

TU MÜNCHEN

PROSEMINAR PROGRAMMIERSPRACHEN

---

# PUGS

---

*Autor*  
Dominik MEYER

*Betreuer*  
Dr. Markus WENZEL

23. Januar 2007



# Inhaltsverzeichnis

<b>1</b>	<b>Einführung</b>	<b>5</b>
1.1	Entwicklung von Perl . . . . .	5
1.2	Parallele Ansätze zu PUGS . . . . .	6
1.3	Was ist PUGS . . . . .	6
<b>2</b>	<b>Grundlagen</b>	<b>7</b>
2.1	Variablen und Geltungsbereiche . . . . .	7
2.2	Typen . . . . .	8
<b>3</b>	<b>Interessante Operatoren</b>	<b>9</b>
3.1	Smart Match . . . . .	9
3.2	Range Objekte . . . . .	10
3.3	Binding . . . . .	11
3.4	Referencing . . . . .	11
<b>4</b>	<b>Meta Operatoren</b>	<b>13</b>
4.1	Hyper Operator . . . . .	13
4.2	Reduction Operator . . . . .	14
4.3	Cross Operator (Permutationen von Listen) . . . . .	14
<b>5</b>	<b>Subroutines</b>	<b>15</b>
5.1	Subroutine Stubs . . . . .	15
5.2	Parameter . . . . .	16
5.3	Temporäre Variablen . . . . .	19
5.4	Traits . . . . .	19
5.5	Currying und Wrapping . . . . .	20
5.6	Multisubs . . . . .	20
5.7	Makros . . . . .	21
<b>6</b>	<b>Rules</b>	<b>23</b>
6.1	Regexes, Rules und Tokens . . . . .	23
6.2	Grammars . . . . .	24
6.3	Metacharacters . . . . .	24
6.3.1	Escaping . . . . .	25

6.3.2	Quantifiers . . . . .	25
6.3.3	Assertions . . . . .	26
6.4	Capturing . . . . .	26
6.5	Beispiele zu Rules . . . . .	27
<b>7</b>	<b>Zusammenfassung</b>	<b>29</b>

# Kapitel 1

## Einführung

Anfang 2000 wurde auf der vierten Perl Conference beschlossen Perl 6 von Grund auf neu zu entwickeln. Daraufhin begann die erste Phase der Spezifikation der neuen Perlversion. Jeder konnte ein RFC (Request For Comment) abgeben, das mit in den Designprozess einbezogen wurde. Anschließend wurden diese gesichtet und zusammengefasst. So entstanden die sogenannten “Synopses”.

In diesem Dokument will ich einige interessante und neue Aspekte von Perl 6 präsentieren. Natürlich stellt dies nur eine kleine Auswahl dar, da eine gesamte Sprachbeschreibung den Rahmen des Proseminars sprengen würde.

### 1.1 Entwicklung von Perl

Die erste Version von Perl hat Larry Wall alleine konzipiert und implementiert. Mit anfangs noch sehr wenig Features ausgestattet, wuchs die Sprache immer mehr zu einem Universalwerkzeug. Mehr und mehr wurde Perl von einer Community weiterentwickelt, die auch Module im CPAN (Comprehensive Perl Archive Network) bereitstellt.

Durch die Vielzahl von Modulen, Erweiterungen und Dialekten, die mittels “source filters” eingebunden werden können, ist es sehr schwierig die Konsistenz unter den Modulen, beziehungsweise die Kompatibilität bei Verwendung von mehreren Modulen zu erhalten. Außerdem wurde in der Weiterentwicklung der Sprache bewusst auf neue Funktionen verzichtet, die in Konflikt mit alten geraten könnten.

Nun wurde mit Perl 6 erstmals die Rückwärtskompatibilität von Perl aufgegeben und Perl von Grund auf neu entworfen. Dabei ist es weiterhin möglich Perl 5 Programme auszuführen, es wird einfach ein Perl 5 Interpreter in den Perl 6 Interpreter eingebettet. Die Programme sollen, so der Plan, auf einer Virtuellen Maschine namens Parrot ausgeführt werden, die aber auch für andere Skript- oder Programmiersprachen geeignet ist.

## 1.2 Parallele Ansätze zu PUGS

Der erste Ansatz einen lauffähigen Compiler für Perl 6 zu bekommen war P6C, ein Perl 6 nach Parrot Compiler, geschrieben in Perl 5. Aber dieser Ansatz war nicht mächtig genug, die komplizierte Syntax von Perl 6 zu verarbeiten, also wurde P6C aufgegeben.

Die nächste Idee war einen Parser für Perl 6 Rules zu schreiben, und dann die Perl 6 Syntax in Rules zu formulieren. Dies geschah mit der Perl 5 Modul `Perl6::Rules`, doch leider blieb dies für immer ein nur teilweise funktionierender Prototyp.

Schließlich wollte man diesen Ansatz, die Perl 6 Syntax in Rules zu spezifizieren, weiterverfolgen und es wurde ein Compiler geschrieben, der Rules in Parrot Assembler Code kompiliert. Dieser Kompiler wird zwar weiterentwickelt, aber die Syntax von Perl 6 ist noch nicht in Rules spezifiziert.

## 1.3 Was ist PUGS

Schließlich beschloss Frau Audrey Tang einen weiteren Interpreter für Perl 6 zu schreiben und wählte dabei als Programmiersprache Haskell. Dieser Interpreter war innerhalb einer Woche lauffähig und konnte schon einfache Programme ausführen. Mittlerweile ist ein großer Teil von Perl 6 in Pugs implementiert und auch Perl 5 Programme können eingebettet werden. Pugs unterstützt sowohl die interpretierte Ausführung von Programmen, als auch Kompilation in andere Sprachen (Perl 5, JavaScript) und Parrot Bytecode.

