Environnement de développement sous Linux
Module 2l012-2017fev

Dominique.Bereziat@lip6.fr
Valérie Menissier-Morain

Février 2017

Quatrième partie IV

Expressions régulières

Plan

Qu'est-ce qu'une *Regex*, à quoi ça sert ?

- Regex - REGular EXpressions en anglais
- En français : expressions régulière OU rationnelles OU motif
- Langage pour décrire les modèles de phrases, utilisé dans ce cours pour :
  - filtrer (`grep`, `sed`)
  - en extraire de l'information (`sed`)
  - substituer de l'information (`sed`)
- Langage à part entière, largement employé en traitement de l'information :
  - commande/langage : `perl`, `awk`, `php`, `lex`, `tcl`, `JS`, `...`
  - une bibliothèque (incluse dans la bibliothèque standard sous Linux, `man regex`) existe

Plusieurs systèmes

- Les expressions POSIX :
  - les *basiques* : BRE (Basic Regular Expression)
  - les *étendues* : ERE (Extended Regular Expression)
- Les expressions non POSIX :
  - celles de *perl* : PCRE (Perl Compatible Regular Expression), les plus puissantes
  - celles des commandes GNU, ce sont des ERE avec des extensions
- Dans ce cours : nous n'utiliserons que les *ERE*
- Passer des *BRE* et *ERE* demande une certaine gymnastique intellectuelle, il est plus aisée de se cantonner à un seul style
- Important : nous devrons indiquer aux commandes `grep` et `sed` l'utilisation des *ERE* !
  - `grep -E` ou `egrep`
  - `sed -r` (ou `sed -E`, POSIX)
- L'oubli de ces options peut donner des cheveux blancs pendant les examens
- En complément de ce cours :
  `https://www.regular-expressions.info`

Plan

Syntaxe des motifs ERE

- Langage faisant partie de la famille des langages rationnels
- On désigne par p (comme *pattern*) l'expression rationnelle
- Un motif est constitué :
  - d'opérateurs (ou caractères spéciaux) : `.^$?*+[]()\{\}\`
  - les autres caractères sont ordinaires, ils désignent la lettre à capturer
  - Pour désigner un opérateur comme caractère ordinaire : on l'échappe avec `\`.
    Exemple : `\?`, ou encore `\\`
  - Attention, certains caractères opérateurs peuvent devenir ordinaire selon le contexte et réciproquement !
- On appelle capture la chaîne de caractère qui a été effectivement capturée par un motif (donc ne pas confondre motif et capture)

Définition formelle du langage ERE

- Alphabet : les caractères du code ASCII (+lettres accentuées) + `^` (début de ligne) et `$` fin de ligne
- Les mots du langage sont appelés *motif*, ils sont définis de façon inductive par :

motif	signification
	mot vide
c	une lettre de l'alphabet (`a`, `b`, `\*`,...)
$p*$	répéter 0 fois (mot vide) ou plus p
(p)	factorisation de p
p_1p_2	concaténation de p_1 avec p_2
$p_1 \mid p_2$	motif p_1 ou motif p_2

- Exemples :
  - `aa*` la lettre `a` éventuellement répétée
  - `a*b` la lettre `b` éventuellement précédée de répétition de la lettre `a` : `b`, `ab` ou `aab` ou `aaaaaaaab`, ...
  - `foo|foobar` le mot `foo` ou le mot `foobar`
  - `foo(|bar)` idem

Définition formelle du langage ERE

- Le language est enrichi par :

motif	synonyme	signification
.	a | b | $\cdots$	toute lettre de l'alphabet
$p+$	$p(p*)$	répéter 1 fois ou plus p
$p?$	$(\mid p)$	0 ou 1 occurence de p
$p\{n\}$	$\underbrace{p\cdots p}_{n \text{ fois}}$	répéter n fois p
$p\{n,\}$	$\underbrace{p\cdots p}_{n \text{ fois}}(p*)$	répéter au moins n fois p
$p\{n,m\}$	$\underbrace{p\cdots p}_{n \text{ fois}}(\mid p \mid pp \mid \cdots \mid \underbrace{p\cdots p}_{m-n \text{ fois}})$	répéter entre n et m fois p
$p\{,m\}$	$(\mid p \mid pp \mid \cdots \mid \underbrace{p\cdots p}_{m \text{ fois}})$	répéter 0 ou au plus m fois p
$[c_1c_2\cdots c_n]$	$(c_1 \mid c_2 \mid \cdots \mid c_n)$	liste non ordonnée de lettre
$[c_1 - c_n]$	$(c_1 \mid c_2 \mid \cdots \mid c_n)$	liste ordonnée (ASCII)
$[\hat{}\cdots]$		exclusion de lettres

