Utilitaires UNIX

AWK

Présentation de awk

awk est un langage interprété du système Unix, qui a été conçu pour simplifier les commandes du shell afin de traiter les fichiers de textes. Il a été inventé par M^{rs} Aho , Weinberger , et Kernigan (d'où son nom), afin de constituer un filtre qui permet d'extraire des lignes et des champs d'un fichier ASCII, et qui peut faire un affichage formaté avec une syntaxe identique au "printf" du C.

Les fichiers de données présentent en général la structure suivante :

champ_1	....	champ_n	← ligne 1
....	....	....	
champ_1	....	champ_n	← ligne m

Les champs sont séparés par des caractères séparateurs qui peuvent être : des espaces , des points , des virgules , ou tout autres caractères. Ils sont cependant généralement uniques (il n'y a pas 2 caractères différents qui servent en même temps de séparateur).

Dans awk il y a un certain nombre de variables globales qui donnent des informations sur les champs :

FS = ' '	indique le séparateur de champ (Field Separator) du fichier. Ici c'est un espace, mais il peut être modifié quand on le désire.
NF	donne le nombre de champ (Number of Field)
NR	donne le nombre de ligne (Number of Return)

Chaque champ d'une ligne du fichier peut être atteint par des variables qui sont : $1 , $2 , $3 , etc. avec la particularité de $0 qui représente la ligne entière.

La syntaxe d'une routine de awk a la forme suivante :

motif { action }

- *motif* une ***expression régulière***. Lorsqu'elle est absente cela signifie que l'action est exécutée dans tous les cas.
- *action* c'est une séquence d'instructions de syntaxe identique à celle du C. Quand action est absente la ligne en cours de traitement est intégralement recopiée sur la sortie standard.

awk traite chaque ligne séquentiellement.

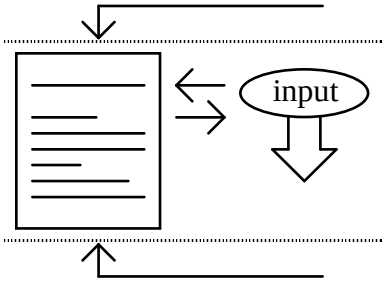
On peut utiliser *awk* comme *grep* pour faire de la recherche de motifs dans un fichier.

ex.

- rechercher les lignes qui commencent par *s* dans */etc/passwd*.
  awk /^s/ /etc/passwd
- afficher les lignes qui commencent par *root* jusqu'à celle qui contient *adm*.
  awk /root/,/adm/ /etc/passwd
- afficher les lignes dont le 1er champ contient la chaîne *uucp*(-F sert au séparateur de champ ≡ FS=":")
  awk -F: '$1 ~ /uucp/' /etc/passwd
- afficher les lignes se terminant par la chaîne */ksh*. ($ indique la fin de ligne)
  awk '/\/ksh$/' /etc/passwd
- afficher les lignes dont le 1er champ se termine par la chaîne *tp*.
  awk -F: '$1 ~ /tp$/' /etc/passwd
- afficher les lignes dont le 5^{em} champ contient la chaîne *Pierre* ou *pierre*.
  awk -F: '$5 ~ /[pP]ierre/' /etc/passwd
- afficher les lignes dont le 4^{em} champ est égal à 100. (pour alphanumérique placer entre ')
  awk -F: '$4 == 100' /etc/passwd
- afficher les lignes dont le 3^{em} champ est compris entre 700 et 760.
  awk -F: '$3 >760 && $3<700' /etc/passwd

Organisation de awk

Quand awk reçoit un fichier à traiter, il travaille sur 3 parties



La 1er routine est exécutée avant d'entrer dans le fichier

la 2ième routine (la boucle principale) est exécutée pour chaque ligne du fichier

la 3ième routine est exécutée quand on sort du fichier

La forme complète de awk est alors :

BEGIN { action }
motif { action }
END { action }

nota : Quand le séparateur n'est pas un espace et qu'on désire le changer, il est d'usage de le faire dans la partie BEGIN. Quand on a terminé de parcourir le fichier de données on désire peut-être afficher un message, il est d'usage de le faire dans la partie END.

Comme l'écriture en ligne de toutes les instructions des routines de awk risque d'être longue, on préfère en général la placer dans un fichier dont awk interprètera les commandes.

La syntaxe sera alors :

awk -f **fichier_prog liste_de_fichiers_de_données**

avec

fichier_prog	un fichier qui contient les commandes que doit effectuer awk
liste_de_fichiers_de_données	les fichiers, les uns derrière les autres, que l'on veut traiter

ex. Le fichier *src_awk_1* permet de savoir si l'on a des lignes vides, avec seulement un nombre, ou seulement du texte.

src_awk_1

```
# test de ligne composée de : nombres , mots ,ou vide
/[0-9]+$/  { print "c'est un nombre entier" }
/[A-z]+/   { print "c'est un mot" }
/^$/       { print "c'est une ligne vide" }
```

```
% awk -f src_awk_1
   2
   c'est un nombre entier
   toto
   c'est un mot
   1K
   c'est un nombre entier
   c'est un nombre entier
   ↵
   c'est une ligne vide
   CTRL-d
%
```

on exécute sur l'entrée standard (le clavier)

↵ = retour chariot

Les print peuvent être plus compliqués et utiliser les champs dans des calculs (expression arithmétique)

ex. soit le fichier de données suivant :

donnees_1

Nom	nb. de cigarettes	prix du paquet
Laurent	50	11
Pierre	0	0
Jacques	25	12
Annie	40	25

Le programme suivant va donner le nombre de paquets de cigarettes fumées en 1 semaine (8 jours)

src_awk_2

```
# fumé en 1 semaine
$2>0 {print $1,$2*8/20}
```

```
% awk  -f   src_awk_2   donnees_1
   Laurent 20
   Jacques 4.8
   Annie 10
%
```

Le programme suivant va donner le prix dépensé en 1 semaine (8 jours)

src_awk_3

```
# prix parti en fumée en 1 semaine
{printf ("argent de la semaine pour %s \t %.2f\n" , $1 , $2*$3*8/20 )}
```

```
% awk  -f   src_awk_3  donnees_1
   argent de la semaine pour Laurent     220
   argent de la semaine pour Pierre      0
   argent de la semaine pour Jacques     57.6
   argent de la semaine pour Annie       250
%
```

Evolution de awk

awk a évolué pour donner *nawk* (new awk) et *gawk* (gnu awk).

- Les auteurs de awk l'ont amélioré avec *nawk* qui est malheureusement resté longtemps chez AT&T et n'est pas disponnible partout.
- La FSF a conçu dans le projet GNU un *gawk* qui a implémenté de nouvelles fonctionnalités.

Syntaxe du langage

Un langage est composé de variables , d'opérations et de commandes. Nous en dressons une liste ci-dessous. Elle sera suivie d'ex. qui permettront de comprendre comment on peut les utiliser.

Variables

Elles peuvent être des mots utilisés directement dans l'action ou être des variables prédéfinies. Elles sont affectées par = . Ex. FS = ","

Variables prédéfinies

De awk :

FILENAME	fichier de données courant
FS	séparateur de champs (par défaut : l'espace)
NF	nombre de champs dans la ligne courante
NR	nombre de lignes
OFS	séparateur de sortie (par défaut : l'espace)
ORS	séparateur de ligne de sortie (par défaut : retour chariot)
RS	séparateur de ligne (par défaut : retour chariot)
$0	le fichier d'entrée entier
$i	le $i^{ème}$ champ dans la ligne (les champs sont séparés par FS)

De nawk :

ARGC	nombre d'arguments sur la ligne de commande
ARGV	le tableau des arguments passés en ligne de commande
FNR	idem NR mais relatif au fichier courant
OFMT	format de sortie de nombre (par défaut : %.6g)
RSTART	la 1^{er} position dans la chaîne reconnue par le motif
RLENGTH	longueur de la chaîne reconnue par le motif
SUBSEP	caractère séparateur pour un tableau (par défaut : \034)

De gawk :

ENVIRON	tableau associatif des variables d'environnement
IGNORECASE	tableau associatif des variables d'environnement

