Bash : Variables, globs étendus, ERb, ERe

- Objet : suite de la série de wiki visant à maîtriser bash via les différents caractère spéciaux.
- Niveau requis :
  avisé
- Commentaires : Fix Me!
- Suivi :
  - Création par Hypathie le 10/04/2014
  - Testé par Hypathie Avril 2014
- Commentaires sur le forum : Lien vers le forum concernant ce tuto [1)]

Nota : Contributeurs, les Fix Me! sont là pour vous aider, supprimez-les une fois le problème corrigé ou le champ rempli !

Différents contextes, différentes significations, globs et regexp

La “représentation symbolique”[2)] de caractères alpha-numériques par des “métacaractères”[3)] est de deux types.
Et on peut distinguer ces deux types relativement à l'utilisation que l'on en fait :

1. sélectionner des noms de fichiers dans un répertoire ;
2. déterminer si une chaîne est conforme à un format désiré.

Rappel : les métacaractères

Le premier type de représentation symbolique se fait avec les globs ou patterns simples[4)] ; ils servent aux recherches sur les chaînes de caractères et se font généralement en ligne de commandes.
Il a été donné un simple récapitulatif des caractères utilisés ici : définition usuelle de métacaractère".

Distinguer le "globbing" étendu des expressions régulières

Le deuxième type de de représentation symbolique se fait (à partir de `bash 2.01`) avec les globs étendus [5)], étudiés ci-dessous ; on les utilise pour effectuer des tests de correspondances simples, en ligne de commandes ou dans des scripts, ou pour modifier les paramètres d'une variable.

Distinguer les expressions régulières utilisable avec le shell, de celles d'autres programmes

- Ce deuxième type de représentation symbolique s'est développé avec `Bash 3.0` auquel il a été intégré la possibilité d'une utilisation restreinte des expressions régulières.
  Mais attention, il ne s'agit pas de recherches de noms de fichiers ou de mots (le contexte n'est plus la ligne de commandes), ni de modifications de paramètres.
  On s'en sert uniquement pour la correspondance entre une variable et une expression régulière, jamais pour la substitution ou la correspondance telle que le permet sed et awk.
- On retrouve ce deuxième type de représentation symbolique avec l'usage avancé des expressions régulières.
  Mais elles permettent bien plus que ce que nous allons voir ici. En particulier, la substitution de caractères dans un fichier (exécutable ou non).
  Il est nécessaire pour utiliser les regexp d'en passer par des programmes externes et spécifiques à l'utilisation des expressions régulières (`egrep`, `sed`, `awk` par exemple).
  Ci-dessous l'index 1 donne un simple aperçu des caractères utilisables pour les *ERb* et *ERe* des différents programmes externes au shell Bash, et déborde donc du sujet, si ne n'est que le shell Bash permet de les installer puis de les lancer.
  À voir :
  - les-metacaracteres
  - grep
  - sed

> **Nous allons développer dans cette page le deuxième type de de représentation symbolique, ce qui recouvrira l'utilisation des globs étendus et l'utilisation des expressions régulières spécifiques au shell Bash mais utilisables uniquement dans le contexte de la correspondance avec la chaîne d'une variable.**

Puisque tout est bien clair, c'est parti ! 😎

Correspondance de motifs avec les globs étendus

On trouve dans man bash, à la fin de la page “Développement des noms de fichiers”, “Motifs génériques”, le tableau suivant :

Regroupement	**signification**
?(liste-motif)	zéro ou une occurrence des motifs indiqués
*(liste-motif)	zéro ou plusieurs occurrence des motifs indiqués
+(liste-motif)	une ou plusieurs occurrence des motifs indiqués
@(liste-motif)	une occurrence exactement des motifs indiqués
!(liste-motif)	tout sauf les motifs indiqués les motifs indiqués

Comment se servir des caractères de ce tableau ?
Avec la commande interne de bash “shopt -s extglob”, bash prend en charge les motifs étendus (extended globs).
On peut alors définir plusieurs motifs, séparés par **le caractère |** et regroupés dans des parenthèses.

Le premier caractère placé avant les parenthèses (@, !, +, *, ?), fixe le type de correspondances avec les motifs.
Voir : http://www.linuxjournal.com/content/bash-extended-globbing
Notons l'option shopt -s nocasematch (bash version 3.1) permet de retirer la sensibilité à la case.

