Cours PHP – Les expression régulières

Nicolas Chambrier – naholyr@yahoo.fr
phpFrance - http://phpfrance.com/forums/

1. Introduction

1.1. Un peu de théorie

On appelle « alphabet » un ensemble de symboles.
On appelle « mot » une séquence de symboles.
On appelle « vocabulaire » l'ensemble de tous les mots de l'alphabet.

Par exemple, si on prend pour alphabet l'ensemble {0,1,2,3,4,5,6,7,8,9}. On appelle un symbole de cet ensemble un « chiffre » (et oui !). un mot de cet alphabet sera un « nombre » (sauf le mot vide) et le vocabulaire sera l'ensemble des entiers (plus le mot vide).

J'avais prévu de vous raconter beaucoup de choses sur la théorie des expressions. Mais je me rends compte que ce n'est pas ce que vous attendez de ce cours. Je laisse cependant cette section, si jamais vous avez envie que j'y inclus un rappel de la théorie plus complet, ecrivez-moi.

1.2. Les expressions régulières, pourquoi ?

Imaginons ce cas pratique. Vous avez envie de trouver dans un texte tous les mots qui commencent par 'a'.
Vous pouvez parcourir tout le texte, caractère par caractère, et quand le caractère précédent était un caractère d'espacement, et que le caractère courant est un 'a' récupérer le mot complet, et le mettre de côté.
Ou alors vous pouvez dire à votre programmer : « trouve tous les mots qui commencent par 'a' dans le texte. ». Ce serait mieux non ?

Imaginons autre chose : vous voulez entourer tous les liens et les e-mails d'un texte entre crochets [ ]. Il va donc falloir détecter tous les liens qui commencent par http:// ou ftp:// (ou je ne sais quoi) et les mail qui sont de la forme : nom@adresse.domaine. Pas facile hein ?

Et bien dans les deux cas, avec une seule instruction, et les expressions régulières, vous auriez pu régler le problème. Convaincu ?

1.3. Les expressions régulières c'est quoi ?

Les expressions régulières sont des chaînes de caractères, permettant de représenter des mots de façon générale.
Par exemple, elle pourront signifier « les mots qui commencent par 'a' » ou « les mots qui contiennent 'ex' » ou encore « les mots en majuscules » ou je ne sais quoi encore.
Les applications sont immenses et vous en aurez besoin un jour ou l'autre, croyez-moi.
Nous allons voir comment construire des expressions régulières en pratique dans le chapitre 2.
Concrètement, dans la plupart des langages de programmation, il existe des librairies ou des

extensions permettant d'utiliser les expressions régulières. Ces extensions vous donneront accès à des fonctions, qui prendront en paramètre une chaîne décrivant une expression régulière, et une chaîne sur laquelle appliquer l'expression régulière (et éventuellement d'autres paramètres).
Typiquement, la fonction de base prend ces deux arguments, et répond « oui » si l'expression régulière reconnaît la chaîne, et « non » dans les autres cas.
En PHP, cette fonction s'appelle ereg, ou preg_match.

1.4. Que puis-je faire et ne pas faire avec les expressions régulières ?

A priori je dirais que rien n'est impossible avec les expressions régulières.
En pratique vous verrez qu'il existe deux types d'expressions régulières, deux normes en fait : POSIX, et PERL.
Les expressions régulières reconnus en standard par PHP sont les POSIX. Avec l'extension PCRE (en standard sur pratiquement tous les serveurs), on peut bénéficier des « Perl Compatible Regular Expression ».

Les expressions régulière POSIX ne vous permettront pas par exemple de remplacer tous les mots commençant par un 'a' par leur longueur. Alors que les PCRE vous le permettront.

En gros, tout traitement sur des chaînes de caractères pourra se réduire à une (la plupart du temps) ou plusieurs instructions basées sur les expressions régulières. Je n'ai pas encore vu de problème de ce type qui ne puisse être réglé par les PCRE.

Quelques exemples très classiques :
- vérifier qu'une adresse e-mail est syntaxiquement correcte.
- rendre cliquable tous les liens d'une page html.
- mettre en majuscule la première lettre de chaque mot.
- récupérer le contenu d'une page html (transformer les tables en tableaux PHP par exemple).
- mettre en place un système de balises (comme dans le forum phpBB par exemple).

1.5. Quelques notions préliminaires sur les chaînes de caractères

Les expressions régulières permettant de manipuler les chaînes de caractères, il convient de savoir précisément utiliser les chaînes de caractères.
Les pré-requis pour ce manuel sont les suivants (je ne reviendrai à aucun moment sur l'un de ces points) :
- interprétation de variables au sein d'une chaîne (différence entre '$var' et "$var").
- caractères spéciaux usuels (par exemple \r, \n, \t).

Je ne reviendrai sur ces points, sauf si la demande l'exige. Dans ce cas-là, je prévoirai un petit paragraphe dans ce chapitre pour les rappels nécessaires.

2. Comment ça marche en PHP ?

2.1. Les fonctions utilisant les expressions régulières.

Voici les fonctions manipulant les expressions régulières POSIX :
- ereg : http://fr.php.net/ereg
- eregi : http://fr.php.net/eregi
- ereg_replace : http://fr.php.net/ereg-replace
- eregi_replace : http://fr.php.net/eregi-replace
- split : http://fr.php.net/split
- spliti : http://fr.php.net/spliti

Voici les fonctions manipulant les expressions régulières PCRE :
preg_match : http://fr.php.net/preg-match
preg_match_all : http://fr.php.net/preg-match-all
preg_grep : http://fr.php.net/preg-grep
preg_replace : http://fr.php.net/preg-replace
preg_replace_callback : http://fr.php.net/preg-replace-callback
preg_split : http://fr.php.net/preg-split

Il en existe d'autres, qui permettent de manipuler les expressions régulières plus ou moins directement. Je ne commenterai pas toutes ces fonctions, mais je vous conseille vivement de suivre les liens donnés (documentation PHP en ligne) et si vous n'avez pas internet, de lire la documentation concernant ces fonctions.
Cette documentation est extrêmement fournie, et représente un travail énorme que j'admire chaque jour. Je vous assure que cette documentation saura répondre à la plupart de vos questions pour peu que vous fassiez l'effort d'y chercher la bonne information. Pour la télécharger : http://www.php.net/download-docs.php

2.2. syntaxe des expressions régulières POSIX

PHP utilise les expressions régulières avancées de POSIX (POSIX 1003.2) *[source : documentation PHP]*.