Exemples d'expressions simples

- Tous ces motifs capturent le mot `foo` (parmi d'autre) :

`foo`	toute chaîne de caractères qui contient `foo`
`foo|foobar`	toute chaîne de caractères qui contient le mot `foo` ou le mot `foobar`
`^foo`	toute chaîne de caractères qui commencent par `foo`
`foo$`	toute chaîne de caractères qui finit par `foo`
`^foo$`	seul le mot `foo` est reconnu, rien ne peut le précèder ni le suivre dans la chaîne de caractères
`f.o`	`'.'` désigne n'importe quel caractère, toute chaîne de caractères qui contient `f` puis un caractère puis `o`
`f[mnopq]o`	tous les caractères de la *liste* `mnopq` entre crochets autorisés à cet emplacement
`f[m-p]o`	`[m-p]` tous les caractères de l'*intervalle* de caractères entre `m` et `p`
`f[^a-lR-W0-9]o`	`[^a-lR-W0-9]` reconnaît tous les caractères autres que ceux de la liste d'intervalles indiquée
`f+oo`	suite d'au moins un `f` suivit de deux `o`
`fo{2}`	`f` suivit de deux `o` exactement

Encore d'autres exemples classiques

- une ligne de longueur paire : `^(..)+$`
- ... ou impaire : `^.(..)*$`
- un nombre entier naturel : `[0-9]+`
- un nombre entier relatif : `-?[0-9]+`
- en ajoutant l'opérateur unaire `+` : `[-+]?[0-9]+`
- en étant moins strict : `[-+]? *[0-9]+`
- un hexadécimal : `0x[0-9A-F]+`
- un flottant : `[-+]?([0-9]*\.[0-9]|[0-9]+\.|\.[0-9]+)*([eE][-+]?[0-9]+)?`

Sur les ensembles de caractères (`[]`)

- Attention aux caractères hors code ASCII (lettres accentuées) ! l'ordre est en fait l'encodage courant : dépendant de l'encodage donc pas portable
  - pour être portable, variable d'environnement `LC_CTYPE=C` (voir fonction `man setlocale`)
- Les caractères spéciaux perdent leur sémantique à l'intérieur des crochets : `[$^.+*?[(|)-]` désigne `$` ou `^` ou *etc.*
  Ce sont des listes de caractères, pas de raison d'y trouver un opérateur autre que `^` au début ou `-` au milieu !
  - `[*+^]` : `*` ou `+` ou `^`
  - `[ab-]` : `a` ou `b` ou `-`
  - `[[\]]` : `[` ou `]`

Classes de caractères

- Il existe des classes POSIX de caractères. Par exemple :
  - `[:digit:]` synonyme de `0-9`
  - `[:lower:]` synonyme de `a-z` + lettres accentuées (sauf si `LC_CTYPE=C`)
  - `[:upper:]` synonyme de `A-Z` + lettres accentuées (sauf si ...)
  - `[:alpha:]` alphabet synomyme de `a-zA-Z` + lettres accentuées ...
  - `[:alnum:]` alpha numérique
  - `[:punct:]` caractères de ponctuation
  - `[:blank:]`, `[:space:]`,...
  - voir `https://www.regular-expressions.info/posixbrackets.html`
- Les crochets font partie du nom et ne définissent pas la liste !
  - `[[:digit:]]` signifie exactement un chiffre
  - `[:digit:]` signifierai un caractère parmi `d`, `i`, `g`, `t` et `:` !

