

# Perl 6

- ✓ Entwicklungstand
- ✓ Gestaltungsprinzipien
- ✓ Beispiele



# Warum ?

- Es ist die coolste und durchdachteste Sprache.
- Eine Bereicherung für jeden denkenden Geist.
- Hat glücklicherweise was mit Perl zu tun.

# Teil 1 – das Projekt

- Entwicklung
  - Stand
- Beobachtungen

# Wie alles begann ...

- Juli 2000 kündigt Larry Perl6 an, denn:
  - Interna schwer wartbar
  - XS ist Hinderniss für Perlmodule
  - Syntax braucht Säuberung
    - angehängte Feature : `do {{ }} while`
    - Inkonsistenzen : token, höhere Regex feature
    - anfängerunfreundlich : sigils, deref, ! Signatur
    - Wurmfortsätze : `Package'sub`
  - Entwicklung ging weiter : Haskell Ruby Python

# Weiterer Verlauf ...

- Parrot startete 2002 (langsam & stet gen Ziel)
  - VM für viele Sprachen
- Ponie 2003 ausgerufen (jetzt offiziell begraben)
  - Perl5 Interpreter auf Parrot, verbesserte Perl5
- Pugs startete Februar 2005 (sehr schnell)
  - Interpreter in Haskell
  - evalbot sehr nützlich
- erste Lebenszeichen des Übersetzers 2006 (MAD)

# Sprachentwicklung

- 2000 gabs eine Einladung mitzumachen  
... keine Releaseankündigung

# Sprachentwicklung

- 2000 gabs eine Einladung mitzumachen  
... keine Releaseankündigung
- öffentlicher Designprozess, (ca. 2Jahre)
  - Larrys Apokalypsen, Damians Exegesen

# Sprachentwicklung

- 2000 gabs eine Einladung mitzumachen  
... keine Releaseankündigung
- öffentlicher Designprozess, (ca. 2Jahre)
  - Larrys Apokalypsen, Damians Exegesen
- seit dem Teamarbeit an einzelnen Themen
  - in Maillisten an Synopsen
  - immer wieder änderungen
  - Gesamtbild konsistent halten
  - Stark mit Pugs verknüpft : Tests, Docs, Beispiele



# Parrot – die VM

- von Dan Sugalski (VMS Port) entworfen und begonnen, verließ leider Projekt

# Parrot – die VM

- von Dan Sugalski (VMS Port) entworfen und begonnen, verließ leider Projekt
- rel. geschlossenes Projekt mit wenigen Mitarbeitern : Chip Salzenberg(Perl 5.4), Leo Tötsch, Allison Randal(TPF), Luke Palmer, neuer Leiter MichaelP

# Parrot – die VM

- von Dan Sugalski (VMS Port) entworfen und begonnen, verließ leider Projekt
- rel. geschlossenes Projekt mit wenigen Mitarbeitern : Chip Salzenberg(Perl 5.4), Leo Tötsch, Allison Randal(TPF), Luke Palmer, neuer Projektleiter Patrick Michaud
- Auch die VM entwickelt sich schnell
  - ... verglichen mit Java (fast 10 J. Vorarbeit)
  - ... und gemessen am anspruchsvollen Ziel (moderere Technik, viele Sprachen, PASM, PIL)

# Generationen

- Perl6 ist nächste Generation / Revision
  - gibt Perspektive und Hoffnung
  - Spielwiese für Kinder der Zukunft
- nächste Version ist
  - 5.10 kommt planmäßig
  - und profitiert sehr von Perl6
    - err(//) - Operator (if defined)*
    - case - Anweisung (given when)*
  - *sanfterer Übergang*

# Perl heute

- Perl 5 wird nicht mehr gehyped

# Perl heute

- Perl 5 wird nicht mehr gehyped, also tot wie BSD

# Perl heute

- Perl 5 wird nicht mehr gehyped, also tot wie BSD
- Perl 5 geht's gut

# Perl heute

- Perl 5 wird nicht mehr gehyped, also tot wie BSD
- Perl 5 geht's gut
  - Interpreterentwicklung ist stabil, vorhersehbar
  - CPAN gedeiht, Standard erhöht sich (Phalanx, annotated)
  - viele schöne Entwicklungen
    - WebGUI / Catalyst / Jifty / Kephra u.v.m.



