

Tables de hachage, regexps



Sylvain Lhullier
contact@formation-perl.fr
<https://formation-perl.fr/>

La reproduction et/ou diffusion de ce document, même partielle, quel que soit le support, numérique ou non, est strictement interdite sans autorisation écrite des ayants droit.



Plan des 4 premiers cours Perl

Introduction	Tables de hachage
Prise en main	Expressions régulières
Scalaire	Fichiers, entrée/sorties
Structures de contrôle	Références
Listes et tableaux	Modules
Fonctions et programme	Programmation objet
Fonctions sur les listes	



Table de hachage : principe

Tableau associatif : association clef → valeur

Clef : chaîne de caractères Valeur : scalaire

Dans la documentation : *hash*

%h	
Virginie	=> 06.01.52.12
Paul	=> 01.45.32.63
Pierre	=> Heu?...

Principes :

- unicité des clefs
- avec la clef, on obtient la valeur (inverse non vrai)
- pas d'ordre connu : optimiser l'accès à un élément



Table de hachage : déclaration

- Table de hachage vide :

```
my %h;  
my %h = (); # Attention ! () et non {}
```
- Déclaration et initialisation :

```
my %h = ('Paul'=>'01.45.32.63',  
        'Virginie'=>'06.01.52.12',  
        'Pierre'=>'Heu?...');
```



Table de hachage : accès, ajout ou modification

- `my %h = ();`
`$h{'Jacques'}` valeur associée à la clef Jacques
`if($h{'Jacques'} eq 'Heu?...') { ... }`
`print $h{'Jacques'}."`
`$h{'Jacques'} = '04.43.72.34'; #Ajout couple clef/valeur`
`# ou modification de valeur`
- Clef stockée dans une variable :
`$k = 'Jacques';`
`print "$h{$k}\n";`
`$v = '04.43.72.34';`
`$h{$k} = $v; # Ajout couple clef/valeur`
`# ou modification de valeur`



Table de hachage : règles de nommage des clefs

- Cas général
 \Rightarrow La clef doit être placée entre quotes " ou '
`%h = ("f v"=> 21, '3@f'=> "toto");`
`$h{"fs-v2"} = 34;`
`$h{'a9='} = 'info';`
- Cas d'une clef $\in [a-zA-Z_-][0-9a-zA-Z_-]^*$
 \Rightarrow Pas besoin de quotes
`%h = (Jacques => '04.43.72.34');`
`$h{Jacques} = '04.43.72.34';`



Table de hachage : (in)existence d'une clef

- Clef inexistante : valeur `undef`
`$x = $hash{inexistant}; \Rightarrow $x vaut undef`
 Mais une clef existante peut avoir `undef` comme valeur ...
- Tester l'existence d'une clef ? `exists($hash{clef})`
 Renvoie vrai ou faux selon l'existence du couple clef/valeur
`if(exists($h{Marie})) {`
`print "Contacter Marie : $h{Marie}\n";`
`}`
`if(exists($h{$clef})) { ... }`
- Ne pas faire `defined($hash{clef})` car teste la valeur à `undef`



Table de hachage : suppression d'un couple clef/valeur

- `delete($hash{clef})`
 Supprime un couple clef/valeur si la clef existait
 (ne fait rien si elle n'existait pas)
`delete($h{Marie}); # Ne plus pouvoir contacter Marie`
`delete($h{$clef});`
- Ne pas faire `$hash{clef} = undef`
 Créé (modifie) une clef avec pour valeur `undef`
 que l'on retrouvera lors d'un parcours de la table de hachage
 avec `keys/values/each` (lire la suite)



Table de hachage : parcours

keys : liste des clefs

- `@t = keys(%h);`
 \Rightarrow `@t = ('Virginie', 'Pierre', 'Paul');`
 L'ordre est quelconque.
- ```
foreach my $k (keys %h) {
 print "$k : $h{$k}\n";
}
```

Virginie : 06.01.52.12  
 Pierre : Heu?...  
 Paul : 01.45.32.63



## Table de hachage : parcours (suite)

**values** : liste des valeurs

- `@t = values(%h);`
- ```
foreach my $v ( values %h )
{ print "$v\n"; }
```

\Rightarrow Obtention de la clef difficile

each : itération sur les couples (clef,valeur)

- ```
while(my ($k,$v) = each(%h))
{ print "$k : $v\n"; }
```

