Le langage awk

Awk est un filtre qui facilite le traîtement de texte. awk est programmable et contient des constructions pour des instructions conditionnelles, des boucles et des variables, dans une syntaxe similaire au langage C. La commande

awk programme fichier$_1$ fichier$_2$ ...

execute les instructions awk dans la chaîne programme sur l'entrée provenant des fichiers donnés en argument. La sortie de awk est - généralement - dirigée vers la sortie standard. awk cherche dans chaque ligne d'entrée un sélecteur et s'il s'applique, l'action associée est executée. La forme générale d'un programme awk est :

BEGIN{ **instructions initiales**
}
sélecteur { **action** }
...
END{ **instructions finales**
}

Il peut y avoir autant de couples sélecteur/action qu'on veut. Après chaque lecture d'une ligne d'entrée, chacun des sélecteurs est évalués (dans l'ordre de leur écriture). Si le sélecteur filtre est vrai, l'action associée est exécutée. S'il n'y a pas de sélecteur, l'action est exécutée pour chaque ligne d'entrée. S'il n'y a pas d'action associée à un sélecteur, la ligne d'entrée est copiée vers la sortie.

Une action peut consister de plusieurs instructions séparées par un ;. BEGIN et END indiquent des actions qui ne sont exécutées qu'au début et à la fin du fichier. awk lit chaque ligne d'entrée et la divise en des champs, normalement séparés par un séparateur. Dans une expression awk, $0 représente la ligne d'entrée dans sa totalité et $1, $2, ... représentent le premier, deuxième, etc, champ de cette ligne. Le nombre de champs de la ligne traîtée est disponible dans la variable NF et le nombre de lignes lues est disponible dans la variable NR. Le séparateur de champs est accessible à tout instant dans la variable FS, et le séparateur de lignes dans RS. Le séparateur par défaut est le caractère espace ou tabulation, et la valeur par défaut de la variable RS est le caractère newline. Si l'on veut traiter un fichier ou les champs sont sépares par :, comme le fichier /etc/passwd, l'instruction

FS = ":"

doit être dans la section BEGIN. Pour revenir vers le défaut, il suffit d'affecter la chaîne vide à la variable FS. Le nom du fichier actuellement traîté se trouve dans la variable FILENAME.

Ci dessous la liste complète des variables awk prédéfinies :

variable	description	valeur par défaut
ARGC	nombre d'arguments de la commande	
ARGV	tableau des arguments de la commande	
FILENAME	nom du fichier d'entrée actuel	
FNR	nombre d'enregistrements dans le fichier actuel	
FS	séparateur de champs à la lecture	" "
NF	nombre de champs dans l'enregistrement actuel	
NR	nombre d'enregistrements lu	
OFMT	format d'impression de nombres	"%.6g"
OFS	séparateur de champs à l'impression	" "
ORS	séparateur d'enregistrements à l'impression	"\n"
RLENGTH	longueur de la chaîne filtrée par la fonction `match`	
RS	séparateur d'enregistrement à la lecture	"\n"
RSTART	début de chaîne filtrée par la fonction `match`	
SUBSEP	séparateur de sousscripts	"\034" (")

Tableau 1: variables prédéfinies

Ci-dessous quelques exemples simples d'utilisation de awk :

```
awk '/srb/'
```
imprime toutes les lignes du fichier d'entrée contenant la chaîne srb

```
awk 'END{print NR}'
```
imprime le nombre de lignes du ficher d'entrée

```
awk '{print $3}'
```
imprime le troisième champ de chaque ligne.

