Regular Expressions – Doing magic with text

- PHP Professional Training
- 20th of November 2007

eZ®
The Information sharing company

About me

- Kore Nordmann
  - Studying computer science at the University Dortmund
  - Working for eZ systems on
    eZ components
  - Maintainer and / or Developer in multiple open source projects: Image_3D, KaForkl, ezcGraph, Torii, Busimess, PHPUnit, WCV, ...
- Contact
  - kore@php.net
  - mail@kore-nordmann.de

Goal:

- Parse BBCodes
  - Example: "`Hello [b]world[/b]!`"
  - Simplified XML-like markup language commonly used in forums.
    - Uses [] instead of <>
    - Different attribute notation
- *(I do not want to discuss the sense of using it, here)*

Agenda:

- A **short** theoretical overview
- The required basics
- Extensions of regular expression
- Matching recursive structures
- Building a regular expression

Regular expressions

- In computer science:
  - Commonly tell you if a word (string) is element of a language (set of strings)
  - Matching only regular languages
- In practice:
  - Check if some input string matches some pattern
  - Tokenizing

Regular expressions in PHP (1)

- POSIX compatible regular expressions
- Represented by the ereg*() functions
  - Also used by some other functions like split()
  - Available in some extemsions like mb_ereg_*()
- Slow, featureless, deprecated, will be moved to PECL

Regular expressions in PHP (2)

- Perl compatible regular expressions (PCRE)
- Represented by the pcre_*() functions
- Fast, mighty, ... better
  - Also supports less mighty POSIX regular expressions

Agenda:

- A **short** theoretical overview
- The required basics
- Extensions of regular expression
- Matching recursive structures
- Building a regular expression

Simple first expression

- We use PCRE, and do not bother with POSIX compatible regular expression
- Match an opening BBCode bold tag: `[b]`
  - `(\[b\])i`

Special characters

- Regexp: `(\[b\])i`
- Special characters in a regular expressions need to be escaped by a \
- State of character depends on context
  - Global regular expression
    - `\^$.[]|()?*+{}`
  - Character classes (later)
    - `\^-]`

Delimiters

- Regexp: `(\[b\])i`
- You may use a lot of different delimiters (except the backslash and alphanumeric characters)
  - Common delimiters: /, #, @
  - My personally preferred delimiters: ()
    - No escaping in the regular expression
    - Intuitive match count. (More later)

eZ®
The Information sharing company

Pattern modifiers

- Regexp: `(\[b\])i`
- Alphanumeric characters appended after the closing delimiter
- Modify the behaviour of the regular expression
  - Common modifiers:
    - i: Case insensitive
    - S: Perform optimizations for non-anchored expressions
    - u: Pattern string is UTF-8
    - U: Ungreedy (explained later)
    - s: . matches all (explained later)
    - m: multiline (explained later)

Using regular expression with PHP

- Regexp: `(\[b\])i`
- PHP-Code:
  - ```
    <?php
    var_dump( preg_match(
        '(\\[b\\])i', 'Hello [b]world[/b]!'
    ) ); ?>
    ```
  - => int(1)
    ```

Why use so many backslashes?

- Backslashes are used for escaping in regular expressions AND PHP itself
  - Double escape everything
- Regexp: Backslash followed by closing brace
  - Regexp in PHP: `'(\\\])'`?
    - Results in: `string(5) "(\\])"`
  - This works: `'(\\\\\])'`
    - Results in: `string(6) "(\\\])"`
  - Absolute correct usage: `'(\\\\\\])'`
    - Results in: `string(6) "(\\\])"`

eZ®
The Information sharing company

Getting back the matches

- Regexp: `(\[b\])i`
- PHP-Code:
  - ```
    <?php preg_match(
        '(\\[b\\])i', 'Hello [b]world[/b]!',
        $matches );
    var_dump( $matches ); ?>
    ```
  - `=> array(1) { [0]=> "[b]" }`
    ```