Sur les répétitions (*, +, { })

- Attention les répétiteurs sont gloutons : ils capturent la plus grande chaîne possible !

```
[a-z]*h[a-z]* aaaaaaaaahaaaaaaaaahaaaaaaaahaaaa
              <-------------------------> <-->
```

- En PCRE (`perl`) il existe des répétiteurs non gloutons, pas en [BE]RE !
- Syntaxe : contrairement aux opérateurs `[]` et `()` (un crochet ou une parenthèse non refermée provoque une erreur), un répétiteur qui est incorrectement défini est identifié comme des caractères ordinaires :
  - `a{` signifie la lettre `a` suivi de `{`
  - `(a` ou `[a` : erreur
  - `a}`, `a]`, `a)` : OK caractères ordinaires

Sur le bornage (ou ancrage)

- On peut trouver plusieurs captures d'un motif sur une ligne, comment extraire le bon ?
- Exemple classique : les champs d'un fichier CSV, comportant nom, prénom, et des notes, comment différentier le nom du prénom, d'une note, d'une autre ?
- Une stratégie : décrire comme motif la ligne entière, en utilisant les marqueurs `^` début de ligne et/ou `$` fin de ligne
- Exemple du fichier CSV `nom;prénom;note 1;note 2`
  - chercher tous les étudiants de prénom Martin : `^[^;]+;Martin;`
  - chercher tous les étudiants de nom Martin : `^Martin;`
- Remarques :
  - dans cet exemple, il n'est pas utile de décrire la ligne entière (c-a-d avoir un motif de type `^MOTIF$`) car on a utilisé le délimiteur `;` du fichier CSV
  - avec les délimiteurs, il vaut mieux procéder par exclusion plutôt que par classe : `[^;]+;` est plus robuste que `[[:alpha:][:space:]]+;`
  - on peut s'en sortir facilement avec `cut` qui extraira le bon champ (`cut -d';' -f1`)

Plan

Références arrières

- Consiste à nommer des captures (et pas des motifs) pour y faire référence ultérieurement dans le motif ou en dehors
- Nommer une capture : avec les opérateurs `(` et `)` (les mêmes que ceux utilisés pour la factorisation)
- Ces références sont numérotées dans l'ordre d'apparition dans le motif

```
^([^.]+)\.([^@]+)@([^.]+)\.(.+)$
 1          2         3          4
```

- Elles peuvent être imbriquées :

```
^([-_a-z0-9A-Z.]+)@(([a-z0-9A-Z.]+)\.(fr|edu))$
 1                  23                    4
```

- On y fait référence avec la syntaxe : `\n` où `n` numéro d'ordre. Dans l'exemple précédent : `\1` est le nom, `\2` le domaine et le TLD [1], `\3` le domaine et `\4` le TLD

1. Top Level Domain

Usage des références arrières (suite)

- Définir un motif qui cherche des répétitions :
  - capture immédiatement répétée :
    `(.+)\1`
    `toto` est reconnu mais `touto` non !
  - capture répétée éventuellement plus loin : `(.+).*\1`
    `toto` et `touto` sont reconnus (attention : `to` deux fois et pas `t` deux fois car l'opérateur `+` est glouton !)
- Analyse/extraire des information : `((.+)\2)`
  - `\1` est la capture complète du motif
  - `\2` est la répétition
- Extraire la capture (avec `bash`) pour la remplacer (avec `sed`)
- `grep` et `sed` utilisent quotidiennement les références arrières
- `perl` va beaucoup plus loin dans les notions de captures, et autorise la récursivité

Bilan sur le rôle des parenthèses

1. Délimite la portée de certains opérateurs : `*`, `?`, `!`, `|`, `{...}`
2. Nomme des captures pour :
   - y faire référence dans le motif
   - les extraire
   - les remplacer

Plan

`grep` en bref

- Nous parlons de la version GNU. Attention : la version BSD n'a pas exactement les mêmes options ni le même comportement !
- `grep` : Globally search Regular Expression and Print
- Elle filtre les lignes d'un texte par un motif
- Usage : voir `man grep`

```
grep -E MOTIF  [fichier [fichier ... ]]  # -E: ERE
egrep MOTIF ...      # synomyme que grep -E
zgrep -E MOTIF ...   # traiter des fichiers compressés
zegrep MOTIF ...     # synomyme de zgrep -E
```

