AWK

Alice

2016

Table des matières

1 Introduction 2

2 Pourquoi et comment 2

3 Structure générale 3

4 Syntaxe de base 5

5 Les variables plus en détail 6

6 Tableaux 8

7 Format de sortie des nombres 10

8 Fonctions 11

9 Expressions rationnelles 12

10 Mélanger du script shell et du AWK comme un gros crade 13

1 Introduction

Il existe sans doute déjà pas mal de tutoriels AWK sur le web, mais je n'ai jamais rien vu de super pratique, et j'ai toujours l'impression que les internautes s'interdisent de mettre des espaces ou des retours à la ligne dans leurs scripts, ce qui m'énerve particulièrement. De plus, en faire un moi-même est plutôt rigolo et me permet de tester des trucs en LaTeX. Donc : go, go.

La plupart des distributions Linux (toutes ?) sont fournies avec une implémentation de AWK, mais il n'est pas rare qu'il s'agisse de MAWK : une version minimaliste. Je recommande d'installer GAWK (GNU AWK), qui est minuscule et ajoute des trucs utiles. Si on ne touche à rien, la commande `awk` lance la version la plus cool qu'on a installée. Normalement, `man awk`, `which awk` ou `awk --version` vous permettront de voir facilement ce que vous faites tourner.

2 Pourquoi et comment

Le principe de AWK est de lire un ou plusieurs fichiers, ligne par ligne, et d'exécuter un script réagissant aux données qui passent. La plupart du temps, on balance des choses sur la sortie standard ; AWK ne modifie pas les fichiers d'entrée.

En général, on utilise AWK d'une de ces quatre manières :

```
1 commandes | awk 'script en dur'
2 commandes | awk -f 'script.awk'
3 awk 'script en dur' 'fichier_1' ... 'fichier_n'
4 awk -f 'script.awk' 'fichier_1' ... 'fichier_n'
```

Bref, rien de très révolutionnaire : deux manières de passer les données (entrée standard ou fichiers passés en arguments), et deux manières de passer un script à exécuter (fichier passé avec `-f` ou script directement écrit comme un argument).

Vous pouvez aussi pondre un fichier exécutable AWK à la manière de ce qu'on fait avec Bash. Il faut généralement pour cela écrire un *shebang* tel que `#!/usr/bin/awk -f`. Et n'oubliez pas de vous donnez les droits d'exécution sur le fichier !

La figure 1 contient un schéma de toute beauté qui montre rapidement comment AWK perçoit les données qu'on lui passe en entrée.

Les fichiers sont donc lus successivement, et les données ainsi formées sont découpées en « records » (terme anglais utilisé par AWK ; on pourrait parler d'enregistrements) et en « fields » (les bons vieux champs). Par défaut, les enregistrements sont les lignes, et les champs sont séparés par des espaces ou des tabulations. AWK est donc très pratique dès que vos données sont un peu structurées. Bien entendu, les séparateurs d'enregistrements et de champs peuvent facilement être changés afin de s'adapter à des besoins particuliers. On peut aller jusqu'à mettre des séparateurs vides pour examiner les caractères un à un, donc bon...

FIGURE 1 – Données !

3 Structure générale

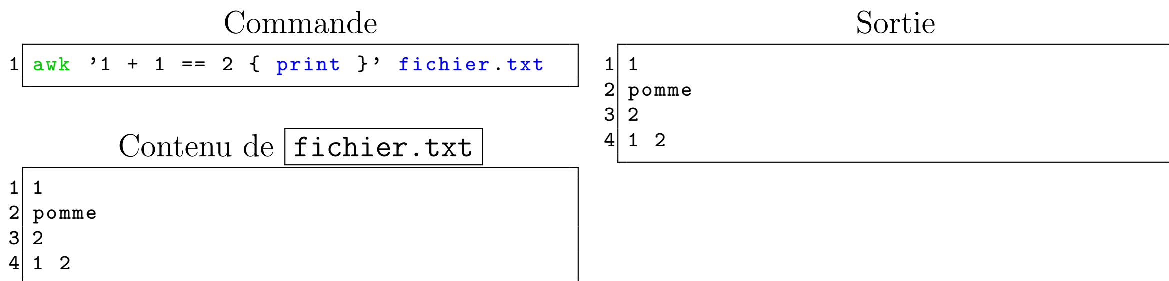
Un script AWK typique est composé de blocs, délimités par des accolades :

```
{
    blablabla
}

{
    blabla
    bla
    blablabla
}
```

Chaque bloc peut être associé à une condition (qui peut être très simple ou très complexe, c'est selon). À chaque fois qu'AWK lit une ligne de données, il parcourt les blocs dans l'ordre en exécutant ceux dont la condition est vérifiée, puis passe à la ligne suivante des données.

