1. Les commandes grep et find

1.1 Les expressions régulières

On a vu auparavant ce qu'étaient les métacaractères. Les expressions régulières sont aussi des suites de caractères permettant de faire des sélections. Elles fonctionnent avec certaines commandes comme **grep**.

Les différentes expressions régulières sont :

- **^** début de ligne
- **.** un caractère quelconque
- **$** fin de ligne
- **x*** zéro ou plus d'occurrences du caractère **x**
- **x+** une ou plus occurrences du caractère **x**
- **x?** une occurrence unique du caractère **x**
- **[...]** plage de caractères permis
- **[^...]** plage de caractères interdits
- **\ {n\}** pour définir le nombre de répétition **n** du caractère placé devant

Exemple l'expression **[a-z][a-z] *** cherche les lignes contenant au minimum un caractère en minuscule. **[a-z]** caractère permis, **[a-z]*** recherche d'occurrence des lettres permises.

L'expression **[0-9]\ {4\}$** a pour signification, du début à la fin du fichier **$**, recherche les nombres**[0-9]** de 4 chiffres \ **{4\}**.

1.2 La commande grep

La commande **grep** permet de rechercher une chaîne de caractères dans un fichier. Les options sont les suivantes :

- **-v** affiche les lignes ne contenant pas la chaîne
- **-c** compte le nombre de lignes contenant la chaîne
- **-n** chaque ligne contenant la chaîne est numérotée
- **-x** ligne correspondant exactement à la chaîne
- **-w** lignes où le mot apparaît tel quel
- **-l** affiche le nom des fichiers qui contiennent la chaîne

Exemple avec le fichier **carnet-adresse** :

```
olivier:29:0298333242:Brest
marcel:13:0466342233:Gardagnes
myriam:30:0434214452:Nimes
yvonne:92:013344433:Palaiseau
```

On peut utiliser les expressions régulières avec **grep**. Si on tape la commande :

```
grep ^[a-d] carnet-adresse
```

On va obtenir tous les lignes commençant par les caractères compris entre a et d. Dans notre exemple, on n'en a pas, d'où l'absence de sortie.

```
grep Brest carnet-adresse
```

Permet d'obtenir les lignes contenant la chaîne de caractère Brest, soit :

```
olivier:29:0298333242:Brest
```

Il existe aussi les commandes fgrep et egrep équivalentes.

1.3 La commande find

La commande **find** permet de retrouver des fichiers à partir de certains critères. La syntaxe est la suivante :

```
find <répertoire de recherche> <critères de recherche>
```

Les critères de recherche sont les suivants :

- **-name** recherche sur le nom du fichier,
- **-perm** recherche sur les droits d'accès du fichier,
- **-links** recherche sur le nombre de liens du fichier,
- **-user** recherche sur le propriétaire du fichier,
- **-group** recherche sur le groupe auquel appartient le fichier,
- **-type** recherche sur le type (d=rép., c=car., f=fichier normal),
- **-size** recherche sur la taille du fichier en nombre de blocs (1 bloc=512octets),
- **-atime** recherche par date de dernier accès en lecture du fichier,
- **-mtime** recherche par date de dernière modification du fichier,
- **-ctime** recherche par date de création du fichier.

On peut combiner les critères avec des opérateurs logiques :

- **critère1 critère2** ou **critère1 -a critère2** correspond au **et** logique,
- **!critère** non logique,
- **\ (critère1 -o critère2\)** ou logique,

La commande **find** doit être utilisé avec l'option **-print**. Sans l'utilisation de cette option, même en cas de réussite dans la recherche, **find** n'affiche rien à la sortie standard (l'écran, plus précisément le shell).

La commande **find** est récursive, c'est à dire où que vous tapiez, il va aller scruter dans les répertoires, et les sous répertoires qu'il contient, et ainsi de suite.

Recherche par nom de fichier

Pour chercher un fichier dont le nom contient la chaîne de caractères **toto** à partir du répertoire **/usr**, vous devez tapez :

```
find /usr -name toto -print
```

En cas de réussite, si le(s) fichier(s) existe(nt), vous aurez comme sortie :

```
toto
```

En cas d'échec, vous n'avez rien.

Pour rechercher tous les fichiers se terminant par **.c** dans le répertoire **/usr**, vous taperez :

```
find /usr -name " *.c " -print
```

Vous obtenez toute la liste des fichiers se terminant par **.c** sous les répertoires contenus dans **/usr** (et dans **/usr** lui même).

Recherche suivant la date de dernière modification

Pour connaître les derniers fichiers modifiés dans les 3 derniers jours dans toute l'arborescence (/), vous devez taper :

```
find / -mtime 3 -print
```

Recherche suivant la taille

Pour connaître dans toute l'arborescence, les fichiers dont la taille dépasse 1Mo (2000 blocs de 512Ko), vous devez taper :

```
find / -size 2000 -print
```

Recherche combinée

Vous pouvez chercher dans toute l'arborescence, les fichiers ordinaires appartenant à olivier, dont la permission est fixée à 755, on obtient :

```
find / -type f -user olivier -perm 755 -print
```

Redirection des messages d'erreur

Vous vous rendrez compte assez rapidement qu'en tant que simple utilisateur, vous n'avez pas forcément le droit d'accès à un certain nombre de répertoires, par conséquent, la commande find peut générer beaucoup de messages d'erreur (du genre permission denied), qui pourraient noyer l'information utile. Pour éviter ceci, vous pouvez rediriger les messages d'erreur dans un fichier poubelle (comme **/dev/null**), les messages d'erreur sont alors perdus (rien ne vous empêche de les sauvegarder dans un fichier, mais ça n'a aucune utilité avec la commande **find**).

```
find . -name bobo -print 2>/dev/null
```

Recherche en utilisant les opérateurs logiques

Si vous voulez connaître les fichiers n'appartenant pas à l'utilisateur **olivier**, vous taperez :

```
find . ! -user olivier -print
```

! -user olivier, est la négation de **-user olivier**, c'est à dire c'est tous les utilisateurs sauf **olivier**.

