Année Spéciale Informatique
ENSIMAG

Grenoble 1992

LE LANGAGE AWK

Frédéric LACROIX
Dan Bog

Remerciements

Nous adressons ces pages d'encre noircies à notre maître Serge Rouveyrol, Grand Prêtre des Brumes Astrales et Xinurales.

Nous remercions la famille roumaine Bog et la famille BOZO composée de :

Gimo : trop petit pour sa gratte
Gillou : sans qui nous n'aurions jamais connu Arthur
Jackie : notre synthétiseuse de rêve
Nathalie dite Caro : ma chérie, je t'adore
Le père PNEC : il en bave à l'armée
Le ptose : fournisseur officiel pour le mois de juin
La mère Michel : pour avoir fait la lessive et la bouffe

Enfin, j'embrasse le vieux Nain, tous les grands Steaks et toute la famille au pays ...

Introduction

Ce livre est un hymne à l'Obscure, à cet état de l'Esprit élu par le fruit inconscient de notre Ame. C'est un art conçu pour faire naitre la passion. Aussi ces mots qui ne sont qu'assemblages et imbrications de lettres vous paraitront sans signification aucune mais ils sont en fait la nature fondamentale qui régit l'univers de la réalité et ces alinéas qui sont noyés dans ce brouillard de silhouettes noirs sont cette part de rêve sans quoi le monde ne pourrait exister.
C'est le langage AWK, étendue infinie de verts paturages issus d'un savoir-faire malthusien encore ignoré. Mais comme c'est étrange de se retrouver dans cet état oublié dont il est possible de rêver les nuits de somnolences inconscientes.

La première partie de ce livre explique les principes essentiels de AWK de façon simple et de telle sorte que l'utilisateur les assimile facilement.
La deuxième partie de ce livre détaille chaque point du langage AWK. On pourra s'y référer après avoir bien compris chacun des exemples de la partie 1.
La dernière partie de ce livre donne quelques applications réelles traitées par AWK.

Chapter 1

Les principes essentiels de AWK

1.1 Introduction et définitions

1.1.1 structure d'un fichier de données

Le langage de programmation AWK est très efficace dans la gestion de fichiers. En général, un fichier de données admet une structure cohérente qui peut se définir de la façon suivante :

champ 1	champ2	...	champ n	(ligne 1)
champ 1	champ2	...	champ n	(ligne 2)
.....	.....	.....	.....	(ligne i)
champ 1	champ2	...	champ n	(ligne q)

Les exceptions dépendent de la définition même du type de données et peuvent être traitées lorsqu'elles sont parfaitement connues.

L'exemple de fichier de données qui sera utilisé dans toute la partie 1 sera le suivant :

Fichier **fic.data** :

```
Nom       Nombre de    Marque de    Prix du
          cigarettes   cigarettes   paquet
          fumees par                de 20
            jour                   cigarettes

Serge       20         Toute        10.50
Fredo       15         Camel        10.50
Marc        10         Camel        10.50
Alain       10         Peter_Bleu   10.50
Agnes       20         Goldo_Leg     6.50
Isabelle    15         Royal_Ment   12.50
Pyr          0         Aucune          0
Thyll       10         Camel        10.50
LeSaint      0         Aucune          0
```

Quelques notions de champs et de séparateurs :

1 Dans notre exemple, chaque champ est séparé par un ou plusieurs blancs. Le séparateur peut être modifié comme on le veut par la variable globale **FS**. Pour l'instant, FS = " " .

2 AWK compte le nombre de champs d'une ligne courante; ce nombre est placé dans la variable globale **NF** (dans l'exemple, NF = 4).

3 AWK compte le nombre de lignes du fichier d'entrée; ce nombre est placé dans la variable globale **NR** (dans l'exemple, NR = 9).

Exemples : visualisation du fichier d'entrée

• imprime chaque ligne du fichier.

```
awk '{print}' fic.data   ou   awk '{print $0}' fic.data
Serge      20      Toute       10.50
Fredo      15      Camel       10.50
Marc       10      Camel       10.50
Alain      10      Peter_Bleu  10.50
Agnes      20      Goldo_Leg    6.50
Isabelle   15      Royal_Ment  12.50
Pyr         0      Aucune          0
Thyll      10      Camel       10.50
LeSaint     0      Aucune          0
```

• écrit chaque champ séparé d'un blanc (virgule), ligne par ligne.

```
awk '{print NR,$1,$2,$3,$4}' fic.data
1 Serge 20 Toute 10.50
2 Fredo 15 Camel 10.50
3 Marc 10 Camel 10.50
4 Alain 10 Peter_Bleu 10.50
5 Agnes 20 Goldo_Leg 6.50
6 Isabelle 15 Royal_Ment 12.50
7 Pyr 0 Aucune 0
8 Thyll 10 Camel 10.50
9 LeSaint 0 Aucune 0
```

• écrit le nombre de champ de la ligne, le premier puis le dernier champ de chaque ligne prise en entrée.

```
awk '{print NF,$1,$NF}' fic.data
4 Serge 10.50
4 Fredo 10.50
4 Marc 10.50
4 Alain 10.50
4 Agnes 6.50
4 Isabelle 12.50
4 Pyr 0
4 Thyll 10.50
4 LeSaint 0
```

1.1.2 exécution d'un programme AWK

La commande "awk" est tout d'abord une commande UNIX et par conséquent en présente tous les avantages (pipe, écriture de scripte shell avec des instructions awk).
Mais surtout, AWK est un véritable langage de programmation .
Ainsi, il y a 2 façons générales d'exécuter des instructions AWK:

1 pour des applications simples :

awk 'program' fichier1 ... fichier n
(liste des fichiers de données en entrée)

'program' est une liste d'instructions se présentant sous la forme qui sera définie dans la partie suivante : pattern { action 1; action 2; ... ; action n }.
La commande exécute le programme, fichier par fichier, sur chaque ligne de façon séquentielle, ou s'il n'y a pas de fichier en entrée, prend le standard input en tant que fichier d'entrée.

Exemple :

imprime les lignes où la chaine "Camel" apparait en troisième champ, c'est-à-dire les personnes fumant exclusivement des Camels.

```
awk '$3 == "Camel" {print}' fic.data fic.data
Fredo      15       Camel       10.50
Marc       10       Camel       10.50
Thyll      10       Camel       10.50
Fredo      15       Camel       10.50          imprime une deuxieme fois
Marc       10       Camel       10.50
Thyll      10       Camel       10.50
```

2 pour des applications plus complexes :
On ouvre un fichier d'instructions awk et l'exécution s'effectue par la commande :

awk -f fichierprog liste optionnelle de fichiers de données

La commande exécute séquentiellement le programme se trouvant dans le fichier programme sur chaque ligne des fichiers en entrée.

Exemple : Cohérence des données

On met les instructions suivantes dans le fichier "fichierprog" :

```
NF != 4 {print $0, "le nombre de champ n'est pas egal a 4"}
$2 < 0 {print $0, "le nombre de cigarettes fumees doit etre positif ou nul"}
$4 < 0 {print $0, "le prix doit etre positif ou nul"}
$4 > 20 {print $0, "le prix des cigarettes est cher"}
```

On lance le programme par la commande :

awk -f fichierprog fic.data

1.1.3 structure d'un programme AWK

Nous avons vu précédemment un exemple où AWK était employé comme une commande UNIX et où des conditions et des actions étaient combinées :

```
$2<0 {print ...}
```

Ces deux arguments définissent la structure d'un programme AWK.
Un programme AWK se présente toujours de la façon suivante :

PATTERN { ACTION }
PATTERN { ACTION }
....

Un pattern est une condition portant exclusivement sur une ligne du fichier d'entrée ou sur le standard input.
Une action s'effectue lorsque la condition est validée ou si elle est effectivement exécutable.

Exemples :

• imprime le nom des personnes qui fument.

```
awk '  $2>0    {print $1}  ' fic.data
      pattern    action
Serge
Fredo
Marc
Alain
Agnes
Isabelle
Thyll
```

• la présence de pattern ou d'action n'est pas obligatoire :
écrit toutes les lignes où le second champ est nul, c'est-à-dire les lignes des personnes ne fumant pas.

```
awk '$2 == 0' fic.data     ou     awk '{if ($2 == 0)  print}' fic.data
Pyr         0       Aucune         0
LeSaint     0       Aucune         0
```

L'opération de base de AWK consiste à échantillonner la séquence des lignes d'entrée les unes après les autres, recherchant les lignes qui sont "matchées" par chacun des patterns du programme. AWK travaille de façon séquentielle d'où son intérêt pour la gestion de fichiers.

1.1.4 quelques exemples

Nous supposerons maintenant que les lignes d'instruction sont écrites dans le fichier "fichierprog" et sont lancés par l'alias crée précédemment "a".
Nous donnons ici quelques exemples d'utilisation de format de sortie avec les deux fonctions print et printf dont la dernière s'apparente exactement à la fonction printf du langage C.

• Imprime le nom des fumeurs et le nombre de cigarettes fumées en deux jours

```
$2>0 {print $1,$2*2}
Serge 40
Fredo 30
Marc 20
Alain 20
Agnes 40
Isabelle 30
Thyll 20
```

• **printf (format, valeur 1, valeur 2, valeur 3,....)**
printf crée n'importe quel type de format en sortie.
Le format est une string contenant des %
Chaque % est associé à une valeur : il y a autant de % que de valeur.

```
{printf ("argent depense apres 1 jour pour %s : F %.2f\n", $1,$2*$4/20)}
            %s   : $1 pour une string (nom)
            %.2f : $2*$4/20 pour un nombre avec 2 chiffres apres la virgule
            \n   : retour chariot

argent depense apres 1 jour pour Serge : F 10.50
argent depense apres 1 jour pour Fredo : F 7.88
argent depense apres 1 jour pour Marc : F 5.25
argent depense apres 1 jour pour Alain : F 5.25
argent depense apres 1 jour pour Agnes : F 6.50
argent depense apres 1 jour pour Isabelle : F 9.38
argent depense apres 1 jour pour Pyr : F 0.00
argent depense apres 1 jour pour Thyll : F 5.25
argent depense apres 1 jour pour LeSaint : F 0.00
```

1.2 Débuter avec AWK : les fonctions et les variables usuelles

Cette partie contient quelques notions essentielles lorsqu'on doit programmer en AWK.

1.2.1 aucune déclaration de variable et de type

L'un des principaux attraits du langage AWK est qu'à la fois les variables utilisées et leur type ne sont pas déclarés et qu'une variable peut aussi bien avoir un type correspondant à une chaine de caractère ou à un nombre ou même, elle peut correspondre aux deux types. AWK convertit la valeur d'une chaine en un nombre, ou vice-versa lors de l'exécution du programme.

Puisqu'une variable peut avoir à priori deux types, il faut faire attention quant à son utilisation mais cela profère au langage une très grande souplesse que ne peuvent se permettre le langage C ou le Pascal.

Exemple :

```
$1 == "Fredo" {
              Fredo=10
              $2=$2+Fredo
              string=$1 " a l'etat stone :"
              print string, $2
              }
Fredo a l'etat stone : 25
```

1.2.2 la sélection par les Patterns : le "Matching"

Les Patterns sont très utilisés pour sélecter les lignes d'un fichier qui nous sont utiles : lorsqu'un pattern est vérifié, la ligne correspondante du fichier est sélectée ou " **matchée** " et s'il y a une action qui lui succède, alors l'action est exécutée sur cette ligne sinon la ligne est affichée en sortie.
Cette section donne plusieurs grandes catégories de patterns souvent employés :

1 : Sélection par comparaison
Les symboles de comparaison usuels $(<, <=, ==, != , >=, >)$ peuvent être utilisés à la fois sur des nombres et sur des chaines.

```
Exemple :
imprime le nom des personnes suivi de la chaine ": gros fumeur"
$2 >= 20 {print $1, ": Gros fumeur"}
Serge : Gros fumeur
Agnes : Gros fumeur
```

2 : Sélection par calcul
On peut effectuer toutes les opérations arithmétiques les plus simples que ce soit dans l'utilisation des patterns ou dans l'écriture d'une action.

Exemple :

pattern sélectionnant les lignes contenant les fumeurs qui dépensent par jour une somme supérieure ou égale à 10,50 F.

```
$2*$4/20 >= 10.50
Serge      20       Toute       10.50
```

3 : Sélection par texte
On peut sélecter des lignes spécifiant simplement un mot.

Exemple :

pattern sélectionnant toutes les lignes dont le premier champ est par ordre alphabétique avant la chaine "Fredo".

```
$1 < "Fredo"
Alain      10       Peter_Bleu  10.50
Agnes      20       Goldo_Leg    6.50
```

Mais aussi, il est possible d'effectuer des recherches sur n'importe quel type de caractère en utilisant des expressions régulières, similaires au expression régulières sous "ed" (nous détaillons cet aspect dans le chapitre 2).

Exemples :

• matche toutes les lignes où apparait la chaine "Camel" même sous la forme "RallyeCamel" ou sous la forme "TETEDECamel".

```
/Camel/
Fredo      15       Camel       10.50
Marc       10       Camel       10.50
Thyll      10       Camel       10.50
```

• matche les lignes contenant le symbole y et les imprime.

```
/y/
Isabelle   15       Royal_Ment  12.50
Pyr         0       Aucune          0
Thyll      10       Camel       10.50
```

4 : Combinaison de patterns

Comme en C, les opérateurs booléens AND, OR et NOT sont représentés par `&&`, `||` et `|` .

```
&& ----> AND
|| ----> OR
 | ----> NOT
```

Exemple :

pattern imprimant les lignes où le nombre de cigarettes fumées est supérieur à 10 ET où la marque est Camel.

```
$2 > 10 && $3 == "Camel"    ou    | ($2 <= 10 || $3 != "Camel")
Fredo      15       Camel       10.50
```

5 : BEGIN et **END**

Les patterns BEGIN et END jouent un rôle particulier et sont très souvent employés. Lorsque BEGIN est utilisé (respectivement END), les actions qui lui succèdent sont effectuées avant (respectivement après) que la première (respectivement dernière) ligne du fichier d'entrée soit lue. Ainsi, on peut effectuer des programmes AWK sans avoir nécessairement de fichier en entrée.

