AWK & Sed

Parfois les filtres de base trouvent leurs limites et nécessitent de faire quelques acrobaties pour arriver à ses fins.

Parfois il est tout simplement impossible de réaliser certains traitements avec des filtres de base. Il faudrait écrire un bout de code pour effectuer le traitement souhaité.

Du code ? Du C ? Compiler ? Installer ? C'est lourd !

Partant de ce besoin d'aller parfois plus loin avec les filtres, sont nés 2 outils, des filtres programmables :

- AWK (Aho, + Weinberger + Kernighan, les créateurs)
- Sed (Stream Editor)

Comme tous les filtres, AWK & Sed traitent un flux de données (textes) entrant et produisent un flux sortant, résultat du traitement fait par un bout de code écrit dans des langages spécifiques à chacune des 2 commandes.

1 - AWK

Le principe de fonctionnement de AWK est similaire aux autres filtres : il exécute une boucle implicite qui lit les données au “fil de l’eau”, une ligne à la fois.

Le “bout de code” qu’on doit écrire ne s’occupe pas de gérer la boucle, il s’occupe juste du traitement de la ligne en cours. Ce code s’appelle un script.

Le langage AWK est assez simple, mais suffisant pour rendre de précieux services sans avoir un coder en C.

Le langage interne de AWK permet de faire, avec la ligne en cours :

- des affichages.
- des tests.
- des conversions.
- des recherches.
- des extractions de sous-chaînes.
- des calculs.

Il dispose aussi de variables, non volatiles (entre 2 lignes).

Syntaxes :

- awk '<script>' fichier
- awk '<script>' < fichier
- commande(s)... | awk '<script>' | commande(s)...

Le script peut être long, il est possible de le placer dans un fichier. La syntaxe devient alors, par exemple :

awk -f nomscript.awk < fichier

Script AWK

Un script est décomposé en 3 sections de code.
Chacune est optionnelle mais il en faut au moins une sur les 3, sinon… ben y'a pas de script !

Si le script est placé dans un fichier, le nom du fichier et son extension (facultative) sont quelconques, pas besoin de mettre un .awk, mais ça peut quand même aider à "deviner" qu'il s'agit d'un script AWK.

Les 3 sections :

- BEGIN : exécutée avant la lecture du fichier à traiter.
- END : exécutée après le traitement intégral du fichier.
- Le reste : exécutée pour chaque ligne du fichier

Un script peut contenir 0, 1 ou plusieurs portions de code pour chaque section (BEGIN, END, Le reste).

Généralement on trouve 0 ou 1 section BEGIN, 0 ou 1 section END et 1 ou plusieurs sections “Le reste”.

BEGIN

La section BEGIN est exécutée avant la lecture du fichier, avant même l'ouverture du fichier.

Même si le fichier n'existe pas, BEGIN est exécutée !

Cette section sert souvent à mettre en place l'environnement de traitement. Par exemple : initialiser des variables qui vont servir au traitement des lignes.

IMPORTANT :

Quand BEGIN est exécutée, ça se passe avant le début de lecture du fichier, donc AUCUNE ligne n'a encore été lue.

Il est impossible de faire référence à la ligne "courante", c'est un non sens.

Syntaxe :

BEGIN { ... }

Exemples :

- BEGIN { compteur = 0 }
- BEGIN { min = 99999; max = 0 }
- BEGIN { print "Debut du traitement" }

Le code est toujours placé entre { }.

END

La section END est exécutée après le traitement du fichier, et uniquement s'il n'y a pas eu d'erreur (fichier inexistant, erreur de syntaxe, erreur de calcul, etc.).

Cette section sert souvent à faire une synthèse du traitement réalisé. Par exemple : un comptage, un calcul final, le résultat d'une recherche, etc.

IMPORTANT :

Quand END est exécutée, ça se passe après le traitement du fichier, donc TOUTES les lignes ont déjà été lues.

Il est impossible de faire référence à la ligne “courante”, c’est un non sens.

Syntaxe :

END { ... }

Exemples :

- END { printf "Trouvé %d occurrences", nb }
- END { moy = somme / nb }
- END { print "Fin du traitement. Moy = " moy }

Le code est toujours placé entre { }.

“Le reste”

Les sections autres que BEGIN et END (nommées “Le reste”), sont TOUTES exécutées, séquentiellement, pour et sur CHAQUE ligne du fichier à traiter.

Le code de ces sections peut accéder au contenu de la ligne “courante”, et uniquement cette ligne là. Le contenu de la ligne courante change évidemment à chaque tour de boucle.

