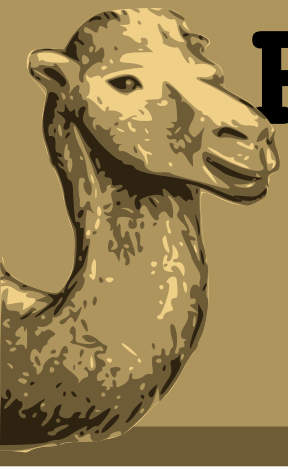
Perl Mongers im Ruhrgebiet



Ruhr.pm

Perl Source Filters

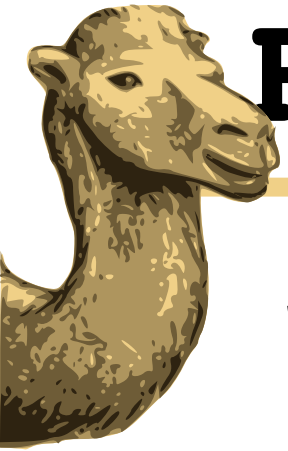
Eine praktische Einfuehrung mit Filter::Simple

Autor: Veit Wahlich

EMail: veit AT ruhr.pm.org

Datum: 13. Oktober 2008

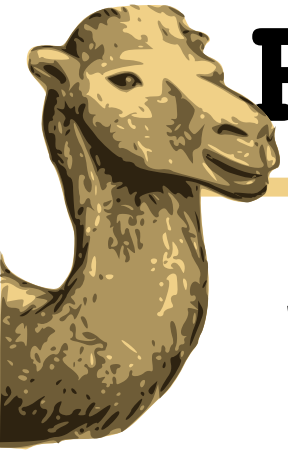
http://ruhr.pm.org/



Ruhr.pm

Was sind Perl Source Filters?

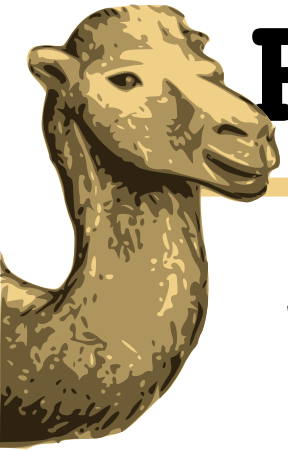
- ermoeglichen die Anpassung oder Erweiterung der Perl-Syntax
- erlauben komplette Umstrukturierung, Ersetzung oder Verschluesselung des Quellcodes
- Basis vieler Acme::*- sowie Lingua::*-Module im CPAN
- Programme, die Source-Filter verwenden, lassen sich leider nicht kompilieren
  - weder mit Bytecode- noch C-Backend



Ruhr.pm

Was ist Filter::Simple?

- ein einfaches Interface zu Perl Source Filters
  - stellt einige Zusatzfunktionalitaet zur Verfuegung
    - erlaubt z.B. das Filtern nach “Art” des Quellcodes
      - Code ohne POD und Datensektion
      - Code ohne POD, Datensektion und Kommentare
      - Code ohne POD, Datensektion und Quotelikes
      - Code ohne POD, Datensektion, Kommentare und Quotelikes
      - nur Quotelikes
      - nur Regexe
      - nur Strings



Ruhr.pm

Was ist Filter::Simple?

- erlaubt das Schreiben besonders kurzer Filter
  - oft nicht mehr als ein einziger Regex
- dennoch volle Flexibilitaet
- Ausfuehrung in Reihenfolge der **`use`**-Statements
- viele Filter lassen sich mit **`no`** wieder deaktivieren
- mittlerweile als Teil der meisten Perl-Distributionen bereits enthalten
- entwickelt von Damian Conway

Ruhr.pm

Erweiterung der Syntax

- beispielsweise durch
  - neue Operatoren
  - neue Statements oder Konstrukte
  - Einbetten fremder Sprachelemente direkt in den Quellcode
    - z.B. Perl6-Code, XML-Daten oder SQL-Statements
- haeufige Anwendung: Switch
  - ergaenzt die Perl-Syntax um switch()-case-Konstrukte
  - enthalten in den meisten Perl-Distributionen

Ruhr.pm

Erweiterung der Syntax

```
#!/usr/bin/perl

use strict;
use warnings;



use Filter::MultilineComments;

# Wir sind hier doch nicht in Bayern!
#{
Oh nein! Ein Block-Kommentar!
my $a = '  Franz faehrt im komplett '
  .'verwahrlosten Taxi quer durch Bayern  ';
}#

# So ist es besser:
my $a = '  Franz faehrt im komplett '
  .'verwahrlosten Taxi quer durch den '
  .'Ruhrpott  ';
```

Perl kennt von Hause aus lediglich Zeilen-Kommentare. Moechte man einen ganzen Programmabschnitt auskommentieren, vermisst man schnell Block-Kommentare, wie sie z.B. in C ueblich sind (`/*...*/`).