In Zukunft soll Pugs immer mehr zum Compiler für Perl 6 werden und schließlich sollte der Perl 6 Compiler in Perl 6 selbst implementiert werden. Dann kann man Pugs benutzen, um diesen zu kompilieren und hat so eine Programmiersprache, die sich quasi selbst ausführen kann (self-hosting).

# Kapitel 2

## Grundlagen

Viele grundlegende Dinge in Perl 6 wurden übernommen, denn man wollte keine völlig von den Vorgängern verschiedene Sprache entwerfen, sondern eine neue Perlversion, die auch Perl bleiben sollte. Außerdem sollte die Sprache einfach zu verstehen sein und auch der natürlichen Programmierweise entsprechen. So gibt es weiterhin viele kontextabhängige Ausdrücke und es wird versucht, die Sprachelemente dahingehend zu entwickeln, dass eine für den Programmierer ganz intuitive Ausdrucksweise entsteht.

In dem Buch “Perl 6 and Parrot” ([2]) wird auch von der “Wasserbett-Theorie” gesprochen. Diese vergleicht die Komplexität einer Sprache mit einem Wasserbett. Drückt man es an einem Punkt nieder, das heißt übertragen, reduziert man die Komplexität bestimmte Dinge auszudrücken, so erhebt sich die Matratze, also auch die Komplexität an einem anderen Punkt. Es ist nicht möglich eine Sprache zu entwerfen, in der es für alle Probleme einfache Konstrukte gibt.

### 2.1 Variablen und Geltungsbereiche

Grundsätzlich sind Variablen Container für Daten. Diese Daten können in Strings, Zahlenwerten oder auch Objekten repräsentiert sein. Variablen können aber auch Referenzen auf Objekte oder spezielle Daten, wie Filehandles, enthalten.

In Perl 6 wird, wie auch schon in den Vorgängerversionen, zwischen drei grundsätzlichen Typen von Variablen unterschieden. Diese Unterscheidung wird in einem Präfixzeichen vor dem Variablennamen ausgedrückt. Skalare Werte erhalten ein \$, Arrays ein @ und Hashes ein %.

Dabei sind die skalaren Variablen die flexibelsten von allen. Sie können nicht nur Strings und Zahlen, sondern auch Objekte, Filehandles und Referenzen (siehe 3.4) enthalten. Arrays sind sicherlich schon aus anderen Programmiersprachen bekannt. Auf die einzelnen Elemente kann dabei mit einer Zahl, beginnend bei 0, in eckigen Klammern hinter dem Variablennamen zu-

gegriffen werden.

Eine Besonderheit von Perl ist, dass Hashes als grundsätzlicher Variablentyp unterstützt werden. Konstruiert werden Hashes durch Listen von Paaren, bestehend aus einem Key und einem Wert.

Unter einem Geltungsbereich versteht man in Perl immer einen Block, der mit { und } abgeschlossen ist. Wird in einem solchen Bereich eine Variable mit `my` deklariert, so ist diese nur für diesen Bereich gültig und zugreifbar. Das Gegenteil bewirkt `our`. Hiermit kann man globale Variablen erstellen, die im gesamten Programm gültig sind. Desweiteren gibt es noch die Sonderformen `has` (im Zusammenhang mit Objekten) und `tmp` (siehe hierzu Kapitel 5.3).

```
my $variable = "String";
my @array = (1, 2, 3);
my @array1 = ("a", "b", "c");
my @array2 = <a b c>;           # Dasselbe
our %hash = ('key' => 'value',
            'key1' => 'value1');

my $skalar;
$skalar = @array1;           # $skalar erhaelt eine
                             Referenz auf @array1
```

## 2.2 Typen

Komplett neu eingeführt wurde in Perl 6 ein Typsystem. Dieses muss aber nicht benutzt werden und es kann weiterhin, wie gewohnt, untypisiert programmiert werden. Der Vorteil ist aber, dass, wenn man Typen verwendet, der Compiler optimieren kann und Fehler vermieden werden.

Es stehen, wie von anderen Sprachen bekannt, die Grundtypen `Int`, `Num`, `String`, ... zur Verfügung. Darüberhinaus gibt es aber auch noch Perl spezifische Type, wie `Regex`, `Match`, `Hash`, `Scalar`, ....

Zusätzliche eigene Typen lassen sich als Klasse selbst definieren.

```
my $skalar is Num;           # $skalar ist vom Typ Num
my Num $skalar;             # ----- '-----
my $skalar of Num;          # $skalar enthaelt Num und ist vom
                             Typ Scalar
```

## Kapitel 3

# Interessante Operatoren

Perl 6 bietet eine Reihe von sehr interessanten Operatoren an, die so in anderen Programmiersprachen nur teilweise oder gar nicht vorhanden sind. Natürlich können auch eigene Operatoren definiert werden (siehe hierzu Kapitel 5 und 5.4).

### 3.1 Smart Match

Wollte man in Perl 5 einen regulären Ausdruck auf einen String anwenden, so benötigte man den `~` Operator. Diesen Operator gibt es in Perl 6 nicht mehr. Stattdessen gibt es den Smart Match Operator `~~`, der aber viel mehr ist, als die Anwendung eines regulären Ausdrucks (siehe hierzu Kapitel 6). Zum anderen kann man simple Vergleiche durchführen, indem man skalare Werte angibt, aber auch komplexere Strukturen miteinander in Beziehung führen. Kann der Typ der zu vergleichenden Variablen nicht zu Kompilierzeit festgestellt werden, so muss zur Ausführungszeit festgestellt, welche Art von Vergleich durchgeführt werden soll.