Ces sections “Le reste” sont les seules à accéder aux données.

Donc ce sont elles qui font tout le travail de calcul, de recherche, d’extraction, de transformation etc.

Elle peuvent être conditionnelles ou systématiques.

Conditionnelle

Une section conditionnelle est préfixée par un test.

Si le résultat est évalué “Vrai”, la section est exécutée pour la ligne en cours. Sinon l’exécution passe à la section suivante.

C’est une sorte de IF qui peut ne pas être écrit.

La condition peut être exprimée de 2 façons :

- Un test simple, précédé de if ( ) comme en C.
- Un motif egrep, placé entre / /, ou !/ / pour négation

Le code est toujours placé entre { }.

Exemples :

- if (NR == 1) { print "Première ligne du fichier" }
- /,FRANCE,/ { print "FRANCE trouvée" }
- !/[0-9]{5}/ { print "Pas trouvé de code postal" }

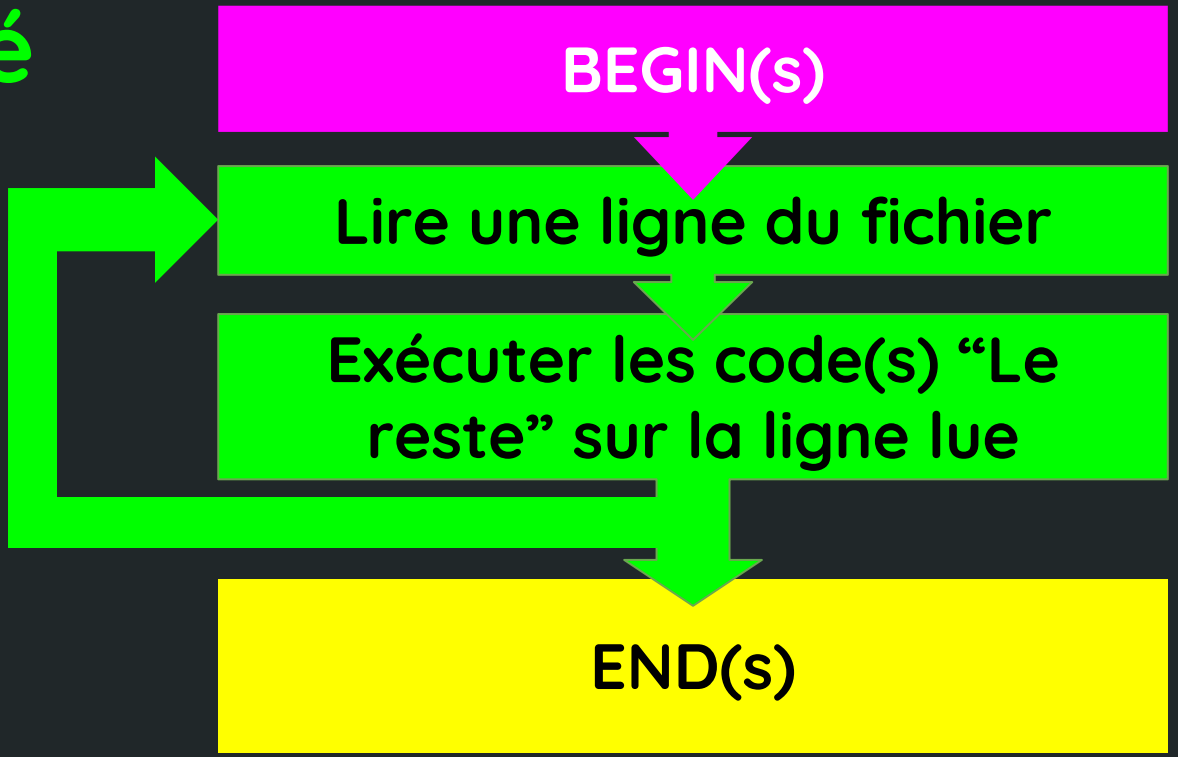
Systématique

Une section qui n'est pas conditionnelle, qui n'est pas préfixée par un test, est donc systématique. Elle est exécutée pour toutes les lignes. Le code est toujours placé entre { }.

Exemples :

- { printf "J'ai lu la ligne n°%d\n", NR }
- { print "Champ n°3 : " $3 }

Déroulé



Il n'est pas obligatoire que les sections soient ordonnées BEGIN… Le reste… END.