Exemple :

```
BEGIN {print; print "NOM       NOMBRE    MARQUE       PRIX";print}
      {print}      # commentaires : action sur le fichier fic.data
END   {print; print "Nombre de lignes lues :", NR
       print "VISUALISATION TERMINEE"; print}

NOM       NOMBRE    MARQUE       PRIX

Serge      20       Toute       10.50
Fredo      15       Camel       10.50
Marc       10       Camel       10.50
Alain      10       Peter_Bleu  10.50
Agnes      20       Goldo_Leg    6.50
Isabelle   15       Royal_Ment  12.50
Pyr         0       Aucune          0
Thyll      10       Camel       10.50
LeSaint     0       Aucune          0

Nombre de lignes lues : 9
VISUALISATION TERMINEE
```

1.2.3 la programmation par les actions : les tests et les boucles

La syntaxe des actions est tout à fait similaire à celle du langage C mais la grande différence réside dans le fait qu'il n'y a aucune déclaration de variable comme nous l'avons dit précédemment.
Nous donnons quelques exemples d'instructions usuelles.

1 : if-else

Les patterns et les actions de test ou de condition sont à priori similaires mais il faut bien voir qu'un pattern agit sur toutes les lignes d'un fichier en entrée (réaction en chaine) alors qu'une action de test peut agir sur toute variable tout en contrôlant l'exécution du programme.

Exemple :

Moyenne sur l'argent depensée par fumeur et par jour.

```
$4 > 0  {n=n+1; argent=$2*$4/20 + argent}
END     {
        if (n>0)
           {
           printf (" Argent depense en moyenne par fumeur par jour :")
           printf ("%5.2f pour %d fumeurs\n\n", argent/n, n)
           }
        else
           print "pas de fumeur parmi les", NR, "personnes"
        }
 Argent depense en moyenne par fumeur par jour : 7.14 pour 7 fumeurs
```

2 : while

Exemple :

Sachant que le prix du paquet de cigarettes augmente tous les 6 mois de 10 %, donnons les prix après 2 années.

```
BEGIN   {printf("\n\t    Marque\t6 mois\t12 mois\t18 mois\t24 mois\n")}
$4 != 0 {i=1; prix=$4;printf("\n\t%10s",$3)
         while (i<=4)
               {
               prix=prix*1.1
               printf("\t%4.1f",prix)
               i=i+1
               }
        }
END     {printf("\n\n")}
```

```
     Marque      6 mois  12 mois 18 mois 24 mois

      Toute      11.6    12.7    14.0    15.4
      Camel      11.6    12.7    14.0    15.4
      Camel      11.6    12.7    14.0    15.4
 Peter_Bleu      11.6    12.7    14.0    15.4
  Goldo_Leg       7.2     7.9     8.7     9.5
 Royal_Ment      13.8    15.1    16.6    18.3
      Camel      11.6    12.7    14.0    15.4
```

3 : for

Exemple :

imprime tous les fumeurs du fichier fic.data en créant une tabulation (\t)

```
$3 != "Aucune" {for (i=1;i<=NF;i++) printf("\t%10s",$i)
                printf("\n")
               }
      Serge         20      Toute      10.50
      Fredo         15      Camel      10.50
       Marc         10      Camel      10.50
      Alain         10 Peter_Bleu      10.50
      Agnes         20  Goldo_Leg       6.50
   Isabelle         15 Royal_Ment      12.50
      Thyll         10      Camel      10.50
```

1.2.4 les opérations AWK sur les chaines de caractères

1 : la concaténation de chaine

En AWK, on peut assembler des chaines de caractères sans que cela pose de problèmes en écrivant les chaines les unes à la suite des autres.

Exemple :

assemble tous les noms des personnes du fichier fic.data dans la variable "noms" puis imprime le résultat

```
{noms=noms $1 " "}
END {print noms}
Serge Fredo Marc Alain Agnes Isabelle Pyr Thyll LeSaint
```

2 : les fonctions AWK

Comme la plupart des langages, AWK permet l'utilisation de fonctions pré-construites admettant des paramètres et retournant le résultat de leur exécution.

• Les fonctions arithmétiques sont donnés dans le chapitre 2 et leur utilisation est très simple:

Exemple :

écrit le logarithme népérien puis le logarithme décimal du prix des paquets.

```
$4 != 0 {print "log du prix :",log($4),"log10 du prix :",log($4)/log(10)}
log du prix : 2.35138 log10 du prix : 1.02119
log du prix : 2.35138 log10 du prix : 1.02119
log du prix : 2.35138 log10 du prix : 1.02119
log du prix : 2.35138 log10 du prix : 1.02119
log du prix : 1.8718 log10 du prix : 0.812913
log du prix : 2.52573 log10 du prix : 1.09691
log du prix : 2.35138 log10 du prix : 1.02119
```

• Les fonctions sur les chaines sont spécifiques au langage AWK puisque toutes les variables sont définies comme des chaines de caractères.
Nous donnons ici deux des fonctions les plus employées : **split** et **gsub**.

La fonction split(s,a) éclate la chaine s, caractère par caractère suivant le séparateur FS et place chaque élément éclaté dans le tableau a (nous voyons les tableaux plus bas). Elle retourne le nombre d'éléments du tableau a.

Exemple :

explosion de la ligne contenant la chaine Fr.

```
/Fr/  {max=split($0,tab)
        for (i=1;i<=max;i++) print i, tab[i]
       }
1 Fredo
2 15
3 Camel
4 10.50
```

La fonction gsub(r,s,t) substitue par s ce qui est matché par l'expression régulière r, globalement dans la chaine t.

Exemple :

suppression du dernier caractère dans la chaine Fredo jusqu'à sa fin définitive et alors sortie forcée du programme par "exit".

```
$1 == "Fredo" {printf("  Mort lente de %s \n", $1)
               chaine=$1
               while ($1 != "")
                     {
                     printf("\t%s\n", $1)
                     gsub(/[a-zA-Z]$/,"", $1)
                     }
               printf("  %s est mort mais il meurt avec $1\n", chaine)
               exit
               }
  Mort lente de Fredo
        Fredo
        Fred
        Fre
        Fr
        F
  Fredo est mort mais il meurt avec $1
```

1.2.5 les tableaux - la notion de tableaux associatifs

AWK permet l'utilisation de tableaux pour stocker des chaines ou des nombres sans qu'il y ait besoin de les déclarer.

Exemple :

écrit le fichier d'entrée dans l'ordre inverse.

```
{x[NR]=$0}
END {for (i=NR;i>0;i--) print x[i]}
LeSaint     0       Aucune          0
Thyll      10       Camel       10.50
Pyr         0       Aucune          0
Isabelle   15       Royal_Ment  12.50
Agnes      20       Goldo_Leg    6.50
Alain      10       Peter_Bleu  10.50
Marc       10       Camel       10.50
Fredo      15       Camel       10.50
Serge      20       Toute       10.50
```

La caractéristique qui fait que les tableaux en AWK sont différents des tableaux employés par les principaux langages est que les indices sont des chaines. Pour cette raison, les tableaux sous AWK sont appelés **tableaux associatifs**.

Exemple :

```
/Camel/ {Nb[Camel]+=$2}
/Peter_Bleu/ {Nb["Peter_Bleu"]+=$2}
END {print " Nombre de Camel fumees :", Nb[Camel]
     print " Nombre de Peter_Bleu fumees :", Nb["Peter_Bleu"]}
 Nombre de Camel fumees : 35
 Nombre de Peter_Bleu fumees : 10

$3 ~ /Toute/ {
      tab[6]="clo";tab[3]="x";tab[4]="eur "
      tab[5]="de ";tab[2]="ta";tab[1]="pes"
      printf("%s ",$1)
      for (c in tab) printf("%s", tab[c])
      print ""
      }
Serge taxeur de clopes

Avec l'option "in", l'ordre d'ecriture est quelconque.
```

1.2.6 créer une fonction par "function"

En plus des fonctions pré-définies, un programme AWK peut contenir ses propres fonctions. De même qu'en C, une fonction se définit de la façon suivante :

function nom (liste de paramètres) { instructions }

La définition des fonctions est entièrement séparée du corps du programme principal. Le corps de la focntion peut contenir l'instruction "return" qui retourne alors un état ou une valeur vers l'instruction appelante.

Exemple :

Lorsque Fredo et Marc se rencontrent, il y a une consommation de cigarettes qui croit par rapport à la normale de façon géométrique (raison 2). On veut calculer au bout de combien de temps chacune des deux personnes aura consommé 20 cigarettes sachant qu'il y a 15 heures dans une journée de fumeur :
On utilise pour cela une fonction calcul qui évalue le temps en minute à partir du nombre de cigarettes fumées par jour.
On utilise aussi la fonction arithmétique "int" qui prend la partie entière d'une valeur numérique.

```
$1=="Fredo" {tab["fredo"]=calcul($2)}
$1=="Marc"  {tab["marc"]=calcul($2)}

END {
     printf ("Temps de rencontre apres 20 cigarettes fumees\n")
     printf ("\t pour Fredo : %d heures", int(tab["fredo"]/60))
     printf (" %d minutes\n", tab["fredo"]-60*int(tab["fredo"]/60))
     printf ("\t pour Marc : %d heures", int(tab["marc"]/60))
     printf (" %d minutes\n", tab["marc"]-60*int(tab["marc"]/60))
    }

function calcul(nombre)
    {
    normal=nombre/15
    terme=normal
    x=0;i=0
    while(x<20)
         {
         i++
         xold=x
         x=x+terme
         terme=2*terme
         }
    cigaminute=(x-xold)/60
    min=(20-xold)/cigaminute
    temps=int((i-1)*60+min)
    return(temps)
    }
Temps de rencontre apres 20 cigarettes fumees
 pour Fredo : 4 heures 18 minutes
 pour Marc : 4 heures 56 minutes
```

1.2.7 les entrées : la fonction "getline "

Il y a plusieurs façons de créer des entrées à un programme AWK. La façon la plus simple est celle que nous utilisons depuis le début, c.a.d. placer des données dans un fichier fic.data puis le lire séquentiellement par les commandes :

```
- awk 'program' fic.data
- awk -f fichierprog  fic.data
```

Une autre façon est d'utiliser les pipes :

Exemple : on utilise la commande grep pour sélectionner la ligne contenant "Fredo" puis on travaille dessus.

```
grep 'Fredo' fic.data | awk '$2>0 {print "Fredo est un fumeur"}'
Fredo est un fumeur
```

La fonction " **getline** " lit l'enregistrement sur l'entrée et incrémente NR. Elle retourne 1 s'il n'y a pas eu d'erreur 0 si c'est la fin de fichier et -1 en cas d'erreur.

Exemples :

• lit la ligne pointée du fichier "fichier" :

```
getline < "fichier"
```

• place dans la variable x la ligne pointée du fichier "fichier" :

```
getline x < "fichier"
```

• tous les exemples précédents peuvent être écrit avec la fonction getline en plaçant le corps du programme dans BEGIN :
Mort lente de Fredo.

```
BEGIN {
      while (getline<"fic.data">0 && $1=="Fredo")
            {
             printf("  Mort lente de %s \n", $1)
             chaine=$1
             while ($1 != "")
                   {
                   printf("\t%s\n", $1)
                   gsub(/[a-zA-Z]$/,"", $1)
                   }
             printf("  %s est mort mais il meurt avec $1\n", chaine)
             exit
             }
      }
```

```
  Mort lente de Fredo
        Fredo
        Fred
        Fre
        Fr
        F
  Fredo est mort mais il meurt avec $1
```

Le programme se lance alors par la commande :

```
awk -f  fichierprog
```

1.2.8 les sorties et les intéractions avec d'autres programmes : la fonction "system"

Nous avons vu avec "print" et "printf" des exemples d'affichage en sortie sur le standard output mais il est possible aussi de diriger la sortie à l'intérieur d'un programme awk vers un fichier.

Exemples :

le premier exemple imprime les noms des fumeurs dans un fichier nommé "nom".
le second exemple imprime à la suite du fichier nom ainsi créé la marque de cigarette fumée.

```
 awk '$2>0 {print $1 > "nom"}'  fic.data
 awk '$2>0 {print $3 >> "nom"}'  fic.data

fichier "nom" resultant

Serge
Fredo
Marc
Alain
Agnes
Isabelle
Thyll
Toute
Camel
Camel
Peter_Bleu
Goldo_Leg
Royal_Ment
Camel
```

Les intéractions avec d'autres programmes se font de façon différente :

1 : les pipes en sortie et en entrée

```
print | commande
commande | getline
```

Exemples :
tri suivant le nom des fumeurs ; on utilise l'option "in" dans for ce qui nous permet d'obtenir tous les éléments du tableau constitué.

```
$2>0 {Marque[$1]=$3}
END  {
     for (c in Marque)
         {
         printf("%8s \t%10s\n", c, Marque[c]) | "sort  -b +0"
         }
     }
   Agnes         Goldo_Leg
   Alain        Peter_Bleu
   Fredo             Camel
Isabelle        Royal_Ment
    Marc             Camel
   Serge             Toute
   Thyll             Camel
```

décompte du nombre de personnes loggées sur le système Arthur à un instant donné.

```
BEGIN {
       while ("who" | getline x) {n++;print x}
       print "personnes loggees sur Arthur :", n
      }
fredo      ttyq1       Jun 12 15:28
bonhair    ttyq2       Jun 12 17:21
gillou     ttyq3       Jun 12 17:35
gemo       ttyq4       Jun 12 16:01
jackie     ttyq6       Jun 12 17:35
renz       ttyq8       Jun 12 16:47
leptose    ttyq13      Jun 12 16:52
personnes loggees sur Arthur : 7

peut aussi etre effectue par la commande Unix :
who | awk '{n++;print} END {print "personnes loggees sur Arthur :", n}'
```

2 : la fonction **"system"** La fonction "system (expression)" exécute la commande "expression" donnée sous forme de string :

Exemple :

plusieurs exemples d'ecritures sur le standard erreur.

```
BEGIN  {
       printf("\t Message : ")
       getline message
       print 1, message | "cat 1>&2"
       system("echo '2 "message"' 1>&2")
       print 3, message >"/dev/tty"
       }
          Message : And my blood is colder, closed the doors of my mind
1 And my blood is colder, closed the doors of my mind
2 And my blood is colder, closed the doors of my mind
3 And my blood is colder, closed the doors of my mind
```

Chapter 2

Le langage AWK en détail

Ce chapitre explique, à l'aide d'exemples, tous les ingrédients de programmation en AWK.
La première section décrit les patterns en détail. La seconde donne la description des actions par les expressions, les affectations et les actions de tests et de boucles. Les sections restantes couvrent la définition des fonctions, des sorties, des entrées et comment des programmes awk peuvent intéragir sur d'autres programmes.