Ein Smart Match mit Skalaren Werten durchzuführen bedeutet, dass die beiden Operanden auf Gleichheit überprüft werden. Unterschiedlich hiervon verhält sich der Smart Match Operator, wenn als Operand ein Rule Objekt angegeben wurde. Dann wird ein Pattern Match durchgeführt. Ist das Rule Objekt eine Ersetzung, so wird diese durchgeführt und der Rückgabewert je nach Erfolg der Ersetzung bestimmt.

```
$zahl ~~ 42
((5 * 8) + 2) ~~ 42

$variable ~~ undef           # ist $variable definiert?

$string ~~ /blubb/          # ist blubb im String
                             enthalten?

$string ~~ s/hihi/hoho/     # ersetze hihi durch hoho
```

```
# Smart Match im Listenkontext
$sache ~~ (Spaten, Spachtel, Spritze)
$sache ~~ (2, /Spa/, Ziegel)

($zahl, $text) ~~ (3, /text/)
```

Mit einem Array überprüft ein Smart Match, ob das Array den Wert enthält, oder ob ein Pattern Match erfolgreich ist. Führt man den Smart Match mit einem Integer Wert aus, so wird überprüft, ob ein Wert an der Stelle des Zahlenwertes existiert und Wahr ist.

```
@array ~~ /Fahrrad/

2 ~~ @array    # @array[2] != undef and @array[2] == True
```

Mit Skalaren Werten in Verbindung mit Hashes verhält sich der Smart Match Operator so, dass jeweils die Keys des Hashes gematcht werden (also auch bei Patterns). Interessant wird es, wenn zwei Hashes oder ein Hash und ein Array gematcht werden. Hierbei wird überprüft, ob es eine Überschneidung der Keys der beiden Hashes gibt, beziehungsweise, ob die Elemente des Arrays als Keys existieren und ob der Hashwert bei diesen Keys einen wahren Wert besitzt.

```
$string ~~ %hash    # existiert %hash{$string}?
%hash ~~ /String/

%hash1 ~~ %hash2    # existiert eine Schnittmenge
                    # der Keys?

%hash ~~ @array
```

Eine weitere sehr interessante Verwendung des Smart Match Operators ist in Verbindung mit Subroutines. Standardmäßig wird hier einfach der Rückgabewert der Subroutine als bool'scher Wert interpretiert. Nimmt die Subroutine auch Argumente und passen die Argumente auf der einen Seite des Smart Match zur Signatur der Subroutine, so wird die Routine mit den zugehörigen Werten aufgerufen. Dabei wird ein Block als anonyme Subroutine interpretiert.

```
$wert ~~ true_or_false

$wert ~~ &sub_test    # sub_test($wert)
$wert ~~ {$_ ~~ 5}   # $_ ist $wert
```

## 3.2 Range Objekte

Der Operator “..” konstruiert Range Objekte. Dabei werden die Grenzen wahlweise mit oder ohne vorangestelltem `^` für Exklusion an diesem Rand des

Intervalls angegeben. Zusätzlich kann auch noch mittels `:by(x)` angegeben werden mit welcher Schrittweite `x` hochgezählt werden soll.

Im Gegensatz zu Perl 5 sind Ranges in Perl 6 nicht reversibel. Das heißt `3..1` ist immer null. Stattdessen kann man entweder `1..3:by(-1)` oder `reverse 1..3` (funktioniert auch für alphabetische Ranges) schreiben.

Range Objekte werden immer lazy ausgewertet, das heißt, es wird nicht sofort eine Liste mit allen Werten gespeichert, sondern nur die Randwerte. So ist es möglich unendliche Listen zu erstellen `0..*`. Wird ein Smart Match mit einem Range Objekt durchgeführt, so wird immer mit den Randwerten verglichen.

```
0..*           # 0..+Inf
^4            # 0..3

                # Smart Match
2.1 ~~ 1..3   # 1 <= 2.1 <= 3
2.1 ~~ 1^..^3 # 1 < 2.1 < 3
```

### 3.3 Binding

Mit Binding kann man einen Aliasnamen für Variablen und Objekte einführen. Hierbei wird nur eine Kopie der Variable im Speicher gehalten und wenn eine der beiden Variablen beschrieben wird, so ändert sich der Wert von beiden. Ein solches Alias kann mit dem `:=` Operator erstellt werden. Möchte man überprüfen, ob zwei Variablen ein Alias der jeweils anderen sind, so kann man dies mit `==` tun.

```
my $x = 2
my $y := $x
$y = 5           # $x = 5 und $y = 5

$x == $y        # True
```

### 3.4 Referencing

Wird ein Array oder Hash in skalarem Kontext verwendet, so wird automatisch eine Referenz auf das Array oder den Hash erstellt. Möchte man jedoch eine Referenz erzwingen, so kann man dies mit dem Operator erzwingen. Nun ist es möglich ein Array zu erstellen, das mehrere Referenzen auf Arrays enthält. Das Gegenteil des Operators ist der "Splat" Operator `*`. Hierbei wird erzwungen die einzelnen Elemente einer Liste zu betrachten, anstatt eine Referenz auf die Liste zu erzeugen oder die Liste zu konkatenieren.

```
my @refarray = (\@a, \@b, \@c) # @refarray enthaelt nur
```

```
my @array = (@a, @b)
my @arr = (\@a, \@b)
@c := @arr
(@d, %e) := *@c
```

Referenzen  
# @array enthaelt alle  
Elemente von @a und @b  
  
# @d ist @a, %e ist %b

## Kapitel 4

# Meta Operatoren

Wird in Perl ein Operator definiert, so muss dies für alle Kontexte gesondert geschehen. Möchte man beispielsweise einen neuen Negationsoperator definieren, muss dieser für Numerischen, String und Booleschen Kontext gesondert implementiert werden.

Im Gegensatz dazu gibt es in Perl 6 die sogenannten Meta Operatoren, die einen schon existierenden Operator in ein mächtigeres Konstrukt verwandeln.