# Perl heute

- Perl 5 wird nicht mehr gehyped, also tot wie BSD
- Perl 5 geht's gut
  - Interpreterentwicklung ist stabil, vorhersehbar
  - CPAN gedeiht, Standard erhöht sich (Phalanx, annotated)
  - viele schöne Entwicklungen
    - WebGUI / Catalyst / Jifty / Kephra u.v.m.
- Keine Panik : Perl6 hat Zeit zur Entwicklung

# Teil 2 – die Sprache

- Perl Tugenden
- Designprinzipien
- Sprachumfang

# Perl Tugenden

- TIMTOWTDI

# Perl Tugenden

- TIMTOWTDI : läßt viele Wege

# Perl Tugenden

- TIMTOWTDI : Bescheidenheit der Sprache
  - ohne künstl. Grenzen, alles frei kombinierbar
  - Vertrauen in den Schaffenden (seine Intuition)
  - gute Gefühl zu coden wie man denkt

# Perl Tugenden

- TIMTOWTDI : Bescheidenheit der Sprache
  - ohne künstl. Grenzen, alles frei kombinierbar
  - Vertrauen in den Schaffenden (seine Intuition)
  - gute Gefühl zu coden wie man denkt
- einfaches einfach, schweres möglich
  - minimalistisch (genügsam wie ein Kamel)
  - flexibel, reichhaltig (schweizer Kettensäge)
  - für Praktiker („get the Job done“)

# Perl Tugenden

- TIMTOWTDI : Bescheidenheit der Sprache
  - ohne künstl. Grenzen, alles frei kombinierbar
  - Vertrauen in den Schaffenden (seine Intuition)
  - gute Gefühl zu coden wie man denkt
- einfaches einfach, schweres möglich
  - minimalistisch (genügsam wie ein Kamel)
  - flexibel, reichhaltig (schweizer Kettensäge)
  - für Praktiker („get the Job done“)

Ziel von Perl6 :

Tugenden nicht aufgegeben, sondern erneuert

# Neue Tugend

- nicht wirklich neu
- teilweisebereits verwirklicht
- eher Schwerpunkte der Umgestaltung



# Neue Tugend

Einfachheit und Konsistenz :

- nur teilweise erreicht

# Neue Tugend

Einfachheit und Konsistenz :

- nur teilweise erreicht
- Perl orientierte sich an C, Shell, Unix tools
- 19 Jahre nachträglich zugefügte Feature
- oft keine einheitliche Sprache
- Regex, Operatoren, Token, Pragma, Spez. Var.

# Neue Tugend

## Einfachheit und Konsistenz :

- nur teilweise erreicht
- Perl orientierte sich an C, Shell, Unix tools
- 19 Jahre nachträglich zugefügte Feature
- oft keine einheitliche Sprache
- Regex, Operatoren, Token, Pragma, Spez. Var.
  
- schwerer erlernbar : viele Regeln, Sonderfälle
- schwerer zu parsen -> startet langsamer

# Neue Tugend

## Einfachheit und Konsistenz :

- nur teilweise erreicht
- Perl orientierte sich an C, Shell, Unix tools
- 19 Jahre nachträglich zugefügte Feature
- oft keine einheitliche Sprache
- Regex, Operatoren, Token, Pragma, Spez. Var.
  
- schwerer erlernbar : viele Regeln, Sonderfälle
- schwerer zu parsen -> startet langsamer
  
- natürliche und logische Grundprinzipien (Linguistik)
- kombinierbar zur freien Anwendung (Tim Today)

# Neue Tugend

menschlich verständlich :

# Neue Tugend

menschlich verständlich :

- optisch übersichtlich
  - ähnliche Dinge sehen ähnlich aus
  - verschiedene Dinge verschieden
  - verrückte Dinge sollen verrückt aussehen
  - wichtige Dinge sollen sich heben (label, topicalizer)
  - unwichtiges weglassen : if  $x > 5$  { ... }, }

# Neue Tugend

menschlich verständlich :