Les bases :

Soit une expression régulière ***e***, et une expression régulière ***e'***
- ***ee'*** reconnaît toute concaténation de deux mots. Le premier reconnu par ***e*** et le second par ***e'***.
- ***(e)*** est équivalent à ***e***.
- ***(e)**** reconnaît le mot vide, et toute concaténation de mots reconnus par ***e***. Cette expression régulière est équivalente à ***(|(e)+)*** (le mot vide, ou ***e***+).
- ***(e)|(e')*** reconnaît tous les mots reconnus par ***e*** ou par ***e'***.
- ***(e)***+ reconnaît toute concaténation de mots reconnus par ***e*** mais pas le mot vide ! cette expression régulière est équivalente à ***e(e*)*** (***e*** concaténée à ***e****).

Quelques exemples :

Les mots qui ne contiennent que ‘a’ et ‘b’ : ***(a|b)+*** (a ou b, plusieurs fois)
Les mots qui commencent par un ‘a’, et qui ne sont composés que de ‘a’ et de ‘b’ : ***a(a|b)****
Les mots du genre aaabbbbb : ***a+b+***
Les mots composés des lettres a, b, c et qui contiennent au moins un a : ***(a|b|c)*a(a|b|c)****

Les répétitions finies :

On aimerait définir les mots contenant exactement 20 caractères.
Par exemple, un expression régulière permettant de reconnaître les mots commençant par un ‘a’, suivi de 30 caractères ‘a’ ou ‘b’, et finissant par un ‘b’.
On définit les répétitions grâce aux opérateurs { }.
Soit une expression régulière ***e*** et deux entiers ***x*** et ***y*** :
- ***e{x}*** reconnaît les mots définis comme concaténation de ***x*** mots reconnus par ***e***.
- ***e{x,y}*** reconnaît les mots définis comme concaténation de ***x*** à ***y*** mots reconnus par ***e***.
- ***e{x,}*** reconnaît les mots définis comme concaténation de ***x*** mots au moins, reconnus par ***e***.
- ***e{,y}*** reconnaît les mots définis comme concaténation de ***y*** mots au plus, reconnus par ***e***.

Le point d’interrogation ***?*** est l’équivalent de ***{,1}***.
Ainsi ***e?*** reconnaît le mot vide ou un mot reconnu par ***e***.

Les classes de caractères :

Comme vous le devinez, il serait intéressant de pouvoir dire « n’importe quel caractère ».
Ou encore, pour décrire « tous les mots composés de caractères de ‘a’ à ‘z’ », pour le moment on est condamné à utiliser l’expression régulière
(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)* ce qui est – convenons-en – très lourd.
Heureusement, les classes de caractères sont là pour nous simplifier la vie.

Une classe de caractère est entre crochets [et], et définit un ensemble de caractères admis dans la classe. Chaque caractère apparaissant dans les crochets fait partie de la classe.
On est donc réduit à cette syntaxe : ***[abcdefghijklmnopqrstuvwxyz]****.
C’est déjà beaucoup mieux mais pas encore fantastique. Là encore, il y a plus simple : on peut définir des *intervalles* de caractères grâce au tiret ‘-‘.
Soit deux caractères ‘c1’ et ‘c2’. La classe ***[c1-c2]*** définit la classe des caractères situés dans l’ordre standard (ASCII) entre c1 et c2. Comme le codage ASCII est bien fait, les caractères alphanumériques se suivent : entre ‘a’ et ‘z’ se trouvent toutes les minuscules de a à z, de même qu’entre ‘A’ et ‘Z’, et entre ‘0’ et ‘9’.
Dans notre cas, l’expression régulière se réduit alors à ***[a-z]****.
Voici la syntaxe des classes de caractères :
[classes] où **classes** est une séquence de **classe**, où **classe** est de la forme **intervalle** ou **liste de caractères**.
La classe des caractères alphanumériques est donc **[a-zA-Z0-9]**.
L’accent circonflexe définit la négation d’une classe. Nous verrons cette particularité en détails un peu plus loin.

Les classes de caractères prédéfinies :

[[:alpha:]] définit la classes des caractères alphabétiques, équivalent à **[a-zA-Z]**.
[[:alnum:]] définit la classes des caractères alphanumériques, équivalent à **[a-zA-Z0-9]**.
[[:digit:]] définit la classe des caractéres numériques, équivalent à **[0-9]**.
[[:space:]] définit la classes des caractères d'espacement, équivalent à **[\t]** (espace, ou tabulation).
Cette liste est non exhaustive, mais il s'agit là des plus utilisés.
On peut cumuler les classes prédéfinies, par exemple **[[:alpha:][:digit:]]** est équivalent à **[[:alnum:]]**.
Notez bien que ces classes prédéfinies sont obligatoirement utilisées à l'intérieur de crochets.
La classe **[:digit:]** ne reconnaîtra pas **O** ! En fait cette classe correspond à la reconnaissances des caractères ':', 'd', 'i', 'g', et 't'.

Les caractères spéciaux :

Imaginons que vous vouliez reconnaître les mots commençant par un 'a' suivi d'une étoile, et suivi d'au moins 4 caractères alphanumériques. Vous seriez tenté d'écrire ***a*[:alnum :]{4,}***. Or si l'on regarde cette expression régulière, elle reconnaîtra les mots contenant 0 ou plusieurs 'a' suivis d'au moins 4 caractères alphanumériques. Le problème étant que l'étoile fait partie de la syntaxe des expressions régulières.
Or ces caractères spéciaux sont nombreux, par exemple {, }, (,), -, [,], *, +, et d'autres…
Pour avoir le droit de les utiliser dans une expression régulière, en tant que symboles comme les autres, il faut les précéder d'un antislash (backslash '\'). On dit « échapper le caractère ».
Notre expression régulière devient alors ***a*[:alnum :]{4,}*** et maintenant elle est juste !
Tous les caractères spéciaux doivent ainsi être échappés.

