

# Die wunderbare Welt der Regulären Ausdrücke

Jakob Westhoff

LinuxWochen Wien 2011

Über uns

Jakob Westhoff

@jakobwesthoff

jakob@php.net

# Agenda

- Terminologie
- Einstieg in RegExp
- Quantifier
- Der Punkt
- Zeichenklassen
- Escaping
- Anker
- Subpattern

# Agenda II

- Unicode
- Performance
- Spaß mit Backtracking

Fragen?

Fragen!

# Terminologie

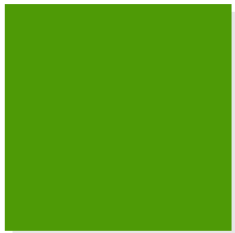
# Terminologie



RegExp



Subject



Match

# Terminologie



RegExp

## Regulärer Ausdruck

- Beschreibung einer Menge von Zeichen und Zeichenketten

# Terminologie



Subject

## Satzgegenstand

- Zeichenkette auf die eine RegExp angewendet wird

# Terminologie



Match

## Treffer

- Der Teil der Zeichenkette, der Bestandteil der RegExp ist

RegExp

# Aufbau einer RegExp

`/foobar/i`

# Aufbau einer RegExp

*/foobar/i*



## ○ Delimiter

- Trennung zwischen Pattern und Modifiern

# Aufbau einer RegExp

( foobar ) i



The diagram shows the regular expression `( foobar ) i` in red text. Two black arrows point upwards from below the text to the opening parenthesis `(` and the closing parenthesis `)`, highlighting their role as delimiters.

## ● Delimiter

- In PCRE sind Klammern erlaubt
  - ( ) [ ] { }

# Aufbau einer RegExp

( foobar ) i



## Pattern

- Beschreibung der gesuchten Zeichenkette

# Aufbau einer RegExp

( foobar ) i



## Modifier

- Zusätzliche Optionen

# Einfach eine Zeichenkette

- Das Pattern einer RegExp ist ein String

Open Source

# Einfach eine Zeichenkette

- Das Pattern einer RegExp ist ein String

(Open Source)

# Einfach eine Zeichenkette

- Das Pattern einer RegExp ist ein String

(Open Source)

Free and Open Source Conference

# Einfach nur ein String

- Das Pattern einer RegExp ist ein String

(Open Source)

Free and Open Source Conference

- Das Pattern muss mind. einmal auftauchen
- Die Position innerhalb des Subject ist egal

# Metacharacters

- Einige Zeichen innerhalb einer RegExp besitzen eine besondere Bedeutung

( [0p]en \s\* So+u.ce )

# Quantifier

# Quantifier

- Quantifier definieren Wiederholungen des vorherigen Zeichens oder Gruppe

(Op\*en So+ur?c{1,3}e)

# Quantifier

- Quantifier definieren Wiederholungen des vorherigen Zeichens oder Gruppe

(0p\*en So+ur?c{1,3}e)



- \* Beliebig oft ( $0 \rightarrow \infty$ )

# Quantifier

- Quantifier definieren Wiederholungen des vorherigen Zeichens oder Gruppe

$(Op^*en\ So+ur?c\{1,3\}e)$



- $*$  Beliebig oft ( $0 \rightarrow \infty$ )
- $+$  Beliebig oft aber mind. einmal ( $1 \rightarrow \infty$ )

# Quantifier

- Quantifier definieren Wiederholungen des vorherigen Zeichens oder Gruppe

$(Op^*en\ So+ur?c\{1,3\}e)$



- $*$  Beliebig oft ( $0 \rightarrow \infty$ )
- $+$  Beliebig oft aber mind. einmal ( $1 \rightarrow \infty$ )
- $?$  Null oder einmal ( $0 \rightarrow 1$ )

# Quantifier

- Quantifier definieren Wiederholungen des vorherigen Zeichens oder Gruppe

(Op\*en So+ur?c{1,3}e)