**values** et **each** peu utilisés



## Table de hachage

- **%ENV** : variables d'environnement  
`$ENV{PATH} = '/bin:/usr/bin';`  
`$ENV{HOME}`
- `%h = ( a=>1, a=>2 );`  
 La clef `a` aura pour valeur `2` (la dernière valeur l'emporte).
- Contexte scalaire : nombre de clefs  
`$n = %h;`  

```
if(%h) { }
while(%h) { }
```

Jusqu'en 5.24 : nb d'alvéoles occupées/nb total d'alvéoles



## Table de hachage et liste

Conversion vers une liste (contexte de liste) :

- `@t = %h;`  
 $\Rightarrow$  `@t = ('Virginie', '06.01.52.12', 'Pierre', 'Heu...', 'Paul', '01.45.32.63');`  
 L'ordre des couples (clef,valeur) est quelconque.

Conversion depuis une liste :

- `@t = ('Paul', '01.45.32.63', 'Virginie', '06.01.52.12', 'Pierre', 'Heu...');`  
`%h = @t;`
- `%h = ('Paul', '01.45.32.63', 'Virginie', '06.01.52.12', 'Pierre', 'Heu...');`  
 Usage de la virgule ou la double-flèche `=>`
- `%h = qw(a 1 b 2);`



## Table de hachage : ordre préservé

Si la table de hachage n'est pas modifiée, l'ordre est garanti :

```
%h = (a=>1, b=>2, c=>3);
```

- keys %h            c, a, b
- values %h        3, 1, 2
- @t = %h        c,3,a,1,b,2

```
%h = (nom=>'Durand', prenom=>'Paul', age=>35);
insert into personne(keys %h) values (values %h)
```



## Table de hachage : affectations et parcours

```
%h2 = %h;
```

La table de hachage %h2 est une copie conforme de %h

Perte de toutes les anciennes valeurs de %h2

```
%h2 = reverse %h;
```

Les clefs seront les valeurs.

Les valeurs seront les clefs (transformées en chaînes).

Attention à l'unicité des valeurs.

Que font ces deux boucles ?

```
foreach my $e (%h) { print $h{$e}; }
foreach my $e (keys %h) { print $h{$e}; }
```



## Table de hachage : autovivification

Accéder à une clef inexistante ⇒ création avec valeur undef

Exemple : \$hash{\$clef}++    création avec valeur 1

```
my %cpt = ();
```

```
foreach my $m (@mots) {
 if(not exists $cpt{$m}){
 $cpt{$m}=1;
 } else {
 $cpt{$m}++;
 }
}
```

```
foreach my $mot (keys %cpt) {
 print "Le mot '$mot'";
 print "est inclus $cpt{$mot} fois\n";
}
```

→ Automagiquement !

```
foreach my $m (@mots) {
 # Autovivification :
 $cpt{$m}++;
}
```

| %cpt    |       |
|---------|-------|
| le      | => 15 |
| conseil | => 2  |
| pour    | => 4  |



## Table de hachage et appel de fonction

Comme pour les tableaux :

- Appel : par copie (aplatissement)
 

```
action(%h);
sub action { my %v = @_; /..../ } # Ok !
action(%h, %h2); # Aplatissement
sub action { my (%v,%v2) = @_; /..../ } # Ko !
```

- Retour : par copie (aplatissement)
 

```
sub action { /..../ return %h; }
%v = action(); # Ok !
sub action { /..../ return %h, %h2; } # Aplatissement
(%v,%v2) = action(); # Ko !
```



## Création de hash avec map

Syntaxe connue :

```
@s1 = map { fonction($_) } @t;
```

```
@s2 = map { $_, fonction($_) } @t;
```

Un élément donne  
deux éléments.

```
%h = @s2;
```

```
%h = map { $_ => fonction($_) } @t;
```

⇒ Cache de calcul d'une fonction

```
$compteur = 0;
```

```
%h = map { $_ => $compteur++ } @t;
```

```
%config = map { split /\s/, $_, 2 } @lignes;
```

|   |   |   |   |   |   |
|---|---|---|---|---|---|
| 0 | a | 1 | b | 2 | c |
| 0 | 1 | 1 | 2 | 2 | 3 |

|   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | a | 1 | 1 | 2 | b | 3 | 2 | 4 | c | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|

|   |    |   |
|---|----|---|
| c | => | 3 |
| a | => | 1 |
| b | => | 2 |



## Tri : manœuvre or-cache

But : accélérer un tri effectué selon un calcul qui coûte cher.