Les globs étendus dans le terminal

Préparation

```
mkdir Test && cd Test && touch image.bmp  image.jpg  nom.txt && cd ..
```

> Création d'un fichier de type répertoire nommé “Test”, contenant les fichiers : image.bmp image.jpg nom.txt

Exercice

```
shopt -s extglob
```

> Pour les commandes qui suivent, il faut valider toujours dans le même terminal.

```
ls ~/Test/!(*jpg|*bmp)
```

> Cela signifie : liste-moi le ou les fichiers dont le nom ne comporte pas (`!`)
>
> soit “zéro ou plein de caractères” (*) puis `jpg`
>
> soit “zéro ou plein de caractères” (*) puis `bmp`

retour de la commande

```
/home/hypathie/Test/nom.txt
```

> C'est le chemin absolu (nom complet) du (ou des) autre(s) fichiers que ceux finissant par `jpg` ou `bmp`.

- à propos de cet exemple :

```
cd ~/Test/
```

```
ls !(*jpg|*bmp)
```

retour de la commande

```
nom.txt
```

> C'est le nom simplifié

Les globs étendus dans les scripts

Contexte :

Tester une chaîne par rapport à un motif (représentatif) et non par rapport à une constante littérale. Par exemple :

script

```
#!/bin/bash
shopt -s extglob
nom=image.jpg
if [ "$nom" == *.jpeg ]          # correspondance vue précédemment
then
  echo "bonne correspondance"
  else
  echo "mauvaise correspondance"
fi

nom=image.jpg
if [[ "$nom" = *.@(jpg|jpeg) ]]  # (ligne 12)
                                 # emploi de globs (ou motifs) étendus
: @ voir tableau ci-dessus
then
  echo "bonne correspondance"
  else
  echo "mauvaise correspondance"
fi
```

```
mauvaise correspondance
bonne correspondance
```

Quelques remarques sur la ligne 12.

1. Contrairement à la correspondance simple rappelée au-dessus les doubles crochets sont obligatoires. C'est eux qui enclenchent le mécanisme de comparaison. On peut mettre un double égal, pour plus de clarté.
2. Le “métacaractère” ou glob simple * signifie “un nombre quelconque de caractères” et le ? signifie “un seul caractère”, comme pour les globs simples.
3. CES SIGNIFICATIONS SONT CONSERVÉES lors de l'emploi des globs étendus, et leurs significations sont conservées mais s'appliquent à une syntaxe différente dans les expressions régulières.
4. Ne surtout pas mettre de “ ” autour de * : le shell chercherait alors les chaînes

dont le premier caractère est une *, ni dans les cas simples, ni lorsqu'on utilise les motifs étendus.

Correspondance de motifs avec les expressions rationnelles

L'opérateur de correspondance =~

Bash peut utiliser les expressions régulières mais de façon restreinte ;

- elles ne peuvent pas être utilisées comme modèle de comparaison avec des noms de fichier ou pour effectuer des recherches en ligne de commandes.
- elles ne peuvent pas servir à modifier le contenu d'un fichier.

Elles servent seulement à “matcher” des variables, et elles ne s'utilisent que dans le cadre des scripts.

Depuis Bash 3.0, on peut pour se faire utiliser l'opérateur **=~**.

Cet opérateur **=~** permet :

- de vérifier la correspondance entre une chaîne (valeur d'une variable à gauche) et le modèle regex à droite ;
- ou encore de vérifier la correspondance entre une variable (constituée d'une chaîne de caractères littérales), et une variable constituée d'une regex.

À savoir :
Lorsque la chaîne correspond au motif, le code retour du test renvoie 0 pour vrai, sinon, il renvoie 1 pour faux.
Si la syntaxe du pattern n'est pas valide le code de retour est 2.

Lorsqu'une chaîne correspond, chacune des parties du motif est disponible dans la variable `$BASH_REMATCH`.

Caractères servant aux expressions régulières de correspondance avec une variable

Encore une fois, pour le shell bash, tous les caractères ci-dessous, ont la signification décrite uniquement dans ce contexte de correspondance de motif entre variable et *ERb* ou *ERe*.

Ne pas confondre la signification de certains des caractères présentés par ce tableau, avec celle qu'ils ont pour le shell dans le contexte de la ligne de commandes.

Le tableau rappelle pour cette raison, la signification des caractères pour lesquels la confusion est possible avec leur utilisation en ligne de commandes[6)].