- \* Beliebig oft ( $0 \rightarrow \infty$ )
- + Beliebig oft aber mind. einmal ( $1 \rightarrow \infty$ )
- ? Null oder einmal ( $0 \rightarrow 1$ )
- {x,y} Zwischen x und y mal ( $x \rightarrow y$ )

Der Punkt

# Der Punkt

- Der Punkt ( . ) matcht jedes beliebige Zeichen
  - außer einem Zeilenvorschub

(Ohne Punkt und .omma)



# Der Punkt

- Der Punkt ( . ) matcht jedes beliebige Zeichen
  - außer einem Zeilenvorschub

(Ohne Punkt und .omma)

Ohne Punkt und Komma ✓

# Der Punkt

- Der Punkt ( . ) matcht jedes beliebige Zeichen
  - außer einem Zeilenvorschub

(Ohne Punkt und .omma)

Ohne Punkt und Komma ✓

Ohne Punkt und Somma ✓

# Der Punkt

- Der Punkt ( . ) matcht jedes beliebige Zeichen
  - außer einem Zeilenvorschub

(Ohne Punkt und .omma)

Ohne Punkt und Komma ✓

Ohne Punkt und Somma ✓

Ohne Punkt und \_omma ✓

# Der Punkt

## ○ Umschalten der Engine auf single line

- Modifier **s**

(Ein .unkt)s



- Der Punkt matcht ebenfalls Zeilenvorschub

Zeichenklassen

# Zeichenklassen

- Zeichenklassen definieren eine Menge beliebiger Zeichen

a b c d e f

# Zeichenklassen

a b c d e f

# Zeichenklassen

abcdef

- Keinerlei Trennzeichen

# Zeichenklassen

[abcdef]

- Keinerlei Trennzeichen
- Umschlossen von eckigen Klammern ([ ])

# Zeichenklassen

( [abcdef] + )

- Keinerlei Trennzeichen
- Umschlossen von eckigen Klammern ( [ ] )
- Werden behandelt wie ein Zeichen

# Zeichenklassen

$([a-f]^+)$

- Können als Bereiche definiert werden

# Zeichenklassen

$([a - cd - f]^+)$

- Können als Bereiche definiert werden
- Dürfen aus mehreren Bereichen bestehen

# Zeichenklassen

( [abc.] + )

- Sonderzeichen verlieren ihre Bedeutung

# Zeichenklassen

( [abc . - ]+ )

- Sonderzeichen verlieren ihre Bedeutung
- Neue Sonderzeichen existieren

# Zeichenklassen

(<sup>^</sup>abcdef<sup>+</sup>)

- Negationen sind möglich

# Zeichenklassen

(<sup>^</sup>abcdef<sup>+</sup>)

- Negationen sind möglich
- Zeilenvorschub ebenfalls in der Negation

# Zeichenklassen

`( [^\n]+ )`

- Negationen sind möglich
- Zeilenvorschub ebenfalls in der Negation
- Zeilenvorschub kann ausgeschlossen werden

# Zeichenklassen

- Es gibt vordefinierte Zeichenklassen
  - `\d` Jede Ziffer (0,1,2,...)
  - `\s` Jeder Whitespace (<Space>, <Tab>, ...)
  - ...
- Großbuchstaben negieren die Klasse
  - `\D` Alles außer Ziffern
  - ...

# Escaping

# Escaping

- Besondere Bedeutung von Zeichen kann abgeschaltet werden (Escaping)

`jakob.westhoff@gmail.com`

# Escaping

- Besondere Bedeutung von Zeichen kann abgeschaltet werden (Escaping)

`(jakob.westhoff@gmail.com)i`

# Escaping

- Besondere Bedeutung von Zeichen kann abgeschaltet werden (Escaping)

(jacob.westhoff@gmail.com)i



- Dies ist ein echter Punkt und nicht ein beliebiges Zeichen

# Escaping

- Besondere Bedeutung von Zeichen kann abgeschaltet werden (Escaping)

(jakob\.westhoff@gmail\.com)i

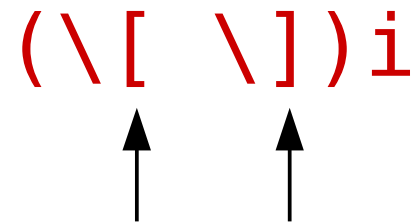


- Einsatz des Backslashes (\)