Le script est analysé intégralement avant son exécution. Il est donc possible d'avoir un END avant un BEGIN sans impact sur le séquencement. Plusieurs occurrences d'une section sont exécutées dans l'ordre de leur définition.

C'est comme dans un programme en C où les fonctions n'ont pas besoin d'être triées dans le code.

2 - AWK, Les instructions

Les instructions, le code situé entre les { }, ressemblent beaucoup à du C, simplifié.

Deux instructions sont séparées par un ;
On peut avoir autant d'instructions consécutives qu'on souhaite. Elles peuvent être sur une seule ligne ou sur plusieurs tant que le tout tient entre { }.

Exemple :
{ print "Début du traitement"; nb = 0; somme = 0; }

Arithmétique

Les opérations arithmétiques sont les mêmes qu'en C :

- 4 opérations de base (+, -, /, *) : val *10
- La division est réelle : 5/2 = 2.5
- Usage des ( ) pour changer les priorités.
- Modulo % (reste de la division euclidienne) : 5%2 = 1

Exemple : { print (6 * 3 / (5 - 3)) % 4 }
Affiche : 1.

Chaînes de caractères

Les chaînes de caractères sont encadrées par des " " uniquement. Ainsi, un script “en ligne de commande” pourra être encadré par des ' ', sans conflit entre les 2 :

```
awk '{ print "Traitement de la ligne n°" NR }'
```

Il n'y a aucun opérateur de concaténation de chaînes de caractères, il suffit de les juxtaposer :

```
{ print "Un" "bout" "de" "texte !" }
```

affichera :

```
Unboutdetexte !
```

Variables

Une variable peut contenir :

- Une chaîne de caractères.
- Une valeur numérique (indifféremment entière ou réelle).
- Un tableau.

Un nom de variable doit commencer par une lettre minuscule ou MAJUSCULE, puis un nombre quelconque de chiffres, de lettres (min/MAJ) ou d'underscores (_).

Les min/MAJ sont différenciées : val != VAL

Les mots clés du langage sont des noms interdits pour une variable : if, print, etc.

Variables - Affectation

Il n'y a pas réellement de définition d'une variable mais une affectation.

Le fait d'affecter une variable inconnue, fait en même temps sa définition et son initialisation. Exemples :

- { nb=0 }
- { nom="Dubois" }

Variables - Utilisation

Pour utiliser une variable, il suffit d'écrire son nom.

Si elle a été affectée, donc initialisée, sa valeur est utilisée dans l'expression.

Si elle n'a jamais été affectée, il n'y a aucune erreur mais sa valeur dépend du contexte.

- { print "Val = " val }
  Affiche “Val =” seulement. val (qui est inconnue) est utilisée comme une chaîne de caractères.
- { print "Val = " (val*10) }
  Affiche “Val = 0”. val (qui est inconnue) est utilisée comme un entier, à cause de la multiplication.
- { val=10; print "Val = " (val * 10) }
  Affiche “100”. val est connue, pas d’ambigüité.

En cas d’ambigüité, c’est donc le contexte qui détermine la nature d’une variable.

Variables - Incréments/Décréments

Les mêmes qu'en C :

- val++ : post-incrément de 1.
- val-- : post-décrément de 1.
- ++val : pré-incrément de 1.
- --val : pré-décrément de 1.

Variables - Opération+Réaffectation

Les mêmes qu'en C :

- val+=5 : incrément de 5 et réaffectation du résultat.
- val-=2 : décrément de 2 et réaffectation du résultat.
- val*=10 : multiplication par 10 et réaffectation du résultat.
- val/=2 : division (réelle) par 2 et réaffectation du résultat.
- val%=3 : modulo 3 et réaffectation du résultat.

Variables spéciales

Il existe des variables spéciales, internes à AWK, qui sont affectées automatiquement.

Certaines changent de valeur au fur et à mesure de la lecture des données.

Variable FS

FS : Field Separator.

Chaîne de caractères (Regex egrep, mais souvent 1 seul caractère), servant de séparateur de champ. Par défaut il s'agit de l'espace.

Elle peut être spécifiée dans BEGIN, ou au lancement :
awk -F':' '{ print "FS = " FS }'

Variable RS

RS : Record Separator.

Chaîne de caractères (Regex egrep, mais souvent 1 seul caractère), servant de séparateur d'enregistrement.
Par défaut il s'agit du \n.

Elle peut être spécifiée dans BEGIN, ou au lancement :
awk -v RS=';' '{ print "RS = " RS }'

Variable OFS

OFS : Output Field Separator.

