La commande awk

PHILOSOPHIE

On parle souvent de **awk** comme d'une version améliorée de **sed**. En effet, on retrouve la notion de portée de commande et de manipulation de texte, mais à mon avis, cette comparaison s'arrête la. Si **sed** dispose d'une commande **if**, l'on ne peut parler de langage de programmation alors que cette notion s'applique totalement à **awk**. Il n'est pas question ici de traiter totalement **awk** mais juste de vous donner un bref aperçu de sa puissance.

Dû à la complexité de **awk**, il ne nous sera pas possible de traiter ce langage de façon extensive. Aussi, nous allons procéder à un rapide tutorial dans la première partie et donner une référence rapide des fonctionnalités les plus utilisées dans une seconde.

UTILISATION SIMPLISTE DE AWK

LA NOTION DE COLONNE ET LE SÉPARATEUR DE COLONNES

Lorsqu'**awk** lit une ligne dans le fichier qui lui est passé en entrée, il tente de la découper en colonnes. Le terme consacré dans la littérature consacrée à **awk** est celui de *champ* ou *field* en Anglais. Pour continuer dans le vocabulaire de **awk**, une ligne sera régulièrement appelée *enregistrement*, soit *record* en Anglais.La frontière entre 2 colonnes est spécifiée par un caractère spécial appelé : le séparateur de colonnes. Ce dernier peut être *n'importe quel caractère*, en fait, l'on peut même spécifier, une classe de valeurs, *i.e.* un ensemble de caractères. Sa valeur par défaut est la classe {espace, tabulation}.

Il est possible de spécifier le séparateur de colonnes sur la ligne de commande avec l'option **-F*caractere_ou_classe***, où à l'intérieur du script en appliquant la commande **FS=*caractere_ou_classe***. Mnémonique : **FS** signifie *Field Separator*.

Considérons la commande simpliste suivante :

```
awk '{print $1}' fichier
```

Tout d'abord, quelques remarques d'ordre général :

- Les commandes de **awk** doivent être placées entre accolades
- Il est important d'utiliser des quotes simples avec **awk** car le caractère **$** est utilisé comme spécificateur de colonne, les colonnes sont numérotées à partir de 1, l'expression **$0** désignant la ligne entière.
- Dans la plupart des cas, vous n'utiliserez pas de commande en ligne comme ci-dessus, mais plutôt des scripts.

La commande que nous avons entrée affiche à l'écran la première colonne de chaque ligne du **fichier** lu.

Appliquons la au fichier suivant en changeant le caractère de séparation de colonnes.

```
Ceci est une ligne separee par des espaces
Ceci   est   une   ligne   separee   par    des    tabulations
Ceci;est;une;ligne;separee;par;des;point-virgules
Ceci,est,une,ligne,separee,par,des,virgules
```

Commande	Résultat	Commentaire

awk '{print $1}' fichier	Ceci Ceci Ceci;est;une;ligne ... Ceci,est,une,ligne ...	Cette première colonne nous permet de voir que le séparateur de colonne par défaut peut être indifféremment un espace ou un caractère de tabulation. En outre, vous remarquerez que les colonnes sont numérotées à partir de 1
awk -F\; '{print $1}' fichier	Ceci est une ligne ... Ceci est une ... Ceci Ceci,est,une	Le séparateur a été changé pour le point virgule. Comme en atteste le traitement des 2 première lignes, spécifier un caractère de séparation remplace le précédent. Si vous ne banalisez pas le point virgule, il risque d'être évalué par le shell comme une fin de commande.
awk -F,'{print $1}' fichier	Ceci est une ligne ... Ceci est une ... Ceci;est;une .... Ceci	Utilisation avec une virgule comme séparateur
awk -F[,;] '{print $1}' fichier	Ceci est une ligne ... Ceci est une ... Ceci Ceci	Ici, l'on spécifie une classe entre deux crochets. Ainsi, plusieurs caractères peuvent servir de séparateur de colonne.

SYNTAXE GÉNÉRALE D'UN PROGRAMME AWK

Un programme awk est constitué de trois grandes sections :

La section initiale

Les commandes qu'elle contient sont lues et traitées avant la lecture du fichier en entrée. C'est le bon endroit pour initialiser des variables par exemple ou pour ouvrir des fichiers auxiliaires.
Elle commence par le mot clef **BEGIN** et est entourée d'une paire d'accolades.

La section terminale

Comme son nom l'indique, la section terminale n'est traitée qu'une fois le fichier en entrée fermé. Typiquement, c'est l'endroit idéal pour fermer des fichiers auxiliaires ou réaliser des statistiques. Bref, tout ce qui ne peut se faire qu'une fois toutes les données connues et traitées.
Elle est introduite par le mot clef **END** et est entourée d'une paire d'accolades

La section des règles

C'est la section ou vous mettez les lignes de code de traitement du fichier. Elle est appelée ainsi, car, de la même manière que vous spécifiez une portée pour les commandes de **sed**, vous allez spécifier des règles d'activation pour les ordres de **awk**.

Chaque ensemble de commandes est ceint d'une paire d'accolades et précédé d'une portée. Celle ci peut être sous la forme d'une expression régulière entourée par des slash **/** à la manière de **sed** mais également toute expression arithmétique permettant de discriminer les lignes, par exemple, des conditions sur le nombre de champs. Elle peut également être vide, auquel cas, les commandes seront appliquées à toutes les lignes.
Dans cette section, vous pourrez également définir des fonctions utilisateur réutilisables dans le code.

UN PREMIER EXEMPLE

Enoncé

Supposons que nous voulions connaître la moyenne de l'occupation en disque des utilisateurs. La commande à utiliser est assurément **du -s rep_util** sur chaque répertoire d'utilisateur avec l'aide de **find**. Nous avons donc la commande :

```
find /home -prune -exec du -s {} \;
```

Le résultat est de la forme :

```
util1:1234
util2:3451
util3:342
util4:2342134
```

Comment pouvez vous traiter un tel fichier pour obtenir la moyenne ?