La liste des caractères spéciaux : ^, $, ., ?, *, +, {, }, (,), [,], -, |, \

Les caractères ^ ('chapeau') et $ ('dollar') :

Ces caractères ont un rôle très spécial, et '^' en particulier qui n'a pas le même sens, suivant où il est utilisé.
Leur sens commun est une sémantique de positionnement : ***^e*** reconnaît les mots dont le début est reconnu par ***e***. ***e$*** reconnaît les mots dont la fin est reconnue par ***e***.
Ainsi, ***^e$*** reconnaît le mot m si et seulement s'il est reconnu exactement par ***e***.
'^' n'a pas strictement besoin d'être placé en début du masque, s'il est placé dans une alternative. Mais il doit être placé au début du masque qu'il spécifie.
Le comportement de '$' est le même (placé en fin).
Par exemple, reconnaître tous les mots qui commencent par b, ou qui contiennent un a, et qui finissent par c (avec n'importe quel caractère au milieu) : ***(^b|a).*c$***. Cette expression reconnaîtra par exemple ***bac***, ***fac***, ***marc***, ***bic***…

Le caractère '^' peut cependant avoir un autre rôle, s'il est utilisé dans des classes de caractères, il a alors un rôle de négation.
Par exemple ***[^a]*** reconnaît tout sauf les 'a'.
Les intervalles peuvent toujours être utilisés : ***[^[:alnum:]]*** reconnaît les caractères non alphanumériques.

2.3. syntaxe des expressions régulières Perl (PCRE)

Les expression régulières Perl sont disponibles quand l'extension PCRE est disponible. C'est en pratique toujours le cas (elle est en standard avec PHP).
Les expressions régulières Perl sont plus performantes (plus rapides) et plus puissantes que les expressions régulières POSIX. Leur syntaxe est plus étendue que pour POSIX, évidemment.
Je vous encourage vivement (comme d'habitude) à lire la documentation (chapitre « IV. LXXXIX. Expressions régulières compatibles Perl ») qui donne des détails importants que je ne développerai sûrement pas aussi profondément dans ce chapitre. En particulier, je n'aborderai pas les différences avec Perl, qui sont explicités dans la documentation au chapitre indiqué. Je précise également que je n'enseigne ici que la **base** de la syntaxe des masques, alors qu'elle est bien plus riche et complexe : LISEZ LA DOC !

Les bases :

Une expression régulière Perl utilise des « *délimiteurs* ».
Ces délimiteurs indiquent le début du pattern (ou *masque*) et sa fin, et peuvent être suivis d'options.
Voici la forme d'un masque classique :
délimiteur regexp délimiteur options
où 'délimiteur' peut être un caractères quelconque, mis à part un caractère numérique, ou le caractère d'échappement '\' que vous avez pu voir précédemment.
Depuis PHP4, les délimiteurs « symetriques » (), [] et {} sont autorisés.
Par exemple, {regexp} ou |regexp| sont des masques valides (pour peu que regexp soit une expression régulière compatible Perl valide). Par la suite j'utiliserai les délimiteur ' (apostrophe) qui a l'avantage d'être peu utilisé et le délimiteur /, traditionnellement utilisé.
Nous verrons les options un peu plus loin.

Quelques exemples :

Les mots qui ne contiennent que 'a' et 'b' : ***'(a|b)+'***
Les mots qui commencent par un 'a', et qui ne sont composés que de 'a' et de 'b' : ***'a(a|b)*'***
Les mots du genre aaabbbbb : ***'a+b+'***
Les mots composés des lettres a, b, c et qui contiennent au moins un a : ***'(a|b|c)*a(a|b|c)*'***
Comme vous pouvez le voir, les expressions régulières simples sont strictement équivalentes en POSIX ou en Perl.

Similitudes avec POSIX :

Les répétitions finies, les caractères ^ et $, ainsi que les classes de caractères ont strictement la même utilisation en Perl.
La seule différence notable se situe au niveau des classes de caractères prédéfinies, et des options de recherche.
Il y a bien d'autres différences, mais très peu connues, et trop complexes a appréhender pour ce

cours. Nous verrons déjà des détails rarement abordés dans ce genre de document, référez-vous à la documentation pour plus de détails (le chapitre sur la syntaxe des masques est très complet).

Les classes de caractères prédéfinies :

Contrairement à POSIX, en Perl on n'utilise pas forcément les classes prédéfinies à l'intérieur de crochets. En plus ces classes n'incluent pas elle-mêmes de crochets, ce qui a pour premier avantage d'éviter la redondance des dits crochets.
Par contre cela rend la compréhension des expression régulières plus difficile puisqu'il sera plus difficile d'y détecter les classes de caractères. Bon, quittons le flou pour lister les différentes classes de caractères prédéfinies en Perl :
\d définit la classe des caractères décimaux. Equivalent de **[[:digit:]]**
\D définit la classe des caractères non décimaux. Equivalent de **[^[:digit:]]**
\s définit la classe des caractères d'espacement. Equivalent de **[[:space:]]**
\S définit la classes des caractères non blancs. Equivalent de **[^[:space:]]**
\w définit la classe des caractères de mot.
\W définit la classe des caractères non-mot.
Attention, ces deux dernières n'ont pas d'équivalent simple en POSIX. Elles incluent les caractères alphanumériques, le caractère de soulignement '_', mais aussi les caractères alphabétiques accentués, qui ne sont pas reconnus par **[[:alpha:]]**.

Les options de recherche :

Il existe 11 options de recherches, nous n'en étudierons que les plus utilisées. Comme d'habitude, je vous renvoie à la documentation pour avoir des détails sur les options non documentées ici.