- optisch übersichtlich
  - ähnliche Dinge sehen ähnlich aus
  - verschiedene Dinge verschieden
  - verrückte Dinge sollen verrückt aussehen
  - wichtige Dinge sollen sich heben (label, topicalizer)
  - unwichtiges weglassen : `if $x > 5 { ... }, }` enthält immer ;
- Visuelle Metaphern
  - `==>`, `<==` (pipe operatoren)
  - `Y`, (zip operator)

# Neue Tugend

menschlich verständlich :

- optisch übersichtlich
  - ähnliche Dinge sehen ähnlich aus
  - verschiedene Dinge verschieden
  - verrückte Dinge sollen verrückt aussehen
  - wichtige Dinge sollen sich herausheben (label, topicalizer)
  - unwichtiges weglassen : if  $x > 5$  { ... }, } enthält immer ;
- Visuelle Metaphern
  - $\Rightarrow$ ,  $\Leftarrow$  (pipe operatoren)
  - Y, (zip operator)
- einfaches englisch
  - loop {}, given \$a when 5, has method, wie bereits @ISA



# Neue Tugend

klare Vorgaben :

# Neue Tugend

klare Vorgaben :

- hat die Sprache oft nicht gesetzt
- TIMTOWTDI war zu sehr wie TCL ausgelegt
- Betrifft vor allem OOP, parameter von subs
- amorpher Brei an Möglichkeiten
- zu wenig Vorgaben verwirrt Anfänger
- klare Ansagen verbessern Verständigung
- Standards = Feature, sparen Arbeit
- nichts neues : map, grep, for
- Genauso jetzt : module, package, class
- eindeutiger: .. => .., ff, fff(...)
- aber auch mehr zu lernen

# Neue Tugend

klare Vorgaben :

- manche Grenze kannte Perl5 nicht
- privaten Methoden, Konstanten oder Typen
- nur mit Aufwand machbar (Scopes)
  
- Sprache erweitern : macros, traits
  
- Rest der Welt nicht aus Augen verlieren : -> wird zu .
  
- Hauptorientierung: english, perl5, computer science

# Ziele

- wachstumsbedingte Altlast aus 19 J. Aufräumen
- einfacher für Anfänger (use strict & -w default)
  - Einfacher bei Sigils und Dereferenzierung, OOP
  - simple english
- weniger Regeln, mehr Konsistenz (Standarts)
  - einfacher für Maschinen
    - only perl can parse Perl > one pass parsing
- viel mehr Spielsachen für Fortgeschrittene :

# buzzwordkompatibel

- OOP wie Ruby
- AOP besser als in C#
- funktional wie Haskell
- Kontrakte wie Eiffel
- Exceptions einfacher als in Java
- Parsetrees wie mit lex und yacc
- typisierter als Python
- mächtiges Perldoc System
- glue lang im Sinne alten Code besser warten
- oder trickreich wie bisher in Perl5

# buzzwordkompatibel

- OOP wie Ruby
- AOP besser als in C#
- funktional wie Haskell
- Kontrakte wie Eiffel
- Exceptions einfacher als in Java
- Parsetrees wie mit lex und yacc
- typisierter als Python
- mächtiges Perldoc System
- glue lang im Sinne alten Code besser warten
- oder trickreich wie bisher in Perl5

Das ist TIMTOWTDI 2006, 2007, 2008 ....

# buzzwordkompatibel

- OOP wie Ruby
- AOP besser als in C#
- funktional wie Haskell
- Kontrakte wie Eiffel
- Exceptions einfacher als in Java
- Parsetrees wie mit lex und yacc
- typisierter als Python
- mächtiges Perldoc System
- glue lang im Sinne alten Code besser warten
- oder trickreich wie bisher in Perl5

„all your Paradigms belong to us“

# Teil 3 – Syntax

- Golf
- neue Regeln
- idiomatisches Perl6



# Kopfsprung

```
#!/usr/bin/perl
use strict;
use warnings;

sub biggest_of_3 {
    my $a = shift;
    my $b = shift;
    my $c = shift;
    if ($a > $b and $a > $c) { return $a }
    elsif ($b > $a and $b > $c) { return $b }
    else { return $c }
}

print „Geben sie Bitte 3 Zahlen ein: \n“;
$werte[ $_ ] = <STDIN> for 0..2;
print "Groessten Zahlwert hatte : " . biggest_of_3(@werte) . "\n";
```