- `MOTIF` est l'expression rationnelle qui désigne un motif de caractères à capturer (ou reconnaître)
- Il faut souvent protéger le motif du shell : `'MOTIF'` ou `"MOTIF"`

`grep` en bref

- Les lignes qui contiennent ce motif sont imprimées dans la sortie standard de `grep`
- `grep` lit les fichiers donnés en argument (après le motif) ou l'entrée standard si absent
- `grep` lit une ligne, s'il trouve un motif et que la ligne n'a pas fini d'être lue, il continue sa recherche, puis passe à la ligne suivante
- `grep` retourne `0` si une ligne a été imprimé (motif capturé), `1` si aucune ligne a été imprimé, `2` si le motif contient une erreur

Des exemples avec `grep`

- Soient les deux fichiers C suivant `f.c` :

```
1  /* $Id$ */
2  #include "f.h"
3  #include <stdlib.h>
4  #include <assert.h>
5
6  static int taille = 80;                          /* data: static   */
7
8  int *size_address = &taille;                     /* data: exported */
9  char codes[255];                                 /*  bss: exported */
10
11 int get_size () { return *size_address; } /* text: exported */
12
13 static void set_size (char *s)                   /* text: static */
14 {
15       long result;
16       char *end = s;
17
18       assert( s != NULL );
19       result = strtol(s, &end, 0);
20       assert( end != s );
21       taille = (int) result;
22 }
23 /* end of f.c */
```

Des exemples avec `grep`

- et `fmain.c` :

```
1  /* $Id$ */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include "f.h"
5
6  int main (int argc, char *argv[])
7  {
8       printf("%d\n", get_size());
9       exit(EXIT_SUCCESS);
10 }
11
12 /* end of fmain.c */
```

Des exemples avec `grep`

- Lignes avec `f` immédiatement suivi de parenthèses :

```
% grep -E 'f\(.*\)' *.c
fmain.c:     printf("%d\n", get_size());
```

- L'option `-F` (recherche rapide) n'utilise pas d'expressions régulières :

```
% grep -F 'f\(.*\)' *.c | echo Rien !
Rien !
```

- Lignes qui contiennent `f` :

```
% grep -E 'f' *.c
f.c:#include "f.h"
f.c:/* end of f.c */
fmain.c:#include "f.h"
fmain.c:     printf("%d\n", get_size());
fmain.c:/* end of fmain.c */
```

- Nombre de lignes qui commencent par `int` :

```
% grep -E '^int' *.c | wc -l
3
```

```
% grep -E '^int' *.c -c
f.c:2
fmain.c:1
```

Options essentielles de `grep` (autres que `-E`)

`-e MOTIF`	pratique si le motif commence par `-`
`-e M1 -e M2 ...`	recherche `M1` ou `M2` ou `...`
`-i`	ne tient pas compte de la casse
`-v`	recherche les lignes ne contenant pas le motif
`-w`	cherche le motif comme un mot (délimité par des caractères non alphanumérique, ainsi que `^` et `$`)
`-x`	équivaut à `^MOTIF$`
`-c`	compte les lignes qui contiennent le motif
`-l`	affiche le nom des fichiers qui capturent le motif
`-n`	affiche le numéro des lignes qui capturent le motif

Options moins essentielles mais utiles

`-F`	recherche rapidement une chaîne de caractères (`fgrep` est un alias de `grep -F`)
`-m=n`	terminaison après que `n` lignes ait été capturées
`-B=n`	afficher `n` lignes avant capture d'un motif
`-A=n`	afficher `n` lignes après capture d'un motif
`-q`	ne rien afficher sur `stdout`, utile pour les tests
`-s`	ne rien afficher sur `stderr`, utile pour les scripts
`-f FILE`	lit le fichier `FILE` qui décrit les motifs