- m
Par défaut, PCRE considère la chaîne sujet comme une seule ligne, ^ ne sera valide qu'au début de la chaîne, et $ à la fin de la chaîne, ou avant le retour chariot final.
Avec cette option, l'expression régulière est vérifiée sur chaque ligne.
Par exemple ***'^B$'*** ne reconnaîtra pas **"A\nB\nC"** (en fait elle ne reconnaîtra que **"B"** et **"B\n"**), alors qu'avec l'option m, ***'^B$'m*** reconnaîtra cette chaîne, puisque la deuxième ligne est reconnue par l'expression régulière.

- i
Effectue une recherche insensible à la casse.

- s
Une particularité de PCRE par rapport à POSIX se situe au niveau du caractère spécial '.'. Ce méta-caractère (c'est comme ça qu'on appelle les caractères spéciaux des expressions régulières) ne représente pas absolument tous les caractères, puisque les retours chariots (\n et/ou \r) ne sont pas reconnus.
Avec cette option, le point a le même comportement qu'avec POSIX.

- e
Cette option sera documentée plus tard, n'étant utilisable que dans la fonction preg_replace.

Les sous-masques, et les options internes :

Les sous-masques sont les masques placés entre parenthèses dans un masque.
Ils existent également en POSIX, et ont le même rôle (dans les fonctions de remplacement, cf. ereg_replace, et preg_replace).
Cependant une particularité de PCRE est de permettre l'application d'options de recherche uniquement dans des sous-masques.
Imaginons que l'on veuille chercher 'Nicolas' de façon insensible à la casse, mais en assurant que la première lettre soit en majuscules. Si l'on applique le masque ***'Nicolas'*** il ne reconnaîtra que la chaîne exacte. Si l'on applique ***'Nicolas'i*** il sera complètement insensible à la casse, et n'assurera donc pas que le premier 'N' soit en majuscule.
L'idéal serait de faire une recherche insensible à la casse seulement sur la fin de la chaîne.

Pour appliquer une option interne dans un sous-masque, on utilise la syntaxe suivante :
(?options) tout ce qui suit dans le sous masque utilise alors les options.
On peut également utiliser le tiret pour annuler des options.
Par exemple ***'(ab(?i)cd(?-i)ef)'*** reconnaîtra les chaînes contenant "abcdef", "abCdef", "abcDef", ou encore "abCDef".
Définir des options internes à l'extérieur d'un sous-masque, où que ce soit, équivaut à appliquer l'option à tout le masque. Par exemple, ***'a(?i)b'*** sera équivalent à ***'ab'i***.

Dans notre cas pratique, on va donc rechercher la fin en étant insensible à la casse, et la première lettre en y étant sensible.
'(N(?i)icolas)' sera un masque valide pour cette recherche (on place le sujet dans un sous-masque, ou l'on applique l'option 'i' avant la partie insensible à la casse).
'((?-i)N)icolas'i aurait également convenu (on place 'N' dans un sous-masque, où l'on annule l'option 'i' appliquée au masque entier).

3. Les expressions régulières POSIX en détails

3.1. ereg / eregi

Il s'agit des fonctions de base pour vérifier si une chaîne est reconnue ou non par l'expression régulière POSIX passée en paramètre.
Ces fonctions prennent 2 paramètres obligatoires, et un troisième facultatif, qui est un paramètre résultat.

le comportement de base

Quand on ne passe que deux paramètres, le premier est une chaîne de caractère, représentant l'expression régulière, et le second est la chaîne sujet.
ereg renverra TRUE si l'expression régulière reconnaît la chaîne, FALSE sinon.
eregi aura le même résultat, mais est insensible à la casse.

exemples

Vérifier qu'une chaîne correspond bien à une adresse e-mail :
Une adresse e-mail est en deux parties séparée par l'arobase : un login (on autorisera des caractères alphanumériques, et le point, le caractère de soulignement, et le tiret), et un domaine lui-même composé d'un nom (on autorisera les même caractères), d'un point, et d'une extension qui n'est composée que de caractères alphabétiques (n'oubliez pas les domaines genre .fr.st, on les prend bien en compte puisque le point est autorisé dans le nom du domaine).
On utilisera donc ***[[:alnum:]_\.]+@[[:alnum:]_\.]+\.[[:alpha:]]+***.
Comme on veut vérifier que la chaîne EST une adresse mail, pas qu'elle en contient une, on doit vérifier que la chaîne est reconnue du début à la fin, on va donc ajouter ^ et $.
Le test sera donc le suivant :

```
if (ereg("^[[:alnum:]_\.]+@[[:alnum:]_\.]+\.[[:alpha:]]+$", $adresse)) {
      echo "$adresse est invalide.";
}
else {
      echo "$adresse est valide.";
}
```

Note : l'utilisation de ereg est préférée ici puisqu'on utilise de toute façon des classes de caractères générales, incluant majuscules et minuscules.

Le troisième paramètre

Le paramètre optionnel dont je vous ai parlé est un *récupérateur* d'éléments reconnus par l'expression régulière.
Ce paramètre ne doit pas être déclaré avant (il sera de toute façon réinitialisé) utilisation de ereg ou eregi.
Après l'appel, il s'agit d'un tableau, contenant les différents éléments de la chaîne reconnus par

des sous-masques.
Je vous ai déjà parlé des sous-masques : il s'agit de morceaux d'expression régulières placés entre parenthèse. En voici une utilité autre que celle d'*organiser* l'expression régulière.
Le tableau commence toujours à 0, et contient *n*+1 éléments, où *n* est le nombre de sous-masques de profondeur 1. On donne une profondeur aux sous-masques suivant leur degré d'imbrication dans l'expressions. Par exemple dans ***a+(b*(cd)e+)(f{3})*** l'expression elle-même est le sous-masque de profondeur 0 (c'est toujours le cas). Les sous-masques ***(b*(cd)e+)*** et ***(f{3})*** sont de profondeur 1, et ***(cd)*** est de profondeur 2.
ATTENTION ! cette notion de profondeur n'est pas officielle, je l'ai mise en place pour une plus grande précision dans les explications de ce cours.