Tableau

Les tableaux peuvent être créés par la fonction split (voir fonctions), ou simplement être appelés dans une instruction d'affectation. ++ , += ou -= sont utilisés pour incrémenter ou décrémenter le tableau comme en C.
Les éléments du tableau peuvent être indicés avec des nombres (Tab[1] , ... , Tab[n]) ou avec des noms.

ex. Pour compter les occurrences d'un motif on peut écrire :

```
# remplacer motif par le motif recherche
BEGIN { printf ( "on cherche combien de motif il y a dans ce texte\n" ) }
/motif/ { Tableau["/motif/"]++ }
END { print Tableau["/motif/"] }
```

Opérateurs

= , =+ , -= , *= , /= , %= , ^=	affectation , avec calcul type C
?:	opérateur ternaire de choix type C (dans nawk et gawk)
||	ou logique
&&	et logique
~ , !~	reconnaît une expression régulière , et négation
< , <= , > , >= , != , ==	opérateurs de relation type C
( )	concaténation
+ , -	adition , soustraction
*** , / , %**	multiplication , division , modulo
+ , - , !	signe unaire , et négation
^	exponentiel
++ , --	incrément , décrément type C (post et préfixe)
$	référence à un champ

ex. d'utilisation de variable

NF>5 { .... }	cherche les lignes qui contiennent plus de 5 champs
NF>5 && /toto/	cherche les lignes qui contiennent plus de 5 champs et où *toto* apparaît
NR==10,NR==20	cherche de la ligne 10 à 20

Instructions

Elles ressemblent beaucoup à celles du C.

Classées par type

Arithmétique		Chaîne		Contrôle		In/out	
atan2	*	gsub	*	break		close	*
cos		index		continue		delete	*
exp		length		do/while	*	getline	*
int		match	*	exit		next	
log		split		for		print	
rand	*	sub	*	if		printf	
sin		substr		return	*	sprintf	
sqrt		tolower	**	while		system	*
srand	*	toupper	**				

* généralement n'existe pas dans le awk d'origine , se trouve dans nawk

** seulement dans gawk

Liste des instuctions

fonction	syntaxe	commentaire
atan2	atan2(y , x)	donne arctangente de y/x en radians
break		sortie de boucle *while* ou *for*
close	close(fichier) close(cmd)	Dans les implémentations de *awk* on peut avoir 10 fichiers ouverts et 1 pipe. *nawk* permet de fermer un fichier ou un pipe.
continue		repart au début de la prochaine itération de la boucle while ou for
cos	cos(x)	donne le cosinus de x
delete	delete (T[E])	suprime un élément E du tableau T
do/while	do corps while (expr)	boucle faire .... tant que. termine quand expr est faux
exit		n'exécute pas les instructions restantes. Seule END est exécutée.
for	for (init; expr; inc) corps	boucle for. Le corps peut être composé de plusieurs commandes groupées entre { }. ex. : for (i = 1 ; i <= 10 ; i++) ....
	for (item in array) corps	pour chaque *item* trouvé dans le tableau associatif on exécute le corps. On indice chaque élément par array[item].
getline	[var][<file] cmd | getline[var]	lit la ligne suivante de l'entrée. Les lignes sont lues une par une, et affectées à $0. NF , NR , FNR sont mises à jour
gsub	gsub (r , s , c)	substitution globale *s* de chaque recherche d'expression régulières *r* dans la chaîne *c* (par défaut $0). Rend le nombre de substitution.
if	if (condition) corps_1 [else corps_2]	si *condition* est vrai exécute corps_1, sinon exécute corps_2 si else existe. *condition* utilise les opérateurs relationnels ou une recherche de motif ex. if $1 ~ /[Aa].*/
index	index (sc , c)	donne la position de la sous-chaîne *sc* dans la chaîne *c*.
int	int (arg)	donne la valeur entière de *arg*.
length	length (arg)	donne la longueur de *arg*. Si *arg* n'existe pas $0 est pris par défaut.
log	log (arg)	donne le log de *arg*.

macht	macht (c , r)	recherche l'expression régulière *r* dans la chaîne *c* et donne sa position ou 0 si ne le trouve pas. Positionne les variables RSTART et RLENGTH
next		lit la nouvelle entrée et recommence le cycle motif / action.
print	print [args] [dest]	affiche *args* sur la sortie. *args* peuvent être des chaînes qui doivent être entre "". *args* sont en général des champs. S'ils sont séparés par des , alors ils le seront en sortie par le caractère présent dans OFS. S'ils le sont par des espaces ils seront concaténés en sortie. *dest* est une redirection unix ou un pipe.
printf	[format [, expr (s)]	affiche en sortie en formatant les *expr*. Même syntaxe qu'en C. %s une chaîne %d un nombre entier %n.mf un nombre flottant avec *n* avant le . et *m* après %[-]nc *n* donne la taille minimum. - justifie à gauche
rand	rand ( )	donne un nombre aléatoire compris entre 0 et 1. Cette fonction renvoie le même nombre à chaque fois que le script est exécuté. Pour avoir un nombre différent utiliser *srand*.
return	return [exp]	utilisé à la fin d'une fonction pour transmettre le résultat *exp*.
sin	sin (x)	donne le sinus de x
split	split(s, array[, ser])	découpe une chaîne *s* en éléments de *array*. La chaîne est découpée en autant d'éléments qu'il y a de séparateurs *sep*. Si sep n'est pas spécifié c'est le contenu de FS qui est utilisé. Le nombre d'éléments créés est retourné
sprintf	[format [, expr (s)]	donne le résultat du formatage de *expr*. Les données sont formatées mais pas sorties.
sqrt	sqrt (x)	donne la racine carrée de x
srand	srand (expr)	utilise *expr* pour initialiser le générateur aléatoire (par défaut l'heure)
sub	sub (r , s , c)	substitue par *s* la 1ère expression régulière *r* trouvée dans la chaîne *c*. Donne 1 si c'est fait 0 sinon.
substr	substr (c , m , [n])	donne la sous-chaîne de la chaîne *c* qui commence à la position *m* et qui fait *n* caratère de long. Si *n* est omis elle prend tout jusqu'à la fin de la chaîne.
system	system (cmd)	appelle une commande système unix. Elle renvoie le status de celle-ci. La sortie de la cmd n'est pas disponible dans le script. Il faut utiliser *cmd* | *getline* pour cela
tolower	tolower (c)	transforme tous les caractères majuscules de la chaîne *c* en minuscules
toupper	toupper (c)	transforme tous les caractères minuscules de la chaîne *c* en majuscules
while	while (condition) corps	boucle d'exécution de *corps* tant que la condition est vraie

Exemple

Ces exemples sont donnés sans "enrobage". Il faut les placer dans un fichier de texte et le rendre exécutable avant de pouvoir les tester.

- Compter les lignes vides

```
# compter les lignes vides
/^$/ {          ++x   }
END { print x  }
```

- Afficher le nom et la taille des fichiers d'un répertoire

```
# afficher le nom et la taille des fichiers
ls -l $* | awk '{ print $5 , "\t", $9 }' -        # - à la place du nom de fichier pour une entrée standard
```

- Afficher le nom et la taille des fichiers d'un répertoire et la taille totale occupée

```
# afficher le nom et la taille des fichiers
ls -l $* | awk '
      BEGIN { print "BYTES" "\t" "FILE" }
      {   sum += $5
          ++nbrfile
          print $5 , "\t", $9
      }
      END { print "\n" , "total : " , sum , "bytes issus de " , nbrfile , " fichiers" , "\n"  }
' -
```

