

Les expressions rationnelles

Dans le chapitre précédent, nous avons utilisé des expressions rationnelles pour créer le nom de la table dans le programme [mini-forum.php](#).

Dans l'expression suivante, nous avons extrait le nom du fichier du chemin que nous avons écourté de son extension :

```
preg_match("#^/.*/(.*)\.[A-z0-9]{3,4}$#", $PHP_SELF, $tableau);
```

Ces signes vous paraissent sans doute des hiéroglyphes mais j'espère qu'à la fin de ce chapitre, vous y verrez plus clair dans ce langage particulier appelé expressions rationnelles ou régulières. Nous utiliserons l'adjectif "rationnelle" pour caractériser ces expressions car elles sont plutôt irrégulières dans le sens qu'elles ne régulent rien et qu'elles sont utilisées différemment pour chaque cas particulier. En fait "régulière" a été traduit de regular qui dans l'expression "regular expression" signifie "expression consacrée". Encore une de ces expressions traduites dans l'à-peu-près qui deviennent courantes mais dont le sens finit par devenir obscur. Nous emploierons donc "expressions rationnelles" car ces expressions permettent une recherche générique ou abstraite sur un certain nombre de caractères ou de groupes de caractères. Le type de caractère et sa place sont traduits sous forme de caractères de description. Nous avons donc traduit, sous forme abstraite, un mot ou une expression recherchée. Ce langage est puissant mais aussi très complexe. Nous n'en ferons pas le tour dans ce livre, nous vous aiderons seulement à en comprendre certains principes et nous vous en proposerons les expressions les plus courantes.

La méthode

Vu le comportement très complexe des moteurs d'expressions rationnelles, l'approche ne peut être qu'empirique. Avec un peu d'habitude, la solution est trouvée plus rapidement. Dans chaque recherche, vous avez un objectif. Vous voulez vérifier, extraire ou remplacer une chaîne de caractères, c'est le centre de votre stratégie. Il faut faire en sorte d'aider le moteur à trouver des repères autour de cet objectif. Si vous recherchez [www](#), vous savez qu'il est entouré de [http://](#) et d'un point. Plus vous caractérisez votre objectif, plus vous aurez de chances de le trouver. Le moteur a quelques "défauts". Ainsi il ne s'arrêtera pas au premier motif trouvé mais ira jusqu'à la fin de la cible. C'est ce qu'on appelle l'avidité, elle peut pousser le moteur à être trop perfectionniste dans sa recherche et à ne pas trouver ce qu'il a "sous les yeux" car il cherche toujours un motif complémentaire.

signe	description
^	Début de ligne sauf quand il se trouve à l'intérieur de crochets, auquel cas il signifie une négation.
\$	Fin de ligne
Début et fin de ligne	

Les parenthèses permettent de délimiter des sous-motifs qui seront stockés dans un tableau dans le cas d'une recherche ou dans les variables spéciales `\\0` pour la totalité du motif, `\\1` pour la première parenthèse...

En ce qui concerne les expressions rationnelles, PHP a hérité du module du langage Perl et des normes POSIX. Nous allons étudier un cas assez simple pour commencer puis nous verrons la syntaxe et nous approfondirons. Auparavant, nous allons faire le panorama des fonctions du langage PHP qui utilisent les expressions rationnelles

POSIX

POSIX est une norme qui permet la portabilité sur tous les systèmes

Voici les fonctions du langage PHP aux normes POSIX :

Fonction	Description
<code>int ereg (string motif, string chaîne [, array tableau])</code>	Retourne vrai si la chaîne contient le motif.
<code>string ereg_replace (string motif, string nouveau_motif, string chaîne)</code>	Remplace le motif à l'intérieur de la chaîne par le nouveau motif.
<code>eregi()</code>	Comme <code>ereg()</code> mais insensible à la casse
<code>eregi_replace()</code>	Comme <code>ereg_replace()</code> mais insensible à la casse
<code>array split (string motif_séparateur, string chaîne [, int limite])</code>	Découpe la chaîne en sous-chaînes au moyen du séparateur.
<code>string sql_regcase (string chaîne)</code>	Crée une expression rationnelle insensible à la casse à partir de la chaîne, ex.: pour php, le résultat sera <code>[Pp][Hh][Pp]</code>
Fonctions pour des expressions rationnelles POSIX	

Quelques études de cas

Vous voulez par exemple savoir si vous avez des chiffres dans un texte

```
if ereg('[0-9'],$table){
}
```

Vous voulez par exemple savoir si vous avez des chiffres dans un texte

```
$expression="fs5dqfdsf";
if (ereg('[0-9'],$expression)){
}
```

l'expression est vraie car elle contient au moins un chiffre. Ici `[0-9]` représente la classe chiffres qui pourrait aussi être exprimée avec l'expression `[:digit:]`. Le signe "-" représente l'intervalle. Il s'agit ici de tous les chiffres dans l'intervalle de 0 à 9 inclus.

