Magic with regular expressions

- Speaker
  - Thomas Weinert
  - Kore Nordmann
- PHP Unconference Hamburg
  - 27.04.2008

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 them, here)*

Agenda:

- A **short** overview
- The required basics
- Matching recursive structures

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** overview
- The required basics
- Matching recursive structures

Simple first expression

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

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)

Pattern modifiers

- Regexp: `(\[b\])i`
- Alphanumeric characters appended after the closing delimiter
- Modify the behaviour of the regular expression
  - Common modifiers:
    - i: Case insensitive
    - Explained later: U, s, m

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?
    ```

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 using ^: `[^a-z]`
  - Define ranges by using `-`
  - Most special chars are no special chars in character classes: `[()]`
- A number span would not look like: `[13-68]`, but: `(1[3-9]|[2-5][0-9]|6[0-8])`

Backreferences

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

Modifiers S & M

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

More modifiers: U

- 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')

Subpattern greedyness

- Regexp: `(\[([a-z]+)\](.*?)\[/\1\])is`
  - You may also use .*? to make some sub-expression ungreedy / greedy

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

Conditional Subpatterns

- Regexp: `(\[([a-z]+)(?:=((?(?=ftp)ftp://[^@]+@[^\]]+|[^\]]+))?\](.*)\[/\1\])i`
- Requires a username (and password) for all parameters with FTP URLs
  - Example code

Agenda:

- A **short** overview
- The required basics
- Matching recursive structures

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:

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

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

Choose n, so that the preconditions are fulfilled:

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

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

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

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

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

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

Questions?

- Please ask!

The end

- Thanks for listening
- Ressources:
  - http://php.net/pcre
  - http://kore-nordmann.de