- Afficher le nom et la taille des fichiers d'un répertoire et la taille totale occupée avec les sous-directories

```
# afficher le nom et la taille des fichiers et les sous-directories
# en entrée : la liste produite par ls -l
ls -l $* | awk '
      # nbrfile : donne le nbr. total de fichier
      # sum  : la taille totale en bytes
      # 1)  donne les en-têtes d affichage              ### ne pas mettre d apostrophe s.v.p.
      BEGIN { print "BYTES" "\t" "FILE" }
      # 2) test du 1er champ s'il commence par un - (=> donc c est un fichier)
      NF = = 9 && /^-/ {    ### test s'il y a bien 9 champs et que le 1er commence par "-"
          sum += $5             ### ajoute la taille
          ++nbrfile             ### augmente le nbr. de fichiers
          print $5 , "\t", $9   ### affiche taille et nom
      }
      # 3) test du 1er champ s'il commence par un d (=> donc c est un sous-répertoire)
      NF = = 9 && /^d/ {    print "<dir>" , "\t", $9 }        ### affiche <dir> et nom
      # 4) test du 1er champ s'il commence par un . (on l'affiche en entier)
      $1 ~ /^\..*:$/ {print "\t", $0    }         ### affiche la ligne
      # 5) c est fini
      END { print "\n" , "total : " , sum , "bytes issus de " , nbrfile , " fichiers" , "\n"  }
' -
```

- Conversion d'une date *mm-dd-yy* ou *mm/dd/yy* en *jour mois année*
  Si le fichier s'appelle conv_date , l'appeler en faisant : echo 7/3/97 | conv_date

```
# conversion d'une date
awk '
    # 1)  construire un tableau de mois
    BEGIN { # on utilise 2 listes pour construire le tableau
        liste_mois="Janvier,Fevrier,Mars,Avril,Mai,Juin,Juillet,"
        liste_mois = liste_mois "Aout,Septembre,Octobre,Novembre,Decembre"
        split( liste_mois,mois,"," )
    # 2) test de ce qu'il y a en entrée
    $1 != ""{
        # coupe sur le "/" les champs et les place dans un tableau
            szOfArray = split($1,date,"/")
        # test si le champ est unique auquel  cas tester avec un "-"
            if (szOfArray == 1)
                # test avec un "-"
                    szOfArray = split($1,date,"-")
                # la taille doit être plus grande que 1
                    if (szOfArray == 1)
                        exit                        ### sortie immédiate
        # ajoute 0 au nombre de mois pour transformer les champs en nombres
            date[1] +=0
    # 5) c'est fini, on affiche
    END { print date[2] , mois[date[1]]  " 19" date[3] "n"  }     ### , separé par OFS ;   et concaténé
' -
```

Les expressions régulières

Expressions régulières

Une expression régulière est un moyen de trouver un motif dans une ligne de texte. Elles sont utilisées par presque tous les programmes utilitaires d'Unix tels que : vi , sed , awk , grep , ...etc.

Une expression régulière est composée de caractères ou de chaînes de caractères qui forment le motif et de méta-caractères qui indiquent une action à faire avec ce motif.

nota : Quand on utilise une expression régulière elle comporte des signes qui doivent être pris comme expression et non comme une chaîne de caractères. Il faut généralement quoter l'expression.
ex. Recherche dans le fichier *toto* de *Le* en début de ligne, et affichage du rang des lignes trouvées
grep -n '^Le' toto

Les caractères ou les chaînes

Cela s'écrit tout simplement :

a est le caractère *a*
abc est la chaîne *abc*

Les Méta-caractères

Liste des méta-caractères, de leurs utilisations et de leurs significations.

symbole	ed	ex	vi	sed	awk	grep	egrep	signification
\	•	•	•	•	•	•	•	inhibiteur de méta-caractère
.	•	•	•	•	•	•	•	remplace un caractère
*****	•	•	•	•	•	•	•	permet 0 , 1 ou plusieurs répétitions de motifs
^	•	•	•	•	•	•	•	indique le début de ligne
$	•	•	•	•	•	•	•	indique la fin de ligne
[ ]	•	•	•	•	•	•	•	encadre un ensemble de caractères
\(\)	•	•		•				enregistre le motif pour de multiples recherches
\{ \}	•			•		•	•	recherche un nombre de motifs
\< \>	•	•	•					indique un début ou une fin de mot
+					•		•	permet 1 ou plusieurs répétitions de motifs
?					•		•	permet 0 ou 1 répétition de motifs
|					•		•	sépare les motifs à rechercher
( )					•		•	groupe les expressions

Le point

Le point indique n'importe quel caractère, excepté le caractère de fin de ligne.

ex. ^.$ indique une ligne avec un seul caractère (pour ^ et $ voir *caractères d'ancrages dans les lignes*)

Les ensembles

La définition d'ensemble se fait par une liste de caractères entre crochets. On recherche un des caractères listés.

ex.

[abcd]	indique que l'on cherche soit a , ou b , ou c ou d
[a-d]	est la forme compacte de [abcd]
[A-Za-z]	recherche une lettre minuscule ou majuscule
[A-Za-z0-9_]	cherche une lettre ou un chiffre ou _

On peut combiner les expressions afin de rechercher très précisément un motif.

ex. rechercher un mot qui :

- commence par S — S
- est le premier de la ligne — ^ (à mettre au début)
- la 2ième lettre est une minuscule — [a-z]
- la 3ième lettre est une voyelle — [aeiou]
- la 4ième lettre est une minuscule — [a-z]
- la 5ième lettre est un espace — □ (□ indique le caractère espace qui n'est pas visible)

- donc l'expression régulière qui recherchera ce mot est : **^S[a-z][aeiou][a-z]□**

Exceptions sur les ensembles

Le caractère ^ indique un ensemble **à ne pas prendre**.

ex. d'ensemble

[0-9]	n'importe quel chiffre
[^0-9]	n'importe quel caractère autre qu'un chiffre
[-0-9]	n'importe quel chiffre ou un -
[0-9-]	n'importe quel chiffre ou un -
[^-0-9]	n'importe quel caractère autre qu'un chiffre ou un -
[]0-9]	n'importe quel chiffre ou un]
[0-9]]	n'importe quel chiffre suivit d'un]

Caractères d'ancrages dans les lignes

^ et $ sont utilisés pour indiquer la position de début ou de fin de ligne

ex. d'utilisation de ^et de $

^A	un A en début de ligne
A$	un A en fin de ligne
A^	les caractères A^ n'importe où dans la ligne
$A	les caractères $A n'importe où dans la ligne
^\^	un ^ au début de la ligne
^^	idem que ^\^
\$$	un $ en fin de ligne
$$	idem \$$

nota : Par convention sous unix, les caractères ^ et $ indiquent le début et la fin d'une ligne.
ex. sous C shell !^ et !$ indiquent le premier et le dernier argument de la ligne.

La répétition

Indéfinie

Lorsque l'on veut un nombre inconnu de motifs à rechercher on place * à la fin du motif désiré.
ex.

[0-9]*	recherche 0 , 1 ou plusieurs chiffres
[0-9][0-9]*	recherche 1 ou plusieurs chiffres
^*	recherche 0 ou plus espaces au début de ligne

Attention :

^A* indique n'importe quelle ligne ⇔ A* : c'est 0 ou plus de A. 0 c'est n'importe quoi. Alors ^A* c'est le début de n'importe quoi, donc toutes les lignes.

Un multiple précis de répétitions

Avec * on ne peut pas préciser le nombre de répétitions que l'on désire. C'est \{ ... \} qui permet de le dire.

\{n\}	exactement n fois
\{n,\}	au moins n fois
\{n,m\}	entre n et m fois

ex. de répétitions

^*	n'importe quelle ligne commencant par *
^A*	n'importe quelle ligne
^AA*B	une ligne qui commence par 1 A ou plus suivie par un B
^A\{4,8\}B	une ligne qui commence par 4 , 5 , 6 , 7 ou 8 A suivie par un B
^A\{4,\}B	une ligne qui commence par 4 ou plus A suivie par un B
^A\{4\}B	une ligne qui commence par AAAAB
\{4,8\}	une ligne avec {,4,8 et }
A{4,9}	une ligne avec A{4,9}

Répétition étendue

Les caractères + et ? servent à indiquer des répétitions.
On peut considérer que :

- **?** est identique à **\{0,1\}**
- **+** est identique à **\{1,\}**

Recherche de mot

La recherche de mot est difficile avec les expressions régulières sans un méta-caractère spécial.

ex. Recherchons le mot *il* dans la ligne : ***il** est bien dans les **île**s des Ph**il**ippines*.
On voit aisément que rechercher le motif *il* n'est pas suffisant, on le trouverait 3 fois. Rechercher □*il*□ ne permet pas non plus de détecter il en début ou en fin de ligne, ni non plus si cela se trouve terminé par un point.

