwas komplexer werden. Da gibt es dann, wie in XS auch, einen ganzen Baukasten an Hilfsfunktionen. Leider sind diese aber nicht kompatibel zu den von XS verwendeten, so dass es nicht einfach möglich ist, Projekte von XS auf `Inline::C` umzustellen und umgekehrt.

Ein weiterer Nachteil ist, dass das Modul scheinbar nicht mehr gewartet wird; zumindest gab es seit 6 Jahren keine neue Version mehr auf dem CPAN. Auch scheint der Autor selbst wohl nicht (mehr) ganz überzeugt von `Inline::C` zu sein. Er hat zwischenzeitlich zwar mehrere Module veröffentlicht, die XS benutzen, aber keines, das `Inline::C` verwendet.

Hello XS

Es gibt viele Wege zum ersten XS-Modul. Zum Beispiel kann man sich ein (einfaches) XS-Modul im CPAN suchen und dieses dann nach den eigenen Wünschen modifizieren. Eine andere Möglichkeit ist die Verwendung von `h2xs`:

```
$ h2xs -An Hello
Defaulting to backwards compatibility with
perl 5.11.0
If you intend this module to be compatible
with earlier perl versions, please
specify a minimum perl version with the -b
option.

Writing Hello/ppport.h
Writing Hello/lib/Hello.pm
Writing Hello/Hello.xs
Writing Hello/Makefile.PL
Writing Hello/README
Writing Hello/t/Hello.t
Writing Hello/Changes
Writing Hello/MANIFEST
```

Das Tool erzeugt für das neue Modul `Hello` ein Verzeichnis `Hello` mit den oben aufgeführten Dateien. Alle Dateien bis auf `Hello.xs` sollten jedem bekannt sein, der schon mal ein Perl-Modul geschrieben hat. `Hello.xs` sollte etwa so aussehen:

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"

#include "ppport.h"


MODULE = Hello              PACKAGE = Hello
```

Am Anfang werden mehrere Dateien eingebunden, wobei die ersten drei obligatorisch sind. Auf die Datei `ppport.h` wird später noch genauer eingegangen. Sie hilft dabei, ein XS-Modul zu älteren Perl-Versionen kompatibel zu machen.

Direkt nach den `#include`-Direktiven kann in der XS-Datei beliebiger C-Code stehen, beispielsweise kleinere Funktionen, die man später aus dem XS-Code heraus aufrufen möchte.

Der XS-Code beginnt mit der ersten MODULE-Direktive. Mit dieser wird gleichzeitig der Modulname gesetzt. Mit Hilfe von PACKAGE können innerhalb einer XS-Datei XS-Routinen mehreren Packages zugeordnet werden. Das PACKAGE muss dabei auf der gleichen Zeile angegeben werden wie die MODULE-Direktive.

Hier endet das von h2xs generierte Template. Ab hier können wir nun unsere erste XS-Funktion – oder kurz XSUB – schreiben:

```
void
Hello()
    CODE:
        printf("Hello XS!\n");
```

Abgesehen von der CODE:-Zeile und den fehlenden geschweiften Klammern sieht diese XSUB doch stark nach einer C-Funktion aus. Das ist im Grunde auch gar nicht so falsch, denn beim Übersetzen der XSUB wird lediglich noch ein wenig C-Code eingefügt, der die Verbindung zum Perl-Kern herstellt.

Nach einem

```
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Hello
$ make
cp lib/Hello.pm blib/lib/Hello.pm
Please specify prototyping behavior for
Hello.xs (see perlxs manual)
Running Mkbootstrap for Hello ()
Manifying blib/man3/Hello.3
```

finden wir im Modulverzeichnis auch eine Datei Hello.c, die genau diesen generierten C-Code enthält:

```
XS(XS_Hello_Hello)
{
#ifdef dVAR
    dVAR; dXSARGS;
#else
    dXSARGS;
#endif
    if (items != 0)
       Perl_croak(aTHX_ "Usage: %s(%s)", \
"Hello::Hello", "");
    PERL_UNUSED_VAR(cv); /* -W */
    {
#line 13 "Hello.xs"
        printf("Hello XS!\n");
#line 38 "Hello.c"
    }
    XSRETURN_EMPTY;
}
```

Außerdem haben wir damit gerade unser erstes XS-Modul fertig kompiliert. Dieses kann nun einfach aus der Shell heraus getestet werden:

```
$ perl -Mblib -MHello -e'Hello::Hello()'
Hello XS!
```

Tools und Dokumentation

Ein Tool, h2xs, haben wir eben schon kennen gelernt. Seine eigentliche Aufgabe, neben dem Erstellen eines Modul-Templates, ist es, C-Headerdateien zu interpretieren und daraus automatisch ein XS-Interface zu erstellen. Dazu ist es allerdings notwendig, das CPAN-Modul C::Scan zu installieren.

Eher unbewußt haben wir aber auch noch XSUB compiler, xsubpp, verwendet, und zwar beim Aufruf von make. Dieses Tool liest die XS-Datei ein und generiert daraus eine reine C-Datei, die dann vom C-Compiler übersetzt wird. Der xsubpp kümmert sich bei der Codegenerierung beispielsweise um das Stack-Management, die Konvertierung der Übergabe- und Rückgabewerte der einzelnen XSUBs und die Erzeugung des Bootcodes des Moduls. Dieser Bootcode führt unter anderem die Registrierung der XSUBs beim Perl-Interpreter durch.

Die Dokumentation zum Thema XS verteilt sich auf mehrere Manpages. Neben der Referenz des XS-Formats in perlxs und dem Tutorial in perlxstut ist es sinnvoll, die C-Level Perl-API zu kennen, die in perlapi beschrieben ist. perlcall beschreibt, wie man aus XS bzw. C heraus Perl-Subroutinen aufrufen kann. Die Perl-Interna, die im nächsten Abschnitt kurz behandelt werden, sind ausführlich in perlguts und anschaulich unter *http://gisle.aas.no/perl/illguts/* beschrieben.

Perl Interna

Dieser Abschnitt behandelt die von Perl intern genutzen Datentypen sowie grundlegende Konzepte der Perl-Interna, die man spätestens bei größeren XS-Projekten sehr wahrscheinlich benötigt.

Die Datentypen, die man aus Perl kennt, z.B. Skalare, Arrays, Hashes oder Referenzen, haben direkte Pendants auf C-Ebene. Der C-Typ `SV` repräsentiert einen Skalar, `AV` ein Array, `HV` einen Hash und `RV` eine Referenz. Das `V` steht dabei jeweils für *Value*, also *Scalar Value* im Falle des *SV*. Weiterhin gibt es noch `IV`, `NV` und `PV`, die Integerwerte, Fließkommawerte und Strings repräsentieren. Wie wir ja von Perl wissen, kann ein `SV` sowohl einen `IV` als auch einen `NV` oder einen `PV` darstellen. Dem einen oder anderen ist vielleicht sogar bekannt, dass ein `SV` sogar einen Integer und einen davon unabhängigen String gleichzeitig speichern kann (was z.B. bei `$!` ausgenutzt wird). Wie das funktioniert, wird deutlich, wenn man sich die Beziehungen zwischen den einzelnen Datentypen genauer ansieht.

```
          SVt_NULL
         /   |    \
  SVt_IV  SVt_PV  SVt_NV
         \   |    /
          SVt_PVIV /
             |   /
          SVt_PVNV
             |
          SVt_PVMG
         /   |    \
SVt_PVAV SVt_PVHV SVt_PVCV
```

Dies ist ein Auszug aus der Hierarchie der Perl-Objekte. `SVt_NULL` ist dabei die Basisklasse aller Perl-Datentypen. Eine Instanz dieser Klasse repräsentiert den Wert `undef` in Perl. Auf C-Ebene sieht die Instanz folgendermaßen aus:

```
+---------------------+
| sv_any              |
+---------------------+
| sv_refcnt           |
+---------------+-----+
| sv_flags      |(SVt)|
+---------------+-----+
| sv_u                |
+---------------+-----+
```

Dieser Teil ist für jeden Perl-Datentyp exakt gleich. Der Zeiger `sv_any` zeigt auf weitere mit diesem `SV` assoziierte Daten. Im Falle von `SVt_NULL` ist dieser Zeiger immer `NULL`, er zeigt auf nichts. Die beiden Elemente `sv_refcnt` und `sv_flags` sind jeweils 32-Bit Integerwerte. In `sv_refcnt` steht der Reference Count der Instanz, also die Anzahl der Objekte, von der die Instanz referenziert wird.

`sv_flags` besteht aus zwei Teilen. In den unteren 8 Bit steht der Typ (z.B. `SVt_NULL` oder `SVt_PVNV`) der Instanz, in den oberen 24 Bit stehen einzelne Flags, die bestimmte Eigenschaften der Instanz beinhalten. Typischerweise sind für den XS-Entwickler nur wenige dieser Flags wirklich interessant: beispielsweise `SVf_READONLY`, das anzeigt, ob die Instanz nur gelesen werden darf, oder `SVf_IOK`, `SVf_NOK`, `SVf_POK`, die angeben, ob die Elemente (*Slots*), in denen der `IV`, `NV` oder `PV` der Instanz liegen, gültige Werte enthalten. Hintergrund ist, dass z.B. der Skalar `SVt_PVIV`, der sowohl einen String als auch einen Integer enthalten *kann*, nicht auch zwangsläufig einen gültigen String- oder Integerwert enthalten *muß*. Es ist völlig legitim, wenn er nur einen String enthält.

`sv_u` kann nun, je nach Typ, noch ein zusätzliches Datum enthalten; bei einem `SVt_IV` beispielsweise den `IV`. Dieses Element ist aus Optimierungsgründen mit Perl 5.10 neu hinzugekommen. Bisher war es für alle Typen außer `SVt_NULL` notwendig, eine weitere Datenstruktur anzulegen und diese über `sv_any` zu referenzieren. Gerade bei einem Integerwert oder einer Referenz bedeutet dies jedoch erheblichen Mehraufwand.

Devel::Peek

`Data::Dumper` ist für den Perl-Entwickler eine fast unverzichtbare Hilfe. Eine ähnliche Bedeutung hat das Modul `Devel::Peek` für den XS-Entwickler. Die wichtigste Funktion dieses Moduls ist `Dump`.

```
use Devel::Peek;

Dump $a;

$a = "42";
Dump $a;

print $a == 13;
Dump $a;

$a = "foo";
Dump $a;
```

Dieses Skript erzeugt die folgende Ausgabe:

```
SV = NULL(0x0) at 0x816c2d0
  REFCNT = 1
  FLAGS = ()
SV = PV(0x8159048) at 0x816c2d0
  REFCNT = 1
  FLAGS = (POK,pPOK)
  PV = 0x8163c38 "42"\0
  CUR = 2
  LEN = 4
SV = PVIV(0x8162bec) at 0x816c2d0
  REFCNT = 1
  FLAGS = (IOK,POK,pIOK,pPOK)
  IV = 42
  PV = 0x8163c38 "42"\0
  CUR = 2
  LEN = 4
```

```
SV = PVIV(0x8162bec) at 0x816c2d0
  REFCNT = 1
  FLAGS = (POK,pPOK)
  IV = 42
  PV = 0x8163c38 "foo"\0
  CUR = 3
  LEN = 4
```

Man kann an diesem Beispiel sehr schön die Entwicklung von `$a` verfolgen. Am Anfang is `$a` nicht definiert, demzufolge verbirgt sich dahinter ein `SVt_NULL`. Da es lediglich eine Referenz auf den `SV` gibt (nämlich `$a`), ist der Reference Count (`REFCNT`) `1`. Es sind keine Flags gesetzt.

Die Zuweisung des Strings `"42"` erzwingt ein "Upgrade" des `SV`s zum Typ `SVt_PV`. Es sind nun die Flags `SVf_POK` und `SVp_POK` gesetzt, die anzeigen, dass der `SV` einen gültigen String (`PV`) beinhaltet. `CUR` und `LEN` geben die Länge des Strings und die Größe des Stringpuffers an. Am Reference Count hat sich natürlich nichts geändert.

Im nächsten Schritt wird `$a` nun im numerischen Kontext verwendet. Dies erzwingt einen erneuten Upgrade zum Typ `SVt_PVIV` und gleichzeitig eine numerische Interpretation des Strings. Da nun auch der `IV`-Slot gültig ist, sind die Flags `SVf_IOK` und `SVp_IOK` gesetzt.

Wird nun erneut ein String (`"foo"`) zugewiesen, wird der `PV`-Slot überschrieben und der `IV`-Slot ungültig, die entsprechenden Flags sind also nicht mehr gesetzt. Trotzdem ist der Inhalt des `IV`-Slots weiterhin unverändert vorhanden. Am Typ des `SV`s ändert sich also nichts.

Auf XS-Ebene hat man jedoch die Möglichkeit, die einzelnen Slots und Flags beliebig zu manipulieren, so dass es recht einfach möglich ist, unterschiedliche Werte in den `PV`- und `IV`-Slots zu speichern und beide als gültig zu erklären. Auf genau diese Art funktionieren `$!` oder auch die `dualvar` Funktion aus `Scalar::Util`.

Reference Counting

Eine besondere Bedeutung kommt dem Reference Count einer Perl-Variable zu. Ein falscher Reference Count kann zu Ressourcenlecks (in erster Linie Speicherlecks, aber z.B. auch Dateien, die zwar geöffnet, aber nicht wieder geschlossen werden) oder zu Programmabstürzen führen.

Jede Perl-Variable hat initial einen Reference Count von 1. Jedes Objekt, das fortan diese Variable referenziert, muss deren Reference Count mit Hilfe der Funktion `SvREFCNT_inc` um 1 erhöhen. Sobald die Variable nicht mehr referenziert wird, muss der Reference Count mittels `SvREFCNT_dec` dekrementiert werden. Sobald der Reference Count dabei auf 0 dekrementiert wird, wird die Variable automatisch zerstört. Handelt es sich bei der Variablen z.B. um ein Array, so wird dabei der Reference Count jedes einzelnen Elements dekrementiert. Auf diese Art *leben* Variablen nur so lange, wie sie gebraucht werden.

Interessant wird das Ganze, wenn man z.B. aus einer XSUB eine Variable zurückgeben möchte, die man danach aber nicht mehr referenziert. Streng genommen müsste man ja den Reference Count dieser Variablen dekrementieren. Das darf man aber nicht, da dies die Variable zerstören würde. Aus diesem Grund gibt es das Konzept der *Mortality*, der Sterblichkeit von Variablen.

Man kann eine Variable z.B. mittels der Funktion `sv_2mortal` als "sterblich" markieren, was zur Folge hat, dass der Reference Count dieser Variable *kurze Zeit später* dekrementiert wird. Sollte er zu diesem Zeitpunkt noch auf 1 stehen, so wird die Variable zerstört. Wurde sie aber in der Zwischenzeit von einem Empfänger referenziert, der Reference Count also auf 2 inkrementiert, wird der Reference Count lediglich auf den korrekten Wert 1 dekrementiert.

Perl API

Perl bietet dem XS-Entwickler eine reichhaltige API, in der sich alle notwendigen Funktionen finden, um z.B. Skalar-, Array- und Hashvariablen zu manipulieren, Speicher zu verwalten oder auf den Stack zuzugreifen. Die Perl-API ist ausführlich in `perlapi` beschrieben, im Folgenden werden nur einige Beispiele gezeigt.

Die Namen der Funktionen zur Manipulation von `SV`s, `AV`s und `HV`s beginnen typischerweise mit `sv_`, `av_` bzw. `hv_`. Eine Ausnahme bilden die Konstruktoren, die `newSV`, `newAV` und `newHV` heißen.

Es gibt eine ganze Reihe von globalen Variablen, deren Namen mit `PL_` beginnen. Für häufig benötigte Werte wie `undef`, wahr oder falsch gibt es z.B. die Variablen `PL_sv_undef`, `PL_sv_yes` und `PL_sv_no`.

Zur Speicherverwaltung dienen Funktionen wie `Newx` zum Anlegen von Speicher, `Copy` und `Move` zum Kopieren und Verschieben von Speicher oder `Safefree` zum Freigeben von angelegtem Speicher.

Außerdem gibt es noch jede Menge nützliche Funktionen, z.B. `strEQ` zum Vergleichen von Strings auf Gleichheit, `croak` und `warn`, die die gleiche Bedeutung haben wie `die` und `warn` in Perl, oder `GIMME_V`, mit dem sich wie mit `wantarray` in Perl der Kontext eines Funktionsaufrufs ermitteln läßt.

Anbinden von Bibliotheken

Ein großer Anwendungsbereich von XS ist die Anbindung externer C-Bibliotheken an Perl. Im Idealfall ist das recht einfach, da `h2xs` den Großteil der Arbeit erledigt.

Nehmen wir eine sehr einfache C-Bibliothek an, die aus den folgenden drei Dateien besteht. Zuerst die Implementierung in `hello.c`:

```
#include <stdio.h>
#include <math.h>

#include "hello.h"

void hello(void)
{
  printf("Hello World!\n");
}

double round(double arg)
{
  if (arg > 0.0)
    arg = floor(arg + 0.5);
  else if (arg < 0.0)
    arg = ceil(arg - 0.5);

  return arg;
}
```

Die dazu gehörende Header-Datei `hello.h`:

```
#ifndef _HELLO_H
#define _HELLO_H

void hello(void);
double round(double);

#endif
```

Und schließlich das `Makefile`:

```
LIB=libhello.a
OBJ=hello.o

all: lib

lib: $(LIB)

$(LIB): $(OBJ)
      ar cru $(LIB) $(OBJ)

clean:
      rm -f $(OBJ)

realclean: clean
      rm -f $(LIB)
```

Das Makefile wird in dieser Form vermutlich nur auf Unix-ähnlichen Systemen funktionieren. Legt man diese drei Dateien in ein Verzeichnis `libhello` und ruft `make` auf, sollten anschließend die Objektdatei `hello.o` und die Bibliothek `libhello.a` in dem Verzeichnis liegen.

Jetzt können wir mit Hilfe von `h2xs` vollautomatisch den XS-Code zur Anbindung der Bibliothek erzeugen:

```
$ h2xs -Axn Hallo libhello/hello.h
Defaulting to backwards compatibility with
perl 5.11.0
If you intend this module to be compatible
with earlier perl versions, please
specify a minimum perl version with the -b
option.

Writing Hallo/ppport.h
Scanning typemaps...
 Scanning
/home/mhx/perl/blead/lib/5.11.0/ExtUtils/typ
emap
Scanning libhello/hello.h for functions...
Scanning libhello/hello.h for typedefs...
Writing Hallo/lib/Hallo.pm
Writing Hallo/Hallo.xs
Writing Hallo/Makefile.PL
Writing Hallo/README
Writing Hallo/t/Hallo.t
Writing Hallo/Changes
Writing Hallo/MANIFEST
```

In dem dabei generierten `Makefile.PL` müssen wir lediglich noch die Bibliothek eintragen und den Include-Pfad anpassen, also die entsprechenden Zeilen wie folgt abändern:

```
LIBS => ['-L../libhello -lhello'],
INC  => '-I..',
```

Jetzt sollte sich das Perl-Modul mit dem üblichen

```
$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Warning: -L../libhello changed to
-L/home/mhx/src/perl/xstutorial/2008/externa
l-lib/Hallo/../libhello
Writing Makefile for Hallo
$ make
cp lib/Hallo.pm blib/lib/Hallo.pm
Please specify prototyping behavior for
Hallo.xs (see perlxs manual)
Running Mkbootstrap for Hallo ()
/usr/bin/ld: warning: creating a DT_TEXTREL
in object.
Manifying blib/man3/Hallo.3
```

bauen lassen, und wir können es ausprobieren:

```
$ perl -Mblib -MHallo -e'Hallo::hello()'
Hello World!
$ perl -Mblib -MHallo=round \
  -le'print round($_) for -1.3,2.2,3.7'
-1
2
4
```

Fairerweise sollte jedoch nicht verschwiegen werden, dass das Anbinden von Bibliotheken auf diese Art und Weise meist nicht ganz so problemlos funktioniert. Das liegt unter anderem daran, dass `C::Scan` nur Teile der C-Syntax versteht. In den meisten Fällen wird man daher nicht umhin kommen, den generierten XS-Code noch selbst zu bearbeiten.

Ausblick

Der nächste Teil wird sich hauptsächlich mit Beispielen zur fortgeschrittenen XS-Programmierung und dem Debuggen von XS-Code beschäftigen.

Marcus Holland-Moritz

TPF-TICKER

CIO-Interview mit Richard Dice

In einem Interview über dynamische Programmiersprachen wurde für "Perl" Richard Dice - Präsident der Perl Foundation - von der Zeitschrift CIO befragt. In dem Interview gehen die Vertreter der einzelnen Sprachen auf die Zukunft von dynamischen Programmiersprachen und ihre Sprache im Speziellen ein.

Das Interview ist unter http://www.cio.com/article/446829/PHP_JavaScript_Ruby_Perl_Python_and_Tcl_Today_The_State_of_the_Scripting_Universe?contentId=446829&slug=& erreichbar

Grants für das 3. Quartal 2008

Auch für das 3. Quartal 2008 sind wieder gute Grant-Vorschläge bei der Perl Foundation eingegangen. Das Grant Committee hat die Vorschläge bewertet, nachdem sie öffentlich diskutiert werden konnten.

Die folgenden fünf Vorschläge werden jetzt gefördert:
* Perl cross-compilation for linux and wince
* Barcode support in Act
* Tcl/Tk access for Rakudo
* Embedding perl into C++ applications
* Extending BSDPAN

Perl 6 Tutorial - Teil 5 : Captures und Subroutinen

Willkommen zum fünften Teil dieses ausführlichen Perl 6-Tutorials, der sich nur mit dem kleinen "sub"-Befehl beschäftigen wird und einigen Dingen die dazu gehören.

Subroutinen oder Funktionen, die in Perl 6 immer noch mit `sub` deklariert werden, gehören zu den wichtigsten und grundlegendsten Techniken der Programmierkunst. Es sind Teilprogramme, die mit einem möglichst aussagekräftigen Namen aufgerufen werden. Meist werden ihnen Werte übergeben und oft liefern sie auch ein Ergebnis zurück. Jeder der schon etwas Perl kennt, hat bereits ähnliches geschrieben, wie folgende Funktion, welche die Länge der Hypotenuse berechnet.

```
sub hypotenuse {
        my ($a, $b) = @_;
        sqrt( $a**2 + $b**2 );
}
```

Und die gute Nachricht für Lernfaule lautet: Dies ist vollständig gültiges Perl 6. Menschen die jedoch gerade von C oder Java zu Perl wechseln, hätten die Subroutine wohl so geschrieben:

```
sub hypotenuse ($a, $b) {
        return sqrt( $a*$a + $b*$b );
}
```

Und die gute Nachricht für diese Neulinge lautet: Dies ist ebenfalls vollständig gültiges Perl 6. Die zweite Lösung liegt näher an dem, was die meisten anderen Sprachen da draußen betreiben (entspricht häufig den Lesegewohnheiten) und sie spart mühevolle Fingerbewegungen (ebenfalls ein Indikator ob etwas perlish ist). Aber was genau wurde geändert? Folgen dem `sub`-Namen runde Klammern, definiert das eine Signatur (Liste der Parameter) und keine Prototypen mehr wie in Perl 5. Diese Parameter (auch `@_`) sind keine lokalen Variablen und (wenn vorher nicht anders deklariert) schreibgeschützt. Folgt keine Signatur, landen Parameter in `@_`, wie aus Perl 5 gewohnt. TIMTOWTDI.

Geforderte Parameter

Signaturen helfen jedoch nicht nur Neulingen, sondern nehmen jedem Arbeit ab. Zum Beispiel kann `sub hypotenuse` nur dann sinnvolle Ergebnisse erzielen, wenn sie zwei Parameter bekommt. Im alten System der Parameterübergabe wäre es aufwendig, dies einzufordern. In Perl 6 braucht der Programmierer dafür gar nichts zu tun, denn würde man die zweite Routine mit `hypotenuse(5)` oder `hypotenuse(2,3,4)` aufrufen, gäbe es eine dicke Fehlermeldung zur Kompilierungszeit. Wäre die `sub` wie im ersten Beispiel implementiert, würde der Kompiler lautlos schnurren wie ein Kätzchen, auch wenn die Ergebnisse der Funktion nicht immer brauchbar wären. Doch wenn schon die Parameter prüfen, dann auch auf den Datentyp.`Num` ist ein nativer Datentypen für Skalare und entspricht dem Zahlenbereich rationaler Zahlen.