Recherche des fichiers qui ont pour nom **a.out** et des fichiers se terminant par **.c**. On tape :

```
find . \  ( -name a.out -o -name " *.c " \  ) -print
```

On recherche donc les fichiers dont le nom est **a.out** ou les fichiers se terminant par ***.c**, une condition ou l'autre.

Recherche des fichiers qui obéissent à la fois à la condition a pour nom **core** et à la condition a une taille supérieure à 1Mo.

```
find . \  (-name core -a size +2000 \  ) -print
```

Les commandes en option

L'option **-print** est une commande que l'on passe à **find** pour afficher les résultats à la sortie standard. En dehors de **print**, on dispose de l'option **-exec**. **find** couplé avec **exec** permet d'exécuter une commande sur les fichiers trouvés d'après les critères de recherche fixés. Cette option attend

comme argument une commande, celle ci doit être suivi de **{}\ ;**.

Exemple recherche des fichiers ayant pour nom **core**, suivi de l'effacement de ces fichiers.

```
find . -name core -exec rm {}\   ;
```

Tous les fichiers ayant pour nom core seront détruits, pour avoir une demande de confirmation avant l'exécution de rm, vous pouvez taper :

```
find . -name core -ok rm {}\   ;
```

Autres subtilités

Une fonction intéressante de **find** est de pouvoir être utilisé avec d'autres commandes UNIX. Par exemple:

```
find . -type f -print | xargs grep toto
```

En tapant cette commande vous allez rechercher dans le répertoire courant tous les fichiers normaux (sans les répertoires, fichiers spéciaux), et rechercher dans ces fichiers tous ceux contenant la chaîne **toto**.

2. Expressions régulières et sed

2.1 Les expressions régulières

Présentation

Une expression régulière (en anglais Regular Expression ou RE) sert à identifier une chaîne de caractère répondant à un certain critère (par exemple chaîne contenant des lettres minuscules uniquement). L'avantage d'une expression régulière est qu'avec une seule commande on peut réaliser un grand nombre de tâche qui seraient fastidieuses à faire avec des commandes UNIX classiques. Les commandes ed, vi, ex, sed, **awk**, **expr** et **grep** utilisent les expressions régulières.

L'exemple le plus simple d'une expression régulière est une chaîne de caractères quelconque **toto** par exemple. Cette simple expression régulière va identifier la prochaine ligne du fichier à traiter contenant une chaîne de caractère correspondant à l'expression régulière.

Si l'on veut chercher une chaîne de caractère au sein de laquelle se trouve un caractère spécial (/, *****, **$**, **.**, **[**, **]**, **{**, **}**, **!**, entre autres) (appelé aussi métacaractère), on peut faire en sorte que ce caractère ne soit pas interprété comme un caractère spécial mais comme un simple caractère. Pour cela vous devez le faire précéder par \ (backslash). Ainsi si votre chaîne est **/dev**, pour que le **/** ne soit pas interprété comme un caractère spécial, vous devez tapez \ /dev pour l'expression régulière.

Le métacaractère .

Le métacaractère **.** remplace dans une expression régulière un caractère unique, à l'exception du caractère retour chariot (**\ n**). Par exemple chaine. va identifier toutes les lignes contenant la chaine chaine suivit d'un caractère quelconque unique. Si vous voulez identifier les lignes contenant la chaîne **.cshrc**, l'expression régulière correspondante est **\ .cshrc**

Les métacaractères [ ]

Les métacaractères **[]** permettent de désigner des caractères compris dans un certain intervalle de

valeur à une position déterminée d'une chaîne de caractères. Par exemple **[Ff]raise** va identifier les chaînes **Fraise** ou **fraise**, **[a-z]toto** va identifier une chaîne de caractère commençant par une lettre minuscule (intervalle de valeur de **a** à **z**) et suivi de la chaîne **toto** (**atoto**, **btoto**, ..., **ztoto**). D'une manière plus générale voici comment **[]** peuvent être utilisés:

- **[A-D]** intervalle de **A** à **D** (**A**, **B**, **C**, **D**) **par exemple bof[A-D] donne bofA**, **bofB**, **bofC**, **bofD**
- **[2-5]** intervalle de **2** à **5** (**2**, **3**, **4**, **5**) par exemple **12[2-5]2 donne 1222**, **1232**, **1242**, **1252**
- **[2-56]** intervalle de **2** à **5** et **6** (et non pas **56**) (**2**, **3**, **4**, **5**, **6**) **par exemple 12[2-56]2 donne 1222**, **1232**, **1242**, **1252**, **1262**
- **[a-dA-D]** intervalle de **a** à **d** et **A** à **D** (**a**, **b**, **c**, **d**, **A**, **B**, **C**, **D**) **par exemple z[a-dA-D]y donne zay**, **zby**, **zcy**, **zdy**, **zAy**, **zBy**, **zCy**, **zDy**
- **[1-3-]** intervalle de **1** à **3** et **-** (**1**, **2**, **3**, **-**) **par exemple [1-3-]3 donne 13**, **23**, **33**, **-3**
- **[a-cI-K1-3]** intervalle de **a** à **c**, **I** à **K** et **1** à **3** (**a**, **b**, **c**, **I**, **J**, **K**, **1**, **2**, **3**)

On peut utiliser **[]** avec un pour identifier le complèment de l'expression régulière. En français pour identifier l'opposé de l'expression régulière. Vous avez toujours pas compris ? Voici un exemple: **[0-9]toto** identifie les lignes contenant une chaîne **toto**, le caractère juste avant ne doit pas être un chiffre (exemple **atoto**, **gtoto** mais pas **1toto**, **5toto**). Autre exemple **[a-zA-Z]** n'importe quel caractère sauf une lettre minuscule ou majuscule. Attention à la place de , si vous tapez **[1-3]**, c'est équivalent aux caractères **1**, **2**, **3** et .

Les métacaractères ^ et $

Le métacaractère ^ identifie un début de ligne. Par exemple l'expression régulière **^a** va identifier les lignes commençant par le caractère **a**.

Le métacaractère **$** identifie une fin de ligne. Par exemple l'expression régulière **a$** va identifier les lignes se terminant par le caractère **a**.

L'expression régulière **chaine$** identifie les lignes qui contiennent strictement la chaîne **chaine**.

L'expression régulière **$** identifie une ligne vide.

Le métacaractère *

Le métacaractère ***** est le caractère de répétition.

L'expression régulière **a*** correspond aux lignes comportant 0 ou plusieurs caractère **a**. Son utilisation est à proscrire, car toutes les lignes, même celles ne contenant pas le caractère **a**, répondent aux critères de recherche. **x*** est une source de problèmes, il vaut mieux éviter de l'employer.

L'expression régulière **aa*** correspond aux lignes comportant 1 ou plusieurs caractères **a**.

L'expression régulière **.*** correspond à n'importe quelle chaîne de caractères.

L'expression régulière **[a-z][a-z]*** va chercher les chaînes de caractères contenant 1 ou plusieurs lettres minuscules (de **a** à **z**).

L'expression régulière **[ ][ ]*** est équivalent à tout sauf un blanc.

Les métacaractères \ (\)

Pour le traitement complexe de fichier, il est utile parfois d'identifier un certain type de chaîne pour pouvoir s'en servir dans la suite du traitement comme un sous programme. C'est le principe des sous chaînes, pour mémoriser une sous chaîne, on utilise la syntaxe \ **(expression régulière)**, cette sous chaîne sera identifié par un chiffre compris par 1 et 9 (suivant l'ordre de définition).

Par exemple \ **([a-z][a-z]*)** est une sous chaîne identifiant les lignes contenant une ou plusieurs lettres minuscules, pour faire appel à cette sous chaîne, on pourra utiliser \ **1**. Voir dans le paragraphe **sed** pour un exemple.

2.2 La commande sed

Présentation

sed est éditeur ligne non interactif, il lit les lignes d'un fichier une à une (ou provenant de l'entrée standard) leur applique un certain nombre de commandes d'édition et renvoie les lignes résultantes sur la sortie standard. Il ne modifie pas le fichier traité, il écrit tout sur la sortie standard.

sed est une évolution de l'éditeur **ed** lui même précurseur de **vi**, la syntaxe n'est franchement pas très conviviale, mais il permet de réaliser des commandes complexes sur des gros fichiers.

La syntaxe de **sed** est la suivante:

```
sed -e 'programme sed' fichier-a-traiter
```
ou
```
sed -f fichier-programme fichier-a-traiter
```

Vous disposez de l'option **-n** qui supprime la sortie standard par défaut, **sed** va écrire uniquement les lignes concernées par le traitement (sinon il écrit tout même les lignes non traitées). L'option **-e** n'est pas nécessaire quand vous avez une seule fonction d'édition.

La commande **sed** est une commande très riche, ne vous sont présentées ici que les fonctions les plus courantes, pour plus de détails faites un **man sed** et/ou **man ed**.

La fonction de substitution s

La fonction de substitution **s** permet de changer la première ou toutes les occurences d'une chaîne par une autre. La syntaxe est la suivante:

- **sed "s/toto/TOTO/" fichier** va changer la première occurence de la chaîne **toto** par **TOTO** (la première chaîne **toto** rencontrée dans le texte uniquement)
- **sed "s/toto/TOTO/3" fichie**r va changer la troisième occurence de la chaîne **toto** par **TOTO** (la troisième chaîne **toto** rencontrée dans le texte uniquement)
- **sed "s/toto/TOTO/g" fichier** va changer toutes les occurences de la chaîne **toto** par **TOTO** (toutes les chaînes **toto** rencontrées sont changées
- **sed "s/toto/TOTO/p" fichier** en cas de remplacement la ligne concernée est affichée sur la sortie standard (uniquement en cas de substitution)
- **sed "s/toto/TOTO/w resultat" fichier** en cas de substitution la ligne en entrée est inscrite dans un fichier résultat
- La fonction de substitution peut évidemment être utilisée avec une expression régulière.