# Escaping

- Funktioniert für alle Zeichen mit besonderer Bedeutung

(\ [ \ ] ) i



# Escaping

- Funktioniert für alle Zeichen mit besonderer Bedeutung

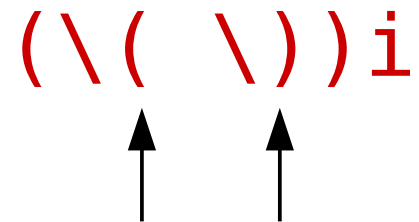
`(\*)i`



# Escaping

- Funktioniert für alle Zeichen mit besonderer Bedeutung

(\ ( \) ) i



# Escaping

- Funktioniert für alle Zeichen mit besonderer Bedeutung

`(\+)i`



# Escaping

- Funktioniert für alle Zeichen mit besonderer Bedeutung



# Escaping in der Realität

- RegExp sind meist ein String

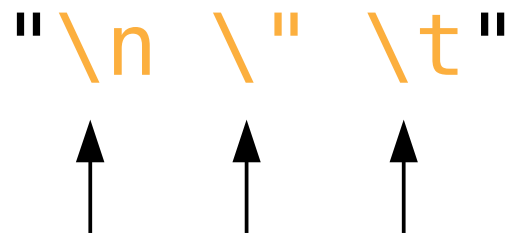
"(jakob\\.westhoff@gmail\\.com)i"



# Escaping in der Realität

- Strings besitzen oft eigenes Escaping

"\n \" \t"



The diagram illustrates string escaping. It shows a string enclosed in double quotes: `"\n \" \t"`. The backslashes (`\`) are highlighted in orange. Three black arrows point upwards from below the string to each of the backslashes, indicating that these characters are being escaped.

# Escaping in der Realität

- Backslashes (\) in einem RegExp-String müssen escaped werden

"(jakob\\.westhoff@gmail\\.com)i"



# Escaping in der Realität

Was matcht dieser RegExp String?

" ( [ \\ \\ [ \\ ] ] \\ \\ ++ ) "

# Escaping in der Realität

"([\[\[\[\[\[\[\]]\]\]\]\]\]\]++)g"

\[\[\+]\[\+\+\]\+\+\+\

# Escaping in der Realität

↓ ↓

" ( [ \ \ \ [ \ \ ] \ \ + + ) g "

\ [ \ + ] [ + + \ ] + + + \



- Backslash gefolgt von mindestens einem Pluszeichen

# Escaping in der Realität

↓ ↓

"([\[\]\+\+\])g"

\[\+\]\+\+\



- Eckige Klammer gefolgt von mindestens einem (2) Pluszeichen

# Escaping in der Realität

↓ ↓  
" ( [ \\ \\ \\ [ \\ \\ ] \\ \\ ++ ) g "

\\ [ \\ + ] [ ++ \\ ] + + + \\



- Geschlossene Eckige Klammer gefolgt von mindestens einem (3) Pluszeichen

Anker

# Anker

- Anker gehören zur Familie der Assertions
  - to assert sth. = etw. durchsetzen
- Durchsetzung von Bedingungen unabhängig vom Match
- Hier: Anfang und Ende eines Subjects

# Anker

- ⦿ ^ Anfang des Subject
- ⦿ \$ Ende des Subject

# Anker

- ⦿ ^ Anfang des Subject
- ⦿ \$ Ende des Subject

(Ana) i

# Anker

- ⦿ ^ Anfang des Subject
- ⦿ \$ Ende des Subject

(An) i

Ananas

# Anker

- ^ Anfang des Subject
- \$ Ende des Subject

(Ana) i

Ananas ✓

# Anker

- ⊙ ^ Anfang des Subject
- ⊙ \$ Ende des Subject

(Ana) i

Ananas ✓

Banane ✓

# Anker

- ^ Anfang des Subject
- \$ Ende des Subject



(^Ana)i

Ananas ✓

Banane ✗

# Anker (Modifier)

- Multiline Mode

- Modifier **m**

(**^abcdef\$**)**m**



- Anker treffen Anfang und Ende jeder Zeile innerhalb des Subject

# Anker (Modifier)

(^abcdef\$)

