Komsky Typ-3 Sprachen – oder hab Spaß mit Regex

PHP Conference| 9.11.2005 | Alex Aulbach

Der Referent

- Alexander Aulbach
  - 37 Jahre alt
  - Angestellt bei der Mayflower GmbH
  - Webapplikationsprogrammierung seit 9 Jahren
  - Datenbankentwicklung seit 7 Jahren
  - Artikel im PHP-Magazin zum Thema Regex vor 2 Jahren

Mayflower GmbH

- Gegründet 1997
- München und Würzburg
- zahlreiche Projekte bei europäischen Firmen/Konzernen
  - Vaillant
  - Telefonica
  - HypoVereinsbank
- starkes Wachstum
- thinkPHP bündelt die PHP und LAMP Aktivitäten von MAYFLOWER
  - Core Developer PHP und Apache
  - Starkes Open Source Engagement
    - PHProjekt
    - Lighttpd
    - PHP-Support

Reguläre Ausdrücke in PHP

Regexe sind Beschreibungen für Sprachen vom Typ-3 der Chomsky-Hirarchie (reguläre Sprachen, auch rechts- bzw. linkslineare Sprachen genannt).

Es gibt sie in vielen verschiedenen Ausprägungen (PRCE, Posix-Regex, Suche in Word) und haben eine feste und sehr wichtige Bedeutung in der Programmierung und Konfiguration.

Jede dieser Beschreibungs-Sprachen hat ihre eigene Syntax.

Wir befassen uns in diesem Vortrag speziell mit der von PCRE (perl compatible regular expression), in der Version, welche in PHP momentan hauptsächlich verwendet wird.

Grundlagen

- 48 Folien, daher können wir nicht alles durcharbeiten
- Grundlagen (halbes Semesters Compilerbau in 5 Minuten)
- Anker
- Wiederholungen (Quantifier)
- Alternativen
- Gruppierungen
- Kochrezepte
- Backtracking (und Folgen) und wie es funktioniert
- PCRE-(Inline-)Parameter
- Voraus- und Vorherbehauptungen
- Einmalpattern
- Bedingtes Suchen
- jede Menge Anhang

Grundlagen

- Was ist ein Regex?
- Regex-Engine (wie sie ungefähr funktioniert, dass man den Regex compilieren muss)
- Grobe Syntax und Vermeidung von Escape-Orgien
- Beispiel
- Zeichen, Zeichenketten und Zeichenklassen
- Literale

Was ist ein Regex?

- Rein mathematisch gesehen: Ein Regex findet heraus, ob eine bestimmte Zeichenkette (oder Teile der Zeichenkette) einer bestimmten *„Sprache“* zugeordnet werden können.
- Eine *„Sprache“* besteht aus *„Wörtern“*.
- *„Wörter“* bestehen aus *„Zeichen“* eines bestimmten *Alphabets (Zeichenmenge)*.
- *In unserem Fall ist das Alphabet der Zeichensatz (ISO-8859-15 o. ä.), Wörter wären demnach beliebige so zusammengesetzte Zeichenketten (Strings).*

Was ist ein Regex – so dass man es versteht

- Ein Regex ist ein (extrem komprimiertes) Programm um eine ganz bestimmte Kombination von Zeichenfolgen in einer Zeichenkette zu finden.
- Ein Regex matcht oder matcht nicht.
- Ein Regex findet immer nur die erste passende Kombination (Sprache) aus einer Zeichenkette, er findet also möglichst weit „vorne“ passende Zeichenketten.
- Ein Regex kann nur Sprachen vom Typ 3 oder höher (also „schlechter”) matchen, es gibt aber Erweiterungen in den realen Implementierungen von Regex-Maschinen, die davon eine Ausnahme bilden.

Komsky Typ-3 Sprachen

Was sind sie nicht?

- Ein Regex matcht niemals nur „ein bischen“.
  Entweder der komplette Regex matcht oder nicht!

- Regexe sind nicht unlogisch. (*)
  Sie funktionieren extrem logisch, das heißt wenn sie nicht funktionieren, dann ist man immer selbst schuld.