et encore un ensemble de commande awk util :

```
NR == 10
```
imprime la dixième ligne de l'entrée

```
{champ = $NF}
END { print champ}
```
imprime le dernier champ de la dernière ligne

```
NF > 4
```
imprime toute ligne à plus de 4 champs

```
$NF > 4
```
imprime toute ligne dans le dernier champ est supérieur à quatre

```
{nf = nf + NF}
END { print nf}
```
imprime le nombre de champs de toutes les lignes d'entrées

```
/Jean/ {nlignes + = 1}
END { print nlignes}
```
imprime le nombre de lignes contenant le mot 'Jean'

```
$1 > max {max = $1; maxligne = $0}
END {print max, maxligne}
```
imprime le plus grand premier champ et la ligne qui le contient

```
lenght($0) > 80
```
imprime toute ligne plus longue que 80 caractères

```
{temp = $1; $1 = $2; $2 = temp; print}
```
imprime chaque ligne en échangeant le premier et le deuxième champ

```
{print NR, $0}
```
imprime chaque ligne précédée par le numéro de ligne

```
{for (i = NF; i > 0; i = i - 1)
 printf("%s ", $1)
 printf("\n")
}
```
imprime les champs de chaque ligne en ordre inverse

Une expression awk peut prendre des valeurs numériques ou des valeurs sous forme de chaînes de caractères. Les opérateurs +, -, *, /, %, ^ et l'opérateur de concatenation (indiqué par le caractère espace), ainsi que les opérateurs ^, + +, - -, + =, - =, *=, /=, ^ = et %= sont disponibles. Les variables peuvent être des scalaires, des éléments de tableaux (notés x[i]) ou des noms de champs tels que $1 ou $2. Les variables sont initialisées à la chaîne vide. Les opérateurs relationnels sont les mêmes que ceux du langage C (<, < =, = =, ! =, > =, >), et ils peuvent être appliqués aussi bien aux nombres qu'aux chaînes de caractères. Par exemple :

```
NF > 5
```

est vraie si les enregistrements ont plus de 5 champs.

```
$1 > "s"
```

est vraie si le premier champs de la ligne analysée est lexicographiquement supérieur à "s", tel que la chaîne "st", par exemple.

```
$2 >= 0
```

est vraie si le deuxième champs est numériquement positif.

Il existe également des opérateurs de filtrage explicite. Ce sont les opérateurs "~" et "!~". Ainsi, l'expression

```
$1 ~ /srb|SRB/
```

n'est vraie que pour les enregistrements dont le premier champs est égal à srb ou SRB, et l'expression

```
$1 ~/[rR]ue/
```

est vraie si $1 est égale à rue ou Rue.

Comme en ed, la portée d'une action peut être limitée par des adresses séparées par l'opérateur ",". Par exemple :

```
/début/,/fin/{ ... }
```

applique les opérations entre accolades à la section débutant par la ligne contenant le mot début et se terminant par la ligne contenant le mot fin. Le filtre :

```
NR==10, NR==20 { ... }
```

applique les opérations entre accolades aux lignes 10 à 20 de l'entrée.

Le tableau ci-dessous donne la syntaxe des expressions régulières admises en awk :

expression	description
c	le caractère normal c
\c	séquence d'échappement (\b, \f, \n, \r, \t, \ddd) ou caractère litéral c
^	début de chaîne
$	fin de chaîne
.	un caractère quelconque

[$c_1c_2...c_n$]	tout caractère dans c_1, c_2, ..., c_n
[^ $c_1c_2...c_n$]	tout caractère différent de c_1, c_2, ..., c_n
[c_1-c_2]	tout caractère dans l'interval c_1 à c_2
[^ c_1-c_2]	tout caractère extérieur à l'interval c_1 à c_2
r_1 | r_2	toute chaîne filtrée soit par r_1 soit par r_2
(r_1)(r_2)	toute chaîne xy, où r_1 filtre x et r_2 filtre y les parenthèses ne sont nécessaires qu'autour des arguments avec alternations
(r)*	zéro ou plusieurs chaînes consécutives filtrées par r
(r) +	une ou plusieurs chaînes consécutives filtrées par r
(r)?	zéro ou une chaîne filtrée par r
(r)	toute chaîne filtrée par r

Une action de awk se compose d'une ou de plusieurs instructions séparées par des point-virgules ou des line feed. Une instruction peut être une affectation, une instruction conditionnelle, une boucle ou l'appel d'une fonction standard telle que print. En générale, la syntaxe de awk est la même que celle du langage C.