# Testing Pugs

```
#!/usr/bin/pugs
```

```
use strict;
```

```
use warnings;
```

```
sub biggest_of_3 {
```

```
    my $a = shift;
```

```
    my $b = shift;
```

```
    my $c = shift;
```

```
        if ($a > $b and $a > $c) { return $a }
```

```
        elsif ($b > $a and $b > $c) { return $b }
```

```
        else { return $c }
```

```
}
```

```
print „Geben sie Bitte 3 Zahlen ein: \n“;
```

```
$werte[ $_ ] = <STDIN> for 0..2;
```

```
print "Groessten Zahlwert hatte : " . biggest_of_3(@werte) . "\n";
```

# strict & -w default

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 {  
  my $a = shift;  
  my $b = shift;  
  my $c = shift;  
  if ($a > $b and $a > $c) { return $a }  
  elsif ($b > $a and $b > $c) { return $b }  
  else { return $c }  
}  
  
print „Geben sie Bitte 3 Zahlen ein: \n“;  
$werte[ $_ ] = <STDIN> for 0..2;  
print "Groessten Zahlwert hatte : " . biggest_of_3(@werte) . "\n";
```

- Explizites no strict

# sub mit Signatur

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
    if ($a > $b and $a > $c) { return $a }  
    elsif ($b > $a and $b > $c) { return $b }  
    else { return $c }  
}
```

```
print „Geben sie Bitte 3 Zahlen ein: \n“;  
$werte[ $_ ] = <STDIN> for 0..2;  
print "Groessten Zahlwert hatte : " . biggest_of_3(@werte) . "\n";
```

- wie der Rest der Welt es auch macht
- Parameter können optional(name?), schreibbar(is rw), typisiert sein
- Perl5 verhalten und slurpy Array (\*@werte)

# opt. Paranthesis

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if $a > $b and $a > $c { return $a }  
  elsif $b > $a and $b > $c { return $b }  
  else { return $c }  
}
```

```
print „Geben sie Bitte 3 Zahlen ein: \n“;  
$werte[ $_ ] = <STDIN> for 0..2;  
print "Groessten Zahlwert hatte : " . biggest_of_3(@werte) . "\n";
```

- auch bei schleifen etc.

# chaining ops

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
    if $a > $b == $a > $c { return $a }  
    elsif $b > $a == $b > $c { return $b }  
    else { return $c }  
}  
  
print „Geben sie Bitte 3 Zahlen ein: \n“;  
$werte[ $_ ] = <STDIN> for 0..2;  
print "Groessten Zahlwert hatte : " . biggest_of_3(@werte) . "\n";
```

- $3 < \$a < 7$  ist andere Form von chaining

# print statt say

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if $a > $b == $a > $c { return $a }  
  elsif $b > $a == $b > $c { return $b }  
  else { return $c }  
}
```

```
say 'Geben sie Bitte 3 Zahlen ein:';  
$werte[ $_ ] = <STDIN> for 0..2;  
say 'Groessten Zahlwert hatte : ' . biggest_of_3(@werte);
```

- Ein REXX feature (Ruby sagt puts)

# ~ statt .

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if $a > $b == $a > $c { return $a }  
  elsif $b > $a == $b > $c { return $b }  
  else { return $c }  
}
```

```
say 'Geben sie Bitte 3 Zahlen ein:';  
$werte[ $_ ] = <STDIN> for 0..2;  
say 'Groessten Zahlwert hatte : ' ~ biggest_of_3(@werte);
```

- „~“ steht für Strinkontext (ausser x)



# Array immer mit @

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if $a > $b == $a > $c { return $a }  
  elsif $b > $a == $b > $c { return $b }  
  else { return $c }  
}
```

```
say 'Geben sie Bitte 3 Zahlen ein:';  
@werte[ $_ ] = <STDIN> for 0..2;  
say 'Groessten Zahlwert hatte : ' ~ biggest_of_3(@werte);
```

- Hashes natürlich mit %

# STDIN ist glob. Var

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if $a > $b == $a > $c { return $a }  
  elsif $b > $a == $b > $c { return $b }  
  else { return $c }  
}
```

```
say 'Geben sie Bitte 3 Zahlen ein:';
```

```
@werte[ $_ ] = <$*IN> for 0..2;
```

```
say 'Groessten Zahlwert hatte : ' ~ biggest_of_3(@werte);
```

