Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

Regulære udtryk, 1. del

Validere input - udtrække tekst af tekst efter specielle kriterier - efterbehandle tekst. Regulære udtryk kan alt dette og mere til. Denne artikel prøver at give en oversigt over de mest almindelige ingrediens er, som bruges i regex.

Skrevet den **02. Feb 2009** af **nielle** I kategorien **Programmering / Reg.Exp.** | ★★★☆☆

Indledning

Regulære udtryk er et emne der dukker op med regelmæssige mellemrum her på Eksperten. Alligevel er der ingen som har skrevet en artikel (-serie) om emnet endnu. Dette er et forsøg på at rette op på denne mangel.

Regulære udtryk, eller blot *regexp* eller *regex*, bruges groft sagt til at lede efter tekst i tekst. Man kalder ofte selve regex'en for et *mønster* (engelsk: *pattern*) og man taler om at det *matcher* den tekst man leder efter.

Man skal i øvrigt være opmærksom på at selvom der er regexp understøttelse i mange programmeringssprog, er der forskelle i hvad de enkelte "smagsvarianter" (man snakker om *regex-flavors*) kan gøre, og der er endvidere stor forskel på, hvordan koden skal se ud. Når du derfor stiller et regexp spørgsmål her på Eksperten bør du altid huske at nævne hvad du programmer i.

I denne artikel har jeg valgt at holde mig til preg_xxxx() smagsvarianten af PHP idet jeg gætter på at det (pt.) er der de fleste støder på regexp for første gang.

v. 1.0: 4/12/2007 - Første version.
v. 1.1: 15/12/2007 - Rettelse af nogle mindre fejl, noget formatering, og et løft af hatten til el_barto's kommentar.
v. 1.2: 07/03/2008 - Tilføjelse af feedback afsnit.

Hvad bruges de til?

Regulære udtryk kan bruges til:

(o) Verificere : at en tekst ser ud på en given måde; bruges f.eks. til validering at input-felter fra en form.
(o) Udtrække : en deltekst af en større tekst; bruges f.eks. til at parse en større tekst for bestemte oplysninger.
(o) Erstatte : en deltekst af en tekst med noget andet; bruges f.eks. efterbehandle en tekst for at få den til at se rigtig ud (til den sammenhæng den skal bruges i).

3 hurtige eksempler

Verifikation:

```
<?
$adresse =
"
```

```
IDG Danmark A/S
Carl Jacobsens Vej 25
2500 Valby
Tlf.: 77300300
Fax: 77300301
";

if (preg_match("/\d{8}/", $adresse)) {
    echo "Adressen indeholder et telefonnummer.";
} else {
    echo "Adressen mangler mangler et telefonnummer.";
}
?>
```

Resultat: "Adressen indeholder et telefonnummer."

Udtræk af tekst:

```
<?
$adresse = " ... ";

if (preg_match("/\d{8}/", $adresse, $matches)) {
    echo "Telefonnummeret er: " . $matches[0];
} else {
    echo "Adressen mangler mangler et telefonnummer.";
}
?>
```

Resultat: "Telefonnummeret er: 77300300".

Søg og erstat:

```
<?
$adresse = " ... ";

$adresse = preg_replace("/\d{8}/", "xxxxxxxx", $adresse);
echo "Adressen: " . $adresse;
?>
```

Resultat: Adressen: IDG Danmark A/S Carl Jacobsens Vej 25 2500 Valby Tlf.: xxxxxxxx Fax: xxxxxxxx

Et regulært udtryk som matcher et telefonnummer

(hvor jeg går let og elegant hen over det faktum at der er noget som hedder landekoder, lokalnumre, eller at folk kan finde på at opdele et telefonnummer med mellemrum osv.)

De tre eksempler ovenfor er alle sammen baseret på et og samme regulære udtryk:

(o) \d{8}

Man skal ikke lade sig narre af at der står "/.../" rundt om dette - sådan er syntaksen simpelthen bare i PHP (den er i øvrigt arvet fra Perl).