`Filter::MultilineComments` implementiert dieses Sprachelement. Da `/.../` in Perl bereits verwendet wird, greifen wir zum Einleiten des Kommentars auf **`#{`** und zum Schliessen auf **`}#`** zurueck.

Erweiterung der Syntax

```
use Filter::VariableModifiers;

my @data = (
    ['Original',      $a   ],
    ['Grossschrift',  $^a  ],
    ['Kleinschrift',  $_a  ],
    ['Gestutzt',      $/a  ],
    ['Kapitaelchen',  $=a  ],
    ['Rueckwaerts',   $<a  ],
    ['Kombinationen', $/^<a]
  );
```

Filter::VariableModifiers implementiert Modifikatoren fuer String-Skalare. Diese erlauben es, beim Zugriff auf die Variable den Inhalt in einer vom gespeicherten Wert abweichenden Darstellung zu erhalten.

Wird zwischen **$** und Variablennamen ein **^** gesetzt, ist der Wert gross geschrieben, durch _ klein, mit **/** wird getrimmt, **=** setzt den String in Kapitaelchen und **<** gibt ihn rueckwaerts zurueck. Alle Modifikatoren sind kombinierbar.

Ruhr.pm

Erweiterung der Syntax

```
use Filter::EmbedHTML;

# Hier regulaerer Perl-Code.

%>
<!-- Hier beginnt HTML-Code. -->
<html>
  <head>
    <title>Beispiele: Filter::Simple</title>
  </head>
  <body>
    <table>
<% foreach(@data){ %>
      <tr>
        <td><% print($_->[0]) %>: </td>
        <td><% print($_->[1]) %></td>
      </tr>
<% } %>
    </table>
  </body>
</html>
<%

# Und hier ist wieder reiner Perl-Code.
```

`Filter::EmbedHTML` erlaubt es, HTML-Code direkt in den Perl-Code einzubetten. Hier wird einfach in ein Grundgeruest ausgegeben sowie in einer Schleife die Ausgabe des Arrays **`@data`**, wobei jedes Element eine Zeile mit jeweils 2 Zellen in der Tabelle ergibt.

HTML-Code wird zwischen **`%>`** und **`<%`** definiert, anders herum gesehen laesst sich Perl-Code zwischen **`<%`** und **`%>`** schreiben.

Ruhr.pm

Funktioniert das auch?

```
$ ./example.pl

<!-- Hier beginnt HTML-Code. -->
<html>
  <head>
    <title>Beispiele mit Filter::Simple</title>
  </head>
  <body>
    <table>

      <tr>
        <td>Original: </td>
        <td>  Franz faehrt im komplett verwahrlosten Taxi quer durch den Ruhrpott  </td>
      </tr>

      <tr>
        <td>Grossschrift: </td>
        <td>  FRANZ FAEHRT IM KOMPLETT VERWAHRLOSTEN TAXI QUER DURCH DEN RUHRPOTT  </td>
      </tr>

      <tr>
        <td>Kleinschrift: </td>
        <td>  franz faehrt im komplett verwahrlosten taxi quer durch den ruhrpott  </td>
      </tr>
```

Ruhr.pm

Funktioniert das auch?

```
        <tr>
          <td>Gestutzt: </td>
          <td>Franz faehrt im komplett verwahrlosten Taxi quer durch den Ruhrpott</td>
        </tr>

        <tr>
          <td>Kapitaelchen: </td>
          <td>  Franz Faehrt Im Komplett Verwahrlosten Taxi Quer Durch Den Ruhrpott  </td>
        </tr>

        <tr>
          <td>Rueckwaerts: </td>
          <td>  ttoprhuR ned hcrud reuq ixaT netsolrhawrev ttelpmok mi trheaf znarF  </td>
        </tr>

        <tr>
          <td>Kombinationen: </td>
          <td>TTOPRHUR NED HCRUD REUQ IXAT NETSOLRHAWREV TTELPMOK MI TRHEAF ZNARF</td>
        </tr>

      </table>
    </body>
  </html>
```

Ruhr.pm

Filter::MultilineComments

```
package Filter::MultilineComments;
use Filter::Simple;

FILTER_ONLY(code => sub{s|#\{.*?\}#||gs});
```

Unser Filter-Modul **`Filter::MultilineComments`**.