Un premier script

Les paragraphes suivant étudient en détail les trois grandes sections qui composent le fichier **awk** associé au traitement du problème.

Section initiale

Dans la section initiale, nous créons deux variables, l'une pour le nombre de lignes et l'autre pour la somme des nombres, nous en profitons également pour fixer le séparateur de colonnes au caractère deux points (:).

```
BEGIN{
  FS=":";
  NbLignes=0;
  Total=0;
}
```

Bien que très simple, cette section n'en est pas moins déjà instructive, en effet, vous y apprenez que :

- Il est impératif que l'accolade de début de section soit placée juste après le mot clef **BEGIN**
- Les chaînes de caractères sont à spécifier entre *double quotes*
- Les déclarations de variables sont automatiques, il n'y a pas de sous section dédiée à la déclaration
- Toutes les lignes de code se terminent par un point virgule

Section finale

Examinons maintenant la section finale :

```
END{
  printf("Nombre d'utilisateurs %d Total %d Moyenne %8.2f\n",
         NombreLignes,
         Total,
         Total/NombreLignes);
}
```

Cette section finale est elle aussi très simple : elle ne fait qu'imprimer le résultat. Vous noterez que nous utilisons ici la fonction **printf** qui fonctionne exactement comme son homologue du langage C. Cette ligne est à comparer avec celle du premier exemple qui, elle, utilisait la macro **print**, de syntaxe plus simple, mais limitée à des affichages sommaires.

Vous notez également ici que le type des variables varie dynamiquement selon la fonction qui l'utilise. En effet, selon le contexte, une variable agira comme un entier, un réel ou bien une chaîne de caractères. En fait, les variables sont toujours stockées sous la double forme d'une chaîne de caractère et d'un nombre. Il peut être nécessaire de forcer l'évaluation numérique dans certains cas. Par exemple l'expression **var1==var2** est par défaut évaluée sur des chaînes de caractères. Si vous voulez qu'elle ait lieu dans un espace numérique, il faudra utiliser **(var1 + 0)==(var2 + 0)** car chacune des additions force l'évaluation numérique.

La section des règles

```
{
NombreLignes++;
```

```
Total+=$2;
}
```

Quelques remarques :

1. Comme nous n'utilisons pas de spécification de portée, le même code s'applique à toutes les lignes
2. Le langage **awk** supporte la quasi intégralité des expressions du C.

Fonctionnement du script

Afin de pouvoir exécuter le script, vous le sauvez dans un fichier nommé, par exemple **utils.awk**. Ensuite, vous exécuterez ce script à l'aide de la commande **awk** gràce à la ligne de commande **awk -f utils.awk -** où le **-** final indique que **awk** doit traiter son entrée standard.

Il ne reste plus alors qu'à « piper » **awk** à la commande précédente, soit :

```
find /home -prune -exec du -s {} \; | awk -f utils.awk -
```

Tout d'abord, il y a exécution de la partie initialisation, c'est à dire définition et initialisation à 0 des 2 variables **NombreLignes** et **Total**. Le séparateur de colonnes est fixé au caractère « deux points », **:**.

Dans un second temps, le code de la partie règles est exécuté sur chacune des lignes du fichier. Chaque ligne est tronçonnée en colonnes en fonction du caractère séparateur, c'est à dire les deux points.

A ce moment là, $1 correspond au nom de l'utilisateur et $2 au total de ses fichiers, ce qui explique que nous fassions la sommation sur $2

Il ne reste plus alors qu'à procéder aux affichages dans la section finale.

Simplication du script

Tel quel, ce script est fonctionnel, toutefois, il est possible de l'améliorer à l'aide de simplications simples :

- Tout d'abord, il faut savoir que l'on peut définir, et donc déclarer une variable n'importe où, que ce soit dans la section de tête, la section des règles ou la section finale, et que celles-ci sont toujours initialisées à 0. Il n'était donc pas nécessaire d'initialiser nos deux variables dans la section initiale, l'on pouvait directement les utiliser sans définition préalable à l'intérieur de la section des règles.
-
- La variable prédéfinie **NR** contient (dynamiquement au cours de l'exécution) le nombre de lignes lues (*Number of Records*), à l'intérieur d'une règle, il s'agit donc du numéro de la ligne courante. Ainsi, dans la section finale, cette variable contient le nombre total de lignes du fichier. Aussi, nous pouvions nous passer de la variable NombreLignes.

Ainsi simplifié, notre script devient alors :

```
BEGIN{
FS=":";
}
{
Total+=$2;
}
END{
  printf("Nombre d'utilisateurs %d Total %d Moyenne %8.2f\n",
         NR,
         Total,
         Total/NR);
}
```

Un exemple d'utilisation de NR

Dans cet exemple, nous cherchons à connaître, en plus de la moyenne, la taille des plus petits et plus

grands comptes accompagnés du nom de l'utilisateur associé ! Pour ce faire, nous séparons le traitement en fonction du numéro de la ligne : s'il s'agit de la première, nous initialisons les variables, sinon, nous les mettons à jour ! Le résultat est le suivant :

```
BEGIN{
FS=":";
}
{
  if (NR==1)
  {
    mini=$2;
    nomMini=$1;
    maxi=$2;
    nomMaxi=$1;
  }
  else
  {
    if ($2 < mini)
    {
      mini=$2;
      nomMini=$1;
    }
    else
      if ($2 > maxi)
      {
        maxi=$2;
        nomMaxi=$1;
      }
  }
  Total+=$2;
}
END{
  printf("Nombre d'utilisateurs %d Total %d Moyenne %8.2f\n",
         NR,
         Total,
         Total/NR);
  printf("Plus gros  compte : %d blocs attribues a %s\n",mini,nomMini);
  printf("Plus petit compte : %d blocs attribues a %s\n",maxi,nomMaxi);

}
```

Ceci me permet au passage de vous présenter la structure **`if ... else`** en tout point semblable à celle du C. Les autres stuctures disponibles (toujours avec la syntaxe du C), sont **`while`**, **`do`** et **`for`**. Les commandes **`exit`**, **`break`** et **`continue`** ont également la même signification qu'en C.

