

# Skript zum Thema Funktionsparser

Xenia Rendtel

16. März 2010

## Inhaltsverzeichnis

<b>1 Motivation</b>	<b>1</b>
<b>2 Der Algorithmus</b>	<b>1</b>
<b>3 Beispiele</b>	<b>3</b>
<b>4 Programmcode</b>	<b>3</b>

## 1 Motivation

Bei der Arbeit an meinem Plotterskript für Pstricks stieß ich auf die Problematik, dass ich eine Funktion mit mehreren Variablen zeichnen wollte. Dazu schaute ich als erstes in die Dokumentation für `PST-3dplot`, um zu sehen, wie es intern in Pstricks realisiert ist. Leider gab es hier nicht die Möglichkeit einfach als Parameter `algebraic` anzugeben, um in der gewohnten algebraischen Notation den Funktionsterm aufzuschreiben.

Da ich mich nicht so recht mit der umgekehrt-polnische-Notation (UPN) anfreunden konnte, schlug mir Stephan vor, doch einen Funktionsparser zu schreiben, der die algebraische Notation in die umgekehrt-polnische-Notation umwandelt. Dieses Skript eröffnet dem dem Plotterskript viele neue Möglichkeiten eröffnet, z.B. auch die Umwandlungen beliebiger Funktionsausdrücke in die UPN.

Tja, da stand ich nun vor einem großen Problem, was gelöst werden sollte. Wie geht man denn da heran? Was für einen Algorithmus nutzt man dafür? Ist es nicht doch einfacher die UPN zu lernen und sich nicht einen Algorithmus auszudenken zu müssen? Aber man wächst ja an seinen Aufgaben...

Zu zweit spielten wir mehrere Ansätze durch, bis wir dann auf die folgende Lösung kamen:

## 2 Der Algorithmus

Ein algebraischer Ausdruck wird zunächst in seine Einzelteile aufgeteilt. Handelt es sich um eine Zahl mit einem Vorzeichen, eine Rechenoperation, eine Funktion, einen Klammerterm öffnende oder schließende Klammer?

Dies ist die Funktion des `tokenizers`. Die vom `tokenizer` gefundenen Formelteile werden anschließend vom `regelwerk` schrittweise zu größeren Termen zusammengesetzt. Dies geschieht nach den Regeln der Mathematik, z.B. Punkt vor Strichrechnung, und wird solange wiederholt, bis nur noch ein Term übrig bleibt oder die Funktion `regelwerk` einen Fehler findet.

Die Regeln der `regelwerk`-Funktion sind die folgenden:

1. Ausdrücke der Form  $( \text{Liste} )$  werden in den Ausdruck `Liste` verwandelt und das Regelwerk wird rekursiv auf die `Liste` angewendet.
2. Ausdrücke der Form  $\text{Term} \wedge \text{Term}$  werden in den Term `(Term ^ Term)` verwandelt.
3. Ausdrücke der Form  $\text{Term} * \text{Term}$  oder  $\text{Term} / \text{Term}$  werden in den Term `(Term * Term)` bzw. `(Term / Term)` verwandelt.
4. Ausdrücke der Form  $\text{Funktion} \text{Term}$  werden in den Term `(Funktion (Term))` verwandelt.
5. Ausdrücke der Form  $\text{Term} + \text{Term}$  oder  $\text{Term} - \text{Term}$  werden in den Term `(Term + Term)` bzw. `(Term - Term)` verwandelt.
6. Ausdrücke der Form  $- \text{Term}$  werden in den Term `(0 - Term)` verwandelt.

Die Funktion `regelwerk` sucht in jedem Durchlauf eine anwendbare Regel und führt diese aus. Dabei wird die Regel mit der kleinsten Nummer zuerst probiert. Trifft eine Regel zu, so wird diese angewendet und die Funktion startet neu mit der ersten Regel. Wenn eine Regel an mehreren Stellen anwendbar ist, so wird sie an der am weitesten links stehenden Stelle angewendet. Der Prozess wird wiederholt, bis nur noch ein Term übrig ist oder keine Regel mehr anwendbar ist. Im letzteren Fall war die Ausgangsformel nicht gültig und es wird ein Fehler ausgegeben.

