L'outil `awk`

Guy Tremblay
Professeur

Département d'informatique
UQAM
http://www.labunix.uqam.ca/~tremblay_gu

INF600A
11–18 septembre 2018

Contenu

1 Introduction : Qu'est-ce que `awk` ?

2 Principe de fonctionnement

3 Exemples
- Exemples avec le délimiteur de champs par défaut = l'espace
- Exemples avec un délimiteur de champs explicite
- Exemples avec des fonctionnalités plus avancées

4 Spécification du script et guillemets

1. Introduction : Qu'est-ce que `awk` ?

Qu'est-ce que `awk` ?

`awk` — *dont le nom vient des trois créateurs, Alfred* A *ho, Peter* W *einberger et Brian* K *ernighan — est un langage de traitement de lignes, disponible sur la plupart des systèmes Unix et sous Windows avec Cygwin ou Gawk.*

Il est principalement utilisé pour la manipulation de fichiers textuels pour des opérations de recherches, de remplacement et de transformations complexes.

Source: `https://fr.wikipedia.org/wiki/Awk`

Qu'est-ce que `awk` ?

- Créé en 1977 — peu de temps après `sed` (1974)
- A popularisé l'utilisation des expressions régulières — reprises ensuite par Perl (1987)
- Possède un (très !) grand nombre de fonctionnalités — variables, tableaux, structures de contrôle — qui le rendent Turing-complet…

Qu'est-ce que `awk` ?

- Créé en 1977 — peu de temps après `sed` (1974)
- A popularisé l'utilisation des expressions régulières — reprises ensuite par Perl (1987)
- Possède un (très !) grand nombre de fonctionnalités — variables, tableaux, structures de contrôle — qui le rendent Turing-complet...
  - Mais nous ne verrons pas toutes ces fonctionnalités, juste assez pour écrire des petits scripts — style «*one liners*», utilisables dans des pipelines.
  - Si on a besoin de structures plus complexes, alors c'est souvent préférable d'utiliser un langage de script de haut niveau !

2. Principe de fonctionnement

man awk (version simplifiée — zeta.labunix)

SYNOPSIS

```
awk [-Fc] 'prog' filename...
awk [-Fc] -f progfile filename...
```

DESCRIPTION

The awk utility scans each input filename for lines that match any of a set of patterns specified in prog. The prog string must be enclosed in single quotes (') to protect it from the shell.

For each pattern in prog there can be an associated action performed when a line of a filename matches the pattern.

The set of pattern-action statements can appear literally as prog or in a file specified with the -f progfile option. Input files are read in order ; if there are no files, the standard input is read.

Un script `awk` est un filtre, comme `sed`

Un script `awk` reçoit un flux de lignes en entrée et
émet un flux de lignes en sortie

Format du flux d'entrée (fichier texte) :

- Flux d'entrée
  = suite de lignes

- Enregistrement
  = suite de champs (*fields*) séparés par un délimiteur.
  = `$1 $2 $3 ...`

- Délimiteur
  = typiquement un unique caractère.

  Délimiteur par défaut = espace blanc.

Un script `awk` est un filtre, comme `sed`

Un script `awk` reçoit un flux de lignes en entrée et
émet un flux de lignes en sortie

Format du flux d'entrée (fichier texte) :

- Flux d'entrée
  = suite de lignes
  = suite d'enregistrements.
- Enregistrement
  = suite de champs (*fields*) séparés par un délimiteur.
  = `$1 $2 $3 ...`
- Délimiteur
  = typiquement un unique caractère.

  Délimiteur par défaut = espace blanc.

Un script `awk` est un filtre, comme `sed`

Un script `awk` reçoit un flux de lignes en entrée et émet un flux de lignes en sortie

Soit «•» le caractère utilisé comme délimiteur.

