Renée Bäcker

RegEx Module - Zahme Regenechsen...

Reguläre Ausdrücke (RegEx) zählen eindeutig zu den Stärken von Perl und man landet doch recht häufig bei diesem Mittel. Die RegEx können dabei recht verwirrend sein.

In diesem Artikel werden einige Module im Zusammenhang mit Regulären Ausdrücken gezeigt. Das geht von Modulen, die dem Programmierer die Zusammenstellung der RegEx abnehmen bis hin zu Modulen, die bei der "Analyse" von Regulären Ausdrücken helfen.

Heinzelmännchen für Programmierer

Regexp::Common

Eine große Sammlung von "alltäglichen" Regulären Ausdrücken wie IP-Adressen, Integer-Zahlen und anderes ist in Regexp::Common vereint. Durch einen einfachen Mechanismus können auch Plugins recht schnell geschrieben werden. Der Vorteil der Sammlung gegenüber selbstgeschriebenen Ausdrücken liegt darin, dass `Regexp::Common` schon viel getestet wurde.

Die Verwendung von `Regexp::Common` ist sehr einfach. Die Regulären Ausdrücke sind "hierarchisch" in einem Hash abgelegt.

So sind unterhalb von `$RE{num}` alle Ausdrücke, die Zahlen behandeln, zu finden. Mit `$RE{num}{int}` bekommt man den Regulären Ausdruck für Integerzahlen.

Ein kleines Beispiel, in dem Dezimalzahlen gesucht werden, ist in Listing 1 zu sehen.

Regexp::Assemble

1000 einzelne Regexes, doch wie kann man das zusammenfassen? Die Lösung heißt `Regexp::Assemble`. Mit diesem Modul können mehrere einzelne Reguläre Ausdrück mit "ODER‘ verknüpft werden. Das Modul macht dabei keine sture Aneinanderreihung der Teile, sondern versucht "intelligent" Schnittmengen zu finden und das Resultat als kompakten Ausdruck darzustellen.

Wenn in einem String überprüft werden soll, ob er die Worte "Perl" oder "Pelle" enthält, dann sieht das Skript so aus (Listing 2):

```
#!/usr/bin/perl

use strict;
use warnings;
use Regexp::Common;

my @zahlen = qw(1.000 0.5 3 15 15e21);

for my $val ( @zahlen ){
    if( $val =~ /^$RE{num}{int}$/ ){
        print $val, ": yes\n";
    }
}
```

Listing 1

```
#!/usr/bin/perl

use Regexp::Assemble;

my @words = qw(Perl Pelle);
my $string = "Perl ist toll";

my $re = Regexp::Assemble->new;
$re->add($_) for @words;

print "yes\n" if $string =~ /$re/;
```

Listing 2

Der Reguläre Ausdruck, der von `Regexp::Assemble` erzeugt wird, sieht so aus: `(?-xism:Pe(?:lle|rl))`

Leider kann das Modul keine 'UND' Verknüpfung, aber es ist trotzdem sehr hilfreich. Auch Buchstaben-Bereiche werden von dem Modul nicht erkannt. Füttert man Regexp::Assemble einzeln mit den Ziffern 1 bis 9, so macht es daraus den RegEx `(?-xism:[123456789])` statt `(?-xism:[1-9])`.

Ein weiteres Modul in diese Richtung ist `Regex::PreSuf`

Regexp::MatchContext

Gern verwendete Variablen bei Regulären Ausdrücke sind `$``, `$&` und `$'` beziehungsweise deren langnamigen Pendants `$PREMATCH`, `$MATCH` und `$POSTMATCH`. Diese Variablen sind hilfreich, aber sie haben einen ganz großen Nachteil: Die Regulären Ausdrücke werden extrem langsam. Und das wirkt sich nicht nur auf den Regulären Ausdruck aus, in dem diese Variablen verwendet werden, sondern auf **alle** RegEx im Programmlauf.

```
$ cat regex.pl
#!/usr/bin/perl
use strict;
use warnings;
my $x = $&;
my $string = "123" x 1000;
for (0..100000) {
    if ($string =~ m/3/) {
        my $x = "matched";
    }
}
$ time perl regex.pl

real    0m0.108s
user    0m0.100s
sys     0m0.008s
$ time perl regex.pl

real    0m0.107s
user    0m0.104s
sys     0m0.000s
```

Listing 3

```
$ time perl regex.pl

real    0m0.056s
user    0m0.052s
sys     0m0.004s
$ time perl regex.pl

real    0m0.055s
user    0m0.052s
sys     0m0.004s
```

Listing 4

Tina Müller hat mal mit einem ganz einfachen Skript die Zeitunterschiede verdeutlicht (Listing 3), danach wurde die Zeile `my $x = $&;` auskommentiert und die Aufrufe wiederholt (siehe Listing 4).

So richtig benchmarken kann man das Problem nicht, da die Auswirkungen auf die Laufzeit sehr stark vom Programm abhängen. Ein Programm mit sehr vielen Regulären Ausdrücken hat natürlich wesentlich größere Laufzeitnachteile als ein Programm mit sehr wenigen RegEx.