Chaîne de caractères (par défaut espace), servant de séparateur de champ lors de l'écriture avec print et des valeurs séparées par des ,

Elle peut être spécifiée dans BEGIN, ou au lancement : awk -v OFS=':' '{ print "a", "b", "c" }', affiche "a:b:c"

Variables dynamiques

Les variables changeant dynamiquement à chaque ligne sont :

- NR : Numéro de la ligne en cours (à partir de 1)
- NF : Nombre de champs sur la ligne en cours.
- $1, $2… : La valeur des champs (jusqu'à $<NF>)
- $0 : le contenu de la ligne complète (sans le séparateur de ligne, qui est généralement le \n).

Conditions

L'écriture de conditions est similaire à la syntaxe du C :

```
{
if (<condition1>) { ... }
else if (<condition2>) { ... }
else { ... }
}
```

Les différents types de conditions sont :

- == : L'égalité.
- != : La différence.
- <, <=, > et >= : Les inégalités classiques.

On trouve aussi in, pour les tableaux (voir plus loin), et la négation : !

Boucle for

L'écriture de boucles "for" est similaire à la syntaxe du C :

```
{
for (<initialisation>; <condition>; <expression>) { ... }
}
```

Exemple : `for (i = 1; i < 10; i++) { print i }`

Boucle while

L'écriture de boucles "while" est similaire à la syntaxe du C :

```
{
while (<condition>) { ... }
}
```

Exemple : `while (i <= NF) { print $i; i++ }`

Boucle do...while

L'écriture de boucles "do...while" est similaire à la syntaxe du C :

```
{
do { ... } while (<condition>)
}
```

Exemple : `do { print $i; i++ } while (i < NF)`

3 - AWK, Les fonctions

Les fonctions de AWK sont assez nombreuses.

Nous allons juste en évoquer quelques unes.

Pour une liste exhaustive :

https://www.tutorialspoint.com/awk/awk_built_in_functions.htm

match

Syntaxe : match(chaine, motif)

Renvoie la position (1 à N) du motif (Regex egrep) dans chaine ou 0 si absent.

Exemple : { if match("[a-z]{3}$", $2) { ... } }
Teste si le 2ème champ se termine par 3 minuscules.

sub

Syntaxe : sub(quoi, par_quoi, dans_quoi)

Remplace (substitute) la chaîne “quoi” par la chaîne “par_quoi”, dans la chaîne “dans_quoi”.

Exemple : { sub("2018-", "18-", $1) }
Remplace “2018-” par “18-” dans le 1er champ de chaque ligne.

substr

Syntaxe : substr(chaine, debut, longueur)

Extrait une sous-chaîne (substring) de “chaine” en commençant à la position debut (1ère position = 1), sur longueur caractères.

Exemple : { print substr($0, 5, 10) }
Affiche du 5è au 14è caractère de la ligne en cours.

length

Syntaxe : length(chaine)

Retourne la longueur de “chaine”.

print & printf

Ces 2 fonctions sont similaires avec quelques petites nuances et subtilités.

Syntaxe :

- print "chaine1" "chaine2" ...
- print "chaine1", "chaine2" ...
- printf "format", param1, param2 ...

printf est très semblable au printf du C :

Exemples :

- { printf "Ligne complète : %s\n", $0 }
- { printf "Champ1 : %s, champ 2 : %s\n", $1, $2 }

Les paramètres suivent le printf avec des ,
Il faut mettre un \n pour avoir un retour à la ligne à l'affichage.

print permet d'afficher des chaînes :

- bout à bout (si pas de , entre les paramètres)
- avec un séparateur (voir variable OFS) si , entre les paramètres.

Exemples :

- { print "Champ1 : " $1 " champ 2 : " $2 }
- { print "Champs : ", $1, $2 }

Pas besoin de \n en fin de ligne, print le met lui-même.

4 - AWK, Les tableaux

Les tableaux sont un peu différents de ceux du C. Ils ne sont pas indexés mais associatifs.

Les indices, qu'on appelle aussi des clés, peuvent être des valeurs numériques mais aussi des chaînes de caractères.

Syntaxe : tab[<indice>] = <valeur>

Exemples :

- mois["janvier"] = 31;
- fruits["banane"] = "jaune";

Test si un indice existe dans un tableau :

- if (<indice> in tableau) { ... }
- if (tableau[<indice>] != "") { ... }

Exemples :

- if ("toto" in joueurs) { ... }
- if (12 in notes_ds) { ... }
- if (tarifs[$1] != "") { ... }

Parcours

Parcourir un tableau se fait en parcourant les indices :

```
for (indice in tableau) {
    printf "Tableau[%s] = %s\n", indice, tableau[indice]
}
```

5 - Sed

Sed (Stream Editor) est un éditeur de texte fonctionnant en mode “flux” (stream).