Un fichier avec des enregistrements de quatre (4) champs aurait l'allure suivante — $champ_{i\,j}$ = i^e champ de la j^e ligne :

$champ_{11}$•$champ_{21}$•$champ_{31}$•$champ_{41}$
$champ_{12}$•$champ_{22}$•$champ_{32}$•$champ_{42}$
.
.
.
$champ_{1k}$•$champ_{2k}$•$champ_{3k}$•$champ_{4k}$

Notes :

- $champ_{ij}$ ne contient pas le caractère «•» !
- Les champs sont (typiquement) de longueur variable.

Un script `awk` est un filtre, comme `sed`

Un script `awk` reçoit un flux de lignes en entrée et émet un flux de lignes en sortie

$champ_{11}$•$champ_{21}$•$champ_{31}$•$champ_{41}$\n
$champ_{12}$•$champ_{22}$•$champ_{32}$•$champ_{42}$\n
.
.
.
$champ_{1k}$•$champ_{2k}$•$champ_{3k}$•$champ_{4k}$\n

```
Ligne 1
Ligne 2
.
.
.
Ligne k
```

NR = 1 : `$1` = $champ_{11}$; `$2` = $champ_{21}$; `$3` = $champ_{31}$; `$4` = $champ_{41}$

NR = 2 : `$1` = $champ_{12}$; `$2` = $champ_{22}$; `$3` = $champ_{32}$; `$4` = $champ_{42}$

...

NR = k : `$1` = $champ_{1k}$; `$2` = $champ_{2k}$; `$3` = $champ_{3k}$; `$4` = $champ_{4k}$

Note : $0 = ligne dans son ensemble (avant découpage) !

Structure d'un script `awk`

- Script `awk` = suite d'instructions
- Instruction = `garde` + `action`
- Effet du script : On exécute l'`action` sur chaque ligne pour laquelle la `garde` est vraie

Trois formes possibles

```
garde { action }    # Exécute action si garde vraie

garde               # Imprime enreg. si garde vraie

{ action }          # Exécute action sur chaque enreg.
```

Notes :

- Au moins un des deux éléments doit être présent !
- Typiquement, la *garde* est une expression de *pattern matching* (avec expr. rég. étendue), mais peut aussi être `BEGIN`, `END` ou une expr. bool.

3. Exemples

3.1 Exemples avec le délimiteur de champs par défaut = l'espace

Le fichier utilisé dans les exemples

À moins d'indication contraire. . .

```
$ cat profs.txt
Tremblay Guy Informatique tremblay.guy@uqam.ca PK-4435
Mili Hafedh  Informatique mili.hafedh@uqam.ca PK-4340
Laforest Louise Informatique laforest.louise@uqam.ca PK-4165
Trudel Sylvie Informatique trudel.s@uqam.ca PK-4720
```

Le script utilisé pour évaluer les exemples

À moins d'indication contraire. . .

Le script bash qui définit et appelle la fonction `exemple`, laquelle évalue un script `awk` sur `profs.txt`

```
$ cat ex-awk.sh
#!/bin/bash -

exemple()
{
    script="$1"

    echo "# script = '$script' #"
    cat profs.txt | awk "$script"
    echo ""
}
#########################################################
exemple '{ print }'
exemple '1 { print }'
exemple '1'
exemple '/Tremblay/ { print }'
...
```

La garde par défaut = 1

Condition style C : 1 = *true*


```
# Juste une action => condition (garde) toujours vraie.
# script = '{ print }' #
Tremblay Guy Informatique tremblay.guy@uqam.ca PK-4435
Mili Hafedh Informatique mili.hafedh@uqam.ca PK-4340
Laforest Louise Informatique laforest.louise@uqam.ca PK-4165
Trudel Sylvie Informatique trudel.s@uqam.ca PK-4720


# Condition: 1 = True
# script = '1 { print }' #
Tremblay Guy Informatique tremblay.guy@uqam.ca PK-4435
Mili Hafedh Informatique mili.hafedh@uqam.ca PK-4340
Laforest Louise Informatique laforest.louise@uqam.ca PK-4165
Trudel Sylvie Informatique trudel.s@uqam.ca PK-4720
```