Ci-dessous la liste des instructions disponibles :

```
if ( condition ) instruction [ else instruction ]
while ( condition ) instruction
for (expression ; condition ; expression ) instruction
for ( variable in tableau ) instruction
break
continue
{ [ instruction ] ... }
variable = expression
print [ liste-d'expressions ] [ >expression ]
next # ignore les selecteurs restants de cette ligne d'entrée
exit #ignore le reste de l'entrée et continue à l'action END
do instruction while (condition)
```

Les instructions se terminent par des point-virgules, des line-feeds ou des accolades droites. Une liste-d'expressions vide represente la ligne d'entrée entière.

Par exemple :

```
if ($2 > $1) {
        x = $1
        $1 = $2
        $2 = x
}
```

copie l'entrée standard vers la sortie standard en échangeant le premier et le deuxième champ si le deuxième champs est supérieur au premier.

De même qu'en C, les instructions print et printf permettent des arguments pour le formattage des impressions. Par exemple :

```
i = 1
while (i < = NF) {
        printf "%s, ", $i
       i++
}
```

imprime chaque champs suivi d'une virgule. L'instruction print sans arguments imprime l'enregistrement courant en entier et, avec des arguments, print peut être utilisé pour imprimer des champs sélectionnés, comme dans :

```
print $2, $1
```

qui imprime d'abord le deuxième champs, ensuite le premier, et ensuite un line-feed. Voici un exemple supplémentaire d'utilisation de printf :

```
printf "%s, %s\n", $1, $2
```

qui imprime les deux premiers champs de l'enregistrement, séparés par une virgule.

Ci dessous le tableau listant l'ensemble des caractères de contrôle du format d'impression :

caractère	imprime l'argument comme
c	ASCII caractère
d	entier décimal
e	[-]d.dddddE[+-]dd
f	[-]ddd.dddddd
g	conversion e ou f : on choisit celle qui est plus courte, le zéros non-significatifs sont éliminés
o	nombre octal non-signé
s	chaîne de caractères
x	nombre hexadécimal non-signé
%	s'imprime comme le signe % sans user un argument

et ci-dessous quelques exemples d'utilisation :

fmt	$1	printf(fmt, $1)
%c	97	a
%d	97.5	97
%5d	97.5	97
%e	97.5	9.750000e+01
%f	97.5	97.500000
%7.2f	97.5	97.50
%g	97.5	97.5
%.6g	97.5	97.5
%o	97	141
%06o	97	000141
%x	97	61
|%s|	Janvier	|Janvier|
|%10s|	Janvier	| Janvier|
|%-10s|	Janvier	|Janvier |
|%.3s|	Janvier	|Jan|
|%10.3s|	Janvier	| Jan|
|%-10.3s|	Janvier	|Jan |
%%	Janvier	%

Les tableaux awk peuvent être indexés par des chaînes de caractères, ce qui permet de les utiliser comme des mémoires associatives. Par exemple, le premier champs du fichier /etc/passwd contient les noms de login des utilisateurs. Le programme suivant vérifie que chaque nom de login n'y a qu'une seule occurrence :

```
awk '
BEGIN{ FS=":"
}
{ if (user[$1]) {
      print $1, "dupliqué"
   }
  user[$1] = NR
}' </etc/passwd
```

Tous les éléments d'un tableau associatif sont accessibles par l'instruction for. Par exemple :

```
for (i in user)
 instruction
```

accède à toutes les indices du tableau user.