Hvorfor er det lige \d{8} ? Forklaringen er at:

(o) \d : er en forkortelse for *digit* (engelsk) og betyder *ciffer* på dansk.
(o) {8} : betyder "præcis 8 ...".
(o) \d{8} : betyder derfor "præcis 8 cifre".

Grunden til at der står '\' foran 'd' er at det ellers ville blive opfattet som bogstavet 'd':

(o) d{8} : betyder "præcist 8 gange bogstavet 'd'".

At placere et '\' på den måde foran d'et kaldes at *escape*. \d er kun en af flere muligheder, og dem komme jeg ind på senere.

Men lad os begynde ved begyndelsen

Et almindeligt tegn vil matche sig selv.

Adresse-eksemplet ovenfor indeholder både et telefonnummer og et faxnummer. Hvis man ville være helt sikker på at det var telefonnummeret man matchede på og ikke faxnummeret, så kunne man gøre det med følgende mønster:

Tlf.: \d{8} : matcher på "teksten 'Tlf. ' efterfulgt af præcist 8 cifre". Læg mærke til at der er et enkelt mellemrumstegn inkluderet i mønsteret.

Teksten "Tlf.: " matcher altså sig selv og brugt i netop denne sammenhæng betyder det at vi ikke ved et uheld kan komme il at matche på faxnummeret.

Specialtegn og escape

Og så alligevel ikke...

Ikke alle almindelige tegn matcher sig selv. Nogle har specielle betydninger som jeg kommer ind på senere. Det er tegnene: '[', ']', '\', '^', '$', '.', '|', '?', '*', '+', '(' og ')'.

Lad mig tage '.' og '\' med det samme:

(o) . : matcher "et vilkårligt tegn". Eneste undtagelse er at '.' ikke normalt matcher linjeskift tegnene (\r og \n).

Det er altså en slags trumfkort. Det er vores held at '.' derfor også (helt tilfældigt...) matcher et punktum, for ellers ville mit eksempel med telefonnummeret for før ikke virke. Men faktisk ville det mønster også have matchet sådan noget som:

TlfX: 12345678

fordi at et '.' også matcher et 'X'.

Hvis man ønsker at annullere den specielle betydning af disse tegn, så skal de *escapes* ved at man sætter

et '\' foran. Hvis man f.eks. ikke ønsker at mønsteret fra tidligere også skal kunne matche versionen med X'et så skal det se sådan her ud:

(o) Tlf\.: \d{8}

Tegnet '\' bruges altså til at escape med; men nogen gange er det dog bare et '\' tegn man faktisk er interesseret. F.eks. som hvis man forsøger at matche en filsti:

C:\WINDOS\system32\notepad.exe

For at matche den, skal '\' tegnene (og '.') selv escapes:

(o) C:\\WINDOS\\system32\\notepad\.exe

Dertil kommer at mange programmeringssprog så også selv kræver at '\' tegn i en streng skal escapes. Man kan derfor komme til at skulle "dobbelt-escape":

```
<?
$filePath = 'C:\WINDOWS\system32\notepad.exe';

$pattern = "C:\\\\WINDOWS\\\\system32\\\\(.+)";  // Dobbelt escape!
$phpPattern = "/$pattern/";

preg_match($phpPattern, $filePath, $matches);

echo "Filen hedder: " . $matches[1];
?>
```

Resultat: Filen hedder: notepad.exe

Altså en escape for regex'en og yderlig en escape for programmeringssproget. Lækkert ikke?