Voici les caractères utilisables :

“signes” regex ERE basique	**significations (comparez avec le tableau de l'index 1)**
.	Correspond à tout caractère unique. (Attention en ligne de commandes le point représente le répertoire courant.)
*****	Correspond à zéro ou plusieurs fois l'élément précédent. Par exemple, `ab*  c` correspond à “ac”, “abc”, “abbbc», etc. Ou encore, `[xyz]*` correspond à x, y, z, zx, zyx, xyzzy, et ainsi de suite. `(ab)*` correspond à a, b, abab, ababab, et ainsi de suite. (Attention en ligne de commandes, ***** signifie “tout” : `rm blabla*` → supprime tout ce qui commence par blabla dans le répertoire courant)
****	Échappement du caractère spécial. Par exemple **\.** sélectionne un point littéral. (Attention en ligne de commandes, **** permet de couper les longues commandes et signifie “la ligne de commandes se termine à la ligne suivante”.)
^	Correspond à la position de départ dans la chaîne.
$	Correspond à la position de fin de la chaîne ou la position juste avant un saut de ligne de chaîne interminable.
[ ]	Correspond à un seul caractère qui est contenue dans `[ ]`. On peut mélanger [abcx-z] correspond à a, b, c, x, y ou z, de même que `[a-cx-z]`. Le caractère `-` est traité comme un caractère littéral si c'est le dernier ou le premier. (Même signification ici que **[ ]** et **[-]** en tant que métacaractère d'une ligne de commandes.)
[^]	Correspond à un caractère qui n'est pas contenu dans les parenthèses. Par exemple, `[^abc]` correspond à tout caractère autre que a, b ou c.
[:class:] les classes prédéfinies	`[:alnum:] [:alpha:] [:blank:] [:cntrl:] [:digit:] [:graph:] [:lower:] [:print:] [:punct:] [:space:] [:upper:] [:xdigit:]`
\(\) et **\{ \}**	Avec les ERb le caractère \ donne un sens particulier aux parenthèses et accolades. (anciennes versions de Bash) Mais avec les ERe le caractère \ échappe la signification spéciale des parenthèses et des crochets
Caractères des ERe reconnus depuis Bash 3.0	**Significations**
?	Correspond zéro ou une fois à le regroupement précédent. Par exemple, `[er]?` correspond à la sous-chaîne “er” pouvant être présente ou pas dans une chaîne. (*Attention en ligne de commandes **?** signifie “un caractère quelconque et un seul”*)
+	Correspond une ou plusieurs fois à le regroupement précédent. Par exemple, `[er]+` correspond à `er`, `erer`, `ererer`, et ainsi de suite, mais pas être absente.
|	Alternative `er1|er2|er3`. Par exemple, `abc|def` correspond à `abc` ou `def`.

“signes” regex ERE basique	**significations (comparez avec le tableau de l'index 1)**
Caractères ERe pouvant servir à élaborer des sous-chaînes	**Significations**
( )	Regroupement
{m,n}	l'élément précédent correspond au moins à m fois, mais pas plus de n fois . Par exemple, a{3,5} correspond uniquement aaa, ou aaaa, ou aaaaa.
{ }	“exactement trois fois” le caractère ou le regroupement par exemple a{3} correspond strictement à aaa ; **[er]{2}** deux fois “er”.
{n,}	“n” fois ou plus, le caractère ou le regroupement précédent. Par exemple, {3,} “trois fois ou plus” correspond aaa ou aaaa, etc. (équivalent de a*)
{,n}	au plus “n” fois, le caractère ou le regroupement précédent. [ER]{,3} “jusqu'à trois fois” groupe de ER.

- Référence (ce tableau résume le contenu de ces trois liens) :
  - https://en.wikipedia.org/wiki/Regular_expression#Standards
  - https://en.wikipedia.org/wiki/Regular_expression#Standards#Uses
  - https://en.wikipedia.org/wiki/Regular_expression#Standards#Character%20classes
- Autre référence :
  http://mywiki.wooledge.org/RegularExpression
  Regular Expressions/POSIX

Apprenons à construire une expression régulière

script

```
#!/bin/bash
for nombre in "1234567" "123478985" "123498761" "12396590"
do
  if [[ $nombre =~ ^[0-9]{9}$ ]]  # 9 nombres ({ }) compris entre 0 et 9 ([0-9])
  then
      echo "$nombre comporte 9 nombres"
  else
      echo "$nombre ne correspond pas à 9 nombres"
  fi
done
```

```
1234567 ne correspond pas à 9 nombres
123478985 comporte 9 nombres
123498761 comporte 9 nombres
12396590 ne correspond pas à 9 nombres
```