abcdef  
ghijkl  
mnopqr

# Anker (Modifier)

(<sup>^</sup>abcdef<sup>\$</sup>)

abcdef  
ghijkl  
mnopqr

- Kein Match, da Anker Anfang und Ende des Subjekt Strings sind

# Anker (Modifier)

$(^abcdef\$)m$

abcdef  
ghijkl  
mnopqr

- Anker sind nun Anfang und Ende jeder Zeile

# Anker (Modifier)

## End only Mode

- Modifier **D**

(**^abcdef\$**)**D**



## \$ trifft nur noch das echte Ende des Subject

- Normalerweise: abschließende Newline erlaubt

# Anker

- Newline Mode (**m**) unabhängige Anker
- **\A** Anfang des Subject
- **\z** Ende des Subject

Subpattern

# Subpattern

- Runde Klammern unterteilen Pattern

`((abc)(def))`

`abcdef`

# Subpattern

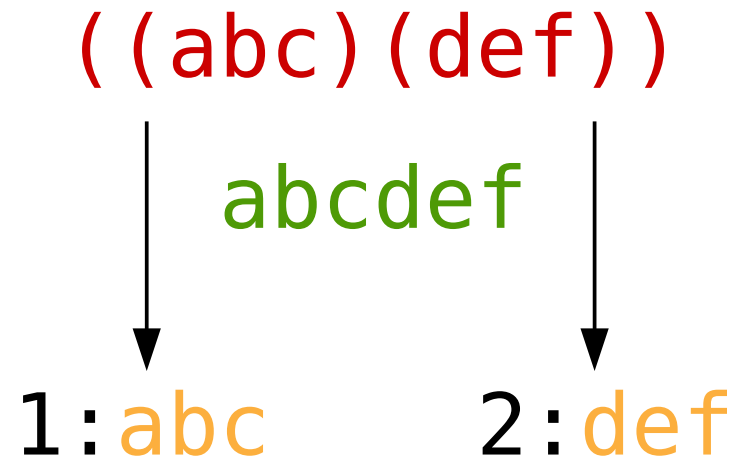
- Runde Klammern unterteilen Pattern

`((abc)(def))`

`abcdef`

# Subpattern

- Runde Klammern unterteilen Pattern



- Extraktion von Teilen eines Matches

# Subpattern

`((abc)\1)`

`abcabc`

- Wiederverwendung innerhalb eines Pattern

# Optionen in Subpattern

- Optionen für einzelne Bereiche setzen

`((?#Ich bin ein Kommentar))`

# Optionen in Subpattern

● Optionen für einzelne Bereiche setzen

`(?OptionPattern)`

- Allgemeine Syntax für alle Optionen

# Benannte Subpattern

- Subpattern können benannt werden

`((?P<Vorname>Jakob))`



- Einsatz der Option **P**

# Benannte Subpattern

`((?P<Vorname>Jakob))`

Jakob Westhoff

# Benannte Subpattern

`((?P<Vorname>Jakob))`



Jakob Westhoff

Vorname: Jakob

- Extraktion mit Benennung

Lesbarkeit

# Lesbare RegExp

- Kommentare, Einrückungen und Zeilenumbrüche in RegExp
  - Modifizier **x**

( foobar ) x



# Lesbare RegExp

`(^[a-z0-9_%. - ]+@[a-z0-9. - ]+\.[a-z]{2,4}$)iD`

Lesbar? Verständlich? Wartbar?

# Lesbare RegExp

```
(  
  ^           #Start des Subject  
  [a-z0-9_%. - ]+ #Benutzer  
  @           #Trennzeichen @  
  [a-z0-9. - ]+ #Domain  
  \.         #Trennzeichen .  
  [a-z]{2,4} #TLD  
  $           #Ende des Subject  
)iDx
```

Lesbar? Verständlich? Wartbar?