(*) Ausnahme: Betaversionen

Regex-Engine

- Die Regex-Engine ist in der Regel eine Funktionsbibliothek die eine API zur Verfügung stellt
- Im Fall der PCRE werden Funktionen wie „initialisiere()“, „kompiliere()“, „starte suche()“ usw. zur Verfügung gestellt.
- Beim Kompilieren wird eine Art Mini-Programm erzeugt, das sagt „Suche nach Zeichen X“, „Wenn gefunden suche nach Y“...

PCRE-Engine in PHP

- Leider wird die API in PHP verschleiert, d. h. wenn man preg_match() aufruft, laufen im Hintergrund viele verschiedene Operationen auf der PCRE-Lib ab.
- PHP hasht bereits kompilierte Regexe, d. h. wenn ein Regex bereits einmal benutzt wurde, dann wird er nicht nochmal kompiliert.
- Vorsicht: Ein Regex wird nur als String gehasht, also
  `preg_match("/$hugo/",$bla);`
  kann nur gehasht werden, wenn $hugo konstant bleibt.

Syntax

- Ein Regex in PCRE sieht ungefähr so aus:
  ```
  /eigentlicher Regex/Parameter
  ^------------------^-- Delimiter
  ```
- Delimiter können beliebige Zeichen sein:
  `/` (default aus der Literatur und Geschichte), `|` oder `!` (als Alternative), aber auch `"`, `%`, `&`. Speziell auch Klammernpaare: `()`, `[]`, `{}`, `<>`.
- Problem: Einmal benutzt müssen Delimiter im Regex gequotet werden und verlieren daher die Bedeutung als Metazeichen!
- Ich verwende bevorzugt ° (Grad), weil man das sehr selten benötigt, es auf der deutschen Tastatur leicht erreichbar ist und das Auge sich daran orientieren kann.
- Achtung: Unicode-Editoren! Dort wird ° zu zwei Bytes, und damit für PCRE zu einem Fehler!

Vermeidung von Escape-Orgien

- Durch die freie Wahl des Delimiters kann man also hervorragend sogenannte Escape-Orgien vermeiden, die sich speziell ergeben, wenn man nach `\`- oder `/`-Zeichen sucht.
- Vorsicht beim Quoten in PHP: PHP hat anderen Literal-Quote-Mechanismus wie PCRE!
- Vorsicht: Innerhalb von PHP möglichst nicht in "" schreiben, sondern `''` verwenden (zum Beispiel wegen Doppeldeutigkeit von $).

(Extrem-)Beispiel zum Quoten

Suche nach `\?`, ersetzen durch `?`

- Fragezeichen ist ein Regex-Metazeichen, muss also hier escapet werden.
- Erster Versuch:
  `preg_replace('°\\?°','?',$string)`
  Regex kommt in PCRE als **°\?°** an.
- PHP interpretiert unbekante Escapes als Zeichenkette!
  `\\` -> `\`, aber `\%` -> `\%` und nicht `%`
- PCRE dagegen interpretiert beliebige Escape-Sequenzen als Zeichen:
  `\\` -> `\` und `\%` -> `%`
  (Hinweis: Es gibt einen Parameter **X**, der im Fall von unbekannten ESC-Sequenzen einen Fehler auslöst! Gut zum debuggen!)
- Der korrekte Ausdruck ist demnach
  `preg_replace('°\\\\\\?°','?',$string)`
  Regex kommt in PCRE als `°\\\?°` an.
- \? wird korrekt gematcht und durch ? ersetzt.

Zeichen, Zeichenketten, Zeichenklassen

Typische „Fehler“ oder spezielle Sachen:

- `|blafasl|blubb|`
  Durch Verwendung von | als Delimiter verliert | seine Bedeutung als Alternative, escapen nicht möglich.
- JEDES `[]` matcht immer nur ein Zeichen!
  `°[20-49]°` matcht die Zeichen 0, 1, 2, 3, 4 und 9 (und nicht Zahlen von 20 bis 49)! Richtig:
  `°[234][0-9]°` bzw. `°[2-4][0-9]°`
- Innerhalb von Zeichenklassen gelten andere Escape-Regeln –
  die Metazeichen `?, *, ., + , [, (), {}` usw. verlieren dort ihre Bedeutung:
  `°\*°` == `°[*]°`
  Man kann in Zeichenklassen natürlich weiterhin escapen.
- Doppelt escapen vermeiden: `°[0-9\\d]°`