- Ou encore :

script

```
#!/bin/bash
#REGEX="^[[:upper:]]{1}[[:lower:]]{4}$"
REGEX="^[A-Z]{1}[a-z]{4}$"
var=Hello
if [[ $var =~ $REGEX ]]
then
echo "match"
else
echo "pas de match"
fi
```

```
match
```

Compliquons un peu en utilisant plusieurs autres caractères du tableau ci-dessus :

script

```
#!/bin/bash
regex="^([[:alpha:][:blank:]]*)- ([[:digit:]]*) -
([[:alpha:]]?)(.*)jpg$"
# ou regex="^([[:alpha:][:blank:]]*)- ([[:digit:]]*) -
([[:alpha:]]?)(.*)[a-z]{3}$"
#ou encore regex="^([[:alpha:][:blank:]]*)- ([[:digit:]]*) -
([[:alpha:]]?)(.*)[a-z]\3$"
var="image linux - 01 - pingouin.jpg"

if [[ $var =~ $regex ]]
then
echo "Le nom de l'image correspond à l'expression rationnelle."
else
echo "mauvaise regex"
fi
```

```
Le nom de l'image correspond à l'expression rationnelle.
```

- Explication :

> ^ : début de l'expression
>
> `([[:alpha:][:blank:]]*)` :
>
> entre parenthèses: première sous-expression,
>
> avec une paire de crochets contenant deux autres paires de crochets `[:apha:]` et `[:blank:]`,

avec * pour signifier que le groupe [alpha et blank] doivent apparaître 0 ou plusieurs fois ; suivi d'un espace.

`-` : un tiret avec un espace après comme dans l'expression littérale.

`([[:digit:]]*) - ([[:alpha:]]?)` : une sous expression faite d'un groupe composé d'un nombre quelconque d'alphanumérique, un espace, un tiret, une autre sous-expression qui apparaît

0 ou 1 fois (?).

La sous-expression `(.*)` signifie n'importe quel nombre (*) de tous caractères (`.`),

puis littéralement `jpg`,

puis $ qui signifie fin de l'expression.

Le tout entre ” “ et sans espace autour du égal qui affecte la variable “regex” par la *ERE*[7].

Et voilà comment avec le shell bash, on peut dresser une expression rationnelle étendue fonctionnant dans les tests ! 😀

Pour résumer :

- Les *expressions régulières* utilisées avec le shell Bash nécessitent l'opérateur =~
- On retrouve dans ce contexte les caractères génériques (globs) basiques * ? mais avec un sens différent.
- On retrouve dans ce contexte les caractères génériques étendus |, @, !, +, ^ (là avec un sens différent et une nouvelle syntaxe).
- Les nouveaux caractères sont + , | , [-] , { , } , ( ) , \n, ainsi que [ ]{n} et [:class:].
- Elles ne s'utilisent jamais avec la syntaxe de correspondance ou de substitution de type : `$var/.../...` ou `$var//.../...`.
- Elles s'utilisent uniquement pour la correspondance avec des variables.
- Les caractères spécifiques aux expressions régulières (*ERb* et *ERe*) utilisées ici uniquement dans le contexte de correspondance avec une variable, sont identiques et ont une signification similaire lorsqu'ils sont utilisés avec d'autres programmes, grep, grep -E, sed, sed -r, awk, mais dans un contexte beaucoup plus large.
  (comparez avec l'index1).

Un script pour s'exercer au "ER"