Repetition-operatore (*qualifers*)

Et telefonnummer består af 8 cifre (med de førnævnte forbehold) og det kan derfor matches med mønsteret:

(o) \d\d\d\d\d\d\d\d

men så er det nu nemmere at skrive \d{8} og det er egentlig også nemmere at gennemskue meningen med denne kortere form. Som tidligere nævnt er {8} en repetition-operator som siger "præcis 8 ...". Der er andre i samme familie:

(o) {4} : matcher "præcis 4 ...".
(o) {4,} : matcher "4 eller flere ...". Derfor vil f.eks. [a-z]{2,} matche "2 eller flere af bogstaverne 'a' til 'z'". Dette er f.eks. nyttigt når man ønsker at matche toplevel-domain delen af en webadresse.
(o) {4,8} : matcher "mellem 4 og 8 ...". Derfor vil \d{3,4} matche "3 til 4 cifre". Dette kan bruges til at matche et postnummer (på Færøerne har de kun 3 cifre i postnumrene!).
(o) {,8} : matcher "højest 8 ...".

Og så er der nogen som bliver brugt så tit at de har fået et slags alias:

(o) * : matcher "0 eller flere ...".
(o) + : matcher "1 eller flere ...". Derfor vil f.eks. [a-f]+ matche "1 eller flere af bogstaverne 'a', 'b', 'c', 'd', 'e' eller 'f'".
(o) ? : matcher "0 eller 1 ...". Derfor vil f.eks. "billeder?" matche både "billede" og "billeder".

Faktisk er '*' og '+' og '?' blot forkortelser af hhv. {0,} og {1,} og {0,1}.

Karakter klasser

Jeg har allerede fortalt om \d som matcher "et ciffer". Dette er en ud af flere lignende:

(o) \d : matcher "et ciffer".
(o) \D : matcher et vilkårligt "tegn som ikke er et ciffer".
(o) \w : matcher et vilkårligt "tegn som er et 'a' til 'z' eller 'A' til 'Z' eller '0' til '9' og tegnet '_'".
(o) \W : matcher et vilkårligt tegn som ikke er "et 'a' til 'z' eller 'A' til 'Z' eller '0' til '9' eller tegnet '_'".
(o) \s : matcher et vilkårligt "whitespace-tegn", dvs. et mellemrum, et tabulatortegn (matches også af \t), et carriage-return tegn (\r) eller et newline-tegn (\n).
(o) \S : matcher et vilkårligt "tegn som ikke er et whitespace-tegn".

Bemærk at \w normalt ikke inkluderer de danske bogstaver 'æ', 'ø' og 'å'. I visse tilfælde kan den dog godt finde på at udvide sig til at inkludere landespecifikke tegn. Det afhænger desværre af programmeringssproget og af omstændighederne, så du må kigge i dokumentationen eller ... endnu bedre ... lave dig et lille kode eksempel som tjekker efter.

En anden (besværlig) måde man kan skrive \d på er sådan: [0123456789]. Hvorfor? Jo, fordi at:

(o) [abc] : matcher et vilkårligt "tegn som enten er et 'a' eller et 'b' eller et 'c'".
(o) [^abc] : matcher er vilkårligt "tegn som hverken er et 'a' eller et 'b' eller et 'c'".

[...] er altså en slags "OR operator" på enkelttegns niveau. Og ved at inkludere et '^' tegn lige efter den indledende '[' får det den modsatte betydning.

Nogen gange er det smart at bruge et '-' tegn til at forkorte med:

(o) [a-z] : matcher et vilkårligt af tegnene "'a' til 'z'".
(o) [a-zæøå] : matcher et vilkårligt af tegnene "'a' til 'z' eller 'æ' eller 'ø' eller 'å'".

(o) \d = [0123456789] = [0-9]
(o) \D = [^0123456789] = [^0-9]
(o) \w = [a-zA-Z0-9_]
(o) \W = [^a-zA-Z0-9_]

Specieltilfældene ']' og '-': Disse to tegn har en speciel funktion når de er inde i en [...]. De skal derfor placeres med omhu, hvis de bare skal være "sig selv":

(o) []abc] : matcher et vilkårligt af "tegnene ']' eller 'a' eller 'b' eller 'c'".
(o) [abc-] : matcher et vilkårligt af "tegnene 'a' eller 'b' eller 'c' eller '-'".