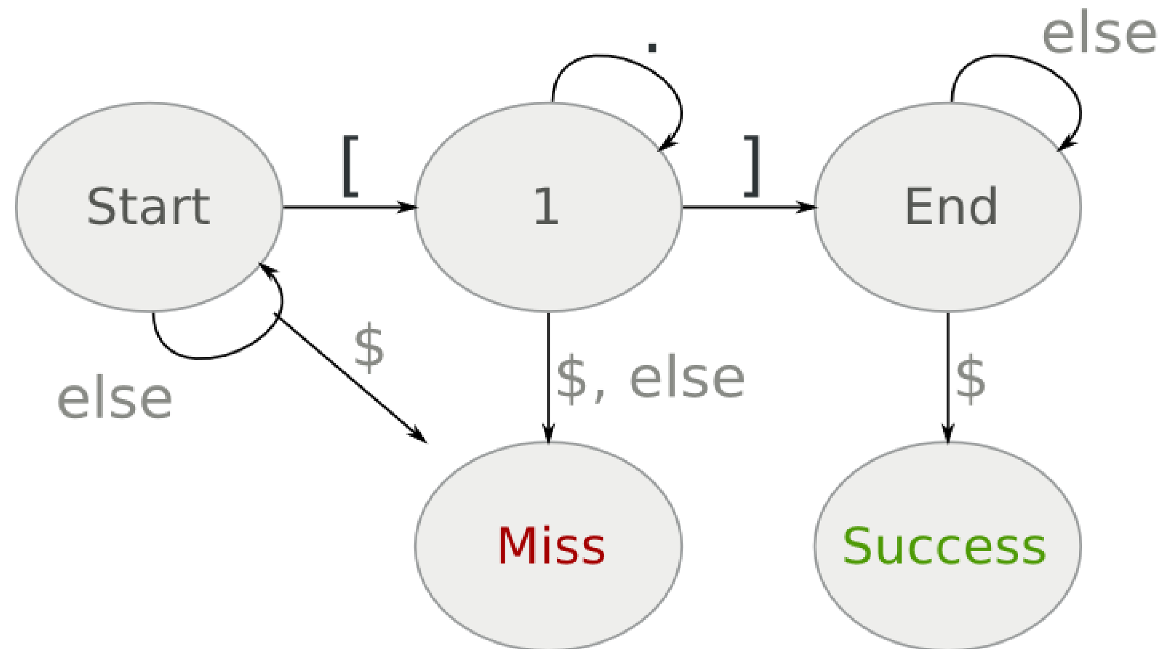
Matching all BBCodes

- Regexp: `(\[[a-z]+\])i`
- Character classes to defined sets of characters
  - Invert matching by ^ at the beginning: `[^a-z]`
  - Define ranges by using `-`
  - Most special chars are no special chars in character classes: `[()]`

eZ®
The information sharing company

How does the matching work?

Regular Expression: (\[.*\])
Input: Hello [b]world[/b]!

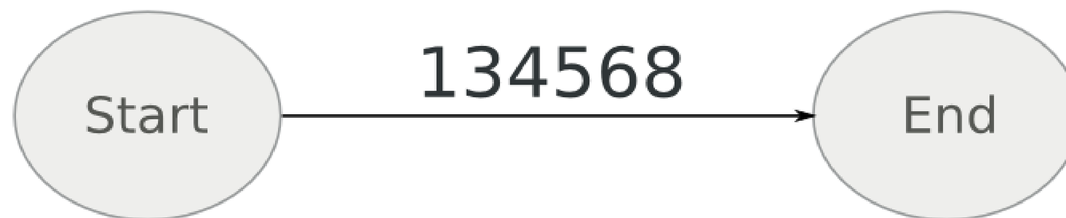


Example: Number spans (1)