Revenons à notre troisième paramètre, il contient donc dans l'ordre la sous-chaîne reconnue par le masque principal, puis les *sous-sous-chaînes* reconnues par les sous-masques de profondeur 1, dans l'ordre où ils ont été donnés.
Si on met une répétition sur un sous-masque, il semble que le résultat soit la première des sous-chaînes rencontrées, mais évitez ce genre d'utilisation (par exemple ***a(b|c)*d*** appliqué sur ***ZabcdY*** renvoie TRUE, et met dans le tableau [`"abcd", "c"`] ce qui est difficilement utilisable, et surtout peu prévisible sur des expressions plus complexes).

exemple

On va maintenant déterminer si une chaîne CONTIENT une adresse e-mail, et si oui afficher séparément le login, et le domaine.
On va modifier notre expression régulière précédente pour qu'elle reconnaisse les chaîne CONTENANT une ou plusieurs adresses e-mail, en supprimant ^ et $. Puis on va isoler les deux parties que l'on veut afficher séparément.
Le code devient :

```
if (ereg("([[:alnum:]_\.]+)@([[:alnum:]_\.]+\.[[:alpha:]]+)", $str, $regs)) {
      echo "On a trouvé l'adresse {$regs[0]}.\n";
      echo "LOGIN : {$regs[1]}.\n";
      echo "DOMAINE : {$regs[2]}.";
}
else {
      echo "aucune adresse e-mail dans cette chaîne.";
}
```

3.2. ereg_replace / eregi_replace

Ces fonctions sont encore un peu plus puissantes, puisqu'elles permettent le remplacement de sous-chaînes reconnues par une expression régulière par une autre chaîne.
Elles sont plus puissantes que str_replace (doc : http://fr.php.net/str-replace) mais moins rapides évidemment, préférez donc toujours str_replace pour des remplacements n'ayant pas besoin de la puissance des expressions régulières.
La différence entre ereg_replace et eregi_replace est la même qu'entre ereg et eregi : la seconde reconnaît les chaînes en étant insensible à la casse.
Le premier paramètre est l'expression régulière, le second la chaîne à tester, et le troisième la chaîne de remplacement.

exemple

Je veux supprimer tous les tags d'un code HTML. Cette fonction est très bien réalisée par strip_tags (doc : http://fr.php.net/strip-tags) et je vous conseille d'utiliser cette fonction plutôt que les expressions régulières, mais je choisis cet exemple pour son aspect concret.
Les tags HTML sont entre crochets < et >. Dans un tag il ne peut pas y avoir de crochet fermant > puisque c'est ce dernier qui clôt le tag. Pour reconnaître un tag, on va donc se baser sur cette observation, et construire l'expression régulière ainsi : ***<[^>]+>*** qui correspond bien à tous les tags existants.
Pour supprimer tous les tags d'une chaîne de caractères nous utiliserons ereg_replace, en signifiant qu'on va remplacer tout sous-chaîne reconnue par cette expression régulière par la chaîne vide :

```
$chaine = ereg_replace("<[^>]+>", "", $chaine);
```

Ce qui est bien sûr équivalent à un appel de strip_tags.

Pour simuler nl2br (doc : http://fr.php.net/nl2br) on aurait utilisé str_replace !

```
$chaine = str_replace("\n", "<br/>", $chaine);
```

utilisation des sous-masques de profondeur 1

Comme pour ereg et eregi, on peut dans les fonctions de remplacement basées sur les expressions régulières utiliser les sous-masques. Nous sommes là encore limité en profondeur par le profondeur 1.
On a avec la norme POSIX la possibilité de définir 9 sous-masques de profondeur 1, et de les utiliser dans le chaîne de remplacement !
On peut copier la sous-chaîne reconnue par le *i*_ième sous-masque en insérant dans la chaîne de remplacement la chaîne "*i*" (il s'agit de \i, mais comme dans les chaînes à doubles quotes, il faut échapper le \, on doit donc le doubler, donc dans une chaîne a simples quotes, on aurait simplement *i*).

exemple

On dispose d'un texte, ne contenant pas de tags HTML. On veut rendre les liens cliquables en les plaçant dans un tag <A>. On se place dans un texte n'en contenant aucun pour éviter le cas ou le lien se trouve déjà dans un tag, auquel cas on aura des résultats bizarres ;-).
On va considérer que les liens commencent soit pas http:// soit pas ftp://, puis par une suite quelconque de caractères incluant caractères alphanumériques, caractère de soulignement, tiret, point, et slash : ***[[:alnum:]_\-\./]***. On va remplacer tout lien de ce type par son adresse simple (pas de …:// devant) dans un tag <A>.
On va donc récupérer tous le lien, en deux sous-masques, le premier contenant http ou ftp, le second contenant le lien simple.
Voici l'expression régulière : ***(ftp|http)://([[:alnum:]_\-\./]+)***
On va donc extraire deux sous-chaines : la première contenant http ou ftp, la seconde contenant l'adresse sans le préfixe (ce que l'on veut afficher pour alléger le texte) :

```
$search = '(ftp|http)://([[:alnum:]_\-\./]+)';
$replace = '<a href="\1://\2" target="_blank">\2</a>';
$texte = eregi_replace($search, $replace, $texte);
```

Notez l'utilisation des simples quotes (ce qui évite les caractères d'échappement trop lourdement présents sinon), et la séparation des différents paramètres.
Passer par des variables auxiliaires comme je l'ai fait ici n'est pas plus lent, et permet une plus grande lisibilité, et donc un débuggage plus facile.