```
sub hypotenuse (Num $a, Num $b) {
        return sqrt( $a*$a + $b*$b );
}
```

Der Vorteil einer solchen Prüfung: die Fehlermeldung kommt, wenn Unbeabsichtigtes passiert (hier bei nicht-numerischen Längenangaben), nicht erst Zeilen später, wenn ein Folgebefehl versagt. Somit sind Ursachen wesentlich leichter auffindbar. Und um konsequent zu sein, sollte die `sub hypotenuse` nur Zahlen größer als 0 zulassen, da sie mit physischen Längen rechnet. Dazu definieren wir den eigenen Datentyp `Num+` (analog zu Q+) als Untermenge von `Num`.

Perfekt wäre es, wenn auch der Typ des Rückgabewertes festgelegt werden könnte. Das wirkt in diesem Beispiel vielleicht etwas übertrieben, aber Programme werden so zuverlässiger. Diese Art der Programmierung, bei der der Compiler die Anzahl und Typen der Ein- und Ausgabe einer Routine prüft, wird "Design by Contract" genannt und wurde zuerst

durch die Sprache Eiffel bekannt. Da das Perl 6 Motto heißt "All your Paradigm's belong to us", ist das Beschriebene auch hier möglich:

```
Num+ sub hypotenuse (
      Num+ $a, Num+ $b) { ... }
sub hypotenuse (
      Num+ $a, Num+ $b --> Num+) { ... }
```

Bla Bla Bla

Die "..." in den letzten Beispielen waren nicht nur Dekoration, die andeutet, dass an dieser Stelle der eigentliche Code später eingefügt wird. Auch Perl 6 versteht es in dem Sinne. Der Interpreter geht davon aus, dass der Programmierer nicht ohne Grund beginnt, eine Subroutine zu schreiben, und möchte ihm mit einem freundlichen Kompilerfehler daran erinnern, dass im Falle von

```
sub hypotenuse;     # oder
sub hypotenuse { }
```

etwas fehlt. Der geschickte Softwarearchitekt kann aber mit dem Operator namens "yadayadayada" (zu deutsch bla bla bla) dem Interpreter mitteilen, dass er später die Routine schreiben wird. Je nachdem, ob er möchte, das die jetzige Routine ein fail, eine Warnung (`warn`), oder einen Error (`die`) liefert, kann er die Schreibweisen `...`, `???` oder `!!!` verwenden. Doch kehren wir zurück zum ersten Beispiel.

Optionale Parameter

Manchmal braucht es Routinen, die je nach Situation unterschiedlich viele Parameter bekommen. Wie deklariere ich das in der Signatur, ohne dass sich der Interpreter beschwert? Indem ich den Parametern ein Fragezeichen anhänge und sie damit als optional markiere.

```
sub hypotenuse (Num+ $a, Num+ $b,
           Str $txt? --> Num+) {
       my $h = sqrt( $a*$a + $b*$b );
       say "$txt $h." if $txt;
       return $h;
}
```

Man muss nur darauf achten, in der Signatur die optionalen Parameter nach den Notwendigen zu positionieren, da alle Parameter in der Reihenfolge befüllt werden, in der sie der Routine übergeben werden. Deshalb sollte auch die Reihenfolge der optionalen Parameter sorgfältig überdacht werden, um sie von links nach rechts von mehr zu weniger wichtig zu sortieren. Der Befehl `if $txt` wird auf keinen Fall Probleme bereiten. Denn auch wenn kein Antwortsatz gewünscht ist und `$txt` leer bleibt, wird die Variable auf jeden Fall mit `undef` initialisiert. Um andere default-Werte festzulegen könnte man schreiben:

```
Num+ sub hypotenuse (Num+ $a, Num+ $b,
                     Str $txt?) {
        my $h = sqrt( $a*$a + $b*$b );
        $txt //= 'Länge:
        say "$txt $h.";
        return $h;
}
```

oder

```
Num+ sub hypotenuse (Num+ $a, Num+ $b,
              Str $txt = 'Länge: ') {
        my $h = sqrt( $a*$a + $b*$b );
        say "$txt $h.";
        return $h;
}
```

Da optionale Parameter auch an der Zuweisung in der Signatur erkannt werden, darf das Fragezeichen im letzten Beispiel weggelassen werden. Das Gegenteil des Fragezeichens ist das Ausrufezeichen (siehe dem Bedingunsoperator `??!!`) Deshalb werden notwendige Parameter mit einem angehängten `$var!` deklariert, was aber bei positionalen Parametern nicht notwendig - da default - ist.

Schlürfende Parameter

Manchmal ist es aber auch praktisch eine unbekannte Anzahl von Parametern in einem Array zusammenzufassen. Dies erreicht man durch einen Stern als Präfix:

```
sub summe (*@a) { [+] @a }
```

Aber auch Hashes und Skalare aller Art dürfen als "slurpy" deklariert werden. Im folgenden Beispiel implementieren wir Perls map-Funktionen. Mit dem Unterschied, dass bei unserem `map` der anonyme Block an beliebiger Stelle stehen darf und der Interpreter die Parameter anhand der Signatur passend zuordnet.

```
sub map (*&code, *@werte) {
        return gather for @werte -> $wert {
                take $code($wert);
        }
}
```

Würde die Sigantur `(*@werte, *&code)` lauten, müsste der Block immer an letzter Stelle übergeben werden.

Benannte Parameter

Wie angedeutet waren alle bisherigen Parameter positional, richten sich also nach der Reihenfolge im Funktionsaufruf.

```
sub hypotenuse ($a, $b) {
        sqrt( $a*$a + $b*$b );
}
```

Selbst bei einem Aufruf wie `hypotenuse($b, $a)` würde der Inhalt der Variable `$b` in den Parameter `$a` kopiert werden und analog Variable `$a` in `$b`. Es gibt jedoch sehr gute Gründe Parameter manchmal direkt beim Namen anzusprechen. Der Quellcode wird nachvollziehbarer und nicht jeder kann sich die Reihenfolge von 12 Parametern für jede Routine oder Methode merken. Oft ist auch nicht die Eingabe von jedem Parameter notwendig. Aber wie soll der Computer erkennen, dass ich dieses Mal die Parameter 3, 5 und 12 übergebe? Selbst die bereits vorgestellten optionalen Parameter waren positional. Aus der Überlegung erschließt sich auch warum in Perl 6 benannte Parameter per default optional sind. Wie bekannt kann das angehängte ! sie erzwingen. Benannte Parameter erkennt man am Präfix ":". Und sehr zu beachten: sie folgen den positionalen Parametern in der Signatur.

```
sub new(:$parent, :$ID, :$value :@size,
            :@pos, :@item, $:style)
```

Es gab Momente, da wünschte ich mir beim WxPerl-Programmieren, Perl 6 wäre schon da und z.B. die `new`-Methode einer Combobox hätte eine solche Signatur. Denn beim Erzeugen des Widgets sind bei mir nur die Eltern (Platz in der Objekthierarchie) und der Sytle (Aussehen und Verhalten) wichtig. Die ID lasse ich automatisch generieren, Größe und Position bestimmen die Sizer und Textwert und die Items werden bei Bedarf gesetzt.

```
my $cb = Wx::Combobox.new(
           parent => $win, :style($style) );
```

So würde mir das gefallen. Zu Demonstrationszwecken enthält das letzte Beispiel beide Paar-Schreibweisen. Sie wurden bereits in der vorigen Folge erläutert. Nur ließe sich hier `:style($style)` auch zu `:$style` zusammenfassen.

Hat eine Routine keine Signatur, erhält sie ihre mit Namen zugewiesenen Parameter aus `%_`, so wie `@_` nur die positionalen Parameter enthält. Verwendet man Platzhalter-Variablen wie `$^a` in Routinen ohne Signatur (obwohl die nur für einfache Blöcke gedacht sind), erscheinen die Werte dieser Variablen nicht mehr in `%_` oder `@_`.

Möchte man ein Paar als positionalen Parameter angeben, muss es in runde Klammern gesetzt werden. Wie nachfolgend zu sehen, kann man die umschließenden Klammern einer Signatur meist weglassen.

```
# Aufruf mit einem positionalem Argument
Wx::Combobox.new (:parent<$win>), ;
```

Ich weiß auch: Tk und viele andere Module kennen heute bereits benannte Parameter. Die Übergabe eines anonymen Hashes hat zumindest optische Ähnlichkeiten, Typen und Anzahl der Geforderten Parameter werden dabei nicht geprüft. In Perl 6 könnte man aber auch eine Routine mit einem Hash aufrufen. Damit Perl die Paare des Hashes als Name und Wert benannter Parameter wertet, muss der mit einem senkrechten Strich dereferenziert werden.

```
Wx::Combobox.new( |%default );
```

"|" ist keine Sigil für einen Datentyp, so wie ein "&" für Codereferenzen steht, es dient lediglich zum Interpolieren in den Capture-Kontext, vergleichbar mit "@@", dass für den bereits behandelten slice- oder auch multislice-Kontext steht.

Was zum $@%& sind Capture?

Von allen Neuheiten in Perl 6 fordern *Capture* wohl am stärksten die Fähigkeit, sich neue Nervenverbindungen wachsen zu lassen, denn soweit mir bekannt ist, gibt es nichts vergleichbares in anderen Sprachen. Ein Capture ist ein Datentyp, der einem Skalar zugewiesen wird und alle Parameter eines Routinenaufrufs speichern kann. Da Signaturen sowohl positionale als auch benannte Parameter haben können, erscheinen Captures anfangs als seltsame Hybride aus Array und Hash. Nur anders als die Letztgenannten kann eine Capture nicht nachträglich verändert werden. Sie ist "immutable" wie eine Liste. Weil es jetzt keine Referenzen mehr gibt, wurde den Captures der "\" vererbt, der von nun an "capture composer" heißt.