Mit dieser `regelwerk`-Funktion wird gleichzeitig parallel ein Rechenbaum aufgebaut, bei dem im Knoten immer die Operation steht. Die Blätter bestehen aus Zahlen oder Variablen. Bei den Funktionen wird der Funktionsname der Knoten und der rechte Teilbaum bleibt undefiniert.

Die Funktion `polnisch` liefert dann die UPN, indem der aufgebaute Rechenbaum ausgelesen wird. Das Abfallprodukt des `regelwerk` ist ein vollständig geklammerter Rechenausdruck.

### 3 Beispiele

Hier nun ein paar Beispiele:

Rechenausdruck	Rechenausdruck, vollständig geklammert	UPN
$2 * (3+4)$	$(2 * (3+4))$	2 3 4 + *
$2 + (x+2)^2$	$(2 + ((x+2)^2))$	2 x 2 + 2 ^ +
$(x+2)^2 + 2$	$((x+2)^2 + 2)$	x 2 + 2 ^ 2 +
$2 + \sin(2 * (x+3)) * 3$	$(2 + (\sin((2 * (x+3)) * 3)))$	2 2 x 3 + * 3 * sin +

Tab. 1: Beispiele für den Funktionsparser

### 4 Programmcode

Und hier nun der vollständige Programmcode in Perl:

```

funktionsparser.pl
1  #!/usr/bin/perl -w
2  #####
3  #
4  # Funktionsparser
5  #
6  # Autoren: Stephan Hoehrmann, Xenia Rendtel
7  #
8  # VERSION 2
9  # Letzte Aenderung: 07.11.2009
10 #
11 # Dieser Parser ersetzt eine Funktion in umgekehrter polnischer Notation.
12 # dazu wird ein Baum nach dem folgenden Prinzip erstellt:
13 # Zunaechst wird die Zeichenkette mit der Funktion tokenizer in
14 # ihre Einzelteile zerlegt.
15 # Dabei wird unterschieden, ob es sich um einen Term, einen Operator,
16 # eine Funktion, eine oeffnende oder schliessende Klammer oder eine Variable handelt.
17 # Dann wird diese neu gewonnene Liste mit dem
18 # folgenden Algorithmus in einen Baum verwandelt:
19 #
20 # 0. Solange Liste nicht leer, suche von Position 0 aufsteigend
21 # durch die Liste mit folgenden Kriterien:
22 # 1. ( Liste ) wird zu einem Ausdruck der Form Liste ohne Klammern,
23 # auf die der Algorithmus erneut angewendet wird. goto 0.
24 # 2. Term | Variable ^ Term | Variable wird zu einem Term mit dem Inhalt
25 # (Term | Variable ^ Term | Variable)
26 # 3. Term | Variable mal | geteilt Term | Variable wird zu einem Term
27 # mit dem Inhalt (Term | Variable mal | geteilt Term | Variable). goto 0
28 # 4. Funktion Term | Variable wird zu einem Term
29 # mit dem Inhalt (Funktion Term | Variable). goto 0
30 # 5. Term | Variable plus | minus Term | Variable wird zu einem Term
31 # mit dem Inhalt (Term | Variable plus | minus Term | Variable). goto 0
32 # 6. plus | minus Variable wird zu einem Term
33 # mit dem Inhalt (0 plus | minus Variable). goto 0
34 # 7. Fertig!
35 # Es wird dabei Schritt fuer Schritt ein Baum mit aufgebaut.
36 #####
37 use strict ;
38 use warnings;
39 use Data::Dumper;