UN EXEMPLE PLUS COMPLET DE AWK

Enoncé du problème

L'exemple suivant montre comment combiner efficacement les diverses options de **`awk`** pour analyser un fichier résultat d'expérience. Le fichier à étudier est le suivant :

```
probleme p1
  racine=12 -> 10 chemins, 12 iterations, valeur = 15, temps = 30
  racine=11 -> 11 chemins, 19 iterations, valeur = 19, temps = 31
  racine=14 -> 11 chemins, 20 iterations, valeur = 16, temps = 34
  racine=16 -> 18 chemins, 24 iterations, valeur = 18, temps = 39
probleme p2
  racine=12 -> 10 chemins, 11 iterations, valeur = 15, temps = 30
  racine=11 -> 11 chemins, 10 iterations, valeur = 10, temps = 31
```

```
  racine=14 -> 11 chemins, 20 iterations, valeur = 12, temps = 34
  racine=16 -> 19 chemins, 24 iterations, valeur = 14, temps = 30
probleme fini
```

Il s'agit, comme vous vous en doutez surement, de résultats d'expériences fourni par un programme de recherche opérationnelle.

Ce programme traite plusieurs problèmes, et pour chacun des problèmes, nous avons les renseignements suivants :

1. Germe du générateur de nombres aléatoires
2. Nombre de chemins générés
3. Nombre d'itérations
4. Valeur de la solution
5. Temps nécessaire au calcul

Le but du script est de fournir, pour chaque problème :

1. Le nom du problème
2. Le nombre moyen de chemins découvert
3. La meilleure solution
4. La valeur moyenne des solutions
5. Le temps moyen de résolution

Résolution

Ceci n'est guère compliqué et reprend en grande partie le même fonctionnement que le script précédent, à une différence prêt : il va falloir arrêter les calculs à chaque fois que l'on traite un nouveau problème. Heureusement, on voit qu'à chaque nouveau problème, la ligne commence par le mot **probleme**. La dernière ligne commençant également par ce mot. Il suffira d'utiliser une variable entière qui nous indiquera si l'on est en train de traiter de problème ou non pour déterminer s'il faut faire un affichage ou non.

On sait donc qu'il faut changer de problème lorsque la ligne commence par **probleme**. Sinon, c'est à dire si la ligne commence par **racine**, alors on extrait les renseignements et l'on met à jour les données internes.

```
BEGIN {
  dejaProbleme=0;
}

#  Traitement des lignes d'intitule de probleme
#  Elles commencent par le mot probleme en tete de ligne
/^probleme/ {

# Si la variable dejaProbleme est differente
# de sa valeur initiale 0, cela veut dire
# que l'on a deja traite un probleme
# dans ce cas, il faut afficher les statistiques

  if (dejaProbleme)
  {
    moyenne=sommeValeurs/nbIterations;
    printf("Probleme %s\n",nomProbleme);
    printf("  Nombre moyen de chemins      %f\n",sommeChemins/nbIterations);
    printf("  Meilleure solution           %f\n",meilleureSolution);
    printf("  Valeur moyenne des solutions %f\n",moyenne);
    printf("  Temps  moyen de resolution   %f\n",sommeTemps/nbIterations);
  }
  else
  {
# Bon cette fois on attaque le premier probleme !
    dejaProbleme=1;
```

```
  }

# De toute facon, il faut traiter les donnees presentes sur la ligne
  nomProbleme=$2

  sommeTemps=0;
  sommeChemins=0;
  sommeValeurs=0;
  nbIterations=0;
  nbProblemes++;
}

# Traitement d'une ligne de donnees
/racine.*/{
  nbIterations++;

  sommeChemins+=$3;
  valeur=$9;
  sommeValeurs+=$9;

  if (nbIterations==1)
  {
    meilleureSolution=valeur;
  }
  else
    if (valeur < meilleureSolution)
      meilleureSolution=valeur;

  sommeTemps+=$12;
}
```

1. Les portées sont spécifiées par une expression régulière entre caractères slash **/**. Toutefois, nous verrons qu'il est possible de combiner les expressions régulières avec des opérateurs booléens pour les rendre plus efficaces.
2. Toujours à l'image de **sed**, les commentaires commencent par ***#espace***.

Le fonctionnement de **awk** est décidément très semblable à celui de **sed**. Une fois la partie initiale exécutée, les portées sont examinées une par une, et le code associé à chaque portée qui vérifie la ligne est exécuté.

Compliquons un peu nos affaires. L'utilisateur souhaite désormais avoir des informations relatives à l'écart type sur la valeur des solutions. Or, l'écart type se calcule selon une formule nécessitant de connaître à la fois la moyenne et la liste des valeurs, il va donc falloir stocker cette liste des solutions dans une structure de données ! Ouf, nous disposons de tableaux avec **awk**. A l'instar des autres variables, les tableaux n'ont pas besoin d'être déclarés préalablement, et, de surcroît, ils sont redimensionnables en temps réel ! L'accès aux éléments se fait de manière traditionnelle, à l'aide de crochets.