- Autres options : `man grep`

Options de mise au point de `grep`

- La mise au point d'un motif correct peut être fastidieux pour le débutant
- Deux options bien pratique :
  - `-o` : n'affiche que les motifs qui ont été capturés
  - `--color` affiche en couleur les captures des motifs, numéros de ligne, et noms des fichiers
- Exemples :

```
% egrep --color 'include' *.c -n
f.c:1:#include <stdlib.h>
f.c:2:#include <assert.h>
fmain.c:1:#include <stdio.h>
fmain.c:2:#include <stdlib.h>
% egrep --color 'f\(.*\)' *.c -n
fmain.c:6:      printf("%d\n", get_size());
% echo "Ton Thé a-t-il oté ta toux ?" | grep --color t
Ton Thé a-t-il oté ta toux ?
% echo "Ton Thé a-t-il oté ta toux ?" | grep -i --color t
Ton Thé a-t-il oté ta toux ?
% echo "Ton Thé a-t-il oté ta toux ?" | grep -o t.
t-
t?
ta
to
%
```

Plan

`sed` en bref

- `sed` : Stream EDitor
- Usage :
  - transformation des lignes (substitution de motifs)
  - filtrage (élimination de ligne)
- Principe de fonctionnement :
  - Pour chaque ligne lue (fichier ou entrée standard) faire :
    1. copier la ligne dans le *pattern space* en retirant le `\n` final
    2. Appliquer la ou les commandes sur le *pattern space* (tampon est modifié sur place)
    3. à moins que l'option `-n` soit spécifié, imprimer le pattern space dans la sortie standard en ajout un `\n` final
  - Si plus de lignes à traiter, on termine

sed en bref

- `sed OPTIONS 'COMMAND[;COMMAND[;...]]' [FILE [FILE ...]]`
- `[]` partie optionnelle
- Si pas de fichiers, `sed` lit dans l'entrée standard
- `sed` écrit dans la sortie standard (sauf si `-i`)
- `;` délimiteur de commande
- Les options essentielles :

`-e COMMAND`	ajoute une commande à exécuter (séparateur `;` non POSIX)
`-n`	pas d'affichage systématiques des lignes
`-r`	utilisation des ERE
`-i`	modification *sur place* du fichier traitée
`-iEXT`	comme `-i` création d'un fichier backup (extension `EXT`)
`-f file`	lire le fichier `file` qui contient les commandes
`-rn`	groupement d'options : signifie `-r -n`

Les commandes de sed

- Elles sont la forme suivante (`[]` : partie optionnelle) :
  `[address[,address]] command[arguments]`
- `command` : une lettre
  - `d` *delete*
  - `p` *print*
  - `s` *substitute*
  - et d'autres : `y`, `q`, `w`, ...
- la commande s'applique à toutes lignes, ou alors aux lignes qui correspondent à une adresse
- `address` :
  - un nombre : numéro de ligne
  - `$` la dernière ligne
  - `/motif/` : les lignes capturées par le motif (`/` délimiteur de motif, on peut prendre n'importe quel caractère)
  - `adr1,adr2` : toutes les lignes comprises entre la ligne d'adresse `adr1` et celle d'adresse `adr2` (incluses)
- `address!` les lignes qui ne correspondent pas à `address`
- groupement de commandes : `{COMMAND;COMMAND}` utile derrière une adresse. Par exemple : imprimer deux fois la dernière ligne `${p;p}`

Commandes principales de `sed`

- `d` supprime les lignes concernées :
  - `1d` : supprime la première ligne
  - `/^#/d` : supprime les lignes commençant par `#`
- `p` imprime les lignes concernées :
  - **`sed`** `p` : double chaque ligne
  - **`sed`** `-rn '/MOTIF/p'` : synomyme de =grep -E 'MOTIF'=
  - **`sed`** `-r '/#/p'` : n'imprime que les lignes commençant par `#`
  - **`sed`** `-n '2,4p'` : imprime les lignes 2 à 4
- `q` termine la commande sans attendre la fin des lectures
  - **`sed`** `2q` : affiche les deux premières lignes d'un fichier
- `y/source/destination/` : équivalent de la commande `tr` (*translate characters*)
- `=` imprime le numéro de la ligne lue.
  Cette commande calcule le nombre de lignes de *file* :

```
sed -n $= file
```