L'action par défaut = `print`, qui émet l'enregistrement complet

(enregistrement possiblement modifié)

```
# Action par defaut = print.
# script = '1' #
Tremblay Guy Informatique tremblay.guy@uqam.ca PK-4435
Mili Hafedh Informatique mili.hafedh@uqam.ca PK-4340
Laforest Louise Informatique laforest.louise@uqam.ca PK-4165
Trudel Sylvie Informatique trudel.s@uqam.ca PK-4720


# script = '/Tremblay/ { print }' #
Tremblay Guy Informatique tremblay.guy@uqam.ca PK-4435


# script = '/Tremblay/' #
Tremblay Guy Informatique tremblay.guy@uqam.ca PK-4435

# Sauf s'il n'y ni garde, ni action...
# script = '' #
```

Le séparateur de champ de sortie est utilisé par `print` si des items sont séparés par «,»

Séparateur de champ de sortie par défaut = espace

```
# script = '{ print $2, $1, ":", $4 }' #
Guy Tremblay : tremblay.guy@uqam.ca
Hafedh Mili : mili.hafedh@uqam.ca
Louise Laforest : laforest.louise@uqam.ca
Sylvie Trudel : trudel.s@uqam.ca


# Le separateur de champ de sortie est omis si ',' absente
# script = '{ print $2 "-" $1, ": " $5 }' #
Guy-Tremblay : PK-4435
Hafedh-Mili : PK-4340
Louise-Laforest : PK-4165
Sylvie-Trudel : PK-4720
```

Note : La juxtaposition de chaines concatène les chaines, donc :

`ch1 ch2` (awk) ≈ `ch1 + ch2` (Java)

Les gardes BEGIN/END permettent d'exécuter une action au début/à la fin du script


```
# BEGIN = Condition vraie avant la premiere ligne.
# END = Condition vraie apres la derniere ligne.

# script = 'BEGIN { print "**** LISTE DES PROFS ****" }\
                  { print $2, $1, ":", $5 }\
            END   { print "**************************" }' #
**** LISTE DES PROFS ****
Guy Tremblay : PK-4435
Hafedh Mili : PK-4340
Louise Laforest : PK-4165
Sylvie Trudel : PK-4720
**************************
```

Les actions sont exécutées une après l'autre, sur la ligne telle que modifiée par les actions précédentes


```
# script = '{ $1 = "|"; $2 = ""; $4 = "" }\
            { print }\
            { print "---" }' #
|  Informatique  PK-4435
---
|  Informatique  PK-4340
---
|  Informatique  PK-4165
---
|  Informatique  PK-4720
---
```

C'est OFS, le séparateur des champs en sortie, qui est utilisé lorsque des «,» séparent les items

OFS = *Output field separator*

```
# OFS = Separe les champs en sortie.
# script = 'BEGIN{ OFS="|" } { print $2, $1, $5 }' #
Guy|Tremblay|PK-4435
Hafedh|Mili|PK-4340
Louise|Laforest|PK-4165
Sylvie|Trudel|PK-4720


# Mais OFS pas utilise si ',' absente.
# script = '{ print $2 "|" $1 "::" $5 }' #
Guy|Tremblay::PK-4435
Hafedh|Mili::PK-4340
Louise|Laforest::PK-4165
Sylvie|Trudel::PK-4720
```

Note : Valeur par défaut de OFS = espace

ORS, le séparateur des enregistrements en sortie, est émis après chaque enregistrement

ORS = *Output record separator*

```
# ORS = Separe les enregistrement en sortie.
# script = 'BEGIN {ORS="\n---\n"} {print $2, $1, $5}' #
Guy Tremblay PK-4435
---
Hafedh Mili PK-4340
---
Louise Laforest PK-4165
---
Sylvie Trudel PK-4720
---
```