### 4.1 Hyper Operator

Der Hyper Operator erweitert Operatoren auf Listen, Arrays und Hashes. Wendet man den Hyper Operator an, wird die Operation zwischen den Spitzen Klammern “>>” “<<” auf die gesamte Liste oder die Listen, die als Argumente gegeben wurden, angewendet.

```
(1, 2, 3, 4) >>+<< (1, 2, 3, 4)    # (2, 4, 6, 8)
```

Natürlich ergeben sich Probleme, wenn die zwei Listen, die verarbeitet werden sollen, nicht die gleiche Länge haben. Tritt dieser Fall ein, so muss eine der Listen erweitert werden. Dies geschieht dadurch, dass die spitzen Klammern auf der Seite umgedreht werden, auf der erweitert werden soll.

```
(1, 2, 3, 4) >>+>> (1, 2, 3)      # (2, 4, 6, 7)
(1, 2, 3, 4) >>+>> 1              # (2, 3, 4, 5)
(1, 2, 3, 4) >>+>> 1              # (2, 3, 4, 5)
-<< (1, 2, 3)                      # (-1, -2, -3)
```

Sind Listen ungleich lang und wird nicht erweitert, so wird eine Exception generiert.

Auf Arrays angewendet sind Hyper Operatoren sogar rekursiv definiert.

```
-<< [[1, 2,], 3]                  # [-<<[1, 2], -<<3]
                                   # = [[-1, -2], -3]
```

## 4.2 Reduction Operator

Mit dem Reduction Operator können Infix Operatoren in der Hinsicht erweitert werden, dass sie eine Liste reduzieren. Dabei wird so vorgegangen, als seien der Operator und die einzelnen Elemente der Liste “von Hand” im Programmtext getippt worden. Folglich ist der Reduction Operator kein wirklicher Listen Operator.

```
[+] 1, 2, 3                # 1 + 2 + 3 = 6

sub postfix:<!> ($x) {[*] 1..$x} # Fakultaet
3!                          # 3 * 2 * 1 = 6
```

Dadurch, dass der Reduction Operator auf alle Infix Operatoren angewendet werden kann, also auch auf beispielsweise Vergleichsoperatoren, werden unübliche Programmkonstrukte möglich.

```
if [<] $a, $b, $c {...}    # if $a < $b < $c {...}
[,] 1, 2, 3                # (1, 2, 3)
```

## 4.3 Cross Operator (Permutationen von Listen)

Der Cross Operator  $X$  erweitert jeden Infix Operator dahingehend, dass er alle Permutationen von Elementen der ersten Liste mit Elementen der zweiten Liste bildet und den Operator darauf anwendet. Mit dem Konkatenations Operator für Strings erhält man beispielsweise:

```
<a b> X~X <c d>            # 'ac', 'ad', 'bc', 'bd'

# das Gleiche:
('a', 'b') X~X ('c', 'd')

<a b> X,X 1,2 X,X <x y> # Alle Tupel, die a oder b, 1 oder 2,
                        x oder y enthalten
```

Für die spezielle Form zur Konkatenation ( $X X$ ) und zur Konstruktion von Listen ( $X,X$ ) gibt es auch abkürzende Schreibweisen:

```
<a b> XX <c d>            # <a b> X,X <c d>
<a b> X <c d>             # <a b> X~X <c d>
```

# Kapitel 5

## Subroutines

Eine Subroutinendeklaration beginnt immer mit dem Schlüsselwort `sub`. Darauf folgt ein Name, optionale Parameter und Traits (siehe Kapitel 5.4 Traits). Der Körper der Routine wird in einen Block, abgeschlossen von geschweiften Klammern, gefasst. Gegebenenfalls kann noch der Gültigkeitsbereich mit `my` oder `our` angegeben werden, wobei `our` Standardvorgabe ist. Möchte man den Rückgabebetyp der Routine spezifizieren, so kann man dies entweder vor dem Schlüsselwort `sub`, oder mittels der `-->` Syntax tun.

```
my RETTYP sub NAME (PARAMETER) TRAITS {...}

my Num sub add (Num $x) is inline {return $x + 1}
my sub add (Num $x --> Num) is inline {...}

sub add ($x) {return $x + 1}
sub add {$_ + 1}
```

Gibt man bei der Deklaration von Subroutinen keinen Namen an, so erhält man eine anonyme Subroutine. Damit diese benutzt werden kann, ist es natürlich sinnvoll eine Referenz zu speichern. Man kann einen normalen Codeblock (abgeschlossen von `{` und `}`) als anonyme Subroutine verwenden. Allerdings ist dann kein `return` aus diesem Block mehr möglich. Dasselbe gilt für anonyme Blöcke konstruiert mit `->`, wobei hier noch die Angabe von Parametern möglich ist.

```
$kochen = sub ($wasser, $energie) {...}
$kochen = -> $wasser, $energie {...}

$zeit = {say time()}
```

### 5.1 Subroutine Stubs

Möchte man nur die Signatur und den Rückgabebetyp einer Routine spezifizieren ohne den Körper angeben zu müssen, so kann man dies tun indem man

den `...` Operator verwendet. Es kann nun Code geschrieben werden, der wie Pseudocode aussieht, aber gültige Definitionen enthält. Wird eine Subroutine, die nur den `...` Operator enthält ausgeführt, so gibt diese immer `False` zurück. Alternativ kann man den `!!!` oder `???` Operator verwenden, der das Programm immer beendet beziehungsweise eine Warnung ausgibt.

```
sub sehr_kompliziert {...}
sub noch_zu_tun {!!!}
```

## 5.2 Parameter

Für Parameter gelten in Perl 6 eine Vielzahl von Regeln. Zum einen gibt es verschiedene Arten von Parametern (positionsabhängig, nötig, optional, benannt, ...), zum anderen gibt es mehrere Möglichkeiten Argumente an Routinen zu übergeben. Es gibt aber auch Subroutines ohne Parameter, bei denen die Argumente an Platzhalter Variablen übergeben werden, oder in Standard Variablen gespeichert werden.