3.3. split / spliti

La différence est encore une fois la même, split est sensible à la casse, spliti ne l'est pas.
Pour bien comprendre leur rôle, il faut savoir manipuler explode (doc : http://fr.php.net/explode).
Ces fonctions permettent de séparer une chaîne en plusieurs éléments, en utilisant comme séparateur non pas une chaîne fixe comme pour explode, mais toute sous-chaîne reconnue par l'expression donnée.

exemple

Je veux extraire à partir d'un texte, la liste des mots qui s'y trouvent. Je vais donc *exploser* le texte selon les séparateurs de mots.
Plutôt que de définir ce qu'est un séparateur, je vais définir ce qu'il n'est pas :
Un séparateur n'est pas un caractère alphanumérique, un apostrophe (dans notre cas, on considèrera que « l'orage » constitue un seul mot, et pas deux), un tiret ou un caractère de soulignement.
Ainsi on va définir un séparateur ainsi : ***/^[:alnum:]_\-'/***. On va couper selon les groupes de séparateurs de mots.
Et l'utilisation de split se fera ainsi :

```
$sep = '[^[:alnum:]_\-\']+';
$mots = split($sep, $texte);
```

Cas particuliers

Le résultat de ce code sur un exemple : « j'aime trop le PHP pour m'en passer ! » donnera $mots = array(″J'aime″, ″trop″, ″PHP″, ″pour″, ″m'en″, ″passer″, ″″).
Il s'agit du résultat attendu, à ceci près qu'on a une chaîne vide à la fin, ce qui peut paraître étrange. Ca ne l'est pas tant que ça en fait, car on a bien après ″passer″ un groupe de séparateurs, et split a donc ajouté ce qui suivait dans le résultat : la chaîne vide.

Ce genre de cas particulier arrive extrêmement souvent avec split, spliti, et explode. Il sont prévisibles dans l'absolu, mais en pratique on est souvent surpris de certains résultats, il faut donc toujours avoir ce comportement à l'esprit afin de prévoir les éventuels tests à faire sur le résultat.

De même l'application de split, spliti, ou explode sur une chaîne vide renvoie un tableau contenant un élément vide. Ce comportement est systématique, et même s'il est logique, peut entraîner des conséquences dans un programme où ce cas n'est pas prévu.

3.4. fonctions auxiliaires

Les expressions régulières POSIX ne bénéficient pas d'une pléthore de fonctions auxiliaires (PCRE dispose d'un peu plus d'outils, mais pas tant que ça non plus).
En fait il n'y en a qu'une directement utilisable.

sql_regcase

doc : http://fr.php.net/sql-regcase
Cette fonction transforme une chaîne passée en paramètre en expression régulière insensible à la casse.
Elle permet principalement (dixit la doc.) de préparer des expressions régulières pour les logiciels ne permettant pas de faire de recherche insensible à la casse (par exemple une recherche avec LIKE dans une requête à une base de données MySql sur un champ de type BLOB sera sensible à la casse, utiliser REGEXP au lieu de LIKE, et préparer l'expression avec sql_regcase fonctionnera très bien).

Cette fonction pourra également permettre de faire des recherches partiellement insensible à la casse, comme le font les options internes en PCRE.

Pour reprendre notre exemple de tout à l'heure, où nous avions construire l'expression régulière PCRE ***/N((?i)icolas)/*** pour faire une recherche en étant insensible à la casse seulement sur la fin du mot. Cette recherche est possible en POSIX, avec l'expression régulière ***N[iI][cC][oO][lL][aA][sS]*** qui correspondra tout-à-fait (chaque classe correspond à une lettre en minuscule et son homologue en majuscule).
Evidemment cette expression n'est pas difficile à construire, mais si l'objet de la recherche est déterminé dynamiquement, cela devient plus difficile.
Et bien rien de plus simple avec sql_regcase :

```
$mot = 'Nicolas'; // objet de la recherche
$debut = substr($mot, 0, 1); // initiale
$fin = substr($mot, 1); // le reste
$regexp = $debut . sql_regcase($fin);
echo $regexp;
```

Affichera bien **N[iI][cC][oO][lL][aA][sS]**.

4. Les expressions régulières Perl en détails

4.1. preg_match

preg_match fonctionne EXACTEMENT comme ereg et eregi, à ceci près qu'elle prend en premier paramètre une expression régulière Perl. Cette fonction est également plus performante (en temps d'exécution) que ses homologues POSIX.
Le troisième argument optionnel a exactement le même rôle également.

exemples

Nous allons reprendre les deux exemples traités dans la section sur ereg/eregi.
Le premier exemple permettait de dire si une adresse e-mail était syntaxiquement valide ou non.
En POSIX, l'expression régulière était ***^[[:alnum:]_\.]+@[[:alnum:]_\.]+\.[[:alpha:]]+$***
En PCRE, on va coller à cette expression, en utilisant simplement les classes réservées propres : ***/^[\w\.]+@[\w\.]+\.[a-z]+$/i***.
Cela donnera :

```
if (preg_match("/^[\w\.]+@[\w\.]+\.[a-z]+$/i", $adresse)) {
      echo "$adresse est invalide.";
}
else {
      echo "$adresse est valide.";
}
```

Le deuxième exemple permettait de trouver une adresse e-mail dans un texte, et d'en extraire les deux composantes.
Le raisonnement est à nouveau le même, et mène au code suivant :

```
if (preg_match("/([\w\.]+)@([\w\.]+\.[a-z]+)/i", $texte, $regs)) {
      echo "On a trouvé l'adresse {$regs[0]}.\n";
      echo "LOGIN : {$regs[1]}.\n";
      echo "DOMAINE : {$regs[2]}.";
}
else {
      echo "aucune adresse e-mail dans cette chaîne.";
}
```

À ce stade vous devriez avoir bien compris le concept, et surtout assimilé que la différence entre POSIX et PCRE n'est pas si profonde que ça. Il ne s'agit que d'une différence de syntaxe, et également du moteur qui tourne derrière tout ça, celui des PCRE étant plus performant.
C'est pourquoi, il est intéressant d'utiliser plus particulièrement les PCRE que les expressions régulières POSIX. La difficulté n'étant pas spécialement accrue, et les performances meilleures.

4.2. preg_match_all

Cette fonction permet de faire quelque chose qui n'est pas possible avec les expressions POSIX (du moins impossible aussi simplement).
Elles permet en effet d'extraire d'une chaîne TOUTES les sous-chaînes reconnues par la PCRE passée en paramètre.

Les paramètres

preg_match_all prend 3 à 4 paramètres.
Les trois premiers sont dans l'ordre : l'expression régulière PCRE, la chaîne sujet, et le paramètre-résultat qui contiendra la tableau des occurrences rencontrées. Ce tableau a toujours deux dimensions.
Le 4e est un entier, qui indique dans quel ordre seront rangés les résultat.