# Lesbare RegExp

```
(  
  ^           #Start des Subject  
  [a-z0-9_%. - ]+ #Benutzer  
  @          #Trennzeichen @  
  [a-z0-9. - ]+ #Domain  
  \.        #Trennzeichen .  
  [a-z]{2,4} #TLD  
  $         #Ende des Subject  
)iDx
```

- Zeilenumbrüche an beliebigen Stellen

# Lesbare RegExp

```
(  
  ^           #Start des Subject  
  [a-z0-9_%. - ]+ #Benutzer  
  @           #Trennzeichen @  
  [a-z0-9. - ]+ #Domain  
  \.         #Trennzeichen .  
  [a-z]{2,4} #TLD  
  $         #Ende des Subject  
)iDx
```

- Alles nach einer Raute (#) bis zum Zeilenende wird ignoriert

# Lesbare RegExp

```
(  
  ^           #Start des Subject  
  [a-z0-9_%. - ]+ #Benutzer  
  @          #Trennzeichen @  
  [a-z0-9. - ]+ #Domain  
  \.        #Trennzeichen .  
  [a-z]{2,4} #TLD  
  $         #Ende des Subject  
)iDx
```

- Alle Leerzeichen werden ignoriert es sei denn sie sind escaped ( \ )

Unicode

# Unicode

## UTF-8 Mode

- Modifier **u**

(^abcdef\$)u



## Valides UTF-8 in Pattern und Subject erforderlich

# Unicode

## ● UTF-8 Encoding

- Byte für alle ASCII Zeichen (0-127) identisch
- 2-4 Byte für weitere Zeichen (Codepoints)

## ● Codepoints

- Behandlung als ein Zeichen

# Unicode

Русский

# Unicode

`(\x{0420})u`

Русский

# Unicode

`(\x{0420})u`

Русский



- `\x` Escape zur Angabe einzelner Codepoints

# Unicode



`( [\x{0400} - \x{A697} ]+ )u`

Русский

- `\x` Escape zur Angabe einzelner Codepoints
- Funktioniert in Zeichenklassen

# Unicode



`(\p{Cyrillic}+)u`

Русский

- `\x` Escape zur Angabe einzelner Codepoints
- Funktioniert in Zeichenklassen
- Es gibt vordefinierte Zeichenklassen

# Unicode

`(\p{Cyrillic}+)u`

Русский 한국어



- `\x` Escape zur Angabe einzelner Codepoints
- Funktioniert in Zeichenklassen
- Es gibt vordefinierte Zeichenklassen

# Unicode



`(\p{L}+)`

Русский 한국어

- `\x` Escape zur Angabe einzelner Codepoints
- Funktioniert in Zeichenklassen
- Es gibt vordefinierte Zeichenklassen

Performance

# Performance

- Die PCRE Engine verwendet Backtracking

# Performance

- Die PCRE Engine verwendet Backtracking

( [a-z0-9]+\d )

abc42def

# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

`abc42def`



# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

`abc42def`



# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

`abc42def`



# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

`abc42def`



# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

abc42def



# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

abc42def



# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

`abc42def`



# Performance

- Die PCRE Engine verwendet Backtracking



`([a-z0-9]+\d)`

`abc42def`



# Performance

- Die PCRE Engine verwendet Backtracking

↓  
( [a-z0-9]+\d )

abc42def



# Performance

- Die PCRE Engine verwendet Backtracking

↓  
( [a-z0-9]+\d )

abc42def



# Performance

- Die PCRE Engine verwendet Backtracking

↓  
( [a-z0-9]+\d )

abc42def



# Performance

- Die PCRE Engine verwendet Backtracking

↓  
( [a-z0-9]+\d )

abc42def



# Performance

- Die PCRE Engine verwendet Backtracking

↓  
( [a-z0-9]+\d )

abc42def



# Performance

- Die PCRE Engine verwendet Backtracking

`([a-z0-9]+\d)`

`abc42def`

# Performance

`([a-z0-9]+\d)`

- Durch nicht disjunkte Zeichenklassen hier relativ langsam

# Performance

`( [a-z0-9]+\d )`

- Durch nicht disjunkte Zeichenklassen hier relativ langsam

Geht das besser?