Ainsi, dans l'exemple qui suit, l'intégralité du fichier d'entrée est restituée car la condition, `1 + 1 == 2`, est toujours vraie, et car le bloc contient la commande `print`, qui affiche la ligne de données considérée.



Commande

```
awk '1 + 1 == 2 { print }' fichier.txt
```

Contenu de `fichier.txt`

```
1
pomme
2
1 2
```

Sortie

```
1
pomme
2
1 2
```

À l'inverse, la commande suivante restera muette car sa condition sera toujours

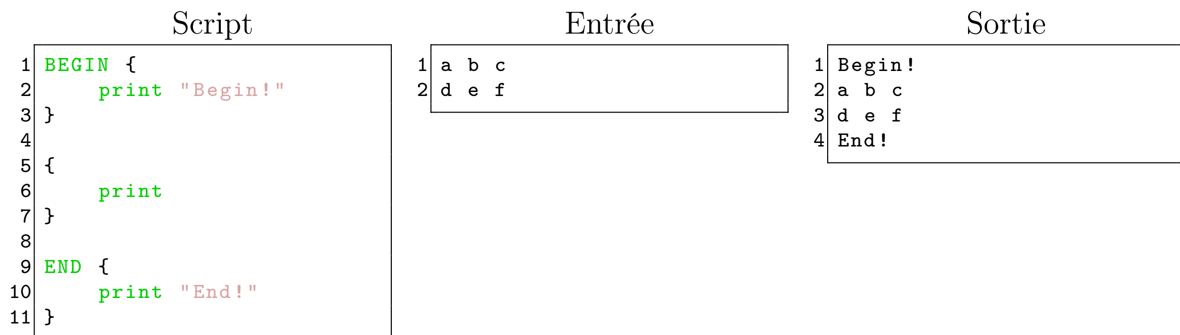
fausse et le bloc associé jamais exécuté :

```
awk '1 + 2 > 9 { print }' fichier.txt
```

En cas d'absence de condition, le bloc est *toujours* exécuté. `awk '{ print }'` aura donc le même effet que notre exemple du `1 + 1 == 2`. Les blocs sans condition sont très utiles, par exemple pour appliquer un traitement systématique à chaque ligne des données ou pour incrémenter des compteurs.

Une condition peut également n'être suivie par aucun bloc, auquel cas AWK fera comme s'il y avait un bloc `{ print }` associé à cette condition. Cela permet de rapidement faire des filtres simples n'affichant que certaines lignes de l'entrée.

Il existe deux conditions particulières à connaître impérativement : `BEGIN` et `END`.



Script

```
BEGIN {
    print "Begin!"
}

{
    print
}

END {
    print "End!"
}
```

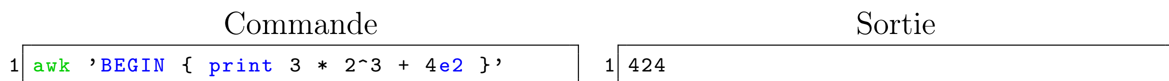
Entrée

```
a b c
d e f
```

Sortie

```
Begin!
a b c
d e f
End!
```

Rien de très traumatisant, donc : les blocs `BEGIN` sont exécutés avant de lire les données, et les `END` à la fin. Notez cependant qu'il y a une différence subtile : le bloc `BEGIN` est exécuté même s'il n'y a aucune donnée en entrée, tandis que le `END` est alors ignoré. Ainsi, `awk 'BEGIN { print "plop" }'` affiche bien « plop », mais `awk 'END { print "plop" }'` glandouille en attendant que vous balanciez quelque chose sur son entrée standard. Ainsi, avec un script contenant juste un bloc `BEGIN`, on peut faire des calculs ou des traitements un peu débiles mais parfois très utiles, et ce sans s'embêter avec trop de paramètres bizarres :



Commande

```
awk 'BEGIN { print 3 * 2^3 + 4e2 }'
```

Sortie

```
424
```

Remarquez au passage via cet exemple que AWK arrive assez bien à se dépatouiller avec les notations scientifiques, exposants, etc.