Literale

- Vermeide °.° (Punkt)!
  Böse Falle: Punkt findet alle Zeichen bis auf Newline!
  Parameter s schaltet um:
  °.°s findet wirklich alle Zeichen!
- Sowohl als auch:
  °\d° == °[\d]° ==
  °[0-9]° == °[[:digit:]]°
- Folglich:
  °[\d\s]° == °[[:digit:][:space:]]°,
- aber != °\d\s° !!
  (Zwei Zeichenklassen! °[\d][\s]°)
- °[0-9\d]° – doppelt gemoppelt

Noch mehr Zeichenklassen und Literale

- °[0-9]° findet die Zeichen 0 bis 9, °[^0-9]° dagegen alle anderen! Also z.B. auch „unsichtbare“ Zeichen an die man gar nicht gedacht hat (Newline, CR).
- °\h° matcht h und nicht \h, weil das kein bekanntes Literal ist. Nochmal Hinweis: In PHP ist “\h“ \h und nicht h
  °\h°X liefert Fehlermeldung!
- Speziell beim Scannen von XML nie vergessen, dass da statt Space auch Tab, Newline oder CR stehen darf und kann: Statt
  °<a href="[^"]*">° besser
  °<a\s+href\s*=\s*(["'])[^\1]*\1[^>]*>°

Anker

- Anker matchen kein Zeichen, sondern „hängen“ sich an eine Position zwischen zwei Zeichen (oder an den Anfang oder das Ende der Suchzeichenkette),
- daher sind °^° und °$° (bzw °\A° und °\Z°) stets wahre Ausdrücke, finden aber kein einziges Zeichen!
- Anker können die Suche ungemein beschleunigen
- Anker können schlau eingesetzt einen Regex wesentlich einfacher verständlich machen, bzw. die Anzahl der möglichen Fälle drastisch reduzieren.
- Anker machen einen Regex „sicherer”

Übersicht über ^ , $, \A, \Z, \z, sowie m, und D-Parameter

	Ohne Param.	m	D
^	Am Anfang	Anfang/nach NL	nur Anfang
\A	Immer nur am Anfang	Immer nur am Anfang	nur Anfang
$	Ende/vor NL am Ende	Ende/vor NL am Ende/vor NL	nur am Ende
\Z	Ende/vor NL am Ende	Ende/vor NL am Ende	nur am Ende
\z	nur am Ende	nur am Ende	nur am Ende

Parameter D gibt's übrigens nicht in Perl...

Wiederholungen (Quantifier)

- Vor einem Quantifier kann ein (beliebig) komplexer Ausdruck stehen: Zeichenklasse, aber auch Gruppe, z. B. `°(XYZ)+°` - matcht beliebige Wiederholungen von XYZ.
- Statt `°X{2,}°` evtl. `°XX+°` verwenden (schneller und übersichtlicher, außer wenn X ein zu komplexer Ausdruck ist)
- Vermeide `°X*°`!
  Weil Falle: Matcht auch, wenn gar nichts gefunden wurde!
  Besser: `°X+°`
- Aufpassen: `°X+?°` matcht nicht XX, weil es auch X matchen kann! Daher genügsame Quantifier nur zusammen mit Zusatzbedingen benutzen!
- Parameter u (ungreedy, genügsam) möglichst vermeiden, da er zu Mißverständnissen führen kann. Gierige und genügsame Quantifier am besten auch so hinschreiben. Außnahme wäre wenn sehr viele genügsame Quantifier im Regex kommen, dann sollte man das kommentieren.

Alternativen

- Alternativen finden in PCRE die erste passende Alternative, im Gegensatz z. B. zu POSIX-Regex, welche alle Alternativen durchspielen muss, um die „längste“ zu finden.
- `°Hugo|hugo°` ist etwas schneller als `°[Hh]ugo°`
- Um einen String exakt zu matchen Anker verwenden: **`°^(Hugo|hugo)$°`**

