Ecole Nationale de l'Aviation Civile

ENAC

Département SINA

Division INF

Subdivision SAR

Alain Bannay

UNIX - LINUX

Commandes filtres et script bash

Travaux pratiques (corrigés)

IENAC 2e année – AVI & SITA

RÉFÉRENCE:
V1.0.0 FR du 01.09.2024 / IO4003

WEBSITE:
http://www.enac.fr/

E-MAIL:
alain.bannay@enac.fr

TABLE DES MATIÈRES

1 EXPRESSIONS RÉGULIÈRES ET GREP

COMMANDES ET MÉTA-CARACTÈRES À UTILISER

- Expressions régulières
- `grep` et options

RÉFÉRENCES

Chapitre 1 Les expressions régulières

Chapitre 2 Les commandes filtres :
- **grep** page 17.

SUJETS

Grep et options de base

1. Récupérer sur `https://e-campus.enac.fr/` le fichier *proverbes*. Copiez le dans un de vos répertoires, puis visualisez le.

   ```
   $ less proverbes
   ```

2. Afficher toutes les lignes du fichier *proverbes* contenant la chaîne `travail`, quelque soit la casse. Utiliser pour cela une option particulière de `grep`.

   ```
   $ grep -i travail proverbes
   ```

3. Ajouter une option à la commande précédente pour afficher en couleur le motif recherché.

   ```
   $ grep -i --color travail proverbes
   ```

4. Ajouter une option à la première commande pour afficher uniquement le nombre de lignes correspondant à la recherche.

   ```
   $ grep -i -c travail proverbes
   ```

5. Ajouter une option à la première commande pour afficher uniquement la portion de chaîne correspondant à la recherche.

   ```
   $ grep -i -o travail proverbes
   ```

6. Afficher toutes les lignes du fichier *proverbes* contenant la chaîne `mal`, quelque soit la casse et en mettant en couleur le motif recherché.

   ```
   $ grep -i --color mal proverbes
   ```

   Ajouter l'option `-w` à la commande précédente et déduisez-en le rôle de cette option.

   L'option `-w` permet d'appliquer le motif de sélection sur des mots entiers.
7. Afficher toutes les lignes du fichier *proverbes ne* contenant *pas* le mot `le`, quelque soit la casse.

   ```
   $ grep -v -i -w le proverbes
   ```

Grep et expressions régulières

Remarque : Pour exploiter les opérateurs du jeu étendu des expressions régulières, on utilisera systématiquement `grep -E`.

1. Afficher les lignes du fichier *proverbes* qui possède un thème composé de 4 lettres. Le thème du proverbe est le premier mot de la ligne, suivi par le caractère `:`.

   ```
   $ grep -E '^....:' proverbes
   ```

2. Afficher les lignes du fichier *proverbes* qui possède un nom d'auteur composé de 6 lettres. Le nom de l'auteur d'un proverbe est indiqué entre parenthèses et se situe en fin de ligne.

   ```
   $ grep -E '\(.{6}\)$' proverbes
   ```

3. Afficher les lignes du fichier *proverbes* qui contiennent les mots (option `-w`) `le` ou `la`.

   ```
   $ grep -E -i -w '(le)|(la)' proverbes
   ```

4. Afficher les lignes du fichier *proverbes* qui contiennent le mot (option `-w`) `vertu`, au singulier ou au pluriel.

   ```
   $ grep -E -i -w 'vertus?' proverbes
   ```

5. On souhaite afficher tous les mots commençant par `s` et se terminant par `t`. On tape pour cela la commande suivante :

   ```
   $ grep -w --color 's.*t' proverbes
   ```

   Qu'est-ce qui ne va pas ? Quelle est la correction à apporter ?