Le fichier d'entrée choisi comme exemple dans cette partie sera un fichier texte

Fichier **fields** :

```
Would you pay lifes pleasure
to see me
Does it heart for I want you to remain
I run your hair through, in another decade
Summerlands holds me in sumerian haze
Between the spaces, along the wall
appearing faces that disappear at dawn
Were getting closer, I can see the door
closer and closer
Kthulhu calls
forever remain, forever remain
```

2.1 Les patterns

Les patterns contrôlent l'exécution des actions : quand un pattern est vérifié, il matche la ligne concernée et l'action est exécutée sur cette ligne. Cette partie décrit les six types de patterns et les conditions sous lesquelles elles sont vérifiées.

PATTERN	EXEMPLE	MATCHE
BEGIN	BEGIN	avant que toute entrée soit lue
END	END	après que toute entrée soit lue
expression	NF>5	matche les lignes où le nombre de champs est supérieur à 5
patterns sur chaines	/closer/	matche les lignes contenant la string "closer"
composé de patterns	NF>5 && /I/	matche les lignes contenant plus de 5 champs et contenant la string "I"
intervalle	NR==10, NR==20	matche les lignes de 10 à 20

2.1.1 BEGIN et END

Les patterns BEGIN et END ne matchent aucune ligne en entrée. Les instructions qui succèdent le pattern BEGIN sont exécutées avant toute lecture en entrée. De façon similaire, les instructions qui succèdent le pattern END sont exécutées après toute lecture en entrée.
BEGIN et END ne se combinent avec aucun autre pattern.
Si plusieurs BEGIN apparaissent dans un programme, les actions associées sont exécutées dans l'ordre auquel ils apparaissent dans le programme.

Exemple :

```
BEGIN {
      print "on met en general les actions BEGIN en tete de fichier"
      }
NF > 5
BEGIN {
      printf("je suis toujours execute avant la premiere ligne d'entree\n")
      }
on met en general les actions BEGIN en tete de fichier
je suis toujours execute avant la premiere ligne d'entree

Does it heart for I want you to remain
I run your hair through, in another decade
Summerlands holds me in sumerian haze
Between the spaces, along the wall
appearing faces that disappear at dawn
Were getting closer, I can see the door
```

2.1.2 Les expressions

Comme la plupart des langages de programmation, AWK est riche dans la description des opérations arithmétiques. Par contre, contrairement à tous les langages, AWK comprend des expressions sur les chaines de caractères.
Une string est une chaine de zéro ou plus de zéro caractères. Elles peuvent être stockées dans des variables, ou peuvent apparaitre littéralement en tant que des constantes telles que "Fredo". La chaine "" est appelée la chaine nulle. Le terme de substring signifie une séquence de zéro ou plus de zéro caractères à l'intérieur d'une string.
Toute expression peut être utilisée en tant qu'opérande de tout opérateur. Si une expression a une valeur numérique mais qu'un opérateur requiert une string, alors la valeur numérique est automatiquement transformée en une string et réciproquement.
Les patterns les plus typiques sont les comparaisons entre les chaines ou nombres. Ces opérateurs sont listés dans le tableau qui suit.
L'opérateur ~ (tilde) sera expliqué dans la partie suivant.

OPERATEUR	SIGNIFICATION
<	plus petit que
<=	plus petit ou égal à
==	égal à
!=	différent de
>=	plus grand ou égal à
>	plus grand que
~	matché par
!~	non matché par

Dans le cas d'une comparaison, si les 2 opérandes sont numériques, une comparaison numérique est effectuée; sinon, tout opérande numérique est converti en une string et alors les opérandes sont comparés en tant que string.
Les strings sont comparées caractère par caractère selon l'ordre défini par la machine, souvent selon les codes ASCII. Une chaine est dite inférieure à une autre si elle apparait avant l'autre selon cet ordre.

Exemple :

```
"Fumer" < "Stone"
"Fumee" < "Fumeur"
"Fumeur" < "fumeur"
```

Lorsqu'un pattern de comparaison est validée sur la ligne courante, alors la ligne courante est dite " matchée ", c'est-à-dire sélectée.

2.1.3 Les patterns spécifiques aux chaines

AWK permet de sélecter ou de matcher des chaines de caractères quelconques à partir d'expressions spécifiques appelées **EXPRESSIONS REGULIERES** qui sont très souvent utilisées en Shell et sous éditeur.
Un pattern sur une chaine teste si la chaine contient effectivement la substring matchée par l'expression régulière.
Le tableau ci-dessous donne les 3 grands types de patterns spécifiques à la recherche de chaines de caractères :

TYPE	SIGNIFICATION
/expression régulière/	Matche la ligne courante quand une substring est matchée par l'expression régulière
expression∼/expr.reg./	Matche la ligne courante si la chaine "expression" contient une substring matchée par l'expression régulière
expression!∼/expr.reg./	Matche la ligne courante si la chaine "expression" ne contient pas une substring matchée par l'expression régulière

L'expression régulière la plus simple est une suite de lettres ou de chiffres entre slashs.

Exemple : /Fredo/
Ce pattern recherche sur chaque ligne les occurences de la substring "Fredo" et si cette chaine est trouvée (même sous la forme FredoLeBon), alors la ligne est matchée.

Remarque : / Fredo /
Ce pattern recherche la string "Fredo"; une ligne contenant FredoLeBon ne sera donc pas matchée.

Les deux autres types de patterns spécifiques aux chaines utilisent un opérateur : l'opérateur tilde : ∼

```
          expression ~ /r/
          expression !~ /r/
r represente une expression reguliere
```

L'opérateur ∼ signifie "est matché par" et l'opérateur !∼ constitue son complément.

Exemples :

```
$1 ~ /e/
# matche toutes les lignes dont le premier champ
# contient au moins un "e".
Does it heart for I want you to remain
Summerlands holds me in sumerian haze
Between the spaces, along the wall
appearing faces that disappear at dawn
Were getting closer, I can see the door
closer and closer
forever remain, forever remain

$1 !~ /e/
# matche toutes les lignes dont le premier champ
# ne contient pas de "e".
Would you pay lifes pleasure
to see me
I run your hair through, in another decade
Kthulhu calls
```

2.1.4 Les expressions régulières

Une expression régulière est une expression spécifique contenant des symboles particuliers, qui permet la sélection de n'importe quel type de substring d'un texte par exemple.
Elles sont toujours placées entre slash : /r/ comme nous l'avons déjà vu dans la partie précédente. On les utilise aussi dans toutes les actions de conditions car nous savons que les actions de conditions (tests et boucles) s'apparentent beaucoup aux patterns.
Leur utilisation en tant que pattern se résume donc à :

```
/r/
expression ~ /r/
expression !~ /r/
r represente une expression reguliere
```

Les caractères \,^,$,[ ],|,( ),*,+,? sont à la base de toute construction d'expressions régulières; ils sont appelés "métacaractères". Si on veut matcher l'un de ses symboles dans une expression régulière sous sa définition naturelle, on doit placer un antislash devant (\$ matche le caractère $ par exemple).

1. \,ˆ,$,. :

Le métacaractère \ placé devant un autre métacaractère le transforme en un caractère normal.
Le métacaractère \ placé devant certains symboles spéciaux matche aussi une séquence de tabulations (voir le tableau ci-après)

SEQUENCE	SIGNIFICATION
\b	backspace : supprime le dernier caractère d'une chaine
\f	formfeed : nouvelle page
\n	newline : nouvelle ligne
\r	carriage return : retour à la ligne
\t	tabulation : crée une tabulation de 10 espaces
\ddd	valeur octale ddd où ddd est trois nombres compris entre 0 et 7
\c	tout caractère pris sous sa forme littérale (excepté pour les caractères : \ : \\ et " : \")

Le métacaractère ˆ matche le début d'une string.
Le métacaractère $ matche la fin d'une string.
Le métacaractère . matche un caractère quelconque et un seul.

Exemples :

```
^F   matche un F au debut d'une chaine.
F$   matche un F en fin de chaine.
^F$  matche la chaine composee du simple caractere F.
^.$  matche toute chaine composee de un seul caractere.
...  matche toute suite de trois caracteres.
\.$  matche le point en fin de chaine.
```

2. [ ] :
Une expression régulière consistant en un groupe de caractères entouré de crochets est appelée classe de caractères; elle matche chacun des caractères à l'intérieur des crochets.

Lorsqu'on veut matcher une série de caractères qui se suivent d'après l'ordre définie par la machine, on peut utiliser le tiret.

La classe complémentaire d'une classe quelconque est dénotée par le métacaractère ^ après l'ouverture du crochet.

Exemples :

```
        [AEO]  matche chacun des caracteres A,E et O.
       [^AEO]  matche tous les caracteres sauf A,E et O.
      ^[^ABC]  matche tout caractere en debut de chaine
               sauf A,B et C.
      ^[^a-z-] matche toute chaine consistant en un
               seul caractere sauf les minuscules et "-".
[a-zA-Z][0-9]  matche toute chaine consistant en une lettre
               minuscule ou majuscule suivie d'un nombre.
```

A l'intérieur d'une classe de caractère, tout caractère admet sa signification normale excepté les caractères \ , ^ placé en début et - placé entre 2 caractères.
Ainsi :

```
   [.]   matche le point ".".
  ^[^^]  matche tout caractere en debut de chaine sauf l'accent ^.
```

3. ( ),| :

Il y a deux opérateurs binaires sur les expressions régulières : la concaténation et le ou.
Si r_1 et r_2 sont des expressions régulières, alors :
$r_1|r_2$ matche toute string matchée par r_1 ou r_2.
$(r_1)(r_2)$ matche toute string de la forme xy où r_1 matche x et r_2 matche y.
Si r_1 ou r_2 ne contiennent pas d'opérateur |, les parenthèses peuvent être omises.

Exemple :

```
/([a-zA-Z])(ee)/
# matche les substring contenant un caractere suivi
# de la chaine "ee".

/(forever|closer) (remain|and)/
# matche les lignes contenant l'un des deux
# mots presents dans la premiere parenthese
# suivi par l'un des deux mots presents dans
# l'autre parenthese.
to see me
Between the spaces, along the wall
Were getting closer, I can see the door
closer and closer
forever remain, forever remain
```

4. +,∗,? :

Ces opérateurs unaires sont utilisés pour spécifier des répétitions dans des expressions régulières.
$(r)*$ matche toute chaine consistant en zéro ou plus de zéro substring consécutives matchées par r.
$(r)+$ matche toute chaine consistant en une ou plus de une substring consécutives matchées par r.
$(r)?$ matche la chaine nulle ou toute chaine matchée par r.

Exemple :

```
    F*    :  matche la chaine nulle ou F ou FF, etc...
  FR*E    :  matche FE ou FRE ou FRRE, etc...
  FR+E    :  matche FRE ou FRRE ou FRRRE, etc...
 FRR*E    :  matche FRE ou FRRE ou FRRRE, etc...
  FR?E    :  matche FE ou FRE.
[A-Z]+    :  matche toute chaine de une ou plus de une lettre majuscule
(FR)+E    :  matche FRE, FRFRE, FRFRFRE, etc...

^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)([eE][+-]?[0-9]+)?$
  matche les nombres reels avec un signe optionnel
  et un exposant optionnel.
```

Nous récapitulons dans le tableau ci-dessous toutes les expressions régulières que l'on peut rencontrer et leur signification.

EXPRESSION	MATCHE
c	le non métacaractère c
\c	séquence de tabulation ou le caractère littéral c
^	début de string
$	fin de string
.	tout caractère
[c_1c_2...]	tout caractère c_1, c_2 ....
[^c_1c_2...]	tout caractère sauf c_1, c_2 ....
[c_1-c_2]	tout caractère de c_1 à c_2
[^c_1-c_2]	tout caractère non de c_1 à c_2
$r_1 \vert r_2$	toute chaine matchée par r_1 ou r_2
$(r_1)(r_2)$	toute chaine xy où r_1 matche x et où r_2 matche y les parenthèses ne sont pas obligatoires si on n'utilise pas $\vert$
$(r)*$	zéro ou plus de zéro chaines consécutives matchées par r
$(r)+$	une ou plus de une chaines consécutives matchées par r
$(r)?$	la chaine nulle ou toute chaine matchée par r
(r)	toute chaine matchée par r

2.1.5 Les composés de patterns

Un composé de patterns est un expression qui combine une série de patterns par un opérateur logique : || (OU), && (ET) et | (NON).
Un composé de pattern matche la ligne courante si l'expression évaluée est vraie.

Exemple :

```
/(forever|closer)/ && /^[fc]/
closer and closer
forever remain, forever remain
```

2.1.6 Les intervalles

On définit un intervalle de patterns comme étant deux patterns séparés par une virgule telle que :

$$pat_1,\ pat_2$$

Un intervalle de patterns matche chaque ligne entre une occurence de pat_1 et la prochaine occurence de pat_2, incluse; pat_2 peut matcher la même ligne que pat_1 donnant ainsi en résultat de l'intervalle une seule ligne.

Exemples :

```
/see/, /I/
to see me
Does it heart for I want you to remain
Were getting closer, I can see the door

/I/, /see/
Does it heart for I want you to remain
I run your hair through, in another decade
Summerlands holds me in sumerian haze
Between the spaces, along the wall
appearing faces that disappear at dawn
Were getting closer, I can see the door
```

2.2 Les actions

Dans la structure d'un programme AWK :

PATTERN { **ACTION** } instruction
PATTERN { **ACTION** } instruction
....

les patterns déterminent quand une action doit s'exécuter.

Parfois, une action peut être très simple : print ... ou elle peut être une affectation; souvent, elle regroupe une série de lignes contenant des fonctions, accolades et autres éléments du vocabulaire non terminal qui définissent toute la richesse du langage AWK.

Le tableau ci-dessous donne tous les éléments du vocabulaire de haut niveau intervenant dans une action :

ACTIONS

expressions avec des constantes, variables, affectations, appel de fonctions, etc..
print(*format,liste d'expression*)
printf(*format,liste d'expression*)
if (*expression*) *instructions*
if (*expression*) *instructions* else *instructions*
while (*expression*) *instruction*
for (*expression;expression;expression*) *instructions*
for (*variable* in *tableau*) *instruction*
do *instructions* while (*expression*)
break
continue
next
exit
exit *expression*
{ *expression* }

2.2.1 Les Expressions

Les expressions les plus simples sont des blocs primaires constitués de constantes, de variables, de tableaux, d'appels à des fonctions et de variables pré-définies telles que les séparateurs.