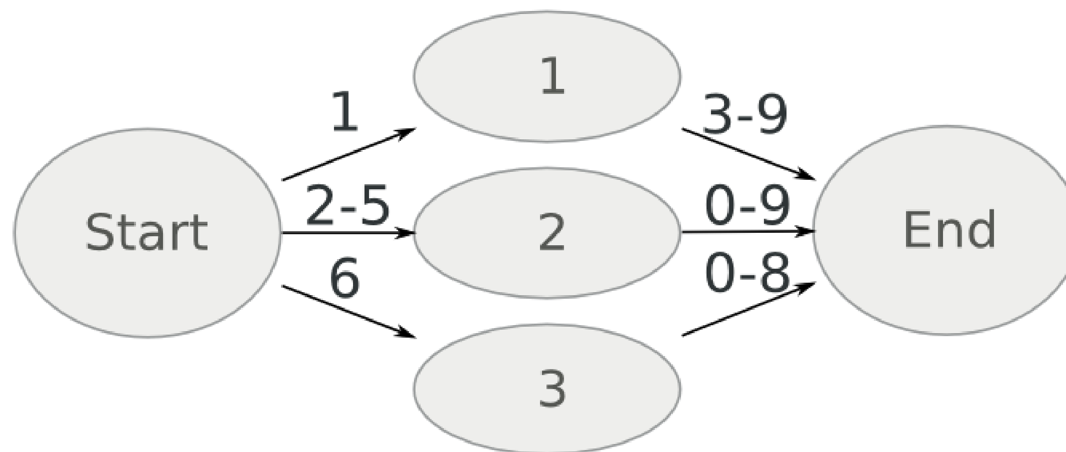
- Regexp: `([13-68])`
  - Matches exactly one character out of “134568”
- Regexp: `(1[3-9]|[2-5][0-9]|6[0-8])`
  - Matches all numbers from 13 to 68

How does the matching work?

Regular Expression: ([13-68])



Regular Expression: (1[3-9]|[2-5][0-9]|6[0-8])



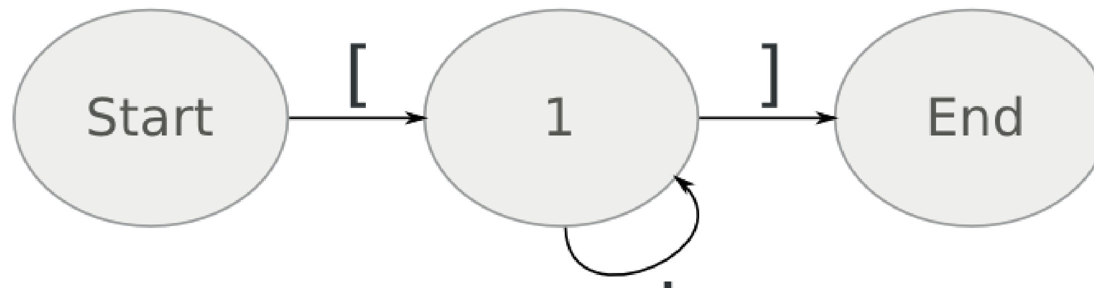
Modifiers S & M

- Multiline string:
  - Hello
    world!
- `(^(.+)$)` => NULL
- `(^(.+)$)`**`m`** => array('Hello', 'world!')
- `(^(.+)$)`**`ms`** => array('Hello world!')
- `(^(.+)$)`**`s`** => array('Hello world!')

More modifiers: U

- Will come back to this again...

Regular Expression: (\[.*\])U
Input: Hello [b]world[/b]!