Le mode flux signifie que les manipulations que l’éditeur peut faire sur ou avec le texte (modifications, suppressions, ajouts) ne peuvent se faire qu’au moment où le texte passe dans le flux.

L’éditeur est incapable de se déplacer en amont ou en aval de la ligne courante, il doit suivre le flux de texte.

Contrairement à un éditeur classique, sed est autonome, il n'y aucune interactivité avec l'utilisateur.

Comme avec les autres filtres, on prépare la ou les actions qu'on demande à sed de faire, en les indiquant en paramètres, puis on lance la commande qui s'exécute ensuite sans aucune interaction/question possible.

Les principales actions que sed sait faire :

- Remplacer du texte.
- Modifier du texte.
- Supprimer du texte.
- Afficher tout ou partie du texte.

Remplacer du texte est sans doute l'usage le plus courant de sed.

Syntaxe #1

La syntaxe la plus simple est : sed -e '<cmde>' fichier

Elle permet d'exécuter une action sur chaque ligne du fichier. Exemple :

sed -e 's/ /,/g' fichier

Remplace tous les espaces par des virgules.

Syntaxe #2

Il est possible d'exécuter plusieurs actions successivement sur chaque ligne du fichier :

```
sed -e '<cmde1>' -e '<cmde2>' -e '<cmde3>' fichier
```

ou :

```
sed -e '<cmde1>; <cmde2>; <cmde3>' fichier
```

Comme avec awk, on peut préfixer une commande avec une expression régulière (egrep) + option “-r”

Si l’expression matche la ligne, la commande est exécutée sur la ligne, sinon elle est conservée intacte :

sed -r -e '/^[0-9]/s/ /_/g' fichier

Remplace les espaces par des _ uniquement sur les lignes commençant par un chiffre.

Syntaxe #3

Par défaut, toutes les lignes lues et modifiées ou pas par sed, sont ensuite affichées après être passées par toutes les phases (les actions sed) de modification.

Il est possible de supprimer ce comportement par défaut avec l'option -n. Dans ce cas il faut spécifier les lignes à afficher en utilisant la commande sed "p".

Exemple :

sed -n '/[A-Z]{3}/p' fichier

- Suppression de l'affichage automatique de chaque ligne.
- On se focalise ensuite uniquement sur les lignes contenant 3 MAJUSCULES.
- On affiche les lignes qui correspondent au motif.

Remplacer

Syntaxe : /motif1/s/motif2/chaine/<portee_de_ligne>

- Cible les lignes qui matchent motif1.
- Remplace (substitute) les parties de la ligne qui matchent motif2, en les remplaçant par chaine.
- Si portee_de_ligne vaut g (global), toutes les occurrences sont remplacées sur la ligne, sinon, uniquement la 1ère.

Exemple :

- sed -re '/ +/s/^...//' fichier
  Si la ligne contient 1 ou plusieurs espaces successifs, remplacer les 3 premiers caractères par rien.

On peut aussi utiliser le groupage des Regex :

- sed -re '/=/s/([a-z]+)=([0-9]+)/\2=\1/' fichier
  Sur les lignes contenant un =, échange 2 parties : xxx=val devient val=xxx (si elles matchent leurs motifs respectifs)

Autre syntaxe :
<portee_globale>s/motif2/chaine/<portee_de_ligne>

- Cible les lignes qui correspondent à portee_globale.
- Remplace (substitute) les parties de la ligne qui matchent motif2, en les remplaçant par chaine.
- g pour global ou rien pour uniquement la 1ère occurrence.

La portée globale définit les lignes du fichier qui sont ciblées par la commande sed :

- %s = toutes les lignes.
- n,m = de la ligne n à la ligne m. On commence à 1.
- n,$ = de la ligne n à la fin du fichier.
- 1, m = de la 1ère ligne à la ligne m.

Exemple :

- sed -re '1,5s/,/;/g' fichier
  Sur les lignes de 1 à 5, remplace toutes les virgules par des points-virgules.

Supprimer

Syntaxe :

- /motif1/d
- portee_globaled

Exemples :

- sed -re '/^[0-9]/d' fichier
- sed -re '1,10d' fichier
- sed -re '5,$d' fichier

Ajouter

- /motif1/atexte_a_ajouter
  Exemple : sed -re '/^[0-9]/ablabla' fichier
  Ajoute blabla après (càd en dessous de) chaque ligne commençant par un chiffre.