   C'est le motif `.` qui ne convient pas. Un mot est une répétition de caractères alphabétiques, et non pas de n'importe quel caractère. La commande correcte est donc :

   ```
   $ grep -E -w --color 's[[:alpha:]]*t' proverbes
   ```

6. Afficher les lignes du fichier *proverbes* qui contiennent au moins un mot (option `-w`) qui se termine par `de`, mais pas le mot `de` lui-même.

```
$ grep -E -i -w '[[:alpha:]]+de' proverbes
```

7. Afficher en couleur les mots du fichier *proverbes* qui se terminent par `ent`.

   ```
   $ grep -E -w --color '[[:alpha:]]*ent' proverbes
   ```

8. Afficher les lignes du fichier *proverbes* qui contiennent au moins un mot (option `-w`) commençant et se terminant par la lettre `s` ou `t` (pas forcément la même).

   ```
   $ grep -w '[st][[:alpha:]]*[st]' proverbes
   $ grep -E -w '(s|t)[[:alpha:]]*(s|t)' proverbes
   ```

9. Afficher les lignes du fichier *proverbes* qui contiennent au moins un mot (option `-w`) commençant et se terminant par la même lettre (`s` ou `t`).

   Note : S'inspirer de la question précédente en ajoutant une mémorisation (`(...)`) et un rappel (`\1`).

   ```
   $ grep -E -w '([st])[[:alpha:]]*\1' proverbes
   ```

Grep, tube et expression régulière

1. À l'aide d'un tube, rediriger la sortie d'un `ls -l` vers un `grep` pour n'afficher que les lignes correspondant à des répertoires.

   ```
   $ ls -l | grep -E '^d'
   ```

2 | LES COMMANDES FILTRES

COMMANDES À UTILISER

- `cat`, `wc`, `cut`, `sort`, `uniq`, `grep`, `tee`
- `|`

RÉFÉRENCES AU COURS

Chapitre 2 Les commandes filtres.

SUJETS

La commande cat

1. Utiliser la commande `cat` conjointement à une redirection en sortie pour concaténer le contenu des fichiers */etc/hostname* et */etc/lsb-release* dans le fichier *info.txt*.

   ```
   $ cat /etc/hostname /etc/lsb-release > info.txt
   ```

2. Afficher les lignes numérotées du fichier */etc/passwd*.

   ```
   $ cat -n /etc/passwd
   ```

La commande wc

1. Expliquer la différence d'affichage entre les commandes :

   ```
   $ wc /etc/passwd
   ```

   et

   ```
   $ wc < /etc/passwd
   ```

   `wc /etc/passwd` : le nom du fichier est passé en argument de la commande, `wc` connait donc son nom et l'affiche.
   `wc < /etc/passwd` : le contenu du fichier est envoyé (par le `bash`) sur l'entrée standard de `wc`. `wc` ne sait pas d'où proviennent les données.

2. Afficher la longueur de la plus longue ligne du fichier *proverbes*.

   ```
   $ wc -L proverbes
   ```

3. Afficher *uniquement* le nombre de caractères du fichier *proverbes*.

   ```
   $ wc -c < proverbes
   ```

4. Afficher le nombre de lignes cumulé de tous les fichiers `.conf` du répertoire */etc*.
   Rappel : le motif `*.conf` permet de sélectionner tous les éléments du répertoire courant dont le nom se termine par `.conf`.

   ```
   $ wc -l /etc/*.conf
   ```

5. Afficher le nombre d'élément du répertoire */etc* qui se termine par `.conf` .

   ```
   $ ls -ld /etc/*.conf | wc -l
   ```

La commande cut

1. Afficher uniquement les 10 premiers caractères de toute les lignes du fichier *proverbes*.

   ```
   $ cut -c -10 proverbes
   ```

2. Le format d'une ligne du fichier *proverbes* est le suivant :
   ⟨*Thème*⟩`:`⟨*Proverbe*⟩ `(`⟨*Auteur*⟩`)`
   Afficher les lignes du fichier *proverbes* sans le ⟨*Thème*⟩ du proverbe (uniquement le proverbe lui-même et l'auteur).
   On supposera qu'il n'y a pas de caractères `':'` dans un proverbe.