- **sed -e "s/[Ff]raise/FRAISE/g" fichier** substitue toutes les chaînes **Fraise** ou **fraise** par **FRAISE**

La fonction de suppression d

La fonction de suppression **d** supprime les lignes comprises dans un intervalle donné. La syntaxe est la suivante:

```
sed "20,30d" fichier
```

Cette commande va supprimer les lignes 20 à 30 du fichier fichier. On peut utiliser les expressions régulières:

```
sed "/toto/d" fichier
```

Cette commande supprime les lignes contenant la chaîne **toto**. Si au contraire on ne veut pas effacer les lignes contenant la chaîne **toto** (toutes les autres sont supprimées), on tapera:

```
sed "/toto/!d" fichier
```

En fait les lignes du fichier d'entrée ne sont pas supprimées, elles le sont au niveau de la sortie standard.

Les fonctions p, l, et =

La commande **p** (print) affiche la ligne sélectionnée sur la sortie standard. Elle invalide l'option **-n**.

La commande **l** (list) affiche la ligne sélectionnée sur la sortie standard avec en plus les caractères de contrôles en clair avec leur code ASCII (deux chiffres en octal).

La commande = donne le numéro de la ligne sélectionnée sur la sortie standard.

Ces trois commandes sont utiles pour le débogage, quand vous mettez au point vos programmes **sed**.

```
sed "/toto/=" fichier
```

Cette commande va afficher le numéro de la ligne contenant la chaîne **toto**.

Les fonctions q, r et w

La fonction **q** (quit) va interrompre l'exécution de **sed**, la ligne en cours de traitement est affichée sur la sortie standard (uniquement si **-n** n'a pas été utilisée).

La fonction **r** (read) lit le contenu d'un fichier et écrit le contenu sur la sortie standard.

La fonction **w** (write) écrit la ligne sélectionnée dans un fichier.

```
sed "/^toto/w resultat" fichier
```

Cette commande va écrire dans le fichier resultat toutes les lignes du fichier **fichier** commençant par la chaîne **toto**.

Les fonctions a et i

La fonction **a** (append) va placer un texte après la ligne sélectionnée. La syntaxe est la suivante:

```
a\
le texte
```

La fonction **i** (insert) va placer un texte avant la ligne sélectionnée. La syntaxe est la suivante:

```
i\
le texte
```

Si votre texte tient sur plusieurs lignes la syntaxe pour le texte est la suivante:

```
ligne 1 du texte\
ligne 2 du texte  \
ligne n du texte  \
dernière ligne
```

Concrètement vous pouvez appeler la fonction **i** ou **a** dans un fichier de commande de **sed**. Par exemple, soit votre fichier **prog.sed** suivant:

```
1i\
début du traitement
s/[tT]oto/TOTO/g
$a  \     fin du traitement\
de notre fichier
```

On exécute la commande en tapant:

```
sed -f prog.sed fichier-a-traiter
```

prog.sed a pour effet d'inscrire avant la première ligne (**1i**) le texte "**début de traitement**", et après la dernière ligne (**$a**) le texte "**fin du traitement** (retour à la ligne) **de notre fichier**".

sed et les sous chaînes

La commande:

```
sed -e "s/\  ([0-9][0-9]*\  )/aa\  1aa/" fichier
```

La sous expression (sous chaîne) \ **([0-9][0-9]*\)** désigne un ou plusieurs chiffres, chacun sera entouré des caractères **aa**. La chaîne **to2to** deviendra **toaa2aato**.

3. La commande awk

3.1 Présentation

Présentation et syntaxe

awk est une commande très puissante, c'est un langage de programmation a elle tout seule qui permet une recherche de chaînes et l'exécution d'actions sur les lignes sélectionnées. Elle est utile pour récuperer de l'information, générer des rapports, transformer des données entre autres.

Une grande partie de la syntaxe a été empruntée au langage c, d'ailleurs **awk** sont les abréviations de ces 3 créateurs dont k pour Kernighan, un des inventeurs du c.

La syntaxe de **awk** est la suivante:

```
awk [-F] [-v var=valeur] 'programme' fichier
```
ou
```
awk [-F] [-v var=valeur] -f fichier-config fichier
```

L'argument **-F** doit être suivi du séparateur de champ (**-F:** pour un "**:**" comme séparateur de champ).

L'argument **-f** suivi du nom du fichier de configuration de awk.

L'argument **-v** définit une variable (**var** dans l'exemple) qui sera utilisée par la suite dans le programme.

Un programme **awk** possède la structure suivante: **critère de sélection d'une chaîne {action}**, quand il n'y a pas de critère c'est que l'action s'applique à toutes les lignes du fichier.

Exemple:

```
awk -F":" '{print $NF}' /etc/passwd
```

Il n'y a pas de critères, donc l'action s'applique à toutes les lignes du fichier **/etc/passwd**. L'action consiste à afficher le nombre de champ du fichier. **NF** est une variable prédéfinie d'awk, elle est égale au nombre de champs dans une ligne.