- „\*“ ist sekundäre Sigil der globalen Variablen, keine typeglobs mehr

# prefix:<=> ist Iterator

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if $a > $b == $a > $c { return $a }  
  elsif $b > $a == $b > $c { return $b }  
  else { return $c }  
}
```

```
say 'Geben sie Bitte 3 Zahlen ein:';  
@werte[ $_ ] = = $*IN for 0..2;  
say 'Groessten Zahlwert hatte : ' ~ biggest_of_3(@werte);
```

- Prefix-Operator „=“ ist Iterator, was vorher < > war

# unary range op.

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if    $a > $b == $a > $c { return $a }  
  elsif $b > $a == $b > $c { return $b }  
  else                               { return $c }  
}
```

```
say 'Geben sie Bitte 3 Zahlen ein:';
```

```
@werte[ $_ ] = = $*IN for ^3;
```

```
say 'Groessten Zahlwert hatte : ' ~ biggest_of_3(@werte);
```

- for  $0^{\dots 4}$  ginge auch,  $^$  schliesst grenzwerte aus
- $2.5 \sim\sim 2^{\dots 3}$  ist wahr, aber nicht  $3 \sim\sim 2^{\dots 3}$

# Kleine Variation

```
#!/usr/bin/pugs
```

```
say 'Geben sie Bitte 3 Zahlen ein:';  
@werte[ $_ ] == $*IN for ^3;
```

```
if    [>] @werte { say 'Die Eingabe war absteigend sortiert.' }  
elsif [<] @werte { say 'Die Eingabe war aufsteigend sortiert.' }  
else          { say 'Die Eingabe war nicht sortiert.' }
```

- Operatoren in eckigen Klammern sind Reduktionoperatoren
- [**<**] @w == @w[0] < @w[1] && @w[1] < @w[2] && @w[2] < @w[3]

# nochmal Perl5

```
#!/usr/bin/perl
use strict;
use warnings;

sub biggest_of_3 {
    my $a = shift;
    my $b = shift;
    my $c = shift;
    if ($a > $b and $a > $c) { return $a }
    elsif ($b > $a and $b > $c) { return $b }
    else { return $c }
}

print „Geben sie Bitte 3 Zahlen ein:\n“;
$werte[ $_ ] = <STDIN> for 0..2;
print "Groessten Zahlwert hatte : " . biggest_of_3(@werte) . "\n";
```

# Endfassung

```
#!/usr/bin/pugs
```

```
sub biggest_of_3 ($a, $b, $c) {  
  if $a > $b == $a > $c { return $a }  
  elsif $b > $a == $b > $c { return $b }  
  else { return $c }  
}
```

```
say 'Geben sie Bitte 3 Zahlen ein:';
```

```
@werte[ $_ ] = = $*IN for ^3;
```

```
say 'Groessten Zahlwert hatte : ' ~ biggest_of_3(@werte);
```

# Unterschied

- Golf:  $376 - 294 = 82$  Zeichen  $\sim 22\%$  Einsparung
- besser lesbar weil auf wesentliches beschränkt
- einiges perlischer
- vieles blieb gleich



# Unterschied

- Golf:  $376 - 294 = 82$  Zeichen  $\sim 22\%$  Einsparung
- besser lesbar weil auf wesentliches beschränkt
- einiges perlischer
- vieles blieb gleich
  
- Noch einige kleinere Beispiele:

# Noch ein Beispiel

```
#!/usr/bin/perl
```

```
for my $a ( qw( 3 5 8 12 16 19 21 ) ) {  
    if ( $a == 2 or $a == 12 or $a == 21 )  
        { print "bitte zum gelben Schalter\n" }  
    elsif ( $a == 5 or $a == 16 or $a == 19 )  
        { print "bitte zum pinken Schalter\n" }  
    else { print "bitte komen sie später wieder\n"}  
}
```