Filtere ausschliesslich Perl-Code und ersetze dabei alle Vorkommnisse von **`#{...}#`** durch nichts, betrachte dabei die gesamte Eingabe als eine einzige Zeile.

Ruhr.pm

Filter::EmbedHTML

```
package Filter::EmbedHTML;
use Filter::Simple;

FILTER{
    s/%>(.*?)<%
     /";print('".escapeSq($1)."');\n"/xgse;
  };

sub escapeSq($){
        ($_) = @_;
        s/(?=[\\\'])/\\/gs;
        return($_);
}
```

Unser Filter-Modul **`Filter::EmbedHTML`**.

Filtere den gesamten Inhalt der Datei, finde alle Vorkommnisse von %> bis <%, speichere alles dazwischen und ersetze diese Vorkommnisse durch print()-Befehle mit Ausgabe der gespeicherten Werte, diese dabei sauber escapen.

Ruhr.pm

Filter::VariableModifiers

```
package Filter::VariableModifiers;
use Filter::Simple;

FILTER_ONLY(
    code_no_comments => \&modifiers
  );

sub modifiers{
  s/\$([<\/=\^_]+)(?=\w)
   /insertCalls($1).'$'/xge;
  return($_);
}

sub insertCalls($){
  my($mods) = @_;
  my $calls = '';
  foreach(reverse(split(//, $mods))){
    if    (/</)  { $calls .= 'Filter::VariableModifiers::reverseString ' }
    elsif (/\//) { $calls .= 'Filter::VariableModifiers::truncateString ' }
    elsif (/=/)  { $calls .= 'Filter::VariableModifiers::capitalizeString ' }
    elsif (/\^/) { $calls .= 'uc ' }
    elsif (/_/)  { $calls .= 'lc ' }
  }
  return($calls);
}
```

Filtere allen Perl-Code (ausser Kommentare) und ersetze dabei alle **$** gefolgt von min. einem Modifikator-Zeichen (**<**, **/**, **=**, **^** oder _) und einem Word-Zeichen (Variablenname) durch entsprechende Aufrufe von Modifikator-Funktionen auf diese Variable.

Ruhr.pm

Filter::VariableModifiers

```
sub reverseString($){
  return(join('',
    reverse(split(//, $_[0]))));
}


sub truncateString($){
  ($_) = @_;
  s/^\s+//s;
  s/\s+$//s;
  return($_);
}


sub capitalizeString($){
  ($_) = @_;
  s/(^|\s)(\w)
   /(defined($1)?$1:'').uc($2)/xge;
  return($_);
}
```

`reverseString()` spiegelt den uebergebenen String.

`truncateString()` beschneidet den uebergebenen String. Saemtlicher Whitespace am Anfang und Ende wird entfernt.

`capitalizeString()` stellt die Anfangsbuchstaben aller Woerter, d.h. den ersten Buchstaben am Anfang des Strings oder nach Whitespace (Leerzeichen, Tab, ...), in Grossschrift.

Ruhr.pm

Quellcode-Transformation

- beispielsweise
  - “Verschluesselung”/Verschleierung des Quellcodes
  - Programmierung mit Sprachelementen fremder natuerlicher Sprachen, z.B. auf Deutsch statt Englisch
- haeufig verwendet: Lingua::Romana::Perligata
  - Programmierung vollstaendig auf Latein, Syntax quasi ohne nichtalphabetische Zeichen
    - Reihenfolge unwichtig, Bezuege durch Flexion definiert
    - Variablen-Arten unterschieden durch Deklination
    - sogar lateinische Zahlen

Ruhr.pm

Mfpmmmpmfpmfppf Fppmpppmffmp!

```
#!/usr/bin/perl

use Filter::McCormickized;

pfmpffmffpppfmp "Mfpmmmpmfpmfppf “
   .”Pfffmfmfppff.pfmppm!\n";

no Filter::McCormickized;

print “Es geht auch normal!\n”
```

Ein einfaches
`print “Hallo Ruhr.pm!\n”;`
in Kenny-Speak.

Unser Modul
`Filter::McCormickized` fuehrt Programme im Kenny-Code aus.
Alles ab
`use Filter::McCormickized;`
wird als Code in Kenny-Speak interpretiert, mit
`no Filter::McCormickized;`
schaltet man wieder zurueck zu regulaerem Perl-Code.