Gruppierungen und Rückwärtsreferenzen

- PCRE kann bis zu 65535 Gruppen bilden, die auf bis zu 256 Ebenen ineinander verschachtelt werden können.
- Gefundene Gruppen werden der Reihenfolge nach in die Register #0, #1, #2 usw. geschrieben.
- Die erste (und immer) gefundene Gruppe ist immer der gesamte Regex (#0)! Kann aber nicht als Rückwärtsreferenz benutzt werden, da #0 ja erst außerhalb des Regex gefunden wird (außer `preg_replace()`). Siehe aber z. B. preg_match_all().
- Erinnerungsloses Gruppieren:
  `°hu(go)°` -> #0 ist hugo, #1 ist go
  `°hu(?:go)°` -> #0 ist hugo, #1 nicht gesetzt,
  etwa doppelt so schnell!
- Referenzen mit Quantifiern enthalten am Ende immer den zuletzt gefundenen Match.

Ausflug: Kochrezepte (1)

- Suche nach Zeichenketten der Form
  :abc:def:ghi:
  (siehe z.B. /etc/passwd)
- gesucht wird z. B. abc
- Erster Versuch
  °:(.*):° - findet alles (#1 ist abc:def:ghi) – nicht gudd
- siehe auch Grundregel: Match mit . möglichst vermeiden!
- Zweiter Versuch:
  °:[^:]*:° - findet als #1 abc
- Ebenfalls möglich (in PCRE):
  °:.*?:°

Ausflug: Kochrezepte (2)

- Suche in (einmal) verschachtelten Kommentaren:
  /* abc *** /* cde */ efg */
- Erster Versuch:
  °/*(.*)*/° findet in #1 abc *** /* cde */ efg
  – erstaunlicherweise korrekt
- Zweiter Versuch:
  °/*([^*]*)*/° - * darf durchaus als einzelnes Zeichen vorkommen
- Dritter Versuch:
  °/*(([^*]|*+[^*])***)*/° - unübersichtlich, aber geht
- Analog: Suche in gequoteten Texten der Form
  “Text enthält \“-Quotes“
  °"([\\\\]*\\\\.[^\\\\]*)"°s (siehe oben)

Ausflug: Kochrezepte (3)

- BTN-XML-Parser (BTN – better than nothing) Version 0.1:
  `°<(\w+)[^>]*>.*?</\1>°s`
  Schönes Beispiel für
  - Manchmal ist ein guter Hack besser als eine perfekte Lösung
  - `.` ist prima, wenn man weiß was man tut
  - `*` ist prima, wenn man wirklich weiß was man tut
  - Genügsame Quantifier nur im Zusammenhang mit Zusatzbedingungen nutzen

Ausflug: Kochrezepte (4)

- BTN-XML-Parser, V 0.2:
  `°<([^>\s]+?)(\s+[^>]+?)?(/)?>(?(3)(?# war 3 leer?)|(.*?)</\1>)°mx`
  - Ausführen mit `preg_match_all()`! Beispiel: huhu
  - #1: Enthält den Tagnamen: a
  - #2: den Rest des Tags href="bla"
  - #3: War Tag ein Einfachtag: war es nicht
  - #4: Taginhalt (inclusive aller Tags): huhu - Aufruf dann über PHP rekursiv
  - Vorsicht: Version 0.2!

Backtracking (und Folgen)

- Backtracking wird ausgelöst, wenn ein Regex nicht matcht, er aber noch nicht alle Möglichkeiten durchprobiert hat, also speziell im Zusammenhang mit Quantifiern
- Viele Möglichkeiten „versehentlich“ Backtracking zu erzeugen werden von PCRE erfolgreich unterbunden.
- Beispiele:
  ° `(h+)*ugo` ° auf den Suchstring
  hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhuhugo
  benötigt ca. 20 Minuten (heute dank weiterer Optimierung schneller!)
  Kombination aus `*` und `+` etwas besser versteckt:
  ° `(\D+|<\d+>)*[!?]` ° - gleicher Suchstring...

Komsky Typ-3 Sprachen

Wie funktioniert Backtracking?

