
Deeper down the rabbit hole

Advanced Regular Expressions

Jakob Westhoff <jakob@php.net>
[@jakobwesthoff](#)

PHPBarcamp.at
May 3, 2010

- Jakob Westhoff
 - PHP developer for several years
 - Computer science student at the TU Dortmund
 - Co-Founder of the PHP Usergroup Dortmund
 - Active in different Open Source projects

Asking the audience

- Who does already work with regular expressions?

Asking the audience

- Who does already work with regular expressions?
- Regular expressions like this:

```
/[a-zA-Z]+/
```

Asking the audience

- Who does already work with regular expressions?
- Regular expressions like this:

```
/[a-zA-Z]+/
```

- Or like this:

```
(?P<image>(?:none|inherit)|(?:url\\(\\s*(?:'|")  
    ?(?:\\\\"'"\\)|\\\\"^\\'"\\)|\\["'"\\])\\s*(?:'|")?  
    \\s*\\))
```

Goals of this session

- Learn advanced techniques to use in (PCRE) regular expressions

Goals of this session

- Learn advanced techniques to use in (PCRE) regular expressions
 - Assertions
 - Once only subpatterns
 - Conditional subpatterns
 - Pattern recursion
 - ...

Goals of this session

- Learn advanced techniques to use in (PCRE) regular expressions
 - Assertions
 - Once only subpatterns
 - Conditional subpatterns
 - Pattern recursion
 - ...
- Learn howto to handle Unicode in your regular expressions

What Regular Expressions are. . .

- In **theoretical** computer science:
 - Express **regular languages**

What Regular Expressions are. . .

- In **theoretical** computer science:
 - Express **regular languages**
 - Languages which can be described by **deterministic finite state automata**

What Regular Expressions are. . .

- In **theoretical** computer science:
 - Express **regular languages**
 - Languages which can be described by **deterministic finite state automata**
 - **Type 3 grammars** in the **Chomsky hierarchy**

What Regular Expressions are. . .

- In **practical** day to day usage:

“[. . .]regular expressions provide concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters.”

– Wikipedia [1]

What Regular Expressions are. . .

- In **practical** day to day usage:

“[. . .]regular expressions provide concise and flexible means for **identifying strings of text of interest**, such as particular characters, words, or patterns of characters.”

– Wikipedia [1]

What Regular Expressions are. . .

- In **practical** day to day usage:

“[. . .]regular expressions provide concise and flexible means for identifying strings of text of interest, such as **particular characters**, words, or patterns of characters.”

– Wikipedia [1]

What Regular Expressions are. . .

- In **practical** day to day usage:

“[. . .]regular expressions provide concise and flexible means for identifying strings of text of interest, such as particular characters, **words**, or patterns of characters.”

– Wikipedia [1]

What Regular Expressions are. . .

- In **practical** day to day usage:

“[. . .]regular expressions provide concise and flexible means for identifying strings of text of interest, such as particular characters, words, or **patterns of characters**.”

– Wikipedia [1]

Building Blocks of a Regular Expression

- Basic structure of every regular expression

`/[a-z]+/im`

Building Blocks of a Regular Expression

- Basic structure of every regular expression

`/[a-z]+/im`

- Delimiter

Building Blocks of a Regular Expression

- Basic structure of every regular expression

`/[a-z]+/im`

- Delimiter
 - **Equal** characters of arbitrary **choice** (must be escaped in expression)

Building Blocks of a Regular Expression

- Basic structure of every regular expression

`/[a-z]+/im`

- Delimiter
 - **Equal** characters of arbitrary **choice** (must be escaped in expression)
 - May be (and) in **PCRE**

Building Blocks of a Regular Expression

- Basic structure of every regular expression

`/[a-z]+/im`

- Delimiter
 - Equal characters of arbitrary choice (must be escaped in expression)
 - May be (and) in PCRE
- Expression

Building Blocks of a Regular Expression

- Basic structure of every regular expression

`/[a-z]+/im`

- Delimiter
 - Equal characters of arbitrary choice (must be escaped in expression)
 - May be (and) in PCRE
- Expression
- Modifier