Idée : effectuer une seule fois le calcul pour chaque élément.

```
@fruits = ('pomme', 'groseille', 'melon', 'kiwi');
```

```
@fruits = sort { length($a) <=> length($b) } @fruits;
```

```
my %cache;
```

```
@fruits = sort { ($cache{$a} //= length $a) <=>
```

```
($cache{$b} //= length $b) } @fruits;
```

```
$cache{fruits} //= length fruits
```

```
$cache{fruits} = $cache{fruits} // length fruits
```

Limitations : le calcul doit se faire sur un seul élément,

2 éléments différents doivent être représentés par 2 chaînes différentes

Inconvénients, à chaque comparaison : 2 tests et affectations

Amélioration : Transformée schwartzienne (avec les références)



## Tranches de tableau ou de hash

Tableau

• @jours ⇔ (\$jours[0], \$jours[1] .... \$jours[n])

• @jours[2,4] ⇔ (\$jours[2], \$jours[4])

• @jours[2..4] ⇔ (\$jours[2], \$jours[3], \$jours[4])

Table de hachage %mois

• @mois{'jan', 'nov'} ⇔ (\$mois{'jan'}, \$mois{'nov'})

Usage :

```
• @jours[2,4] = ('a', 5);
```

```
• ($uid, $gid, $size) = (stat('/'))[4,5,7];
```

```
• @mois{'jan', 'nov'} = (31, 30); # @mois{qw(jan nov)} =
```

```
• @mois{@clefs} = @valeurs;
```



## Types de données

| structure | globalité | élément     | tranche     |
|-----------|-----------|-------------|-------------|
| Scalaire  | \$v       |             |             |
| Tableau   | @v        | \$v[indice] | @v[indices] |
| Hash      | %v        | \$v{clef}   | @v{clefs}   |

NB: Le sigil indique le type de l'expression.



## Questions pour un tri

### 1. Qu'est-ce que je trie ?

Ici : clefs de la table de hachage (noms de mois), car **keys**  
 ⇒ **\$a** et **\$b** sont des noms de mois (nom de variable de boucle)

### 2. Comment je trie ?

Ici : nombre de jours (numérique croissant)

### 3. Comment je passe de l'un à l'autre ?

Ici : comment je passe d'un nom de mois à un nombre de jours ?

```
$mois $jours{$mois}
```

```
$a $jours{$a}
```

```
$b $jours{$b}
```

Numérique ? Spaceship <=>

Croissant ? **\$a** avant **\$b**

```
foreach my $mois (sort { $jours{$a} <=> $jours{$b} } keys %jours) { }
```



## Regexp : introduction

Anglais : *regular expression* (*regexp*)

Trad. mot-à-mot : *expression régulière* Trad. correcte : *expression rationnelle*

*Pattern matching* = correspondance de motif

On retrouve les regexp du shell (**grep sed awk**) très augmentées

Natives en Perl : intégrées au langage ⇒ facilité de manipulation

En Perl, utilisées dans **split** et **grep**.

Perl est LA référence dans le monde des regexp (PCRE, preg) :

nombreux langages (C, PHP, JAVA...) et outils (Postfix, Apache...)