Si vous voulez rechercher un chiffre particulier :

`ereg('0',$expression)` recherche le chiffre 0

`ereg('[09'],$expression)` recherche les chiffres 0 ou 9 dans l'expression.

Si vous voulez que ces chiffres ne soient pas dans l'expression, employez le caractère `^` dans les crochets. Si vous l'employez en dehors des crochets, il n'a plus même sens.

`!ereg('[^0'],$expression)` sera vrai si le chiffre 0 n'est pas dans l'expression.

Rappelez-vous

```
if !ereg('[^0-9'],$table){
$table.='_';
}
```

Ici nous vérifions que le nom de la table créé à partir d'un nom de fichier ne contenait pas que des chiffres. Si c'était le cas, nous ajoutons un blanc souligné.

```
$contenu_txt =ereg_replace("([[:space:]]|^)[A-z0-9]{1,2}[[:space:]]", " ", $contenu_txt);
```

Dans cette expression, nous cherchons un mot (donc entouré d'espaces) qui contienne de 1 à 2 lettres. `([[:space:]]|^)` signifie qu'il peut être en début de chaîne ou après un espace.

Il reste un caractère que nous n'avons pas vu, c'est le caractère point (.). Il remplace n'importe quel nombre et type de caractère. Si vous voulez représenter un vrai point, il vous faudra utiliser un caractère d'échappement (\.).

Voici les expressions POSIX et leurs équivalents

séquence	équivalent	description
<code>[[:alnum:]]</code>	<code>[A-Za-z0-9]</code>	caractères alphanumériques
<code>[[:alpha:]]</code>	<code>[A-Za-z]</code>	caractères alphabétiques
<code>[[:digit:]]</code>	<code>[0-9]</code>	caractères numériques
<code>[[:blank:]]</code>	<code>[\x09]</code>	espaces ou tabulations
<code>[[:lower:]]</code>	<code>[a-z]</code>	caractères en bas de casse
<code>[[:upper:]]</code>	<code>[A-Z]</code>	caractères en capitales
<code>[[:xdigit:]]</code>	<code>[0-9a-fA-F]</code>	caractères hexadécimaux
<code>[[:punct:]]</code>	<code>[!-/:-@[-`{--~]</code>	caractères de ponctuation
<code>[[:space:]]</code>	<code>[\t\v\f]</code>	tout caractère d'espace
<code>[[:cntrl:]]</code>	<code>[\x00-\x19\x7F]</code>	caractères de contrôle
<code>[[:graph:]]</code>	<code>[!--]</code>	caractères affichables et imprimables
<code>[[:print:]]</code>	<code>[--~]</code>	caractères imprimables sauf caractères de contrôle

Les normes POSIX

Les caractères sont dans l'ordre du tableau de caractères ASCII. Ainsi dans `[a-z]` ne figure pas les lettres accentuées.

Remarque Les lettres accentuées

Pour ce cas de lettres accentuées vous devrez utiliser les caractères `\x` suivi du numéro dans la table ASCII. Ainsi pour trouver si votre texte contient des lettres accentuées françaises, vous devez écrire `[x128-x151\153\154]` c'est-à-dire les caractères 128 à 151 puis les 153 et 154. De la même façon si vous voulez caractère littéral `^`, il vous faut utiliser `\x94`.

Pour affiner notre recherche, nous avons à notre disposition ce qu'on appelle la cardinalité, c'est-à-dire que nous pouvons sélectionner le nombre de caractères

`ereg('[0-9]{1}', $expression)` recherche s'il existe 1 chiffre dans l'expression.

`ereg('[0-9]{1,3}', $expression)` recherche s'il existe entre 1 et 3 chiffres dans l'expression.