Agenda:

- A **short** theoretical overview
- The required basics
- Extensions of regular expression
- Matching recursive structures
- Building a regular expression

Backreferences (1)

- Regexp: `(\[([a-z]+)\](.*)\[/\1\])is`
- Matches: array(0 => ..., 1 => 'b', 2 => 'world')

eZ®
The information sharing company

Backreferences (2)

- Input: `[b]Hello[/b] [b]world[/b]!`
- Regexp: `(\[([a-z]+)\](.*)\[/\1\])is`
  - Matches:
    array(0 => ..., 1 => 'b', 2 => 'Hello[/b] [b]world')
- Regexp: `(\[([a-z]+)\](.*)\[/\1\])is`**U**
  - Matches:
    array(0 => ..., 1 => 'b', 2 => 'Hello')
- You may also use .*? to make some sub-expression ungreedy / greedy
- You may use *preg_match_all()* to get all matches

eZ®
The information sharing company

Optional parameters

- Input: `[url=http://kore-nordmann.de/blog/why_are_you_using_bbcodes.html]Look here![/url]`
- Regexp:
  ```
  (\[([a-z]+)(?:=([^\]]+))?\](.*)\[/\1\])is
  ```
- Matches:
  array(
  0 => ..., 1 => "url", 2 => "http://kore[...].html", 3 => "Look here!"
)

Named matches

- Input: `[url=http://kore-nordmann.de/blog/why_are_you_using_bbcodes.html]Look here![/url]`
- Regexp: `(\[(?P<code>[a-z]+)(?:=(?P<parameter>[^\]]+))?\](?P<content>.*)\[/\1\])is`
- Matches:
  array(
  0 => ..., 'code' => "url", 'parameter' => "http://kore[...].html", 'content' => "Look here!"
)

Subpattern assertions

- Input: `[b]Hello[/b] [cmsobject=12]world[/cmsobject]!`
- Regexp: `(\[`**`(?<!cms)([object]+)`**`(?:=([^\]]+))?\](.*)\[/\1\])is`
- Matches: ?
- Other assertions:
  - `foo(?!bar)` foo NOT followed by bar.
  - `for(?=bar)` foo followed by bar.
  - `(?<!foo)bar` bar NOT preprended by foo
  - `(?<=foo)bar` bar preprended by foo

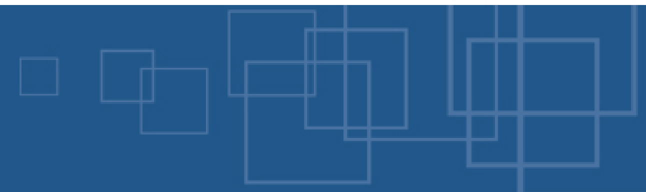
eZ®
The Information sharing company

Conditional Subpatterns

- Regexp: `(\[([a-z]+)(?:=((?(?=ftp)ftp://[^@]+@[^\]]+|[^\]]+)))?\](.*)\[/\1\])i`
- Requires a username (and password) for all parameters with FTP URLs
  - `Click [url=http://foo/]here[/url]!`
    - `=> bool( true )`
  - `Click [url=ftp://foo/]here[/url]!`
    - `=> bool( false )`
  - `Click [url=ftp://user:password@foo/]here[/url]!`
    - `=> bool( true )`

Agenda:

- A **short** theoretical overview
- The required basics
- Extensions of regular expression
- Matching recursive structures
- Building a regular expression



Limitations of regular expressions

- Regular expressions neither know back references, nor subpattern modifiers
- Regular expressions can NOT match recursive structures.
  - Really.
- The very simple language: "**n opening braces, followed by n closing braces**" can not be validated

The mathematical proof

Pumpinglemma:

$$\exists n : \forall x \in L : |x| > n \Rightarrow (\exists u, v, w : (x = uvw, |v| \geq 1, |uv| \leq n, \forall i \in \mathbb{N} : uv^i \in L))$$

Consider the language:

$$L = \{a^n b^n : n \in \mathbb{N}\}$$

Assumption: L is a regular language.
Conclusion: The pumpinglemma is valid.