Le résultat est illustré par le script suivant. Les lignes ajoutées sont en italique.

```
BEGIN {
  dejaProbleme=0;
}

#  Traitement des lignes d'intitule de probleme
/^probleme/ {

# Si la variable dejaProbleme est differente
# de sa valeur initiale 0, cela veut dire
# que l'on a deja traite un probleme
# dans ce cas, il faut afficher les statistiques

  if (dejaProbleme)
```

```
  {
    moyenne=sommeValeurs/nbIterations;

#   calcul de l'ecart type

    ecartType=0.0;
    for (i=1;i<=nbIterations;i++)
    {
      ecartType+=(valeurs[i]-moyenne)*(valeurs[i]-moyenne);
    }

    ecartType/=nbIterations;
    ecartType=sqrt(ecartType);

    printf("Probleme %s\n",nomProbleme);
    printf("  Nombre moyen de chemins      %f\n",sommeChemins/nbIterations);
    printf("  Meilleure solution           %f\n",meilleureSolution);
    printf("  Valeur moyenne des solutions %f\n",moyenne);
    printf("  Ecart type des solutions     %f\n",ecartType);
    printf("  Temps  moyen de resolution   %f\n",sommeTemps/nbIterations);
  }
  else
  {
# Bon cette fois on attaque le premier probleme !
    dejaProbleme=1;
  }

# De toute facon, il faut traiter les donnees presentes sur la ligne
  nomProbleme=$2

  sommeTemps=0;
  sommeChemins=0;
  sommeValeurs=0;
  nbIterations=0;
  nbProblemes++;
}

# Traitement d'une ligne de donnees
/racine/{
  nbIterations++;

  sommeChemins+=$3;
  valeur=$9;
  sommeValeurs+=$9;

  if (nbIterations==1)
  {
    meilleureSolution=valeur;
  }
  else
    if (valeur < meilleureSolution)
      meilleureSolution=valeur;

  valeurs[nbIterations]=valeur;


  sommeTemps+=$12;
}
```

Comme vous pouvez le voir, l'utilisation des tableaux est des plus simples. Toutefois, nous n'avons vu ici qu'une partie minime de leur utilisation !

Par exemple, il existe une autre forme pour la boucle **for** qui consiste à balayer l'espace des indices d'un tableau. La syntaxe est la suivante :

```
for (variable in tableau) code;
```

Attention ! contrairement à ce que laisse supposer la syntaxe (et les habitudes de programmation shell), la variable n'est pas associée directement à chaque élément du tableau mais à un index. Par exemple, dans le cas précédent, le calcul de l'écart type se ferait par :

Afin de pouvoir utiliser correctement cette astuce, il faut vider le tableau afin que la boucle sur les indices reste correcte. Supprimer un élément d'un tableau utilise la commande **delete**. Par exemple **delete tableau[1]** supprimera l'élément 1 du tableau s'il existe.

Le vidage du tableau pourra se faire très simplement en intercalant un ordre **delete** dans la boucle précédente qui devient :

```
ecartType=0.0;
    for (i in valeurs)
    {
      ecartType+=(valeurs[i]-moyenne)*(valeurs[i]-moyenne);
      delete valeurs[i];
    }
```

Cet exemple montre comment l'on peut, dans certains cas, s'affranchir d'une variable de comptage des éléments présents dans un tableau. Toutefois, il convient d'énoncer un avertissement solennel : Il ne faut pas ajouter d'éléments dans le code suivant une instruction **for (... in ...)** sous peine de planter lamentablement le programme.

,Dans tous les cas précédents, nous avons utilisé de l'adressage *a priori* numérique sur les tableaux. Or, il faut savoir que l'adressage se fait toujours par une chaîne de caractères ce qui fait de awk un langage à « tableaux associatifs »

Nous allons illustrer ce principe sur un exemple simple. Considérons par exemple, le fichier suivant recapitulatif de consommations téléphoniques.

```
Consommations telephoniques
Nom            : Unites
Mireille Dumas : 12
Claire Chazal  : 5
Claire Chazal  : 19
Mireille Dumas : 8
PPDA           : 39
Mireille Dumas : 7
PPDA           : 51
Claire Chazal  : 1
```

Chaque ligne est constituée du nom de la personne, suivi d'un point virgule et de la consommation en unités d'une communication. Nous voulons obtenir la somme des consommations pour chaque personne. Voici comment nous allons procéder :

1. Dans la section initiale, nous allons définir le caractère de séparation comme étant les deux points **:**
2. Dans la section des règles, il nous faudra :
    - définir une portée isolant les lignes de consommation. Deux attitudes sont possibles :
        - Considérer une chaîne de caractères (en fait une successions de caractères différents des deux points) suivi de **:** puis d'un espace puis d'un nombre, soit :

          ```
          /^[^:]+:/ [0-9]+
          ```

        - Supprimer toute ligne commençant par "Consommation" ou par "Nom", soit :

          ```
          ($0 !~ /^Consommations/) && ($0 !~ /^Nom/)
          ```

          que l'on peut également écrire :

```
! ( /^Consommations/ || /^Nom/)
```

en sortant la négation de l'expression.
Il ne vous reste qu'à choisir votre camp … Personnellement, je préfère toujours décrire quelque chose de positif plutôt qu'une négation, bien que dans ce cas particulier, la dernière expression soit particulièrement simple.

- Il faudra également ajouter la consommation à chaque individu. C'est ici qu'intervient la fameuse notion d'adressage symbolique. En effet, nous allons utiliser comme clef d'adressage le nom complet de la personne, soit ici **$1**, la consommation étant **$2**.
  Rappelons nous également que les tableaux sont agrandis dynamiquement lorsque l'on tente d'accéder à un indice non présent et que toute variable est initialisée à la chaîne vide/0 par défaut. Munis de ce bagage, nous pouvons écrire :

```
{
conso[$1]+=$2;
}
```

ce qui consiste bien à ajouter de la consommation à un individu ! la puissance de l'adressage associatif est sans borne !