- Beispiel-Regex: `°(h+)*ugo°`
- Beispiel-Eingabe: 'hhhuhugo'
- Abarbeitung von innen nach außen (Klammern zuerst) und links nach rechts
- `h+` matcht gierig also 'hhh'.
- `()*` muss sich mit einem Match begnügen ('hhh' matcht einmal)
- `u` matcht
- `g` matcht nicht -> Backtracking
- **`()*`** matcht jetzt 0-mal (obwohl es einmal könnte)
- **`u`** matcht nicht -> Backtracking
- **`()*`** kann nicht weniger matchen, also geht es in die Klammer
- **`h+`** matcht ein Zeichen weniger: 'hh' (2-mal statt 3-mal)
- **`()*`** kann nun einmal 'hh' matchen. Zweimal geht nicht. **`u`** macht nicht -> Backtracking
- **`h+`** gibt noch ein h frei.
- **`()*`** matcht nun dreimal 'h'
- **`u`** matcht
- **`g`** matcht nicht -> Backtracking
- ... und so weiter

(Inline-)Parameter: x

- ```
  $pattern='°h # sucht h
  u # findet ein u
  g # dann das g
  o # und abschließend o
  °x';
  ```
- ```
  °h(?# suche nach h und dann ugo)ugo°
  ```

Parameter: e

- e-Parameter:

```
1: $input='hugo="boss"&power="78%"';
2: $encoded=preg_replace('°([%"])°e','sprintf("%%%
x",ord("\1"))',$input);
3: echo "E: $encoded"; // E: hugo=%22boss%22&power=%
2278%25%22
4: $decoded=preg_replace('°%([[:xdigit:]]
[[:xdigit:]])°e','chr(hexdec(\'\1\'))',$encoded);
```

- Bei komplexeren Problemen eine eigene Funktion dafür schreiben:

```
function aulencode($str) {   return( sprintf("%%%
x",ord($str)) );   }
...
$encoded=preg_replace('°([%"])°e','aulencode
("\1")',$input);
```

- bzw. `preg_replace_callback()`

Voraus- und Vorherbehauptung

- Nichts weiter als „Anker-Baukasten“
- Kann sehr viel Arbeit sparen
- Ungeheuer mächtig
- spart u. U. enorm an Rechenzeit
- Man kann bis zu 200 Vorher- und Vorrausbehauptungen ineinander schachteln

- Aufgabe: Nachbilden des Wortankers `\b` durch eine Voraus-/Vorherbehauptung
- Nachbilden der POSIX-Ausdrücke \> und \<

Einmalpattern

- Werden benutzt um Backtracking zu vermeiden.
- Gierige Quantifier werden geizig, genügsame werden faul.
- Können unglaubliche Geschwindigkeitsvorteile bringen.
- In der Regel kommt man „konventionell“ genauso weit.

Bedingtes Suchen

- Das „If“ unter den Regex-Erweiterungen.
- Hier verlässt man endgültig Typ-3-Sprachen und bewegt sich hinauf bis zu Typ-1 oder gar Typ-0.
- Reichlich komplex, daher verwendet man es nur, wenn das Problem auch entsprechend komplex ist, oder man beherrscht das aus dem ff.
- Ungeheuer mächtig (kann verschachtelt werden) und sauschnell.

Rekursion mit Regex

- Grundsätzlich: Finger weg. Alle Aufgaben, die rekusive Regex benötigen, sind nicht mehr Komsky-Typ-3!
- Rekursion: Zum Beispiel Kommentar in Kommentar in Kommentar ...
- In PCRE ursprünglich als „Designstudie“ implementiert
- Rekursion sollte meiner Meinung nach im Regelfall „per Hand“ durchgeführt werden, sprich schreiben eines echten Parser/Scanners (zum Scannen kann nach wie vor PCRE sehr dienlich sein).
- Rekursionen bis 2 Ebenen kann man mit „normalen“ PCRE-Bordmitteln auch noch auflösen (schwierig, aber möglich).

Anhang

- Regexe und Sicherheitslöcher?
- Zeichensätze und wie sie PCRE beeinflussen
- Was ist generell zu beachten?
- Konventionen beim Kommentieren
- Diskussion
- Was wurde nicht berücksichtig?
- Meine Wünsche für die Weiterentwicklung

Sicherheitslöcher?!