Die einfachste und auch intuitivste Art von Parametern sind die Positionsabhängigen. Hierbei werden bei der Argumentübergabe des Routinenaufrufs je nach Aufeinanderfolge der Argumente die Werte zugeordnet. Positionsabhängige Parameter sind standardmäßig immer notwendig. Das heißt das Weglassen beim Aufruf erzeugt eine Exception. Ein Parameter kann innerhalb der Routine nur gelesen werden. Möchte man auch schreibend darauf zugreifen, so muss dies per Trait (`is rw`, `is copy`) erlaubt werden.

```
sub quad ($x) {return $x ** 2}
$y = quad(2)                # $y = 4

sub exp ($x is rw, $y) {$x **= $y}
exp($y, 2)                  # $y = 16
```

Wie bereits gesagt, sind positionsabhängige Parameter standardmäßig nicht optional. Möchte man aber einen solchen Parameter als optional definieren, so kann man dies durch ein nachgestelltes `? tun`. Das Gegenteil wäre ein nachgestelltes `!`.

```
sub kaffee ($bohnen, $wasser, $zucker?, $milch?) {...}
kaffe('gemahlen', '1 Liter', True)    # Ok
kaffe('1 Liter', True, True)          # Fehler
                                       $bohnen sind nun '1 Liter'
                                       $wasser ist True
```

Im obigen Beispiel kann man nicht angeben, dass man gerne Milch in seinem Kaffee hätte ohne auch Aussagen über den Zucker zu machen. Dieses Problem löst die benannte Argumentübergabe. Hierbei werden die Werte als Paare in der Form `name => $wert` angegeben. Man kann auch einfach kurz

`:name($wert)` schreiben. Möchte man hingegen kein benanntes Argument, sondern ein Paar als positionsabhängiges Argument übergeben, so muss das Paar entweder in Klammern geschlossen oder der Key-Teil mit Anführungszeichen versehen werden (`'name' => $wert`).

```
sub fun {...}

fun name => $wert, 1, 2, 3      # name ist benanntes Argument
fun :name($wert), 1, 2, 3     # dasselbe
fun (name => $wert), 1, 2     # name ist positionsabhaengiges
                             # Argument
fun 'name' => $wert, 1, 2     # dasselbe
fun (:name($wert)), 1, 2     # -- ' ' --

fun :$name, 1, 2              # Kurzform fuer :name($name)
```

Der andere Weg ein Paar, das eigentlich als positionsabhängiges Argument verwendet würde, als benanntes Argument zu verwenden ist natürlich auch möglich. So kann man beispielsweise die Keys und Values eines Hashes als Argumente verwenden. Um dies zu erreichen muss den Paaren oder dem Hash ein `|` Symbol vorangestellt werden.

```
%args = {'name' => $wert, 'name1' => $wert1}
%args1 = (:name($wert), :name1<string>)

fun %args                      # positionsabh"angig Hash
fun |%args                     # benannt
fun |%args1                    # - ' ' -

fun |('name' => 'wert')
```

Um bei der Routinendefinition eine benannte Argumentübergabe zu erzwingen, muss man in der Signatur vor den Argumenten, die benannt werden sollen, den Doppelpunkt sowie den Namen angeben. Auch hier ist wieder die Kurzform mit nur dem Doppelpunkt und dem entsprechenden Variablennamen erlaubt.

```
sub fun (:name($wert), :param($param)) {...}
sub fun (:name($wert), :$param) {...}      # :$param kurz fuer
                                           # :param($param)
```

Enthält die Signatur einer Subroutine eine `@array` Variable, so wird, wie erwartet, auch ein Array als Argument verlangt. In Perl gibt es aber auch einen Sonderfall der Verwendung eines Arrays als Parameter: Steht dem Array ein `*` voraus, so bedeutet dies, dass alle restlichen Argumente, die nicht von anderen Parametern bereits aufgenommen wurden, in dieses Array gespeichert werden. Infolgedessen muss dieser spezielle Parameter natürlich am Ende der Parameterliste stehen.

Wird eine Routine ganz ohne Parameter definiert, so bedeutet dies nicht, dass man keine Argumente beim Aufruf übergeben darf, sondern es ist immer ein standardmäßiger Parameter namens `*@_` definiert, der die Argumente aufnimmt. Möchte man dieses Verhalten unterbinden, so muss eine leere Signatur definiert werden.

Weiterhin gibt es die Form des Parameters mit Stern auch von Hashes. Dieser Hashparameter nimmt nicht alle restlichen Argumente auf, sondern nur die restlichen benannten Argumente.

```
sub fun ($a, $b, *@c) {...}

fun ('A', 'B', 'Das alles', 'in', 'C')

sub no_par {...}
no_par ('a', 'b', 'c')           # @_ := ['a', 'b', 'c']

sub really_no_par () {...}
really_no_par ('a')             # Fehler
```

Man kann auch Skalare Parameter mit einem Stern versehen. Dies bewirkt dann im Zusammenspiel mit Arrayparametern, dass jeweils das erste oder letzte Element des Arrays, das als Argument gegeben wurde, in diesem Skalar gespeichert wird.

```
my @array = <a b c>;
sub is_in($elem, *$head, *@tail) {
    return True if $head eq $elem;
    return False if +@tail le 0;    #+@tail Anzahl der Elemente
    is_in($elem, @tail);
}

is_in('d', @array);               # False
```

In einer Anonymen Routine, die nur als Block (mit `{` und `}`) definiert wurde, kann keine Signatur angegeben werden. Stattdessen kann man hier Platzhaltervariablen verwenden. Das heißt, man verwendet im Programmcode der Routine Variablen und benennt diese mit einem zusätzlichen `^` nach ihrem Typzeichen. Also beispielsweise `skalar` oder `array`. Es wird dann automatisch eine Signatur erzeugt, die als Parameter genau diese Variablen enthält, alphabetisch geordnet.

```
$add = {$^a + $^b}

say &$add(2, 3)                   # 5
```