Les seules fonctions accessibles au programmeur de awk sont les fonctions standard, c'est-à-dire : celles qui font partie de la définition même de awk. La liste de ces fonctions se trouve ci-dessous :

sqrt, log, exp, int, cos, sin, rand, srand
: livrent, respectivement, la racine carrée, le logarithme en base e, l'exponentielle, la partie entière, le cosinus et le sinus de l'argument entre parenthèses. rand() livre un nombre aléatoire entre 0 et 1 et srand(x) donne une nouvelle initialisation au générateur de nombres aléatoires utilisé par rand.

length
: livre la longueur de la chaîne de caractères donnée en argument. Sans argument, length donne la longueur de l'enregistrement courant.

substr(s, m, n)
: livre la sous-chaîne de la chaîne s qui commence au $m^{ème}$ caractère et qui contient au plus n caractères.

index(s, t)
: livre la position de la première occurrence de la chaîne t à l'intérieure de la chaîne s. Le compte commence à partir de la position 1. Si t n'est pas une sous-chaîne de s, index retourne la valeur zero.

split(s, a, fs)
: met s dans le tableau a, à raison d'une entrée par champ, en utilisant fs comme séparateur de champ. Le défaut pour fs est FS.

substr(s, p, n)
: retourne le suffix de s qui commence à la position p et qui est de longuer n. exemple : si x = "Canada", alors substr(x, 1, 3) retourne Can

Avant de donner une suite de programmes awk exemples, un liste exhaustive des opérateurs awk, chacun avec un exemple d'application commenté :

opération	opérateurs	example	commentaire

affectation	= += -= *= /= %= ^=	x *= 2	x = x * 2
conditionnelle	?:	x?y:z	si x est vrai alors y sinon z
OU logique	||	x || y	1 si x ou y sont vrai, sinon 0
ET logique	&&	x && y	1 si x et y sont vrai, sinon 0
appartenance à tableau	in	i in a	1 si a[i] existe, sinon 0
filtrage	~ !~	$1 ~ /x/	1 si le premier champ contient un x, sinon 0
relationnelle	< <= == != >= >	x == y	1 si x est égal à y, sinon 0
concatenation		"a" "bc"	"abc"; il n'y a pas d'opérateur de concatenation explicite
addition, soustraction	+ -	x + y	somme de x et y
multiplication, division, modulus	* / %	x % y	reste de la division entière de x par y
plus et moins unaire	+ -	-x	0 - x
NON logique	!	!$1	1 si $1 est zéro ou la chaîne vide, sinon 0
exponentiation	^	x ^ y	x^y
incrementation, décrementation	++ —	++x, x++	ajoute 1 à x
champ	$	$i+1	valeur du $i^{ème}$ champ plus 1
groupement	( )	($i)++	ajoute 1 à la valeur du $i^{ème}$ champ

Ci-dessous alors les quelques petit programmes awk utiles :

```
# un programme qui compte de mots d'un texte
{ gsub( /[\[.,;;!?(){}\]]/, " ")
  for (i = 1; i < = NF; i+ + )
       count[$i]+ +
}
END { for (w in count)
       print count[w], w
}
```

```
# un programme qui genere ARGV[1] nombres aleatoire
BEGIN {
       for (i = 1; i < = ARGV[1]; i+ + )
          print int (101 * rand())
}
```

```
# un programme de tri par insertion
{ ligne[NR] = $0 }
END{
       isort( ligne, NR )
       for(i = 1 ; i < = NR ; i+ + ) print ligne[i]
}
function isort( A, n,i, j, hold)
{
       for( i = 2 ; i < = n ; i+ + )
       {
              hold = A[j = i]
              while ( A[j-1] > hold)
                 { j— ; A[j+1] = A[j] }
              A[j] = hold
       }
}
```

```
# un programme qui tri par des heap
{ a[NR] = $0 }
END{   hsort( a, NR );
       for (i = 1; i < = NR; i+ + )
          { print a[i] }
   }

function hsort (a, n, i)
 {
```

```
        for (i = int(n/2); i >= 1; i—)
                { heapify(a, i, n) }
        for (i = n; i > 1; i—) {
                { swap(a, 1, i) }
                { heapify(a, 1, i-1) }
        }
}

function heapify(a, left, right,    p, c) {
        for (p = left; (c = 2 * p) <= right; p = c) {
                if (c < right && a[c+1] > a[c])
                    { c++ }
                if (a[p] < a[c])
                    { swap(a, c, p) }
        }
}

function swap(a, i, j,t) {
        t = a[i]; a[i] = a[j]; a[j] = t
}
```

```
# le fameux quicksort de C.A.R. Hoare
{ A[NR] = $0 }

END{ qsort( A, 1, NR)
        for (i = 1; i <= NR; i++)
        print A[i]
}
function qsort(A, left, right,i, last) {
        if (left >= right)
                return
        swap(A, left, left + int((right-left+1)* rand()))
        last = left
        for( i = left + 1; i <= right; i++)
                if (A[i] < A[left])
                    swap(A, ++last, i)
        swap(A, left, last)
        qsort(A, left, last-1)
        qsort(A, last+1, right)
}

function swap(A, i, j, t) {
        t = A[i]; A[i] = A[j]; A[j] = t}
```

