Regulære udtryk, 2. del

Artiklen uddyber mange af de begreber, som blev introduceret i den 1. artikel. Den burde dog kunne læses alene, hvis man allerede er bekendt med regexp. Desuden introduceres nye begreber som modifiere, matchning på hele ord, backreferencer og lookaround.

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

Indledning

Regulære udtryk er et emne der kan skrives tykke bøger om, og det er der i øvrigt allerede gjort. Jeg satser derfor heller ikke på at kunne beskrive hver eneste mulighed og faldgruppe. I stedet vil jeg forsøge at beskrive de mest nyttige muligheder i denne lille serie af artikler.

Artikel 1 kom rundt i de helt elementære begreber i regexp. Denne artikel går mere i dybden med eksempler med en del af disse. Desuden introduceres nye begreber fra regexp værktøjskassen.

I artikel 3 viser hvordan at man arbejder med regulære udtryk i nogle forskellige udvalgte programmeringssprog. Her vil jeg dog fortsat holde mig til PHP fordi det er min antagelse at det er der at mange af Ekspertens brugere støder på regexp for første gang.

v. 1.0: 10/12/2007 - Første version.
v. 1.1: 15/12/2007 - Retter op på noget formatering.

Et eksempel - Ostesangen

Antag at man f.eks. ønsker at fremhæve bestemte ord på sit site (guestbook, tagwall, blog, whatever):

```
<?
$lyrik =
"Roquefort og emmentaler,
gorgonzola, jeg betaler
hvad som helst for feta!
Brie eller camembert,
hvad der lugter endnu værre,
gammel ost fra Kreta.
Ost, jeg vil ha' ost
jeg må ha' ost!
Ost, jeg vil ha' ost
jeg må ha' ost!
ost, ost, ost, ost, ost
jeg må ha' o-hooost!";  // (c) 2004 Sebastian

$lyrik = preg_replace("/(ost)/i", "$1", $lyrik);

echo $lyrik;
?>
```

Resultat:

Roquefort og emmentaler,
gorgonzola, jeg betaler
hvad som helst for feta!
Brie eller camembert,
hvad der lugter endnu værre,
gammel ost fra Kreta.
Ost, jeg vil ha' ost
jeg må ha' ost!
Ost, jeg vil ha' ost
jeg må ha' ost!
ost, ost, ost, ost, ost
jeg må ha' o-hooost!

Forklaring:

(o) ost : matcher simpelthen ordet "ost", som er det ord vi ønsker at fremhæve.
(o) (...) : parenteserne danner en gruppe, og fanger derfor ordet "ost" (eller rettere sagt positionen af ordet) ...
(o) ... i variablen $1. Denne erstattes med $1 altså med ost i dette tilfælde.
(o) Replace'n slår igennem alle de steder hvor at ordet optræder; ikke kun det 1. sted.

Match-modifiere

Der er desuden en enkelt ingrediens mere:

(o) Det lille 'i' i "/.../i" kaldes en *modifier* til regexp'en. 'i' står for *ignore case* eller *case insensitive* lidt efter hvem man spørger. Det betyder under alle omstændigheder at store og små bogstaver skal behandles ens.

Vi ser også lidt af begrundelsen for at der som regel er en '/' i hver ende af en regexp i PHP; det er for at adskille selve regexp'en fra modifierne. '/' tegnene kommer dog ultimativ af at preg_xxxx() funktionerne stammer fra Perl, men det er en historie til en kommende artikel.

Uden adgang til i-modifieren skulle vi have skrevet noget i stil med denne grimmert:

```
$lyrik = preg_replace("/([oO][sS][tT])/", "$1", $lyrik);
```

I dette her tilfælde kunne den dog sagtens se en smule "pænere" ud fordi at det kun er varianten "Ost", i starten af en sætning, der ellers ville volde problemer:

```
$lyrik = preg_replace("/([oO]st)/i", "$1", $lyrik);
```

Eller endnu et par varianter:

```
```

```
$lyrik = preg_replace("/(o|O)st/", "$1", $lyrik);

// og:

$lyrik = preg_replace("/(Ost|ost)/", "$1", $lyrik);
```

Der er andre modifierer, og de kan endda kombineres:

(o) /.../i : Store og små bogstaver behandles som det samme bogstav.
(o) /.../m : En variabel kan indeholde flere linjer tekst. Med m-modifieren matcher '^' og '$' på hver linje i variablen i stedet for kun at matche på start og slut af det hele.
(o) /.../s : Normalt vil '.' matche på et vilkårligt tegn ... med undtagelse af linjeskift (tegnene \r og \n). Med s-modifieren matcher den også på disse
(o) /.../is : kombinere effekten af i- og s-modifierne.