*Certaines personnes, face à un problème, pensent «Je sais, je vais utiliser les regexp.» Maintenant elles ont deux problèmes.* – Jamie Zawinski

Explorateur interactif : <https://regex101.com/>

Jeu de mots croisés regexp : <https://regexcrossword.com/>



## Regexp : fonctionnalités

- Correspondance (*matching*) : **m**

```
m/regexp/modifieurs
```

```
/regexp/modifieurs
```

```
if($v =~ m/^toto/) { ... } Vrai si $v matche le motif.
```

- Substitution : **s**

```
s/regexp/chaîne/modifieurs
```

```
$v =~ s/toto/titi/g; Substitue toto en titi dans $v.
```

(le reste est inchangé)

Séparateur **! " # \$ % & ' ) \* + , - . / : ; = ? @ \ ] ^ \_ { | ~**  
**m@regexp@modifieurs** (**m** obligatoire) **s:regexp:chaîne:modifieurs**

Couple séparateur : **s{regexp}{chaîne}modifieurs s()() m[]**



## Regexp : bind

Attacher une regexp à une variable : **=~ !~**

- if( \$v =~ m/voiture\$/ ) {...}**

- if( \$v !~ m/voiture\$/ ) {...}**

⇔ **if( !( \$v =~ m/voiture\$/ ) ) {...}**

- \$v =~ s/voiture/pieds/; # Pas de if !!**

- Sinon sur **\$\_**



### Regexp : caractères

- Méta-caractères : \ | ( ) [ ] { } ^ \$ \* + ? .  
à protéger avec \    Idem pour le séparateur choisi
- . un caractère quelconque (sauf \n)  
/t.t./ → tata t,tK tot9 ...
- Autre caractère : leur valeur exacte  
m/a/ → tous les séquences qui comportent a matchent  
m/toto/ → tous les séquences qui comportent toto matchent  
m/f 2/ → l'espace vaut pour lui-même
- \n saut de ligne    \r retour chariot    \e échappement  
  \t tabulation    \f saut de page

Q: `if( $v =~ m/\$/ ) { ... }`

Q: `$v =~ s/t.\\/\\\/hooo!;`



### Regexp : assertions

Ne consomme aucun caractère. Correspond à une position.

- ^ début de chaîne  
/^a/ → commençant par a
- \$ fin de chaîne  
/a\$/ → finissant par a
- \b limite de mot (boundary)  
/\bFred\b/ → correspond à Fred  
mais pas à Frederic ni à alFred

Il en existe d'autres ...



### Regexp : classes

- [*chaîne*] un caractère de la chaîne  
[bpm] → b, p ou m            [bpmb] ⇔ [bpm]  
/[bcr]at/ → bat cat rat
- chaîne avec intervalles : a-z A-Z 0-9 f-v 4-8 ...  
/tot[a-z]/ → tota totb ... totz  
/tot[a-zA0-9]/ → tota ... totz totA tot0 ... tot9
- Si - doit être dans la classe :    [*chaîne*-]  
/tot[a-z09-]/ → tota ... totz tot0 tot9 tot-
- ^ en début de chaîne : complément de la classe  
[~*chaîne*] un caractère absent de la chaîne  
[~ao] → tout caractère sauf a et o  
[~0-9] → tout caractère non numérique



### Regexp : quantificateurs

Ils s'appliquent à l'expression précédente

- \* doit être présent 0 fois ou plus {0,}  
/ta\*s/ → ts, tas, taas, taaas, taaaas ...
- + doit être présent 1 fois ou plus {1,}  
/ta+s/ → tas, taas, taaas, taaaas ...
- ? doit être présent 0 ou 1 fois {0,1}  
/ta?s/ → ts, tas
- {n} doit être présent exactement n fois  
/ta{3}s/ → taaas  
{n,m} doit être présent entre n et m fois  
/ta{2,5}s/ → taas, taaas, taaaas, taaaaas  
{n,} {,n} doit être présent au moins/plus n fois

Q: /`[0-9A-F]+`/    /`[^ ]{7}`/    /`.*`/    /`.+`/



## Regexp : alternative et regroupement

- | alternative  
/Fred|Paul|Julie/ → Fred, Paul, Julie
- (...) regroupement :
  - /Fred|Paul|Julie Martin/  
→ Fred, Paul, Julie Martin  
/(Fred|Paul|Julie) Martin/  
→ Fred Martin, Paul Martin, Julie Martin
  - /meuh{3}/ → meuhhh  
/(meuh){3}/ → meuhmeuhmeuh