```

```

40 use Math::Trig;
41 use Math::Complex;
42 use POSIX qw /floor ceil/;
43
44 my @funktionpolnisch;
45
46 ### Wichtige Funktionen:
47
48 sub abrunden {
49     my ( $wert, $schritt ) = @_ ;
50     return $schritt * floor( $wert / $schritt );
51 }
52
53 sub aufrunden {
54     my ( $wert, $schritt ) = @_ ;
55     return $schritt * ceil( $wert / $schritt );
56 }
57
58 sub fakultaet {
59     my ($n) = shift;
60
61     if ( ( $n == 1 ) || ( $n == 0 ) ) { return 1; }
62     elsif ( $n < 0 ) { return "Fehler"; }
63     else { return $n * fakultaet( $n - 1 ); }
64 }
65
66 sub round {
67     my ($zahl) = shift;
68     return int( $zahl + 0.5 );
69 }
70
71 # Die Funktion tokenizer erstellt aus einem String eine
72 # Liste mit den einzelnen Rechenoperationen des Strings
73 sub tokenizer {
74     my ($text) = @_ ;
75     my ( $token, $vorzeichen );
76     my @listevontokens;
77     $vorzeichen = 1;
78     $text =~ s/ //g;
79     $text = lc($text);
80     while ( $text ne "" ) {
81         if ( ( $text =~ /^( [-+]? [0-9]+ ( \. [0-9]+ )? ([eE] [ -+ ]? [0-9]+ )? )? / )
82             && ( $vorzeichen == 1 ) )
83         {
84
85             # Zahlen mit Vorzeichen
86             $token = $1;
87             $vorzeichen = 0;
88             push @listevontokens,
89                 {
90                     'Inhalt' => $token,
91                     'Typ' => "Term",
92                     'Baum' => {
93                         'knoten' => $token,
94                         'links' => undef,
95                         'rechts' => undef
96                     }
97                 };
98         }
99     }
100     elsif ( ( $text =~ /^( [0-9]+ ( \. [0-9]+ )? ([eE] [ -+ ]? [0-9]+ )? )? / )
        && ( $vorzeichen == 0 ) )

```

```

101 | {
102 |
103 |   # Zahlen ohne Vorzeichen
104 |   $token    = $1;
105 |   $vorzeichen = 0;
106 |   push @listevontokens,
107 |     {
108 |       'Inhalt' => $token,
109 |       'Typ'    => "Term",
110 |       'Baum'  => {
111 |         'knoten' => $token,
112 |         'links'  => undef,
113 |         'rechts' => undef
114 |       }
115 |     };
116 | }
117 |
118 | elsif ( $text =~ /^(\^)/ ) {
119 |
120 |   # Hochzeichen
121 |   $token    = $1;
122 |   $vorzeichen = 1;
123 |   push @listevontokens,
124 |     {
125 |       'Inhalt' => $token,
126 |       'Typ'    => "hoch",
127 |       'Baum'  => {
128 |         'knoten' => $token,
129 |         'links'  => undef,
130 |         'rechts' => undef
131 |       }
132 |     };
133 | }
134 |
135 | elsif ( $text =~ /^([-])/ ) {
136 |
137 |   # Operatoren
138 |   $token    = $1;
139 |   $vorzeichen = 1;
140 |   push @listevontokens,
141 |     {
142 |       'Inhalt' => $token,
143 |       'Typ'    => "minus",
144 |       'Baum'  => {
145 |         'knoten' => $token,
146 |         'links'  => undef,
147 |         'rechts' => undef
148 |       }
149 |     };
150 | }
151 | elsif ( $text =~ /^[+]/ ) {
152 |
153 |   # Operatoren
154 |   $token    = $1;
155 |   $vorzeichen = 1;
156 |   push @listevontokens,
157 |     {
158 |       'Inhalt' => $token,
159 |       'Typ'    => "plus",
160 |       'Baum'  => {
161 |         'knoten' => $token,

```

```

162     'links' => undef,
163     'rechts' => undef
164 }
165 ];
166 }
167 elif ( $text =~ /^(.*)/ ) {
168
169     # Operatoren
170     $token = $1;
171     $vorzeichen = 1;
172     push @listevontokens,
173     {
174         'Inhalt' => $token,
175         'Typ' => "mal",
176         'Baum' => {
177             'knoten' => $token,
178             'links' => undef,
179             'rechts' => undef
180         }
181     };
182 }
183 elif ( $text =~ /^[[:\]]/ ) {
184
185     # Operatoren
186     $token = $1;
187     $vorzeichen = 1;
188     push @listevontokens,
189     {
190         'Inhalt' => $token,
191         'Typ' => "geteilt",
192         'Baum' => {
193             'knoten' => $token,
194             'links' => undef,
195             'rechts' => undef
196         }
197     };
198 }
199 elif ( $text =~
200 /^(sin | cos | tan | acos | asin | log | ln | ceiling | floor | truncate | round | sqrt | abs | fact | exp)/
201 )
202 {
203
204     # Funktionen
205     $token = $1;
206     $vorzeichen = 1;
207     push @listevontokens,
208     {
209         'Inhalt' => $token,
210         'Typ' => "Funktion",
211         'Baum' => {
212             'knoten' => $token,
213             'links' => undef,
214             'rechts' => undef
215         }
216     };
217 }
218 elif ( $text =~ /^(\()/ ) {
219
220     # oeffnende Klammer
221     $token = $1;
222     $vorzeichen = 1;