Généralement on utilisera **awk** en utilisant un script.

```
#!/bin/sh
awk [-F] [-v var=valeur] 'programme' $1
```

Vous appelerez votre script **mon-script.awk**, lui donnerez des droits en exécution (**755** par exemple), et l'appelerez ainsi:

```
mon-script.awk fichier-a-traiter
```

Dans la suite du cours, on utilisera **awk** en sous entendant que celui-ci est à insérer dans un script.

Le quote **'** se trouve sur un clavier azerty standard avec le 4 et éventuellement l'accolade gauche.

ATTENTION: ils existent plusieurs "variétés" de **awk**, il se pourrait que certaines fonctions ou variables systèmes qui vous sont présentées dans ce cours n'existent pas sur votre UNIX. Faites en sorte si vos scripts awk doivent fonctionner sur des plates-formes différentes d'utiliser **gawk** sous licence GNU qui est totalement POSIX.

J'ai constaté des grosses différences de comportement entre le awk natif qu'on soit sous HP-UX, Solaris et sous LINUX, de même quand on insère la commande dans un script, on fait appel à un shell, suivant son type (bash shell, csh, ksh, ...), vous pouvez avoir quelques surprises.

Enregistrements et champs

awk scinde les données d'entrée en enregistrements et les enregistrements en champ. Un enregistrement est une chaîne d'entrée délimitée par un retour chariot, un champ est une chaîne délimitée par un espace dans un enregistrement.

Par exemple si le fichier à traiter est **/etc/passwd**, le caractère de séparation étant "**:**", un enregistrement est une ligne du fichier, et un champ correspond au chaîne de caractère séparé par un "**:**" (login:mot de passe crypté:UID:GID:commentaires:home directory:shell).

Dans un enregistrement les champs sont référencés par **$1**, **$2**, ..., **$NF** (dernier champ). Par exemple pour **/etc/passwd $1** correspond au login, **$2** au mot de passe crypté, **$3** à l'UID, et **$NF** (ou **$7**) au shell.

L'enregistrement complet (une ligne d'un fichier) est référencé par **$0**.

Par exemple, si l'on veut voir les champs login et home directory de **/etc/passwd**, on tapera:

```
awk -F":" '{print $1,$6}' /etc/passwd
```

3.2 Critères de sélection

Présentation

Un critère peut être une expression régulière, une expression ayant une valeur chaîne de caractères, une expression arithmétique, une combinaison des expressions précédentes.

Le critère est inséré entre les chaînes **BEGIN** et **END**, avec la syntaxe suivante:

```
awk -F":" 'BEGIN{instructions} critères END{instructions}' fichier
```

BEGIN peut être suivi d'instruction comme une ligne de commentaire ou pour définir le séparateur. Exemple **BEGIN { print"Vérification d'un fichier"; FS=":"}**. Le texte à afficher peut être un résumé de l'action de **awk**. De même pour **END** on peut avoir **END{print "travail terminé"}** qui indiquera que la commande a achevé son travail. Le **END** n'est pas obligatoire, de même que le **BEGIN**.

Les expressions régulières

La syntaxe est la suivante:

```
/expression régulière/ {instructions}
$0  /expression régulière/ {instructions}
```

les instructions sont exécutées pour chaque ligne contenant une chaîne satisfaisant à l'expression régulière.

```
expression  /expression régulière/{instructions}
```

les instructions sont exécutées pour chaque ligne où la valeur chaîne de l'expression contient une chaîne satisfaisant à l'expression régulière.

```
expression !/expression régulière/ {instructions}
```

les instructions sont exécutées pour chaque ligne où la valeur chaîne de l'expression ne contient pas une chaîne satisfaisant à l'expression régulière.

Soit le fichier adresse suivant (nom, numéro de téléphone domicile, numéro de portable, numéro quelconque):

```
gwenael | 0298452223 | 0638431234 | 50
marcel | 0466442312 | 0638453211 | 31
judith | 0154674487 | 0645227937 | 23
```

L'exemple suivant vérifie que dans le fichier le numéro de téléphone domicile (champ 2) et le numéro de portable (champ 3) sont bien des nombres.

```
awk 'BEGIN { print "On vérifie les numéros de téléphone; FS="|"}
   $2 ! /^[0-9][0-9]*$/ { print "Erreur sur le numéro de téléphone domicile,
ligne n°""NR": \  n"$0}
   $3 ! /^[0-9][0-9]*$/ { print "Erreur sur le numéro de téléphone du portable,
ligne n°""NR": \  n"$0}
END { print "Vérification terminé"} ' adresse
```

BEGIN est suivi d'une instruction d'affichage qui résume la fonction de la commande, et de la définition du séparateur de champ. L'expression **$2** se référe au deuxième champ d'une ligne (enregistrement) de adresse soit le numéro de téléphone domicile, on recherche ceux qui ne

contiennent pas de chiffre (négation de contient des chiffres), en cas de succés on affichera un message d'erreur, le numéro de ligne courante, un retour à la ligne, puis le contenu entier de la ligne. L'expression **$3** se référe au troisième champ d'une ligne (enregistrement) de adresse soit le numéro du portable, on recherche ceux qui ne contiennent pas de chiffre (négation de contient des chiffres), en cas de succés on affichera un message d'erreur, le numéro de ligne courante, un retour à la ligne, puis le contenu entier de la ligne. **END** est suivi d'une instruction d'affichage indiquant la fin du travail.