```
my (@a, $b, %c) = [1 .. 5], 6, {
                        'sonnen' => 'schein'};
$capture = \(@a, $b, %c);
```

Diese Capture enthält keine Referenzen auf die Variablen sondern nur die Inhalte.

Anstatt zu referenzieren, kann man in Perl 6 einen Alias auf eine Variable in der Symboltabelle erstellen. Dies geht mit einer sehr einfachen Syntax und gänzlich ohne Typeglobs.

```
$alias := $kathete;
$kathete = 5;
say $alias;          # ist 5
# bindet während Kompilierung
$alias ::= $kathete;
```

Multi Subs

Erinnern wir uns an das erste Beispiel. Wollte man eine Routine schreiben, die die Länge einer beliebigen Seite des rechtwinkligen Dreiecks berechnet, würden die bisher vorgestellten Mittel nicht ausreichen. Mit benannten Parametern wäre die richtige Zuordnung der Seiten gesichert, aber nicht die Forderung, dass 2 von 3 gegeben sein müssen. Eine Lösung bestünde darin das Problem aufzuteilen, was Perl völlig neue Möglichkeiten eröffnet:

```
multi sub pythagoras (:$kathete!,
  :$kathete!) {
      sqrt(@kathete[0]**2 + @kathete[1]**2);
}

multi sub pythagoras (:$kathete!,
  :$hypotenuse!) {
      sqrt($hypotenuse**2 - $kathete**2);
}
```

Das Schlüsselwort `multi` kündigt an, dass es mehrere Routinen gleichen Namens gibt. Mit einem `only` könnte man ausschließen, das nachträglich noch eine `multi` zu einem Namen deklariert wird. Das ist aber meist nicht notwendig, da normale `sub` per default "`only`" sind.

Wird die Routine mit `pythagoras( :kathete<3>, :hypotenuse<5> );` aufgerufen, prüft der Interpreter, welche Signatur zu den Parametern passt. Manch einer ahnt es schon. Auch dafür wird intern wie bei `given/when` der "smartmatch" benutzt. Es kann auch sehr praktisch sein, selbst zu überprüfen, ob ein Satz von Parametern bei einer Routine Erfolg gehabt hätte.

```
$capture ~~ &routine.signature;
```

Parameter Traits

Alle bisherigen Paramter konnten in der Routine nicht verändert werden, was meist sinnvoll ist, aber zuweilen unpraktisch. In diesen Fällen können einzelne Parameter als veränderbar (rw steht für read/write) gekennzeichnet werden, was einer "<->" Zuweisung in *Pointy-Blocks* entspricht.

```
sub incr (*@vars is rw) { $_++ for @vars }
```

Der Befehl `is` definiert *Traits* (Charakteristiken) von Variablen. Das sind neben dem Inhalt zusätzliche Werte oder Eigenschaften, die zur Kompilierungszeit Variablen gegeben werden können. Im Gegensatz dazu werden mit `but` *Properties* (zusätzliche Laufzeiteigenschaften) bestimmt. Somit wird der alte Perl 5-Witz "0 but True" lauffähiger Code.

Eine andere Möglichkeit veränderbare Parameter zu erhalten, ist der Trait `copy`. Wie der Name aussagt, sind solche Parameter veränderbare Kopien der übermittelten Variable.

Eingewickelte Routinen

Es gibt sogar Situationen, da muss man eine Signatur rückwirkend anpassen. Unsere großartige `hypotenuse-sub` könnte Teil eine Matematik-Bibliothek sein, die wir benutzen wollen. Sie kann sogar die Hypotenuse berechnen, wenn ein Winkel und die gegenüberliegende Seitenlänge gegeben ist. Nur leider rechnet sie mit *gon* (Neugrad) und unser Programm mit *Grad* (Altgrad). Die Bibliothek zu verändern kommt nicht in Frage, da die Patches in jede neue fehlerreduzierte Version der Bibliothek eingepflegt werden müssten. Zum Glück gibt es auch dafür in Perl 6 eine elegante Lösung.

```
sub hypotenuse($l, $winkel) {...}
$handle = &hypotenuse.wrap( {
       callwith( $^l, $^winkel/360*400 )
} );
# funktioniert einwandfrei
hypotenuse(2,20);
&hypotenuse.unwrap($handle);
```

Der letzte Befehl hebt die Umhüllung auf und es können selbstverständlich beliebig viele Umhüllungen stattfinden. Sind irgendwelche andere vor- und nachbereitende Tätigkeiten auszuführen, und die Parameter sollen unverändert an die ursprüngliche Routine weitergereicht werden, kann man statt `callwith` auch `callsame` nehmen. Beide Befehle liefern die Ergebnisse der originalen Routine, die dann noch nach Wunsch nachbereitet werden können.

Rückgabekontext

Nachbearbeitungen werden aber oft vermieden, wenn die Routine auf den Kontext eingeht, in dem sie aufgerufen wird. Das ist meist eine Signatur über alle Variablen, denen das Ergebnis der Routine zugewiesen wird. Diese Signatur erhält man mit dem Befel `caller.want` und man könnte ohne Damian Conways Modul *Contextual::Return* schreiben:

```
given caller.want {
when :($)        {...}    # Skalarkontext
when :(*@)       {...}    # Arraykontext
# Ein lvalue wird erwartet
when :($ is rw) {...}
# 2 Werte werden erwartet
when :($,$)      {...}
       ...
}
```

Für all das gibt es auch noch eine andere Schreibweise.

```
if    want.item      {...}
elsif want.list      {...}
elsif want.void      {...}
elsif want.rw        {...}
```

Dieser Kontexte gibt es noch viele mehr. Auch ist `.want` bei weitem nicht die einzige Methode zur Introspektion, aber ich will hier kein Handbuch schreiben, sondern nur einige Möglichkeiten andeuten. Eine vollständige Auflistung aller Details soll das Tutorial in der Wiki unter http://wiki.perl-community.de/bin/view/Wissensbasis/PerlTafel werden. In diesem Beispiel wäre ein want anstatt `caller.want` ausreichend gewesen, aber würde der return-Befehl innerhalb eines Blocks stehen, wären `caller.want` und `context.want` verschieden.

`return` verlässt immer die innerste umgebende Routine. Wird lediglich gewünscht den Block zu verlassen, empfiehlt sich `leave` zu nehmen. Logischerweise wird nur der Rückgabewert von `return` gegen die Signatur der innersten Routine "gematcht".

Module und Scope

Was wäre Perl ohne Module. Deshalb bietet Perl 6 auch für Modulautoren etliche Verbesserungen zur Vorgängerversion. Da aber in diesem Bereich vieles noch nicht in trockenen Tüchern ist, jetzt nur einige Grundzüge. Mit `package` werden weiterhin Namensräume definiert, für Namensräume mit zusätzlichen Eigenschaften gibt es jetzt `module`. Der Befehl `module Name;` besagt, dass für den Rest der Datei der Namensraum *Name* gilt. Für mehrere Module in einer Datei schreibe man `module Name{ ... }`. So können Namensräume auch verschachtelt werden und mit `my module Name { ... }` Module sogar als lexikalisch lokal bestimmt werden.

Analog dazu dürfen auch Subroutinen jetzt mit `my` lokal sein. Standardmäßig entspricht aber weiterhin ein `sub routine {...}` einem `our sub routine {...}`.

Eine der Hauptfähigkeiten von Modulen ist das Exportieren von Routinen. Dazu benötigt man kein `use Exporter;` mehr, sondern markiert die entsprechenden Routinen mit einer *Trait* als `is export`. `sub`-Traits haben ihre Position nach der Signatur. Module werden wie bekannt mit `use` oder `require` geladen, jedoch wurden auch diese Befehle wesentlich mächtiger, um einige Probleme zu lösen, die ein wachsendes CPAN mit sich bringt.