Nous avons aussi des équivalents

signe	équivalent	description
*	<code>{0,}</code>	0 ou plus
+	<code>{1,}</code>	1 ou plus

?	{0,1}	0 ou 1
Cardinalité		

`ereg('[0-9]+', $expression)` recherche s'il existe au moins 1 chiffre dans l'expression.

En voyant le tableau précédent, vous pouvez vous interroger sur l'utilité de l'étoile `*`. Ce signe renvoie vrai si le caractère existe ou n'existe pas. En fait, il s'emploie dans des expressions plus complexes.

`ereg('[0-9]*', $expression)` recherche s'il existe au moins 0 chiffre dans l'expression. Vous pouvez par exemple rechercher un mot qui peut être au pluriel ou au singulier :

`!ereg('régulières*', $expression)` sera vrai si le mot "régulière" ou "régulières" n'existe pas dans l'expression ;=).

Si vous recherchez le mot événement qui peut aussi s'écrire évènement, vous emploieriez l'expression `ereg('évé|ènement*', $expression)`. Le signe `|` signifie ou. Cette expression est équivalente à `ereg('év[éè]nement*', $expression)`.

Vous pouvez aussi faire porter la précision sur plusieurs caractères. Ainsi vous cherchez un mot masculin ou féminin comme docteur ou docteresse `'docteu*r(esse)*'`. Le problème ici c'est que docteresse fonctionne aussi. `'docteu*r'` sera mieux adapté. Nous verrons un peu plus loin ces problèmes d'avidité qui font que dès que l'une des expressions est trouvée, le moteur retourne vrai sans aller jusqu'à la fin de l'expression.

Nous allons reprendre les fonctions `date_nombre()` et `nombre_date()` avec les expressions rationnelles. Le code sera plus court et l'exécution plus rapide.

```
<?
function nombre_date($cette-date){
    $ladate=$cette-date;
    $longueur=strlen($ladate);
    if ($longueur==8){
        $ladate=ereg_replace("^[0-9]{4})([0-9]{2})([0-9]{2})", "\\3/\\2/\\1",
        $ladate);
    }
    elseif ($longueur==12){
        $ladate=ereg_replace("^[0-9]{4})([0-9]{2})([0-9]{2})([0-9]{2})", "\\3/\\2/\\1
        \\4h\\5", $ladate);
    }
    elseif ($longueur==14){
        $ladate=ereg_replace("^[0-9]{4})([0-9]{2})([0-9]{2})([0-9]{2})([0-
        9]{2})", "\\3/\\2/\\1 \\4h\\5mn\\6", $ladate);
    }
    if ($longueur>8){
        $heure= (int)substr($ladate,8,2);
        $minutes= substr($ladate,10,2);
        $ladate.=" ${heure}h ${minutes}mn";
        if ($longueur>12){
            $secondes= substr($ladate,12,2);
            $ladate.=" $secondes";
        }
    }
    return $ladate;
}
```

Nous vérifions, dans cette fonction la longueur de la date et, selon la taille, nous en extrayons aussi l'heure, les minutes et les secondes. Les jours, mois et heure sont forcés en `int` (typées) car nous voulons éviter qu'ils commencent par un 0.

```
<?
function date_nombre($cette-date){
    $ladate=$cette-date;
    $ladate=explode('/', $ladate);
    $jour=$ladate[0];
    $mois=$ladate[1];
```

```

$an=$ladate[2];
$longueur=strlen($an);
if ($longueur==6){
if ($an>date("Y")){
$an='19'.$an;
}
else {
$an='20'.$an;
}
}
$longueur=strlen($mois);
if ($longueur==1){
$mois='0'.$mois;
}
$ladate=$an.$mois.$jour;
return $ladate;
}
?>

```

Nous voyons cette même recherche de motif à la manière de Perl dans les pages qui suivent

Les fonctions compatibles Perl

L'intérêt majeur de ces fonctions compatibles avec le langage Perl est que vous pouvez utiliser des options qui sont parfois indispensables ou, pour le moins, utiles

Fonction	Description
<code>array preg_grep (string motif, array tableau)</code>	Retourne dans un tableau les éléments extraits d'un autre tableau qui correspondent au motif
<code>int preg_match (string motif, string chaîne [, array tableau])</code>	Retourne vrai si la chaîne contient le motif donné et emplit le tableau éventuel avec les correspondances
<code>int preg_match_all (string motif, string chaîne [, array tableau])</code>	Renvoie dans un tableau toutes les occurrences du motif dans la chaîne. Un ordre peut être envoyé par le paramètre optionnel
<code>mixed preg_replace (mixed motif, mixed nouveau_motif, mixed chaîne [, int limite])</code>	Remplace le motif par le nouveau motif dans la chaîne et retourne la nouvelle chaîne. Si la limite est posée, le nombre de remplacements sera limité.
<code>array preg_split (string motif, string chaîne [, int limite])</code>	Découpe une chaîne en se servant du motif comme séparateur.