La solution est d'utiliser les méta-caractères \< et \> , simulaires à ^ et $, et qui ne s'occupent pas de la position des caractères

ex. \<[Hh]ello\> recherchera les mots *Hello* ou *hello*.

Mémorisation de motif

La répétition de recherche nécessite aussi un méta-caractère spécial.

ex. [a-z][a-z] permet de rechercher 2 minuscules dans une ligne.
Si l'on désire recherchee 2 motifs identiques dans une ligne cette expression n'est pas suffisante car il faut mémoriser le 1er motif pour le réutiliser après.

Dans certains utilitaires on peut mémoriser un motif en l'encadrant avec \(\) suivi de \α. Avec α un chiffre compris entre 1 et 9, qui indique le "buffer" utilisé.

Ainsi pour rechercher 2 lettres identiques dans une ligne, il faudra utilser l'expression suivante : \([a-z]\)\1

ex. pour rechercher le palindrome *radar* il faut écrire : \([a-z]\)\([a-z]\)[a-z]\2\1

- Le 1er \([a-z]\) pour le *r* stocké en \2
- Le 2ième \([a-z]\) pour le *a* stocké en \1
- le dernier [a-z] pour le *d*

Exemple de recherche de motif

- Motifs généraux

sac	la chaîne *sac*
^sac	*sac* en début de ligne
sac$	*sac* en fin de ligne
^sac$	sac comme seul mot sur la ligne
[Ss]ac	*Sac* ou *sac*
s[aeiou]c	la 2ième lettre est une voyelle
s[^aeiou]c	la 2ième lettre est une consonne ou une majuscule ou un symbole
s.c	la 2ième lettre est n'importe quel caractère
^...$	une ligne qui contient exactement 3 caractères
^\.	une ligne qui commence par .
^\.[a-z][a-z]	une ligne qui commence par . et avec 2 minuscules ensuite
^\.[a-z]\{2\}	idem que précédent, mais pour grep et sed seulement
^[^.]	une ligne qui ne commence pas par .
err*	er, err , errr , ...etc
"mot"	mot entre guillemets
"*mot"*	mot entre ou sans guillemets
[A-Z][A-Z]*	une ou plusieurs majuscules
[A-Z]+	idem mais pour egrep ou awk
[^0-9A-Za-z]	un symbole (qui n'est pas un chiffre ni une majuscule ni une minuscule)
[123]	un chiffre soit 1 ou 2 ou 3

- Motif pour **egrep** ou **awk**

un|deux|trois	l'un des mots : *un* ou *deux* ou *trois*
80[234]?86	le mot : *8086* ou *80286* ou *80386* ou *80486*
compan(y|ies)	l'un des mots *company* ou *companies*

- Motif pour **ex** ou **vi**

\<the	un mot comme *the* , *theatre* , ...etc
te\>	un mot comme *binette* , *te* , ...etc
\<est\>	le mot *est*

- Motif pour **grep** ou **sed**

0\{5,\}	5 zéros ou plus
[0-9]\{4\}	un nombre sur 4 chiffres ex. une année

GREP

grep

- **grep** est un filtre qui compare le contenu d'un fichier avec une expression régulière.
- Il sert essenciellemnet à rechercher les fichiers contenant un motif. ex. on sait que l'on a écrit le mot *anticonstitutionnellement* quelque part dans un fichier mais on ne se rappelle pas dans lequel.
- On fait alors : ***grep*** *-l anticonstitutionnellement ** ce qui permettra d'avoir le nom de tous les fichiers (-l) qui contiennent le mot dans l'ensemble de tous les fichiers de la directorie courante (*).

grep est certainement l'utilitaire le plus utilisé dans le monde Unix.

Tous ont voulu avoir un grep qui repondait à leurs besoins, c'est pourquoi il en existe plusieurs.

- "le bon vieux" ***grep***
- le grep étendu : ***egrep*** (expression grep)
- le grep rapide : ***fgrep*** (fast grep)
- ***agrep*** est une version "approximative" de grep. Il recherche des lignes ressemblantes plus ou moins.

- La syntaxe de **grep** a la forme :

  grep *[option] expressions régulières [fichiers ...]*

  Les options sont :
  - ◊ -b précède chaque ligne par le numéro du bloc où a été trouvé le motif.
  - ◊ -c donne seulement le nombre de ligne qui contiennent le motif.
  - ◊ -l affiche seulement le nom des fichiers contenants le motif.
  - ◊ -i ignore la distinction majuscule/minuscule dans le motif.
  - ◊ -n précède chaque ligne par le numéro de la ligne où a été trouvé le motif.
  - ◊ -s supprime les messages d'erreurs
  - ◊ -v affiche toutes les lignes ne contenant pas le motif

  nota : Si aucun fichier n'est donner grep suppose que c'est l'entrée standard (le clavier) qui est spécifié.

- Pour **egrep** la syntaxe à la forme :

  egrep *[option] expressions régulières [fichiers ...]*

  Les options suplémentaires par rapport à *grep* sont :
  - ◊ -e chaine cherche les chaînes spéciales (ex. celles commençants par -).
  - ◊ -f fichier prend les listes de chaines dans un fichier.

- Pour **fgrep** la syntaxe à la forme :

  fgrep *[option] chaîne de caractères [fichiers ...]*

  Les options suplémentaires par rapport à *grep* et *egrep* sont :
  - ◊ -x affiche seulement les lignes qui correspondent absolument avec la chaîne de caractères.

rappel d'expressions régulières

÷ Pour **grep**

[...]	plage de caractères permis à cet emplacement
[^....]	plage de caractères interdits à cet emplacement
.	un caractère quelconque
*	la répétition agit sur le caractère avant l'étoile
$	fin de ligne
\{...\}	répétition de caractères placés devants

÷ Pour **egrep**

+	caractère de répétition (1 fois au moins)
?	caractère de répétition (1 fois ou pas du tout)
|	*ou* avec des critères à gauche et à droite du signe
(...)	groupement de critères partiels

Exemple de grep

- grep

grep '^[A-Z][A-Z]*' *fichier*	toutes les lignes du *fichier* commençant par une majuscule
grep '^[A-Z]\{1,\}' *fichier*	idem
grep '^[0-9]\{3\}$' *tableau*	cherche dans *tableau,* toutes les lignes contenant seulement 3 chiffres
grep ':[0-9]\{2,\}:' *tableau*	cherche dans *tableau,* toutes les lignes contenant un minimum de 2 chiffres placés entre 2 double points
grep "[Rr]adis" *legume*	cherche dans le fichier *legume*, toutes les lignes qui contiennent *Radis* ou *radis*
grep "^.o" *legume*	cherche dans le fichier *legume*, toutes les lignes qui commencent par une lettre suivie de o
grep '[A-Z][A-Z]*' *legume*	cherche toutes les lignes de *legume* contenant au minimum une majuscule

- egrep

egrep '[a-z]+' *legume*	cherche dans *legume*, toutes les lignes contenant au minimum une minuscule
egrep '(Une|Deux)fois' *fichier*	cherche dans *fichier*, toutes les lignes contenant *Unefois* ou *Deuxfois*
egrep '80[234]?86' *montexte*	cherche, dans *montexte,* toutes les lignes contenant *8086* , *80286* , *80386* ou *80486*

Exemple d'utilisation de grep

$ cat info

Le systeme d'information de l'entreprise
remplace progressivement le monde des
applications cloisonnees.
Le nombre de salaries disposant d'un
terminal augmente, l'informatique est
omnipresente.

$ grep infor info recherche les lignes contenant "infor" dans le fichier info

Le systeme d'information de l'entreprise
terminal augmente, l'informatique est

$ grep –v infor info recherche les lignes ne contenant pas "infor" dans le fichier info

remplace progressivement le monde des
applications cloisonnees.
Le nombre de salaries disposant d'un
omnipresente.