## Regexp : classes (suite)

- \d un chiffre [0-9]  
\D un non-numérique [^0-9]
- \w un alphanumérique [0-9a-z\_A-Z]  
\W un non-alphanumérique [^0-9a-z\_A-Z]
- \s un espacement [ \n\t\r\f]  
\S un non-espacement [^ \n\t\r\f]
- [[:alpha:]] [[:lower:]] [[:upper:]] [[:xdigit:]] (héxa)

Exemples :

- /^ab\w+/ mots commençant par ab
- /[+-]?[0-9]\*/ nombres décimaux
- if( \$v =~ m/^D\*[0-9]+\$/ ) { ... }



## Regexp : mémorisation

- \$1 \$2... sous-chaîne précédemment matchée entre parenthèses  
À utiliser dans le membre droite de la substitution  
\$v =~ s/(w+)/mot trouvé : \$1 !/;  
La séquence '%=stylo+@'  
devient '%=mot trouvé : stylo !+@'
- \$v =~ s/(d+)/"\$1"/;  
\$v =~ s/(w+)\s\*=\s\*(d+)/\$1=\$2/;
- (?: ) regroupement non mémorisant  
s/(w+) (?:- )+(w+)/\$1 - \$2/  
'Label - - - texte' devient 'Label - texte'  
Le regroupement est nécessaire pour le quantificateur +  
s/(w+) (- )+(w+)/\$1 - \$3/ possible, mémorisation inutile



## Regexp : variables définies et extraction

Variables définies après l'expression dans le bloc :

- \$1 \$2 ... sous-chaînes matchées entre parenthèses
- \$& sous-chaîne matchant *À éviter pour des raisons ...*
- \$' \$' ce qui précède/suit \$& *... de performances des regexps*

```
my $v = 'za aa et tfe';
if($v =~ m/(a+) et ([a-z])/) {
 print "$1\n"; ⇒ 'aa'
 print "$2\n"; ⇒ 't'
 print "$&\n"; ⇒ 'aa et t'
 print "$'\n"; ⇒ 'za '
 print "$'\n"; ⇒ 'fe'
}
```

Pas très maintenable !





## Regexp : extraction dans des variables

L'expression `$v =~ m//` a pour valeur :

- en contexte scalaire : une valeur booléenne (vrai/faux)  

```
if($v =~ m/(.*)"(.*)" /) { print "$1 $2\n"; }
```
- en contexte de liste : la liste des séquences mémorisées  

```
($x,$y) = $v =~ m/(.*)"(.*)" /;
```

 Parenthèses obligatoires (même pour un seul élément)

```
if(my ($x,$y) = $v =~ m/(.*)"(.*)" /)
{ print "$x $y\n"; }
```



## Regexp : parenthèses et quantificateur

Texte : 143:71:930:72:ab:584

- `(\d+:)+` ne mémorise que la dernière occurrence 72:
- Pour tout mémoriser, il faut :
  - regrouper sans mémoriser pour le quantificateur `(?:\d+:)+`
  - mémoriser le tout `((?:\d+:)+)`
- NB: Avec deux parenthèses mémorisantes : `((\d+:)+)`
  - `$1` vaut 143:71:930:72:
  - `$2` vaut 72:



## Regexp : quantificateurs non gloutons

Problème : `"cf 'abc' g 'cde' ff"` `/'.*'/` → `'abc' g 'cde'`  
 ⇒ `/'[~']*'/` OK

Problème : `"cf <B>abc</B> g <B>cde</B> ff"`  
 ⇒ quantificateur non glouton `/<B>.*?</B>/` `/'.*?'/`

| Glouton | Non glouton |
|---------|-------------|
| *       | *?          |
| +       | +?          |
| ?       | ??          |
| {n,m}   | {n,m}?      |

Vocabulaire : glouton, gourmand, avide, *greedy* / paresseux, *lazy*



## Regexp : exemples (version)

1. `if( $v =~ m/\w+ \d* ?:/ ) { ... }`
2. `if( $v =~ m:^( [a-z]{4,} )/, : ) {print $1;}`
3. `if( $v =~ m/[ \w.-]+@[ \w.-]+\.[a-z]{2,4}/ )`