Trait	Tabelle 5.1: Subroutine Traits Bedeutung
<code>returns / is returns</code>	Return Typ einer Routine
<code>is inline</code>	Routine kann durch Inlining optimiert werden
<code>is tighter</code>	Operator hat Vorrang (Beispiel: <code>sub infix:&lt;*&gt; is tighter&lt;+&gt; {...}</code> )
<code>is looser/is equiv</code>	Analog zu <code>is tighter</code>
<code>is assoc</code>	Festlegen der Assoziativität eines Operators (Beispiel: <code>sub infix:&lt;+&gt; is assoc(left) {...}</code> bedeutet <code>1 + 2 + 3</code> wird als <code>(1 + 2) + 3</code> interpretiert)

Trait	Tabelle 5.2: Parameter Traits Bedeutung
<code>is rw</code>	Argument kann verändert werden
<code>is ref</code>	Parameterübergabe by Reference
<code>is copy</code>	Parameterübergabe by Value

### 5.3 Temporäre Variablen

In Perl 6 kann man mittels dem `temp` Schlüsselwort temporäre Variablen oder Subroutinen definieren. Diese ersetzen, falls nötig, bereits vorhandene Objekte mit dem gleichen Namen. Wird aber der aktuelle Geltungsbereich verlassen, so werden die temporären Variablen gelöscht und der originale Wert wiederhergestellt.

### 5.4 Traits

Traits sind spezielle Zusätze zu Subroutinen oder auch Parametervariablen. Sie können als Eigenschaften gesehen werden, die bereits zur Kompilierzeit zugewiesen werden. Hierbei muss zwischen Traits für Subroutines und Parameter unterschieden werden.

Besonders interessant sind die Subroutine Traits `is tighter/is looser/is equiv` und `is assoc`, denn mit ihnen kann der Vorrang von Operatoren, beziehungsweise die Assoziativität genauer festgelegt werden. Natürlich gibt es noch viel mehr vordefinierte Traits. Hier sei nur ein kleiner Auszug gegeben (siehe Tabelle 5.1 und Tabelle 5.2).

## 5.5 Currying und Wrapping

Mit Currying können einzelne Parameter eines Codeblocks festgelegt werden und somit ein neuer Codeblock erzeugt werden, der zwar die gleiche Funktionalität besitzt, jedoch einige seiner Parameter auf bestimmte Werte oder Variablen festgelegt hat, so dass diese nicht mehr explizit beim Aufruf bestimmt werden müssen. Dazu hat jedes Code Objekt in Perl 6 eine `.assuming` Methode, die dazu verwendet werden kann. Welche Parameter dabei auf welchen Wert festgelegt werden sollen wird dabei durch Paare angegeben.

```
sub exp($x, $n) {...}

&eul := &exp.assuming($x => 2.7)
&eul := &exp.assuming :x(2.7)      # Kurzform
```

Die Methode `per .assuming` einzelne Parameter auf Werte festzulegen ist jedoch leider bei komplizierteren Anwendungen sehr begrenzt. Möchte man beispielsweise einen Parameter nicht fest ersetzen, sondern immer mit einem Faktor multiplizieren, so ist dies nicht möglich.

Perl bietet hierzu die Möglichkeit eine Routine in eine andere sozusagen einzubetten, beziehungsweise die Argumente vor- und die Ergebnisse nachzuverarbeiten. Dazu kann man ein `.wrap` auf die Subroutine anwenden und ein Handle erhalten, das einen dazu befähigt die Änderung auch wieder rückgängig zu machen. Es ist somit möglich beliebig viele "Schichten" um eine Routine herum aufzubauen.

```
sub exp($x, $n) {...}

$handle = &exp.wrap( callwith($x, 2.7) ); # wie oben (Currying)

&exp.unwrap($handle)                    # Rueckgaengig
```

## 5.6 Multisubs

Möchte man eine Routine definieren, die sich für verschiedene Signaturen auch unterschiedlich verhält, muss das Schlüsselwort `multi` verwenden. So kann man mehrere Subroutines erstellen, die den gleichen Namen besitzen jedoch unterschiedliche Signaturen und Funktionalität. Dies ist beispielsweise bei der Definition von eigenen Operatoren sehr nützlich.

```
multi sub postfix:<!> (Num $x) {[*] 1..$x} # Fakultaet
multi sub postfix:<!> (Str $x) {
    die('Nonsense') }
```

## 5.7 Makros

Makros sind spezielle Subroutinen, die schon beim Parsen ausgeführt werden, sobald sie gelesen werden. Das heißt, man kann mit Makros, wie beim C Präprozessor, den Sourcecode zur Kompilierzeit verändern. Dabei wird das Makro ausgeführt und der Rückgabewert ist es ein String, wiederum als Perlcode geparkt. Ist der Rückgabewert ein Syntaxbaum, so wird dieser anstelle des Makroaufrufs in den Syntaxbaum des Programms eingebaut.

Makros haben ein spezielles `is parsed` Trait, das bestimmt, wie das Makro angewendet werden soll. Man kann hier in einer Rule (siehe Kapitel 6) spezifizieren in welcher Weise das Argument gefiltert werden soll.



# Kapitel 6

## Rules

Bereits in Perl 5 sind reguläre Ausdrücke vorhanden, die weitaus mächtiger sind als reguläre Ausdrücke wie sie im traditionellen Sinn reguläre Sprachen beschreiben. In Perl 6 haben sich diese regulären Ausdrücke nocheinmal so stark verändert, dass diese quasi zu einer eigenen Programmiersprache in der Programmiersprache geworden sind. Also musste ein neuer Name her, sodass es nicht zu Verwechslungen kommen kann. Ab nun sind reguläre Ausdrücke in Perl 6 Rules genannt (man sagt aber trotzdem noch manchmal Regex (**R**egular **E**xpression)).