Fonctions pour des expressions rationnelles compatibles Perl

Un certain nombre d'options affinent encore la recherche. Ces expressions rationnelles compatibles avec le langage Perl sont caractérisées par des contenants qui sont généralement des barres obliques ou qui peuvent être # ou %... Tout caractère non utilisé dans le motif recherché peut être mis à contribution. Reprenons l'exemple de l'extraction du nom du fichier dans le mini-forum.

```
preg_match("#^/.*/(.*)\.[A-z0-9]{3,4}$#", $PHP_SELF, $tableau);
```

Ici les caractères de limite d'expression sont #. Le premier caractère ^ désigne un début de ligne alors qu'il signifie une négation dans des crochets. Le dernier caractère \$ signifie une fin de ligne. Nous avons utilisé # et non pas / qui est déjà utilisé dans le motif recherché : un chemin avec des répertoires.

Ensuite le premier caractère est une barre oblique qui est présente au début de la valeur de \$PHP_SELF qui affiche le chemin à partir du répertoire web. Ensuite nous avons .* Le point représente n'importe quel caractère et n'importe quel nombre de caractères, il peut s'agir de plusieurs répertoires ou d'aucun. En suivant l'expression, nous trouvons une autre barre oblique. Si nous regardons plus loin, nous trouvons le schéma du fichier avec le point et les 3 ou 4 caractères qui caractérisent son format et qu'on appelle son extension (.doc, .jpg, .php, .html, .php3...).

Nous savons que ce que nous voulons récupérer est le mot entre la dernière barre oblique et le point. Nous caractérisons l'extension, le point et la barre oblique pour bien délimiter le motif qui représente notre objectif. Cet objectif est représenté par le (.*)

caractère	description
\w	caractère de mot. les caractères de mot sont les alphanumériques et le blanc souligné (_).
\W	caractère de non mot
\b	limite de mot (entre le \W et le \w)
\B	limite de non mot
\d	caractère numérique
\D	caractère non numérique
\n	caractère de nouvelle ligne
\s	caractère d'espace
\S	tout caractère sauf un caractère d'espace
Les caractères d'échappement dans les expressions rationnelles	

Faisons un système de fichier très simple. Imaginons que vous ayez un répertoire où vous voulez mettre tous vos fichiers sans fioritures. Ce qui vous intéresse et ce qui intéresse vos visiteurs, c'est juste le contenu des fichiers. Vous avez peu de temps et vous voulez bien coller votre texte dans un modèle ou gabarit mais il vous pèse d'ajouter les liens de navigation à chaque nouveau fichier. Nous allons donc créer un fichier qui sera à l'écoute de votre répertoire. Ce fichier dans un premier temps ira chercher la liste des fichiers dans une table. Chaque fichier sera affiché selon ses caractéristiques : nom, titre, date de création et dans l'ordre de date c'est-à-dire que les plus récents seront en haut de la liste.

Il subsiste une objection en ce sens que certains fichiers seront pérennes et d'autres purement circonstanciels. Pour différencier les deux types de fichier, nous appellerons les fichiers d'actualité par un nom de date de type AAAAMMJJ. Plutôt que de mettre la date du jour, nous mettrons la date de péremption, ainsi il sera simple de les archiver dès que la date sera dépassée.