Du kan læse mere om de andre muligheder her:

http://dk2.php.net/manual/da/reference.pcre.pattern.modifiers.php

Søgning på hele ord

Men hov, der gik jo noget galt i:

"... jeg må ha' o-hooost!"

Her indgår ordet "ost" i et større ord (teknisk set er det vel ikke et ord, men for regexp er alle katte nu grå ;^)

Hvis vi kun ønsker at erstatte hele ord kan vi bruge:

(o) \b : matcher på "starten eller slutningen af et ord".
(o) \B : matcher på noget som "ikke er hverken start eller slutning på et ord".

Koden ser sådan her ud:

```
<?
$lyrik = "... ost, ost, ost, ost, ost
jeg må ha' o-hooost!";  // (c) Sebastian

$lyrik = preg_replace("/\b(ost)\b/i", "$1", $lyrik);

echo $lyrik;
?>
```

Resultatet:

... ost, ost, ost, ost, ost
jeg må ha' o-hooost!

På denne måde minder \b lidt om '^' og '$', men hvor disse matcher på hhv. starten og slutningen af en sætning, vil \b matche på start eller slutning af et ord.

Et mere komplekst eksempel - HTML tags

Regex'er kan hurtigt blive komplekse. Lad os kigge på følgende kode:

```
<?
$htmlCode = "Sådan noget som <em>1000 > 1</em> er jo trivielt sandt!";

$pattern = "<([^>]+)>(.*?)</[^>]+>";
$phpPattern = "#$pattern#";

$htmlCode = preg_replace($phpPattern, "[$1]$2[/$1]", $htmlCode);

echo $htmlCode;
?>
```

Resultat: Sådan noget som [em]1000 > 1[/em] er jo trivielt sandt!

Regexp'en er denne her:

<([^>]+)>(.*?)</[^>]+>

og den kan til at starte med deles op i 3 del-mønstre:

<([^>]+)> ... (.*?) ... </[^>]+>

Del 1:

(o) < : matcher simpelthen "tegnet '<'".
(o) [^...] : matcher "et vilkårligt tegn som ikke er et af tegnene ...".
(o) [^>] : matcher derfor "et vilkårligt tegn som ikke er et '>'". Dette er en konstruktion som man ofte se brugt når regex' bliver anvendt imod HTML tags.
(o) + : matcher "1 eller flere ...".
(o) [^>]+ : matcher "en eller flere tegn som ikke er et '>'".
(o) (...) : danner en gruppe, som fanger det der bliver matchet imellem parenteserne. I dette tilfælde fanger de altså det der matches af [^>]+.
(o) > : matcher simpelthen "tegnet '>'".

I alt:

(o) <([^>]+)> : matcher "et '<', fulgt af et eller flere tegn som ikke er et '>', og så til sidst et '>'. Det der ligger imellem <...> bliver endvidere fanget af gruppen og bliver husket til senere som $1 (da det jo er den 1. parentes).".

Del 2:

(o) . : matcher "et vilkårligt tegn".
(o) * : matcher "0 eller flere ...".

(o) .* : matcher "0 eller flere vilkårlige tegn".
(o) ? : ændrer matchningen fra at være grådig til at være doven.
(o) .*?: matcher "0 eller flere vilkårlige tegn, men så få den kan slippe af sted med". Hvis matchningen kan slippe med at matche 0 tegn i alt ville den være "glad". Det kan den bare ikke i dette her tilfælde, men mere om det om lidt.
(o) (...) : danner en *gruppe*, som *fanger* det der matches af ".*?". Da det er den 2. gruppe bliver det matchede gemt som $2.

Del 3

... ligner del 1 til forveksling. Blot er der et '/' mere som skal matches, og der er ikke nogen gruppe som fanger noget.

Alt det ovenstående kan sammenfattes sådan.

"En regex som matcher en HTML tag og dens slut tag samt alt det der er ind imellem. Del 1 matcher start taggen, del 2 matcher indholdet og del 3 matcher slut taggen. Hvis ikke det havde været fordi at regexp'en som helhed skulle matche ville del slet ikke have matchet noget (den er *lazy*), men del 3 tvinger den til at matche indholdet mellem de to tags (men så heller ikke mere)."

Anvendt på teksten:

"Sådan noget som 1000 > 1 er jo trivielt sandt!"

vil del 1 matche "" og "em" vil blive husket som $1; del 2 vil matche "1000 > 1" og det vil blive gemt som $1; del 3 vil matche "".