### Regexp : exemples (version)

4. 

```
if(my ($m,$n) = $v =~ m/(\w+)=(\d+)/)
{ print "$m $n\n"; }
```
5. 

```
if(my ($exp) = $v =~ m/^From: (.*)$/)
{ print "Mail from $exp\n"; }
```
6. 

```
if(my ($x,$y,$z) =
 $v =~ m/^(.*?):(.*?):(.*?):/)
{ print "($x,$y,$z)\n"; }
```

⇒ Attention à contraindre les quantificateurs non-gourmands



### Regexp : exemples (version)

7. 

```
$v =~ s/^<HTML>/<XML>/;
```
  8. 

```
$v =~ s/ +/ /g;
```
- Q: Que donne `s/ */ /g` sur `' ab c' ?`
9. 

```
$v =~ s/C="(.*?)" /D='$1' /g;
```



### Regexp : exemples (thème)

10. Vérifier que `$v` comporte **pieds**
11. Vérifier que `$v` finit par une lettre majuscule
12. Vérifier que `$v` ne commence pas par le mot **stop**



### Regexp : exemples (thème)

13. Extraire de `$v` chacun des deux premiers caractères
14. Extraire de `$v` les 2 premiers mots (suite d'alphabétiques)
15. Extraire de `$v` le dernier caractère numérique



## Regexp : exemples (thème)

16. Remplacer dans `$v` rouge par blanc (quid de Montrouge ?)
17. Supprimer de `$v` les espaces en fin de chaîne
18. Supprimer les doubles quotes autour des nombres entiers de `$v`



## Regexp : modifieurs de `m//` et `s///`

- **g** global sur toute la chaîne (`s///` seulement)  
`my $v = 'id et id';`  
`$v =~ s/id/zep/;   ⇒ $v='zep et id'`  
`$v =~ s/id/zep/g;   ⇒ $v='zep et zep'`
- **i** insensible à la case  
`m/toto/i → TotO, tOtO ...   m/[Tt][Oo][Tt][Oo]/`  
 Modifieurs cumulables : `s/id/zep/gi` ou `s/id/zep/ig`
- **e** évalue l'expression droite (`s///` seulement)  
`$v =~ s/(\d+)/$1+10/e;`  
 Imaginez l'expression dans du code : `print $1+10;`  
`$v =~ s/(\d+)/fonction($1)/e;`  
`sub fonction { my ($v)=@_; return $v+10; }`



## Regexp : modifieur `x`

Le modifieur `x` permet :

- d'écrire la regexp sur plusieurs lignes
- de placer des espaces, si besoin les protéger avec `\`
- et de mettre de commentaires

```
my $v = 'a:b:c:d';
$v =~ s/^ # début de chaîne
 (.*?) : # $1 : login
 (.*?) : # $2 : password
 (.*?) : # $3 : uid
 # la fin de la ligne est intacte
 /$1:$3:$2:/x;
print "$v\n";
```



## Regexp : modes ligne unique ou multiple

- ```
my $v = "mot\nlu";
```
- par défaut : mode intermédiaire
`^ $` : début/fin de chaîne `$v =~ m/mot$/` ⇒ faux
`.` ne matche pas `\n` `$v =~ m/t.lu/` ⇒ faux
 - **s** travaille en ligne unique (single-line), binaire
`.` matche `\n` `$v =~ m/t.lu/s` ⇒ vrai
 - **m** travaille en ligne multiple (multi-line)
`^ $` : début/fin de ligne `$v =~ m/mot$/m` ⇒ vrai
 - **ms** ou **sm** mélange des deux
`^ $` : début/fin de ligne `$v =~ m/mot$/sm` ⇒ vrai
`.` matche `\n` `$v =~ m/t.lu/sm` ⇒ vrai



Regexp : modifieur r (depuis 5.14)

Sans modifieur r, si on veut la chaîne modifiée et non-modifiée :

```
$v = "J'aime le rouge de Montrouge";
$w = $v;
$w =~ s/\brouge\b/blanc/;
```

Avec modifieur r, la substitution renvoie la chaîne modifiée mais ne modifie pas la variable :

```
$v = "J'aime le rouge de Montrouge";
$w = $v =~ s/\brouge\b/blanc/r;
```