Les opérateurs sur les expressions combinent les expressions entre elles et ils entrent dans cinq grandes catégories : arithmétique, comparaison, logique, conditionelle et affectation.

Enfin, les fonctions prédéfinies et les fonctions que l'on peut construire utilisent les expressions et représentent aussi une grande classe d'expressions.

• **Les constantes, variables, variables pré-définies**
Il existe deux types de constantes, les chaines et les nombres.
Une constante chaine est créée en entourant la chaine de quotes : " telle que "pleasure ", "holds", "me ", "for", "", "?".
Une constante numérique peut être un entier (2000) ou un réel sous toutes les formes suivantes :
2000.01
0.07E-1 ou 0.07e-1
1e6, 1.00E6, 10e5, 0.1e7 et 1000000 sont identiques
Tous les nombres ont la précision maximale dépendant de la machine.

• **Les variables**
Il y a deux types de variables sous AWK :

1. les variables définies par l'utilisateur : ce sont toutes les séquences de lettres, chiffres et soulignés qui ne commencent pas par un chiffre et qui ne font pas partie du vocabulaire non terminal du langage AWK.
Puisque le type d'une variable n'est pas déclaré, AWK détermine son type de part le contexte du programme. Ainsi, AWK convertit la valeur d'une chaine en un nombre ou en une chaine suivant la façon dont on veut utiliser la variable.

2. les variables pré-définies : il existe sous AWK des variables globales qui sont fixées au départ et que l'on ne peut utiliser aléatoirement. Par contre, la valeur de ces variables peut être modifiée par l'utilisateur tout en restant cohérent avec sa définition.

Le tableau ci-dessous liste la forme littérale, la définition et la valeur par défaut de chacune de ces variables.

VARIABLE	DEFINITION	DEFAUT
ARGC	nombre d'arguments de la ligne de commande	-
ARGV	tableau d'arguments de la ligne de commande	-
FILENAME	nom du fichier d'entrée courant	-
FNR	nombre d'enregistrement du fichier courant	-
FS	controle le séparateur de champ en entrée	" "
NF	nombre de champ de l'enregistrement courant	-
NR	nombre d'enregistrements lus	-
OFMT	format de sortie pour les nombres	"%.6g"
OFS	séparateur de sortie	" "
ORS	séparateur des enregistrements	"\n"
RLENGTH	longueur d'une chaine matchée par la fonction "match"	-
RS	controle le séparateur d'enregistrement en entrée	"\n"
RSTART	début de la chaine matchée par la fonction "match"	-
SUBSEP	séparateur d'indice	"\034"

Exemple :

```
BEGIN {
      OFS="  "
      print "nombre d'arguments :", ARGC "\n"
      for (c in ARGV) print c,ARGV[c]
      printf("\n")
      for (i=0;i<ARGC;i++) printf("  %s",  ARGV[i])
      printf("\n\n")
      printf("NOM du fichier1 : %s\n", FILENAME)
      }
{print "FNR=" FNR,"NR=" NR,$0}
END   {
      printf("NOM du fichier2 : %s\n", FILENAME)
      }
nombre d'arguments :  3

2  fields
0  awk
1  fields

  awk  fields  fields

NOM du fichier1 : fields
```

```
FNR=1  NR=1  Would you pay lifes pleasure
FNR=2  NR=2  to see me
FNR=3  NR=3  Does it heart for I want you to remain
FNR=4  NR=4  I run your hair through, in another decade
FNR=5  NR=5  Summerlands holds me in sumerian haze
FNR=6  NR=6  Between the spaces, along the wall
FNR=7  NR=7  appearing faces that disappear at dawn
FNR=8  NR=8  Were getting closer, I can see the door
FNR=9  NR=9  closer and closer
FNR=10  NR=10  Kthulhu calls
FNR=11  NR=11  forever remain, forever remain
FNR=1  NR=12  Would you pay lifes pleasure
FNR=2  NR=13  to see me
FNR=3  NR=14  Does it heart for I want you to remain
FNR=4  NR=15  I run your hair through, in another decade
FNR=5  NR=16  Summerlands holds me in sumerian haze
FNR=6  NR=17  Between the spaces, along the wall
FNR=7  NR=18  appearing faces that disappear at dawn
FNR=8  NR=19  Were getting closer, I can see the door
FNR=9  NR=20  closer and closer
FNR=10  NR=21  Kthulhu calls
FNR=11  NR=22  forever remain, forever remain
NOM du fichier2 : fields
```

On peut additionner des arguments à ARGV ou modifier son contenu; ARGC peut aussi être modifié. A chaque fin de fichier d'entrée, awk traite l'élément suivant non nul de ARGV (jusqu'à la valeur de ARGC-1) comme le nom du prochain fichier d'entrée. Ainsi, mettre un élément de ARGV à zéro signifie qu'il ne sera pas traité comme un fichier d'entrée. Le nom "-" peut être utilisé pour l'entrée standard. nous voyons un exemple dans la partie Intéraction avec les autres programmes.

• **les opérateurs arithmétiques, logiques et d'affectation**
AWK permet la combinaison des expressions par les opérateurs.
Nous listons les différents opérateurs dans le tableau ci-dessous.

<table>
<tr><th>OPERATION</th><th>OPERATEURS</th><th>EXEMPLE</th><th>SIGNIFICATION</th></tr>
<tr><td>affectation</td><td>= += -= *=
/= %= ^=</td><td>$x* = 2$</td><td>$x = x * 2$</td></tr>
<tr><td>conditionnel</td><td>? :</td><td>x?y:z</td><td>si x est vrai alors y sinon z</td></tr>
<tr><td>logique OU</td><td>||</td><td>x || y</td><td>1 si x ou y sont vrais
0, sinon</td></tr>
<tr><td>logique ET</td><td>&&</td><td>x && y</td><td>1 si x et y sont vrais
0, sinon</td></tr>
<tr><td>élément de tableau</td><td>in</td><td>i in tab</td><td>1 si tab[i] existe, 0 sinon</td></tr>
<tr><td>matching</td><td>~ !~</td><td>$1 ~ /x/</td><td>1 si le premier champ
contient un x, 0 sinon</td></tr>
<tr><td>relationel</td><td>< <= ==
!= >= ></td><td>x == y</td><td>1 si x est égal à y
0,sinon</td></tr>
<tr><td>concaténation</td><td></td><td>"Bad" " Deal"</td><td>"Bad Deal"; il 'y a pas
d'opérateur explicite
de concaténation</td></tr>
<tr><td>somme, soustraction</td><td>+ -</td><td>x + y</td><td>somme de x et de y</td></tr>
<tr><td>mult.,div.,modulo</td><td>* / %</td><td>x % y</td><td>reste de la division de x par y</td></tr>
<tr><td>plus et moins unaire</td><td>+ -</td><td>-x</td><td>opposé de x</td></tr>
<tr><td>logique NON</td><td>|</td><td>|$1</td><td>1 si $1 n'est pas zéro ou ""
0, sinon</td></tr>
<tr><td>puissance</td><td>^</td><td>x^y</td><td>xŷ</td></tr>
<tr><td>incrément
décrément</td><td>++ –</td><td>++x, x++</td><td>additionne 1 à x</td></tr>
<tr><td>champ</td><td>$</td><td>$i+1</td><td>valeur du ième champ
additionné de 1</td></tr>
<tr><td>groupement</td><td>( )</td><td>($i)++</td><td>additionne 1 à la valeur
du ième champ</td></tr>
</table>

Opérateurs arithmétiques : Ce sont +,-,*,/,% et ^ qui s'effectuent uniquement sur des nombres. Toutes les opérations arithmétiques sont faites en précision maximale. Si une variable sous forme de chaine constitue l'un des membres de l'opérateur, elle est convertie en sa valeur qui doit être un nombre.

Opérateurs logiques : Les opérateurs && (ET), || (OU) et | (NON) sont uitlisés pour créer des expressions logiques en combinant d'autres expressions. Une expression logique a une valeur de 1 si elle est vraie ou de 0 dans le cas contraire. AWK évalue les opérations de gauche à droite et ne fait pas de travaux superflus.
Exemple : pour $expr_1$ && $expr_2$, $expr_2$ n'est pas évalué si $expr_1$ est faux.

Expressions de condition : Une expression de condition a la forme :

$$expr_1 \ ? \ expr_2 : expr_3$$

$expr1$ est évaluée; si elle est vraie, c.a.d. non nulle ou non égale à "", la valeur de l'expression est égale à la valeur de $expr_2$, sinon la valeur de l'expression est égale à la valeur de l'expression $expr_3$.

Opérateurs d'affectation : Il y a 7 types d'affectations dont la plus simple est de la forme $variable = expression.$
Les 6 autres opérateurs d'affectation sont +=,-=,*=,/=,%= et ^= qui représentent une simplification d'utilisation du premier type d'opérateur d'affectation pour certains cas particuliers.
Exemple : `age=age-10` peut s'écrire plus simplement `age-=10` ce qui fait rajeunir.

Opérateurs d'incrémentation et de décrémentation : l'expression n=n+1 est plus commodément représentée par n++ ou ++n. La forme préfixée ++n incrémente n avant de délivrer sa valeur alors que la forme postfixée n++ incrémente n après avoir délivré sa valeur.
Exemple:

```
BEGIN {
      printf("\t\tParodie de la vie  \n")
      n=0
      i=n++; printf("\t j'ai %d ans et je viens de naitre\n",n*10^i)
      ++n
      j=++i; printf("\t j'ai %d ans et je suis un jeune initie\n",n*10^j)
      printf("\t j'ai %d ans et je suis parvenu a la sagesse\n",(j+n)*10^n)
      }
                Parodie de la vie
         j'ai 1 ans et je viens de naitre
         j'ai 20 ans et je suis un jeune initie
         j'ai 300 ans et je suis parvenu a la sagesse
```

Opérateurs sur les chaines : il n'y a qu'un type d'opérateur sur les chaines : la concaténation. Cet opérateur n'est pas explicite; les expressions sont créées en écrivant des constantes, variables, champs, éléments de tableaux, résultats de fonction ou autres expressions les unes à la suite des autres

```
{ print NR ":" $0 }
1:Would you pay lifes pleasure
2:to see me
3:Does it heart for I want you to remain
4:I run your hair through, in another decade
5:Summerlands holds me in sumerian haze
6:Between the spaces, along the wall
7:appearing faces that disappear at dawn
8:Were getting closer, I can see the door
9:closer and closer
10:Kthulhu calls
11:forever remain, forever remain
```

Chaines en tant qu'expressions régulières : Dans beaucoup de nos exemples précédents, le membre de droite de l'opérateur ~ ou !~ était une expression régulière entourée de slashs. En fait, toute expression peut être utilisée dans le membre de droite de ces deux opérateurs. AWK évalue l'expression, convertit la chaine si nécessaire et interprète la chaine comme une expression régulière.
Exemple : imprime les lignes où apparaissent les mots see et closer.

```
BEGIN { a_trouver = "(see|closer)" }
$0 ~ a_trouver
to see me
Were getting closer, I can see the door
closer and closer
```

Une expression régulière peut être passée dans une expression de variable et ainsi peut aussi être concaténée.
Exemple : reconnaissance d'un nombre réel

```
BEGIN  {
       signe="[+-]?"
       decimal="[0-9]+[.]?[0-9]*"
       fraction="[.][0-9]+"
       exposant="([eE]" signe "[0-9]+)?"
       nombre="^" signe "(" decimal "|" fraction ")" exposant "$"
       }
$0 ~ nombre
```

On peut donc construire une expression régulière entre quotes comme on construit une expression régulière entre slashs pour les patterns; il y a une seule exception pour les métacaractères; lorsque l'on veut matcher le mètacaractère sous sa forme normale, on doit ajouter un extra slash dans la forme quotée :

```
$0 ~ /(\+|-)[0-9]+/
$0 ~ "(\\+|-)[0-9]+"
```

La suite donne quelques exemples d'utilisation d'expressions régulières. Pour cela, nous modifions légèrement le fichier "fields" en insérant en début et en fin de ligne une série quelconque de blancs. Le fichier fields a alors cette allure

```
  Would you pay lifes pleasure
  to see me
  Does it heart for I want you to remain
  I run your hair through, in another decade
  Summerlands holds me in sumerian haze
  Between the spaces, along the wall
  appearing faces that disappear at dawn
  Were getting closer, I can see the door
  closer and closer
  Kthulhu calls
  forever remain, forever remain
```

```
# PATTERNS qui teste si une chaine contient
# une substring en utilisant les expressions regulieres.

BEGIN {print " Les vers qui contiennent le mot 'see' et qui ne"
       print " contiennent pas le mot 'I'"}
$0 ~ /see/ && $0 !~ /I/ {print NR ," " ,$0}
 Les vers qui contiennent le mot "see" et qui ne
 contiennent pas le mot "I"
2      to see me

# Exemples en utilisant des metacaracteres

#On affiche les vers commencant par une majuscule
BEGIN {
      print " Les vers commencant par une majuscule :"
      }
$0~/^( )*[A-Z]/ {print NR, " ",$0}   # prise en compte des blancs
                                     # en debut de ligne
 Les vers commencant par une majuscule :
1      Would you pay lifes pleasure
3      Does it heart for I want you to remain
4      I run your hair through, in another decade
5      Summerlands holds me in sumerian haze
6      Between the spaces, along the wall
8      Were getting closer, I can see the door
10      Kthulhu calls
```

```
#On affiche les vers se terminant par "e"
BEGIN {
      print " Les vers se terminant par un \"e\" :"
      }
$0~/e( )*$/ {print NR," ",$0}    # prise en compte des blancs
                                 # en fin de ligne
 Les vers se terminant par un "e" :
1     Would you pay lifes pleasure
2     to see me
4     I run your hair through, in another decade
5     Summerlands holds me in sumerian haze


#On affiche les champs contenant la chaine "ces"
BEGIN {
      print " Les champs qui contiennent la chaine \"ces\" :"
      }
$0~/ces/ {
         for (i=1;i<=NF;i++)
             {
             if ($i~/ces/) print $i "\n"
             }
         }
 Les champs qui contiennent la chaine "ces" :
spaces,
faces
```

```
#On affiche les lignes ne commencant pas par
#les lettres B,A,D,b,a,d :
BEGIN {
      print "On affiche les lignes ne commencant pas"
      print "par les lettres B,A,D,b,a,d :"
      }
$0 !~ "^( )*[BAD]|^( )*[bad]" {print}
On affiche les lignes ne commencant pas
par les lettres B,A,D,b,a,d :
  Would you pay lifes pleasure
  to see me
  I run your hair through, in another decade
  Summerlands holds me in sumerian haze
  Were getting closer, I can see the door
  closer and closer
  Kthulhu calls
  forever remain, forever remain
```