Læg for resten mærke til at selv om der står et '>' imellem de to tags, så giver det ikke problemer.

Backreference - Grupper og \1, \2, \3, ...

Samme kode som ovenfor, men med noget andet data:

```
<?
$htmlCode = "Lad os fremhæve ordet <b >regexp[/b] på nogle forskellige
måder.";

$pattern = "<([^>]+)>(.*?)</[^>]+>";
$phpPattern = "#$pattern#";

$htmlCode = preg_replace($phpPattern, "[$1]$2[/$1]", $htmlCode);

echo $htmlCode;
?>
```

Resultat: Lad os fremhæve ordet **regexp**[/b] på nogle forskellige måder.

Ikke helt ideelt, for at sige det mildt. Hvad sker der?

(o) <([^>]+)> : matcher stadigvæk "< b>" og taggens navn, "b", gemmes til senere som $1

(o) (.*?) : matcher "et vilkårligt tegn, men så få det kan slippe af med. Vi kan ikke sige hvad det matcher lige nu, for det afhænger af næste led i regexp'en. Men det matchede gemmes i hvert fald som $2.
(o) </[^>]+> : matcher også stadigvæk "et '<', efterfulgt af et '/', efterfulgt af 1 eller flere tegn som ikke er et '>', efterfulgt af et '>'".
(o) Pga. '?' tegnet i "(.*?)" er denne lazy. Dvs. at "</[^>]+>" allerede matcher ""**i stedet for at gå videre til at matche "[/b]" som var det vi egentlig håbede på.**
(o) Nu kan vi så konkludere at delleddet fra punkt 2 matcher på "regexp", og at det er denne værdi som gemmes i $2.

Resultatet er at regexp'en matcher på dette:

<b >[u]regexp

og at det bliver erstattet med dette:

regexp

Indsat i den længere tekststreng bliver det det resultat vi så.

Hvad er kuren? Vi kunne droppe '?' tegnet. Dette ville løse problemet. Prøv selv efter...

Imidlertid var det der af en god grund. Hvis det ikke var der ville den nemlig fejle når den samme tag bliver gentaget flere gange i den tekst der arbejdes med. Prøv selv efter med:

$htmlCode = "Mere BB kode: *dette* og *dette* og *også dette* gøres til kursiv.";

(dette eksempel blev forklaret i detaljer i den 1. artikel)

Så hvad *er* kuren? Vi ønsker at kunne matche så lidt som muligt (lazy) men vi ønsker at kunne tvinge matchningen til at matche henover andre HTML tags så længe de ikke er slut taggen for den HTML tag som vi starter matchningen fra.

Svaret er *backreference* (på dansk ville det "mundrette" ord være noget i stil med *tilbagereference*):

```
<?
$htmlCode = "Lad os fremhæve ordet <b >[u]regexp[/b] på nogle forskellige
måder.";

$pattern = '<([^>]+)>(.*?)</\1>';
$phpPattern = "#$pattern#";

$htmlCode = preg_replace($phpPattern, "[$1]$2[/$1]", $htmlCode);

echo $htmlCode;
?>
```

Resultat: Lad os fremhæve ordet **regexp** på nogle forskellige måder.

Forklaring:

(o) \1 : matcher "det den 1. gruppe fangede".

Eftersom den 1. del af regexp'en matchede "<b >" så er det "b" som er gemt i $1. \1 referere tilbage til det der blev matchet og gemt som $1. Dvs. at det sidste led lige så godt kunne være skrevet som "[/b]" og derfor så matcher det ikke mere på "[/i]" selvom den møder den først. Det er en helt generel feature:

(o) \2 : matcher "det den 2. gruppe fangede".
(o) \3 : matcher "det den 3. gruppe fangede".
(o) osv. osv.

Et par PHP trick, nu vi er ved det

For resten valgte jeg at bruge '-tegn, i stedet for "-tegn, rundt om $pattern i koden ovenfor. Det gjorde jeg fordi at PHP ellers selv ville have forlangt at '\' skulle escapes:

```
$pattern = "<([^>]+)>(.*?)</\\1>";
```

Det er vigtigt at være opmærksom på at der her er to escapes foran '1', og at den ene skal med fordi at det er sådan at backreference virker, og at den anden skal med fordi at '\' tegn i en "-streng skal escapes hvis de bare skal fortolkes som et '\' tegn.