Nous rentrons dans un tableau, les types de fichiers qui seront affichés. Ainsi il est inutile d'afficher les fichiers gif ou jpg.

```
<?
include_once "../commun/fonctions.inc.php";
include_once RACINE."/commun/connexion.inc.php";
$extensions=array ('htm', 'html', 'php', 'php3');
/*si $archi est égale à 0 c'est un fichier normal, si c'est 1, cette
application liste les fichiers archivés. Ce sera le même fichier enregistré
sous un autre nom comme archives.php*/
$archi='0';
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
<title>La page web</title>
</head>
<body bgcolor="#FFFFFF" text="#000000" link="#0033FF" vlink="#990000"
alink="#FF0000">
<table width="90%" border="0" cellspacing="2" cellpadding="2">
```

```

<tr>
  <td colspan="2">
    <div align="center"><b><font face="Arial, Helvetica, sans-serif"
size="3">
      LES PAGES WEB</font></b></div>
    </td>
  </tr>
  <tr>
    <td colspan="2" align="left">
<table width="50%">
<?
/*Si vous mettez le fichier dans le répertoire, laissez ces 3 lignes sinon
utilisez la variable $rep en dessous qui est inactive (mise en commentaire)*/
preg_match("#^(.*)/(.*)\\.([A-z0-9]{3,4})$#", $SCRIPT_FILENAME, $tableau);
$rep=$tableau[1];
/*nous stockons le nom du fichier pour qu'il ne soit pas mis dans la base*/
$fichier_courant=$tableau[2].$tableau[3].$tableau[4];
//$rep = "../BonsVivants";
///AFFICHAGE DES DONNÉES/////
/*ici nous changeons la valeur selon que le fichier soit destiné à des
archives ou à des fichiers pérennes*/
$sql="select * from fichiers where archive='$archi' OR archive='' ORDER BY
moment DESC";
$resultat = @mysql_db_query($dbname, $sql, $id_link);
$nombre = @mysql_num_rows($resultat);
if ($nombre>0){
while($rang=mysql_fetch_array($resultat)){
$nom_fichier=$rang['nom_fichier'];
$titre=$rang['titre'];
$archive=$rang['archive'];
$moment=$rang['moment'];
$moment_crea=date("j/n/Y",$moment);
echo "<tr><td><a href=\"\$nom_fichier\"
target=\"_blank\">$titre</a></td><td><i>$moment_crea</i></td></tr>\n";
}
}
else {
echo "<td colspan=2>Aucun fichier n'est pour l'instant disponible. Revenez
bientôt sur cette page.</td></tr>";
}
///fin de l'affichage//////////
///fourniture de la base de données/////
if (!$archi){
$aujourdhui=date("Ymd");
//cherchons la date la plus récente///
$sql="select MAX(moment) as date_limite from fichiers";
$resultat = @mysql_db_query($dbname, $sql, $id_link);
$nombre = @mysql_num_rows($resultat);
if ($nombre==1){
//la date dans la base est entrée en TIMESTAMP UNIX/////

```

Attention **TIMESTAMP**

Le **TIMESTAMP** Unix est le nombre de secondes depuis le 1^{er} janvier 1970 alors que le **TIMESTAMP** MySQL est un nombre de type **AAAAMMJJHHMMSS**

```

$rang=mysql_fetch_array($resultat);
$date_limite=$rang['date_limite'];
$maintenant=date("Ymd",$date_limite);
}
else {
$date_limite=time()-(60*60*24*365*20);
$sql="CREATE TABLE fichiers (
  clef int(11) NOT NULL auto_increment,
  nom_fichier varchar(60) NOT NULL default '',
  titre varchar(100) NOT NULL default '',
  archive char(1) NOT NULL default '1',
  moment bigint(20) NOT NULL default '0',

```


Si le fichier a un nom de type 20020601, ce nom sera comparé à la date du jour et s'il est moins récent, la valeur d'archive sera mise sur 1. Ceci est intéressant pour des fichiers d'actualité qui deviendront obsolètes après une certaine date et que nous archivons.

```

<?
if (!$archi){
$sql="select * from fichiers where archive='0' OR archive='';
$resultat = @mysql_db_query($dbname, $sql, $id_link);
while($rang=mysql_fetch_array($resultat)){
$nom_fichier=$rang['nom_fichier'];
$clef==$rang['clef'];
preg_match("#^(.*)\.[A-z0-9]{3,4}$#", $nom_fichier, $tableau);
$nom=$tableau[1];
if (ereg("[0-9]{8}",$nom) && $nom<$aujourd'hui){
$sql="update fichiers set archive='1' where nom_fichier='$nom_fichier';
@mysql_db_query($dbname, $sql, $id_link);
}
}
}
?>
<p>&nbsp;</p>
</body>
</html>
LEGENDE ecoute_fichiers table_ecoute.php

```

Vous aurez noté que nous allons chercher la date de dernier changement. Si vous vous contentez de changer votre fichier de répertoire, cette date ne sera pas modifiée. Pour la modifier, il faut faire un changement dans le fichier. Attention si, une fois entré, vous modifiez ce fichier, il apparaîtra 2 fois dans la liste avec deux dates différentes. Avant de faire cette manipulation, vous devrez enlever son entrée dans la base ou bien mettre un autre fichier plus récent dans le répertoire juste avant, ce qui permettra de repousser la date limite et donc votre fichier plus ancien passera le contrôle sans problème.