$ grep –n '^Le' info recherche les n° de lignes commençant par "Le" dans le fichier info

1 : Le systeme d'information de l'entreprise
4 : Le nombre de salaries disposant d'un

$ grep '[es]$' info recherche les lignes terminant par "e" ou "s" dans le fichier info

Le systeme d'information de l'entreprise
remplace progressivement le monde des

$ grep 'informati…' info recherche les lignes contenant "informati" dans le fichier info

terminal augmente, l'informatique est

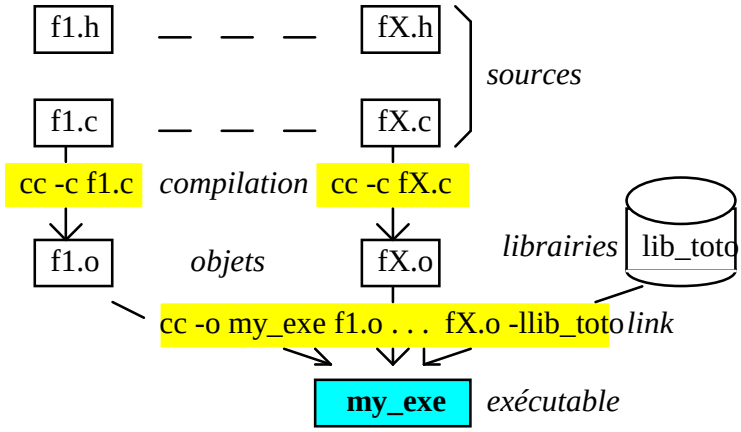
MAKE

Make

C'est un utilitaire Unix qui permet d'effectuer des travaux dépendant les uns des autres.
Le style de travaux dépendants les uns des autres le plus courant est la compilation de programmes, dont l'exécutable dépend des fichiers intermédiaires (objet), qui eux-mêmes dépendent des fichiers sources et headers.
Mais ce n'est pas la seule utilisation de make qui se trouve de plus en plus utilisé pour l'installation de modules.

Make pour la compilation

Les fichiers headers et sources servent à générer des fichiers intermédiaires (objets). Ces fichiers objets seront alors *linker* ensembles pour donner un exécutable.



ex. Lorsque l'on a plusieurs fichiers à compiler pour une application , il est intéressant de ne compiler que le source dans lequel on a changer quelque chose.
On écrit les ordres de compilation et les dépendances dans un fichier , que l'on va nommer ***makefile***.
make ne recompile que les fichiers dont la date du source est plus récente que celle des fichiers objets correspondante. Ainsi seul le fichier modifié sera recompilé et lié aux autres.

Les fichiers qui forment l'application sont donnés dans le fichier makefile.

```
<nom de l'exécutable> : <nom de l'objet 1> . . . <nom de l'objet N>
        cc <nom de l'objet 1> . . . <nom de l'objet N> -o <nom de l'exécutable>
<nom de l'objet 1> : <header 1> . . . <header J>
- - -
<nom de l'objet N> : <header 1> . . . <header K>
```

Le : indique les dépendances.

Soit une application dont l'exécutable s'appelle try , composée des fichiers sources f1 , f2 et f3.
f1 a besoin de h1.h et h2.h , f2 de h2.h.

le fichier makefile sera écrit ainsi :

```
ex.     try : f1.o f2.o f3.o
                cc  f1.o f2.o f3.o -o try
        f1.o : f1.c  h1.h  h2.h
        f2.o : f2.c  h2.h
```

```
f3.o : f3.c
```

Principe

Le principe de make c'est de travailler suivant des règles de dépendances et avec des règles de générations.

- La dépendance est indiquée dans les lignes contenants un [:] .

Une ligne comme *fichier 1 : fichier 2 . . . fichier n* indique que fichier 1 dépend de fichier 2 ... n.

Quand make trouve cette ligne il va chercher ensuite comment faire cette dépendance, c'est la génération.

- La génération est une ligne de commandes qui commence par une [***tabulation***] .

Une ligne comme *cc -o t f1.o . . . fn.o* indique que pour générer t il faut le linker à patir de f1.o ... n.

- Il existe des générations implicites

Une ligne comme *f1.o : f1.c h1.h h2.h* indique que f1.o dépend de f1.c et de h1.h et h2.h mais aussi que pour le générer il faut aussi faire *cc -c fi.c*. On peut écrire seulement la 1ère ligne, la 2ième sera générer implicitement.

nota : Pour un compilateur C++ il faudrait écrire les 2 lignes.

```
f1.o : f1.cpp h1.h h2.h
    c++ -c f1.cpp
```

car make ne sait pas comment générer de *l'objet* à partir du fichier *f1.cpp*.

Macro

Lorsque le nombre de fichiers est important , il est quelquefois plus simple d'utiliser des macros de substitution.

- macro de substitution

Elle est de la forme *<nom de macro> = chaîne_de_caractères*

ex.

```
OBJ = f1.o f2.o f3.o

try : $(OBJ )
        cc  $( OBJ ) -o try
f1.o : f1.c h1.h h2.h
f2.o : f2.c h2h
f3.o : f3.c
```

Il existe 5 macros définies automatiquement

- la macro $@

  Elle a pour valeur le nom de la cible.

  ex. *try : $ (obj)*

  alors $@ vaut *try*

- la macro $?
  Elle a pour valeur la chaîne de caractères composées des noms des dépendancesqui sont plus récentes que la cible.
  ex. *try : $ (obj)*
  alors $? vaut *obj* c.a.d. f1.o ... f3.o

- la macro $*
  Elle a pour valeur le préfixe du nom de la dépendances permettant la génération de la cible.
  ex. *try : $ (obj)*
  alors $* vaut f1 ... f3

- la macro $<
  Elle a pour valeur le nom de la dépendances permettant la génération de la cible.
  ex. *f1.o: f1.c h1.h*
  cc -c $<
  alors $< vaut f1.c

- la macro $%
  La macro est évaluée si la cible est membre d'une bibliothèque de fichiers objets de la forme *lib(file.o)*.

ex. CFLAGS =

```
libZ : libZ(file1.o) libZ(file2.o) libZ(file3.o)# ne pas ecrire libZ( file1.o ... file3.o)
.c.a:
        cc -c $(CFLAGS) $<
        ar rv $@ $*.o
        rm -f $*.o
```

- la macro de substitution de caractère
  La macro $(macro:string1=string2) remplace toutes les occurences de string1 par string2.

ex.
```
SRCS = alpha.c beta.c gamma.c delta.c
OBJS = $(SRCS:.c=.o)
```

ex d'utilisation des macros
soit à écrire ce makefile on peut le réécrire ainsi :

ex. obj = f1.o f2.o f3.o

```
try : $(obj )                       try : $(obj )
        cc  $( obj ) -o try                 cc  -o $@  $*
f1.o : f1.c h1.h h2.h               f1.o : f1.c h1.h h2.h
f2.o : f2.c h2h                     f2.o : f2.c h2h
f3.o : f3.c                         f3.o : f3.c
```

Les règles d'inférence implicites

Ces règles utilisent une "fausse" cible de nom ***.SUFFIXES*** pour déterminer le graphe de génération.

ex.

```
OBJS = f1.o f2.o f3.o
.SUFFIXES :   .o   .c
# Regles implicites
.c.o:
        cc -c $<
# Regles explicites
T : $(OBJS)
        cc -o $@  $(OBJS) -lmy_lib
f1.o : h1.h h2.h
f2.o : h2h
```

nota : En réalité la variables *.SUFFIXES* est déjà initialisé en c avec *.o .c*. Mais ce n'est pas le cas en C++, ou l'on a tout intéret à définir *.SUFFIXES .o .cpp*.