### 6.1 Regexes, Rules und Tokens

Regexes werden definiert entweder im Zusammenhang mit einem Smart Match zur direkten Ausführung oder zur Speicherung in einer Variablen. Dabei konstruiert `/.../` eine anonyme Regex, die je nach Verwendung gespeichert oder auch direkt angewendet werden kann. `m/.../` erstellt eine Regex nur zur direkten Ausführung, genau wie `s/.../.../`, wobei der Teil, der zwischen den vorderen beiden Slashes gematcht wird durch den hinteren Teil ersetzt wird (im hinteren Teil können auch Capture Variablen verwendet werden siehe dazu 6.4).

Nun gibt es aber einen Unterschied zwischen Regexes, wie oben beschrieben, Rules, definiert mit `rule name {...}` und Tokens, definiert mit `token name {...}`. Sind bei Regexes keine Einschränkungen implizit vorgegeben, so sind bei Tokens der `:ratchet` Modifier und bei Rules der `:ratchet` und der `:sigspace` Modifier implizit.

Modifiers sind eine Möglichkeit das Verhalten des Ausdrucks genau zu steuern. Sie werden zu Anfang des Ausdrucks genannt und durch Doppelpunkte abgetrennt. Die zwei interessantesten wurden ja bereits genannt. Es sind `:sigspace`, oder kurz `:s` für die Nichtbeachtung von aufeinanderfolgenden Leerzeichen und `:ratchet`, der Backtracking ausschaltet. Weiterhin ist der `:nth` Modifier auch sehr interessant, da mit ihm der Match an einer

bestimmten Stelle gefunden werden kann. Das heißt, hat eine Regex einen `:2th` Modifier, so wird nur erfolgreich gematcht, wenn der Ausdruck zweimal vorkommt.

## 6.2 Grammars

Grammatiken sind eine Gruppierung von Regexes, Rules und Tokens. Diese drei Typen können so zusammengefasst und zu sinnvollen Konstrukten kombiniert werden. Zur Definition einer Grammatik benötigt man das Schlüsselwort `grammar` und den Namen einer Grammatik. Anschließend kann auf die Elemente der Grammatik (die Rules) auch von außerhalb zugegriffen werden. Grammatiken sind vor allem dann sehr nützlich, wenn Text oder komplexe Daten geparkt werden sollen.

## 6.3 Metacharacters

Metacharacters sind keine normalen Zeichen oder Zahlen. Sie stehen entweder für bestimmte Zeichen, für bestimmte Textkonstrukte (Zeilenanfang, Zeilenende, ...) oder aber sind Syntaktische Elemente der Regex.

Grundsätzlich gibt es drei verschiedene Kategorien von Metacharacters. Die erste wären Metacharacters, die für einzelne Zeichen stehen (wie `.` für alle Zeichen beispielsweise, oder `^` für den Anfang einer Zeile). Struktur- und Syntaxelemente (`|`, `&`, `\` ...) können dazu verwendet werden alternative Patterns anzugeben oder logisch zu gruppieren und bilden damit die zweite Kategorie.

Die dritte Kategorie sind die Klammern, von denen es wiederum vier Typen gibt. Die runden Klammern (`()`), die Patterns gruppieren und Capturing (6.4) durchführen. Patterns zu gruppieren ist beispielsweise nötig, um längere Alternativen für den `|` Metacharacter abzugrenzen. Dasselbe tun die eckigen Klammern (`[]`), nur mit dem Unterschied, dass diese kein Capturing durchführen und nur gruppieren.

Möchte man Perl 6 Code innerhalb einer Regex ausführen, so muss dieser in geschweifte Klammern (`{}`) eingeschlossen werden. Der Rückgabewert des Codes wird dabei nicht als Regex gewertet und weiter ausgeführt, sondern die geschweiften Klammern erzeugen ein Atom, das immer gematcht wird. Möchte man, dass der Rückgabewert des Codes als Regex gewertet wird, so muss man eine spezielle Assertion anwenden, die durch spitze Klammern (`<>`) in Kombination mit weiteren Elementen angezeigt wird.

Tabelle 6.1 zeigt eine Auflistung der Metacharacters.

Tabelle 6.1: Auflistung der Metacharacters

Symbol	Bedeutung
.	Einzelnes Zeichen
^	Anfang eines Strings
\$	Ende eines Strings
^^	Anfang einer Zeile
\$\$	Ende einer Zeile
	Oder Verknüpfung von Patterns
&	Und Verknüpfung von Patterns
\	Durchführen eines Escapings (siehe 6.3.1)
#	Kommentarzeichen
:=	Ergebnis eines Matches einer Variable zuweisen (siehe 6.4)
(...)	Gruppierung (mit Capturing)
[...]	Gruppierung (ohne Capturing)
{...}	Ausführen von Code
<...>	Assertion

### 6.3.1 Escaping

Wie in Tabelle 6.1 zu sehen führt der Metacharacter `\` Escaping durch. Doch was ist Escaping?

Durch Escaping kann ein Metacharacter, der auf `\` folgt, als ein normales Zeichen gekennzeichnet werden. Das heißt, es hat nun für das Pattern keine besondere Bedeutung mehr und steht nur noch für ein Textzeichen im String, der gematcht werden soll. Der umgekehrte Weg ist natürlich auch möglich. Dabei wird aus einem Textzeichen eine Escape Sequenz gemacht, die meist für Zeichen stehen, die schwierig sind in Buchstaben auszudrücken (zum Beispiel `\t` für ein Tabulatorzeichen) oder für ganze Wörter (`\w`) und Zeichenklassen (Zahlen `\d`).