3. Finalement, il ne reste qu'à afficher le nom de chaque individu, associé à sa consommation totale. Pour ce faire nous allons utiliser la version spécialisée du constructeur **for** :

```
for (individu in conso)
{
  printf("La personne %12s consomme %6d unites\n", individu, conso[individu]);
}
```

Pour bien comprendre comment fonctionne ce code, il faut se rappeler que la bouche **for … in** effectue une énumération des *indices*. Ainsi, **individu** passe en revue le nom des personnes et **conso[individu]** rappelle leur consommation totale

Tout ceci nous donne le code très simple suivant :

```
BEGIN{
FS=":";
}
/^[^:]+: [0-9]+/{
  conso[$1]+=$2;
}
END{
  for (individu in conso)
  {
    printf("La personne %12s consomme %6d unites\n", individu, conso[individu]);
  }
}
```

Le résultat, sur le fichier précédent est :

```
La personne Claire Chazal   consomme     25 unites
La personne Mireille Dumas  consomme     27 unites
La personne PPDA            consomme     90 unites
```

Il faut bien remarquer que dans ce cas, en utilisant un format fixe, les données en entrée se prêtent remarquablement bien à ce traitement. Qu'en serait-il si le nom des personnes pouvait changer de formattage, comme, par exemple, dans le fichier suivant :

```
Consommations telephoniques
Nom            : Unites
Mireille Dumas : 12
Claire Chazal  : 5
Claire Chazal: 19
Mireille Dumas : 8
PPDA           : 39
```

```
Mireille Dumas : 7
PPDA: 51
Claire Chazal  : 1
```

où vous ne manquerez pas de remarquer que Claire Chazal et PPDA sont écrits avec deux formatages différents. Le résultat de notre script est le suivant :

```
La personne Claire Chazal   consomme      6 unites
La personne Claire Chazal consomme     19 unites
La personne Mireille Dumas  consomme     27 unites
La personne         PPDA consomme     51 unites
La personne PPDA            consomme     39 unites
```

Comme vous pouvez le constater, les sommes ne se font plus correctement. Ceci est dû au fait que le parsage des chaînes se fait en fonction du caractère de séparation de colonnes. Comment pouvons nous résoudre ce problème pénible ? et bien, il suffit de formater nous même le nom de la personne, par exemple, en supprimant tous les caractères blancs !

Ceci peut être réalisé simplement à l'aide de la fonction standard de traitement de chaîne **gsub**. La syntaxe de cette dernière est la suivante :

```
gsub(motif_recherche, motif_remplacement, cible)
```

Si le paramètre cible est omis, alors **$0** sera traité. Bien entendu, **motif_recherche** et **motif_remplacement** peuvent être tous les deux des expressions régulières, incluant les motif spéciaux de remplacement **&** et **\1 \2 etc** pour le remplacement. Vous noterez que le remplacement est fait *sur place*, le retour de **gsub** indique le nombre de remplacements effectués.

Il s'agit presque d'une instruction de remplacement **s** de **sed** à une (grosse) différence près : les options. Dans ce cas la répétition **g** est toujours considérée. En revanche, la fonction **sub** effectue un seul remplacement, à l'instat de la commande de remplacemnt **s** de **sed** sans l'option **g**.

Donc, dans notre cas, si nous voulons supprimer les espaces dans **$1**, il convient de les remplacer par rien, c'est à dire :

```
gsub(" ","",$1)
```

Si nous incluons cette instruction dans le script, le résultat devient :

```
La personne         PPDA consomme     90 unites
La personne ClaireChazal consomme     25 unites
La personne MireilleDumas consomme     27 unites
```

Ce qui réalise bien le travail requis, mais de façon très sale :(. Aussi, utilisons les expressions régulières pour supprimer les espaces à l'avant (*leading blanks*) et à l'arrière (*trailing blanks*) de **$1**, ce qui peut se faire par les deux expressions :

```
gsub(/^ */,"",$1)
gsub(/ *^/,"",$1)
```

où, de manière raccourcie, et en utilisant une alternative, par l'instruction unique :

```
gsub(/(^ *)|( *$)/,"",$1)
```

Malgrè le côté concis de la seconde manière, je ne suis pas sur de ne pas préférer la première qui m'apparaît beaucoup plus simple ! Le résultat (enfin !) est le suivant :

```
La personne Mireille Dumas      consomme     27 unites
La personne Claire Chazal       consomme     25 unites
La personne PPDA                consomme     90 unites
```

Permettez moi désormais une petite remarque sur les tableaux. Il faut bien faire attention à ne pas

confondre **tableau[toto]** et **tableau["toto"]**. En effet, dans le premier cas, vous utilisez le contenu de la variable **toto** comme indice et dans le second, vous utilisez la chaîne immédiate **"toto"**. Cette remarque paraître sans doute triviale à chacun d'entre vous. Toutefois, j'ai déjà vu pas mal d'erreurs résulter d'une mauvaise compréhension de ce phénomène.

Le stockage des éléments dans un tableau

En fait, les éléments d'un tableau sont stockés dans une chaîne de caractères. Chaque cellule du tableau est séparée de la suivante par un caractère spécial, celui de code ASCII **"\034"**, lequel, d'après les auteurs de **awk** est sensé n'avoir qu'une probabilité anecdotique d'apparition dans un texte.

Il existe également des tableaux à 2 dimensions avec **awk**. Le système de repérage des éléments utilise des marqueurs d'indice spéciaux dans la chaîne de caractères unique de stockage. De ce fait, l'accès aux éléments d'un tableau à double indice est assez lent et l'utilisation de telles structures doit être strictement réservé aux cas désespérés.

De toute façon, il faut bien l'avouer, l'utilisation de tableaux avec **awk** est assez lente. Si la structure utilisée se révèle assez performante pour les tableaux de taille réduite, elle devient carrément innacceptable dès que la taille de ceux-ci atteint quelques dizaines d'éléments. Aussi, si le programme que vous souhaitez écrire nécessite impérativement de nombreux tableaux, il vaudra sans doute mieux se tourner vers un autre langage de programmation disposant de structures de données plus efficaces.