La substitution

- s *substitute* l'opération fondamentale de `sed` !
  `s/motif/remplacement/flags`
  - `motif` le motif à capturer
  - `remplacement` le remplacement de la capture,
    - elle peut contenir des références arrières soit les symboles `\1` à `\9` ou encore `&` qui correspond au motif complet,
    - Elle peut contenir encore d'autres caractères de contrôle par exemple entre `\U` et `\E` les caractères seront convertis en majuscule
  - `flags` :
    - `g` : substitution globale, `sed` substitue tant qu'il le peut dans le *pattern space*
    - `p` : si il y a substitution, le *pattern space* est imprimé
    - n : ne substitue que la $n^{\text{ième}}$ capture
    - `i` : ne tient pas compte de la casse (comme `grep -i`)
    - il existe d'autres *flags*
- Comment fonctionne la substitution globale ? `sed` avance dans le *pattern space* au fur et à mesure qu'il capture et substitue (on dit qu'il consomme les captures)
  - Par exemple : `s/(.)(.)/\2\1/g` permute les caractères
- Voir `info sed`

Exemples

- Supprimer le code du programme (les lignes entre { et })

```
% sed -r '/^\{/,/^\}/d' fmain.c
/* $Id$ */
#include <stdio.h>
#include <stdlib.h>
#include "f.h"

int main (int argc, char *argv[])

/* end of fmain.c */
```

- Supprimer tout sauf le code :

```
% sed -n -r 's/^[^{]*\{/{/;/^\{/,/^\}/p' fmain.c
{
     printf("%d\n", get_size());
     exit(EXIT_SUCCESS);
}
```

ou encore :

```
% sed -r '/^\{/,/^\}/!d' fmain.c
{
     printf("%d\n", get_size());
     exit(EXIT_SUCCESS);
}
```

Exemples (suite)

- Remplacer `assert` par `nana`

```
% sed -r 's/assert\.h/nana.h/;s/assert(.*)\;/DI\1\;/g' f.c
/* $Id$ */
#include "f.h"
#include <stdlib.h>
#include <nana.h>

static int taille = 80;                         /* data: static   */

int *size_address = &taille;                    /* data: exported */
char codes[255];                                /*  bss: exported */

int get_size () { return *size_address; }       /* text: exported */

static void set_size (char *s)                  /* text: static */
{
     long result;
     char *end = s;

     DI( s != NULL );
     result = strtol(s, &end, 0);
     DI( end != s );
     taille = (int) result;
}
/* end of f.c */
```

Exemples (suite)

- Dedoubler les `a` partout et jeter les lignes qui n'en contiennent pas

```
% sed -r -e 's/(a)/\1\1/gp' -e d <fmain.c
int maain (int aargc, chaar *aargv[])
/* end of fmaain.c */
```

est équivalent à :

```
% sed -rn 's/(a)/\1\1/gp' <fmain.c
int maain (int aargc, chaar *aargv[])
/* end of fmaain.c */
```

- Idem mais sans protéger le motif du shell

```
% sed -r -n s/\(a\)/\1\1/gp < fmain.c
int maain (int aargc, chaar *aargv[])
/* end of fmaain.c */
```

Commandes `sed` avancées

- Définir une étiquette (pour y revenir) : `: label`
- Branchement inconditionnel : `b label`
- Branchement conditionnel : `t label`. On branche à l'étiquette `label` si la dernière substitution (commande `s`) a réussi
- `n` : remplacer le *pattern space* par la ligne suivante
- `N` : ajouter `\n` puis la ligne suivante au *pattern space*
- `h` : copier le *pattern space* dans le *hold space* (un presse-papier)
- `H` : idem en ajoutant d'abord un `\n`
- `g`, `G` : copier le *hold space* dans le *pattern space* (avec ou sans saut à la ligne)
- `x` : échanger (*pattern space* $\leftrightarrow$ *hold space*)

Des exemples

- Afficher une ligne sur deux :

```
% seq
1
2
3
4
% seq 4 | sed -n 'p;n'
1
3
```