Mere "OR"

Som nævnt fungerer [...] som en OR på enkelttegns niveau. Der er ofte brug for at kunne lave noget

tilsvarende på hele ord:

(o) abe|hund|gnu : matcher "et af ordene 'abe' eller 'hund' eller 'gnu'".
(o) (han|hun)kat : matcher "et af ordene 'hankat' eller 'hunkat'".

Uden parentesen, ville den sidste have matchet "et af ordene 'han' eller 'hunkat'" - hvilket ikke er helt det samme. For resten kan den sidste også skrives på denne alternative måde:

(o) h[au]nkat : matcher "et 'h' efterfulgt at et af tegnene 'a' eller 'u' efterfulgt af endelsen 'nkat'".

Grupper

Parenteserne fra før har en ekstra effekt: de danner en *gruppe* som *fanger* (engelsk: *capture*) det er matches imellem (...). Denne gruppe kan man efterfølgende gøre ting og sager med.

```
<?
$htmlKode = "Lad os lege lidt med BB kode og gøre dette til kursiv.";

$pattern = "\[i\](.*)\[/i\]";
$phpPattern = "#$pattern#";

$htmlKode = preg_replace($phpPattern, "<i >$1[/i]", $htmlKode);
echo $htmlKode;
?>
```

Resultat: Lad os lege lidt med BB kode og gøre <i >dette[/i] til kursiv.

Der er flere ting at lægge mærke til her:

(o) Da '[' og ']' blot skal matche sig selv og ikke have deres normale funktion, skal de escapes.
(o) Dernæst er der (...) parret. De er placeret rundt om en ".*" som altså matcher et et vilkårligt antal tegn.
(o) Mønsteret \[i\].*\[/i\] vil derfor matche "teksten *efterfulgt at et vilkårligt antal tegn efterfulgt af teksten* ". I eksemplet vil den derfor matche teksten "*dette*".
(o) Men så indsættes en parentes: \[i\](.*)\[/i\]. Dette ændre ikke på hvad der matches, men (...) fortæller regexp-motoren at vi ønsker at huske på det der lå imellem *og* - altså ordet "dette" i dette tilfælde.
(o) Det huskes under navnet $1, og derfor er effekten af preg_replace() ovenfor at "*dette*" erstattes med "<i >$1[/i]" dvs. "<i >dette[/i]".

Endnu et eksempel:

```
<?
$htmlKode = "Al henvendelse skal ske til Eksperten, på forhånd tak!";

$pattern = "\[url=(.*)\](.*)\[/url\]";
$phpPattern = "#$pattern#";

$htmlKode = preg_replace($phpPattern, "<a href='$1'>$2</a>", $htmlKode);
echo $htmlKode;
?>
```

Resultat: Al henvendelse skal ske til Eksperten, på forhånd tak!

Det nye er her at der er to sæt parenteser: Det den 1. gruppe fanger bliver gemt som $1, og det den 2. gruppe fanger bliver gemt som $2.

To af dødsynderne: grådighed og dovenskab

Samme eksempel som før, men med lidt andre data:

```
<?
$htmlKode = "Mere BB kode: *dette* og *dette* og *også dette* gøres til kursiv.";

$pattern = "\[i\](.*)\[/i\]";
$phpPattern = "#$pattern#";

$htmlKode = preg_replace($phpPattern, "<i >$1[/i]", $htmlKode);
echo $htmlKode;
?>
```

Resultat: Mere BB kode: <i >dette[/i] og *dette* og *også dette* gøres til kursiv.

Hvorfor nu det? Det er kun den første *og den allersidste* som erstattes...

Regexp siges at være *grådig* (engelsk: *greedy*)... Når ".*" delmønsteret begynder at matche, vil det fortsætte med at matche så meget som det overhovedet kan få lov til. Det matcher derfor forbi alle de *og* som ligger imellem de to yderpunkter.