```
use Dog:<1.2.1>;
```

So fordert man z.B. eine spezielle Version an. Noch genauer wäre:

```
# bitte keine Version 1.2.7
use Dog:ver(1.2.1..^1.2.7);
```

Auch ein optionaler Mechanismus zur Authentifizieren von Autoren ist im Syntax vorgesehen, jedoch noch nicht voll ausgereift.

Namensräume mit weitaus mehr Eigenschaften werden mit `class` erzeugt. Die OOP wird aber Stoff der nächsten Folge sein.

Herbert Breunung

Summer of Code und Perl war dabei

Das mag selbstverständlich klingen, ist es leider nicht immer. Denn voriges Jahr fehlte die Perl Foundation in der immer länger werdenden Liste von Organisationen, die entscheiden dürfen, welche Projekte für einen Sommer gefördert werden. Diese Hilfe besteht aus klugen Ratschlägen von Mentoren, die ebenfalls von derjenigen Organisation bestimmt werden und aus bis zu 5000 US$ je Projekt, die Google Inc. stiftet. Googles Nutzen dürfte der Kontakt zu motivierten Programmierern und vor allem die gute PR sein. Denn seit ihrem Start in 2005 verschafft diese Kampagne Google jedes Jahr mehrere vollends positive Schlagzeilen über Studenten die in die "Open-Source-Welt" eingeführt werden und freien Code für alle schreiben. Der Name ist dabei eine Anspielung an den "Summer of Love" 1967, der ebenfalls in den Medien sehr präsent war, eher von emotionaler und politischer Freiheit für alle handelte aber ebenfalls dort sein Zentrum hatte, wo Googles Konzernzentrale heute steht.

wxCPANPLUS

Perls Nutzen ist dagegen ein ganz praktischer, da durch den Anreiz auf fachlicher und monetärer Ebene eine ganze Menge Akkordarbeit erledigt werden kann, für die sich bisher keiner begeistern konnte. Darüber hinaus ist es auch gute PR von "G" unterstützt zu werden. Deswegen waren einige Perl-Programmierern überhaupt nicht davon angetan, dass Perl 2007 nicht vertreten war. Damit sich dieses Versäumnis nicht wiederholt, fasste sich Eric Wilhelm ein Herz und begann sehr aktiv zu werden.

Schnell bekam er auch den Segen der TPF, in ihrem Namen auftreten zu dürfen. Und so brauchte er sich nur noch um den Papierkram kümmern, Studenten mit passenden Ideen finden, helfen die Anträge zu schreiben, Mentoren finden und darüber wachen dass alles nach Plan verläuft und jeder Teilnehmer jederzeit weiß was zu tun ist.

Wirkliches Geschick als Organisator bewies Eric, in dem er so viel wie möglich von dieser Arbeit an Andere verteilte und jedem Projekt zudem einen sogenannten Piloten zuteilte, der über den Fortschritt und Einhaltung der Fristen wacht. Lautstark warb er deshalb an den bekannten Plätzen wie perlmonks.org, use.perl, Mailinglisten und IRC-Kanälen um Interessenten. Dabei wies er zunächst niemand ab um im Falle eines Ausfalls genügend Reserve zu haben. Weil einer der Anträge mit WxPerl zu tun hatte und ich Eric schon aus dem IRC kannte, fragte er auch mich im #wxperl-Kanal.

Mit einem unbedachten "ja" wurde ich so Mentor beim "Google Summer of Code". Das Projekt hieß wxCPANPLUS und ich hatte anfangs starke Bedenken, da ich nicht viel über die CPAN-Shell wusste. Doch auch das löste Eric clever, indem er diesem Projekt Jos Boumans, Autor von CPANPLUS höchstselbst, als zweiten Mentor mir zur Seite stellt. Somit beschränkte sich meine Aufgabe vor allem darauf, Samuel Tyler den Einstieg in WxPerl so einfach wie möglich zu machen. Desweiteren versuchte ich ihm im Vorfeld klarzumachen, welche Design-Entscheidungen getroffen werden müssen und welches wohl die besten Werkzeuge sind, diese zu lösen. Danach übernahm Jos und ich hörte von Sam eine Weile nichts mehr.

Als die Applikation langsam Fleisch ansetzte, half ich ihm wieder durch Tests (Bugreports), Vorschläge zu Designfragen ("Sollte dieser Rand nicht etwas breiter sein?") und Vorschlägen zu Codedesign ("Lagern wir das nicht besser in ein Modul aus?"). Auch wenn Samuel begeistert bei der Sache war und ich noch niemanden so schnell habe Wx wirklich begreifen sehen (mich eingeschlossen), machte er natürlich auch den einen oder anderen Anfängerfehler, da er auch mit

Perl noch nicht so vertraut war (Pfade zusammenfügen ohne File::Spec? - NEIN). Hinterher betrachtet wünschte ich mir noch mehr Zeit dafür verwendet zu haben, aber das Ergebnis kann sich auch so sehen lassen.

Ein Großteil der CPANPLUS-API wurd durch die App in eine übersichtliche, visuelle Form gebracht, CPAN wird wahlweise auch unterstützt und sogar ein WxPodviewer ist entstanden, der bald neben CPANPLUS::Shell::Wx als eigenes Modul auch im CPAN erscheinen soll. Das ganze Programm ist aber höchstens beta, da es unter Windows noch abstürzt und ich es auch unter Linux zum Fall bringen kann wenn ich ihm beim Aufbau eines Index die nächsten Befehle gebe. "Skaman" möchte aber über den "GSoC" hinaus dieses Projekt betreuen, denn die Aktion hat ihm sehr viel Spaß bereitet.

Für mich war es auch ein sehr erfreuliches und bereicherndes Erlebnis, dass ich jedem empfehlen kann, der Antrieb dafür verspürt. Eric empfiehlt den willigen Studenten und Mentoren sich jetzt bereits zu melden oder wenigstens die Ideen zu zu sammeln, um nächstes Jahr noch besser vorbereitet zu sein. Und damit ihr einen noch genaueren Eindruck vom "Summer of Code" bekommt, folgt jetzt der Bericht aus der Perspektive von Moritz Lentz, der ebenfalls dieses Jahr Mentor war.

Ein Perl 6 Projekt

Auswahlprozess

Da es mehr Bewerbungen von Studenten als verfügbare Platze gab (etwa zehn ernst zu nehmende Bewerbungen, wie sich nachher herausstellte wurden fünf Projekte von Google unterstützt), musste priorisiert werden. Jeder Mentor oder Backup-Mentor durfte jedem Projekt maximal eine Stimme geben, bald zeichneten sich Favoriten ab. Darüber wurde dann noch auf den entsprechenden Mailinglisten und im IRC diskutiert, bis die Reihenfolge fest stand.

Gegen Ende wurde der Perl Foundation noch ein sechstes Projekt zugeschanzt: ein Student sollte eine Volltextsuche für das CMS *Bricolage* implementieren, da das Bricolage-Projekt wegen verpasster Termine nicht selbst als Mentor-Organisation zugelassen worden war. Leider hat sich der Student nie wieder gemeldet.

Die Rolle der Mentoren

Als Mentor ist man quasi der Manager des Studenten: Man besorgt Zugang zu den source code repositories, dient als Ansprechpartner, lässt sich regelmäßig über den Fortschritt berichten und überwacht die Arbeit des Schützlings, indem man dessen Code liest und testet.

Als weitere Aufgabe muss man dem Studenten die Scheu vor der Community nehmen, falls vorhanden. Viele der Studenten waren vorher kaum in Open-Source-Communites aktiv, da kann es schon ein wenig einschüternd sein an eine Mailinglist zu schreiben, in der Larry Wall und andere Koryphäen mitlesen und sich auch häufig einmischen.

Die Perl 6 Test Suite

Das Ziel "meines" Projekts war, die Perl 6 Test Suite so weit wie möglich aufzuräumen, zu korrigieren und zu erweitern. Adrian Kreher hatte diese Aufgabe üernommen.

Das ist keine leichte Aufgabe, da zum Teil neu geschriebene Tests mit keiner Implementierung funktionieren, und es daher schwer ist festzustellen, ob diese Tests so funktionieren, wie man sich das vorgestellt hat.

Auch sollten vorhandene Tests so modifiziert werden, dass Rakudo (also Perl 6 auf parrot) sie ausführen kann.

Adrian hat diese Aufgaben gut gemeistert, und meine Rolle als Mentor beschränkte sich darauf, seine Änderungen zu überwachen und einen wöchentlichen Statusreport zu fordern.

Alles in allem musste ich viel weniger Zeit in das Projekt investieren als erwartet, und habe viel positives Feedback von "meinem" Studenten und den Autoren der Perl 6-Compiler bekommen. Sollte sich nächstes Jahr die Gelegenheit für ein ähnliches Projekt bieten, werde ich voraussichtlich wieder als Mentor zur Verfügung stehen.

Herbert Breunung, Moritz Lenz

Die unendlichen Tiefen von $_

Perl kennt viele Spezialvariablen (siehe `perldoc perlvar`). Eine sehr praktische davon ist `$_`:

```
for ( 0..10 ){ print }
```

Die Laufvariable der `for`-Schleife ist hier `$_`. Wie der Aufruf von `print` zeigt, kann man viele Perl-eigene Funktionen ohne Parameter aufrufen, wenn man sie auf `$_` anwenden will. Ein paar Beispiele sind in Listing 1 zu sehen.