Finalement le listing d'un programme awk qui permet la recherche de numéro de téléphones dans une base de donnée. Par une rédirection de la sortie vers un modem compatible Hays, l'appel au numéro se fait automatiquement.

```
# phfind.awk — fait des recherches dans une base de donnée
d'adresses et de numéros de téléphone
#
# format de fichier: des champs séparés par des virgules, les enre-
gistrements séparés par des \n.
#
#       $1: nom (notation quelconque sans virgule)
#       $2: numero de téléphone
#               (notation quelconque sans virgule)
#                e.g.:(111) 222-3333 ou
#                    (19)49-7154-74.72)
#       $3 et supérieur: format entièrement libre
# examples d'entrées legales :
#
# post emily, (19)(49) 07541-4257, spricht nur deutsch
# George, (111) 222 3333
# emily carr, , 2 rue de la Liberté, St-Denis
#
#

BEGIN \
        {
                FS = ","

                parseargs()
#                definie des chaînes de commandes pour
#                des modems
                dialprefix = "ATDP"
                dialsuffix = "\n"
        }

$a ~ s \
        {
                if (f != 0)# enregistrement en entier
                {
                    printf("%s\n",$0)
                }
                else if (d != 0)# appel modem
                {
                    printf("%s%s%s",
                   dialprefix,cleanphonenum($2),
                    dialsuffix)
                }
                else
```

```
            # defaut: nom et numero de téléphone
            {
             printf("%s's n° de tel est %s\n",
                 $1,$2)
            }

            if(++hits == n)# arret
               exit(0)
        }

# lecture et analyse de argv. i est locale.
function parseargs(i) \
{
        a = 1
        if (ARGC < 2)
        #       on a meme pas l'option -s
                usageandexit()

        for(i=ARGC-1; i>0; i—)
        {
                if (ARGV[i]=="-d")
                   d=1
                else if (ARGV[i]=="-f")
                   f=1
                else if (index(ARGV[i],"-s")==1)
                   s=substr(ARGV[i],3)
                else if (index(ARGV[i],"-n")==1)
                   n=substr(ARGV[i],3)
                else if (index(ARGV[i],"-a")==1
                   a=substr(ARGV[i],3)
                else
                {
                   printf("mauvaise option: %s\n",
                                ARGV[i])
                   usageandexit()
                }
        }
        # pour eviter de considerer des switch
#               comme des fichiers
        ARGC=1

        # n==1 est le defaut.
        # on force n==1 si l'on a -d.
        if ((d != 0) || (n == 0))
                n=1
}
```

```
# on enleve tout les carcateres non numerique pour l'appel.

function cleanphonenum(ph) \
{
        # enleve espaces, parentheses gauches et droites, tirets et points
        gsub("[ ().-]","",ph)
        return (ph)
}

# montre l'usage et sorte avec un status d'erreur
function usageandexit() \
{
        printf("utilisation:  awk phf {options} addrfile\n")
        printf("options: -sEXP  une expression  à filtrer\n")
        printf("      -nNUM  arrète après l'impression de n enregistrements
(defaut = = 1)\n")
        printf("      -d     imprime la commande pour l'appel modem\n")
        printf("      -f     imprime tout\n")
        printf("      -aNUM  indique sur quel champs il faut faire la recher-
che (0 = = l'enregistrement entier)\n")
        exit(1)
}
```