   ```
   $ cut -d ':' -f 2 proverbes
   ```

3. En utilisant le fichier */etc/passwd* : pour les lignes se terminant par `/bin/bash` (à sélectionner avec un `grep`), afficher uniquement le nom de login (le premier champ) ainsi que le répertoire d'accueil (le sixième champ).

   ```
   $ grep '/bin/bash$' /etc/passwd | cut -d':' -f 1,6
   ```

La commande sort

1. Afficher le contenu du fichier *proverbes* trié selon l'ordre alphabétique des thèmes (rappel : le thème du proverbe est le premier mot de la ligne).

   ```
   $ sort proverbes
   ```

2. Afficher le contenu du fichier *proverbes* trié selon l'ordre alphabétique inverse des thèmes.

   ```
   $ sort -r proverbes
   ```

3. Afficher le contenu du fichier *proverbes* trié selon l'ordre alphabétique des proverbes.

```
$ sort -k 2 proverbes
```

4. Afficher le contenu du fichier *proverbes* trié selon l'ordre alphabétique des auteurs. Remarques :
   - L'auteur d'un proverbes est indiqué entre parenthèses en fin de ligne.
   - On suppose qu'un proverbe ne contient pas de parenthèse.

   ```
   $ sort -t '(' -k 2 proverbes
   ```

5. La commande :

   `du` ⟨*Rep*⟩

   retourne l'espace disque (en kilo-octet) utilisé par chaque élément de ⟨*Rep*⟩.

   Afficher les éléments du répertoire */boot* dans l'ordre croissant de l'espace disque utilisé par chaque élément.

   ```
   $ du /boot | sort -n
   ```

La commande uniq

1. Afficher la liste des mots du fichier *proverbes* qui commencent par la lettre `p`. Exemple d'affichage souhaité :

   ```
   pas
   passent
   pere
   plaisir
   plus
   pour
   ```

   Rappel : `grep -w -o "p[[:alpha:]]*"` pour sélectionner les mots qui commencent par la lettre `p`.

   ```
   $ grep -w -o "p[[:alpha:]]*" proverbes | sort | uniq
   ```

2. À partir du fichier */etc/passwd*, afficher les nombres d'occurrence et les différentes valeurs des derniers champs, triés selon l'ordre croissant. Exemple :

   ```
    1 /bin/sync
    5 /bin/bash
   17 /usr/bin/nologin
   27 /bin/false
   ```

   Note : Pensez à l'option `-c` de la commande `uniq`.

   ```
   $  cut -d':' -f 7 /etc/passwd | sort | uniq -c | sort -n
   ```

La commande tee

1. Réaliser l'affichage :
   - du nombre de répertoire du répertoire courant dans le terminal courant et

- de tous ces répertoires dans un autre terminal lancé au préalable.

Commandes à utiliser : `ls -l`, `grep`, `tee` et `wc -l`.

Pour connaître le fichier spécial de périphérique associé à un terminal, il suffit de taper la commande *tty* dans ce terminal.

```
$ tty
/dev/pts/3
```

puis, dans un autre terminal :

```
$ ls -l | grep '^d' | tee /dev/pts/3 | wc -l
```

3 | INTRODUCTION AUX SCRIPTS

COMMANDES, MÉTA-CARACTÈRES

- `exit`
- `#!/bin/bash`, `$1`, `$2`, `$*`, `$@`, `$#`

RÉFÉRENCES

Chapitre 3 Programmation en bash :
- **Invocation** page 32,
- **Les arguments** page 34,
- **Les codes de retour** page 36.