Note : Valeur par défaut de `ORS` = `\n`

3.2 Exemples avec un délimiteur de champs explicite

FS représente le délimiteur des champs en entrée

FS = *Field separator*

```
# script = 'BEGIN{ FS="@"; OFS=" | " }\
                 { print $2, $1 }' #
uqam.ca PK-4435 | Tremblay Guy Informatique tremblay.guy
uqam.ca PK-4340 | Mili Hafedh Informatique mili.hafedh
uqam.ca PK-4165 | Laforest Louise Informatique laforest.louise
uqam.ca PK-4720 | Trudel Sylvie Informatique trudel.s


# script = 'BEGIN{ FS="."; OFS="; " }\
                 { print $1, $2, $3 }' #
Tremblay Guy Informatique tremblay ; guy@uqam ; ca PK-4435
Mili Hafedh Informatique mili ; hafedh@uqam ; ca PK-4340
Laforest Louise Informatique laforest ; louise@uqam ; ca PK-41
Trudel Sylvie Informatique trudel ; s@uqam ; ca PK-4720
```

Note : Valeur par défaut de FS = espace

Le comportement pour plusieurs délimiteurs consécutifs dépend de la valeur de `FS` (espace ou non)


```
# Deux delimiteurs non espace colles => champ vide.
$ echo "a,b,,,c,d" | awk -F, 'BEGIN{ OFS = "|" }\
                                { $1 = $1; print }'
a|b|||c|d


# Deux espaces ou plus ≡ un seul delimiteur!
$ echo " a b   c   d   " | awk 'BEGIN{ OFS = "|" }\
                                { $1 = $1; print }'
a|b|c|d
```

Note : Le délimiteur de champ peut être spécifié avec l'option «`-F`» ou avec la variable `FS`, donc les deux appels suivants sont équivalents :

```
awk -F, '...'
awk 'BEGIN{ FS = "," } ...'
```

Il faut au moins une affectation pour que `OFS` ait un effet sur `print` sans argument (sans «,»)

Transformation d'un fichier avec blancs vers CSV

```
# script = 'BEGIN{ OFS="," } { print }' #
Tremblay Guy Informatique tremblay.guy@uqam.ca PK-4435
Mili Hafedh Informatique mili.hafedh@uqam.ca PK-4340
Laforest Louise Informatique laforest.louise@uqam.ca PK-4165
Trudel Sylvie Informatique trudel.s@uqam.ca PK-4720


# script = 'BEGIN{ OFS="," } { $1 = $1; print }' #
Tremblay,Guy,Informatique,tremblay.guy@uqam.ca,PK-4435
Mili,Hafedh,Informatique,mili.hafedh@uqam.ca,PK-4340
Laforest,Louise,Informatique,laforest.louise@uqam.ca,PK-4165
Trudel,Sylvie,Informatique,trudel.s@uqam.ca,PK-4720
```

3.3 Exemples avec des fonctionnalités plus avancées

Analyse de $0 avec `length`

La variable $0 dénote la ligne complète, sans le saut de ligne


```
# Selection des lignes avec au moins 4 caracteres
$ cat fich.txt
abcde
abcd
abc
ab
a

$ awk 'length($0) >= 4' fich.txt
abcde
abcd
```

Utilisation de la variable NR

NR = *Number of records*

Script pour ajouter un numéro au début de chaque ligne

```
$ cat fich.txt
2 20
4 40
12 222
99 999


$ cat fich.txt | awk '{print NR ": " $0}'
1: 2 20
2: 4 40
3: 12 222
4: 99 999
```

Utilisation de la variable NR

NR = *Number of records*

Script pour imprimer les lignes paires

```
$ cat fich.txt
2 20
4 40
12 222
99 999


$ cat fich.txt | awk 'NR % 2 == 0'
4 40
99 999
```

Calculs et utilisation de la variable NR

NR = *Number of records*

Script pour calculer la moyenne du premier champ de chaque enregistrement

```
$ cat fich.txt
2 20
4 40
5 50
10 100

$ cat fich.txt |
  awk '     { s += $1 }
       END { print "Moyenne =", s / NR }'
Moyenne = 5.25
```

Note : En `awk` (ainsi que dans plusieurs autres langages de script), les variables ont une valeur initiale par défaut, ici, la chaine vide... convertie en `0` par le «+» !

Note : Si le champ `$1` n'est pas un nombre, sa conversion pour utilisation avec «+» produira la valeur `0`.