C08-03.pcx une table d'écoute : attention de bien mettre un titre pour chaque fichier !

Nous nous sommes aménagé une possibilité car nous pouvons préférer afficher une autre date pour des raisons d'actualité. Nous récupérons la date que nous avons ajoutée dans le fichier lui-même en l'encadrant de balises de notre invention `<date>21/09/1999</date>`. Nous prenons soin de l'insérer dans un marqueur de commentaire HTML pour être certain qu'il ne sera pas affiché. Ici l'expression rationnelle joue le rôle d'un analyseur XML. L'expression rationnelle utilisée par les fonctions compatibles Perl utilisent des délimiteurs et des modificateurs comme ici le modificateur `i` rend l'expression insensible à la casse.

Remarque les motifs sur plusieurs lignes

Le caractère point représente n'importe quel caractère sauf les caractères nouvelle ligne \n. Pour que ce caractère soit assimilé au point, il vous faut utiliser le modificateur `s`.

Ce système de fichiers peut être complexifié à l'échelle de tout un site en prenant bien soin de signaler les répertoires privés ou ceux destinés à classer les images du site pour les protéger. Vous pouvez aussi afficher comme nous l'avons fait pour l'annuaire de liens.

modificateur	description
<code>A</code>	cherche le motif seulement au début de la cible
<code>E</code>	cherche le motif seulement à la fin de la cible et le caractère est pris comme caractère
<code>U</code>	s'arrête dès que le premier motif qui satisfait à la recherche est trouvé
<code>i</code>	le motif est insensible à la casse

m	permet d'utiliser les ancrages <code>^</code> et <code>\$</code> pour aussi des débuts ou des fin de ligne
s	force le moteur à considérer le caractère <code>\n</code> comme assimilé dans le caractère "point"
x	permet de mettre des espaces dans votre expression dont le moteur ne tiendra pas compte. Sert à rendre vos expressions plus lisibles

Nous avons vu dans la partie POSIX, ce qui concerne les quantificateurs. Il est possible avec les expressions rationnelles compatible Perl d'utiliser des quantificateur non-avides c'est-à-dire qu'ils arrêtent la recherche dès que le motif a une occurrence.

non-avide	nombre de fois
*?	0, 1 ou plus
+?	1 ou plus
??	0 ou 1
{min, max}?	au minimum min et au maximum max
{min, }?	au moins min
{nombre}?	exactement nombre

Quantificateurs pour une recherche minimale

Quelques études de cas

Au cours de ces pages, nous avons utilisé des expressions rationnelles compatibles Perl sans les expliquer. Reprenons-les :

```
preg_match("#^(.*)\.[A-z0-9]{3,4}.$#", $f, $tableau);
$nom=$tableau[1];
$extension=$tableau[2];
```

Dans cette expression, les délimiteurs sont #. Nous recherchons un motif qui tient sur une ligne, ce qui simplifie la recherche car cela évite l'avidité du moteur. Sur des expressions avides, le moteur ne se contente pas d'avoir trouvé le motif, il a la tentation d'aller le chercher encore plus loin. Le point précédé d'un signe d'échappement est un vrai point et non pas un caractère joker. Ensuite l'extension du fichier peut contenir des lettres mais aussi des chiffres au nombre de 3 ou 4.

```
$heure=preg_replace("/\d{8}(\d{2})(\d{2})\d{2}/", "\\1", $datedujour);
```

Cet exemple prend une date de type AAAAMMJJHHMMSS dont on extrait l'heure, c'est-à-dire la première parenthèse. Le motif est composé de 8 chiffres suivis d'une série de 3 fois 2 chiffres.

Lexique l'atome

L'atome est la plus petite particule de la matière. En grec, atomos signifie ce qui ne peut être divisé.

```
$contenu_txt=@preg_replace("#(\s)((\S)+?(@)+?(\S)+?(\.)+?[A-z0-9_-]{2,3})((\.\.)*<br>\s)|((\.\.)*<p>\s)|((\.\.)*</p>\s)|((\.\.)*\s))#i", "\\1<A HREF=\"mailto:\\2\">\\2</a>\\7", $contenu_txt);
```

Ce motif est celui d'une adresse email à laquelle nous ajoutons un lien mailto. La seule obligation de concernant cette adresse email est qu'elle soit encadrée d'espaces ou de balises de fin de ligne.