PARTIE RÉFÉRENCE

Loin d'être exhaustive, cette partie propose simplement à l'auteur de scripts **awk** de se repérer dans le langage !

La ligne de commande

Il est possible de passer plusieurs types d'arguments sur la ligne de commande de **awk**. Voici la syntaxe générale :

awk [-F*separateurs*] [-f *script*] ['*Commande en ligne*'] [fichiers] [arguments]

Il est possible de spécifier, soit une commande en ligne, soit un script de commande. L'un de ces deusx arguments devra être *obligatoirement* présent.

Séparateur
: Il s'agit du ou des caractères qui vont permettre de séparerer les différents champs.

Commande en ligne
: Un script shell réduit à quelques instructions et tenant sur si peu de lignes qu'on peut le laisser sur la ligne de commande

Script
: Fichier de commandes écrit en langage awk. Habituellement, mais bien que cela n'ait absolument aucun caractère obligatoire, on l'affuble de l'extension **.awk**

Fichiers
: Fichiers textes sur lequels on va appliquer les commandes du script ou de la commande en ligne. Notons que dans le cas de plusieurs fichiers, la variable **NR** est incrémentée globalement. Si l'on souhaite connaître le nombre de lignes relativement à chaque fichier, il faut utiliser **FNR**.

A partir de la version 2 de **awk**, il est possible d'ajouter des paramètres de ligne de commande genre option au script. Ceux-ci doivent être placés avant le nom du fichier et selon le formalisme **nom_param=valeur_param**, ce qui, avouons le, n'est guère pratique. Tout argument ne respectant pas ce formalisme est considéré comme un nom de fichier auquel il faut appliquer le script !

Les paramètres de ligne de commande sont toujours placés pêle mêle (c'est à dire, aussi bien les noms de fichiers que les arguments d'option) dans un tableau nommé **ARGV**, alors que leur nombre est défini par la variable **ARGC**.

Quelques compléments sur les portées

Les portées ne reposent pas nécessairement sur la reconnaissance d'une expression régulière. En fait, elles sont très générales. On peut, par exemple, utiliser une portée agissant sur le nombre de champs, par exemple :

```
NF > 4 {
...
}
```

Il est également possible de combiner plusieurs expressions à l'aide des connecteurs booléens habituels, en particulier NON (**!**), ET (**&&**) et OU (**!!**). Par exemple, la portée suivante vérifie que la ligne courante a un numéro supérieur à 10, que le nombre de champs est inférieur ou égal à 5 et que, soit le début de ligne est de la forme **[ab]+a**, soit la fin de la ligne est occupée par un caractère **#** :

```
(NR > 10) && (NF <=5) && ( ( $0 ~ /^[ab]+a/ ) || ( $0 ~ /#$/ ) )
```

Notez l'utilisation de l'opérateur de reconnaissance de motif **~**

Les variables prédéfinies

Le tableau suivant énumère la liste des variables prédéfinies dans **awk**

ARGC	Nombre d'arguments passés sur la ligne de commande
ARGV	Tableau des arguments passés sur la ligne de commande
FILENAME	Nom du fichier actuellement traité
FNR	Dans le cadre d'un traitement multi fichier, numéro de ligne dans le fichier courant (*File Number of Record*)
FS	Séparateur de colonnes (*Field Separator*)
NF	Nombre de colonnes (*Number of Fields*)
NR	Numéro de ligne courant (*Number of Record*) Dans la section finale, contient le
OFMT	Contrôle de sortie : format par défaut
OFS	Séparateur de colonnes en sortie
ORS	Séparateur de lignes en sortie
RLENGTH	
RS	Séparateur de lignes, fixé par défaut à un saut de ligne
RSTART	Positionnée par la fonction **match** : début de la chaîne reconnue
SUBSEP	Séparateur des éléments dans un tableau, par défaut, le caractère **\034**

Les fonctions utilisateur

Il est possible de definir des fonctions utilisateur afin de répéter aisément un même traitement. Il est possible de placer une définition de fonction n'importe où dans la section des règles. Toutefois, et bien que certaines versions de **awk** ne le requièrent pas, je vous conseille de toujours placer les définitions de fonction avant leur utilisation, et le plus près possible de la section initiale !

Le format général d'une fonction est le suivant :

function *identificateur* (*param 1*, *param 2*, ... ,*param n*) {

Instructions constituant la fonction

}

Les instructions et les opérateurs

Les instructions et les opérateurs reconnus par **awk** forment un sous ensemble de ceux du langage C. Aussi, tous les opérateurs du langage C sont disponibles, à l'exception de la la prise d'adresse (**&**), du déréférencement (*****) et des opérateurs de décalage (**>>** et **<<**). Pour ce qui est des opérateurs arithmétiques, **awk** a ajouté la mise à l'exposant sous la forme (**a ^ b**).

Autre particularité, seuls les opérateurs booléens purs **||**, **&&** et **!** sont présents, les versions bit à bit ont été exclues.

Trois opérateurs sont vraiment spécifiques à **awk**.

L'opérateur de concaténation de chaînes

Il n'existe pas à proprement parler d'opérateur de concaténation de chaînes : l'on utilise pour cela un simple espace. Ainsi l'expression **toto="c'est" " l'ete";** consiste à concaténer les deux chaînes de droite dans le membre de gauche, la valeur de la variable **toto** est alors la chaîne **"c'est l'été"**.