Damian Conway hat ein Modul geschrieben, das diesen Geschwindigkeitsnachteil zumindest auf den einen Regulären Ausdruck begrenzt. Dazu muss in dem Regulären Ausdruck der Modifier (?p) verwendet werden. Ein Beispiel dazu ist in Listing 5 zu sehen. Mit der Benutzung des Moduls gibt es erst einen Zeitvorteil wenn mehrere Reguläre Ausdrücke verwendet werden, von denen nicht alle diese "Kontext"-Variablen verwenden.

In Perl 5.10 braucht man das zusätzliche Modul nicht mehr. Yves Orton hat dort den p-Modifier in die RegEx-Engine eingebaut, die ähnlich wie Conways Modul bestimmte Variablen nur für den Regulären Ausdruck setzt, der das ausdrücklich anfordert (siehe auch $foo "Sommer 2007").

Was passiert da eigentlich?

YAPE::Regexp::Explain

Reguläre Ausdrück sind teilweise verdammt schwer zu verstehen. Gerade Perl-Einsteiger schauen häufiger mal mit drei Fragezeichen im Gesicht auf ein Perl-Programm. "Was macht denn das da? Das ist ja total kryptisch." bekommt man dann

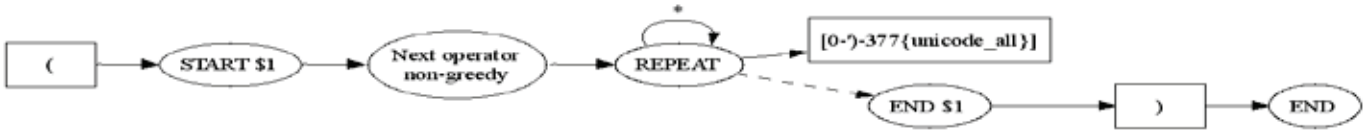
```
#!/usr/bin/perl

use strict;
use warnings;
use Regexp::MatchContext -vars;

my $string = 'Dies ist ein langer Text';
$string =~ /(?p)lang/;

print qq~
PREMATCH:   $PREMATCH
MATCH:      $MATCH
POSTMATCH:  $POSTMATCH
~;
```

Listing 5



Aber das Modul hilft natürlich nicht nur den Einsteigern sondern auch "alten Hasen". Ein einfaches Beispielprogramm ist in Listing 6 dargestellt

In der Ausgabe des Programms (siehe Listing 7) erkennt man, wie das Modul die einzelnen Teile des Regulären Ausdrucks erklärt. Dadurch ist es auch sehr gut dazu geeignet, die verschiedenen Konstrukte wie `(?:)` kennenzulernen.

Durch die Einrückung der Erklärungen ist gut ersichtlich, zu welchem Klammerpaar ein Teil des Ausdrucks gehört.

GraphViz::Regex

Eine graphische Ausgabe kann mit dem Modul `GraphViz::Regex` von Leon Brocard erzeugt werden. Das ist natürlich vor allem für Leute interessant, die in der Schule oder sonstwo schon etwas von "Automaten" gehört haben.

```
#!/usr/bin/perl

use strict;
use YAPE::Regex::Explain;

my $re = '\(([^\(]*?)\)';
print YAPE::Regex::Explain->new($re)-
>explain;
```

Listing 6

Das Modul versucht nämlich aus dem Regulären Ausdruck einen Automaten zu erzeugen. Somit kann man auch "per Hand" versuchen, zu einem gegebenen String nachzuvollziehen warum der RegEx einen Treffer erzeugt oder eben auch nicht. So kann man auch Personen, die mit Regulären Ausdrücken nicht unbedingt etwas zu tun haben, die Ergebnisse näherbringen.

Wer sich gut mit Interna auskennt, kann auch den in Perl eingebauten RegEx-Debugger verwenden. Doch sehr vielen wird die Ausgabe nur wenig sagen. Der RegEx-Debugger ist wieder ein Thema für sich.

```
#!/usr/bin/perl

use strict;
use warnings;
use GraphViz::Regex;

my $re = '\(([^\(]*?)\)';
my $graph = GraphViz::Regex->new( $re );

my $output = 'regex.png';
open my $fh, '>', $output or die $!;
binmode $fh;
print $fh $graph->as_png;
close $fh;
```

Listing 8

```
C:\Perl\FooMagazin>explain.pl
The regular expression:

(?-imsx:\(([^\(]*?)\))

matches as follows:

NODE            EXPLANATION
----------------------------------------------------------------
(?-imsx:        group, but do not capture (case-sensitive)
                (with ^ and $ matching normally) (with . not
                matching \n) (matching whitespace and #
                normally):
----------------------------------------------------------------
  \(              '('
----------------------------------------------------------------
   (               group and capture to \1:
----------------------------------------------------------------
    [^\(]*?         any character except: '\(' (0 or more
                    times (matching the least amount
                    possible))
----------------------------------------------------------------
  )               end of \1
----------------------------------------------------------------
  \)              ')'
----------------------------------------------------------------
  )               end of grouping
  ----------------------------------------------------------------
```

Listing 7