```

```

223     push @listevontokens,
224     {
225         'Inhalt' => $token,
226         'Typ'   => "(",
227         'Baum'  => {
228             'knoten' => $token,
229             'links'  => undef,
230             'rechts' => undef
231         }
232     };
233 }
234 elseif ( $text =~ /^(\\)/ ) {
235
236     # schliessende Klammer
237     $token   = $1;
238     $vorzeichen = 0;
239     push @listevontokens,
240     {
241         'Inhalt' => $token,
242         'Typ'   => ")",
243         'Baum'  => {
244             'knoten' => $token,
245             'links'  => undef,
246             'rechts' => undef
247         }
248     };
249 }
250
251 elseif ( $text =~ /^[a-z][0-9a-z]*/ ) {
252
253     # Variable
254     $token   = $1;
255     $vorzeichen = 0;
256     push @listevontokens,
257     {
258         'Inhalt' => $token,
259         'Typ'   => "Term",
260         'Baum'  => {
261             'knoten' => $token,
262             'links'  => undef,
263             'rechts' => undef
264         }
265     };
266 }
267 else {
268
269     # Rest
270     $token   = $text;
271     $vorzeichen = 1;
272     push @listevontokens,
273     {
274         'Inhalt' => $token,
275         'Typ'   => "Rest",
276         'Baum'  => {
277             'knoten' => $token,
278             'links'  => undef,
279             'rechts' => undef
280         }
281     };
282 }
283

```

```

284     #print "Token " . $token . "\n";
285     $text = substr $text, length $token;
286 }
287 return @listevontokens;
288 }
289
290 # Ein vollstaendig geklammerter Ausdruck wird ausgegeben.
291 sub ausgabeliste {
292     my (@liste) = @_;
293     my $listenlaenge = scalar @liste;
294     my $i;
295     printf("");
296     for ( $i = 0 ; $i < $listenlaenge ; $i++ ) {
297         printf( "%s", $liste[$i]{ 'Inhalt' } );
298         printf(" ") if ( $i < $listenlaenge - 1 );
299     }
300     print "\n ";
301 }
302 }
303
304 # Der eigentliche Algorithmus
305 sub regelwerk {
306     my (@liste) = @_;
307     my ( @klammerterm, $i, $j, $gefunden, $klammerauf, $klammerzu,
308         $klammertiefe );
309
310     do {
311         $gefunden = 0;
312         if ( !$gefunden ) {
313
314             $klammerzu = $klammerauf = $klammertiefe = 0;
315
316             # Klammern:
317             # Wenn eine oeffnende Klammer gefunden wurde, dann wird der Zaehler klammertiefe erhoecht,
318             # bei einer schliessenden Klammer erniedrigt.
319             # Bei korrekter Klammerung ist der Zaehler am Ende 0
320             # und es wird die regelwerk-Funktion auf den Inhalt der Klammer angewendet.
321
322             for ( $i = 0 ; $i < scalar @liste ; $i++ ) {
323
324                 if ( $liste[$i]{ 'Typ' } eq "(" ) {
325                     if ( ( $klammertiefe == 0 ) ) {
326                         $klammerauf = $i;
327                     }
328                     $klammertiefe++;
329                 }
330                 if ( $liste[$i]{ 'Typ' } eq ")" ) {
331                     $klammertiefe--;
332                     if ( $klammertiefe == 0 ) {
333                         $klammerzu = $i;
334                     }
335                 }
336             }
337
338             ## Fehler wenn klammertiefe != 0 muss noch bearbeitet werden
339
340             if ( $klammerauf != $klammerzu ) {
341
342                 @klammerterm = ();
343                 for ( $j = $klammerauf + 1 ; $j <= $klammerzu - 1 ; $j++ ) {
344                     push @klammerterm,