• **Les Fonctions pré-définies en AWK**

AWK a des fonctions définies dans son propre langage qui sont les fonctions arithmétiques et les fonctions sur les chaines.

les fonctions arithmétiques : ces fonctions peuvent être utilisées en tant qu'expression primaire dans toutes les expressions définies dans cette partie. Elles sont listées dans le tableau ci-dessous :

FONCTION	VALEUR RETOURNEE
cos(x)	cosinus de x, avec x en radians
sin(x)	sinus de x, avec x en radians
atan2(y,x)	arctangente de y/x retournée dans l'intervalle $-\pi$ et π
sqrt(x)	racine carrée de x
log(x)	logarithme népérien de x
exp(x)	exponentielle de x
int(x)	partie entière de x
rand()	nombre au hasard r, $0 \leq r < 1$
srand(x)	x définit le point de départ dans la génération au hasard rand() srand() fait débuter le générateur selon l'heure

atan2(0,-1) donne π.
exp(1) donne e.
le logarithme décimal se définit par $\log(x)/\log(10)$.
La fonction rand() retourne un nombre de précision maximale selon un pseudo-hasard. En effet, la fonction srand(x) définit à partir de x le début de la génération au hasard. Si on appelle srand() sans paramètre, alors la génération commencera à partir de l'heure. Si on n'appelle pas srand, rand commencera toujours à la même valeur, chaque fois que le programme est appelé.
Exemple : faites tourner deux fois les deux programmes suivants :

```
BEGIN {srand(43)                BEGIN {srand()
      x=rand();print int(100*x)       x=rand();print int(100*x)
      x=rand();print int(100*x)       x=rand();print int(100*x)
      x=rand();print int(100*x)       x=rand();print int(100*x)
      x=rand();print int(100*x)       x=rand();print int(100*x)
      }                               }
     9                              56
     4                              34
    46                              19
    63                               3

     9                              58
     4                              38
    46                              20
    63                              75
```

<u>les fonctions AWK sur les chaines</u> : AWK permet l'utilisation des fonctions prédéfinies qui opèrent sur les chaines. Ces fonctions sont listées dans le tableau ci-dessous.

FONCTION	ROLE
length(s)	retourne le nombre de caractère de s
index(s,t)	retourne la première position de la chaine t dans s ou 0 si t n'est pas présent
match(s,r)	teste si s contient une substring matchée par r retourne l'index ou 0 valeurs dans RSTART et RLENGTH
sprintf(fmt,list-expr)	retourne list-expr formattée selon le format fmt
substr(s,p)	retourne le suffixe de s commençant à la position p
substr(s,p,n)	retourne la substring de s de longueur n et commençant à la position p
split(s,a)	éclate s dans un tableau a selon le séparateur FS retourne le nombre d'éléments du tableau a
split(s,a,fs)	éclate s dans un tableau a suivant le séparateur fs retourne le nombre d'éléments du tableau a
sub(r,s)	substitue par s la plus à gauche et la plus longue substring dans $0 matchée par r retourne le nombre de substitutions faites
sub(r,s,t)	substitue par s la plus à gauche et la plus longue substring dans t matchée par r retourne le nombre de substitutions faites
gsub(r,s)	substitue par s toutes les chaines matchées par r dans $0 retourne le nombre de substitutions faites
gsub(r,s,t)	substitue par s toutes les chaines matchées par r dans t retourne le nombre de substitutions faites

r représente une expression régulière;
s et t sont des strings; n et p sont des entiers.

- La fonction **length**(*s*) retourne la longueur de la chaine s en tenant compte des blancs.
Exemple :

```
BEGIN  {chaine="Tous  ces vieux en longs habits  de Cours !?"
        print length(chaine)}
44
```

- La fonction **index**(*s,t*) retourne la position la plus à gauche de la chaine t dans la chaine s ou 0 si t n'apparait pas. Le premier caractère d'une chaine a la position 1 et les blancs sont pris en compte.
Exemple :

```
BEGIN   {x=index("Les nana ben's", "na");print x}
5
```

- La fonction **match**(*s,r*) cherche la substring la plus longue et la plus à gauche qui est matchée par l'expression régulière r dans la chaine s. Elle retourne l'index où la substring commence ou 0 si r ne matche rien. Elle implémente aussi les deux variables RSTART à la valeur de l'index et RLENGTH à la longueur de la substring matchée.
Exemple :

```
{if (match($0,/,/)>0)
    {
    printf("la virgule apparait a la ligne %d, a la position %d\n",NR,RSTART)
    }
}
la virgule apparait a la ligne 4, a la position 24
la virgule apparait a la ligne 6, a la position 19
la virgule apparait a la ligne 8, a la position 20
la virgule apparait a la ligne 11, a la position 15
```

- La fonction **sprintf**(*fmt,*$expr_1, expr_2, ..., expr_n$) retourne sans l'imprimer une chaine contenant $expr_1, expr_2, ..., expr_n$ formattée selon le format défini par l'expression *fmt* et en se référant aux spécifications de la fonction "printf".
Exemple :

```
{x=sprintf("%11s %7s", $1, $2);print x}
      Would     you
         to     see
       Does      it
          I     run
Summerlands   holds
    Between     the
  appearing   faces
       Were getting
     closer     and
    Kthulhu   calls
    forever remain,
```

- La fonction **substr**(*s,p*) retourne le suffixe de s qui commence à la position p. Si **substr**(*s,p,n*) est utilisé, alors seulement les n premiers caractères sont retournés. Si le suffixe est de longueur inférieure à n, alors tout le suffixe est retourné.
Exemple : supprime le "s" de "lifes" de la ligne 1 et concatène le résultat avec d'autres chaines.

```
NR==1 {print substr($4,1,4), "is", $5}
life is pleasure
```

- La fonction **split**(*s,a,fs*) éclate la chaine s dans le tableau a selon le séparateur fs et retourne le nombre d'éléments insérés dans le tableau a. Cette fonction sera mieux décrite dans la partie réservée aux tableaux.

- La fonction **sub**(*r,s,t*) cherche d'abord la substring la plus à gauche et la plus longue matchée par r dans la chaine cible t; si une substring est trouvée, elle est alors substituée par la string s. Elle retourne le nombre de substitutions effectuées.
La fonction sub(r,s) est synonyme de sub(r,s,$0).
La fonction **gsub**(*r,s,t*) est similaire sauf qu'elle effectue toutes les substitutions sur la chaine t à chaque fois qu'elle trouve une substring matchée par r.
Dans la string qui substitue, tout caractère apparait sous sa forme littérale.
Exemple :

```
comparez les deux fonctions sub et gsub :
NR<5 {sub(/.ou/,"OU");print}
OUld you pay lifes pleasure
to see me
Does it heart for I want OU to remain
I run OUr hair through, in another decade

NR<5 {gsub(/.ou/,"OU");print}
OUld OU pay lifes pleasure
to see me
Does it heart for I want OU to remain
I run OUr hair thOUgh, in another decade
```

```
Le programme qui suit recupere les premiers champs des lignes tout
en supprimant ceux qui apparaissent deux fois : on concatene
tous les premiers champs pour creer une expression reguliere
a chaque fois qu'un nouveau champ reapparait.
Pour le test, on rajoutera la ligne suivante au fichier "fields" :
    to escape your body from this
    Funeral end

NR==1 {champ="(" $1 ")"}
NR!=1 && $1!~champ {gsub(/\)/,"",champ);champ=champ "|" $1 ")"}
END {
     print champ
    }
(Would|to|Does|Funeral|I|Summerlands|Between|appearing|
Were|closer|Kthulhu|forever)
```

• Les actions de tests et les boucles

La syntaxe des tests et des boucles est la même que celle du langage C.
Nous regroupons tous les types d'actions de tests et les boucles dans la liste ci-dessous :

{ *instruction* }
    groupement d'instructions
if (*expression*) *instruction*
    si *expression* est vraie, exécution de *instruction*
if (*expression*) $instruction_1$ **else** $instruction_2$
    si *expression* est vraie, exécution de $instruction_1$
    sinon exécution de $instruction_2$
while (*expression*) *instruction*
    tant que *expression* est vraie, exécution de *instruction*
do *instruction* **while** (*expression*)
    exécute une fois *instruction*; après, idem que while
for ($expression_1; expression_2; expression_3$) *instruction*
    équivalent à :
    $expression_1$; while ($expression_2$) {*instruction*; $expression_3$}
for (*variable* **in** *tableau*) *instruction*
    exécute *instruction* avec la valeur de *variable*
    égale à chaque indice pris dans un ordre quelconque de *tableau*
break
    sort immédiatement d'une boucle while, for ou do
continue
    force le commencement de la prochaine itération
    dans une boucle while, for ou do
next
    force le commencement de la prochaine itération
    dans le programme principal
exit
exit *expression*
    force le programme à exécuter les actions END s'il en existe
    sinon sort du programme
    retourne *expression*

Il y a donc deux instructions qui modifient le cycle normal des boucles : break et continue.
- l'instruction "break" provoque une sortie forcée d'une boucle while, for ou do.
- l'instruction "continue " provoque le commencement de l'itération suivante : le programme exécute alors l'expression de test du while ou du do ou $expression_3$ du for.

Les instructions next et exit controlent le déroulement d'un programme AWK
- l'instruction "next" provoque le passage à la prochaine ligne du programme et l'exécution du programme reprend à la prochaine instruction pattern{action}.
- dans les actions END, l'instruction "exit" provoque la fin du programme. Dans toute autre action, cela cause la fin de toute lecture d'entrée et les actions END sont alors exécutées.

- **Les Tableaux**

AWK permet la création de tableaux qui n'ont pas besoin d'être déclarés et dont le nombre d'éléments n'a pas besoin d'être spécifié.
On peut ainsi créer un tableau comme sous n'importe quel type de langage : tableau de chaines ou de nombres indicé par des nombres (Ex : $x[NR] = \$0$).
En fait, les indices d'un tableau sont définis comme des chaines ce qui constitue une grande différence vis à vis des langages usuels. On parle alors de TABLEAUX ASSOCIATIFS. Ainsi, puisque la valeur des chaines 1 et "1" est la même, il revient au même d'écrire tab[1] et tab["1"].On remarque par contre que tab[01] est différent de tab[1] et tab[*disappear*] est différent de tab["*disappear*"]; dans le premier cas, l'indice sera la valeur de la variable disappear et comme c'est une constante, les éléments associés à tab[*disappear*] seront accumulés dans l'état disparu tab[""].

Exemple : nombre d'occurence des mots commençant par une voyelle minuscule.
tri sur le résultat.

```
{for (i=1;i<=NF;i++)
     {
     gsub(/,/,"",$i)   # suppression de la virgule
     ++tab[$i]
     }
}
END {printf("nombre d'occurence des mots\n")
     printf("commencant par une voyelle minuscule\n")
     for (mot in tab )
         {
         if (mot~/^[aeiouy]/)
             { printf("%s:%d\n",mot,tab[mot])|"sort" }
         }
     close("sort")
    }
nombre d'occurence des mots
commencant par une voyelle minuscule
along:1
and:1
another:1
appearing:1
at:1
in:2
it:1
you:2
your:1
```

Le programme précédent utilise une forme de l'instruction "for" qui boucle sur tous les indices du tableau :

for (*variable*) in *tab*
instructions

La boucle exécute les instructions avec la variable prenant comme valeur chaque indice du tableau. L'ordre dans lequel les variables prennent les valeurs des indices est aléatoire.
Les résultats sont imprévisibles si de nouveaux éléments sont ajoutés dans le tableau par les instructions.
On peut déterminer si un indice apparait dans un tableau ou non par l'expression

indice in *tab*

Cette expression prend la valeur 1 si tab[*indice*] existe déjà, et 0 sinon.

Exemple : cherche si "Death" est dans le tableau tab créé précédemment.

```
{for (i=1;i<=NF;i++)
     {
     gsub(/,/,"",$i)
     ++tab[$i]
     }
}
END {
    if ("Death" in tab)
       {
       printf("\tO flammes du sommeil sur un visage d'ange\n")
       printf("\tEt sur toutes les nuits et sur tous les visages\n")
       }
    else
       {
       printf("\tSilence. Le silence eclatant de ses reves\n")
       printf("\tQui enivrent d'ouragans une ame delivree\n")
       }
    }
        Silence. Le silence eclatant de ses reves
        Qui enivrent d'ouragans une ame delivree
```

L'instruction **delete** détruit un tableau avec la syntaxe :

delete *tab* [*indice*]

Exemple :

```
for (i in zombi) {delete zombi[i]}  # le tableau zombi est mort
```

La fonction **split** (*str,tab,fs*) éclate la valeur de la chaine *str* en champs selon le séparateur *fs* et place chacun des champs dans le tableau *tab*. Le nombre de champs produits ou le nombre d'éléments de *tab* est retourné par la fonction.

Exemple : éclatement de la chaine "cheveux en petard" selon le séparateur e.

```
BEGIN {chaine="cheveux en petard"
       max=split(chaine, tab, "e")
       for (i=1;i<=max;i++) print tab[i]
      }
ch
v
ux
n p
tard
```

On peut aussi créer des tableaux multidimensionels en AWK qui sont en fait des simulations utilisant des tableaux à une dimension. Bien que l'on puisse utiliser des indices tels que i,j ou s,p,q,r, awk concatène les indices (en glissant un séparateur entre eux) pour former un unique indice qui n'a plus rien à voir avec l'écriture de l'utilisateur.
L'exemple :

```
for (i=1;i<=10;i++)
    for (j=1;j<=10;j++)
        tab[i,j]=0
```

crée un tableau de 100 éléments dont les indices ont la forme :
`1,1`
`1,2  ......`
En fait, les indices sont stockés comme des chaines qui ont la forme :
`1 SUBSEP 1`
`1 SUBSEP 2  ......`
La valeur de la variable pré-construite SUBSEP est par défaut `"\034"`

Pour tester si une entité appartient à un tableau multidimensionel, on peut écrire :
```
if ((i,j) in tab) ...
```
Pour récupérer tous les éléments du tableau, il suffit de faire :
```
for (k in tab) ...
```

```
Exemple :
BEGIN {
for (i=1;i<=2;i++)
    for (j=1;j<=2;j++)
        tab[i,j]=i+j
for (k in tab)
    {
    split(k,x,SUBSEP)
    print "tab[" k "]=",tab[k] "    tab[" x[1] "," x[2] "]=",tab[k]
    }
}
tab[11]= 2    tab[1,1]= 2
tab[12]= 3    tab[1,2]= 3
tab[21]= 3    tab[2,1]= 3
tab[22]= 4    tab[2,2]= 4
```

2.2.2 Créer une Fonction

En plus des fonctions pré-définies, AWK autorise l'utilisateur à créer ses propres fonctions. Une fonction se définit de la façon suivante :

function *nom*(*liste de paramètres*)
{
instructions
}

La forme générale d'un programme AWK devient maintenant une séquence de pattern-action suivie des définitions des fonctions créés par l'utilisateur.
La liste des paramètres est une séquence de variables séparées par des virgules; à l'intérieur du corps de la fonction, ces variables réfèrent aux arguments passés en paramètre lors de l'appel de la fonction.
Le corps de la fonction peut contenir l'instruction **return** qui retourne la valeur d'une expression qui peut servir dans le niveau supérieur appelant. L'expression est optionnelle.

return *expression*

A l'intérieur d'une fonction, les paramètres sont des variables locales; elles ne durent que le temps d'exécution de la fonction et sont ensuite dépilées. Mais toutes les autres variables sont globales; si une variable n'est pas dans la liste des paramètres, elle peut référencer à une variable existant dans le programme appelant et ainsi on peut en écraser son contenu.