Enfin (important !) notez que

```
condition
{
    code
}
```

est interprété comme :

```
Condition sans bloc, qui fait comme s'il y avait un { print }.

Bloc sans condition, exécuté pour toutes les lignes.
```

Il faut bien mettre l'accolade sur la même ligne que la condition :

```
condition {
    code
}
```

Je ne crois cependant pas pour le moment avoir rencontré d'autres cas où AWK attachait de l'importance à la position des accolades ouvrantes : vous pouvez revenir à la ligne sans problème après la condition d'un `if`, avant d'ouvrir une boucle, et ainsi de suite.

4 Syntaxe de base

La syntaxe générale ressemble à celle du langage C, et donc aussi au Java et compagnie (je ne reviendrai pas vraiment sur les opérateurs de base, a priori). Notez cependant que les point-virgules sont facultatifs (sauf si vous ne mettez pas de retour à la ligne : `print print` ne fera pas la même chose que `print; print`).

```
BEGIN {
    a = 5
    a++
    b = a + 3
    # Commentaire.
    # Il n'y a pas de syntaxe pour
    # faire des blocs de commentaires,
    # mais n'en voulez pas trop au
    # langage. Merci.
    b /= 3
    txt = "b = " b
    print txt
}
```

Sortie

```
b = 3
```

On peut remarquer plusieurs choses :

— Pas besoin de déclarer les variables. Ça permet de rapidement coder des trucs utiles, mais certaines fautes de frappe ne pardonnent pas : plantez-vous dans le nom d'une variable, et vous vous retrouverez avec un truc vide, sans qu'aucune erreur ne soit rapportée. Heureusement, il y a pas mal d'outils de débug qui permettent de remarquer ce type d'erreurs, et puis, si on utilise l'autocomplétion d'un éditeur pour pondre un script (je code mon AWK avec Geany), en général tout se passe bien.

— Les variables sont typées dynamiquement : j'ai mis du texte dans la variable `txt` sans vraiment prévenir ni rien. Notez qu'on peut changer le type d'une variable quand on veut, au fil des affectations et des opérations. Quand il s'agit de nombres, ils sont gérés comme des nombres flottants et non comme des entiers.
— On peut concaténer un peu tout et n'importe quoi juste en mettant des éléments côte à côte, comme je l'ai fait avec `"b = " b`.

Ce script d'exemple met donc la valeur `5` dans la variable `a`, incrémente cette variable de `1`, puis initialise `b` avec `a + 3`, ce qui donne `9`. La valeur de `b` est ensuite divisée par `3`, ce qui la ramène à `3`. La variable `txt` est initialisée en mettant bout à bout le texte « `b = ` » et la valeur de `b`, ce qui donne « `b = 3` », et ce texte est affiché.

Puisqu'il s'agit d'un bloc dont la condition est `BEGIN`, tout cela est effectué une et une seule fois, avant même de lire les éventuelles données d'entrée (fichiers à lire ou texte envoyé sur l'entrée standard).

5 Les variables plus en détail

Une variable non initialisée vaut `0` si on l'utilise comme un nombre, et équivaut à une chaîne vide si on l'utilise comme une chaîne de caractères. Ainsi, si `x` n'a jamais servi, `if(x == "") print "plop"` et `if(x == 0) print "plop"` afficheront tous les deux « plop ».

Les variables conservent leur valeur d'un bloc à l'autre, ainsi que lorsqu'on passe à la ligne suivante des données. Cela peut sembler sale, mais c'est en fait généralement ce dont on a besoin.

Pas mal de variables spéciales très utiles existent. Ce qu'elles ont de « spécial », c'est qu'elles sont mises à jour automatiquement. La liste complète traîne dans le manuel, mais en gros :

`NR` (« Number of Records », je suppose) donne le numéro de la ligne (ou de ce qui vous sert de ligne si vous avez bidouillé les séparateurs). Pour la première ligne, c'est 1, pas 0. On peut voir ça comme « le nombre d'enregistrements déjà lus, en comptant celui qu'on est en train de traiter ».

`FNR` Comme `NR`, mais celui-là est réinitialisé quand on passe au fichier de données suivant, alors que `NR` donne vraiment le nombre total d'enregistrements lus.

`NF` (« Number of Fields », sans doute) donne le nombre de champs de la ligne courante. Notez qu'avec les séparateurs par défaut, la ligne :
`a<ESPACE><TABULATION><ESPACE><ESPACE>b`
ne comporte que deux champs, contenant `a` et `b` respectivement.