- Voici un script “exp.reg1”

script exp.reg1

```
#!/bin/bash
#Les "echo" les plus à droite sont là pour expliquer lors du retour ce qui s'y passe dans ce script.

VAR="$1"
     echo "La valeur de VAR est: $VAR."
     echo "Il y a "$#" paramètres."
     echo "Le paramètre n°1 est "$1" (la 'ER')."
     echo "Le paramètre n°2 est "$2" (deuxième argument, chaine1 à matcher)."
     echo "Le paramètre n°2 est "$3" (le troisième argument, chaine2 à matcher."
     echo " "
shift
     echo "Avec 'shift', on se décale d'un paramètre."
     echo "Après 'shift', il reste donc: "$#" paramètre(s)."
     echo "Et ce(s) paramètre(s) sont: "$1", "$2"."
     echo "("$1": ancien deuxième paramètre devenu paramètre 1 après shift)."
     echo "("$2": ancien troisième paramètre devenu paramètre 2 après shift.)"
     echo " "
     echo "MAIS LA VALEUR DE VAR RESTE la 'ER' : "$VAR"."
for i in "$@"
do
     echo " "
     echo "'for i in \$@': la variable i aura, boucle après boucle, les VALEURS: "$@","
     echo "(attribués à chaque tour de boucle à variable "i".)"
     echo "c'est-à-dire lors de la boucle n°1, elle est identique au paramètre n°1: "$1"."
     echo "puis lors de la boule n°2, elle est identique au paramètre n°2 : "$2"."
     echo " "
     echo "On peut donc donner à grep la chaine:$i comme entrée par le tube,"
     echo "et comme motif le 'ER': $VAR."
 echo "$i" | grep -E "$VAR" > /dev/null
  if [ $? -eq 0 ]
  then
    echo " "
    echo " BRAVO ! La ER: $VAR  correspond au motif "$i" "
  else
    echo " "
    echo " ERREUR ! La ER: $VAR ne correspond pas au motif: $i "
    echo " "
  fi
done
```

```
# ligne 33 (if) : $? (code de retour) -eq (égal à) zéro (pas d'erreur de sortie, donc bonne correspondance)
```

- Lancez-le comme ci-dessous, explications détaillées dans le retour :

```
./exp.reg1 "^[a-b]" "abc" "ABC"
```

> La ER comme premier argument de “exp.reg1”, puis le ou les chaînes à vérifier en second, troisième, etc. argument de “exp.reg1”.

Et voilà 🙂

Un grand merci à captnfab pour conseils avisés et toutes ses corrections.

INDEX 1 : caractères des ERb et ERe avec grep, grep -E, sed, sed -r et awk

Comparez ce qui suit avec Caractères servant aux expressions régulières de correspondance avec une variable.

tableau ERb et ERE par commandes

caractères	[...]	.	*	^	$	?	+	|	( )	[:class:]
grep	ok	ok	ok	ok	ok	-	-	-	-	-
grep -E	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok [[:class:]]
sed	ok	ok	ok	ok	ok	-	-	-	-	-
sed -r	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok [[:class:]]
awk	ok	ok	ok	ok	ok	ok	ok	ok	ok	ok [:class:]

détail des classes

- Avec sed il faut l'option -r, bien que les classes relèvent des *ERb*.
- Avec grep, elles ne nécessitent pas l'option -R ou egrep.

`[[:alnum:]]`	Alpha-numerique [a-z A-Z 0-9]
`[[:alpha:]]`	Alphabetic [a-z A-Z]
`[[:blank:]]`	Espaces ou tabulations
`[[:cntrl:]]`	Caractères de contrôle
`[[:digit:]]`	Nombres [0-9]
`[[:graph:]]`	Tous les caractères visibles (à l'exclusion des espaces)
`[[:lower:]]`	Lettres minuscules [a-z]
`[[:print:]]`	Caractères imprimables (tous caractères sauf ceux de contrôle)
`[[:punct:]]`	Les caractères de ponctuation
`[[:space:]]`	Les espaces
`[[:upper:]]`	Les lettres majuscules [A-Z]

`[[:xdigit:]]`	Chiffres hexadécimaux [0-9 a-f A-F]

man grep fr

egrep en

man sed fr

man awk

ERb et ERe : tableaux récapitulatifs

Récapitulatif selon deux types d'expressions régulières :

-les expressions régulières basiques ERb
-les expressions régulières étendues ERe

caractères communs au ERb et ERe

expressions	**Modèles reconnus**
c	Tout non métacaractère c.
\	Échappement du caractère spécial. Par exemple **\.** sélectionne un point littéral.
^	Test effectué au début de la chaîne.
$	Test effectué à la fin de la chaîne.
.	Tout caractère sauf une fin de ligne.
*****	Zéro à n chaînes consécutives validées par l'expression régulière r.
\<	début d'un mot (caractères pouvant faire partie de [A-Z-z0-9]
\>	Fin d'un mot
[liste_de_caractères]	Un caractère cité dans la liste
[^liste_de_caractères]	Un caractère qui n'est pas dans la liste