# Greediness

- Üblicherweise ist die PCRE Engine greedy
  - greedy = gefräßig / gierig
- Es werden immer so viele Zeichen wie möglich verschlungen

# Greediness

- Umschalten der Engine auf ungreedy
  - Modifier **U**

`([a-z0-9]+\d)U`



# Greediness

## ○ Umschalten der Engine auf ungreedy

- Modifier **U**

`( [a-z0-9]+\d )U`

- **?** hinter einem Quantifier

`( [a-z0-9]+\d? )`



# Greediness

- Auswirkungen auf das vorherige Beispiel



`([a-z0-9]+\d)U`

`abc42def`

# Greediness

- Auswirkungen auf das vorherige Beispiel



`([a-z0-9]+\d)U`

`abc42def`



# Greediness

- Auswirkungen auf das vorherige Beispiel



`([a-z0-9]+\d)U`

`abc42def`



# Greediness

- Auswirkungen auf das vorherige Beispiel



`([a-z0-9]+\d)U`

`abc42def`



# Greediness

- Auswirkungen auf das vorherige Beispiel



`([a-z0-9]+\d)U`

`abc42def`



# Greediness

- Auswirkungen auf das vorherige Beispiel



`([a-z0-9]+\d)U`

abc42def



# Greediness

- Auswirkungen auf das vorherige Beispiel



`([a-z0-9]+\d)U`

`abc42def`



# Greediness

- Auswirkungen auf das vorherige Beispiel

`([a-z0-9]+\d)U`

`abc42def`

# Greediness

- Achtung ungreedy ist nur in Einzelfällen schneller
- Meistens ist es sogar langsamer
- Kann andere Matches produzieren als greedy

# Atomic Groups

- Eine weitere Optimierungsmöglichkeit sind Atomic Groups
- Explizite Abschaltung von Backtracking für einen Bereich der RegExp

# Atomic Groups

`([a-z]+42)`

`abcd21`

# Atomic Groups



`([a-z]+42)`

`abcd21`



# Atomic Groups



`([a-z]+42)`

`abcd21`



# Atomic Groups



`([a-z]+42)`

`abcd21`



# Atomic Groups



`( [a-z]+42 )`

`abcd21`



# Atomic Groups



`([a-z]+42)`

`abcd21`



# Atomic Groups



`( [a-z]+42 )`

`abcd21`



# Atomic Groups



`([a-z]+42)`

`abcd21`



# Atomic Groups



`([a-z]+42)`

`abcd21`



# Atomic Groups



`([a-z]+42)`

`abcd21`



# Atomic Groups

`([a-z]+42)`

`abcd21` **x**

# Atomic Groups



`((?>[a-z]+)42)`

abcd21

- Atomic Groups werden durch eine Subpattern Option aktiviert

# Atomic Groups



`((?>[a-z]+)42)`

`abcd21`



# Atomic Groups



`((?>[a-z]+)42)`

abcd21



# Atomic Groups



`((?>[a-z]+)42)`

`abcd21`



# Atomic Groups



`((?>[a-z]+)42)`

abcd21



# Atomic Groups

↓  
`((?>[a-z]+)42)`

abcd21



# Atomic Groups

`((?>[a-z]+)42)`

`abcd21` **x**

- Kein Backtracking für dieses Subpattern erlaubt, daher sofortiger Abbruch

# Atomic Groups

- Backtracking kann auch für einzelne Quantifier verboten werden
  - + Possesive Quantifier

( [a-z]++42 )



# Performance

- PCRE besitzt ein Limit für die Anzahl möglicher Backtracking Schritte
- Kann beim Kompilieren der Library gesetzt werden (Standardwert: 10.000.000)
- Kann in vielen Umgebungen konfiguriert werden

# Spaß mit Backtracking

- Finden von Primzahlen mit RegExp
  - Primzahlen sind nur durch 1 und sich selbst teilbar
  - 0 und 1 sind keine Primzahlen