`FS` (« Field Separator » ?) permet de savoir quel est le séparateur de champs actuel, ou de le changer si vous faites `FS = "un truc"`. Les utilisations les plus courante, de ce que j'en sais, sont de mettre une virgule en séparateur pour lire un fichier CSV (ou, peut-être mieux, `"[ \t]*,[ \t]*"` – notez au passage

que ça marche avec des expressions rationnelles), ou de mettre une tabulation en séparateur pour lire des données qui contiennent des espaces.

`RS` (« Record Separator », si ma logique est bonne) fonctionne comme `FS`, mais pour les enregistrements. J'avoue ne pas trop toucher à celui-là.

Il y a toute une flopée de variables intéressantes de ce genre, mais je n'ai pas envie de me taper toute la liste. Si vous avez un besoin particulier ou que vous êtes curieux, vous pouvez chercher ces « built-in variables » dans le manuel.

Le plus intéressant reste sans doute les variables désignant les champs de la ligne de donnée en cours de traitement : `$1` désigne le contenu du premier champ de la ligne courante, `$2` le second, etc. `$0`, quant à elle, correspond à la ligne dans son intégralité. Tenter d'accéder à un `$<n>` avec un `<n>` dépassant le nombre de champs disponibles (`NF`) ne provoque pas une erreur et renvoie simplement du vide ou 0, comme avec toute variable non initialisée.

Script

```
1 {
2     print $2
3 }
```

Entrée

```
1 a b c
2 1 2 3 4
3 x
4 i j k
```

Sortie

```
1 b
2 2
3
4 j
```

Cette syntaxe du `$` fonctionne également avec une variable contenant un numéro de champ ! Si vous faites `a = 2` puis `$a`, vous obtiendrez le champ numéro 2. Mieux : vous pouvez utiliser le résultat d'une expression comme indice de champ (mais attention alors aux priorités d'opérateurs ; armez-vous de parenthèses) : `$(a + 1)` donne le champ situé juste après l'indice contenu dans `a`.

Script

```
1  NR % 2 == 0 {
2      print "Ligne : [" $0 "]"
3
4      for(k = 1; k <= NF; k++)
5          print "Champ " k " : " $k
6
7      if(NF > 4) {
8          print "Plus de 4 champs !"
9          print "Voici l'avant-dernier :"
10         print $(NF - 1)
11     }
12 }
13
14 END {
15     print "Nombre d'enregistrements lus
        : " NR
16 }
```

Entrée

```
1 a b c
2 pomme poire
3
4 1 2 3 4 5
```

Sortie

```
1  Ligne : [pomme poire]
2  Champ 1 : pomme
3  Champ 2 : poire
4  Ligne : [1 2 3 4 5]
5  Champ 1 : 1
6  Champ 2 : 2
7  Champ 3 : 3
8  Champ 4 : 4
9  Champ 5 : 5
10 Plus de 4 champs !
11 Voici l'avant-dernier :
12 4
13 Nombre d'enregistrements lus : 4
```

Ce script exécute un bloc pour les lignes paires (modulo 2 nul) et finit par afficher le nombre d'enregistrements lus (j'en profite pour vous montrer que `NR` reste utile

dans le bloc `END`). Notez aussi que `print $0` est équivalent à `print`. D'ailleurs, pas mal de fonctions de AWK marchent comme ça, avec des paramètres optionnels qui valent `$0` si on ne spécifie rien. On rencontre notamment ça dans les fonctions de manipulation de chaînes de caractères, qui s'appliquent à la ligne de donnée examinée si on ne donne rien. J'ai aussi voulu vous montrer que la ligne vide ne comptait pas pour du beurre aux yeux de `NR`. Enfin, notez la très classique boucle `for` visant à parcourir tous les champs.

Oh, j'oubliais : il est possible d'affecter des valeurs aux variables en « $ » qui représentent les champs. L'exemple qui suit affiche ainsi chaque ligne après avoir changé brutalement la valeur du second champ et échangé celles du premier et du troisième.

Script

```
{
    $2 = "plop"
    temp = $1
    $1 = $3
    $3 = temp
    print
}
```

Entrée

```
a b c
12 34 56
pomme poire fleur rigolo
```

Sortie

```
c plop a
56 plop 12
fleur plop pomme rigolo
```

Pour régler la valeur d'une variable avant même que le programme ne débute, vous pouvez utiliser l'option `-v` de AWK. Par exemple, `-v FS="plop"` mettra « plop » en séparateur de champs sans que vous ayez besoin de faire une affectation dégueulasse dans votre `BEGIN`. Ainsi, `awk -v FS=$'\t' '{ print $1 }'` affichera la première colonne de l'entrée en ne considérant pas les espaces comme des séparateurs de colonnes ; seules les tabulations serviront au découpage. Et oui, j'en profite pour rappeler la syntaxe bien utile du `$'\t'` qui évite de mettre des tabulations « en dur » à des endroits bizarres (le `$'\n'` existe aussi, entre autres).