- Afficher les lignes impaires :

```
% seq 4 | sed -n 'n;p'
2
4
```

- Regrouper des lignes :

```
% seq 4 | sed -n 'N;s/\n/-/;p'
1-2
3-4
```

Des scripts sed

- Un script `sed` est un script commme un autre !
- Remplacer toutes les fin de ligne par des tirets :

```
1 #! /bin/sed -rf
2 # nldash: converti les fins de ligne en tirets
3
4 : boucle       # début de boucle
5 N              # tamponner les lignes
6 $! b boucle    # itérer jusqu'à la fin du fichier
7 s/\n/--/g      # substituer toutes les newline
```

```
% ./nldash <fmain.c
/* $Id$ */--#include <stdio.h>--#include <stdlib.h>--#include "f.h"--
--int main (int argc, char *argv[])--{--     printf("%d\n", get_size()
);--     exit(EXIT_SUCCESS);--}-- --/* end of fmain.c */
```

- Remarque geek 1 sur le *shebang* : la norme POSIX n'autorise qu'un seul argument pour l'interpréteur, il faut regrouper les options passées à `sed` en une seule (si on veut être portable)
- Remarque geek 2 : la position de `sed` varie d'un système à l'autre. La solution `#! /usr/bin/env sed -f` n'est pas portable à cause de la remarque 1. Des solutions compliquées existent

Des scripts sed (suite)

- Supprimer les étiquettes HTML, même lorsqu'elles s'écrivent sur plusieurs lignes :

```
1 #! /bin/sed -rf
2 # delhtmltag: supprimer les balises HTML sur plusieurs lignes
3
4 :loop
5 s/<[^>]*>//g      # supprimer les balises bien formées
6 /</ {N; bloop}    # accumuler tant qu'on trouve une balise ouvrante
```

- Fichier exemple :

```
% head BasicMatrix.html
<HTML>
<!--
  -- Copyright (c) Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine 2000
  --
  -- Distributed under the Boost Software License, Version 1.0.
  -- (See accompanying file LICENSE_1_0.txt or copy at
  -- http://www.boost.org/LICENSE_1_0.txt)
  -->
<Head>
<Title>BasicMatrix</Title>
```

Des scripts `sed` (suite)

- Résultat :

```
% ./delhtmltag <BasicMatrix.html | head



BasicMatrix
```

- En version *one liner*

```
% sed <BasicMatrix.html -r ':l;s/<[^>]*>//g; /</{N;bl}' | head



BasicMatrix
```

Plan

Motif du shell et expressions rationnelles

- Les motifs du shell ne sont pas des expressions rationnelles !

sh	regexp
`?`	`.`
`*`	`.*`
`{a,bc}`	`(a|bc)`
`[!a]`	`[^a]`

- `[]` a le même comportement que se soit en bash ou regexp et c'est tout !

```
% ls
macros.sty  mot2            processusN.pdf   script1a          shell.tex
mondes.eps  mot3            procshdate1.pdf  script1b          stty.fig
mondes.fig  pbmcs.pdf       procshdate2.pdf  shell-etu.pdf     stty.pdf
mondes.pdf  processus1.pdf  procshdate.pdf   shell.pdf
mot         processus2.pdf  procsh.pdf       shell-slides.pdf
% ls [^p]*
macros.sty  mondes.pdf  mot3      shell-etu.pdf     shell.tex
mondes.eps  mot         script1a  shell.pdf         stty.fig
mondes.fig  mot2        script1b  shell-slides.pdf  stty.pdf
```

Opérateur `[[` et expressions rationnelles

- Depuis `bash` l'opérateur *builtin* `[[` a été étendu pour filtrer avec des expressions régulières
- Les captures sont également possibles (et pratique)
- Syntaxe : `[[` *value* =~ *motif* `]]`
  - *value* est typiquement une variable que l'on déréférence
  - *motif* peut être une expression constante ou une variable que l'on déréférence
  - l'opérateur retourne 0 si le motif a été capturé dans l'opérande de gauche
  - Important sur l'opérande de gauche, `bash` enlève les quotes comme à son habitude, mais pas sur l'opérande de droite qui font parti du motif !
- Langage : ERE sans références arrières
- Les captures sont placées dans le tableau `BASH_REMATCH`
  - `${BASH_REMATCH[0]}` contient la capture du motif complet
  - `${BASH_REMATCH[i]}` contient la capture du $i^{\text{ième}}$ sous-motif