Ruhr.pm

Filter::McCormickized

```
package Filter::McCormickized;
use Filter::Simple;

my $table = {
    map{
        my $i = $_;
        join('',
            map{qw(m p f)[$i/$_%3]}(9,3,1)
          ), chr(97+$i);
      }(0..25)
  };

map{
    s/(.)/uc($1)/e;
    $table->{$_} = uc($table->{lc($_)})
  } keys(%{$table});

FILTER{
    s/([MPFmpf][mpf]{2})/$table->{$1}/gse;
  };
```

Kenny-Speak ist ein ternaerer Code bestehend aus den Elementen m, p und f. Um die Buchstaben a bis z abzubilden, werden Kombinationen aus je 3 Zeichen von “mmm” fuer “a” bis “ffp” fuer “z” verwendet. Grossbuchstaben entsprechen “Mmm” fuer “A” bis “Ffp” fuer “Z”. Alle anderen Zeichen bleiben gleich.

Der Filter ersetzt saemtliche Vorkommen von “mmm” bis “ffp” sowie der entsprechenden Grossschrift durch die Elemente aus der Ersetzungstabelle.

Ruhr.pm

Ein Kenny-Speak-Konverter

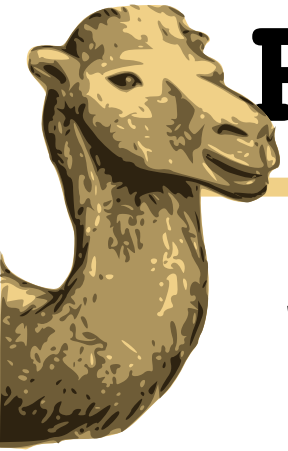
```
use strict;
use warnings;

my $i;
my $table = {
    map{
        my $i = $_;
        chr(97+$i), join('',
            map{qw(m p f)[$i/$_%3]}(9,3,1)
          );
      }(0..25)
  };
map{
    $i = $table->{$_};
    $i =~ s/(.)/uc($1)/e;
    $table->{uc($_)} = $i
  } keys(%{$table});

print("#!/usr/bin/perl\n"
  ."use Filter::McCormickized;\n");

while(<>){
  s/([a-z])/$table->{$1}/igse;
  print;
}
```

Dieses Programm funktioniert prinzipiell genau entgegengesetzt dem Modul **`Filter::McCormickized`**:

Es uebersetzt zeilenweise ein beliebiges Perl-Programm in ein Programm in Kenny-Speak.

Ein Header aus Perl-Shebang und Laden des Modul wird vorangestellt.

Aufruf:
`./kenny.pl` *`<in.pl >out.pl`*



Ruhr.pm

Wo ist das Programm?


```
#!/usr/bin/perl
use Filter::White;
```

Hier steht **`print "Hallo Welt";`**
-- aber wo?

Der Trick:
Der gesamte Quelltext wurde in Whitespace uebersetzt, d.h. in Kombinationen aus " ", **`\t`**, **`\n`** und **`\f`**.

Unser Modul **`Filter::White`** wandelt den "gebleichten" Code zurueck in Perl-Code, bevor er ausgefuehrt wird.

Ruhr.pm

Filter::White

```
package Filter::White;
use Filter::Simple;

my $table = {
    map{
        $i = $_;
        join('',
            map{
                substr(" \t\n\f",
                  $i/4**(3-$_)%4, 1);
              }(0..3)
          ), chr($i);
      }(0..255)
  };

FILTER{
    s/([ \t\f\n]{4})/$table->{$1}/gse;
  };
```

Unser Modul **`Filter::White`** uebersetzt anhand einer Uebersetzungstabelle alle 4-Tupel aus Whitespace-Zeichen **`[ \t\f\n]`** zurueck in je ein Byte.

Dabei ist fuer das Byte mit dem Wert n, 0 <= n <= 255 das Zeichen an Position m, 1 <= m <= 4 des 4-Tupels q bestimmt durch $(n / 4^{4-m}) \bmod 4$ mit (0, 1, 2, 3) := (“ “, \r, \n, \f), also
n = 0 -> q = (“ “, “ “, “ “, “ “),
n = 1 -> q = (“ “, “ “, “ “, \t), ...,
n = 255 -> q = (\f, \f, \f, \f).

Ruhr.pm

Der Konverter fuer Filter::White

```
use strict;
use warnings;

my $i;
my $table = [
    map{
        $i = $_;
        join('',
            map{
                substr(" \t\n\f",
                  $i/4**(3-$_)%4, 1);
              }(0..3)
          );
      }(0..255)
  ];

print("#!/usr/bin/perl\n"
  ."use Filter::White;\n");

while(<>){
  s/(.)/$table->[ord($1)]/gse;
  print;
}
```

Dieser Konverter “bleicht” regulaere Perl-Programme.