Je vais essayer de finir cette section sur un autre truc intéressant : il existe une version « sortie » de `FS` : `OFS`. Ce séparateur (un simple espace, par défaut) est utilisé quand vous employez une syntaxe que je ne savais pas où évoquer : `print a, b, $1, c, $4`. Les virgules, ici, séparent les trucs à afficher, et `print` les séparera par l'`OFS`.

`awk 'BEGIN { OFS = "/"; a = "patate"; print 1, "txt", a, 3 }'` donne « 1/txt/patate/3 ».

6 Tableaux

Les tableaux de AWK sont à la fois assez cools et assez chelous. En fait, ce sont des tableaux associatifs, genre table de hachage ou dictionnaire (je ne sais jamais quels termes sont les plus adaptés). En gros, vous pouvez ranger des données dans un tableau en disant « à telle "clef" correspond telle "valeur" ». L'exemple suivant devrait rapidement clarifier tout ça.

Script

```
{
    tableau[$1] = $2
}

END {
    print tableau["animal"]
    print tableau[12]
    print length(tableau)
}
```

Entrée

```
lieu     Paris
animal   porc
28       vingt-huit
plante   tulipe
12       douze
```

Sortie

```
porc
douze
5
```

Ici, je range dans le tableau, pour chaque ligne, la valeur se trouvant dans la deuxième colonne, en choisissant la valeur de la première colonne comme indice. Pour la retrouver, il suffit de consulter le tableau au bon indice. Bien entendu, si vous n'avez pas besoin de choses compliquées, rien ne vous empêche (je dis souvent « rien ne vous empêche », tiens...) d'utiliser de bêtes nombres de 0 à je ne sais combien comme indices, éventuellement en laissant des trous. Accéder à une « case » non initialisée d'un tableau en donnant un indice qui n'a jamais été utilisé lors d'une affectation donne à peu près la même chose que pour les variables non initialisées (voir section 5).

`length(tableau)` donne le nombre d'éléments qui ont été rangés dans le tableau. Notez cependant qu'à la base elle sert surtout à donner la longueur d'une chaîne de caractères et que cette application aux tableaux est une extension apportée par GAWK. De ce fait, `mawk 'BEGIN { t[0] = 1; print length(t) }'` donnera une erreur, tandis que la même commande avec `gawk` à la place de `mawk` marchera très bien. Il est donc parfois important de savoir à quelle implémentation de AWK on a affaire. Je recommande même d'appeler explicitement `gawk` plutôt que `awk` dès qu'on utilise des fonctionnalités avancées : ça balancera des messages d'erreurs plus clair (genre « wèsh, gawk ça n'existe pas sur ce PC ») que si MAWK s'étouffe sur un truc qu'il ne connaît pas.[1]

On peut assez facilement parcourir l'ensemble des éléments d'un tableau, comme le montre ce nouvel exemple.

Script

```
BEGIN {
    t["plop"] = 12
    t[28] = "blub"
    t[-4] = 99

    for(clef in t) {
        print clef "\t--> " t[clef]
    }
}
```

Sortie

```
plop    --> 12
28      --> blub
-4      --> 99
```

Par contre, quand on parcourt un tableau comme ça, ça peut être un peu le

1. Je n'ai pas trop envie de m'embêter à vous dire ce qui marche ou non avec telle ou telle implémentation ; il y a plein de « AWK feature comparisons » sur le net qui font déjà ça très bien.

bordel niveau ordre (ça n'étonnera pas ceux qui ont déjà bossé avec des tables de hachage). Je vous met un exemple, comme ça vous ne pourrez pas dire que je ne vous ai pas prévenus.

Script

```
BEGIN {
    t[1] = "un"
    t[20] = "vingt"
    t[30] = "trente"

    for(clef in t) {
        print clef "\t--> " t[clef]
    }
}
```

Sortie

```
20      --> vingt
30      --> trente
1       --> un
```

J'étais à deux doigts d'écrire que `t[1]` n'était pas équivalent à `t["1"]`, et en vérifiant j'ai réalisé que si. Ma crédibilité va en prendre un coup... Notez que je vérifie à peu près tout, quand même, donc je vaut a priori mieux que pas mal de gens, tout de même.