- position_globaleatexte_a_ajouter
  Exemple : sed -re '$ablabla2' fichier
  Ajoute blabla2 après la dernière ligne.

- portee_globaleatexte_a_ajouter
  Exemple : sed -re '1,$ablabla3' fichier
  Ajoute blabla3 après chaque ligne.

Fichier script

Comme avec AWK, il est possible de placer toutes les commandes sed dans un fichier, on n'utilise plus la/les option(s) -e dans ce cas.

Exemple : sed -f script.sed fichier
script.sed :

- /[0-9]{5}/s/:/;/g
- 1,5d

Note spéciale sur l'option -i

Les filtres ne modifient (presque) jamais les fichiers directement, ils produisent un résultat sur STDOUT.,Le résultat peut servir à alimenter un tube (pipe) ou être envoyé dans un fichier.

Exception avec sed -i qui permet de modifier un fichier directement. Donc ne jamais mettre sed -i avec un tube ! INTERDIT de faire : sed -i '....' fichier | commande

6 - Jouons un peu

Objectif

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

Transformer des numéros de téléphone pour faire apparaître l'indicatif du pays et formater les numéros par blocs de chiffres séparés par des espaces.

Exemples :

- +33 02 96 31 32 44 (blocs de 2 chiffres)
- +49 5 410 229 615 (1 chiffre puis blocs de 3 chiffres)

AWK

Etape 1 - Séparateur de champ ?

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 1 - Séparateur de champ ?

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 2 - Identifier les cas possibles

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 2 - Identifier les cas possibles

- FR -> +33
- DE -> +49

“FR;” et “DE;” doivent être remplacés
par les indicatifs de pays correspondants.
2 cas = 2 règles (2 “Les autres”) dans le script.

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 3 - Comment extraire les blocs ?

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 3 - Comment extraire les blocs ?

substr(...)

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 3 - Comment extraire les blocs ?

- FR : 5 blocs de chiffres, 5 substr(...).
- DE : 1+3 blocs de chiffres, 4 substr(...).

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 4 - Comment formater l'affichage ?

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Etape 4 - Comment formater l'affichage ?

print ou printf

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

AWK

Le script “tel.awk” en version “print” :

```
/^FR;/ { print "+33 " substr($2, 1, 2) " " substr($2, 3, 2) " " substr($2, 5, 2) " " substr($2, 7, 2) " " substr($2, 9, 2) }

/^DE;/ { print "+49 " substr($2, 1, 1) " " substr($2, 2, 3) " " substr($2, 5, 3) " " substr($2, 8, 3) }
```

AWK

Le script “tel.awk” en version “printf” :

```
/^FR;/ { printf "+33 %s %s %s %s %s\n", substr($2, 1, 2),
substr($2, 3, 2), substr($2, 5, 2), substr($2, 7, 2),
substr($2, 9, 2) }

/^FR;/ { printf "+49 %s %s %s %s\n", substr($2, 1, 1),
substr($2, 2, 3), substr($2, 5, 3), substr($2, 8, 3) }
```

AWK

Appel du script : awk -F';' -f tel.awk < tel

ou alors : awk -f tel.awk < tel
et on ajoute ceci dans le script :

BEGIN { FS = ";" }

Sed

Etape 1 - Identifier les cas possibles

Les mêmes 2 cas, mais pas de notion de séparateur, donc :

- FR;
- DE;

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

Sed

Etape 2 - Comment extraire les blocs ?

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

Sed

Etape 2 - Comment extraire les blocs ?

groupage regex : (...)

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

Sed

Etape 3 - Comment formater l'affichage ?

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

Sed

Etape 3 - Comment formater l'affichage ?

groupes

regex : (...)...\1...

Téléphones

FR;0296313233
DE;5410229615
DE;5551039531
FR;0296541901

+33 02 96 31 32 44
+49 5 410 229 615

Sed

Le script “tel.sed” :

```
/^FR;/s/^FR;([0-9]{2})([0-9]{2})([0-9]{2})([0-9]{2})([0-9]{2})$/+33 \1 \2 \3 \4 \5/

/^DE;/s/^DE;([0-9])([0-9]{3})([0-9]{3})([0-9]{3})$/+49 \1 \2 \3 \4/
```

Sed

Appel du script : sed -r -f tel.sed < tel

https://commons.wikimedia.org/wiki/File:Thats_all_folks.svg