Exemple d'écriture communément utilisée

Soit à compiler des fichiers sources C nommés *F1.C , F2..C , F3.C* qui ont tous un header qui leur est associé *F1.h , F2.h , F3.h* et qui utilisent la librairie *my_lib*. L'éxécutable sera appelé *T* :

makefile

```
OBJS = F1.o F2.o F3.o
.SUFFIXES :   .o   .c
# Regles implicites
.c.o:
        cc -c $<
# Regles explicites
T : $(OBJS)
        cc -o $@  $(OBJS) -lmy_lib
F1.o : F1.h F2.h
F2.o : F2.h
F3.o : F3.h
```

Exemple complet

```
SHELL       = /bin/sh
CC          = cc
SRCS        = dcsh.c crypts.c dcrypts.c cryptu.c charok.c disrand.c msgaudit.c\
              getpws.c sentok.c
HDRS        = def.h
OBJS        = $(SRCS:.c=.o)
dcsh :$(OBJS)
    $(CC)  $(OBJS) -o dcsh
clean :
    rm $(OBJS)
edit :
    vi $(HDRS) $(SRCS)
print:
    lpr $(HDRS) $(SRCS)
cryptu.o :   /usr/include/ctype.h
dcsh.o :     /usr/include/stdio.h  /usr/include/ctype.h  /usr/include/pwd.h\
             /usr/include/signal.h  /usr/include/sys/termio.h\
             /usr/include/sys/types.h  /usr/include/sys/states.h\
             /usr/include/errno.h   def.h
getpws.o :  /usr/include/stdio.h
msgauit.o : /usr/include/stdio.h  /usr/include/sys/time.h  .usr/include/time.h
crypts.o :
dcrypts.o :
charok.o :
disrand.o :
sentok.o :
```

SED

sed

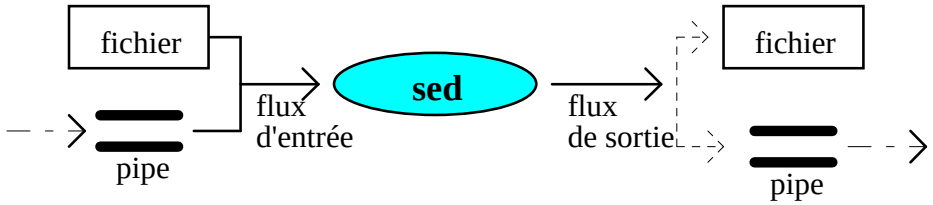
- **sed** veut dire "stream editor", c.a.d. un éditeur "dans un flux". Il prend les données dans un flux (pipe ou fichier) et fait la transformation qui lui a été demandée.
- C'est un éditeur non-interactif , non- destructif et non bufferisé.
- Les transformations sont des substitutions ou des annulations de texte en utilisant les expressions régulières.
- Les commandes sed peuvent être sauvegardées dans un script, afin d'être réutilisées répétitivement. C'est pourquoi sed est aussi appelé quelquefois éditeur "batch".
-

- La syntaxe de **sed** a la forme :

  sed *[option] commandes [fichiers ...]*

  Les options sont :
  - ◊ -n les résultats issus de sed ne sont pas envoyés sur la sortie standard.
  - ◊ -e script utilise le script (chaque script a son -e).
  - ◊ -f scriptfichier utilise le fichier scriptfichier qui stock les commandes à traiter.

⇒ sed ne change pas le fichier sur lequel on le fait travailler. Il prend le fichier et sort un flux traité.
⇒ sed est global et fait le traitement sur tout le fichier.



Utilisation de sed

- Utilisation "à la volée"
  - ♦ *une_cmd* | **sed** 's/old/new/' | *autre_cmd* change *old* par *new* dans tout le flux
- Utilisation sur un fichier
  - ♦ **sed** 's/old/new/' *fichier* change *old* par *new* dans le flux issus de fichier.
  - ♦ **sed** -e 's/old/new/' -e '/bad/d' *fichier* idem et supprime bad.
- Utilisation sur un fichier à partir d'un fichier de script
  - ♦ **sed** -f scriptfile *fichier* scriptfile contient les commandes sed.

Commandes de sed

Elles ont la forme générale suivante :

[*adresse*][,*adresse*][*!*]commande [*arguments*]

- Les adresses indiquent où commencer et où finir l'action sur le texte
- Les commandes consistent en une lettre ou un symbole
- L'argument sont ceux de la commande

Adresses

pas d'adresse	action sur toutes les lignes d'entrée
1 adresse	n'importe quelle ligne correspondant à *adresse*
2 adresses séparées par une ,	de la 1ère ligne correspondant à la 1ère adresse puis toutes les autres jusqu'à celle qui correspont à la 2ième adresse incluse
adresse suivie de !	toutes les lignes qui ne correspondent pas à l'adresse

nota : L'adresse peut être un numéro de ligne , $ pour la dernière ligne , ou une expression régulière quio est alors encadrée par / (/motif/).

ex.

s/xx/yy/g	substitue dans toutes les lignes (globale)
/BSD/d	supprime les lignes contenants *BSD*
/^BEGIN/,/^END/p@	affiche entre *BEGIN* et *END* inclus
/SAVE/!d	supprime toutes les lignes ne contenants pas *SAVE*
/^BEGIN/,/^END/!s/xx/yy/g	substitue dans toutes les lignes sauf celles comprisent entre *BEGIN* et *END*

Quand plusieurs commandes doivent être faites dans les mêmes adresses on utilise les {} pour les regrouper. La syntaxe en est particulière

```
/motif1/,/motif2/
{
cmd1
cmd2
}
```

L'{ doit être à la fin de la ligne.

L'} doit être sur une ligne seule.

Attention a ne pas mettre d'espace après les accolades.

Commandes

\# commence une ligne de commentaires dans un script sed

: *:label*
labellise une ligne dans un script pour utilisé avec *b* ou *t* (*label* doit avoir 7 caractères minimum)

= */motif/=*
écrit sur le flux de sortie le numéro de ligne de chaque lignes contenant *motif*

a *[adr]a\texte*
ajoute *texte* ***après*** chaque lignes contenant *adr*

b *[adr1],[adr2]b[label]*
donne le contrôle après *:label* dans un script, ainsi la prochaine commande sera celle qui suit *label*.

c *[adr1],[adr2]c\texte*
remplace les lignes de *adr1* à *adr2* par *texte*

d *[adr1],[adr2]d*

supprime les lignes de *adr1* à *adr2* (elles ne seront pas passées dans le flux de sortie)

```
# supprime toutes les lignes vides
/^$/d
```

g *[adr1],[adr2]g*

place la copie (voir h ou H) entre les motifs de *adr1* à *adr2*

```
# stocke toutes les lignes contenant Item et les met dans la
ligne à remplacer
/Item:/H
/<remplace cette ligne par la liste de l'item>/g
```

G *[adr1],[adr2]G*

idem g sauf que les lignes sont lues d'abord

```
# déplace toutes les lignes contenant Item
/Item:/{
H
d
}
/<toutes les item>/G
```

h *[adr1],[adr2]h*

copie entre les motifs de *adr1* à *adr2* et le met dans un buffer

```
# edit une ligne , affiche le changement , reaffiche
l'original
/UNIX/{
h
s/.* UNIX \(.*\) .*/\1:/
p
x
}
```

exécution

```
        entrée    ceci décrit la commande UNIX  ls.
                  ceci décrit la commande UNIX  cp.
        sortie    ls
                  ceci décrit la commande UNIX  ls.
                  cp
                  ceci décrit la commande UNIX  cp.
```

H *[adr1],[adr2]H*

ajoute le contenu de l'espace délimité par les motifs dans le buffer. H est comme une copie incrémentale.

voir les ex. de *g* et *G*

i *[adr]i\texte*

insert *texte* ***avant*** chaque ligen contenant *adr*.

l *[adr1],[adr2]l*

liste les lignes contenu entre adr1 et adr2. (les caractères non affichables sont donnés en code ASCII.

n *[adr1],[adr2]n*
lit la ligne suivante d'entrée dans l'espace délimité par les motifs.

N *[adr1],[adr2]n*
ajoute la ligne suivante dans l'espace délimité par les motifs. Les 2 lignes sont séparées par un retour chariot.

```
# merge 2 lignes (remplacement de \n par un espace)
/^\.NH/{
N
s/\n/ /
p
}
```

p *[adr1],[adr2]p*
affiche les lignes adressées.