7 Format de sortie des nombres

Par défaut, AWK a tendance à balancer de la notation scientifique à tout va, ce qui peut être gênant quand on veut traiter sa sortie avec des commandes trop idiotes pour comprendre ce que veut dire « 2e-4 ». Fort heureusement, la variable spéciale `OFMT` (« Output ForMaT », je suppose) permet de choisir le format de sortie par défaut des nombres. On peut aussi choisir un format au dernier moment avec `printf`, dont vous ne manquerez pas de trouver la syntaxe dans le manuel.

Les formats sont gérés par des trucs dégueulasses à base de `%` qui n'étonneront cependant pas ceux qui ont déjà fait du C. Les détails se trouvent là aussi dans le manuel ; il devrait suffire de chercher « The printf Statement ». Voici un petit exemple de rien du tout où le résultat de multiplications est affiché avec quatre formats différents.

Script

```
{
    print " == " $1 " * " $2 " =="

    res = $1 * $2

    OFMT = "%.5g"
    print res

    OFMT = "%d"
    print res

    OFMT = "%.3f"
    print res

    OFMT = "%2d"
    print res

    printf "\n"
}
```

Entrée

```
2.0001  10e-10
0.12345 10
```

Sortie

```
 == 2.0001 * 10e-10 ==
2.0001e-09
0
0.000
 0

 == 0.12345 * 10 ==
1.2345
1
1.235
 1
```

On remarque que limiter à trois le nombre de chiffres après la virgule avec `%.3f` a provoqué un arrondi automatique qui a changé 1.2345 en 1.235.

Pour régler le format une bonne fois pour toutes, personne ne vous empêche de mettre un `OFMT = "truc"` dans un bloc `BEGIN` ou d'utiliser l'option `-v` évoquée en section 5.

8 Fonctions

Comme dans la plupart des langages, AWK permet à l'utilisateur de définir des fonctions pour les réutiliser ensuite à loisir.

Script

```
function une_fonction(param_x, param_y,
     param_z)
{
    print "param_z = [" param_z "]"
    if(param_x == param_y)
        return "egal"
    else
        return "pas egal"
}

BEGIN {
    print une_fonction(1, 2, 3)
    print une_fonction(4, 4)
}
```

Sortie

```
param_z = [3]
pas egal
param_z = []
egal
```

On remarque que tous les paramètres sont considérés comme optionnels, ce qui m'a permis de laisser `param_z` complètement en plan lors du second appel. À vrai dire, le manuel conseille même d'ajouter des paramètres inutiles pour faire des variables locales... La convention est alors de séparer les « vrais » paramètres de ceux qui servent juste à ça par des espaces supplémentaires.

Quand on balance une variable en paramètre d'une fonction, cela copie sa valeur :

si la valeur de la variable est modifiée DANS la fonction, ces modifications ne restent pas effectives une fois que l'on en sera sorti. Pour les tableaux, cependant, cela fait du « passage par référence » : ils peuvent être modifiés dans la fonction. Pour les variables n'étant pas passées en paramètre, vu que grosso modo tout est global, on peut aussi modifier leur valeur. L'exemple suivant met tout cela plus en lumière.

Script

```
function f(var, tab)
{
    truc_global++
    var++
    tab[0]++
}

BEGIN {
    truc_global = 1
    v = 1
    t[0] = 1

    f(v, t)

    print "truc_global = " truc_global
    print "v = " v
    print "t[0] = " t[0]
}
```

Sortie

```
truc_global = 2
v = 1
t[0] = 2
```

9 Expressions rationnelles

En général, quand on bouffe de gros fichiers relous, on a besoin d'expressions rationnelles. AWK les intègre assez bien. Ainsi, une simple commande telle que `awk '/a[0-9]*a/'` suffit à faire un filtre sympathique : ici, ça n'afficherait que les lignes où on a pu trouver deux a séparés par rien d'autre que des chiffres. Enfin bon, je vais ptet arrêter avec les commandes trop simples, sinon vous allez utiliser `grep`... Voyons un « vrai » script.

Script

```
/abc/ && $1 ~ /123/ {
    if($2 !~ /plop/)
        print "OK : " $0
    else
        print "Presque OK : " $0
}
```

Entrée

```
plop123plup hahaplop   abc
zplop       plophoho   1abc1
blub123     truc   abc00
```

Sortie

```
Presque OK : plop123plup hahaplop   abc
OK : blub123     truc   abc00
```

Cet exemple montre trois manières d'utiliser les expressions rationnelles :

- Juste un `/.../`, pour voir si l'enregistrement entier (`$0`, quoi) colle à une expression.
- `truc ~ /.../`, pour voir si un truc colle à une expression.
- La négation de la version précédente, avec un vieux point d'exclamation qui tape l'incruste.