```

```

345     {
346     'Inhalt' => $liste[$j]{ 'Inhalt' },
347     'Typ'   => $liste[$j]{ 'Typ' },
348     'Baum' => {
349         'knoten' => $liste[$j]{ 'Baum' }{ 'knoten' },
350         'links'  => undef,
351         'rechts' => undef
352     }
353     };
354 }
355
356 splice( @liste, $klammerauf, $klammerzu - $klammerauf + 1,
357         regelwerk(@klammerterm) );
358 $gefunden = 1;
359 }
360 }
361 }
362 }
363 if ( !$gefunden ) {
364
365     # Potenzieren
366     for ( $i = 0 ; $i < scalar @liste - 2 ; $i++ ) {
367
368         if ( ( $liste[$i]{ 'Typ' } eq "Term" )
369             && ( $liste[ $i + 1 ]{ 'Typ' } eq "hoch" )
370             && ( $liste[ $i + 2 ]{ 'Typ' } eq "Term" )
371             && ( !$gefunden ) )
372         {
373
374             splice(
375                 @liste, $i, 3,
376                 {
377                     'Inhalt' => "("
378                     . $liste[$i]{ 'Inhalt' }
379                     . $liste[ $i + 1 ]{ 'Inhalt' }
380                     . $liste[ $i + 2 ]{ 'Inhalt' } . ")",
381                     'Typ' => "Term",
382                     'Baum' => {
383                         'knoten' => $liste[ $i + 1 ]{ 'Baum' }{ 'knoten' },
384                         'links'  => $liste[$i]{ 'Baum' },
385                         'rechts' => $liste[ $i + 2 ]{ 'Baum' }
386                     }
387                 }
388             );
389             $gefunden = 1;
390         }
391     }
392 }
393 if ( !$gefunden ) {
394
395     # Multiplikation und Division
396     for ( $i = 0 ; $i < scalar @liste - 2 ; $i++ ) {
397         if (
398             ( $liste[$i]{ 'Typ' } eq "Term" )
399             && ( ( $liste[ $i + 1 ]{ 'Typ' } eq "mal" )
400                 || ( $liste[ $i + 1 ]{ 'Typ' } eq "geteilt" ) )
401             && ( $liste[ $i + 2 ]{ 'Typ' } eq "Term" )
402             && ( !$gefunden )
403         )
404         {
405             splice(

```

```

406     @liste, $i, 3,
407     {
408         'Inhalt' => "("
409         . $liste[$i]{ 'Inhalt' }
410         . $liste[ $i + 1 ]{ 'Inhalt' }
411         . $liste[ $i + 2 ]{ 'Inhalt' } . ")",
412         'Typ' => "Term",
413         'Baum' => {
414             'knoten' => $liste[ $i + 1 ]{ 'Baum' }{ 'knoten' },
415             'links' => $liste[$i]{ 'Baum' },
416             'rechts' => $liste[ $i + 2 ]{ 'Baum' }
417         }
418     }
419 );
420 $gefunden = 1;
421 }
422 }
423 }
424 if ( !$gefunden ) {
425
426 # Funktion
427 for ( $i = 0 ; $i < scalar @liste - 1 ; $i++ ) {
428     if ( ( $liste[$i]{ 'Typ' } eq "Funktion" )
429         && ( $liste[ $i + 1 ]{ 'Typ' } eq "Term" )
430         && ( !$gefunden ) )
431     {
432         if ( $liste[$i]{ 'Inhalt' } eq "exp" ) {
433             splice(
434                 @liste, $i, 2,
435                 {
436                     'Inhalt' => "("
437                     . exp(1) . "\^"
438                     . $liste[ $i + 1 ]{ 'Inhalt' } . ")",
439                     'Typ' => "Term",
440                     'Baum' => {
441                         'knoten' => "\^",
442                         'links' => {
443                             'knoten' => exp(1),
444                             'links' => undef,
445                             'rechts' => undef
446                         },
447                         'rechts' => $liste[ $i + 1 ]{ 'Baum' }
448                     }
449                 }
450             );
451         }
452     else {
453         splice(
454             @liste, $i, 2,
455             {
456                 'Inhalt' => "("
457                 . $liste[$i]{ 'Inhalt' }
458                 . $liste[ $i + 1 ]{ 'Inhalt' } . ")",
459                 'Typ' => "Term",
460                 'Baum' => {
461                     'knoten' => $liste[$i]{ 'Baum' }{ 'knoten' },
462                     'links' => $liste[ $i + 1 ]{ 'Baum' },
463                     'rechts' => undef
464                 }
465             }
466         );