Building Blocks of a Regular Expression

- Basic structure of every regular expression

`/[a-z]+/im`

- Delimiter
 - Equal characters of arbitrary choice (must be escaped in expression)
 - May be (and) in PCRE
- Expression
- Modifier
 - A sequence of characters providing processing instructions

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions

Getting everybody up to speed

- ., .*, .+, .?, .{1,2} - Arbitrary characters and repetitions

Getting everybody up to speed

- ., .*, .+, .?, .{1,2} - Arbitrary characters and repetitions

Getting everybody up to speed

- ., .*, .+, .?, .{1,2} - Arbitrary characters and repetitions

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions
- `^`, `$` - **Start** and end of subject (or line in multiline mode)

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions
- `^`, `$` - Start and **end** of subject (or line in multiline mode)

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions
- `^`, `$` - Start and end of subject (or line in multiline mode)
- `foo|bar` - Logical Or

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions
- `^`, `$` - Start and end of subject (or line in multiline mode)
- `foo|bar` - Logical Or
- `(foo)(bar)` - Subpattern grouping

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions
- `^`, `$` - Start and end of subject (or line in multiline mode)
- `foo|bar` - Logical Or
- `(foo)(bar)` - Subpattern grouping
- `/(foo|bar)baz(\1)/` - Backreferences

Getting everybody up to speed

- `.`, `*`, `+`, `?`, `{1,2}` - Arbitrary characters and repetitions
- `^`, `$` - Start and end of subject (or line in multiline mode)
- `foo|bar` - Logical Or
- `(foo)(bar)` - Subpattern grouping
- `/(foo|bar)baz(\1)/` - Backreferences
- `[a-z]`, `^[a-z]` - Character classes

Grouping Without Subpattern Creation

- Grouping might be needed **without** creating a subpattern

Grouping Without Subpattern Creation

- Grouping might be needed **without** creating a subpattern

```
/(?:foobar)*/
```

Subpattern identification

- Subpatterns are numbered by opening paranthesis

Subpattern identification

- Subpatterns are numbered by opening paranthesis
- `/(foo(bar)(baz))/`

Subpattern identification

- Subpatterns are numbered by opening paranthesis
- `/(foo(bar)(baz))/`
 - 1 foobarbaz

Subpattern identification

- Subpatterns are numbered by opening parenthesis

- `/(foo(bar)(baz))/`

- 1 foobarbaz

- 2 bar

Subpattern identification

- Subpatterns are numbered by opening parenthesis

- `/(foo(bar)(baz))/`

- 1 foobarbaz

- 2 bar

- 3 baz

Subpattern identification

- Subpatterns are numbered by opening parenthesis

- `/(foo(bar)(baz))/`

- 1 foobarbaz

- 2 bar

- 3 baz

- Matches available from within PHP

```
$matches = array(  
    0 => "foobarbaz",  
    1 => "foobarbaz",  
    2 => "bar",  
    3 => "baz",  
)
```

Subpattern Naming

- PCRE allows custom naming

```
/(?P<firstname>[A-Za-z]+) (?P<lastname>[A-Za-z]+)/
```

Subpattern Naming

- PCRE allows custom naming

```
/(?P<firstname>[A-Za-z]+) (?P<lastname>[A-Za-z]+)/
```

- Result with input Jakob Westhoff

```
array (  
  0 => 'Jakob_Westhoff',  
  'firstname' => 'Jakob',  
  1 => 'Jakob',  
  'lastname' => 'Westhoff',  
  2 => 'Westhoff',  
)
```

Assertions

- Formulate **assertions** on the matched string without consuming them

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

- Input

```
foofoofoo
```

- Match

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

- Input

```
foofoofoo
```

- Match

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

- Input

```
foofoofoo
```

- Match

```
foo
```

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

- Input

```
foofoofoo
```

- Match

```
foo
```

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

- Input

```
foofoofoo
```

- Match

```
foofoo
```

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

- Input

```
foofoofoo
```

- Match

```
foofoo
```

Assertions

- Formulate **assertions** on the matched string without consuming them
- Example

```
/foo(?=foo)/
```