En résumé, notre exemple d'il y a deux secondes traite les lignes qui, à la fois, contiennent « abc » et ont un premier champ qui contient « 123 ». À l'intérieur du bloc d'instructions, le `if` n'est content que quand le second champ de l'enregistrement courant ne contient PAS « plop ».

Notez que ce sont des expressions rationnelles en mode étendu, donc pas besoin d'échapper un `+` pour obtenir le sens « une fois ou plus », pas besoin d'échapper les parenthèses pour faire un bloc, etc. Ça plaira à certains et pas à d'autres, et ça ne me regarde pas tant que ça.

À ma grande tristesse, il n'est pas possible de faire de *« backreferences »* (chopper un bloc avec des parenthèses et le référencer avec des trucs genre `\1`) quand on fait de tels tests. Par contre, c'est généralement possible dans les fonctions telles que `gensub` qui charcutent des chaînes de caractères. Il y en a toute une tripotée dans GAWK (et j'ai toujours le né fourré dans le manuel car je ne me rappelle jamais de l'ordre des paramètres). Notez cependant qu'il vous faudra alors échapper les antislashs. J'vais vous mettre un exemple, tiens.

Script

```
1 {
2     print gensub("(.)_(.)", "\\2_\\1",
          "g")
3
4 }
5
6 END {
7     print gensub("a", "X", 3, "
          abababababa")
8 }
```

Entrée

```
1 plop fleur_tulipe rigolo
2 a_b c d e_f g
```

Sortie

```
1 plop fleut_rulipe rigolo
2 b_a c d f_e g
3 ababXbababa
```

J'ai constaté en pondant ces exemples que ces *backreferences* ne fonctionnaient que dans la partie substitution ; je suis un peu déçu. Bref, j'ai voulu vous montrer deux trois trucs : si on zappe le dernier paramètre, qui représente la chaîne « cible », `gensub` s'attaque à l'enregistrement courant. Je vous avais prévenu : pas mal de trucs prennent `$0` comme paramètre par défaut. Ensuite, le « g » permet de remplacer chaque occurrence, tandis qu'un nombre permet de remplacer la $k$ième. Pour les détails ésotériques et les autres fonctions du même style, vous avez le manuel.

10 Mélanger du script shell et du AWK comme un gros crade

Mine de rien, c'est souvent utile. J'aime partir d'un script Bash tout bête et appeler plein de programmes chelous dedans.

Script Bash

```bash
#!/usr/bin/env bash

var_bash_num=1
var_bash_txt="a"

awk_output="$(
    awk '
        BEGIN {
            var_awk_num = '"$var_bash_num"'
            var_awk_txt = "'"$var_bash_txt"'"

            var_awk_num++
            var_awk_txt = var_awk_txt "EDITED"

            print var_awk_num
            print var_awk_txt
        }
    '
)"

var_bash_num="$(head -1 <<< "$awk_output")"
var_bash_txt="$(tail -1 <<< "$awk_output")"

echo "var_bash_num = $var_bash_num"
echo "var_bash_txt = $var_bash_txt"

exit 0
```

Sortie

```
var_bash_num = 2
var_bash_txt = aEDITED
```

(Comme je m'y attendais, la coloration syntaxique part complètement en vrille, avec mes conneries.)

En gros, là, je vous montre comment récupérer, dans votre script AWK, des valeurs de variables du shell, puis l'inverse : je demande à AWK d'afficher ce que j'estime utile, je capture le tout avec Bash, et je parse cette sortie comme je l'entends.

Là où c'est un peu technique, c'est peut-être surtout au niveau des ouvertures et fermetures de guillemets... Dans le `var_awk_num = '"$var_bash_num"'`, le premier guillemet simple met de côté le script AWK pour revenir à quelque chose que Bash pourra interpréter ; le guillemet double suivant protège les éventuels caractères chelous du contenu de la variable Bash. Ensuite, bah je balance le nom de la variable avec un vieux dollar, comme d'habitude en Bash, je ferme mes guillemets doubles de protection, et je remets un guillemet simple pour reprendre l'écriture du script AWK. Pour la variable dans laquelle j'ai mis du texte, c'est encore plus tordu, car il faut préparer des guillemets doubles supplémentaires dans le script AWK pour qu'il puisse réceptionner la chaîne, sinon vous avez un truc genre `var = texte` au lieu de `var = "texte"`, et `texte` sera interprété comme un nom de variable...