Les expressions relationnelles

Un critère peut contenir des opérateurs de comparaison (**-** <, <=,==,**!**=,>=,>). Exemple avec le fichier adresse suivant:

```
awk 'BEGIN { print "On cherche lignes dont le numéro (champ 4) est supérieur à
30"; FS="|"}
 $4 > 30 { print "Numéro supérieur à 30 à la ligne n°"NR": \  n"$0}
END { print "Vérification terminé"} ' adresse
```

Combinaison de critères

Un critère peut être constitué par une combinaison booléenne avec les opérateurs ou (||), et (**&&**) et non (**!**). Exemple:

```
awk 'BEGIN { print "On cherche la ligne avec judith ou avec un numéro inférieur
à 30"; FS="|"}
 $1 = = "judith" || $4 < 30 { print "Personne "$1" numéro "$4" ligne n°"NR": \
n"$0}
END { print "Vérification terminé"} ' adresse
```

Plage d'enregistrement délimitées par des critères

La syntaxe est la suivante **critère1,critère2 {instructions}**. Les instructions sont exécutées pour toute les lignes entre la ligne répondant au critère1 et celle au critère2. L'action est exécutée pour les lignes comprises entre la ligne 2 et 6.

```
awk 'BEGIN NR==2;NR==6 { print "ligne n°"NR":\  n"$0}
END ' adresse
```

3.3 Les actions

Présentation

Les actions permettent de transformer ou de manipuler les données, elles contiennent une ou plusieurs instructions. Les actions peuvent être de différents types: fonctions prédéfinies, fonctions de contrôle, fonctions d'affectation, fonctions d'affichage.

Fonctions prédéfinies traitant des numériques

- **atan2(y,x)** arctangente de x/y en radian (entre -pi et pi)
- **cos(x)** cosinus (radian)
- **exp(x)** exponentielle à la puissance x
- **int(x)** partie entière
- **log(x)** logarithme naturel
- **rand(x)** nombre aléatoire (entre 0 et 1)

- **sin(x)** sinus (radian)
- **sqr(t)** racine carrée
- **srand(x)** définition d'une valeur de départ pour générer un nombre aléatoire

Fonctions prédéfinies traitant de chaînes de caractères

Pour avoir la liste des fonctions prédéfinies sur votre plate-forme vous devez faire un **man awk**, voici la liste des fonctions les plus courantes sur un système UNIX.

- **gsub(expression-régulière,nouvelle-chaine,chaine-de-caractères)** dans chaine-de-caractères tous les caractères décrits par l'expression régulière sont remplacés par nouvelle-chaine. gsub et équivalent à gensub.
- **gsub(/a/,"ai",oi")** Remplace la chaine oi par ai
- **index(chaine-de-caractères,caractère-à-rechercher)** donne la première occurence du caractère-à-rechercher dans la chaine chaine-de-caractères
- **n=index("patate","ta")** n=3 length(chaine-de-caractères) renvoie la longueur de la chaine-de-caractères
- **n=length("patate")** n=6
- **match(chaine-de-caractères,expression-régulière)** renvoie l'indice de la position de la chaîne chaine-de-caractères, repositionne RSTART et RLENGTH
- **n=match("PO1235D",/[0-9][0-9]/)** n=3, RSTART=3 et RLENGTH=4
- **printf(format,valeur)** permet d'envoyer des affichages (sorties) formatées, la syntaxe est identique de la même fonction en C
- **printf("La variable i est égale à %7,2f",i)** sortie du chiffre i avec 7 caractères (éventuellement caractères vides devant) et 2 chiffres après la virgule.
- **printf("La ligne est %s",$0) > "fichier.int"** Redirection de la sortie vers un fichier avec >, on peut utiliser aussi la redirection >>. Ne pas oublier les "" autour du nom du fichier.
- **split(chaine-de-caractères,tableau,séparateur)** scinde la chaîne chaine-de-caractères dans un tableau, le séparateur de champ est le troisième argument
- **n=split("zorro est arrivé",tab," ")** tab[1]="zorro", tab[2]="est", tab[3]="arrivé", n=3 correspond au nombre d'éléments dans le tableau
- **sprintf(format,valeur)** printf permet d'afficher à l'écran alors que sprintf renvoie la sortie vers une chaîne de caractères.
- **machaine=sprintf("J'ai %d patates",i)** machaine="J'ai 3 patates" (si i=3)
- **substr(chaine-de-caractères,pos,long)** Extrait une chaine de longueur long dans la chaîne chaine-de-caractères à partir de la position pos et l'affecte à une chaîne.

- **machaine=substr("Zorro est arrivé",5,3)** machaine="o e"
- **sub(expression-régulière,nouvelle-chaine,chaine-de-caractères)** idem que gsub sauf que seul la première occurence est remplacée (gsub=globale sub)
- **system(chaine-de-caractères)** permet de lancer des commandes d'autres programmes
- **commande=sprintf("ls | grep toto")** Exécution de la commande UNIX "ls |grep toto"
- **system(commande)**
- **tolower(chaine-de-caracteres)** retourne la chaîne de caractères convertie en minuscule
- **toupper(chaine-de-caracteres)** idem en majuscule

Fonctions définies par l'utilisateur

Vous pouvez définir une fonction utilisateur de telle sorte qu'elle puisse être considérée comme une fonction prédéfinie. La syntaxe est la suivant:

```
fonction mafonction(liste des paramètres)
{
  instructions
  return valeur
}
```

3.4 Les variables et opérations sur les variables

Présentation

On trouve les variables système et les variables utilisateurs. Les variables systèmes non modifiables donnent des informations sur le déroulement du programme. Les variables utilisateurs sont définies par l'utilisateur.