```
$_ = 'test';
print;
if ( /es/ ){ print 'enthält "es"' }
chomp;
```

Listing 1

Viele Perl-Bücher für Einsteiger behandeln auch irgendwann, wie man Dateien mit Perl ausliest. Dabei wird häufig erstmal einfach nur der Inhalt einer Datei eingelesen und ausgegeben. Die Beispiele sehen meist ungefähr so aus wie Listing 2.

```
open( FH, '<datei.txt' );
while ( <FH> ){
  print $_;
}
close FH;
```

Listing 2

Auf Probleme wie die Verwendung eines globalen Filehandles oder fehlende Fehlerbehandlung möchte ich hier gar nicht eingehen. Hier möchte ich eine Falle solcher `while`-Schleifen im Zusammenhang mit anderweitiger Verwendung von `$_` aufzeigen.

Separat betrachtet stellt der bewusste Code kein großes Problem dar. Stellen wir ihn nun in etwas weiteren Kontext:

```
use warnings;

$_ = 'hallo';

print;
while ( <DATA> ){}
print;

__DATA__
test
```

Listing 3

Auf den ersten Blick scheint klar zu sein, dass hier "hallohallo" ausgegeben wird. Dem ist aber nicht so. Vielmehr erzeugt Perl eine Warnung, dass ein nicht initialisierter Wert ausgegeben wird, und die Ausgabe lautet nur "hallo". Was ist passiert? Bei einer `for`-Schleife wird die `$_`-Variable nur für die Dauer der Schleife als Alias auf das aktuelle Element genutzt. Bei einer solchen `while`-Schleife dagegen wird die Variable überschrieben!

Um das zu vermeiden, kann man mit `local` einer globalen Variable für einen bestimmten Gültigkeitsbereich einen neuen Wert zuweisen. Nach dem Gültigkeitsbereich wird der alte Wert wiederhergestellt (siehe auch `perldoc -f local`). Wer innerhalb einer `while`-Schleife nicht auf die Verwendung von `$_` verzichten möchte, muss sich als um die `local`isierung selbst kümmern:

```
while ( defined( local $_ = <DATA> ) ){}
```

Ändert man das Beispiel entsprechend, wird tatsächlich "hallohallo" ausgegeben.

Dieses Problem ist nicht auf `while` beschränkt. Man sollte generell sehr vorsichtig bei der Verwendung von `$_` ohne `local` sein, da man sich schnell verrückt machen kann:

```
# Datei hauptprogram.pl
use warnings;

$_ = 'hallo';

print;
require MyTest;
print;

# Datei MyTest.pm
package MyTest;


$_ = 123;
```

Was hat das wohl für eine Ausgabe? Es gibt "hallo123" aus, weil beim Laden des Moduls `MyTest` die Variable `$_` überschrieben wird. Da es nur genau ein `$_` gibt (nämlich das globale), wirken sich die Änderungen im Modul auf das Hauptprogramm aus. Auch hier wäre es angebracht gewesen, die Zuweisung zu `$_` innerhalb des `MyTest`-Moduls mit `local` auf den aktuellen Gültigkeitsbereich (der sich vom Anfang bis zum Ende der Datei `MyTest.pm` erstreckt) zu beschränken. Solche Fehler sucht man häufig sehr sehr lange!

Allgemein sollte man sich angewöhnen, *jede* Verwendung einer Perl-Spezialvariablen mit `local` zu schützen.

Renée Bäcker

LESERBRIEF

Zu Charsets Ausgabe 1/2008:

Vielen Dank für diesen Artikel. Leider hat meiner Meinung nach im Beitrag folgendes gefehlt.

Zum Prüfen, in welchem Charset ein unbekannter Text vorliegt, nutze ich

```
CharsetDetector::detect($text);
```

um anschließend mit

```
open IMPORT,"<::encoding($charset)",
                "$importfile";
```

die Datei zu öffnen.

Allerdings erhalte ich noch bei Latin1 Texten das Charset cp936. (Was momentan aber nicht weiter stört)

Nochmals vielen Dank, und gutes Gelingen für die nächsten Jahre.

Mit freundlichen Grüßen

Stefan Kox

Smart-Websolutions

Windolph und Bäcker GbR

Perl-Programmierung

info@smart-websolutions.de

Merkwürdigkeiten in Perl - Die Version eines Moduls herausfinden

Perl-Programmierer, die viel in Foren oder anderen Gruppen unterwegs sind, sehen bei Problemstellungen immer wieder die Frage "Welche Version des Moduls verwendest Du?". Auf die (Gegen-)Frage, wie man denn die Version herausbekommt, gibt es mehrere Antworten, die häufig genannt werden:

* `perl -MModul -e 'print Modul->VERSION''`
* "Schau in Modul.pm nach, was da bei $VERSION steht"
* `perl -MModul=99999 -e 1`

Die ersten beiden Antworten dürften klar sein, aber was macht der Code der dritten Antwort?

Alles was nach dem = steht, ist wie die Importliste, die man bei `use Modul qw(liste)` angibt. Wenn man den Code mit `Storable` ausführt bekommt man so etwas raus:

```
$ perl -MStorable=9999 -e 1
Storable version 9999 required--this is only
   version 2.15 at /usr/lib/perl5/5.8.3/
   Exporter/Heavy.pm line 121.
BEGIN failed--compilation aborted.
$
```

Wunderbar, hier erkennt man, dass ich im Moment Storable mit der Version 2.15 verwende. Mit der "9999" gebe ich an, dass ich Storable mindestens in Version 9999 verwenden will. Perl bricht ab, wenn es das Modul nicht in der geforderten Version laden kann.

Das gleiche passiert auch, wenn ich im Programm `use Modul 9999` mache.

Alles klar, oder?

Naja, diese Methode funktioniert leider nicht bei allen Modulen. Wenn ich das gleiche mit CGI mache, dann passiert folgendes:

```
$ perl -MCGI=9999 -e 1
$
```

Was? CGI.pm gibt es doch erst in Version 3.42 (Stand August 2008) und das ist weit unter 9999. Warum gibt Perl hier keine Fehlermeldung aus?

Um das zu beantworten muss man wissen wonach man sucht. Der Unterschied liegt darin, dass Storable das Modul Exporter verwendet und CGI.pm nicht. Dadurch verändert sich das Verhalten der `import`-Methode.

`perl` macht aus `-MModul=9999` ein `use Modul "9999"` und alles was an `import` geht, wird als String betrachtet - egal ob es nur Zahlen oder auch andere Zeichen sind. Exporter schaut sich genauer an, was importiert werden soll. Besteht das Argument nur aus Zahlen, dann wird die Versionsüberprüfung gemacht, so als ob `use Modul 9999` gemacht wurde.

Das sieht im ersten Moment nach einem Bug in `perl` aus, ist aber eher ein Fehler in Exporter, weil das Verhalten von `perl` so dokumentiert ist.

Man kann das ganze aber dennoch so kurz machen:

```
perl "-MCGI\ 999" -e 1
```

Renée Bäcker

Aarau.pm - Es Grüezi us de Schwiiz

Keine Angst, ab hier versuche ich deutsch zu schreiben.

Wie ihr bei diesem Titel erraten könnt, handelt es sich bei uns um eine Schweizer Perlmonger-Gruppe. Gegründet haben wir diese Gruppe im September 2004, bei einem kräftigen Schluck "Suure Moscht", in Aarau. Aarau liegt im Norden des Schweizer Mittellandes an der schönen, grünen Aare, wo wir uns (in der Stadt und nicht an der Aare) im Restaurant Waage treffen.

Unsere Treffen finden am Sonntagnachmittag statt, sofern sich mindestens drei bis vier dazu entschließen können. Wir organisieren uns über IRC: irc.freenode.net im Channel: #linux.ch oder über eMail an: aarau-pm@pm.org.

Da wir eine kleine Gruppe von nur 6 Leuten sind, hängt es davon ab, wie aktiv jeder einzelne ist. Wenn also zwei von uns ein halbes Jahr in die Rekrutenschule eingezogen werden, dann ist das schon ein harter Schlag für unsere kleine Gruppe.

Die meisten von uns stammen aus den Regionen um Aarau: Baden, Seetal und Olten. Soweit mir bekannt ist, sind wir die einzige aktive Perlmonger-Gruppe in der Deutschschweiz. Von Zurich.pm (mit Steven Schubiger) gibt es eine Mailingliste.

Bei uns sind auch Programmierer anderer Programmiersprachen willkommen. Sollte sich also ein unverstandener C++, ein belächelter Java oder ein vereinsamter Cobol-Programmierer mit anderen treffen wollen, dann kann er gerne bei uns vorbeischauen. Was wären wir für eine Schweizer Perlmonger-Gruppe, wenn wir nicht zur Mehrsprachigkeit stehen würden?

Eine kleine Gruppe zu sein, heißt noch lange nicht, dass wir nicht an verschiedenen Themen interessiert sind. Da wären Perl, modperl, C, C++, XML, CORBA, Datenbanken, SQL, PL/SQL, Systemmanagement, Netzwerk, Security etc. Zu diesen Themen wurde schon etwas vorgestellt, darüber diskutiert oder es kennt sich einer von uns damit aus.

Es sind also alle herzlich eingeladen sich bei uns zu melden und mitzumachen.

Urs Stotz

CPAN News VIII

Sub::Called

Mit `Sub::Called` kann man verschiedene Sachen über Subroutinen "herausfinden": ob die Subroutine schon einmal aufgerufen wurde oder nicht, oder ob es mit einem vorangestellten "&" aufgerufen wurde:

```
#!/usr/bin/perl

use Sub::Called qw(already_called);

sub foo {
  if( already_called() ){
      print "Sub foo wurde schonmal
             aufgerufen";
    }

  if( Sub::Called::with_ampersand() ){
      print "foo wurde als &foo
             aufgerufen";
   }
}

&foo();
```

Business::Tax::VAT::Validation

Im B2B-Bereich ist es oft wichtig die Umsatzsteuer-ID des Gegenübers zu kennen und zu prüfen. Dieses Perl-Modul bietet zwei Möglichkeiten der Überprüfung: Eine mit Regulären Ausdrücken und eine fragt die VIES-Datenbank ab. Für letzteres ist jedoch eine Verbindung zum Internet nötig.

```
#!/usr/bin/perl

use strict;
use warnings;

use lib qw(./lib);
use Business::Tax::VAT::Validation;

my $validator = Business::Tax::VAT::
Validation->new;
my $vat       = 'DE228935695';

if( $validator->check( $vat, 'DE' ) ){
     print "USt-ID entspricht den Regeln";
  }
  else{
     print $validator->get_last_error;
  }
```

Time::Stats

Was bremst meine Anwendung aus? Ist diese Subroutine wirklich so langsam? Auf der Suche nach dem Flaschenhals in der Anwendung benötigt man Angaben über die Laufzeiten. Mit Time::Stats ist es sehr einfach solche Angaben zu bekommen:

```
#!/usr/bin/perl

use Time::Stats qw(mark stats);

mark();
sleep 5;
mark();

stats();
```

Der erste Aufruf von `mark()` markiert die Zeile ab der die Geschwindigkeit gemessen werden soll, der zweite Aufruf bedeutet das Ende der Zeitmessung. Dieses Skript erzeugt folgende Ausgabe:

```
C:\>stats.pl
File: C:\\stats.pl
Lines 5 to 7: 5.000768
```

Test::Kit

Test::Kit erleichtert die Verwendung von Test-Modulen, die jedoch installiert sein müssen. Damit der Namensraum aber nicht "zugemüllt" wird, kann man einzelne Module einschalten und gegebenenfalls auch Funktionen umbenennen.

```
#!/usr/bin/perl

use Test::Kit qw(
    Test::More
);

plan tests => 1;
is(1,1);
```

Jetzt stehen aber die Funktionalitäten aus den anderen Modulen nicht zur Verfügung. Sollen auch noch Funktionen aus Test::Pod verwendet werden, aber die Funktion `pod_file_ok` soll einfach `pod_ok` heißen, dann geht das so:

```
#!/usr/bin/perl

use Test::Kit (
  'Test::More',
  Test::Pod => {
     rename => {
        pod_file_ok => 'pod_ok',
     }
  }
);

plan tests => 2;
is(1,1);

pod_ok( $0 );
```

ex::lib

Häufig verwendet man eigene Module, die nicht in `@INC` liegen, sondern in einem Verzeichnis, das direkt für das Projekt angelegt wurde. Dann sieht man im Code etwas wie `use lib qw(./lib)`. Das kann unter bestimmten Bedingungen problematisch sein; z.B. gibt es Fälle mit dem IIS von Microsoft oder unter mod_perl, in denen das Skript nicht aus dem Verzeichnis des Skriptes heraus gestartet wird. Dann funktionieren relative Pfade nicht mehr. `ex::lib` Verschafft da Abhilfe. Es macht aus "./lib""einen absoluten Pfad.

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

use ex::lib qw(./lm);

print $_ for @INC;
```

Path::Abstract

`Path::Abstract` ist ein Modul, mit dem man Pfadnamen zusammensetzen und durch den Pfad "navigieren" kann. Im Gegensatz zu `File::Spec` arbeitet es aber nicht mit OS-spezifischen Pfadtrennern, sondern nur mit UNIX-style "/". Der Vorteil gegenüber `File::Spec` besteht darin, dass man immer ein Objekt hat, mit dem man Operationen durchführen kann und nicht immer wieder den Pfad übergeben muss.

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

use strict;
use warnings;
use Path::Abstract;

my $path = Path::Abstract->
       new( "thisdir","temp" );
print "Temp dir is ", $path->stringify;

$path->child( "test" );
print $path->stringify;

if( ! $path->is_root ){
   $path->to_tree;
}

print $path->stringify;
```

Termine

November 2008

01. 3. Russischer Perlworkshop
03. Treffen Vienna.pm
04. Treffen Frankfurt.pm
06. Treffen Dresden.pm
17. Treffen Erlangen.pm
20. Treffen Darmstadt.pm
25. Treffen Bielefeld.pm

Dezember 2008

01. Treffen Vienna.pm
02. Treffen Frankfurt.pm
04. Treffen Dresden.pm
15. Treffen Erlangen.pm
18. Treffen Darmstadt.pm
20. 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

Januar 2009

06. Treffen Vienna.pm
07. Treffen Bielefeld.pm

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 11. Deutsche Perl-Workshop findet 2009 in Frankfurt 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

BESSERE ATMOSPHÄRE? MEHR FREIRAUM?

Wir suchen erfahrene Perl-Programmierer/innen (Vollzeit)

//SEIBERT/MEDIA besteht aus den vier Kompetenzfeldern Consulting, Design, Technologies und Systems und gehört zu den erfahrenen und professionellen Multimedia-Agenturen in Deutschland. Wir entwickeln seit 1996 mit heute knapp 60 Mitarbeitern Intranets, Extranet-Systeme, Web-Portale aber auch klassische Internet-Seiten. Seit 2005 konzipiert unsere Designabteilung hochwertige Unternehmensauftritte. Beratungen 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 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

„Statt mit blumigen Worten umschreiben unsere Programmierer den Job so:

Apache, Catalyst, CGI, DBI, JSON, Log::Log4Perl, mod_perl, SOAP::Lite, XML::LibXML, YAML“

Warum Fachleute auf Schulungen gehen sollten

Externe Schulungen sind wie eine Studienfahrt mit Fremden, die am gleichen Thema arbeiten. 5 Tage lang 'mal nicht mit den eigenen Kollegen im immer gleichen Brei schwimmen. Es ist nämlich nicht richtig, alles im Selbststudium lernen zu wollen: Jede(r) von uns hat die Fundamente seines Könnens in Schulen und Universitäten gelernt, und gerade wer im Betrieb stark belastet ist, hat keine Chance, schwierige Themen „nebenbei“ am Arbeitsplatz zu erlernen oder neue Kollegen anzulernen.

Schauen Sie 'mal: www.Linuxhotel.de

Wer sich wirklich intensiv auf eine Arbeit einläßt, will auch Spaß dabei haben! Im Linuxhotel kombinieren wir deshalb ganz offen eine äußerst schöne und luxuriöse Umgebung mit höchst intensiven Schulungen, die oft bis in die späten Abendstunden zwanglos weitergehen. Natürlich freiwillig! Wer will, zieht sich zurück! Und weil unser Luxushotel ganz weitgehend per Selbstbedienung läuft, sind wir gleichzeitig auch noch sehr preiswert.