- Exemples :

blabla : chaîne contenant "blabla"

^blabla : chaîne commençant par "blabla"

blabla$: chaîne finissant par "blabla"

^[A-Z][5-8].$: chaîne composées de trois caractères, dans l'ordre :
une majuscule, un chiffre compris entre 5 et 8, un caractère quelconque

^$: chaîne vide

^[ ]*$: chaîne contenant zéro ou plusieurs espace(s) ou tabulation(s)

\<tout : mot commençant par "tout", toutefois

\<tout\> : le mot “tout”

Caractères spécifiques aux ERb

expressions	Modèles reconnus
\{m\}	m fois le caractère précédent
\{m,\}	au moins m fois le caractère précédent
\{m,n\}	entre m et n fois le caractère précédent
\(ERb\)	mémorisation d'une ERb
\1, \2, ...	Rappel de mémorisation

→ Le caractère **** donne une signification spéciale aux parenthèses et accolades, au lieu de les rendre littérales.

Caractères spécifiques aux expressions rationnelles étentues (ERe)

Excepté **(er1)(er2)** :

- Tous sont utilisables avec grep -E et egrep
- Tous sont utilisables avec awk
- Certains posent problème avec sed -r

expressions	Modèles reconnus
?	zéro ou une fois le caractère ou le regroupement précédent
+	une à n fois le caractère ou regroupement précédent
{m}	m fois le caractère précédent
{m,}	m fois le caractère précédent
{m,n}	entre m et n fois le caractère précédent
(er1)	regroupement
er1|er2|er3	alternative
(er)+	Une ou plus de une chaîne(s) consécutive(s) validée(s) “er”.
(er)*	zéro ou plus de zéro chaîne(s) consécutive(s) validée(s) par “er”
(er)?	une chaîne bulle ou toute chaîne validée par “er”.
[c1c2...]	Tout caractère expressement listé entre les crochets.
[^c1c2...]	Tout caractère excepté ceux qui sont expressément listés entre les crochets.
[c1-c2]	Tout caractère appartenant à l’intervalle c1 c2, bornes comprises.
er1|er2	Toute chaîne de caractères validée soit par er1 soit par er2.
(er)(er)	Toute chaîne validée par l’expression er, chaîne vide exclue.
(er1)(er2)	Toute chaîne de caractères de type AB, dans laquelle l’expression régulière er1 valide A et l’expression régulière er2 valide B. Avec awk uniquement

Les raccourcis ne sont pas posix

- sed :

Séquences	**Séquences**
\f	Produit ou correspond à un saut
\n	Produit ou correspond à un retour à la ligne
\t	Produit ou correspond à un onglet horizontal
\v	Produit ou correspond à une tabulation verticale
\w	Synonyme de **[[:alnum:]]** → correspond à un mot.
\W	Synonyme de **[^[:alnum]]** → ce qui autre qu'un mot.
\b	Correspond à une chaîne vide (blanc) à l'extrémité d'un mot

- awk :

Séquences	**Séquences**
\b	backspace (supprime le dernier caractère d'une chaîne)
\f	formfeed (nouvelle page)
\r	carriage return (retour à la ligne)
\t	tabulation (crée une tabulation de dix espaces)
\c	tout caractère pris sous sa forme littérale excepté \

Pour aller plus loin

Pour les PCRE (perl) supportée par egrep (grep -E ; sed et et awk) voir pcresyntax(3) et [[http://manpages.courier-mta.org/htmlman3/pcresyntax.3.html|pcrepattern(3)]] - Lien Obsolète

Et en français http://www.expreg.com/pcre.php

INDEX 2 : REGEXP Perl

Un petit rappel non exhaustif des caractères spéciaux qui leur sont communs.