Asterix (erobert Rom) vor dem Gebäude mit den vielen Schalter

# Klammern ...

```
#!/usr/bin/pugs
```

```
for my $a qw( 3 5 8 12 16 19 21 ) {  
    if $a == 2 or $a == 12 or $a == 21  
        { print "bitte zum gelben Schalter\n" }  
    elsif $a == 5 or $a == 16 or $a == 19  
        { print "bitte zum pinken Schalter\n" }  
    else { print "bitte komen sie später wieder\n"}  
}
```

wißt ihr ja schon

# say ...

```
#!/usr/bin/pugs
```

```
for my $a qw( 3 5 8 12 16 19 21 ) {  
  if $a == 2 or $a == 12 or $a == 21  
    { say 'bitte zum gelben Schalter' }  
  elsif $a == 5 or $a == 16 or $a == 19  
    { say 'bitte zum pinken Schalter' }  
  else { say 'bitte komen sie später wieder' }  
}
```

wißt ihr ja schon

# qw wird zu <>

```
#!/usr/bin/pugs
```

```
for my $a < 3 5 8 12 16 19 21 > {  
  if $a == 2 or $a == 12 or $a == 21  
    { say 'bitte zum gelben Schalter' }  
  elsif $a == 5 or $a == 16 or $a == 19  
    { say 'bitte zum pinken Schalter' }  
  else { say 'bitte komen sie später wieder' }  
}
```

steht generell für Metasyntax

# pointy sub

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > -> $a {  
  if $a == 2 or $a == 12 or $a == 21  
    { say 'bitte zum gelben Schalter' }  
  elsif $a == 5 or $a == 16 or $a == 19  
    { say 'bitte zum pinken Schalter' }  
  else { say 'bitte komen sie später wieder' }  
}
```

automatisch lexikalische Variable

# pointy sub

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > -> sub ($a) {  
  if $a == 2 or $a == 12 or $a == 21  
    { say 'bitte zum gelben Schalter' }  
  elsif $a == 5 or $a == 16 or $a == 19  
    { say 'bitte zum pinken Schalter' }  
  else { say 'bitte komen sie später wieder' }  
}
```

block der schleife ist wie eine sub

# autonamed Par.

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > {  
  if    $^a == 2 or $^a == 12 or $^a == 21  
    { say 'bitte zum gelben Schalter' }  
  elsif $^a == 5 or $^a == 16 or $^a == 19  
    { say 'bitte zum pinken Schalter' }  
  else { say 'bitte komen sie später wieder'}  
}
```

automatische Parameter siehe sort



# pointy sub

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > -> $a {  
  if $a == 2 or $a == 12 or $a == 21  
    { say 'bitte zum gelben Schalter' }  
  elsif $a == 5 or $a == 16 or $a == 19  
    { say 'bitte zum pinken Schalter' }  
  else { say 'bitte komen sie später wieder' }  
}
```

automatisch lexikalische Variable

# Junctions

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > -> $a {  
  if $a == 2 | 12 | 21 { say 'bitte zum gelben Schalter' }  
  elsif $a == 5 | 16 | 19 { say 'bitte zum pinken Schalter' }  
  else { say 'bitte komen sie später wieder'}  
}
```

sehr praktisch gell

# als case Anweisung

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > -> $a {  
  given $a {  
    when 2 | 12 | 21 { say 'bitte zum gelben Schalter' }  
    when 5 | 16 | 19 { say 'bitte zum pinken Schalter' }  
    default          { say 'bitte komen sie später wieder'}  
  }  
}
```

darauf haben einige lange gewartet 😊

# als case Anweisung

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > -> $a {  
  given $a {  
    if $_ =~ 2 | 12 | 21 { say 'bitte zum gelben Schalter' }  
    elsif $_ =~ 5 | 16 | 19 { say 'bitte zum pinken Schalter' }  
    else { say 'bitte komen sie später wieder'}  
  }  
}
```

- when führt nur smart match gegen topic aus

# doppelter Topicalizer

```
#!/usr/bin/pugs
```

```
for < 3 5 8 12 16 19 21 > {  
  when 2 | 12 | 21 { say 'bitte zum gelben Schalter' }  
  when 5 | 16 | 19 { say 'bitte zum pinken Schalter' }  
  default          { say 'bitte komen sie später wieder'}  
}
```

dann lassen wir doch einen weg

# Noch mal Perl5