L'opérateur de matching **~**

Cet opérateur est essentiellement utilisé dans les déclarations de portée, cet opérateur booléen renvoie vrai si la variable placée à gauche de **~** est reconnue par l'expression régulière placée à droite (séparée par **/**), par exemple, si vous souhaitez appliquer des commandes aux lignes dont la quatrième est reconnue par l'expression régulière *regex* : **$4 ~ /*regex*/** Notons également qu'il existe un opérateur de non matching : **!~**

Les fonctions standard

Les fonctions arithmétiques

Le langage de programmation **awk** propose en standard la plupart des fonctions arithmétiques communes. Elles sont répertoriées dans le tableau suivant :

atan2(y,x)	Renvoie l'angle formé par x et y, soit, grosso modo, l'arc tangente de y/x
cos(x)	Cosinus de x
sin(x)	Sinus de x
log(x)	Logarithme népérien de x
exp(x)	Exponentielle de x
sqrt(x)	Racine carrée de x
int(x)	Partie entière de x
rand(x)	Génère un nombre pseudo aléatoire x, tel que 0 <=x < 1
srand(x)	Fixe la racine du générateur à x

Les fonctions de traitement de chaînes

Les fonctions de traitement de chaînes de caractères proposées par **awk** sont regroupées dans le tableau suivant :

Nom de la fonction et arguments	Description du travail effectué	Valeur renvoyée
Recherche et extraction de sous chaînes		
substr(s,p)	Extrait la fin de la chaîne **s** commençant à la position **p**	La sous chaîne en question

substr(s,p,n)	Extrait la sous chaine de **s** de longueur **n** et commençant à la position **p** Cette fonction est souvent utilisée pour obtenir les **n** premiers caractères d'une chaîne en fixant **p** à 1.	La sous chaîne en question
index(s,t)	Recherche la chaîne **t** dans la chaîne **s**	La position de la première occurrence de **t** dans **s**, 0 si **t** n'appartient pas à **s**
Fonctions à usage général		
length(s)	Longueur d'une chaîne	Longueur de la chaîne passée en paramètre
match(s,r)	Effectue un test de reconnaissance du motif **r** sur la chaîne **s**	Si la chaîne **s** est reconnue par le motif **r**, la position du premier caractère associé à la reconnaissance est renvoyée. En cas d'échec, la fonction renvoie 0. En outre les variables prédéfinies **RSTART** et **RLENGTH** sont respectivement associées à la position du premier caractère de reconnaissance (identique à la valeur de retour) et à la longueur de la partie de **s** reconnue par le motif **r**
sprintf(format,...)	Effectue le même travail que la fonction du même nom en C *i.e.* créée une chaîne sur le modèle **format** en substituant les caractères spéciaux par les argument à la suite **format** en suivant le même formalisme que **printf**	La chaîne issue des substitutions.
Fonctions de substition		
sub(r,s)	Remplace la première occurrence de **r** par **s** à l'intérieur du tampon de travail courant (**$0**)	Nombre de substitutions effectuées, c'est à dire 0 ou 1
sub(r,s,t)	Remplace la première occurrence de **r** par **s** à l'intérieur de la chaîne **t**	Nombre de substitutions effectuées, c'est à dire 0 ou 1
gsub(r,s)	Remplace toutes les occurrences de **r** par **s** à l'intérieur du tampon de travail courant (**$0**)	Nombre de substitutions effectuées, de 0 à n
gsub(r,s,t)	Remplace toutes les occurrences de **r** par **s** à l'intérieur de la chaîne **t**	Nombre de substitutions effectuées, de 0 à n
Chaînes et tableaux		
split(s,tab)	Découpe la chaîne **s** en champs (respectivement au(x) séparateur(s) indiqués par la variable prédéfinie **FS**) et place ces derniers dans le tableau **tab**.	Nombre de champs obtenus
split(s,tab,sep)	Découpe la chaîne **s** en champs (respectivement au(x) séparateur(s) indiqués par **sep**) et place ces derniers dans le tableau **tab**.	Nombre de champs obtenus

Attention aux faux amis ! **substr** ne signifie pas substitution (c'est le travail de **sub** et **gsub**) mais « extraction de sous chaîne ».

Un petit exemple d'utilisation des fonctions de manipulation de chaînes et des tableaux

Supposons que vous disposiez d'un fichier HTML dont les titres ne sont pas numérotés. Votre souhait est de produire automatiquement une numérotation légale du genre 1.2.3. Comment pouvez vous procéder avec **awk** ?

1. La première chose à reconnaître les lignes contenant un paragraphe. Hormis quelques éventuels espaces ou signes de tabulation, elles commencent par **<H** suivi d'un nombre compris entre 1 et 9. On pourra donc les retrouver grâce à l'expression **/^[\t]*<H[1-9]/**
2.
3. Dans un second temps, il faut récupérer le numéro d'ordre du paragraphe que l'on désire traiter. Afin de faciliter l'opération, nous commençons par supprimer les espaces et les tabulations de début de phrase (*leading blanks* en Angliche) et le motif **<H**. Ensuite, nous extrayons la chaîne de longueur 1 qui correspond au rang.
4. Il nous faut alors construire la chaîne qui formera le numéro du paragraphe. Pour ce faire, nous conservons dans un tableau, le numéro courant de chaque ordre. Ainsi, lorsque nous atteignons un titre d'un certain rang, le numéro courant de celui-ci est incrémenté et nous mettons à zéro le rang suivant.
   Ainsi, la construction complète du numéro consiste à balayer par une boucle **for** les rangs inférieurs, les ajouter à une chaîne suivis d'un point et ajouter le numéro du rang courant
5. La reconstruction de la ligne consiste à positionner le marqueur **<H**, suivi de son rang, du numéro que nous venons de construire et du reste de la ligne que nous extrayons à partir du reste de **$0**
6. Reste à gérer l'affichage. Toutes les lignes ne contenant pas un marqueur de début de titre doivent être affichées sans modification. Il faut donc utiliser une instruction du genre **portee { print $0}** où la portée pourrait être du type **!/^[\t]*<H[1-9]/**. Il est toutefois plus simple de modifier **$0** dans le cas des titres et de n'affecter aucune portée à la commande d'affichage ! On gagne ainsi l'évaluation d'une expression régulière !