- PCRE ist rein grundsätzlich einer Programmiersprache gleichzusetzen (halt nur Typ-3, aber Programmiersprache ist da im wesentlichen Programmiersprache).
- Kein Mensch kann garantieren, dass die PCRE-Lib sicher ist. (Typische Bugs: Speicherfehler, Stacküberlauf usw.)
- Das Problem gilt aber im wesentlichen nur für den Programmierer, also denjenigen der die Regex nutzt, bzw. erstellt, denn die typischen Probleme ergeben sich erst beim Kompilieren und insbesondere beim Parsen des Regex an sich!
- Läuft ein Regex fehlerfrei, stellen freie Benutzereingaben für Suchstrings im allgemeinen kein Sicherheitsloch dar. (außer natürlich der Suchstring ist sehr, sehr lang!)

Sicherheitslöcher!!

- Freie Suchstrings UND freier Regex´in sicherheitskritischer Umgebung: BIG HACK [tm]!!
- Auch freie Regex alleine können schon ein gravierendes Problem auslösen, indem man zum Beispiel einen Regex schreibt, der Backtracking auslöst und so einen DoS auslöst!
- Also generell: Der Regex sollte nur in ganz speziellen Ausnahmefällen direkt vom User eingegeben werden können.
- Geht es nicht anders: Metazeichen entfernen oder sicheren Benutzerkreis schaffen.

Zeichensätze und wie sie PCRE beeinflussen

- Der Zeichensatz beeinflusst das Matching der Zeichenklassen/Literale
  Zum Beispiel: Im ASCII-Zeichensatz matcht °\w° 63 Zeichen, im deutschen sind es schon 128 und im griechischen 132.
- i-Parameter wird (noch) nicht für Umlaute unterstützt!
- Problematik von UTF: noch nicht komplett durch alle Applikationsschichten gezogen

Warum und wie man Regexe immer testen sollte

- Ein Regex ist wie ein zwar sehr kurzes, aber komplexes Programm zu behandeln!
- Leicht durcheinandermischen von genügsamen und gierigen Quantifiern. Daher verwenden von genügsamen Quantifiern:

```
°<a\s+href\s*=\s*(["'])[^\1]*?\1([^>]*)>°
```

- Zum Testen z.B. regtester (zum Beispiel http://www.ssilk.de/) oder sonstige Notlösungen verwenden
- Notlösung Gruppieren:

```
if ( preg_match('°<a\s+href\s*=\s*(["'])
([^\1]*?)\1([^>]*)>°',$matches) )
    print_r($matches);
```

Grundregeln zum Erstellen eines Regex (1)

- Regex nach vorne und hinten begrenzen!
  Eindeutige „Einfangsequenzen“ bauen, falls der Text, den man durchsucht „beliebig böse“ sein darf. Anker und/oder Vorher-/Vorrausbehauptungen verwenden (sowohl am Anfang, als auch am Ende der eigentlichen Suche).
  Beispiel:
  `preg_match('°hugo°',$_REQUEST[input]);`
  vs
  `preg_match('°^hugo$°',$_REQUEST[input]);`
- So eng wie möglich matchen: Den zu suchenden Ausdruck möglichst genau abbilden.
  Das heist: Sowohl was die Zeichenklasse, als auch was die Länge anbelangt möglichst genau der zu suchenden Sprache annähern.
  `°.ugo°` vs. `°hugo°i` vs. `°[Hh]ugo°`

Grundregeln zum Erstellen eines Regex (2)

- Im Zweifel den Text erst in kleinere Teile aufsplitten und dann auf diesen Operieren. (preg_match_all(), preg_split (), explode(), ...). Hier aufpassen wegen Speicherverbrauch! Meist findet man mit dieser Methode systematische Fehler sehr leicht.
- Auf der anderen Seite: Nicht unnötig reglementieren! Habe ich alle Eingabedaten berücksichtigt?
  Siehe HTML: Setze ich z. B. das Zeichen Space vorraus, wo auch mehrere Spaces, kein Space oder gar ein Tab, Newline, Carrige Return stehen könnte?
- Angemessenes Verhältnis aus Genauigkeit und Freizügigkeit finden, indem man möglichst viel zulässt, den Zeichensatz jeweils aber möglichst genau eingrenzt.
- Regex kommentieren, sobald er etwas komplexer wird, auch sich selbst zuliebe...
- Diskussion: Was ist besser? Schneller? Stabiler?
  `°(.*?)\t(.*?)°` oder `°[^\t]*\t[^\t]*°`

Komsky Typ-3 Sprachen

Grundregeln zum Erstellen eines Regex (3)