SUJETS

Export de variable et shell fils

1. Depuis un shell, créer la variable `un` que vous initialiserez à 1.

   ```
   $ un=1
   ```

2. Créer également la variable `deux` que vous initialiserez à 2 et que vous exporterez.

   ```
   $ export deux=2
   ```

3. Créer un fichier `var.sh` contenant :

   ```
   trois=3
   echo $un $deux $trois
   deux=22
   ```

4. Vérifier que vous possédez bien les droits de lecture et d'écriture sur ce fichier.

   ```
   $ ls -l var.sh
   ```

   et éventuellement

   ```
   $ chmod u=rw var.sh
   ```

5. Lancez la commande :

```
$ bash var.sh
```

et expliquez l'affichage produit.

On lance un fichier de commandes par l'intermédiaire d'un shell. Le fichier *var.sh* n'a pas besoin d'être exécutable, seuls les droits rw suffisent.

L'affichage produit `2 3` : la variable `un` n'étant pas exporté, n'est pas connu dans le shell fils.

6. Lancez la commande :

```
$ echo $deux $trois
```

et expliquez l'affichage produit.

Affichage de `2` : la variable `trois` créée dans un shell fils n'est pas visibles dans le shell père (et ce d'autant plus que le shell fils n'existe plus ici !). La variable `deux` n'a pas été modifiée par le shell fils car celui-ci travail sur une copie de la variable exportée.

Syntaxe d'un script

1. Faites le nécessaire (2 choses) pour pouvoir lancer le script de la manière suivante :

```
$./var.sh
```

- signature d'un script : `#!` pour les 2 premiers caractères. Il faut en plus indiquer le nom de l'interpréteur à utiliser, donc : `#!/bin/bash` en première ligne.
- le fichier doit être exécutable :

```
$ chmod u+x var.sh
```

Code de retour

1. Expliquez la séquence de commandes suivante :

```
$ ls /toto
ls: /toto: No such file or directory
$ echo $?
2
$ echo $?
0
```

Interpréter les valeurs 2 et 0.

La variable `$?` contient le code de retour de la dernière commande exécutée.

Le répertoire */toto* n'existe pas. Dans ce cas, `ls` positionne son code de retour à 2 (cf. `man ls`).

Le `echo $?` se déroule sans erreur, le code de retour est positionné à 0 (synonyme de succès).

Arguments d'un script

1. Écrire un script qui affiche :
   - son nom,
   - ses deux premiers arguments,
   - tous ses arguments,
   - le nombre total de ses arguments.

   Tester ce script avec 0, puis 1, puis 2, puis 3 arguments.

```
#!/bin/bash
echo "Nom du script : $0"
echo "Premier argument : $1"
echo "Deuxieme argument : $2"
echo "Tous les arguments : $*"
echo "Nombre d'argument(s) : $#"
```

2. Modifier le script précédent pour qu'il renvoi en code de retour le nombre d'argument du script. Tester.

   Ajouter en fin de script : `exit $#` et taper la commande

   ```
   $ echo $?
   ```

   après l'exécution du script pour tester.

4 PROGRAMMATION EN BASH

COMMANDES, MÉTA-CARACTÈRES

- `[[ ]]`, `(( ))`, `$(( ))`
- Opérateurs `-e -d -f -z = != -eq -ne -gt ==`
- `for in do done`, `if then else elif fi`, `case in esac`, `while do done`
- `break`, `exit`, `printf`, `read`, `$IFS`

RÉFÉRENCES

Chapitre 3 Programmation en bash
- **Tests et conditions** page 37,
- **Structures de contrôle** page 41,
- **Arithmétique** page 49,
- **Entrée de données** page 54.

SUJETS

Exercice 1 : test sur des éléments de l'arborescence

1. Créer un script `testelem.sh` qui réalise le test d'un élément passé en paramètre pour déterminer :
   - si l'élément est un répertoire,
   - si l'élément est un fichier régulier,
   - si l'élément est autre chose,
   - si l'élément n'existe pas.

   Le script devra s'assurer qu'il est appelé avec un seul argument.