Le code de notre script devient alors :

```
# Pas de section initiale nécessaire
# Traitement des lignes de chapitre
/^[ \t]*<H[1-9]/{
  # suppression des blancs de tete et de l'entete du chapitre
  sub(/^[ \t]*<H/,"");
  # recuperation du rang
  rang=substr($0,1,1);
  # mise a jour des niveaux ou numeros courants dans les rangs
  niveau[rang]++;
  niveau[rang+1]=0;
  # construction du numero complet
  numero=;
  for (compteur=1;compteur<rang;compteur++)
    numero=numero courant[compteur] ".";
  numero=numero courant[compteur];
  # reconstruction de la ligne
  $0="<H" rang "> " numero " " substr($0,3,length($0)-2);
}
# Affichage de toutes les lignes, modifiees ou non :)
{
  print $0;
}
# Pas de section finale
```

Vous voyez, c'est pas bien compliqué et ça rend d'innombrables services !

LES ENTRÉES SORTIES

Les sorties

Il existe deux fonctions d'affichage : **print** et **printf**. La seconde fonctionne exactement comme son homologue en C et je ne reviendrai donc pas dessus. En revanche, la fonction **print** est assez déroutante au premier abord.

Sa syntaxe est la suivante :

print [(] liste_de_constantes_et_variables [)]

Tout d'abord, remarquez que les parenthèses sont facultatives, ce qui froisse inévitablement les programmeurs C ! Fonction très versatile, **print** autorise également l'impression d'expressions complexes. Lorsque vous séparez par une virgule les différents éléments à afficher, ceux-ci sont imprimés séparés par le caractère **OFS**, un espace par défaut. Finalement, le caractère **ORS** est ajouté au bout de chaque ligne. Les petits exemples suivants permettent de fixer certaines idées.

- **print $1 $2 $3** fait appel à l'opérateur de concaténation (l'espace entre les champs) et affiche bout à bout $1, $2 et $3
-
- **{**
  OFS="\t";
  print $1,$2,$3
  } imprime le caractère **OFS**, ici une tabulation entre chacun des éléments affichés !

Il est possible d'utiliser des redirections, que ce soit avec **print** (les parenthèses autour des argument sont alors obligatoires) ou avec **printf**) pour envoyer le résultat de la commande vers un fichier ou même un tube avec les commandes habituelles **>**, **>>** et **|**. Il convient toutefois de stipuler la différence de comportement de **>** et **>>** par rapport à leurs équivalents shell. En effet, lors de sa première rencontre, la redirection **>** créée ou écrase le fichier cible, les invocations suivantes fonctionnant en mode ajout. L'utilisation de **>>** se fait toujours en ajout.

Par exemple, considérez les lignes suivantes qui reprennent le fonctionnement du programme de base de données sur la consommation téléphonique. La ligne intéressante est mise en évidence par des caractères gras :

```
BEGIN {
FS=":";
}
/^[^:]+: [0-9]+/{
  gsub(/(^ *)|( *$)/,"",$1);
  conso[$1]+=$2;
  print > $1".conso"
}
END{
  for (individu in conso)
  {
    printf("La personne %-19s consomme %6d unites\n", individu, conso[individu]);
  }
}
```

Cette nouvelle ligne a pour effet de créer un fichier **"Consommateur.conso"** retraçant la consommation individuelle de chaque personne. A chaque lancement du script, ce fichier est supprimé et son contenu est remis à zéro.

L'exemple suivant utilise un pipe pour trier certaines lignes. Notez l'utilisation de quotes autour de la commande à invoker. L'appel ne se fait pas à travers un nouvel interpréteur de commande mais utilise un bon vieux **fork** des familles.

```
NF = 5 {
  print ($1 $4 $2 $3) | "sort +3rn"
}
```

Nous terminerons sur les entrées en précisant qu'il est possible de fermer, c'est à dire en fait de vider les buffers et de cloturer le descripteur, une liaison de sortie. J'entends par liaison vers l'extérieur, soit un

fichier, soit un pipe. A quoi sert l'opération de fermeture ? Tout d'abord, si vous avez écrit dans un fichier que vous voulez relire, il est impératif de le cloturer préalablement à toute lecture afin de s'assurer que les tampons ont bien été vidés. En outre, cela permet d'éviter de saturer les tables de descripteurs de fichiers ouverts.

Les entrées

Ces dernières reposent essentiellement sur l'utilisation de la commande **`getline`** dont voici un résumé des différentes syntaxes possibles :

`getline`	Lit et parse une nouvelle ligne depuis le fichier courant dans les variables standard $0, $1, etc.	$0, $1 ... , NR, FNR, NF
`getline var`	Lit une nouvelle ligne depuis le fichier courant vers la variable **`var`**. La ligne n'est pas parsée.	juste **`var`**
`getline < "fichier"`	Lit et parse une nouvelle ligne depuis le fichier **`"fichier"`** dans les variables standard $0, $1, etc.	$0, $1 ..., NF
`getline var < "fichier`	Lit une nouvelle ligne depuis le fichier **`"fichier"`** vers la variable **`var`**. La ligne n'est pas parsée.	juste **`var`**
`"commande" | getline`	Lit une nouvelle ligne dans le tube *i.e.* depuis la sortie standard de **`commande`** et vers les variables standard $0, $1, etc.	$0, $1 ..., NF
`"commande" | getline var`	Lit une nouvelle ligne dans le tube *i.e.* depuis la sortie standard de **`commande`** et vers la variable **`var`**.	juste **`var`**

Dans tous les cas, **`getline`** renvoie 0 si la fin de fichier est atteinte et autre chose (1, la plupart du temps) dans tous les autres cas. Vous remarquerez que la ligne n'est pas parsée lorsqu'elle est lue dans une variable. Néanmoins, il vous est possible de la scinder avec **`split`**. Utiliser **`getline`** vous permet d'interagir sur différents fichiers, mais aussi de modifier l'algorithme standard de manipulation du fichier en entrée.