Correspondance :	`m/motif/` `/motif/`
Substitution :	`s/motif/chaîne/`
Correspondance entre regex et variable :	=~ $v =~ m/toto/ $v =~ s/toto/titi/
Les caractères spéciaux :	`\ | ( ) [ ] { } ^ $ * + ? .`
Le point représente n'importe quel caractère.	`.`
La paire de crochet “matche” l'un des caractères entre crochet	`[ ]`

Intervalle : Tout intervalle est envisageable, par exemple `u-w` ou toute autre combinaison tant que le numéro ASCII du premier caractère est inférieur à celui du second. Un intervalle peut prendre place au milieu d'un motif quelconque. Pour rechercher un `-` littéral, le mettre en dernier dans un intervalle.	`[a-z]` (l'une des lettres minuscules de l'alphabet) `[A-Z]` (l'une des lettres majuscules de l'alphabet) `[0-9]` (un des caractères numériques)

Raccourcis pour des ensembles courants	`\d` qui correspond à `[0-9]`
	`\D` qui correspond à `[cfl0-9]`
	`\w` qui correspond à `[a-zA-Z0-9_]`
	`\W` qui correspond à `[cfla-zA-Z0-9_]`
	`\t` qui correspond à une tabulation
	`\n` qui correspond à un saut de ligne
	`\n` qui correspond à un saut de ligne
	`\r` qui correspond à un retour chariot
	`\s` qui correspond à un espace blanc
	`\S` qui correspond à n'est pas un espace blanc
	`\t` qui représente une tabulation
	`\f` qui représente un saut de page
	`\e` qui représente un échappement

Quantificateurs :
 	
0 fois ou plus
1 fois ou plus
0 ou 1 fois
n fois exactement
	
	`*`
	`+`
	`?`
	`{n}`

Mémorisation : de variables $1 $2 ...	`( ) ( )` ...
Exemple: `prenom.nom@domaine.ext` (\w+) : $1 (\w+) : $2	 `/\b(\w+)\.(\w+)@\w+\.w{2,4}\b/`
Substitution de variables :	`$s = “toto”;` `if( $v =~ m/$s/ ) { … }`
Options :	`i` : Rend le motif insensible à la case (minuscules/majuscules) l'expression régulière `m/toto/i` `g` : Permet d'effectuer toutes les substitutions, et pas que la première. `e` : Évalue le membre de droite comme une expression Perl. `o` : La compilation a lieu une seule fois lors de la première exécution.

- Référence :
  http://oreilly.com/php/excerpts/php-mysql-javascript/regex-in-php-javascript.html
- Apprendre à utiliser les expressions régulières sous perl :
  https://ensiwiki.ensimag.fr/index.php/Expressions_r%C3%A9guli%C3%A8res
  http://perldoc.perl.org/perlrequick.html

http://perldoc.perl.org/perlre.html

INDEX 3 : l'ordre des caractères ASCII

Voici :

ordre	caractère	ordre	caractère	ordre	caractère	ordre	caractère	ordre	caractère	ordre	caractère
0	NULL	21	NAK	42	*	63	?	84	T	105	i
1	SOH	22	SYN	43	+	64	@	85	U	106	j
2	STX	23	ETB	44	,	65	A	86	V	107	k
3	ETX	24	CAN	45	-	66	B	87	W	108	l
4	EOT	25	EM	46	.	67	C	88	X	109	m
5	ENQ	26	SUB	47	/	68	D	89	Y	110	n
6	ACK	27	ESC	48	0	69	E	90	Z	111	o
7	BEL	28	FS	49	1	70	F	91	[	112	p
8	BS	29	GS	50	2	71	G	92	\	113	q
9	HT	30	RS	51	3	72	H	93	]	114	r
10	LF	31	US	52	4	73	I	94	^	115	s
11	VT	32	space	53	5	74	J	95	-	116	t
12	FF	33	!	54	6	75	K	96	'	117	u
13	CR	34	”	55	7	76	L	97	a	118	v
14	SO	35	#	56	8	77	M	98	b	119	w
15	SI	36	$	57	9	78	N	99	c	120	x
16	DLE	37	%	58	:	79	O	100	d	121	y
17	DC1	38	&	59	;	80	P	101	e	122	z
18	DC2	39	'	60	<	81	Q	102	f	123	{
19	DC3	40	(	61	=	82	R	103	g	124	|
20	DC4	41	)	62	>	83	S	104	h	125	}

ordre	caractère
126	~
127	DEL

tuto précédent :

Bash : les caractères de transformation de parametres

1)
N'hésitez pas à y faire part de vos remarques, succès, améliorations ou échecs !
2)
en anglais : “matching”
3)
termes employé là dans un sens général : http://fr.wiktionary.org/wiki/m%C3%A9tacaract%C3%A8re
4)
désignés aussi de “caractères génériques”
5)
appelés aussi “patterns longs” ; en anglais “extended patterns”

6)

avec echo, ls, rm, etc.

7)

expression rationnelle étendue