- Mit Unicode: Zeichensätze genau beachten!
  PCRE beachtet bei vielen Zeichenklassen (z.B. `\w`) die momentane Spracheinstellung! Es gibt aber zum Beispiel keine einzige vordefinierte Zeichenklasse, die einfach alle ISO-Latin1-Zeichen matcht!
  ```
  preg_match('°\w+°',$bla);
  ```
  vs
  ```
  preg_match('°[\w\xc0-\xd6\xd8-\xf6\xf8-
  \xff]+°',$bla);
  ```
- Besonders übel: Unterschiedliche Behandlung von Umlauten in verschiedenen Editoren (z.B. vim auf einer neuen und alten Distribution).
  Hier hilft leider nur aufpassen!

Grundregeln zum Erstellen eines Regex (4)

- Komplexe Regex sollten vor dem Einbau in ein Programm getestet werden. (z.B. mit regtester von http://www.ssilk.de/)
- Unterscheiden zwischen 08/15-Regex und komplexen Aufgaben! Regex sollen die Arbeit erleichtern und die Suche beschleunigen. Entweder ein Regex ist einfach, dann schreibt man ihn einfach hin, oder komplex, dann kommentiert man ihn und treibt einen größeren Aufwand beim Erstellen und Testen.
- Ein komplexer Regex kann Stunden dauern bis er läuft, sollte aber auch Stunden sparen. Also immer überlegen, ob konventionelle PHP-String-Funktionen nicht genauso den Zweck erfüllen.
- Regex eignen nur Bedingt für Rekursionen, z. B. verschachteltes HTML. Nur sinnvoll mit 3rd Generation-Language-Sprache lösbar.

  (1st GL : Maschinensprache; 2nd GL : Assembler, teilweise auch C; 3rd GL: C, PHP, Perl, Cobol usw.; 4th GL : SQL)

Komsky Typ-3 Sprachen

Grundregeln zum Erstellen eines Regex (5)

- Bevor man anfängt einen Regex zu schreiben, benötigt man gute Beispiele für Suchtexte!
- Nicht jeder Regex muss genau erklärt werden, man sollte die Grundlagen vorraussetzen können.
- Ein Regex ersetzt kein falsches Konzept! Bestes Beispiel: Prüfen einer E-Mail-Adresse oder URL auf Korrektheit.

Beispiele zum diskutieren

- Groß-Kleinschreibung:

```
preg_match('°^hugo$°i',$name)
```

vs

```
preg_match('°^hugo$°',strtolower($name))
```

- Vergleich mehrerer Strings

```
if ( $name == 'hugo' or $name=='egon' or
$name=='servantes' )
```

vs

```
if ( preg_match( '°hugo|egon|servantes°',
$name) )
```

Noch mehr Beispiele

Vergleich von noch mehr Strings

```
$names = array
         ('hugo','egon','servantes');
foreach ( $names as $n )
   if ( $name == $n ) ...
}
```

vs

```
$names = array
          ('hugo','egon','servantes');
if ( preg_match('°^(?:'.
                 explode('|',$names).
                 ')$°',
     $name ) ) {
...
}
```

Schönes/einfaches Beispiel für automatisiert erstellte Regex.

Und noch mehr Beispiele

```
preg_match('°http://(www\.\w+\.(?:org|
mil|com|net|gov|... ~100 tlds ...))(/.*)?
°',$url);
```

vs

```
preg_match('°http://([\w.-]+)(/.*)?
',$url);
```

vs

```
preg_match('°https?://([[:alnum:].-]+\.
[[:alnum:]-]+)(/\S*)?°',$url);
```

vs

```
preg_match('°[[:alpha:]]{2,}://
([[:alnum:].-]+\.[[:alnum:]-]+)(/\S*)?
°',$url);
```

Was ist zu einschränkend, was schon zu allgemein?