Choose n, so that the preconditions are fulfilled:

$$x = a^n b^n \Rightarrow |x| = 2\text{n} \Rightarrow |x| > n$$

$$|uv| \leq n \Rightarrow \exists k : a^k, k \leq n \Rightarrow v = a^l, 0 < l < k$$

$$x = \underbrace{a^{k-l}}_{u} \underbrace{a^l}_{v} \underbrace{a^{n-k} b^n}_{w} \Rightarrow |v| \geq 1 \Rightarrow l \geq 1$$

Pumpinglemma requests, that for regular languages:

$$\forall i \in \mathbb{N} : u v^i w \in L$$

Wie choose i = 2, so that:

$$\Rightarrow uv^2 w \in L \Rightarrow a^{n+1} b^n \in L$$

This obviously opposes the initial definition of L, which means L is **not** a regular language.

PCRE are no regular expressions

- PCRE implements a superset of regular expressions
- The language type matched by PCRE is not yet known for sure
  - Try to implement a Turing Machine using PCRE
- Matching the above language is possible with PCRE:
  - `(\((((?>[^()]+)|(?R))*)\))`

Matching recursive BBCode structures (1)

- Input: `Some [b] longer [i]text[/i][/b].`
- Regexp: `(([^\[\]]*\[([a-z]+)(?:=(?[^\]]+))?\](?>[^\[\]]*|(?R))\[/\2\][^\[\]]*))i`
  - Validates correct BBCode structures
    - `Some [b] longer [i]text[/i][/b].`
      - `=> bool( true )`
    - `Some [b] longer [i]text[/b][/i].`
      - `=> bool( false )`
    - `Some [b] longer [i]te [u] xt[/i][/b].`
      - `=> bool( false )`

Matching recursive BBCode structures (2)

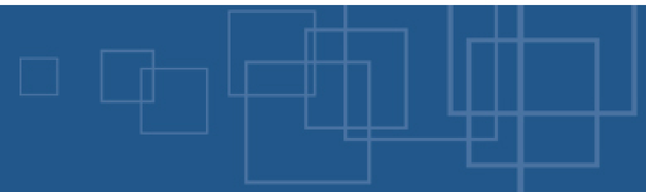
```
(
    (
    [^\[\]]*
        \[([a-z]+)\]
            (?# The actual recursion )
            (?>[^\[\]]* | (?R) )
        \[/\2\]
    [^\[\]]*
    )
)ix
```

Summary

- You should NOT use regular expressions to try such things
  - It is nearly impossible to debug
  - It is not maintainable at all
- You should use (PCRE) regular expressions for:
  - Matching patterns not structures
  - Tokenize

Agenda:

- A **short** theoretical overview
- The required basics
- Extensions of regular expression
- Matching recursive structures
- Building a regular expression



Second Goal:

- Matching RFC compliant URLs
  - Example: `http://kore:password@kore-nordmann.de/blog/index.html?id=1&site=examples#ex1`
  - Regular expression to create:

```
(^(?P<protocol>\w+)://(?:(?P<user>[a-z0-9_]+)(?::(?P<password>[^@]+))?@)?(?P<address>(?:[a-z\d-]+\.)+(?:[a-z]{2,6})|(?:(?:(?:\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])\.){3}(?:\d|[1-9]\d|1\d\d|2[0-4]\d|25[0-5])))(?::(?P<port>[1-9]\d{0,3}|[1-5]\d{4}|6[0-4]\d{3}|65[0-4]\d\d|655[012]\d|6553[0-6]))?(?P<path>(?:/[%.a-z\d_~-]*)*)(?P<parameters>(?:[?&](?:[a-z\d.[\]%-]+)(?:=[a-z\d.[\]%-]*)?)*)(?:\#(?P<anker>\w+))?$)ix
```