- Input

```
foofoofoo
```

- Match

```
foofoo
```

Negative Assertions

- **Negative** assertions are possible

Negative Assertions

- **Negative** assertions are possible
- foo **not** followed by another foo

```
/foo(?!foo)/
```

Backward Assertions

- bar **preceeded** by foo

Backward Assertions

- bar **preceeded** by foo

`/(?=foo)bar/ ?`

Backward Assertions

- bar **preceeded** by foo

```
//(?!#foo)bar//
```

- Backward assertion

```
/(?<=foo)bar/
```

Backward Assertions

- bar preceded by foo

```
//(?!foo)bar//
```

- Backward assertion

```
/(?<=foo)bar/
```

- Negative backward assertion
- bar not preceded by foo

```
/(?<!foo)bar/
```

Inner workings of the PCRE matcher

- PCRE uses backtracking to find matches

Inner workings of the PCRE matcher

- PCRE uses backtracking to find matches
- Pattern: `/\d+foo/`
- Subject: `123456789bar`

Inner workings of the PCRE matcher

- PCRE uses backtracking to find matches
- Pattern: `/\d+foo/`
- Subject: `123456789bar`
- 1 Eat up all the numbers: `123456789`

Inner workings of the PCRE matcher

- PCRE uses backtracking to find matches
 - Pattern: `/\d+foo/`
 - Subject: `123456789bar`
- 1 Eat up all the numbers: `123456789`
 - 2 Try to match `foo`

Inner workings of the PCRE matcher

- PCRE uses backtracking to find matches

- Pattern: `/\d+foo/`

- Subject: `123456789bar`

- 1 Eat up all the numbers: `123456789`

- 2 Try to match `foo`

- 3 Backtrack one number and try to match `foo` again

Inner workings of the PCRE matcher

- PCRE uses backtracking to find matches
 - Pattern: `/\d+foo/`
 - Subject: `123456789bar`
- 1 Eat up all the numbers: `123456789`
 - 2 Try to match `foo`
 - 3 Backtrack one number and try to match `foo` again
 - 4 Repeat step 3 until a match is found or the subjects beginning is reached

Once only subpattern

- Once only subpatterns prevent backtracking once a certain pattern has acquired a match.

Once only subpattern

- Once only subpatterns prevent backtracking once a certain pattern has acquired a match.
- Applying a once only pattern to the shown example

```
/(?>\d+)foo/
```

Once only subpattern

- Once only subpatterns prevent backtracking once a certain pattern has acquired a match.
- Applying a once only pattern to the shown example

`/(>\d+)foo/`

- After matching the numbers and determining the following string is not `foo` the matcher stops
 - `123456789bar`

Once only subpattern

- Once only subpatterns prevent backtracking once a certain pattern has acquired a match.
- Applying a once only pattern to the shown example

```
/(?>\d+)foo/
```

- After matching the numbers and determining the following string is not `foo` the matcher stops
 - `123456789bar`
- Can massively improve regex speed if used correctly

Conditional subpattern

- If statement equivalent in PCRE

Conditional subpattern

- If statement equivalent in PCRE

```
/(?(condition)yes-pattern|no-pattern)/
```

Conditional subpattern

- If statement equivalent in PCRE

```
/(?(condition)yes-pattern|no-pattern)/
```

Conditional subpattern

- If statement equivalent in PCRE

```
/(?(condition)yes-pattern|no-pattern)/
```

Conditional subpattern

- If statement equivalent in PCRE

```
/(?(condition)yes-pattern|no-pattern)/
```

Conditional subpattern

- If statement equivalent in PCRE

```
/(?(condition)yes-pattern|no-pattern)/
```

- Conditions can be direct matches or assertions

Conditional subpattern

- If statement equivalent in PCRE

```
/(?(condition)yes-pattern|no-pattern)/
```

- Conditions can be direct matches or assertions
- Numbers need to be followed by `foo`, while everything else needs to be followed by `bar`

```
/(?(\d+)foo|bar)/
```

Unicode: Character, code points and graphemes

- Unicode consists of different code points

Unicode: Character, code points and graphemes

- Unicode consists of different code points
 - The letter a: U+0061

Unicode: Character, code points and graphemes

- Unicode consists of different code points
 - The letter a: U+0061
 - The mark ‘: U+0300