Utilisation de la variable NF

NF = *Number of fields*

Script pour imprimer les lignes qui ont trois champs ou plus

```
$ cat fich.txt
2 20 4 40
12 222
99
22 1212 5
18 188


$ cat fich.txt | awk 'NF >= 3'
2 20 4 40
22 1212 5
```

Sélection d'un intervalle de lignes

A range pattern is made of two patterns separated by a comma, in the form '`begpat, endpat`*'. It is used to match ranges of consecutive input records. The first pattern,* `begpat`*, controls where the range begins, while* `endpat` *controls where the pattern ends.*

[. . .]

A range pattern starts out by matching `begpat` *against every input record. When a record matches* `begpat`*, the range pattern is turned on, and the range pattern matches this record as well. As long as the range pattern stays turned on, it automatically matches every input record read. The range pattern also matches* `endpat` *against every input record ; when this succeeds, the range pattern is turned off again for the following record. Then the range pattern goes back to checking* `begpat` *against each record.*

Source: *https://www.gnu.org/software/gawk/manual/html_node/Ranges.html*

Sélection d'un intervalle de lignes


```
$ cat fich.txt
2 20
4 40
5 50
18 188
```

Note : Une fois le 1^{er} motif matché, termine dès que le 2^{e} motif est matché, **y compris sur la même ligne. . . ,** mais l'action s'exécute quand même sur cette ligne !

```
$ cat fich.txt | awk /2/,/4/
2 20
4 40

$ cat fich.txt |
  awk '/50/,/5/ { print $1 }'
5

$ cat fich.txt | awk /5/,/9/
5 50
18 188

$ cat fich.txt | awk /9/,/2/

$
```

Sélection d'un intervalle de lignes


```
$ cat fich.txt
2 20
4 40
5 50
18 188
```

```
$ cat fich.txt |
  awk '/0/,/0/ { print $1 }'
2
4
5
```

Note : Une fois le 1^{er} motif matché, termine dès que le 2^{e} motif est matché, y compris sur la même ligne... et cela se peut se faire pour plusieurs intervalles, i.e., tous ceux qui suivent.

Utilisation des «*associative arrays*»

`awk` est un des premiers (le premier ?) langages à avoir introduit de tels tableaux associatifs = *hashes*, *maps* !

```
$ cat script.awk
{ h[$2] += 1 }

END {
  for (key in h) {
    print key ": " h[key]
  }
}
```

```
$ cat foo.txt
g     b200
l     x000
ab    a100
def   b200
pqr   b200
mno   a100
```

```
$ awk -f script.awk foo.txt
??
```

Utilisation des «*associative arrays*»

`awk` est un des premiers (le premier ?) langages à avoir introduit de tels tableaux associatifs = *hashes*, *maps* !

```
$ cat script.awk
{ h[$2] += 1 }

END {
  for (key in h) {
    print key ": " h[key]
  }
}
```

```
$ cat foo.txt
g    b200
l    x000
ab   a100
def  b200
pqr  b200
mno  a100
```

```
$ awk -f script.awk foo.txt
a100: 2
b200: 3
x000: 1
```

Le *pattern-matching* peut aussi se faire sur des champs plutôt que sur la ligne dans son ensemble

Pour ce faire, on utilise l'opérateur « ~ »

```
$ cat foo.txt
ab   100a
def  100a
g    b200
l    x30x
mno  100a
pqr  b200

$ awk '$1 ~ /^.$/' foo.txt
g    b200
l    x30x

$ awk '$1 ~ /^.{3}$/' foo.txt
def  100a
mno  100a
pqr  b200

$ awk '$0 ~ /b.*1/' foo.txt       # = awk '/b.*1/' foo.txt
ab   100a
```

4. Spécification du script et guillemets

À l'appel d'une commande au niveau du *shell*, des blancs *non escapés* indiquent un nouvel argument


```
$ cat cmd.sh
#!/bin/bash -
cat <<EOF
 '$1'; '$2'; '$3'
EOF


$ ./cmd.sh 10   20    30  40
 '10'; '20'; '30'

$ ./cmd.sh 10  20\ 30  40
 '10'; '20 30'; '40'
```