P [a*dr1],[adr2]P*
affiche la 1ère partie, des lignes crées par la commande N.

q [*adr]q*
quitte quand adr est trouvé.

```
# affiche les 50 1eres lignes
50p
```

r [a*dr]r file*
lit le fichier *file* et le met après le motif. Exactement 1 espace sépare *r* du nom de fichier.

s [a*dr1],[adr2]s/motif/remplacement/[flags]*
remplace *motif* par *remplacement* sur chaque adresse de ligne.

flag	signification
n	remplace la n^{ième} instance de motif sur chaque ligne adressée. ($1 \leq n \leq 512$)
g	remplace toutes les instances de motif, pas seulement la 1ère
p	affiche la ligne ou la substitution a été effectuée
w *file*	écrit la ligne, dont le remplacement a été fait, dans *file*. (max 10 *file* différents)

```
# change la 3ième et 4ième " par ( et )
/fonction/{
s/"/(/3
s/"/)/4
}
```

```
# supprime toutes les " dans une ligne donnée
/titre/s/"//g
```

```
# change if par      if mais laisse ifdef
/ifdef/!s/if/    if/
```

t [a*dr1],[adr2]t[label]*

test si une substitution a été faite sur une adresse de ligne. si c'est le cas, va à la ligne marquée par *:label*. La commande t est comme un cas en C.

supposons que l'on ait :

```
ID:  1   Nom:   Arthur  Age:   45
ID:  2   Nom:   Marie   Age:
ID:  3
```

et que l'on veuille :

```
ID:  1   Nom:   Arthur  Age:   45      tel.:  ??
ID:  2   Nom:   Marie   Age:   ???     tel.:  ??
ID:  3   Nom:   ???     Age:   ???     tel.:  ??
```

```
# on est obligé de tester si le champ est présent
/ID/{
s/ID:  .*  Nom:  .*  Age:   .*/&  tel.:  ??/p
t
s/ID:  .*  Nom:  .*/&  Age:   .*  tel.:  ??/p
t
s/ID:  .*/&   Nom:  .*  Age:   .*  tel.:  ??/p
}
```

w [a*dr1],[adr2]w file*

ajoute le contenu d'entre les adr au fichier *file*.

x [a*dr1],[adr2]x*

echange le contenu d'entre les adr avec le contenu du buffer.

voir h.

y [a*dr1],[adr2]y/abc/xyz/*

change les caractères. a deviens x , b deviens y , c deviens z.

```
# change 1, 2, 3 en A, B, C, ...
/^item [1-9]/y/i123456789/IABCDEFGHI/
```

Exemple de sed

$ echo $PATH

/usr/sbin:/usr/bin:/usr/X11R6/bin:/usr/ucb:/usr/ccs/bin

$ echo $PATH | sed 's/:/ /' Enlève le 1er ':'

/usr/sbin /usr/bin:/usr/X11R6/bin:/usr/ucb:/usr/ccs/bin

$ echo $PATH | sed 's/:/ /g' Enlève tous les ':'

/usr/sbin /usr/bin /usr/X11R6/bin /usr/ucb /usr/ccs/bin

$ df –k Affiche les partitions utilisées avec leurs caractéristiques

```
SysFichier          1K-blocs    Utilisé Dispo.   Util% Monté sur
/dev/sda2             8056524   1456844   6190428  20% /
/dev/sda1            15358108   3072268  12285840  21% /mnt/windows
/dev/sda5            11780152   3521024   8259128  30% /mnt/dos
```

$ df –k | awk '/^\/dev/{ printf "%s\n, $1}' Affiche juste les partitions utilisées

```
/dev/sda2
/dev/sda1
/dev/sda5
```

$ df –k | awk '/^\/dev/{ printf "%s\n, $1}' | sed –e 's/[0-9]$/0/' Affiche et renomme les partitions

```
/dev/sda0
/dev/sda0
/dev/sda0
```

$ df –k | awk '/^\/dev/{ printf "%s\n, $1}' | sed –e 's/[0-9]$/0/' | sort | uniq

```
/dev/sda0
```

Le SHELL

Introduction

Le Shell est l'interface de commande d'Unix. Il interprète les commandes de l'utilisateur et les lancent s'il peut le faire.
Il existe plusieurs Shell :

- Le Bourne Shell : *sh* : qui a été le premier Shell existant pour Unix
- Le C-Shell : *csh* : issu de Berckley et fait par Mac Neally actuel patron de SUN
- Le Korn Shell : *ksh* : fait par David Korn
- Le Bash : *bash* : Bourne Again Shell
- etc.

Chacun de ces Shell a des possibilités et des propriétés plus ou moins intéressantes.
En général, sur une machine Unix on peu choisir son Shell de travail il suffit de le demander à l'administrateur. Il change alors le type de Shell que l'utilisateur a par défaut dans le fichier /etc/passwd.

Nota : Pour voir quel *Shell* "on a" il suffit de faire[1] *grep "votre_nom_de_login" /etc/passwd* et de regarder dans la ligne le nom du Shell que l'administrateur vous a donné.

Le Shell permet d'exécuter :

- des commandes internes : cd , pwd, echo , etc.
- des commandes externes : ls , cat , etc.
- un langage de programmation par
  - ♦ l'écriture de commande complexes

ex. $ for i in 1 2 3; do echo salut; done

```
salut
salut
salut
$
```

  - ♦ la possibilité de construire des fichiers de commandes complexes appelés script-shell

ex. on écrit dans un fichier *salutation* (ne pas oublier de faire un *chmod +x salutation*)

```
for i in 1 2 3
do echo salut
done
```

quand on exécute le fichier *salutation* on obtient ;

```
$ salutation
salut
salut
salut
$
```

Caractères spéciaux du Shell

;	sépare les instructions successives d'une ligne de commande
( )	servent à regrouper des commandes
>	redirection de la sortie standard
>>	redirection de la sortie standard sans écrasement (en mode append)
2>	redirection de la sortie erreur standard

[1] Grâce à la commande *cut* on peut avoir le nom du shell seulement. Il faut faire:
- cut –d: -f1,7 /etc/passwd | grep "nom_login" | cut –d/ -f3 ou
- ps | cut –d" " –f10 | grep "sh" (attention on l'a 3 fois à cause des *fork* dus aux |)

2>> redirection de la sortie erreur standard sans écrasement
< redirection de l'entrée standard
& lancement d'un processus en arrière-plan
| Communication par pipe entre 2 processus

Les caractères suivants sont aussi utiles dans le Shell
commentaire jusqu'en fin de ligne
\ inhibiteur de caractère qui le suit
'xxx' délimiteur de chaîne, les variables qu'elle contient ne sont pas évaluées
"xxx" délimiteur de chaîne, les variables qu'elle contient sont évaluées
`xxx` ce qui est entre les 2 accents est interprété comme une commande et est remplacé par la chaîne

```
$ echo "bonjour >toto"
bonjour >toto
$ echo bonjour \>toto
bonjour >toto
$ echo bonjour > toto
$cat toto
bonjour
$
```

Les variables

Une variables est de la forme : ***nom=valeur***
La valeur de la variables nom est donnée par *$nom* ou par *${nom}*.

```
$ x=efeg
$ echo x
efeg
$ echo xerfg
                    # il n'y a rien dans la variable
$ echo ${x}erfg
efegerfg            # il y a concaténation de la valeur de la variable et des caractères qui suivent
$
```

Les accents

Les caractères spéciaux ' " et ` permettent de délimité une chaîne de caractères.

```
$ a=truc
$ x="machin$a"
$ echo $x
machintruc          # concaténation
$ y='machin$a'
$ echo $y
machin$a            #non interprétation
```

Mécanisme de l'accent grave

```
$ a=`pwd`
$ echo $a
/home/projet
```

La commande de lecture

read permet d'affecter à une ou plusieurs variables des valeurs lues sur l'entrée standard.

```
$ read  nom adresse
papa_noel   1 rue des cadeaux
$ echo nom
papa_noel           # la 1ere chaîne
$ echo adresse
1 rue des cadeaux  # tout le reste des chaînes
$
```

Les variables d'environnement

Les variables prédéfinies de l'environnement peuvent se retrouver ainsi que leurs significations en faisant :

man *environ*

La commande *set* permet de visualiser toutes les variables d'environnement du système

La commande *unset* permet de supprimer une variable de l'environnement

```
$ a=bonjour
$ echo $a
bonjour
$ unset a
$ echo a

$
```

Variables exportables

Lorsqu'un nouveau processus est crée par un fork, il hérite d'une partie des variables du Shell. Les variables qui sont ainsi transmises sont dites exportables. La liste des variables exportables peut être connues par la commance ***env***.

```
$ PS1='§ '   # on change le prompt
§ sh          # on appel un nouveau shell
$ echo $PS1
$
§             # on est sortie du nouveau shell par Ctrl-D on retrouve notre PS1 modifié
```

```
$ PS1='§ '
§ export PS1
§ sh          # on appel un nouveau shell
§ echo $PS1
```

§ # on a bien le nouveau prompt
§ # on est sortie du nouveau shell par Ctrl-D on retrouve notre PS1 modifié

S'il existe dans la dir. de l'utilisateur, le fichier .profile sert à exécuter automatiquement à chaque login les commandes qui sont incluses dans ce fichier.