Exemples avec [[

- Extraction de sous-chaînes

```
% if [[ $(date) =~ ([0-9]+):([0-9]+):([0-9]+) ]]; then
   echo ${BASH_REMATCH[1]} heures ${BASH_REMATCH[2]} minutes et \
        ${BASH_REMATCH[3]} secondes
fi
22 heures 08 minutes et 16 secondes
```

- A comparer avec :

```
% date
Mer fév 21 13:17:37 CET 2018
% date |
 sed -r 's/.* ([0-9]+):([0-9]+):([0-9]+).*/\1 heures \2 minutes \3 secondes/'
13 heures 17 minutes 37 secondes
% date |
 sed -r 's/.*([0-9]+):([0-9]+):([0-9]+).*/\1 heures \2 minutes \3 secondes/'
3 heures 17 minutes 37 secondes
```

Plan

Les fonctions `regex` de la bibliothèque standard

- La bibliothèque standard contient des fonctions pour recherche et capture par expressions régulières : `man 3 regex`
- Les appels se font en deux temps :
  1. On doit d'abord *compiler* l'expression régulière :

```
1 #include <regex.h>
2 int
3 regcomp(regex_t *preg, const char *pattern, int cflags);
```

`cflags` est un masque de bits (`REG_EXTENDED`, `REG_NOSUB`, `REG_ICASE`, ...)

  2. Puis on peut procéder à la capture :

```
1 int
2 regexec(const regex_t *preg, const char *string,
3         size_t nmatch, regmatch_t pmatch[], int eflags);
```

`preg` est fourni par `regcomp()`, `pmatch` est un tableau (alloué par nos soins) de taille `nmatch` qui contiendra les captures (peut valoir `NULL` si `nmatch` vaut `0`)

Exemple en C

- Exemple de recherche sans capture (`grep`) :

```c
/* sgrep.c: grep primitif */

#include <regex.h>
#include <stdio.h>

int main( int argc, char **argv) {

  if( argc > 1) {
    regex_t motif;
    char *my_regex = argv[1], line[512];
    int ret, l=1;

    if( !(ret=regcomp( &motif, my_regex,
                       REG_NOSUB | REG_EXTENDED))) {
      while( fgets(line, 511, stdin)) {
        if( !regexec(&motif,line, 0, NULL, 0) )
          fprintf(stdout, "%d:\t %s", l, line);
        l++;
      }
      regfree( &motif);
    } else
      fprintf(stderr, "Une erreur %d s'est produite\n", ret);
  }
  return 0;
}
```

Exemple en C (suite)

```
% gcc sgrep.c -o sgrep
% ./sgrep 're[^(]+\(' <sgrep.c
15:        if( !(ret=regcomp( &motif, my_regex,
18:           if( !regexec(&motif,line, 0, NULL, 0) )
22:          regfree( &motif);
% ./sgrep 're[^(+\(' <sgrep.c
Une erreur 7 s'est produite
```

Plan

Conclusion

- Les expressions régulières sont incontournables pour analyser et extraire du texte
- Non vu dans ce cours : la commande `awk`, version "C" de la commande `sed` permet notamment de faire des calculs numériques
- Non vu dans ce cours : la commande `perl` qui intégre nativement les expressions régulières.
  - Elle possède des extensions très puissante sur les expressions régulières (capture conditionnelle, motif non capturant) qui la rende incontournable pour l'analyse de fichier log
  - Elle s'inspire de `bash`, `sed` et `awk`, peut fonctionner en one-liner ou en scripts, elle est orienté objet et possède de très nombreuses extensions
  - Elle est également très utilisé dans la programmation réseau
- Les autres languages n'utilisent pas directement les expressions régulières autrement que par l'API `regex` de la bibliothèque standard