- I expect you to understand this!
  - After a step by step guide ;)
- Normally you should just use: *parse_url()*

eZ®
The Information sharing company

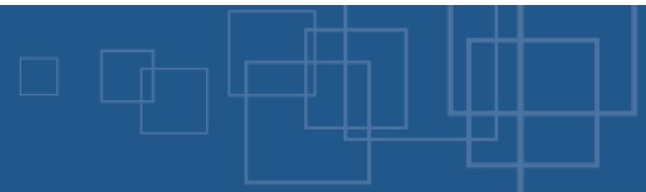
URL - Step 1

- Protocol:
  - ```
    (?P<protocol>\w+)://
    ```
- Optional username and password
  - ```
    (?: (?P<user>[a-z0-9_]+)
        (?: : (?P<password>[^@]+))? @
    )?
    ```

eZ®
The Information sharing company

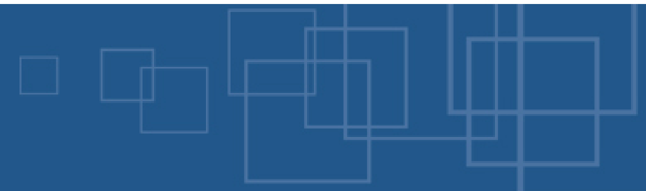
URL - Step 2

- Address:
  -
```
(?P<address>
    (?:[a-z\d-]+\.)+(?:[a-z]{2,6}) |
    (?: (?:(?:[1-9]?\d|1\d\d|2[0-4]\d|
25[0-5]) \. ){3}
           (?:[1-9]?\d|1\d\d|2[0-4]\d|
25[0-5]) )
)
```



URL - Step 3

- Port:
  - ```
    (?: :
      (?P<port>[1-9]\d{0,3}|[1-5]\d{4}|
    6[0-4]\d{3}|65[0-4]\d\d|655[012]\d|
    6553[0-6])
    )?
    ```
- Path:
  - ```
    (?P<path>(?:/[%.a-z\d_~-]*)*)
    ```



eZ®
The information sharing company

URL - Step 4

- Parameters:
  - ```
    (?P<parameters>
      (?: [?&] (?:[a-z\d.[\]%-]+) (?: = [a-z\d.[\]%-]*)? )*
    )
    ```
- Anker
  - ```
    (?:\#(?P<anker>\w+))?
    ```

eZ®
The Information sharing company

URL - Complete expression

```
(^
    (?P<protocol>\\w+)://
    (?:
        (?P<user>[a-z0-9_]+)
        (?: : (?P<password>[^@]+))? @
    )? (?P<address>
        (?:[a-z\\d-]+\\.)+(?:[a-z]{2,6}) |
        (?:(?:(?:\\d|[1-9]\\d|1\\d\\d|2[0-4]\\d|25[0-5])\\.){3}
(?:\\d|[1-9]\\d|1\\d\\d|2[0-4]\\d|25[0-5]))
    ) (?:
        :(?P<port>[1-9]\\d{0,3}|[1-5]\\d{4}|6[0-4]\\d{3}|
65[0-4]\\d\\d|655[012]\\d|6553[0-6])
    )? (?P<path>
        (?:/[%.a-z\\d_~-]*)*
    ) (?P<parameters>
        (?:[?&](?:[a-z\\d.[\\]%-]+)(?:=[a-z\\d.[\\]%-]*)?)*
    ) (?:
        \\# (?P<anker>\\w+)
    )?
$)ix
```

URL - Matching

```
array(9) {
    [0]=> string(79) "http://kore:password@kore-
nordmann.de:80/blog/index.html?
id=1&site=examples#ex1"
    ["protocol"]=> string(4) "http"
    ["user"]=> string(4) "kore"
    ["password"]=> string(8) "password"
    ["address"]=> string(16) "kore-nordmann.de"
    ["port"]=> string(2) "80"
    ["path"]=> string(16) "/blog/index.html"
    ["parameters"]=> string(19) "?
id=1&site=examples"
    ["anker"]=> string(3) "ex1"
}
```

Questions?

- Open questions?
- Ressources:
  - http://php.net/pcre
  - http://kore-nordmann.de

The end

- Thanks for listening
- Contact:
  - kore@php.net
  - mail@kore-nordmann.de