Unicode: Character, code points and graphemes

- Unicode consists of different code points
 - The letter a: U+0061
 - The mark ‘: U+0300
- One character might consist of **multiple** code points

Unicode: Character, code points and graphemes

- Unicode consists of different code points
 - The letter a: U+0061
 - The mark ‘: U+0300
- One character might consist of **multiple** code points
 - The letter a with the mark ‘ (â) : U+0061 U+0300

Unicode: Character, code points and graphemes

- Unicode consists of different code points
 - The letter a: U+0061
 - The mark ‘: U+0300
- One character might consist of **multiple** code points
 - The letter a with the mark ‘ (ã) : U+0061 U+0300
- Some of these combinations exists as single code points

Unicode: Character, code points and graphemes

- Unicode consists of different code points
 - The letter a: U+0061
 - The mark ‘: U+0300
- One character might consist of **multiple** code points
 - The letter a with the mark ‘ (ã): U+0061 U+0300
- Some of these combinations exists as single code points
 - The letter ã: U+00E0

Unicode: Pattern matching

- Unicode processing is enabled using the `u` modifier

Unicode: Pattern matching

- Unicode processing is enabled using the `u` modifier
- **PCRE** works on **UTF-8** encoded strings

Unicode: Pattern matching

- Unicode processing is enabled using the `u` modifier
- `PCRE` works on `UTF-8` encoded strings
- Each `code point` is handled as `one character`

Unicode: Pattern matching

- Unicode processing is enabled using the `u` modifier
- `PCRE` works on `UTF-8` encoded strings
- Each `code point` is handled as `one character`
- Match any unicode code point: `\x{FFFF}`

Unicode: Pattern matching

- Unicode processing is enabled using the `u` modifier
- `PCRE` works on `UTF-8` encoded strings
- Each `code point` is handled as `one character`
- Match any unicode code point: `\x{FFFF}`
- Remember the letter `a` with the mark `‘` (`à`)

Unicode: Pattern matching

- Unicode processing is enabled using the `u` modifier
- `PCRE` works on `UTF-8` encoded strings
- Each `code point` is handled as `one character`
- Match any unicode code point: `\x{FFFF}`
- Remember the letter `a` with the mark `‘` (`à`)

```
/\x{0061}\x{0030}/U
```

Unicode: Extended unicode sequences

- How to match the single and multi code point character?

Unicode: Extended unicode sequences

- How to match the single and multi code point character?
 - Remember: à = U+0061 U+0300 oder U+00E0

Unicode: Extended unicode sequences

- How to match the single and multi code point character?
 - Remember: `â` = `U+0061 U+0300` oder `U+00E0`
- Using escape for extended unicode sequences: `\X`

Unicode: Extended unicode sequences

- How to match the single and multi code point character?
 - Remember: `â` = `U+0061 U+0300` oder `U+00E0`
- Using escape for extended unicode sequences: `\X`
- `\X` is equivalent to `(?>\P{M}\p{M}*)`

Unicode: Extended unicode sequences

- How to match the single and multi code point character?
 - Remember: `â` = `U+0061 U+0300` oder `U+00E0`
- Using escape for extended unicode sequences: `\X`
- `\X` is equivalent to `(?>\P{M}\p{M}*)`
 - Wait. What? → **Unicode character properties**

Unicode: Character properties

- Every unicode code point has a certain property assigned

Unicode: Character properties

- Every unicode code point has a certain property assigned
- Characters may be matched by these properties

Unicode: Character properties

- Every unicode code point has a certain property assigned
- Characters may be matched by these properties
- Escapes `\p` and `\P` are used for this:
 - `\p{xx}`: All code points with the property `xx`
 - `\P{xx}`: All code points **without** the property `xx`

Unicode: Character properties

- Every unicode code point has a certain property assigned
- Characters may be matched by these properties
- Escapes `\p` and `\P` are used for this:
 - `\p{xx}`: All code points with the property `xx`
 - `\P{xx}`: All code points **without** the property `xx`
- Possible properties:
 - **L**: Letter
 - **M**: Mark
 - **P**: Punctuation
 - **Sc**: Currency symbol
 - ...