Er arbeitet dabei umgekehrt zu dem Filter-Modul **`Filter::White`** und stellt dem ausgegebenen Code eine Perl-Shebang sowie das Laden des Filter-Moduls voran.

Aufruf:
`./white.pl` `<`*`in.pl`* `>`*`out.pl`*

Ruhr.pm

Filter::Whiter

```
package Filter::Whiter;
use Filter::Simple;

my $table = {
    map{
        $i = $_;
        join('',
            map{
                substr(" \t",
                  $i/2**(7-$_)%2, 1);
              }(0..7)
          ), chr($i)
      }(0..255)
  };

FILTER{
    s/([ \t]{8})/$table->{$1}/gse;
  };
```

Leider wird der Form-Feed **\f** von einigen Editoren als Sonderzeichen dargestellt, so dass der umgewandelte Quellcode dort nicht unsichtbar erscheint. Zudem wuerde das Speichern mit einigen OS den **\n** ein **\r** voranstellen und damit das Rueckwandeln (ohne weitere Massnahmen) behindern.

Unser Modul **Filter::Whiter** verhindert diesese Probleme, indem es wie **Acme::Bleach** 8-Tupel aus “ “ und **\t** fuer alle Zeichen ausser **\r** und **\n** verwendet.

Ruhr.pm

Der Konverter fuer Filter::Whiter

```
use strict;
use warnings;

my $i;
my $table = [
    map{
        $i = $_;
        join('',
            map{
                substr(" \t",
                  $i/2**(7-$_)%2, 1);
              }(0..7)
          );
      }(0..255)
  ];

print("#!/usr/bin/perl\n"
  ."use Filter::Whiter;\n");

while(<>){
  s/([^\n\r])/$table->[ord($1)]/gse;
  print;
}
```

Das Programm **whiter.pl** “bleicht” wie **white.pl** regulaere Perl- Programme, jedoch fuer das Modul **Filter::Whiter**.

Ausgaben fuer **Filter::Whiter** sind jedoch auch doppelt so gross wie die fuer **Filter::White**, da 8-Tupel statt 4-Tupeln benoetigt werden.

Aufruf:
./whiter.pl <*in.pl* **>***out.pl*

Ruhr.pm

Filter::WhiterShorter

```
package Filter::WhiterShorter;
use Filter::Simple;

my $table = {
    map{
        $i = $_;
        join('',
            map{
               substr(" \t\n",
                 $i/3**(5-$_)%3, 1);
              }(0..5)
          ), chr($i);
      }(0..255)
  };

FILTER{
    s/([ \t\n]{6})/$table->{$1}/gse;
  };
```

Unser Modul **Filter::WhiterShorter** bildet einen Kompromiss zwischen **Filter::White** und **Filter::Whiter**.

Es verwendet 6-Tupel aus “ “, **\t** und **\n**. So sind Ausgaben nur noch 1,5mal so gross wie mit **Filter::White**, sind aber in quasi allen Editoren unsichtbar. Die Ausgaben sind zwar anfaellig fuer **\r\n**-Probleme, dies liesse sich aber mit einem weiteren Regex zum Entfernen aller **\r** beheben.

Ruhr.pm

Konverter f. Filter::WhiterShorter

```
use strict;
use warnings;

my $i;
my $table = [
    map{
        $i = $_;
        join('',
            map{
                substr(" \t\n",
                  $i/3**(5-$_)%3, 1);
              }(0..5)
          );
      }(0..255)
  ];

print("#!/usr/bin/perl\n"
  ."use Filter::WhiterShorter;\n");

while(<>){
  s/(.)/$table->[ord($1)]/gse;
  print;
}
```

whitershorter.pl “bleicht” Perl- Programme fuer das Modul **Filter::WhiterShorter**.

Aufruf:

```
./whitershorter.pl \
  <in.pl >out.pl
```

Ruhr.pm

Vielen Dank
für Eure Aufmerksamkeit

Ruhr.pm

Links

- Filter::Simple
  - http://search.cpan.org/dist/Filter-Simple/lib/Filter/Simple.pm
- Switch
  - http://search.cpan.org/dist/Switch/Switch.pm
- Lingua::Romana::Perligata
  - http://search.cpan.org/dist/Lingua-Romana-Perligata/lib/Lingua/Romana/Perligata.pm

Ruhr.pm

Links

- Kenny-Speak
  - http://www.namesuppressed.com/kenny/
- Acme::Filter::Kenny
  - http://search.cpan.org/dist/Acme-Filter-Kenny/lib/Acme/Filter/Kenny.pm
- Acme::Bleach
  - http://search.cpan.org/dist/Acme-Bleach/lib/Acme/Bleach.pm