... og et andet PHP trick. Det er mest almindeligt at bruge '/' tegn i hver ende af et PHP preg_xxxx() pattern. Hvis man gør det skal alle '/' inde i mønsteret imidlertid også escapes:

```
$phpPattern = "/<([^>]+)>(.*?)<\/\\1>/";
```

Jo mere der skal escapes på denne måde, jo svære bliver det at læse regexp'en:

```
$phpPattern2 = "/http:\/\/www\.eksperten\.dk\/artikler\/(\d+)/";
```

PHP tillader imidlertid at man bruger andre tegn, og jeg har altså valgt at bruge '#' tegn. Dette er kun et problem, hvis man har '#' tegn i selve mønsteret for så skal de selvfølgelig escapes.

Ikke-fangende grupper

(...) bruges til grupper og det matchede fanges (engelsk: *capture*) som hhv. $1, $2, $3, osv.

(...) bruges imidlertid også i andre sammenhænge og der ønsker man ikke nødvendigvis at fange det matchede:

(o) (han|hun)kat : matcher "et af ordene 'han' eller 'hun' efterfulgt af ordet 'kat'".
(o) \d{2}(:\d{2}){2} : matcher "præcis 2 cifre, efterfulgt af et ':' og 2 cifre - præcis 2 gange". En anden måde at sige dette på er at den f.eks. matcher et klokkeslæt (hh:mm:ss).

I det 1. tilfælde bruges (...)'en rundt om en '|' for at adskille de to muligheder i OR'en fra resten af mønsteret:

(o) han|hunkat : ... matcher noget helt andet...

I det 2. tilfælde bruges den til at markere at der er et del-mønster som skal gentages præcist to gange.

I ingen af de viste tilfælde ønsker vi nødvendigvis at bruge $1, $2, $3 senere. Det bliver endnu sjovere hvis mønsteret også indeholder et par (...) som faktisk skal fange. Det er for så vidt ikke så svært at have med dette at gøre. Man kan jo blot ignorere dem og tælle frem til de (...) man er interesseret i:

```
<?
$guid = "{3F2504E0-4F89-11D3-9A0C-0305E82C3301}";

$pattern = "[0-9A-F]{8}(-[0-9A-F]{4}){3}-([0-9A-F]{12})";
$phpPattern = "#$pattern#i";  // Hvorfor mon der er et i her?

$guidPart = preg_replace($phpPattern, "$2", $guid);

echo $guidPart;
?>
```

Resultat: {0305E82C3301}

Man kan som sagt gøre det på den måde, men nogen gange ønsker man kun at fange fra de (...) par som man faktisk har til hensigt at bruge senere.

```
<?
$guid = "{3F2504E0-4F89-11D3-9A0C-0305E82C3301}";

$pattern = "[0-9A-F]{8}(?:-[0-9A-F]{4}){3}-([0-9A-F]{12})";  // Ændret her...
$phpPattern = "#$pattern#i";

$guidPart = preg_replace($phpPattern, "$1", $guid);  // ...og her

echo $guidPart;
?>
```

Resultat: ... det samme ...

(o) (?:...) : fungere på samme måde som (...), men fanger ikke det matchede som $x.

Man kalder det for en *non-capture gruppe* - en *ikke-fangende gruppe*.

Negative Lookahead

Lad os antage at vi ønsker at kunne matche teksten "abc" ... men kun hvis den ikke er efterfulgt af et 'd'.

Det kunne f.eks. se sådan ud:

```
<?
function testAbc($tekst) {
    echo "Tekst: '$tekst'; ";

    $pattern = "abc[^d]";
    $phpPattern = "#$pattern#";

    if (preg_match($phpPattern, $tekst)) {
        echo "Test: Ok<br>";
    } else {
        echo "Test: Ikke ok<br>";
    }
}

testAbc("... abc ...");
testAbc("... abcq ...");
testAbc("... abcd ...");
?>
```

Resultat:

Tekst: '... abc ...'; Test: Ok
Tekst: '... abcq ...'; Test: Ok
Tekst: '... abcd ...'; Test: Ikke ok

Umiddelbart ser det ud til at være i orden, men det er det ikke. Den fejler for denne:

```
testAbc("... abc");
```

Hvorfor?

(o) abc[^d] : matcher "ordet 'abc', efterfulgt af et tegn som ikke er et 'd'".

Det er ikke helt det samme som det vi var ude efter, nemlig noget som matchede "ordet 'abc' ikke efterfulgt af 'd'".

En mulig løsning er denne:

```
    $pattern = "abc([^d]|$)";
```

(o) abc([^d]|$) : matcher "ordet 'abc', enten efterfulgt af et tegn som ikke er et 'd' eller af slutningen af teksten".