Exemple : évaluation à l'aide du programme précédent des mots qui apparaissent le plus de fois dans le fichier "fields".

```
{for (i=1;i<=NF;i++)
     {
     gsub(/,/,"",$i)
     ++tab[$i]
     }
}
END {
    for (c in tab)
        {maxi=max(tab[c],maxi)}
    printf("     Mots les plus frequents dans ce dedale obscur\n\t\t\"")
    for (c in tab)
        {if (tab[c]==maxi) {printf("%s ", c)}}
    printf("\b\"\n\t      C'est la parole des Fields\n")
    }

function max(a, b)
   {
   return a > b ? a : b
   }

     Mots les plus frequents dans ce dedale obscur
                "the I closer remain"
              C'est la parole des Fields
```

2.2.3 Les sorties

Les fonctions **print** et **printf** génèrent les sorties sous AWK. "print" est utilisée comme nous l'avons déjà vu pour des écritures simples alors que "printf" demande un format d'écriture que nous détaillons dans cette partie. Les sorties peuvent aussi bien être redirigées vers un fichier, un pipe que vers le standard output. Nous listons dans le tableau qui suit toutes les façons possibles de générer des sorties.

print
    écrit $0 sur le standard output.
print *expression, expression ...*
    écrit les *expressions* séparées par **OFS**, terminées par **ORS**.
print *expression, expression ...* > *fichier*
    écrit dans le fichier "fichier" en l'ouvrant s'il n'existe pas
    ou en l'écrasant s'il existe déjà.
print *expression, expression ...* >> *fichier*
    écrit à la suite du fichier "fichier".
print *expression, expression ...* | *commande*
    écrit vers l'entrée de *commande*.

printf(*format,expression, expression ...*)
printf(*format,expression, expression ...*) > *fichier*
printf(*format,expression, expression ...*) >> *fichier*
printf(*format,expression, expression ...*) | *commande*
    idem que pour "print" en spécifiant un *format* à chaque fois.

close(*fichier*), **close**(*commande*)
    clôture la connection entre "print" et *fichier* ou *commande*
    fermeture des pipes.

system(*commande*)
    exécute *commande*; la valeur retournée est celle de *commande*.

• La fonction "print" a deux formes :

```
print expr1, expr2, ... , exprn
print(expr1, expr2, ... , exprn)
```

qui ont le même rôle : écriture de chaque expression séparée par le séparateur de champ de sortie (OFS : output field separator) et suivie du séparateur d'enregistrement de sortie (ORS : output record separator).
L'instruction `print` est équivalente à `print $0` : elle écrit l'enregistrement `$0` en sortie.
L'instruction `print ""` écrit une ligne blanche en sortie.

• Les séparateurs de sortie :
Il existe deux séparateurs de sortie : **OFS** qui définit le séparateur entre chaque champ et **ORS** qui définit le séparateur après un enregistrement (un enregistrement étant une ligne de champs).
Exemple :

```
BEGIN { FS="faces"
        OFS=", "
        ORS="\nThat's the last exit for the lost .........\n" }
$0 ~ FS {
         max=split($1,T21," ")
         gsub(/ing/,"ed",T21[1])
         gsub("^a","A",T21[1])
         print "The seven keys","\nIn front of my " FS ,"\n" T21[1] "."
         }
The seven keys,
In front of my faces,
Appeared.
That's the last exit for the lost .........
```

• La fonction "printf" est similaire à celle du langage C. De même que "print", il y a deux formes d'écriture de son expression :

```
printf format, expr1, expr2, ... , exprn
printf(format, expr1, expr2, ... , exprn)
```

Le format est une expression qui contient à la fois du texte normal et des spécifications sur le format de sortie des expressions passées en arguments à la fonction "printf".

Chaque spécification commence par un %, se termine par un caractère qui détermine la conversion requise et peut inclure trois autres modificateurs :

```
-          justifie a gauche l'expression.
largeur    determine l'emplacement en nombre de cases
           ou sera placee la chaine ou le nombre.
.nombre    taille maximale de la chaine ou nombre
           de chiffres apres la virgule.
```

Nous donnons dans les tableaux qui suivent la liste des caractères qui interviennent dans l'écriture des formats et des exemples de leur utilisation.

CARACTERE	FORMAT D'ECRITURE
c	caractère ASCII
d	partie entière d'un nombre
e	[-]d.ddddddE[+-]dd
f	[-]ddd.dddddd
g	e ou f selon ce qui est le plus court; supprime les zéros en trop
o	nombre octal non signé
s	chaine
x	nombre hexadécimal non signé
%	écrit un %, aucun argument n'étant utilisé.

<table>
<tr><th>FORMAT : fmt</th><th>EXPRESSION : $1</th><th>printf(fmt, $1)</th></tr>
<tr><td>%c</td><td>97</td><td>a</td></tr>
<tr><td>%d</td><td>97.5</td><td>97</td></tr>
<tr><td>%5d</td><td>97.5</td><td>   97</td></tr>
<tr><td>%e</td><td>97.5</td><td>9.750000e+01</td></tr>
<tr><td>%f</td><td>97.5</td><td>97.500000</td></tr>
<tr><td>%7.2f</td><td>97.5</td><td> 97.50</td></tr>
<tr><td>%g</td><td>97.5</td><td>97.5</td></tr>
<tr><td>%.6g</td><td>97.5</td><td>97.5</td></tr>
<tr><td>%o</td><td>97</td><td>141</td></tr>
<tr><td>%06o</td><td>97</td><td>000141</td></tr>
<tr><td>%x</td><td>97</td><td>61</td></tr>
<tr><td>|%s|</td><td>Death</td><td>|Death|</td></tr>
<tr><td>|%10s|</td><td>Death</td><td>| Death|</td></tr>
<tr><td>|%-10s|</td><td>Death</td><td>|Death |</td></tr>
<tr><td>|%.3s|</td><td>DEATH</td><td>|DEA|</td></tr>
<tr><td>|%10.3s|</td><td>DEATH</td><td>| DEA|</td></tr>
<tr><td>|%-10.3s|</td><td>DEATH</td><td>|DEA |</td></tr>
<tr><td>%%</td><td>Death</td><td>%</td></tr>
</table>

- Les redirections dans les fichiers s'effectuent par les opérateurs > et >>.

Exemple :

```
BEGIN {
      printf("\t\t\"THE FIELDS OF THE NEPHILIM\"\n") > "newfields"
      printf("\t\t -------------------------- \n\n") > "newfields"
      }
{printf("    *     %-43s    *\n", $0)  > "newfields"}
END   {
      printf("\n\t\tLast song of \"The Nephilim\"\n") > "newfields"
      printf("\t\t---------------------------\n\n") > "newfields"
      }
```

```
Resultat dans le fichier "newfields"

                "THE FIELDS OF THE NEPHILIM"
                 --------------------------

    *     Would you pay lifes pleasure                       *
    *     to see me                                          *
    *     Does it heart for I want you to remain             *
    *     I run your hair through, in another decade         *
    *     Summerlands holds me in sumerian haze              *
    *     Between the spaces, along the wall                 *
    *     appearing faces that disappear at dawn             *
    *     Were getting closer, I can see the door            *
    *     closer and closer                                  *
    *     Kthulhu calls                                      *
    *     forever remain, forever remain                     *

                Last song of "The Nephilim"
                ---------------------------
```

On peut écrire dans deux fichiers à la fois; on peut lire et écrire dans deux fichiers différents mais on ne peut écrire et lire dans un même fichier.
Exemple : `{print $1, ($2 > $3)}` Lors de la lecture du fichier d'entrée, le premier champ de chaque enregistrement est écrit sur le standard output et le deuxième champ est écrit dans le fichier dont le nom est la valeur du troisième champ.

• Les sorties dans les pipes sont possibles uniquement si le système supporte les pipes. L'instruction :

```
print | "commande"
```

redirige la sortie de "print" vers l'entrée du pipe associée à "commande ". Les pipes sont similaires à ceux employés sous Shell. La commande est mise sous quotes.

Exemple :

```
$1 ~ /(ai|os)/  {print $1 | "tr [a-z] [A-Z]"}
$NF ~ /(ai|os)/ {print $NF | "tr [a-z] [A-Z]"}
REMAIN
CLOSER
CLOSER
REMAIN
```

• Il est nécessaire de fermer les pipes ou les fichiers que l'on a ouvert en lecture ou en écriture si on veut à nouveau s'en servir dans le même programme ou dans un programme différent. Dans l'exemple précédent, on doit fermer le pipe de la commande par :

```
close("tr [a-z] [A-Z]")
```

Si on veut écrire dans un fichier après l'avoir lu dans un programme, il est nécessaire de fermer le fichier en lecture par :

```
close("fichier")
```

Nous verrons quelques exemples dans la partie Intéraction avec d'autres programmes.

2.2.4 Les entrées

Il y a plusieurs façons d'invoquer des entrées vers un programme AWK. La plus courante est celle que nous avons employée tout au début du chapitre 1, c.a.d. utiliser "awk" en tant que commande UNIX ; s'il n'y a pas de fichier en entrée, le standard input est pris par défaut.

```
awk 'programme' fichier_entree ou arguments
awk 'programme' standard input
```

En tant que commande UNIX, l'entrée peut aussi être un pipe.

```
Exemple : la commande "grep" recupere une ligne du fichier "fields"
          et un message s'affiche alors a l'ecran.
grep 'forever' fields | awk '{print "Death will come"
                              print "To cover all your sad wishes"
                              $NF="in your faces"
                              print $0}'
Death will come
To cover all your sad wishes
forever remain, forever in your faces
```

• Les séparateurs sur l'entrée :
Il y a deux séparateurs d'entrée comme il y a deux séparateurs de sortie : FS (Field Separator équivalent à OFS) le séparateur de champs et RS (Record Separator équivalent à ORS) le séparateur d'enregistrements qui en général est "\n", aussi les termes de lignes et d'enregistrement sont synonymes.
FS qui par défaut vaut " " peut être modifié en une expression régulière toujours mise entre quotes. Alors, la substring non nulle la plus à gauche et la plus longue matchée par l'expression régulière deviendra le séparateur de champ dans la ligne courante. A chaque fois que l'expression régulière matche une substring, NF est incrementé et la substring devient la chaine nulle "". A chaque enregistrement, NF est réinitialisé à 1.

```
Exemple : FS est initialise a "a" ou a la virgule
          suivi d'une suite de blancs
BEGIN { FS="(,[ ]*|a)" }
      {for (i=1;i<=NF;i++) {printf("%s ", $i)}
       printf("%d\n",NF)
      }
Would you p y lifes ple sure 3
to see me 1
Does it he rt for I w nt you to rem in 4
I run your h ir through in  nother dec de 5
```

```
Summerl nds holds me in sumeri n h ze 4
Between the sp ces  long the w ll 5
 ppe ring f ces th t dis ppe r  t d wn 9
Were getting closer I c n see the door 3
closer  nd closer 2
Kthulhu c lls 2
forever rem in forever rem in 4
```

FS peut aussi être modifié directement dans la ligne de commande en ajoutant l'option **-F**. Sur l'exemple précédent, on peut donc exécuter le programme contenu de le fichier "fichierprog" en ayant préalablement supprimé l'action BEGIN :

```
Le fichierprog se presente donc maintenant comme :
      {for (i=1;i<=NF;i++) {printf("%s ", $i)}
       printf("%d\n",NF)
      }
Commande de lancement du programme :
       awk -F',[ ]*|a' -f fichierprog fields
```

Remarque : les différentes versions de awk ne prévoient pas l'utilisation d'une expression régulière après l'option -F; souvent il est seulement possible d'écrire un seul caractère entre quotes après l'option -F (Ex : -F'a' place FS="a").

RS définit la séparation entre chaque enregistrement et vaut par défaut `"\n"` équivalant donc à une ligne. Si on veut faire des enregistrements de plusieurs lignes, on peut l'initialiser à "" et par exemple définir la ligne complète comme un champ en prenant FS=`"\n"`.

• La fonction **getline** :
La fonction "getline " peut être utilisée pour lire des enregistrements en entrée. Elle cherche l'enregistrement suivant et remet à jour les différentes valeurs de NR, NF et FNR dépendant de l'enregistrement lu et du programme. Elle retourne 1 si un enregistrement est effectivement présent, 0 si la fin de fichier est rencontrée ou -1 si une erreur se produit.
Nous explicitons dans le tableau suivant les possibilités d'occurrence de la fonction getline et leur signification :

EXPRESSION	SIGNIFICATION
getline	lit sur le standard input et remplit les variables $0,NR,NF et FNR
getline *variable*	lit l'enregistrement sur le standard input et le met dans *variable* tout en remettant à jour NR et FNR
getline < *"fichier"*	lit l'enregistrement suivant dans "fichier" et le met dans $0 tout en remplissant NF
getline *variable* < *"fichier"*	lit l'enregistrement suivant dans "fichier" et le met dans *variable*
commande | **getline**	lit la sortie de *commande* et le place dans $0 tout en remettant à jour la valeur de NF
commande | **getline** *variable*	lit la sortie de *commande* et le place dans *variable*

L'expression `getline <"fichier"` lit à partir du fichier "fichier" la ligne courante plutôt qu'à partir du standard input. L'expression ne modifie pas NR et FNR mais installe la nouvelle valeur de NF.
L'expression `getline x<"fichier"` lit à partir du fichier "fichier" le prochain enregistrement et le place dans la variable `x` . L'expression ne modifie pas NF, NR et FNR.