Illustration avec map :

```
my @t = qw(verrouiller oui enfouir fouiller);
my @s = map { s/oui/yes/r } @t;
```



Regexp : références arrières

- \1 \2... sous-chaîne précédemment matchée entre parenthèses
/(.+) et \1/ → 'toto et toto', 'truc et truc' ...
différent de /.+ et .+/
/(.+), (.+), \2 et \1/ → 'a, b, b et a'
- Rappel : (? :) regroupement non mémorisant
/(.*) (? : et)+(.*) avec \2 \1/
→ 'Paul et Julie avec Julie Paul'
→ 'lala et et lili avec lili lala'
- Q: if(\$v =~ m:<([>]+)>.*</\1>:) {print \$1;}
• Q: Vérifier que \$v comporte 2 fois un même mot de suite (séparés par un espace) Que donne 'abc cba' ?



Regexp : références arrière et variables

Quand utiliser \1 ou \$1 ?

```
if( m/____\1____/ ) { .... print $1 .... }

s/____\1____/____$1____/
```

\1 : membre de gauche (car regexp)

\$1 : membre de droite (également \1 récemment) et à l'extérieur (car chaîne entre double-quotes ou expression)

NB: variables substituées dans le membre de droite de s///

```
$v =~ s/hello/$texte/;
```



Regexp : mémorisation avancée (depuis 5.14)

- Numérotation arrière relative
if(\$v =~ m/(\w+) +\g{-1}/) { print "mot 2 fois : \$1\n"; }
Si nombre négatif : référence arrière en relatif
Si nombre positif : variable classique \$1 ou \$2 ou \$3 ...
- Mémorisation nommée selon un label : (?<label>regexp)
"64000 Pau" =~ m/(?<cp>\d{5}) \w+/ ⇒ cp mémorise 64000
Où retrouver la valeur ?
- Table de hachage %+ \${cp}
- Référence nommée : \k<cp> ou \g<cp>
Autre exemple : m/(?<mot>\w+) +\k<mot>/ \${mot}



Regexp : factorisation de compilation

Variable de type regexp :

```

my $reg = qr/^\w\d+;/
      qr<^\w\d+>;
      qr(^\w\d+);

if( $v =~ m/$reg/ ) { ... }
if( $v =~ $reg ) { ... }      m/$reg1$reg2/
$v =~ s/$reg/chaîne/g;      s/$reg1$reg2/chaîne/

```

La regexp est utilisable plusieurs fois

Ne génère pas de recompilation de l'automate

Bonne idée : à combiner avec la mémorisation par label



Regexp : diverses fonctionnalités avancées

- Protéger une chaîne :
 - Recherche le contenu de `$s` : `m/$s/` `s/$s/etc/`
 - Recherche la chaîne `$s` : `m/\Q$s\E/` `s/\Q$s\E/etc/`
 - fonction `quotemeta` : `quotemeta('$$()')` renvoie `\$(\)`
- Spécifier des modifieurs dans une variable : (*?modifieurs*)
`$motif = qr/(?i)bizon/; if($v =~ $motif) {...}`
- Ancre de préfixe de recherche `\K`
 Pour supprimer un texte présent après un autre :
`$v =~ s/(prefixe)a supprimer/$1/;`
`$v =~ s/prefixe \Ka supprimer//;`



Regexp : d'autres assertions

- (*?=regexp*) assertion d'égalité positive en avant
 Correspond si *regexp* est trouvé à cette position.
`/lapin(= chasseur)/` → 'lapin' suivi de ' chasseur'
 Différence avec `/lapin chasseur/` ? Dans les substitutions :
`$v='lapin lapin chasseur';`
`$v =~ s/lapin(= chasseur)/civet/; ⇒ $v='lapin civet chasseur'`
- (*?!=regexp*) assertion d'égalité négative en avant
 Correspond si *regexp* n'est pas trouvé à cette position.
`$v='lapin lapin chasseur';`
`$v =~ s/lapin(?! chasseur)/vieux/; ⇒ $v='vieux lapin chasseur'`
- (*?<=regexp*) assertion d'égalité positive en arrière
- (*?<!=regexp*) assertion d'égalité négative en arrière