Med en lille variation kan den gøres *doven* (engelsk: *lazy*):

```
<?
$htmlKode = "Mere BB kode: *dette* og *dette* og *også dette* gøres til kursiv.";

$pattern = "\[i\](.*?)\[/i\]";
$phpPattern = "#$pattern#";

$htmlKode = preg_replace($phpPattern, "<i >$1[/i]", $htmlKode);
echo $htmlKode;
?>
```

Resultat: Mere BB kode: <i >dette[/i] og <i >dette[/i] og <i >også dette[/i] gøres til kursiv.

(o) * : matcher "0 eller flere ..., og så mange den overhovedet kan slippe af med".
(o) *? : matcher "0 eller flere ..., men det mindste antal den er tvunget til".
(o) + : matcher "1 eller flere ..., og så mange den overhovedet kan slippe af med".
(o) +? : matcher "1 eller flere ..., men det mindste antal den er tvunget til".

Ankre

Eksempel:

```
<form method="post">
Indtast tlf. nummer: <input type="text" name="tlf"><br>
<input type="submit" value="Jeg vil gerne ringes op">
</form>

<?
$telefonNummer = $_POST['tlf'];
if (!empty($telefonNummer)) {
    if (preg_match("/\d{8}/", $telefonNummer)) {
        echo "Ok";
    } else {
        echo "Ikke ok";
    }
}
?>
```

Den skriver "Ok" hvis du indtaster et telefonnummer. MEN den skriver også "Ok" hvis nogen indtaster sådan noget som dette:

qa#¤! 1000000009 cav

Det gør den fordi at \d{8} blot tester på at der er noget i strengen som matcher præcist 8 cifre, og det er der jo (også selvom der faktisk er 10 cifre her). Men der er også så meget mere end det. Den korrekte løsning er:

```
    if (preg_match("/^\d{8}$/", $telefonNummer)) {
        echo "Ok";
    } else {
        echo "Ikke ok";
    }
```

Tegnene '^' og '$' kaldes *ankre* (engelsk: *anchor*):

(o) ^ : matcher starten af teksten. F.eks. vil ^a kun "matche et 'a' som står i starten af teksten".
(o) $: matcher slutningen af strengen. F.eks. vil 10$ kun matche "teksten 10 som står i enden af teksten".

Eksterne referencer

Her er et par referencer som jeg klart vil anbefale uanset hvor øvet eller ej man er:

Regular Expresions, The Complete Tutorial
2006
Jan Goyvaerts

ISBN: 1-4116-7760-9

Bogen er groft sagt en bogversion af hans site:

http://www.regular-expressions.info/

Man kan købe bogen via f.eks. Amazon eller, endnu bedre, direkte fra hans website.

.oOo.

Mastering Regular Expressions
O'Reilly
2002
Jeffrey E. F. Friedl
ISBN: 0-596-00289-0

.oOo.

Wrox har denne, og selvom jeg ikke selv har læst den, går man sjældent galt i byen med dem:

Beginning Regular Expressions
Andrew Watt
2005
ISBN: 978-0-7645-7489-4

.oOo.

http://www.phpartikler.dk/artikler/regexp.php
http://regexlib.com/

Efterord

Regulære udtryk er stærkt værktøj, men man skal selvfølgelig passe på med at tro at de dermed kan bruges til næsten hvad som helst: "For en mand med en hammer begynder alt at ligne søm".

Mange gange klarer funktioner som nl2br(), str_replace(), strcmp(), substr() osv. opgaven lige så godt, og endda hurtigere. Omvendt er der nok ikke meget som *ikke* kunne klares uden regulære udtryk, men den resulterende kode ville ofte være gigantisk og hjemsøgt af fejl.