```

```

467     }
468     $gefunden = 1;
469   }
470 }
471 }
472 if ( !$gefunden ) {
473
474   # Addition und Subtraktion
475   for ( $i = 0 ; $i < scalar @liste - 2 ; $i++ ) {
476     if (
477       ( $liste[$i]{ 'Typ' } eq "Term" )
478       && ( ( $liste[ $i + 1 ]{ 'Typ' } eq "plus" )
479         || ( $liste[ $i + 1 ]{ 'Typ' } eq "minus" ) )
480       && ( $liste[ $i + 2 ]{ 'Typ' } eq "Term" )
481       && ( !$gefunden )
482     )
483     {
484       splice(
485         @liste, $i, 3,
486         {
487           'Inhalt' => "(
488             . $liste[$i]{ 'Inhalt' }
489             . $liste[ $i + 1 ]{ 'Inhalt' }
490             . $liste[ $i + 2 ]{ 'Inhalt' } . )",
491           'Typ' => "Term",
492           'Baum' => {
493             'knoten' => $liste[ $i + 1 ]{ 'Baum' }{ 'knoten' },
494             'links' => $liste[$i]{ 'Baum' },
495             'rechts' => $liste[ $i + 2 ]{ 'Baum' }
496           }
497         }
498       );
499       $gefunden = 1;
500     }
501   }
502 }
503 if ( !$gefunden ) {
504
505   # Term mit Vorzeichen
506   for ( $i = 0 ; $i < scalar @liste - 1 ; $i++ ) {
507     if (
508       (
509         ( $liste[$i]{ 'Typ' } eq "plus" ) || ( $liste[$i]{ 'Typ' } eq "minus" )
510       )
511       && ( $liste[ $i + 1 ]{ 'Typ' } eq "Term" )
512     )
513     {
514       splice(
515         @liste, $i, 2,
516         {
517           'Inhalt' => "(0"
518             . $liste[$i]{ 'Inhalt' }
519             . $liste[ $i + 1 ]{ 'Inhalt' } . )",
520           'Typ' => "Term",
521           'Baum' => {
522             'knoten' => $liste[$i]{ 'Baum' }{ 'knoten' },
523             'links' => {
524               'knoten' => 0,
525               'links' => undef,
526               'rechts' => undef
527             },

```

```

528         'rechts' => $liste[ $i + 1 ]{'Baum'}
529     }
530 }
531 );
532 $gefunden = 1;
533 }
534 }
535 }
536
537 } while ($gefunden);
538
539 ausgabeliste(@liste);
540 return @liste;
541 }
542
543 # polnische Notation wird ausgegeben.
544 sub polnisch {
545     my $ref = shift;
546
547     # suche links (rekursiv)
548     if ( defined( $$ref{'links'} ) ) {
549         polnisch( $$ref{'links'} );
550     }
551
552     # suche rechts (rekursiv)
553     if ( defined( $$ref{'rechts'} ) ) {
554         polnisch( $$ref{'rechts'} );
555     }
556
557     # Ausgabe des Knotens
558     print " $$ref{'knoten'}";
559 }
560
561 sub rechenbaum_ausrechnen {
562     my $baum = shift;
563     my $variablen = shift;
564
565     print $$baum{'knoten'} . "\n";
566     if ( $$baum{'knoten'} eq "*" ) {
567         return rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) *
568             rechenbaum_ausrechnen( $$baum{'rechts'}, $variablen );
569     }
570     elsif ( $$baum{'knoten'} eq ":" ) {
571         return rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) /
572             rechenbaum_ausrechnen( $$baum{'rechts'}, $variablen );
573     }
574
575     elsif ( $$baum{'knoten'} eq "/" ) {
576         return rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) /
577             rechenbaum_ausrechnen( $$baum{'rechts'}, $variablen );
578     }
579     elsif ( $$baum{'knoten'} eq "+" ) {
580         return rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) +
581             rechenbaum_ausrechnen( $$baum{'rechts'}, $variablen );
582     }
583     elsif ( $$baum{'knoten'} eq "^" ) {
584         return rechenbaum_ausrechnen( $$baum{'links'}, $variablen )
585             **rechenbaum_ausrechnen( $$baum{'rechts'}, $variablen );
586     }
587
588     elsif ( $$baum{'knoten'} eq "-" ) {