PREG_PATTERN_ORDER est la valeur par défaut.
Quand cet ordre est spécifié, le tableau résultat est tel que son premier élément (d'indice 0) est la liste des sous-chaînes reconnues par le masque complet (sous-masque de profondeur 0), et les éléments suivants (d'indice *i*) sont la liste des sous-chaîne reconnues par le sous-masque correspondant (le *i*_ème sous-masque).

PREG_SET_ORDER
Quand cet ordre est spécifié, le tableau est une liste de résultats. Chaque résultat étant un tableau tel que son premier élément est la première sous-chaîne reconnue par l'expressions régulière, et les éléments suivants correspondent aux sous-masques appliqués à cette chaîne.

Dans cette fonction, la profondeur n'est pas limitée à 1, et je vous conseille d'éviter les expressions contenant trop d'imbrications car le résultat n'est pas forcément très prévisible.

exemple 1

On va récupérer toutes les adresses e-mail contenues dans un texte.
Nous disposons déjà de la PCRE correspondante, il s'agit de ***/[\w\.]+@[\w\.]+\.[a-z]+/i***.
Nous allons donc effectuer notre recherche en appelant :

```
preg_match_all("/[\w\.]+@[\w\.]+\.[a-z]+/i", $texte, $resultat);
```

On applique alors l'ordre PREG_PATTERN_ORDER ce qui signifie qu'on a :
- $resultat[0] = la liste des e-mail du texte
Et pas d'autre élément puisqu'on n'a pas de sous-masques.

Si l'on avait spécifié un autre ordre (PREG_SET_ORDER), on aurait obtenu un tableau de *n* éléments (où *n* est le nombre d'e-mail dans le texte), chaque élément possédant une seule valeur d'indice 0 correspondant à l'adresse mail en question. C'est ici un ordre à éviter puisqu'il donne un résultat bien trop complexe !

Le code suivant liste tous les e-mail d'un texte aussi simplement que ça :

```
preg_match_all("/[\w\.]+@[\w\.]+\.[a-z]+/i", $texte, $resultat);
print_r($resultat);
```

exemple 2

Imaginons que vous traitiez un fichier contenant des triplets nom/prénom/adresse sous la forme suivante : <NOM~PRENOM~ADRESSE>.
On considère qu'un nom et un prénom ne peuvent contenir que des tirets ou des caractères alphabétiques. Et un adresse est composés de caractères alphanumériques, les espaces et les tirets ainsi que les retours à la ligne sont autorisés.
L'expression régulière correspondant à ce cas est donc ***/<[a-z\-]+~[a-z\-]+~[\w\s\-\r\n]+>/i***.
Rien de très compliqué. Comme on veut récupérer les différents éléments directement, on va placer des parenthèses pour définir les sous-masques : ***/<([a-z\-]+)~([a-z\-]+)~([\w\s\-\r\n]+)>/i***.

On va appliquer directement preg_match_all au contenu du fichier avec cette expression régulière. Observons les différents résultats suivant l'ordre choisi.
Notre fichier contient **<Dupont~Jean~Quelque-part><De-Partout~Albert~Nulle-part><Dalton~Joe~Jails>**.

Voici le code appliqué :

```
preg_match_all("/<([a-z\-]+)~([a-z\-]+)~([\w\s\-\r\n]+)>/i", $txt, $res);
```

Dans ce cas, on applique l'ordre par défaut PREG_PATTERN_ORDER. Nous obtiendrons donc dans $res[0] tous les triplets, dans $res[1] tous les noms, dans $res[2] tous les prénoms, dans $res[3] tous les prénoms.
Ainsi pour récupérer les informations du triplet n° $i, on récupère $res[1][$i], $res[2][$i], $res[3][$i]. Ce genre de rangement pour l'information n'est pas très intéressant, car il désolidarise des données qui à l'origine étaient bien organisées.
Ici pour afficher tous les couples il faudra exécuter un code de ce genre :

```
$nb_triplets = sizeof($res[0]); // le nombre d'enregistrements
for ($i=0 ; $i<$nb_triplets ; $i++) {
  $nom = strtoupper($res[1][$i]);
  $prenom = strtolower($res[2][$i]);
  $adresse = $res[3][$i];
  echo "<b>$nom $prenom</b> habite a <i>$adresse</i><br/>";
}
```

Ce qui n'est certes pas complexe, mais assez peu lisible.

Si on applique l'autre ordre ainsi :

```
preg_match_all("/<([a-z\-]+)~([a-z\-]+)~([\w\s\-\r\n]+)>/i", $txt, $res,
PREG_SET_ORDER);
```

Nous obtenons alors un tableau des résultats. Chaque élément de ce tableau contenant 4 éléments : le premier est le triplet, le second le nom, le troisième le prénom, et le quatrième l'adresse. Nous avons alors une structure de données beaucoup plus proche du format initiale. Et surtout quand on prend cherche le triplet n° $i, toutes les informations le concernant sont dans $res[$i] (on n'a pas besoin à tout instant d'avoir le $res entier).
Ici pour afficher tous les couples il faudra exécuter un code de ce genre :

```
foreach ($triplet as $entry) {
  $nom = strtoupper($entry[1]);
  $prenom = strtolower($entry[2]);
  $adresse = $entry[3];
  echo "<b>$nom $prenom</b> habite a <i>$adresse</i><br/>";
}
```

Et bien que ce code ne soit pas spécialement plus court que le précédent je suis sûr que vous le comprenez plus facilement.

Remarque finale concernant les ordres

Je vous ai montré que le 4e paramètre optionnel peut avoir une véritable importance pour simplifier un programme utilisant preg_match_all.
Ce qu'il faut retenir concernant le choix de l'ordre à appliquer est une règles très simple :
- s'il y a des sous-masques, appliquer PREG_SET_ORDER
- sinon ne pas changer l'ordre (donc appliquer PREG_PATTERN_ORDER).

Si vous suivez cette règle vous aurez dans la plupart des cas les résultats les plus simples à utiliser.