Exécution d'un fichier de commandes

Un fichier de commande s'exécute, s'il est exécutable (chmod +x nom_du_fichier), en donnant son nom.

1) la variable PATH contient :. Ce qui indique que la dir. courante est prise en compte
   alors *nom_du_fichier* suffit
2) la variable PATH ne contient pas :.
   alors *./nom_du_fichier* doit être taper afin d'indiquer que le fichier est à exécuter dans la dir. courante.

Paramètres d'un fichier de commande

Tout Shell peut transmettre à une commande une liste de chaîne de caractères en paramètre.
Les variables sont :

0	le nom de la commande
1	le 1er paramètre
2	le 2ieme paramètre
...	
9	le 9ieme paramètre
*	la liste de chaîne de caractères
#	le nombre de chaînes présentes

Il peut y avoir plus de 9 paramètres mais à 1 instant seul les 9 premiers peuvent être désignés individuellement.
La commande shift permet de décaler la liste : 1 vaut ce que valait 2 , etc. 1 est perdue.

```
$ cat com                    # fichier de commande com
echo "commande : $0"
echo "nbr arg : $#"
echo  "liste des arg : $*"
echo "le 2iem arg : $2"
```

```
$ com a b c d e              # exec de com avec des arguments
commande : com
nbr arg : 5
liste des arg : a b c d e
le 2iem arg : b
$
```

Syntaxe du langage Shell

expr

La commande expr prend la suite des arguments comme une expression. Elle l'évalue et affiche son résultat. Les différents opérandes doivent être séparés par des espaces.

```
$ expr 9 + 3
12
$
$ expr 5 \* \( 6 + 2 \)  # ne pas oublier le \ pour les caractères *, (, )
40
$
```

Les opérandes peuvent être des chaînes de caractères , des variables , des constantes.
Les opérations possible sont
ou logique : *op1* | *op2* a pour valeur *op1 si elle n'est pas nulle* sinon *op2 si elle n'est pas nulle* sinon *0*
et logique : *op1* & *op2* a pour valeur *op1 si op1 et op2 ne sont pas nulle* sinon *0*
comparaisons : < , > , = , >= , <= , ! : le résultat vaut 1 si la comparaison est vrai 0 sinon
arithmétique : + , - , * , / , %

```
$ echo $HOME
/root
$ expr $HOME = /root
1
$ expr $HOME > /usr
0
$
```

Les structures de contrôle du Shell

Les test peuvent se faire sur :

- Les chaînes de caractères
- Les chaînes numériques
- Les fichiers
- Les terminaux

La syntaxe est : *test expression* ou *[expression]*
Le la valeur du résultat de test est **0 si le test est vrai** et différent de 0 sinon.

Test sur les chaînes de caractères

test -z chaîne	vrai si chaîne est vide
test -n chaîne	vrai si chaîne n'est pas vide
test chaîne1 = chaîne2	vrai si chaîne2 est identique à chaîne2
test chaîne1 != chaîne2	vrai si chaîne2 est différente de chaîne2

écriture avec test

```
if test `pwd` = $HOME
then  echo vous etes dans votre home dir
fi
```

écriture avec [ ]

```
if [ `pwd` = $HOME ]
then  echo vous etes dans votre home dir
fi
```

test c1 -eq c2 egalité (equal)

test c1 -ne c2	non égalité (not equal)
test c1 -lt c2	plus petit (less than)
test c1 -le c2	plus petit ou égal (less or equal)
test c1 -gt c2	plus grand (greater than)
test c1 -ge c2	plus grand ou egal (greater or equal)

```
if test $# -lt 1
then  echo commande + 1 paramètre  , il manque le paramètre
fi
```

Test sur les fichiers

Test sur le type

test -p nom	vrai si nom est un pipe
test -f nom	vrai si nom est un fichier
test -d nom	vrai si nom est une dir.
test -c nom	vrai si nom est un fichier spécial en mode caractère
test -b nom	vrai si nom est un fichier spécial en mode bloc

Test sur les droits d'accès

test -r nom	vrai si le fichier nom est autorisé en lecture
test -w nom	vrai si le fichier nom est autorisé en écriture
test -x nom	vrai si le fichier nom est autorisé en exécution

Test sur la taille

test -s nom	vrai si le fichier nom est de taille non nulle

Test d'un rattachement d'un descripteur à un terminal

test -t nom	vrai si le descripteur nom est attaché à un terminal

Opérateurs sur les expressions

On peut combiné les tests avec les opérateurs suivants :

!	la négation
-a	la conjonction
-o	la disjonction

ex. courant de création conditionnelle de dir. si elle n'existe pas

```
if  [ ! -d  $HOME/exo/exo3 ]
then  mkdir -p $HOME/exo/exo3
fi
```

Les conditionnelles

La conditionnelle simple

if commande1
then commande2
else commande3
fi

Commande1 est exécutée ; si son code de retour est vrai (égal à 0) , commande2 est exécutée sinon c'est commande3 qui est exécutée.

ex. écriture d'une commande affich qui affiche soit une dir. soit un fichier

```
if  [ $# -ne 1 ] ; then  echo "syntaxe : $0 nom_de_fichier" ; exit 1 ; fi
if   test -f  $1
then  cat $1
else if  test  -d  $1
      then  ls $1
      else echo  $1 n\'existe pas ou est un fichier spécial          # n\' : le \ pour inhiber le '
      fi
fi
```

L'aiguillage

case chaîne in
  $motif_1$**)** $commande_1$ **;;**
  ...
  $motif_n$**)** $commande_n$ **;;**
esac

ex. écriture d'une commande qui choisie le bon compilateur suivant l'extension du fichier, C pour .c , Pascal pour .p , Fortran pour .f.

```
case  $#   in
0) echo -n "fichier a compiler : " ; read nom ; set $nom ;;
esac
case  $1   in
*.c) cc $1 ;;
*.p) pc $1 ;;
*.f)  ff  $1 ;;
*)    echo compil de $1 impossible ;;
esac
```

Les boucles

for

for variable **in** chaîne chaîne2 ...
do commande
done

Affecte successivement à variable les valeurs de chaîne1 chaîne2 ... en exécutant à chaque fois la commande

- une liste explicite — for i in *1 2 coucou vert noir*
- la valeur d'une variable — for i in $var
- le résultat d'une commande — for i in `ls -a` (attention aux accents)
- le caractère * qui désigne les fichiers dans la dir. — for i in *

while

while commande1
do commande2
done

Exécute successivement commande1 puis commande2 tant que commande1 est **vrai**

ex. commande qui demande oui ou non à l'utilisateur tand qu'il n'a pas répondu

```
echo -n 'répondre par oui ou par non :'
read réponse
while [ "$réponse" != oui -a "$réponse" != non ]
do echo -n 'il faut répondre par oui ou par non :'
read réponse
done
```

nota : il faut écrire "$réponse" pour que même quand l'utilisateur tape ↵ on ait ["" != oui -a "" != non], sinon on aurait eu [!= oui -a != non] qui n'avait aucun sens.

until

until commande1
do commande2
done

Exécute successivement commande1 puis commande2 tant que commande1 est **faux**

ex. idem que précédent sauf qu'un signal sonore est envoyé

```
echo -n 'répondre par oui ou par non :'
read réponse
until [ "$réponse" != oui -a "$réponse" != non ]
do echo "^G répondre par oui ou par non :'        #^G signal sonore
read réponse
done
```

Table des matières