Den løser bare ikke den næste variation: Vi ønsker at matche "ordet 'abc' ... men kun hvis det ikke er efterfulgt af ordet 'def'". F.eks. fejler dette:

```
<?
function testAbc2($tekst) {
    echo "Tekst: '$tekst'; ";

    $pattern = "abc([^def]|$)";
    $phpPattern = "#$pattern#";

    if (preg_match($phpPattern, $tekst)) {
        echo "Test: Ok<br>";
    } else {
        echo "Test: Ikke ok<br>";
    }
}

testAbc2("1 ... abc ...");
testAbc2("2 ... abcdef ...");
testAbc2("3 ... abcde ...");
testAbc2("4 ... abcqqf ...");
testAbc2("5 ... abcfqq ...");
testAbc2("6 ... abc");
?>
```

Resultat:

Tekst: '1 ... abc ...'; Test: Ok
Tekst: '2 ... abcdef ...'; Test: Ikke ok
Tekst: '3 ... abcde ...'; Test: Ikke ok
Tekst: '4 ... abcqqf ...'; Test: Ok
Tekst: '5 ... abcfqq ...'; Test: Ikke ok
Tekst: '6 ... abc'; Test: Ok

Det er en almindelig fælde at tro at [^def] matcher noget som "ikke er ordet 'def'". Det er ikke rigtigt. I stedet matcher den "et tegn som hverken er et 'd' eller et 'e' eller et 'f'". Det er noget helt andet:

Tilfælde 2) Rigtigt resultat, men forkert begrundelse. Den tester udelukkende på tegnet lige efter "abc".
Tilfælde 3) Da den kun kigger på 'd' så tjekker den ikke om der står "def" eller blot "de".
Tilfælde 5) 'f' er et af de tegn der kigges efter i del-mønsteret "[^def]".
Tilfælde 4) Der er godt nok et 'f' her, men det kommer ikke lige efter "abc".

(o) [^...] : matcher kun på enkelttegn-niveau.

For at klare vores problem har vi brug for en løsning på helords-niveau. Løsningen kaldes for *negative lookahead* (*negativt se-frem* hvis man skulle find en mulig dansk oversættelse):

```
    $pattern = "abc(?!def)";
```

(o) xxx(?!yyy) : matcher "det der står på pladsen xxx, hvis og kun hvis det ikke efterfølges af det der står på pladsen yyy".

Det forrige problem med "abcd" kan derfor også løses med:

```
    $pattern = "abc(?!d)";
```

Det behøver ikke kun at være tekst; der kan også stå regexp'er i stedet for xxx og yyy

Lookaround

Negative lookahead er medlem af en lille familie at lignende funktioner:

(o) xxx(?=yyy) : *Positive Lookahead*. Matcher "det der står på pladsen xxx, hvis og kun hvis det efterfølges af det der står på pladsen yyy".
(o) xxx(?!yyy) : *Negative Lookahead*. Matcher "det der står på pladsen xxx, hvis og kun hvis det ikke efterfølges af det der står på pladsen yyy".
(o) (?<=yyy)xxx : *Positive Lookbehind* (dansk: *positiv kig-tilbage*). Matcher "det der står på pladsen xxx, hvis og kun hvis det der står på pladsen yyy står foran det".
(o) (?<!yyy)xxx : *Negative Lookbehind*. Matcher "det der står på pladsen xxx, hvis og kun hvis det der står på pladen yyy ikke står foran".

Eksempel:

```
<?
function beautifyCode($htmlCode) {
    // Sætter "http://" foran de urls der mangler
    $pattern = "(?<!http://)\b(\w+\.){2,}[a-z]{2,}";  // Negative lookbehind
samt \b
    $phpPattern = "#$pattern#i";
    $htmlCode = preg_replace($phpPattern, "http://$0", $htmlCode);

    // Laver alle urls om til link
    $pattern = "http://(\w+\.){2,}[a-z]{2,}";
    $phpPattern = "#$pattern#i";
    $htmlCode = preg_replace($phpPattern, "<a href='$0'>$0</a>", $htmlCode);

    return $htmlCode;
}

echo beautifyCode("... www.eksperten.dk ...") . "<br>";
echo beautifyCode("... http://www.eksperten.dk ...") . "<br>";
?>
```

Resultat:

... http://www.eksperten.dk ...

... http://www.eksperten.dk ...

Prøv selv om du kan greje den. :^)

Kommentar af cf560 d. 13. jan 2008 | 1

God

Kommentar af masik7 d. 16. nov 2008 | 2

God