Exemple : quand on rencontre l'expression `#include "fields"` dans un programme, on decide de remplacer `#include "fields"` par le contenu du fichier `"fields"`.

```
Le fichier programme "fichierprog" se presente de la facon suivante :
/^#include/ {
            gsub(/"/,"",$2)
            while (getline x <$2 >0)
                  print x
            next
            }
{ print }

Le fichier "f" se presente de la facon suivante :
#include "fields"
a world without end where no soul can descend
there will be no sumertime how lost lifes been afraid of waking up
so afraid to take the dream shapes of angels the night casts lie dead
but dreaming in my past and their here they want to meet you
they want to play with you so take the dream can't break free
and I hear them call they want to plange you their here once more
they want to take you to the shame of your past
take the dream take me lead me far away take me there
I'll fade away but I can't hide.

On lance la commande awk -f fichierprog f et on obtient :
Would you pay lifes pleasure
to see me
Does it heart for I want you to remain
I run your hair through, in another decade
Summerlands holds me in sumerian haze
Between the spaces, along the wall
appearing faces that disappear at dawn
Were getting closer, I can see the door
closer and closer
Kthulhu calls
forever remain, forever remain
a world without end where no soul can descend
there will be no sumertime how lost lifes been afraid of waking up
so afraid to take the dream shapes of angels the night casts lie dead
but dreaming in my past and their here they want to meet you
they want to play with you so take the dream can't break free
and I hear them call they want to plange you their here once more
they want to take you to the shame of your past
take the dream take me lead me far away take me there
I'll fade away but I can't hide.
```

L'exemple qui suit exécute la commande UNIX "who" et pipe le résultat vers getline. Chaque itération du while lit une ligne de la liste des utilisateurs loggés et place la ligne dans la variable d. Puisque NF n'est pas modifié, pour récupérer le nom des utilisateurs, nous sommes obligés d'éclater la ligne contenue dans la variable d selon le séparateur de champ habituel FS. Nous avons alors le nom de l'utilisateur dans le premier élément du tableau ainsi créé par l'éclatement de d. Ce programme se lance par la commande `awk -f fichierprog`

```
BEGIN {while ("who" | getline d)
             {
             split(d,a)
             if (a[1]~/fredo/)
                {
                gsub(/f/,"F",a[1])
                printf("\t%s, Bozo et sa Basse\n", a[1])
                }
             }
       }
        Fredo, Bozo et sa Basse
```

Voyez la différence avec le programme suivant où nous n'utilisons plus la variable d; NF est cette fois remis à jour.

```
BEGIN {while ("who" | getline)
             {
             if ($1~/fredo/)
                {
                gsub(/f/,"F",$1)
                printf("\t%s, Caro et ses gros lolos\n", $1)
                }
             }
       }
        Fredo, Caro et ses gros lolos
```

Remarque : dans tous les cas où "getline " est utilisée, il se peut qu'une erreur se produise si le fichier à accéder n'existe pas. Ainsi écrire `while (getline<"fichier") ...` peut être dangereux et peut tourner en rond indéfiniment si le fichier "fichier" n'existe pas. Nous conseillons d'utiliser `while (getline<"fichier">0)` qui teste si aucune erreur se produit dans la lecture du fichier.

• Les variables dans la ligne de commande :
Comme nous l'avons déjà vu, une ligne de commande AWK peut avoir plusieurs formes :

```
awk 'programme' f1 f2 ...
awk -f fichierprog f1 f2 ...
awk -F'separateur' 'programme' f1 f2 ...
awk -F'separateur' -f fichierprog f1 f2 ...
```

Dans ces lignes de commande, f1, f2 ... représentent des arguments qui sont normalement des fichiers.

• Les arguments dans la ligne de commande :
Les arguments d'une ligne de commande AWK sont accessibles via le tableau **ARGV**. La valeur de la variable **ARGC** est égale au nombre d'arguments plus un.
Ecrivant la ligne de commande :

```
awk -f fichierprog b bog=roumain d
```

ARGC a la valeur 4, ARGV[0] contient `awk`, ARGV[1] contient `b`, ARGV[2] contient `bog=roumain` et ARGV[3] contient `d`. ARGC est égal au nombre d'arguments plus un car le nom de la commande `awk` est l'argument zéro.

```
Exemple :
awk -F'a' '$2=="ppe" {print;
                      print "ARGC=" ARGC;
                      for (i=0;i<=ARGC-1;i++)
                           print "ARGV[" i "]=" ARGV[i]
                     }' fields
appearing faces that disappear at dawn
ARGC=2
ARGV[0]=awk  ---> les options et le programme
                  ne sont pas prises en compte
ARGV[1]=fields
```

Les arguments passés dans la ligne de commande peuvent être utilisés en tant que variable uniquement dans les actions BEGIN, sinon ils sont traités comme des noms de fichier et chaque argument est passé dans la variable FILENAME dont le contenu est mis en entrée du programme.

Exemple : On place le programme qui suit dans "fichierprog" et on le lance en passant deux fois le fichier "fields" en entreé. On utilise aussi le fichier "f" précédemment créé.

```
BEGIN {
      print "Actions BEGIN :"
      print "ARGC=" ARGC
      for (i=0;i<=ARGC-1;i++) print "ARGV[" i "]=" ARGV[i]
      print "FILENAME=" FILENAME
      FILENAME="f"
      print "FILENAME=" FILENAME
      print "\nActions sur les arguments en tant que fichier :"
      print "FILENAME reprend la valeur de ARGV[1]"
      }

$0~/^a/ {print "FILENAME=" FILENAME;print}

END   {
      print "\nActions END :"
      FILENAME="f"
      print "FILENAME=" FILENAME
      while (getline<"f">0)
            {if ($0~/^a/) print}
      }
Actions BEGIN :
ARGC=3
ARGV[0]=awk
ARGV[1]=fields
ARGV[2]=fields
FILENAME=fields
FILENAME=f

Actions sur les arguments en tant que fichier :
FILENAME reprend la valeur de ARGV[1]
FILENAME=fields
appearing faces that disappear at dawn
FILENAME=fields
appearing faces that disappear at dawn

Actions END :
FILENAME=f
a world without end where no soul can descend
and I hear them call they want to plange you their here once more
```

2.2.5 Les Intéractions avec les autres programmes

La fonction AWK **system** (*expression*) exécute la valeur de la chaine *expression* prise comme commande. La valeur retournée est celle de la commande exécutée.

Exemple : On prend les deux fichiers suivants :

```
     FICHIER1 :                FICHIER2 :
Je suis le fichier 1       Je suis le fichier 2
```

Le programme qui suit intéragit avec la commande UNIX "cat"

```
BEGIN {
      while (1)
            {
            printf("\tFichier a lire \n")
            printf("\tQuitter : Q\n")
            Reponse=convert()
            print Reponse
            if (Reponse!="Q")
               {
               system("cat " Reponse)
               }
            else exit
            }
      }

function convert ()
      {
      getline chaine
      gsub(/'/,"\\'",chaine)
      "echo "chaine" | tr '[a-z]*' '[A-Z]*'" | getline CHAINE
      return (CHAINE)
      }
```

La fonction "convert" convertit tous les caractères en majuscules.

Exécution :

```
awk -f fichierprog

        Fichier a lire
        Quitter : Q
fichier1
FICHIER1
Je suis le fichier 1
        Fichier a lire
        Quitter : Q
fichier2
FICHIER2
Je suis le fichier 2
        Fichier a lire
        Quitter : Q
fichier1
FICHIER2
Je suis le fichier 2
```

Attention, nous avons oublié de fermer le pipe de
`"echo "chaine" | tr '[a-z]*' '[A-Z]*'"`
Il est necessaire de le fermer par l'instruction :
`close("echo "chaine" | tr '[a-z]*' '[A-Z]*'")`
La fonction convert devient :

```
function convert () {
      getline chaine
      gsub(/'/,"\\'",chaine)
      "echo "chaine" | tr '[a-z]*' '[A-Z]*'" | getline CHAINE
      close("echo "chaine" | tr '[a-z]*' '[A-Z]*'")
      return (CHAINE)
      }
```

Nous pouvons réaliser une commande Shell à partir d'un programme AWK. Dans la plupart de nos exemples, nous avons utilisé l'option -f ou nous avons placé le programme à exécuter entre quotes : `awk '{print $1}' ...`
Pour réduire le nombre d'instruction à taper, nous pouvons mettre la ligne de commande et le programme dans un fichier exécutable que l'on invoquera en tapant uniquement le nom du fichier.

```
awk '{print $1}' $*     est place dans le fichier "champ1"

chmod +x champ1         pour rendre "champ1" executable

champ1  liste de fichiers    execute le programme sur
                               la liste des fichiers
```

Supposons maintenant que l'on veuille créer une commande *champ* qui écrit une combinaison arbitraire de champs sur chacun des fichiers en entrées :

```
champ 2 5 7 ... fichier1 fichier2 ...
  qui ecrirait les 2eme, 5eme et 7eme champs de
  chaque enregistrement, pour chaque fichier en
  entree.
```

Comment reconnaitre les numéros de champ des fichiers ? On peut le faire grâce à la variable ARGV :

```
awk '
BEGIN {
      for (i=1;ARGV[i]~/^[0-9]+$/;i++)
          {
          champ[++nf]=ARGV[i]
          ARGV[i]=""
          }
      if (i>=ARGC)              # pas de noms de fichier
         ARGV[ARGC++]= "-"      # donc force stdin .
      }
{ for (i=1;i<=nf;i++)
      printf("%s%s", $champ[i], i < nf ? " " : "\n")
}
' $*
```

```
champ 2 4 6 fields f
you lifes
see
it for want
run hair in
holds in haze
the along wall
faces disappear dawn
getting I see
and
calls
remain, remain

world end no
will no how
afraid take dream
dreaming my and
want play you
I them they
want take to
the take lead
fade but can’t
```

Chapter 3

Applications

Ce chapitre contient quelques exemples d'applications en AWK et comment on peut utiliser le langage AWK. Les programmes sont des résultats d'études prolongées menées sur la nature qui nous environne.

AWK peut jouer un rôle non négligeable dans plusieurs domaines tels que :

Validation, transformation, traitement de données
Gestion de base de données
Génération de texte, mise à des formats spéciaux
Compilation, Interprétation, Transformation de langages
Expérimentation sur des algorithmes
....

3.1 Base de données

3.1.1 Introduction

Cette partie montre comment AWK peut être utilisé pour extraire des renseignements et générer des résultats à partir de données stockées dans des fichiers.
Une certaine structure de données a été choisie dans notre exemple mais il est clair que la technique peut être appliquée à n'importe quelle structure à partir du moment où elle est bien spécifiée et où elle ne varie pas aléatoirement.

3.1.2 Spécifications

Nous avons choisi un exemple réel d'interrogation d'une base de données qui pourrait être présente dans n'importe quelle discothèque. Elle regroupe par style de musique, les groupes, les titres des albums, le nombre de disques selon le type (cassette, CD ou disque vinyl) présent dans le magasin et les prix associés.
Le fichier se présentera alors de la façon suivante :

```
STYLE  : COLD

GROUPE : THE FIELDS OF THE NEPHILIM
ALBUM  : THE NEPHILIM: ELIZIUM: DAWNRAZOR
SUPPORT: CD=568; K7=54: VINYL=21: K7=1365
PRIX   : 87; 67: 34: 72
...
```

On classe dans la section SUPPORT la disponibilité présente en magasin par type : cassette, CD, disque vinyl et lorsqu'il n'y a plus de disque dans un certain type, le nombre et le prix seront ignorés :

```
Exemple :
Dans le cas precedent, la disponibilite pour chaque album est de :
    THE NEPHILIM : CD      ---->  568 au prix de 87 F par unite
                   K7      ---->   54 au prix de 67 F par unite
    ELIZIUM      : VINYL   ---->   21 au prix de 34 F par unite
    DAWNRAZOR    : K7      ----> 1365 au prix de 72 F par unite
```

On effectue une interrogation par style de musique (liste de tous les groupes avec les renseignements associés dans le style de musique demandé), par groupe (liste de tous les albums présents dans la base, le nombre disponible et leur prix) et par titre d'album (prix, disponibilité, groupe et style) :

```
          --->  Style de musique
          --->  Groupe
          --->  Titre d'album
          --->  Quitter
```

Tant que le choix `Quitter` n'est pas choisie, le programme bouclera et chaque résultat d'interrogation sera à la fois affiché sur la sortie standard et mis dans un fichier "out.data" où tous les résultats seront mémorisés.
Nous permettons aussi la modification de la base de données.

3.1.3 Le fichier de données

Le fichier de données est denommé `musique`. Nous avons choisi une structure de données avec les particularités suivantes :
- Tous les caractères doivent être en majuscule.
- Le champ séparateur entre les éléments virtuels est ": " qui est initialisé en début de programme.
- Le champ de détection du style de musique est "*".
- Le champ "; " est un sous-champ de détection à l'intérieur d'un élément virtuel qui permet de distinguer le type de support et le prix qui lui est associé.