```
#!/usr/bin/perl
```

```
for my $a ( qw( 3 5 8 12 16 19 21 ) ) {  
    if ( $a == 2 or $a == 12 or $a == 21 )  
        { print "bitte zum gelben Schalter\n" }  
    elsif ( $a == 5 or $a == 16 or $a == 19 )  
        { print "bitte zum pinken Schalter\n" }  
    else { print "bitte komen sie später wieder\n"}  
}
```

Golf : 30%; wesentlich klarer

# object oriented Perl

```
#!/usr/bin/perl
package arv::scic;

sub new {
    bless {speed => 0 }, shift;
}

sub speed {
    my $self = shift;
    my $speed = shift;
    if ($speed) { $self->{speed} = $speed }
    else      { $self->{speed} }
}

sub stop { $_[0]->{speed} = 0 }
```

# object oriented Pugs

```
#!/usr/bin/pugs
package arv::scic;

sub new {
    bless {speed => 0 }, shift;
}

sub speed {
    my $self = shift;
    my $speed = shift;
    if ($speed) { $self->{speed} = $speed }
    else      { $self->{speed} }
}

sub stop { $_[0]->{speed} = 0 }
```



# Klassen heissen auch so

```
#!/usr/bin/pugs
class arv::scic {

sub new {
    bless {speed => 0 }, shift;
}

sub speed {
    my $self = shift;
    my $speed = shift;
    if ($speed) { $self->{speed} = $speed }
    else      { $self->{speed} }
}

sub stop { $_[0]->{speed} = 0 }
}
```

# keine new Methoden

```
#!/usr/bin/pugs
class arv::scic {

  has $.speed;

  sub stop { $_[0]->{speed} = 0 }

}
```

- auch kostenlose Accesoren und klonen

# Methoden heissen so

```
#!/usr/bin/pugs  
class arv::scic {  
  
    has $.speed;  
  
    method stop { $.speed = 0 }  
  
}
```

- Genug Wege für hybride Interfaces

# object oriented Perl

```
#!/usr/bin/perl
package arv::scic;

sub new {
    bless {speed => 0 }, shift;
}

sub speed {
    my $self = shift;
    my $speed = shift;
    if ($speed) { $self->{speed} = $speed }
    else      { $self->{speed} }
}

sub stop { $_[0]->{speed} = 0 }
```

# Achtung Falle

- `$a[3]`

# Achtung Falle

- \$a[3] => @a[3] # konsistente Sigils
- %a {'b'}

# Achtung Falle

- `$a[3]`      =>    `@a[3]`      # konsistente Sigils
- `%a {'b'}`   =>    `%a .<b>`      # whitespace matters
- `sub {}`

# Achtung Falle

- `$a[3]`      =>    `@a[3]`      # konsistente Sigils
- `%a {'b'}`   =>    `%a .<b>`    # whitespace matters
- `sub {}`      =>    `sub { ... }`    # keine leeren subs
-



# Neue Regeln : Var

- nur noch \$skalare, @arrays, %hashes, &coderef
  - konsistente sigils, intern Objekte
  - Formate, Handler etc. sind \$obj\_ref
  - kombinierbar mit sekundären Sigils
- zusätzliche Eigenschaften
  - optionale Datentypen
  - Arraydimensionierung, Konstanten
  - 0 but true
  - compiletime:traits; runtime:properties

# Neue Regeln : Op

- einige mehr, konsistenter, Leerzeichen wichtig
  - ? boolescher Kontext;  $?: \Rightarrow ??!$
  - + numerischer Kontext;  $| \Rightarrow +|$
  - ~ string Kontext
  - \* list Kontext
- Verkettbar :  $== ; 5 < \$a < 12$
- Junction :  $\$a == 3 | 7 | 9;$
- Metaoperatoren
  - Reduktionsoperatoren, Hyperoperatoren

# Neue Regeln : Ctrl

- runde Klammern optional
- jeder Block hat 15 traits
- given, when, default, continue
- elsif nach unless
- do ist „do once loop“
- repeat statt do : repeat { ... } while ...
- loop für endlos und C-stil Schleifen
- Signaturen, multi sub, wrapping
- lexikalische subs