Les variables utilisateur

Le nom des variables est formé de lettres, de chiffres (sauf le premier caractère de la variable), d'underscore. Ce n'est pas nécessaire d'initialiser une variable, par défaut, si c'est un numérique, elle est égale à 0, si c'est une chaîne, elle est égale à une chaîne vide. Une variable peut contenir du texte, puis un chiffre, en fonction de son utilisation **awk** va déterminer son type (numérique ou chaîne).

Les variables prédéfinies (système)

Les variables prédéfinies sont les suivantes (en italique les valeurs par défaut):

- **ARGC** nombre d'arguments de la ligne de commande *néant*
- **ARGIND** index du tableau **ARGV** du fichier courant
- **ARGV** tableau des arguments de la ligne de commande *néant*
- **CONVFMT** format de conversion pour les nombres *"%.6g"*
- **ENVIRON** tableau contenant les valeurs de l'environnement courant

- **ERRNO** contient une chaîne décrivant une erreur *""*
- **FIELIWIDTHS** variable expérimentale à ne pas utiliser
- **FILENAME** nom du fichier d'entrée *néant*
- **FNR** numéro d'enregistrement dans le fichier courant *néant*
- **FS** contrôle le séparateur des champs d'entrée *" "*
- **IGNORECASE** contrôle les expressions régulières et les opérations sur les chaînes de caractères *0*
- **NF** nombre de champs dans l'enregistrement courant *néant*
- **NR** nombre d'enregistrements lus jusqu'alors néant
- **OFMT** format de sortie des nombres (nombre après la virgule) *"%.6g"*
- **OFS** séparateur des champs de sortie *" "*
- **ORS** séparateur des enregistrements de sortie
- **RLENGTH** néant longueur de la chaîne sélectionnée par le critère *"\n"*
- **RS** contrôle le séparateur des enregistrements d'entrée *"\n"*
- **RSTART** début de la chaîne sélectionnée par le critère *néant*
- **SUBSEP** séparateur d'indiçage *"\034"*

Opérations sur les variables

On peut manipuler les variables et leur faire subir certaines opérations. On trouve différents types d'opérateurs, les opérateurs arithmétiques classiques (**+**, **-**, ***, /, **%**(modulo, reste de la division), (puissance)), les opérateurs d'affectation (**=**, **+=**, **-=**, ***=**, **/=**, **%=**, **=**). Exemples:

- **var=30** affectation du chiffre 30 à var
- **var="toto"** affectation de la chaîne toto à var
- **var="toto " "est grand"** concaténation des chaînes
- "toto " et "est grand", résultat dans var
- **var=var-valeur var-=valeur** expressions équivalentes
- **var=var+valeur var+=valeur** expressions équivalentes
- **var=var*valeur var*=valeur** expressions équivalentes
- **var=var/valeur var/=valeur** expressions équivalentes
- **var=var%valeur var%=valeur** expressions équivalentes
- **var=var+1 var++** expressions équivalentes
- **var=var-1 var--** expressions équivalentes

Les variables de champ

Comme on l'a déjà vu auparavant les champs d'un enregistrement (ligne) sont désignés par **$1**, **$2**,...**$NF**(dernier champ d'une ligne). L'enregistrement complet (ou ligne) est désigné par **$0**. Une variable champ est a les mêmes propriétés que les autres variables. Le signe **$** peut être suivi par une variable, exemple:

```
i=3
print $(i+1)
```

Provoque l'affichage du champ 4

Lorsque **$0** est modifié, automatiquement les variables de champs **$1**,..**$NF** sont redéfinies.

Quand l'une des variables de champ est modifée, **$0** est modifié. ATTENTION le séparateur ne sera pas celui définit par **FS** mais celui définit par **OFS** (output field separator). Exemple:

```
awk 'BEGIN{print" # Tous les utilisateurs du groupe users(GID 22)basculeront
dans le groupe boulot(GID 24)
          ";FS=":";OFS=":"}
    $4 != 22 {print $0} # Si le groupe n'est pas users on fait rien
    $4 ==22 {$4=24;print $0} # Si le groupe est 22, on lui réaffecte 24
END {print"C'est fini"}' /etc/passwd > passwd.essai
```

3.5 Les structures de contrôle

Présentation

Parmi les structures de contrôle, on distingue les décisions (**if**, **else**), les boucles (**while**, **for**, **do-while**, **loop**) et les sauts (**next**, **exit**, **continue**, **break**).