So soll man's nicht machen

- ```
  $body = @eregi_replace("(((f|ht){1}tp://)[a-
  zA-Z0-9@:%_.~#-\?&]+[a-zA-Z0-9@:%_~#\?&])",
  "<a href=\"\\1\" target=\"_blank\">\\1</a>",
  $body); //http
  ```
- Klammerorgie unnötig. \0 tuts auch.
- Wozu nur ftp/http? was ist mit anderen Protokollen?
- {1} – sinnlos
- Wo wird der / in einer URL gematcht?
- '#-?' (also die Zeichen von # bis ?) oder die Zeichen '#', '-' und '?' ?
- Domain ist ein genau festgelegter Zeichensatz, wohingegen der URI-Teil viel mehr Zeichen erlaubt.
- Es fehlt die Abgrenzung nach vorne und hinten. ('sftp://bla.bla.bl&' -> erlaubt?!)
  ```

… noch mehr Beispiele

- ```
  $body = @eregi_replace(
  "([[:space:]()[{}])(www.[a-zA-Z0-9@:%_.~#-\?&]
  +[a-zA-Z0-9@:%_~#\?&])",
  "\\1<a href=\"http://\\2\"
  target=\"_blank\">\\2</a>", $body); // www.
  ```
- ```
  $body = @eregi_replace(
  "([_\.0-9a-z-]+@([0-9a-z][0-9a-z-]+\.)+[a-z]
  {2,3})",
  "<a href=\"mailto:\\1\">\\1</a>", $body); // @
  ```
- ```
  preg_match_all(
  "#__\\(['\"](.*)(?<!\\\\)['\"]\\)#msiU",
  $content, $res)
  ```
  Preparser der alle Funktionen mit __() findet. Funktioniert nicht in allen Fällen!
  ```

Gewachsene Programme vs gewachsene Regexe

- Ein Regex sollte – im Gegensatz zu üblicher Programmierweise – nicht wachsen, denn sie sind die Lösung für genau ein einziges spezielles Problem!
- Das bedeudet, dass man bei jeder Änderung eines Regex diesen komplett neu testen sollte. (und eigentlich auch muss)
- Das bedeudet auch, dass man möglicherweise (und häufig!) besser damit fährt, ein Problem mit Hilfe von Regex möglichst von Anfang an komplett zu erschlagen, anstatt immer weitere spezielle Erweiterungen einzubauen.
- Das bedeudet auch, das Regexe nicht das Allheilmittel sind! Nicht unnötig Aufwand in etwas stecken!

Beispiel: Seriennummern parsen

```
// 26 chars + N + Checksum
if ( !preg_match(
'°^[\d<]+N\d$°',$sernum) )
    return false;
$el=array(
    'Data-Identifier'=> 2,
    'Prod-Year'      => 2,
    'Prod-Week'      => 2,
    'Art-Number'     => 10,
    'Manufactory'    => 4,
    'Product-counter'=> 6,
    'NewSernum'      => 1,
    'Checksum'       => 1,
);
```

```
[...]
$pos=0;
$ret=array();
foreach ( $el as $name => $elen ) {
    $ret[$name]=substr($sernum,$pos,$elen);
    $pos+=$elen;
}
```

=> Manchmal ist es besser alles mit "herkömmlichen Methoden" zu machen, weil ein entsprechender Regex zum einen ziemlich kompliziert wäre und zum anderen man sich die genaue Dokumentation des Regex "spart", indem man stattdessen Code schreibt, den man viel leichter versteht als einen Regex.

Was wurde nicht berücksichtig?

- rekursive Regexe. Braucht man wie gesagt meiner Meinung nach nicht, bzw. sind so komplex, dass man sich sowieso tief in die Materie einarbeiten muss
- UTF-8 Unterstützung
- neueste PCRE-Library-Version (Gruppenbildung mit Namen und solcherlei „Leckerli“)

Meine Wünsche für die Weiterentwicklung

- Zugriff aufs Kompilat bzw. den Parsebaum, um einen besseren Regtester zu programmieren und um Regexe auf höherer Ebene zu optimieren oder lesbarer zu machen (Autokommentierung)
- Aufruf der einzelnen PCRE-Lib-Funktionen „per Hand“ ermöglichen
- In den Doks für Regexe eingehen auf die immer wieder auftauchenden Fragen zu „wie parse ich eine Url, wie parse ich eine Mail...“

Regex-Test-Suite: http://www.ssilk.de/

Vielen Dank für Ihre Aufmerksamkeit

Alex Aulbach

Mayflower GmbH
Pleichertorstr. 2
97070 Würzburg

+49 (931) 35 9 65 - 23

aulbach@mayflower.de