Regexp : quelques modules

- `use Regexp::Common; # regexp courantes`
`m/$RE{num}{real}/` # Nombre réel
`m/$RE{quoted}/` # Séquence entre ""
`m/$RE{delimited}{-delim=>'.'/}` # Séquence entre //
`m/$RE{balanced}{-parens=>'()'.'/}` # Séquence entre ()
- `use Regexp::Common qw/net/; # regexp réseau`
`m/$RE{net}{IPv4}/`
`m/$RE{net}{MAC}/`
`use Regexp::IPv6;`
`use Regexp::Common::Email::Address;`
- `Regexp::Log::Common` parse le common log format d'Apache
- `Regexp::Assemble` regexp multiples en une seule regexp



Regexp : retour sur grep et split

- `@t = grep { $_ > 0 } @t;`
`@t = grep /^w+:\d+$/, @t;`
 Sélectionne les éléments qui correspondent au motif
- `$chaine = "découpez moi\nj'aime ça";`
`@t = split /\s+/, $chaine;`
 La fonction renvoie les séquences découpées
`@t = ('découpez', 'moi', 'j'aime', 'ça');`

`@t = split /(\s+)/, $chaine;`
 La fonction renvoie en plus les séquences de séparation
`@t = ('découpez', ' ', 'moi', "\n", 'j'aime', ' ', 'ça');`



Exemple d'utilisation de regexp : ré-écriture d'URL dans Apache avec mod_rewrite

```
RewriteRule /article(.*)\.html /article.php?id=$1
RewriteRule /section/(.*)\.html /section.php?id=$1
RewriteRule /(.*)(.*)\.html /index.php?subject=$1&lang=$2

RewriteCond %{SCRIPT_FILENAME} !(data|donnees)[0-9]*\.html
RewriteRule ^(.*)\.html$ $1.php [L]

RewriteCond %{HTTP_USER_AGENT} (MSIE|Internet Explorer)
RewriteRule ^/$ /homepage.ie.html [L]
RewriteRule ^/$ /homepage.std.html [L]

RewriteCond %{HTTP_REFERER} !^$
RewriteCond %{HTTP_REFERER} !^http://(www\.)?monsite\.net/.*$ [NC]
RewriteCond %{HTTP_REFERER} !^http://(www\.)?siteami\.com/.*$ [NC]
RewriteRule .*\. (png|jpg|jpeg|gif)$ /voleur.jpg [NC]
```



Utilisation de regexp : filtrage courriel avec Postfix

EXAMPLE SMTPD ACCESS MAP

```
# Disallow sender-specified routing. needed if you relay mail for other domains
/[!@].*[%!@]/      550 Sender-specified routing rejected
# Postmaster is OK, that way they can talk to us about how to fix their problem
/^postmaster@/      OK
# Protect your outgoing majordomo exploders
if !/^owner-/
    /^(.*)-outgoing@(.*)$/      550 Use ${1}@${2} instead
endif
```

EXAMPLE HEADER FILTER MAP

```
# These were once common in junk mail.
/^Subject: make money fast/      REJECT
/^To: friend@public\.com/        REJECT
```

EXAMPLE BODY FILTER MAP

```
# First skip over base 64 encoded text to save CPU cycles.
~^[[:alnum:]]+/{60,}$~      OK
```



Fonctionnalité tr

Translation lettre à lettre : `tr` ou `y`

`tr/chaîne1/chaîne2/`

`y/chaîne1/chaîne2/`

- `$v =~ tr/abcd/0123/;`
 Tous les a de \$v seront transformés en 0, les b en 1 ...
- Quasi seul cas où `tr` va servir : conversion minuscules/majuscules
`$v =~ tr/a-z/A-Z/;`



Fichiers : lecture

```
# Ouverture du fichier
open( my $fd, '<' , 'nomDuFichier' )
    or die( "open: $!" );
# Lecture ligne par ligne
while( defined( my $ligne = <$fd> ) ) {
    chomp $ligne; # Supprime la fin de ligne
    print "$ligne\n";
}
# Fermeture du fichier
close( $fd );
```

\$fd est le descripteur de fichier.