```

```

589     return rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) -
590     rechenbaum_ausrechnen( $$baum{'rechts'}, $variablen );
591 }
592 elseif ( defined $$variablen{ $$baum{'knoten'} } ) {
593     return $$variablen{ $$baum{'knoten'} };
594 }
595 elseif ( $$baum{'knoten'} eq "sin" ) {
596     return sin( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
597 }
598 elseif ( $$baum{'knoten'} eq "cos" ) {
599     return cos( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
600 }
601 elseif ( $$baum{'knoten'} eq "tan" ) {
602     return tan( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
603 }
604 elseif ( $$baum{'knoten'} eq "asin" ) {
605     return arcsin( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
606 }
607 elseif ( $$baum{'knoten'} eq "acos" ) {
608     return acos( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
609 }
610 elseif ( $$baum{'knoten'} eq "log" ) {
611     return log( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
612 }
613 elseif ( $$baum{'knoten'} eq "ceiling" ) {
614     return aufrunden( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ), 1 );
615 }
616 elseif ( $$baum{'knoten'} eq "floor" ) {
617     return abrunden( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ), 1 );
618 }
619 elseif ( $$baum{'knoten'} eq "truncate" ) {
620     if ( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) < 0 ) {
621         return aufrunden( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ),
622             1 );
623     }
624     else {
625         return abrunden( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ),
626             1 );
627     }
628 }
629 }
630 elseif ( $$baum{'knoten'} eq "round" ) {
631     return round( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
632 }
633 }
634 elseif ( $$baum{'knoten'} eq "sqrt" ) {
635     return sqrt( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
636 }
637 }
638 elseif ( $$baum{'knoten'} eq "abs" ) {
639     return abs( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
640 }
641 }
642 elseif ( $$baum{'knoten'} eq "fact" ) {
643     return fakultaet( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
644 }
645 }
646 elseif ( $$baum{'knoten'} eq "exp" ) {
647     return exp( rechenbaum_ausrechnen( $$baum{'links'}, $variablen ) );
648 }
649 }

```

```
650 else { # ACHTUNG: Keine Fehlerbehandlung!!!
651     return $$baum{'knoten'};
652 }
653 }
654
655 sub funktionsausgabe {
656     my $funktionsausdruck = shift;
657     my @ergebnis          = regelwerk( tokenizer($funktionsausdruck) );
658
659     if ( scalar @ergebnis != 1 ) {
660         printf("Fehler \n");
661         ausgabeliste(@ergebnis);
662     }
663     else {
664         polnisch( $ergebnis[0]{'Baum'} );
665         print "\n";
666     }
667 }
668
669 my @ergebnis = regelwerk( tokenizer("(x-2)*(x+2)") );
670 print rechenbaum_ausrechnen(
671     $ergebnis[0]{'Baum'},
672     {
673         'x' => 2,
674         'y' => 2,
675         'pi' => pi()
676     }
677 )
678 . "\n";
679
680 funktionsausgabe("x*x*-3");
```

## Literatur

[KRIENKE, 2002] KRIENKE, RAINER (2002). Programmieren in Perl. Hanser Fachbuchverlag, 2., erw. Aufl.