Il se présente comme suit :

```
*
STYLE  : COLD

GROUPE : THE CURE
ALBUM  : BOYS DON'T CRY: SEVENTEEN SECONDS: PORNOGRAPHY: FAITH
SUPPORT: CD=23; VINYL=485; K7=354: VINYL=29; K7=45: K7=384: CD=543; K7=453
PRIX   : 102; 46; 87: 34; 74: 70: 103; 75

GROUPE : THE FIELDS OF THE NEPHILIM
ALBUM  : THE NEPHILIM: ELIZIUM: DAWNRAZOR
SUPPORT: CD=568; K7=54: VINYL=21: K7=1365
PRIX   : 87; 67: 34: 72

GROUPE : JOY DIVISION
ALBUM  : STILL: TO THE CENTER
SUPPORT: VINYL=365: CD=1356; VINYL=132; K7=827
PRIX   : 43: 100; 65; 43

GROUPE : BAUHAUS
ALBUM  : BELLALUGO IS DEAD
SUPPORT: CD=5439; K7=324
PRIX   : 74; 62

GROUPE : THE SISTERS OF MERCY
ALBUM  : MARIAN: TEMPLE OF LOVE
SUPPORT: CD=3565; VINYL=2356; K7=543: CD=453; VINYL=2341; K7=3545
```

```
PRIX   : 86; 41; 72: 78; 43; 69

GROUPE : THIS MORTAIL COIL
ALBUM  : DEATH SHALL COME
SUPPORT: CD=654
PRIX   : 83

GROUPE : DEAD CAN DANCE
ALBUM  : THIS NIGHT
SUPPORT: CD=654; K7=5643
PRIX   : 92; 74

GROUPE : THE MISSION
ALBUM  : CARVENDISH
SUPPORT: CD=743; VINYL=764
PRIX   : 84; 34
*
STYLE  : HARD

GROUPE : GUN'S N ROSES
ALBUM  : WELCOME TO THE JUNGLE: DON'T CRY
SUPPORT: CD=653; VINYL=765: CD=931; VINYL=126; K7=43211
PRIX   : 93; 31: 85; 29; 58

GROUPE : WHITE LION
ALBUM  : I DREAM OF YOU
SUPPORT: VINYL=439; K7=5742
PRIX   : 42; 65

GROUPE : AC/DC
ALBUM  : TOUCH TOO MUCH: FOR THOSE ABOUT TO ROCK: BACK IN BLACK
SUPPORT: CD=568; K7=54: VINYL=21: K7=1365
PRIX   : 87; 67: 34: 72

GROUPE : TRUST
ALBUM  : MARCHE OU CREVE: MISSIONNAIRE: BON SCOTT
SUPPORT: CD=578; K7=84: VINYL=51: K7=7365
PRIX   : 87; 62: 33: 76

GROUPE : TELEPHONE
ALBUM  : LA BOMBE HUMAINE: ARGENT TROP CHER
SUPPORT: CD=541; VINYL=877: CD=2346; VINYL=2346; K7=133
PRIX   : 93; 31: 85; 29; 58
```

```
GROUPE : SCORPIONS
ALBUM  : I'M STILL LOVING YOU
SUPPORT: VINYL=432; K7=371
PRIX   : 42; 65

*
STYLE  : TECHNO

GROUPE : FRONT 242
ALBUM  : CRAZY
SUPPORT: VINYL=432; K7=371
PRIX   : 42; 65

GROUPE : ALIEN SEX FIEND
ALBUM  : SEX IS BEAUTIFUL
SUPPORT: CD=5439; K7=324
PRIX   : 74; 62

*
STYLE  : ROCK

GROUPE : NEW MODEL ARMY
ALBUM  : IMPURITY: I'M PRIED OF YOU
SUPPORT: CD=3545; K7=466: VINYL=768; K7=254
PRIX   : 86; 63: 31; 43

GROUPE : ALAN PARSON PROJECT
ALBUM  : EYE IN THE SKY: STEREOTOMY
SUPPORT: CD=3545; K7=466: VINYL=768; K7=254
PRIX   : 86; 63: 31; 43

GROUPE : PINK FLOYD
ALBUM  : THE WALL: THE DARK SIDE OF THE MOON: OBSCURED BY CLOUDS: ANIMALS: RELICS: ATOM HEART MOTHER: WISH YOU WERE HERE: A MOMENTARY LAPSE OF REASON: UMMAGUMA: MORE
SUPPORT: CD=543; K7=543: VINYL=54; K7=320: CD=346; VINYL=3545: K7=32: CD=3456; K7=285: CD=3455: VINYL=761; K7=645: CD=14376: VINYL=176; K7=3: K7=32
PRIX   : 91; 62: 31; 68: 79; 39: 67: 68; 58: 81: 37; 52: 74: 41; 58: 62

GROUPE : BARCLAY JAMES HARVEST
ALBUM  : VICTIMS OF CIRCUMSTANCE
SUPPORT: CD=5439; K7=324
PRIX   : 74; 62

*
```

3.1.4 Programme

Ce programme fonctionne exclusivement sur un système tel que UNIX et montre aussi ses limites. Le programme `fnac` est le suivant :

```
BEGIN {FS="\: "
   while ((Reponse !~ /^( )*Q( )*$/) && (decision !~ /^( )*Q( )*$/))
   {
      printf("\n\tRecherche ou Mise a jour        -----> R , M\n")
      printf("\t            QUITTER             -----> Q\n")
      getline decision
      if (decision ~ /^( )*R( )*$/)
      {
 system("rm out.data")
         Reponse=""
 while ((Reponse !~ /^( )*Q( )*$/) && (Reponse !~ /^( )*P( )*$/))
 {
    printf("\n\n")
    printf("\tRecherche par groupe            -----> 1\n")
    printf("\t          par album             -----> 2\n")
    printf("\t          par style de musique  -----> 3\n\n")
    printf("\tRevenir a l'etat precedent      -----> P\n\n")
    printf("\t            QUITTER             -----> Q\n")
    getline Reponse

    if (Reponse ~ /^( )*1( )*$/) groupe()
    if (Reponse ~ /^( )*3( )*$/) style()
    if (Reponse ~ /^( )*2( )*$/) album()
 }
 close("out.data")
 print "\tVotre choix se trouvent aussi dans le fichier \"out.data\"\n"
      }
      else if (decision ~ /^( )*M( )*$/)
      {
         print "\tVous allez modifier sous l'editeur \"vi\" d'UNIX la base"
         print "\tMais ATTENTION a respecter la syntaxe."
         print "\t Maintenant taper V \n"
         getline vi
         if (vi ~ /^( )*V( )*$/) system("vi musique")
      }
   }
}

function groupe()
{
```

```
   printf("Nom du groupe?\t")
   G=convert()
   trouve=0
   while (getline<"musique">0 && trouve==0)
      if ($1 ~ /GROUPE/ && $2==G)
      {
         trouve=1
         affichage()
       }
   if (trouve==0)
      printf("\n\tLe groupe ne se trouve pas dans la base\n\n")
   close("musique")
   return(trouve)
}

function style()
{
   printf("Style de musique?\t")
   ST=convert()
   trouve=0
   while (getline<"musique">0 && trouve==0)
      if ($1 ~ /STYLE/ && $2==ST)
      {
         trouve=1
 getline<"musique"
 while ($1!="*")
 {
    getline<"musique"
    affichage()
 }
      }
   if (trouve==0)
      printf("\n\tCe style de musique n'existe pas\n\n")
   close("musique")
   return
}

function album()
{
   printf("Titre de l'album recherche?\t")
   A=convert()
   trouve=0
   while (getline<"musique">0 && trouve==0)
   {
```

```
      if ($1 ~ /STYLE/) {sty=$2}
      if ($1 ~ /GROUPE/) {gr=$2}
      if ($1 ~ /ALBUM/)
      {
 i=2
 while (i<=NF && trouve==0)
    if ($i==A)
       trouve=1
    else i++
      }
   }
   if (trouve==0) printf("\n\tL'album n'est pas dans la base\n\n")
   else
   {
      ind_cd = ind_v =ind_k7 = 0
      decode_sup()
      decode_prix()
      printf("\n\tAlbum recherche : %s\n",A) >>"o.data"
      printf("\n\tNom du groupe : %s\n",gr) >>"o.data"
      printf("\tStyle de musique : %s\n\n",sty) >>"o.data"
      printf("\tLe nombre de CD disponibles : %d au prix : %d FF\n",nb_cd\
      ,prix[ind_cd]) >>"o.data"
      printf("\tLe nombre de VINYL disponibles : %d au prix : %d FF\n",\
      nb_v,prix[ind_v]) >>"o.data"
      printf("\tLe nombre de K7 disponibles : %d au prix : %d FF\n",nb_k7\
      ,prix[ind_k7]) >>"o.data"
      print "********************************************************"\
      >>"o.data"
      close("o.data")
      system("cat o.data")
      system("cat o.data >> out.data")
      system("rm o.data")
   }
   close("musique")
   return
}

function affichage ()
{
   if ($1 ~ /GROUPE/)
   {
      printf("\n\tNom du groupe : %s\n",$2) >>"o.data"
      getline<"musique"
      printf("\tTitre des albums : \n") >>"o.data"
```

```
      for (i=2;i<=NF;i++)
 printf("\t\t %s\n",$i) >>"o.data"
      getline <"musique"
      print "*********************************************************" >>\
      "o.data"
      close("o.data")
      system("cat o.data")
      system("cat o.data >> out.data")
      system("rm o.data")
      print "\n"
   }
   return
}

function convert()
{
   getline chaine
   gsub(/'/,"\\'",chaine)
   "echo "chaine" | tr '[a-z]*' '[A-Z]*'" | getline C
   close("echo "chaine" | tr '[a-z]*' '[A-Z]*'")
   return(C)
}

function decode_sup()
{
   nb_cd = nb_v = nb_k7 = 0
   split($i,sup,"; ")
   for (j in sup)
   {
      split(sup[j],nb,"=")
      if (nb[1] ~ /CD/) { nb_cd = nb[2] ; ind_cd = j }
      if (nb[1] ~ /VINYL/) { nb_v = nb[2] ; ind_v = j }
      if (nb[1] ~ /K7/) { nb_k7 = nb[2] ; ind_k7 = j }
   }
   return
}

function decode_prix()
{
   getline <"musique"
   split($i,prix,"; ")
   return(prix)
}
```

Le programme utilise la plupart des pouvoirs du langage AWK sur la gestion d'un fichier de données. On y rencontre explicitement l'utilisation :

1. des expressions régulières par exemple dans l'acquisition des réponses données par l'utilisateur sous toute leur forme : majuscule ou minuscule, séparées par des blancs.

2 . des fonctions pré-définies AWK par exemple dans la construction des sous-champs avec la fonction split.

3 . des fonctions créées par l'utilisateur par exemple la fonction convert () qui convertit toute chaine de caractères en majuscule tout en supprimant les blancs et qui renvoie la chaine ainsi modifiée.

4 . la majorité des entrées-sorties en AWK : la fonction getline lisant le standard input (acquisition de données de l'utilisateur) ou le fichier `musique` (accès à la base de données), les pipes, les fonctions print et printf redirigées vers un fichier (o.data ou out.data) ou vers le standard output et en y associant la fonction close ().

5 . des fonctions UNIX par la fonction system () ou directement en entrée de la fonction getline pour éviter de construire des fonctions existant déjà dans le système UNIX.

Nous donnons un exemple d'exécution du programme à la page suivante.
Nous lançons le programme par `awk -f fnac`

```
       Recherche ou Mise a jour        -----> R , M
                   QUITTER             -----> Q
 R

       Recherche par groupe            -----> 1
                 par album             -----> 2
                 par style de musique  -----> 3

       Revenir a l'etat precedent      -----> P

                   QUITTER             -----> Q
 1

Nom du groupe? The     Cure

       Nom du groupe : THE CURE
       Titre des albums :
                BOYS DON'T CRY
                SEVENTEEN SECONDS
                PORNOGRAPHY
                FAITH
*********************************************************

       Recherche par groupe            -----> 1
                 par album             -----> 2
                 par style de musique  -----> 3

       Revenir a l'etat precedent      -----> P

                   QUITTER             -----> Q
 2

Titre de l'album recherche?  Don't  cry

Album recherche : DON'T CRY

Nom du groupe : GUN'S N ROSES
Style de musique : HARD

Le nombre de CD disponibles : 931 au prix : 85 FF
Le nombre de VINYL disponibles : 126 au prix : 29 FF
Le nombre de K7 disponibles : 43211 au prix : 58 FF
********************************************************
```

```
        Recherche par groupe            -----> 1
                  par album             -----> 2
                  par style de musique  -----> 3

        Revenir a l'etat precedent      -----> P

                    QUITTER             -----> Q
 P

        Votre choix se trouvent aussi dans le fichier "out.data"


        Recherche ou Mise a jour        -----> R , M
                    QUITTER             -----> Q

 M

        Vous allez modifier sous l'editeur "vi" d'UNIX la base
        Mais ATTENTION a respecter la syntaxe.
        Maintenant taper V

 V

------->  MODIFICATIONS.........
```

3.2 Des mini-langages

3.2.1 Introduction

AWK est souvent utilisé pour développer des interpréteurs, des assembleurs et autres expérimentations sur la syntaxe ou la sémantique de mini-langages, c.a.d. des langages destinés à des applications spécifiques.
Nous avons choisi dans cet exemple de générer un calculateur.

3.2.2 Spécifications

Les expressions arithmétiques avec les opérateurs +,-,$*$ et / peuvent être décrites par la grammaire :

```
 expr ----> terme
            expr + terme
            expr - terme
terme ----> fact
            terme * fact
            terme / fact
 fact ----> nombre
            ( expr )
```

Nous pouvons alors écrire des expressions du type $1 + 2 * 3$ qui seront analysées de la façon suivante :

```
1 -- nombre -- fact -- terme -- expr --------\
+ ------------------------------------------- expr
2 -- nombre -- fact -- terme ---\            /
* ------------------------------ terme -----/
3 -- nombre -- fact ------------/
```

Pour réaliser le calculateur, nous avons besoin d'un "Parser" pour les expressions. On écrit une fonction pour chaque élément non terminal de la grammaire **expr** pour *expr*, **terme** pour *terme* et **fact** pour *fact*.

3.2.3 Programme

```
BEGIN {
   print "\tProgramme de la calculatrice"
   print "\tLaissez un blanc entre les operateurs et les operandes.\n"
   print "\tPour QUITTER    -------------------------> CTRL/C\n"
}
NF > 0 {f = 1
        e = expr()
        if (f <= NF) printf("erreur en %s\n",$f)
        else printf("\t%.8g\n",e)}

function expr( e)                          # terme | terme [+-] terme
   {e = terme()
    while ($f == "+" || $f == "-")
       e = $(f++) == "+" ? e + terme() : e - terme()
    return e}

function terme( e)                          # fact | fact [*/^] fact
   {e = fact()
    while ($f == "*" || $f == "/" || $f == "^")
        if ($(f++) == "*") e = e * fact()
        else if ($(f-1) == "/") e = e / fact()
             else e = e ^ fact()
    return e}

function fact( e)                         # chiffre | (expr)
   {if ($f ~ /^[+-]?([0-9]+[.]?[0-9]*|[.][0-9]+)$/)
       return $(f++)
    else if ($f == "(")
         {
             f++
             e = expr()
             if ($(f++) != ")")
                print("erreur : on atendt ) en %s\n",$f)
             return e
         }
         else
         {
            printf("erreur : on attend un numero ou ( en %s\n",$f)
            return 0
         }}
```

Contents