Les décisions (if, else)

La syntaxe est la suivante:

```
if  (condition) si la condition est satisfaite (vraie)
  instruction1 on exécute l'instruction
else sinon
  instruction2 on exécute l'instruction 2
```

Si vous avez une suite d'instructions à exécuter, vous devez les regrouper entre deux accolades. Exemple:

```
awk ' BEGIN{print"test de l'absence de mot de passe";FS=":"}
NF==7
{ #pour toutes les lignes contenant 7 champss
  if ($2=="") # si le deuxième champ est vide (correspond au mot de passe
crypté))
  { print $1 " n'a pas de mot de passe"} # on affiche le nom de l'utilisateur
($1=login) qui n'a pas de mot de passe
  else sinon
  { print $1 " a un mot de passe"} # on affiche le nom de l'utilisateur
possédant un mot de passe
}
END{print"C'est fini") ' /etc/passwd
```

Les boucles (while, for, do-while)

while, **for** et **do-while** sont issus du langage de programmation C. La syntaxe de **while** est la suivante:

```
while  (condition) tant que la condition est satisfaite (vraie)
instruction on exécute l'instruction
```

Exemple:

```
awk ' BEGIN{print"affichage de tous les champs de passwd";FS=":"}
```

```
{ i=11 # initialisation du compteur à 1 (on commence par le champ 1)
while(i=<NF) # tant qu'on n'est pas en fin de ligne
  { print $ii # on affiche le champ
  i+++ # incrémentation du compteur pour passer d'un champ au suivant
  }
}
END{print"C'est fini") ' /etc/passwd
```

La syntaxe de **for** est la suivante:

for (instruction de départ; condition; instruction d'incrémentation)

On part d'une instruction de départ, on incrémente instruction on exécute l'instruction jusqu'à que la condition soit satisfaite

Exemple:

```
awk ' BEGIN{print" affichage de tous les champs de passwd";FS=":"}
{
for (i=1;i=><NF;i++) # initialisation du compteur à 1, on incrémente le compteur
jusqu'à ce qu'on atteigne NF (fin de la ligne)
  { print $i } # on affiche le champ tant que la condition n'est pas satisfaite
}
END{print"C'est fini") ' /etc/passwd
```

Avec **for** on peut travailler avec des tableaux. Soit le tableau suivant: **tab[1]="patate"**, **tab[2]="courgette"**,**tab[3]="poivron"**. Pour afficher le contenu de chacun des éléments du tableau on écrira:

```
for (index in tab)
{
 print "legume :" tab[index]
}
```

La syntaxe de **do-while** est la suivante:

```
do
{instructions}} on exécute les instructions
while  (condition) jusqu'à que la condition soit satisfaite
```

La différence avec while, est qu'on est sûr que l'instruction est exécutée au moins une fois.

Sauts controlés (break, continue, next, exit)

break permet de sortir d'une boucle, la syntaxe est la suivante:

```
for  (;;;) boucle infinie
{instructions on exécute les instructionss
if  (condition)  break si la condition est satisfaite on sort de la boucle
instructions}
```

Alors que **break** permet de sortir d'une boucle, **continue** force un nouveau passage dans une boucle.

next permet d'interrompre le traitrement sur la ligne courante et de passer à la ligne suivante (enregistrement suivant).

exit permet d'abandonner la commande **awk**, les instructions suivant **END** sont exécutées (s'il y en a).

3.6 Les tableaux

Présentation

Un tableau est une variable se composant d'un certains nombres d'autres variables (chaînes de caractères, numériques,...), rangées en mémoire les unes à la suite des autres. Le tableau est dit unidimensionnelle quand la variable élément de tableau n'est pas elle même un tableau. Dans le cas de tableaux imbriqués on parle de tableau unidimensionnels.

Les termes matrice, vecteur ou table sont équivalents à tableau.

Les tableaux unidimensionnels

Vous pouvez définir un tableau unidimensionnel avec la syntaxe suivante:**tab[index]=variable**, l'index est un numérique (mais pas obligatoirement, voir les tableaux associatifs), la variable peut être soit un numérique, soit une chaîne de caractère. Il n'est pas nécessaire de déclarer un tableau, la valeur initiale des éléments est une chaîne vide ou zéro. Exemple de définition d'un tableau avec une boucle **for**.

```
var=1
for (i=1;i<=NF;i++)
  { mon-tab[i]=var++}
```

On dispose de la fonction delete pour supprimer un tableau (**delete tab**). Pour supprimer un élément de tableau on tapera **delete tab[index]**.

Les tableaux associatifs

Un tableau associatif est un tableau unidimensionnel, à ceci près que les index sont des chaînes de caractères. Exemple:

```
age["olivier"]=27
age["veronique"]=25
age["benjamin"]=5
age["veronique"]=3
for (nom in age)
 { print nom " a " age[nom] "ans"
 }
```

On a un tableau **age** avec une chaîne de caractères prénom comme index, on lui affecte comme éléments de tableau un numérique (age de la personne mentionnée dans le prénom). Dans la boucle **for** la variable **nom** est remplie successivement des chaînes de caractères de l'index (**olivier**, **veronique**, ...).

Les valeurs de l'index ne sont pas toujours triées.

Les tableaux multidimensionnels

awk n'est pas prévu pour gérer les tableaux multidimensionnels (tableaux imbriqués, ou à plusieurs index), néanmoins on peut simuler un tableau à deux dimensions de la manière suivante. On utilise pour cela la variable prédéfinie **SUBSEP** qui, rappelons le, contient le séparateur d'indiçage. Le principe repose sur la création de deux indices (**i**, **j**) qu'on va concaténer avec **SUBSEP (i:j)**.

```
SUBSEP=":"
i="A",j="B"
tab[i,j]="Coucou"
```

L'élément de tableau "**Coucou**" est donc indexé par la chaîne "**A:B**".