### 6.3.2 Quantifiers

Mit Quantifiern kann angegeben werden, wie oft ein Pattern wiederholt gematcht werden soll. So kann man aufeinanderfolgende gleiche Stücke matchen, die in einer bestimmten Anzahl vorkommen. Dabei gibt es mehrere Typen von Quantifiern. Zum einen `*` (beliebig oft), `?` (kein oder ein Match), `+` (einmal oder beliebig oft), zum anderen ein Range Objekt abgeschlossen durch `**{` und `}`, die die maximale und minimale Anzahl von Wiederholungen angibt. Es ist nun natürlich möglich, dass ein Quantifiern eine unterschiedliche Anzahl von Übereinstimmungen hat. Zum Beispiel wird die Zeichensequenz `äaa` von `a+` ein-, zwei- oder dreimal gematcht. Möchte man nun den maximalen, das heißt den längsten Match erhalten, so muss nichts

Tabelle 6.2: Verwendungsmöglichkeiten für Assertions (nicht vollständig)

Klammerung	Bedeutung
<code>&lt;...&gt;</code>	Matchen einer Subrule
<code>&lt;\$var&gt;</code>	Matchen einer anonymen Subrule, die in <code>\$var</code> gespeichert ist
<code>&lt;@arr&gt;</code>	Vergleichbar mit <code>&lt;\$var&gt;</code> , nur wird jedes Element des Arrays einzeln gematcht (man könnte auch <code>&lt;\$arr[0]&gt; &lt;\$arr[1]&gt; &lt;\$arr[2]&gt;... schreiben</code> )
<code>&lt;%hash&gt;</code>	Zuerst matchen des Keys, dann matchen des Value Teils. Dabei ist garantiert, dass der Key verwendet wird, der den längsten Teil matcht
<code>&lt;{func()}&gt;</code>	Aufruf der Subroutine oder Methode und Interpretation des Rückgabewertes als Regex (Kurzform: <code>&lt;&amp;func()&gt;</code> )

gesondert angegeben werden. Möchte man den minimalen Match erhalten, so muss noch ein `?` angegeben werden (also `a+?`). Dies funktioniert mit allen Quantifiern.

### 6.3.3 Assertions

Assertions sind die mächtigste Art von Klammern in Perl 6 Rules. Demnach haben diese auch die verschiedensten Verwendungsmöglichkeiten. Zum einen können mit Hilfe der spitzen Klammern einfach Subrules angegeben werden, das heißt, dass Rules, die zu einem früheren Zeitpunkt definiert worden sind, wieder verwendet werden können, zum anderen können auch viel kompliziertere Konstrukte, wie beispielsweise Perl Code ausgeführt werden, dessen Rückgabewert als Regex ausgewertet wird. Tabelle 6.2 gibt einen Überblick, was mit Assertions möglich ist. Dabei ist zu beachten, dass das erste Zeichen, das nach der öffnenden Klammer folgt, über den Typ der Assertion bestimmt.

## 6.4 Capturing

Hat man einen komplexen Ausdruck als Regex formuliert und dabei eine Gruppierung mit runden Klammern verwendet, so kann man die Werte, die von einzelnen Gruppen gematcht werden, einzeln einsehen. Es werden dann automatisch Variablen erstellt, beginnend mit `$0`, `$1`, `...`, die die Werte der einzelnen Gruppen enthalten (eigentlich ist `$0` nur eine Kurzform für `$/[0]`). Dabei ist `$/` das Match Objekt, das alle Rückgabewerte des Matches enthält).

## 6.5 Beispiele zu Rules

```

# Beispiel einer Grammatik fuer einen Taschenrechner
# in polnischer Notation

grammar pol_not {
  token pol {<pol_term>\n};
  token pol_term {
    <pol_term> <pol_term> <op>|
    <number> };
  token op {+|-|*|};
  token number {\d+};
}

# Verdeutlichung des Sigspace Modifier

my $re1 = rule {auf ein wort}; # :sigspace modifier implizit
my $re2 = regex {auf <?ws> ein <?ws> wort};
          # <ws> matcht Leerzeichen, kann undefiniert
          werden

# Beispiel zu Assertions

my $re3 = rule {[<[a..z]>|<[A..Z]>|<[0..9]>]*}; # Zahlen und Buchstaben

# Saetze erkennen: besonders interessant ist hier die Verwendung des |
# Metacharacters, der alternativen zulaesst, sowie das Capturing in die
# Standartvariablen $0 und $1.

my %zuord = ("Bohrer" => "bohren",
            "Hammer" => "nageln",
            "Zange" => "zwicken");
rule werkzeug {Bohrer|Hammer|Zange};
rule methode {bohren|nageln|zwicken};

my $string = "Mit dem Bohrer kann man bohren";
my $string1 = "Mit der Zange kann man bohren";

$string ~~ rule {Mit [dem|der] (<werkzeug>) kann man (<methode>)};
# $0 := "Bohrer", $1 := "bohren"
%zuord<$0> == $1 # True

$string1 ~~ rule {Mit [dem|der] (<werkzeug>) kann man (<methode>)};
%zuord<$0> == $1 # False

```



## Kapitel 7

# Zusammenfassung

Mit der neuen Version haben viele neue Konzepte und Funktionen Einzug in Perl gehalten. Dadurch ist es möglich “schnell mal was zu hacken”, sowie effizient zu Programmieren. Eine effiziente und sichere Programmierweise ist mit dem Typsystem möglich. Rules ermöglichen ein umfangreiches Parsing, eines der großen Einsatzgebiete von Perl, und sind dank der Gruppierung in Grammars und einer nicht so strikten Behandlung von Leerzeichen (beim `:sigspace` Modifier) auch besser zu lesen. Aber auch die Perl-Freaks müssen sich nicht fürchten, denn es ist natürlich weiterhin möglich fast unleserliche Perl-Golfs zu schreiben.

Leider lässt eine vollständige Implementierung der Sprache wohl noch eine Weile auf sich warten.



# Literaturverzeichnis

- [1] Tang, Audrey, “Pugs - Bootstrapping Perl 6 with Haskell”, *Haskell'05*, September, 2005.
- [2] Randal, Allison Sugalski, Alan Tötsch, Leopold, “Perl 6 and Parrot”, *O'Reilly Media, Inc.*, Sebastopol, 2004.
- [3] Wall, Larry, “Perl 6 Synopses”, [online], 2007. Verfügbar unter: <http://dev.perl.org/perl6/doc/synopsis.html>