- Recursion in regular expressions ?

Pattern Recursion

- Recursion in regular expressions ?
- Possible with **PCRE**

Pattern Recursion

- Recursion in regular expressions ?
- Possible with PCRE
- Validate BB-Code using PCRE

```
[b>Hello [i]World[/i]![/b]
```

BB-Code Recursion Example

`[b]Hello [i]World[/i]![/b]`

- Recursive regular expression pattern

```
(  
  [^\[]*  
  \[(b|i)\]  
  (?:[^\[]+|(?R))  
  \[/\1\  
  [^\[]*  
)
```

BB-Code Recursion Example

`[b]Hello [i]World[/i]![/b]`

- Recursive regular expression pattern

```
(  
  [^\[]*  
  \[(b|i)\]  
  (?: [^\[]+ | (?R) )  
  \[/\1\  
  [^\[]*  
)
```

BB-Code Recursion Example

`[b]Hello [i]World[/i]![/b]`

- Recursive regular expression pattern

```
(  
  [^\[]*  
  \[(b|i)\]  
  (?:[^\[]+|(?R))  
  \[/\1\  
  [^\[]*  
)
```

BB-Code Recursion Example

`[b]Hello [i]World[/i]![/b]`

- Recursive regular expression pattern

```
(  
  [^\[]*  
  \[(b|i)\]  
  (?: [^\[]+ | (?R) )  
  \[/\1\  
  [^\[]*  
)
```

BB-Code Recursion Example

`[b]Hello [i]World[/i]![/b]`

- Recursive regular expression pattern

```
(  
  [^\[]*  
  \[(b|i)\]  
  (?: [^\[]+ | (?R) )  
  \[/\1\  
  [^\[]*  
)
```

BB-Code Recursion Example

`[b]Hello [i]World[/i]![/b]`

- Recursive regular expression pattern

```
(  
  [^\[]*  
  \[(b|i)\]  
  (?:[^\[]+|(?R))  
  [/\1\  
  [^\[]*  
)
```

Do NOT Parse Using Regular Expressions

- Even though this is possible you do **NOT** want to do it

Do NOT Parse Using Regular Expressions

- Even though this is possible you do **NOT** want to do it
 - It is not maintainable

Do NOT Parse Using Regular Expressions

- Even though this is possible you do **NOT** want to do it
 - It is not maintainable
 - It is nearly impossible to find errors

Do NOT Parse Using Regular Expressions

- Even though this is possible you do **NOT** want to do it
 - It is not maintainable
 - It is nearly impossible to find errors
 - Useful information extraction (building an AST) is not possible

Do NOT Parse Using Regular Expressions

- Even though this is possible you do **NOT** want to do it
 - It is not maintainable
 - It is nearly impossible to find errors
 - Useful information extraction (building an AST) is not possible
- Use regular expressions for

Do NOT Parse Using Regular Expressions

- Even though this is possible you do NOT want to do it
 - It is not maintainable
 - It is nearly impossible to find errors
 - Useful information extraction (building an AST) is not possible
- Use regular expressions for
 - Match Patterns (not recursive structures)

Do NOT Parse Using Regular Expressions

- Even though this is possible you do **NOT** want to do it
 - It is not maintainable
 - It is nearly impossible to find errors
 - Useful information extraction (building an AST) is not possible
- Use regular expressions for
 - Match Patterns (not recursive structures)
 - Tokenizing strings

Do NOT Parse Using Regular Expressions

- Even though this is possible you do **NOT** want to do it
 - It is not maintainable
 - It is nearly impossible to find errors
 - Useful information extraction (building an AST) is not possible
- Use regular expressions for
 - Match Patterns (not recursive structures)
 - Tokenizing strings
 - Validate really restricted input values

Questions, comments or annotations?

Slides: <http://westhoffswelt.de/portfolio.htm>

Contact: Jakob Westhoff <jakob@php.net>

Twitter: @jakobwesthoff

Please leave comments and vote at: <http://joind.in/1620>

Bibliography I

- [1] Wikipedia.
Regular expressions — wikipedia, the free encyclopedia, 2002.
[Online; accessed 25-February-2002].