Soit un fichier `profs.txt` contenant trois enregistrements, chacun avec trois champs

Format = Nom Prenom Departement

```
$ cat profs.txt
Tremblay Guy Informatique
Mili Hafedh Informatique
Laforest Louise Informatique
```

Tâche = Imprimer les enregistrements qui contiennent `Informatique`

Les guillemets pour le script ne sont pas nécessaires, mais évitent d'avoir à utiliser *escape*

```
$ awk '/Informatique/ { print }' profs.txt
Tremblay Guy Informatique
Mili Hafedh Informatique
Laforest Louise Informatique

$ awk /Informatique/ { print } profs.txt
??
```

Tâche = Imprimer les enregistrements qui contiennent `Informatique`

Les guillemets pour le script ne sont pas nécessaires, mais évitent d'avoir à utiliser *escape*

```
$ awk '/Informatique/ { print }' profs.txt
Tremblay Guy Informatique
Mili Hafedh Informatique
Laforest Louise Informatique

$ awk /Informatique/ { print } profs.txt
awk: can't open file {
source line number 1
```

Tâche = Imprimer les enregistrements qui contiennent `Informatique`

Les guillemets pour le script ne sont pas nécessaires, mais évitent d'avoir à utiliser *escape*

```
$ awk '/Informatique/ { print }' profs.txt
Tremblay Guy Informatique
Mili Hafedh Informatique
Laforest Louise Informatique

$ awk /Informatique/ { print } profs.txt
awk: can't open file {
source line number 1

$ awk /Informatique/\ {\ print\ } profs.txt
Tremblay Guy Informatique
Mili Hafedh Informatique
Laforest Louise Informatique
```

Tâche = Imprimer le champ nom des enregistrements contenant une chaine indiquée en argument

Par exemple, la chaine `Informatique`

Pour généraliser, on veut définir une fonction

```
# $1: 1er argument de la fonction.
$ profs_dept(){ awk '/$1/ { print $1 }' profs.txt; }

$ profs_dept Informatique
??
$
```

??

Tâche = Imprimer le champ nom des enregistrements contenant une chaine indiquée en argument

Par exemple, la chaine `Informatique`

Pour généraliser, on veut définir une fonction

```
# $1: 1er argument de la fonction.
$ profs_dept(){ awk '/$1/ { print $1 }' profs.txt; }

$ profs_dept Informatique

$
```

Pcq. aucun enregistrement ne matche le motif «`$1`» !

Tâche = Imprimer le champ nom des enregistrements contenant une chaine indiquée en argument

Par exemple, la chaine `Informatique`

Pour généraliser, on veut définir une fonction

```
# $1: 1er argument de la fonction.
$ profs_dept(){ awk "/$1/ { print $1 }" profs.txt; }

$ profs_dept Informatique
??

$
```

Tâche = Imprimer le champ nom des enregistrements contenant une chaine indiquée en argument

Par exemple, la chaine `Informatique`

Pour généraliser, on veut définir une fonction

```
# $1: 1er argument de la fonction.
$ profs_dept(){ awk "/$1/ { print $1 }" profs.txt; }

$ profs_dept Informatique




$
```

Le motif est correct = `Informatique`

Par contre, le script = `{ print Informatique }`.
Or : `Informatique` = variable non définie $\Rightarrow$ valeur initiale par défaut = chaine vide (= 0 si requis comme valeur numérique) !

Tâche = Imprimer le champ nom des enregistrements contenant une chaine indiquée en argument

Par exemple, la chaine `Informatique`

Pour généraliser, on veut définir une fonction

```
# $1: 1er argument de la fonction.
$ profs_dept(){ awk "/$1/ { print \$1 }" profs.txt; }

$ profs_dept Informatique
Tremblay
Mili
Laforest

$
```

Pour en savoir plus
Quelques références

- D. Dougherty.
  sed & awk.
  O'Reilly, 1990.
- A. Robbins and N.H.F. Beebe.
  Classic Shell Scripting.
  O'Reilly, 2009.

- `man awk`