Décomposons cette expression :

atome	description
(\s)	espace
(\S)+?	au moins un non-espace (un caractère différent d'un espace)
(@)+?	au moins un arobase
(\S)+?	au moins un non-espace. Juste après l'arobace, le moteur continuera sa route aussi loin qu'il ne rencontrera pas d'espace et jusqu'au point qui suit
(\.)+?	au moins un point
[A-z0-9_-]{2,3}	après le point entre 2 et 3 caractères alphanumériques ou blanc souligné ou tiret
Analyse de la première partie de l'expression : ("#\s)(\S)+?(@)+?(\S)+?(\.)+?[A-z0-9_-]{2,3})	

atome	description
((\.)* \s)	point éventuel suivi d'une balise et un espace comme une fin de ligne
((\.)*<p>\s)	idem avec la balise <p>
((\.)*</p>\s)	idem avec la balise </p>
((\.)*\s)	idem sans balise
Analyse de la seconde partie de l'expression : (((\.)* \s) ((\.)*<p>\s) ((\.)*</p>\s) ((\.)*\s))	

Ces différents atomes sont reliés par le signe | (ou). Ainsi les signes qui suivent la partie qui nous intéresse, l'adresse email, est une fin de ligne ou un espace blanc.

Nous pouvons déjà améliorer l'expression en enlevant la deuxième ligne de cette deuxième partie et en modifiant la ligne suivante

```
((\.)*</p>\s)
```

Ainsi la ligne oblique peut être ou non présente.

À la fin du premier membre de l'expression qui consiste en ces deux parties, nous avons ajouté le modificateur *i* après le délimiteur pour que l'expression soit insensible à la casse.

```
"\1<A HREF="\mailto:\2">\2</a>\7"
```

Nous utilisons dans le deuxième membre de l'expression, la récursivité avec les deux barres obliques inversée suivies d'un chiffre qui correspond à la place des parenthèses de la gauche vers la droite et du général au particulier. La parenthèse 0 est l'ensemble du motif, la parenthèse 1 est ici un espace et la 2, l'adresse email. Enfin nous terminons par la 7^e parenthèse qui est l'espace ou la fin de ligne terminant le motif.

```
preg_match ("/(<(title)[^>]*>)+?((\.\n)*?)(</\2>)/i", $contenu_txt,
$strouvailles);
$titre.= " ".$strouvailles[3];
```

Ici nous cherchons à capturer ce qui est contenu dans les balises <title> et </title>, en incluant l'éventualité qu'elles peuvent chevaucher une ligne. L'atome que vous rencontrez au milieu ((\.\n)*?) est ce que l'on cherche vraiment, c'est-à-dire tous les caractères compris entre les deux balises.

Attention	le caractère point
-----------	--------------------

Le caractère point ne comprend pas le caractère de nouvelle ligne.

Le premier atome est la première balise. Il est composé du premier chevron suivi de `title` puis d'un motif assez obscur : `[^>]*` qui est là pour éviter que le moteur soit trop gourmand mais pour qu'il exige quand même la présence du chevron fermant. C'est le genre d'expression que l'on trouve à la suite de tests et d'approximations. Le motif finit avec la répétition de `title (\\2)` encadré par des chevrons. Le tout est modifié par `i` qui le rend insensible à la casse. Être un orfèvre en expressions rationnelles demande de l'intuition et de l'expérience. Ce n'est pas vraiment une science exacte – si la programmation informatique n'en a jamais été une. C'est cependant un outil indispensable et ses possibilités sont encore largement inexplorées.

Conclusion

Les expressions rationnelles demandent beaucoup de pratique car chaque problème posé est unique. La construction d'une expression part d'une base de caractères puis, par approximations, cherche à atteindre la précision requise. Cette précision se nomme la granularité. L'objectif étant d'atteindre la granularité la plus appropriée. Si elle est trop fine, vous ne trouverez rien, si elle est trop grossière, vous aurez trop de résultats. Vous le voyez dans les moteurs de recherche. Si vous êtes trop précis, vous n'avez aucun résultat.