Feedback

7/3/2008 - Til splazz og andre som måtte have undret sig over det samme:

Dette er først og fremmest en artikel om regulære udtryk. Jeg har derfor forsøgt at skelne skarpt imellem hvad der er selve regexp'en og hvad der så kommer ekstra på fordi at det skal gøres i PHP. Det er grunden til at jeg har lavet underlige konstruktioner som:

```
$pattern = "\[url=(.*)\](.*)\[/url\]";  // Den "rene" regexp ...
$phpPattern = "#$pattern#";  // ... sovset ind I PHP
```

De to #-tegn der er brugt her er en del af PHP's syntaks, og hvorfor at det er sådan er historisk betinget (se f.eks. http://www.eksperten.dk/artikler/1160 hvis man vil vide mere om det). Tegnene bruges til at markere starten og slutningen af det regulære udtryk. Der opstår derfor et problem hvis at #-tegnet

også indgår som et tegn i $pattern - for hvor starter og stopper regexp'en så? Ved den første, den sidste eller den midterste '#'?

Problemet er ikke større end at PHP giver en løsning - nemlig at escape det midterste #-tegn:

```
$pattern = "... \# ...";
$phpPattern = "#$pattern#";
```

Jeg foretrækker bare ikke at bruge denne løsning, da det gær regexp'en unødig tung at læse. Jeg foretrækker derfor at bruge #-tegn, men hvis det også er #-tegn inde i regexp'en så leder jeg efter et alternativ - f.eks. at bruge /-tegnet i stedet. Håber at det forklarede det?

BTW: Mange PHP udviklere bruger /-tegnet - muligvis fordi at de ikke ved at der er alternativer. Uheldigvis bruger det det så sammen med tekst - som f.eks. URL's eller filstier - som allerede indeholder /-tegn. Dette giver anledning til "monstre" som:

```
$phpPattern = "/https:\/\/www\.eksperten\.dk\/artikler\/\d+/i";
```

Det var den slags kode som oprindeligt fik mig til at lede efter alternativer. :^)

Kommentar af fjappe d. 05. Dec 2007 | 1

Rigtig god artikel. Den forklarer regexp på en meget let forståelig måde. Jeg har tit manglet en artikel som denne. Helt sikkert er det, at jeg vil vende tilbage og bruge den her artikel som reference når jeg får brug for regexp igen. Well done nielle

Kommentar af frankeman d. 21. Dec 2007 | 2

Fino

Kommentar af jps6kb d. 04. Dec 2007 | 3

Alletiders artikel

Kommentar af cf560 d. 13. Jan 2008 | 4

God

Kommentar af alex15 d. 21. Apr 2008 | 5

Rigtig god og forklarende artiklen. Lige hvad jeg ledte efter.

Kommentar af camirose d. 28. Dec 2007 | 6

Kommentar af amite d. 02. Jan 2008 | 7

glimrende

Kommentar af splazz d. 06. Mar 2008 | 8

fed artikel, men jeg synes ikke jeg kan finde ud af hvorfor du nogle gange bruger:
$phpPattern = "/$pattern/";

og andre gange:
$phpPattern = "#$pattern#";

Kommentar af ddd_dendummedreng (nedlagt brugerprofil) d. 11. Jan 2008 | 9

Det er rigtig, rigtig godt skrevet.

Kommentar af joggy d. 12. Mar 2008 | 10

Glimrende artikel.. Specielt på et sådan emne, som for mange kan virke meget komplex.

Kommentar af bbcdk d. 10. Nov 2008 | 11

Kommentar af el_barto (nedlagt brugerprofil) d. 12. Dec 2007 | 12

Super artikel...måske kan man indskyde at netop nl2br() og de andre funktioner du nævner i de fleste tilfælde er hurtigere end et regulært udtryk.

Det fortæller PHP-manualen da også. Omvendt er der mange ting man KUN kan lave med regulære udtryk :)

Kommentar af mr-kill d. 09. Dec 2007 | 13

Super artikel - mange tak :)

Kommentar af mstorgaard d. 05. Dec 2007 | 14

Rigtig god artikel. Helt klart en artikel jeg vil benytte mig meget af fremover.

Flere af den slags! (:

Kommentar af zhx d. 08. Dec 2007 | 15

Super artikel, det er en artikel jeg vil komme til at bruge i fremtiden, forsæt det gode arbejde ^ høj herfra