4.3. preg_replace / preg_replace_callback

Ces fonctions permettent d'effectuer des remplacements par expressions régulières Perl.

comportement de base de preg_replace

Cette fonction a strictement le même comportement que ses équivalents POSIX.
On peut de la même façon utiliser les sous-masques et *\i* dans la chaîne de remplacement.
Toutefois, cette fonction est plus performante que les fonctions POSIX (c'est une contante entre PCRE et POSIX de toute façon).

l'option 'e' dans preg_replace

C'est là que toute la force des PCRE se fait sentir… L'option 'e' appliqué au masque complet permet de considérer la chaîne de remplacement comme du code PHP qui sera interprété.
Attention, le code PHP doit bien sûr être valide, et la valeur de retour de ce code sera utilisée pour remplacement.
Par exemple si l'on veut remplacer toutes les sous-chaînes entre crochets < > par la sous-chaîne inverse, il faudra appliquer strrev (doc. : http://fr.php.net/strrev) sur chacune des occurrences trouvées. Aucun autre moyen alors que d'utiliser le PHP directement sur l'occurrence trouvée, et donc d'utiliser l'option 'e'.
L'expression de recherche sera ***/<([^>]*)>/e***.
Et la chaîne de remplacement sera strrev(\1). Seulement il ne faut pas oublier les guillemets, car sinon le remplacement de <abc> sera strrev(abc) qui provoquera une *Parse Error*. Il faut donc utiliser en chaîne de remplacement strrev('\1') :

```
echo preg_replace("/<([^>]*)>/e", "strrev('\\1')", $str);
```

Appliqué à **sa<tul> c'est <iom>** ce code affichera **salut c'est moi** sans plus de difficultés.

Cette option est d'une grande utilité, et pourrait permettre d'utiliser une fonction définie par l'utilisateur, qui puisse à la fois renvoyer une chaîne de remplacement (et ainsi effectuer un traitement sur la chaîne), tout en effectuant une insertion dans une base de données, ou n'importe quel autre traitement de haut niveau.

la fonction preg_replace_callback

preg_replace_callback permet de faire le même genre que preg_replace avec l'option 'e', mais elle apparaît plus simple d'utilisation (quoique…).
Le premier argument est une expression régulière, le second est le nom de la fonction de *callback* et le dernier est la chaîne sujet. Il y a un quatrième argument optionnel, qui correspond au nombre maximum d'occurrences que l'on veut traiter.
La fonction de callback doit être une fonction a un argument. L'argument passé sera un tableau, construit exactement de la même façon que le paramètre résultat optionnel de la fonction preg_match : un tableau dont la valeur d'indice 0 est l'occurrence complète, et les valeurs suivantes les sous-chaînes reconnues par les sous-masques.
N'utilisez pas de fonctions nécessitant plus d'un argument, ou vous recevrez des alertes puisque cette fonction ne sera appelée qu'avec un seul argument.

Nous allons utiliser preg_replace_callback dans un exemple concret.
Imaginons que l'on ait le texte **a la claire fontaine m'en allant promener** et que l'on veuille remplacer tous les couples 'a'+un caractère par le caractère+'b'.
On utilisera alors un code de ce type :

```
function remplacer($regs) { // la fonction de callback
  return $regs[1] . 'b';
}

$texte = "a la claire fontaine m'en allant promener";
echo preg_replace_callback("/a(. ?)/", "remplacer", $texte);
```

Ce programme affichera bien **bl bclibre fontibne m'en lblnbt promener** ce qui ne nous avance pas beaucoup mais prouve que la fonction marche ;).

Ce qu'il faut bien comprendre c'est que partout où l'on peut utiliser preg_replace avec l'option 'e', on peut utiliser preg_replace_callback à la place (et vice-versa). Cela dépend principalement de votre sensibilité et personnellement je préfère preg_replace/e parce que j'en ai plus l'habitude, mais je conçois qu'on puisse trouver l'utilisation de preg_replace_callback plus simple, ou en tous cas plus facile à appréhender. Dans les cas complexes, je conseille fortement d'utiliser plutôt preg_replace_callback.

Le premier problème résolut avec preg_replace_callback au lieu de preg_replace/e :

```
function remplacer($regs) { // la fonction de callback
  return strrev($regs[0]);
}
echo preg_replace_callback("/<([^>]*)>/", "remplacer", $str);
```

Le second problème résolut avec preg_replace/e :

```
$texte = "a la claire fontaine m'en allant promener";
echo preg_replace("/a(. ?)/e", "'\\1b'", $texte);
```

Dans cet exemple on pouvait utiliser directement preg_replace en faisant *echo preg_replace("/a(.?)/", "\\1b", $texte)*.

4.4. preg_split

Cette fonction se comporte exactement comme split, à ceci près qu'elle prend en paramètre une expression régulière Perl.
Je vous renvoie donc à la section traitant de split.

4.5. fonctions auxiliaires

Il existe deux fonctions de PCRE qui pourront peut-être vous être utiles :

preg_grep

doc. : http://fr.php.net/preg-grep
Cette fonction prend en paramètres une expression régulière Perl, et un tableau.
Elle va renvoyer le tableau « purgé » des éléments n'étant pas reconnus par l'expression régulière.
Si vous connaissez la commande grep, cette fonction se comporte bien de la même façon.
Exemple :

```
// ne garder que les éléments ne contenant que des chiffres
$resultat = preg_grep("/^\d+$/", $tableau);
print_r($resultat);
```

Ce code affichera la liste des éléments de $tableau n'étant composés que de chiffres.

preg_quote

doc. : http://fr.php.net/preg-quote
Cette fonction prend en paramètre une chaîne de caractères, et un caractère optionnel : le délimiteur.
Le résultat sera la chaîne dont tous les caractères spéciaux utilisés par PCRE, ainsi que le délimiteur s'il est spécifié, ont été échappés.
La chaîne résultat pourra être utilisée au sein d'une expression régulière Perl sans poser de problèmes.
Je vous renvoie à la documentation pour un exemple d'utilisation.

5. Conclusion

5.1. Comparaison entre les différents standards

Texte

5.2. Cas pratiques très fréquents

Texte

5.3. Pour en finir…

Texte