# Spaß mit Backtracking

## ● Idee:

- Zahl als Wiederholung von 1 darstellen

4 → 1111

- Match = Keine Primzahl
- Kein Match = Primzahl

# Spaß mit Backtracking

```
(^1?$|^(11+?)\1+$)
```

# Spaß mit Backtracking

`(^1?$ | ^(11+?)\1+$)`



Oder: Trennt RegExp in zwei Teile

# Spaß mit Backtracking


`(^1?$ | ^(11+?)\1+$)`



Leere Zeichenkette (0) oder eine 1 (1)

# Spaß mit Backtracking

`(^1?$|^ (11+?) \1+$)`



The diagram shows the regular expression `(^1?$|^ (11+?) \1+$)` in red text. Two black arrows point upwards from below the text to the caret character (^) at the beginning of the first alternative and the dollar sign (\$) at the end of the second alternative.

Anker: Gesamten String untersuchen

# Spaß mit Backtracking

`(^1?$|^((11+?)\1+$))`



Zwei mal oder öfter 1 (ungreedy)

# Spaß mit Backtracking

`(^1?$|^(11+?)\1+$)`



Ein mehrfaches der zuvor gematchten  
Anzahl 1

# Spaß mit Backtracking

`(^1?$|^(11+?)\1+$)`

WTF? Häh?

# Spaß mit Backtracking

`(^1?$|^(11+?)\1+$)`

- Primzahlen sind nur durch 1 und sich selbst teilbar
- 0 und 1 sind keine Primzahlen

# Spaß mit Backtracking

`(^1?$ | ^(11+?)\1+$)`



- Primzahlen sind nur durch 1 und sich selbst teilbar
- 0 und 1 sind keine Primzahlen

# Spaß mit Backtracking

`^(11+?)\1+$`

- Primzahlen sind nur durch 1 und sich selbst teilbar

# Spaß mit Backtracking

$(11+?) \setminus 1+$

- Primzahlen sind nur durch 1 und sich selbst teilbar

# Spaß mit Backtracking

$(11+?) \setminus 1+$

- Primzahlen sind nur durch 1 und sich selbst teilbar

11111

# Spaß mit Backtracking

$(11+?) \setminus 1+$

- Primzahlen sind nur durch 1 und sich selbst teilbar

$11111 = 5$  (Primzahl)

# Spaß mit Backtracking

$(11+?) \setminus 1+$



- Primzahlen sind nur durch 1 und sich selbst teilbar

11111



# Spaß mit Backtracking

$(11+?) \setminus 1+$



- Primzahlen sind nur durch 1 und sich selbst teilbar

11111



# Spaß mit Backtracking

(11+?)\1+



- Primzahlen sind nur durch 1 und sich selbst teilbar

11111



# Spaß mit Backtracking

$(11+?) \setminus 1+$



- Primzahlen sind nur durch 1 und sich selbst teilbar

11111



# Spaß mit Backtracking

(11+?)\1+



- Primzahlen sind nur durch 1 und sich selbst teilbar

11111



# Spaß mit Backtracking

$(11+?) \setminus 1+$



- Primzahlen sind nur durch 1 und sich selbst teilbar

11111



# Spaß mit Backtracking

$(11+?) \setminus 1+$

- Primzahlen sind nur durch 1 und sich selbst teilbar

11111 ✘

- Kein Match = Primzahl

# Spaß mit Backtracking

$(11+?)\backslash 1+$

- Primzahlen sind nur durch 1 und sich selbst teilbar

1111\_ ✓

- Match = Keine Primzahl

# Spaß mit Backtracking

- Suche nach Teilern mittels Backtracking
- Just For Fun
  - Bitte nicht produktiv verwenden!
- Backtracking Limit von 1.000.000 → 22201 (erste falsche Primzahl)

Danke für Eure Aufmerksamkeit!

Fragen?

Jakob Westhoff

Mail: [jakob@php.net](mailto:jakob@php.net)

Twitter: [@jakobwesthoff](https://twitter.com/@jakobwesthoff)