L'idée est de pousser Bash à balancer une partie du script AWK. Vous pouvez même mettre un bloc `$(...)` au milieu de votre script et vous servir d'`echo` pour printer des lignes de AWK qui s'ajouteront au script... Notez que Bash résoudra d'abord tout les trucs à lui qui traînent au milieu du script avant d'appeler AWK,

donc pas question d'essayer d'incrémenter une variable Bash à répétition en faisant `'((var_bash++))'` au beau milieu de votre script AWK. Je viens de vérifier et je ne sais même pas pourquoi j'ai fait ça car ça n'a vraiment aucune raison de marcher !

Comme souvent avec les trucs bien sales, il y a une foule de possibilités. À vous de trouver celle qui vous donne le moins la nausée. Je recommande l'usage des guillemets simples pour délimiter vos scripts AWK, puisque AWK utilise pas mal de dollars et de guillemets doubles et que c'est pénible à échapper. Mais bon, faites comme ça vous chante, hein.

Un dernier pour la route : balancer à AWK un tableau de Bash.

Script Bash

```
 1 #!/usr/bin/env bash
 2
 3 t_bash[0]="zero"
 4 t_bash[1]="un"
 5 t_bash[2]="deux"
 6
 7 awk '
 8     BEGIN {
 9         '"$(
10             for ((k_bash = 0;  k_bash < ${#t_bash[@]};  k_bash++))
11             do
12                 echo 't_awk['"$k_bash"'] = "'"${t_bash[k_bash]}"'"'
13             done
14         )"'
15
16         # Maintenant, t_awk est rempli.
17         for(k_awk = 0;  k_awk < length(t_awk);  k_awk++)
18             print "t_awk[" k_awk "] = " t_awk[k_awk]
19     }
20 '
21
22 exit 0
```

Sortie

```
1 t_awk[0] = zero
2 t_awk[1] = un
3 t_awk[2] = deux
```

Il faut un peu s'accrocher pour comprendre celui-là (ou l'avoir écrit soi-même), mais n'empêche qu"on peut être amené à pondre des trucs de ce genre, des fois... même s'il vaut mieux bien penser son script au préalable pour éviter ça, je suppose.

Bref, dans ce dernier exemple, je rempli, en Bash, un tableau Bash tout bête. Un bloc `$(...)` se trouvant dans mon script AWK est exécuté, et il contient une boucle Bash qui parcourt mon tableau. Cette boucle Bash balance sur la sortie standard, avec `echo`, des lignes de AWK, qui sont en fait des affectations de valeurs dans un tableau (`t_awk[0] = "zero"`, etc.). Une fois l'exécution de ce bloc terminé, Bash lance AWK sur la concaténation du début du script (`'...'`), de la sortie du bloc `$(...)`, et de la fin du script. AWK se retrouve donc à exécuter les affectations que les `echo` ont balancées, ce qui lui permet d'initialiser un tableau avec les valeurs du tableau de Bash. Ensuite, j'affiche ces valeurs avec AWK pour vous montrer que ça marche, quoi.

Dans certains cas, pour éviter ce genre de trucs, on peut simplement balancer des valeurs sur l'entrée standard de AWK ou utiliser, encore une fois, l'option `-v` pour faire une jolie affectation avant que le script ne débute.

Ainsi, en Bash,
`awk '{ print $1 + $2 }' <<< '12 28'` donne 40, et
`a=3; awk -v b="$a" 'BEGIN { print b }'` affiche bien 3.

Il y a aussi une fonction `system` qui exécute du Bash via AWK, mais ça a tendance à lancer un sous-shell, donc les affectations faites sur les variables ne font plus effets dès que la commande passée à `system` se termine. Enfin bon, on peut quand même faire des trucs rigolos :
`awk '{ system("echo " $0 " | tr b _") }' <<< 'abcde'` donne « a_cde ».
Notez que cette fois-ci j'ai utilisé `echo` comme un gros nigaud plutôt que `<<<` car `system` appelle `sh`, qui ne semble pas connaître cette notation. Ça doit se changer d'une manière ou d'une autre, je suppose.

Si vous avez faim de trucs sales de ce genre, le manuel a quelques exemples. Il existe même un *pipe* (`|`) en AWK, qui peut balancer la sortie d'une fonction à une commande Bash qui peut être désignée par une chaîne de caractères. . .