   Exemple :

```
$ ./testelem.sh
Il faut un (seul) argument.
$ ./testelem.sh toto titi
Il faut un (seul) argument
$ ./testelem.sh LePereNoel
LePereNoel n'existe pas
$ ./testelem.sh fic
fic est un fichier régulier
```

```
$ ./testelem.sh rep
rep est un répertoire
$ ./testelem.sh /dev/sda1
/dev/sda1 n'est ni un répertoire ni un fichier régulier
```

Exercice 2 : test numérique, boucle sur les éléments du répertoire courant, exécution imbriquée

1. Créer un script `lsmin.sh` qui affiche uniquement les fichiers réguliers du répertoire courant dont la taille (en nombre de caractères) est supérieur ou égale à un seuil passé en paramètre.

   Si le script est invoqué sans paramètre, le seuil par défaut sera fixé à 1024 caractères.

   La commande qui permet d'obtenir la taille (en nombre de caractères) d'un fichier *fic* est :

   ```
   $ wc -c < fic
   ```

   Dans le script, on stockera cette valeur dans une variable via une exécution imbriquée.

   Algorithme possible :

   ```
   Si le script ne possède pas d'argument Alors
     le seuil vaut 1024
   Sinon
     le seuil vaut le premier argument du script
   FinSi
   Pour tous les éléments du répertoire courant Faire
     Si l'élément courant est un fichier régulier Alors
       Calculer la taille de l'élément
       Si la taille est supérieur au seuil Alors
         Afficher l'élément
       FinSi
     FinSi
   FinPour
   ```

Exercice 3 : calcul numérique, boucle sur tous les arguments

1. Créer un script `somarg.sh` qui réalise la somme de tous les nombres entiers passés en paramètre.

   Remarques :
   - le script acceptera au minimum 1 argument,
   - utilisez une boucle sur la variable positionnelle `$@`.

   Exemple :

```
$ ./somarg1.sh 1 2 3
Somme de tous les arguments : 6.
$ ./somarg1.sh 1 2 -3
Somme de tous les arguments : 0.
```

Exercice 4 : lecture de donnée, boucle infinie, rupture de boucle, test numérique

1. Écrire un script `devine.sh` qui permet de deviner un nombre entre 1 et un nombre max passé en paramètre (valeur par défaut : 10 si pas de paramètre). Indiquer à chaque tentative si le nombre à deviner est plus grand ou plus petit que celui proposé.

   Exemple :

   ```
   $ ./devine.sh 5
   Devinez un nombre entre 1 et 5 :
   Tentative 1 : 3
   Essaie encore... (c'est moins)
   Tentative 2 : 2
   Trouvé !
   ```

   Remarques :
   - la génération d'un nombre aléatoire entre 1 et N s'obtient par :

     `$(($RANDOM%`⟨N⟩` + 1))`
   - le script utilisera une boucle infinie (`while true; do`) avec une rupture de boucle (`break`) en cas de succès.

Exercice 5 : structure **case**, lecture de plusieurs données, boucle infinie, rupture de boucle, opérateurs arithmétiques

1. Écrire un script `calc.sh` permettant de simuler une calculatrice simplifiée.

   Le script effectuera en boucle les opérations demandées par l'utilisateur, jusqu'à ce que celui-ci appuie sur la touche `q`.

   La syntaxe d'une opération sera :

   <*Opérande1*>␣<*Opérateur*>␣<*Opérande2*>

   Le symbole ␣ représente un espace.

   Les opérateurs pris en charge par la calculatrice sont :
   - + pour une addition,
   - - pour une soustraction,
   - x pour une multiplication,
   - / pour une division.

   Le code de retour du script sera représentatif de la dernière opération effectuée :
   - 0 en cas de succès dans l'opération,

- 1 en cas d'erreur.

Les erreurs à détecter sont :

- opérateur inconnu,
- mauvais nombre de paramètre pour l'opération.

Remarques :

- La commande `read` permet de récupérer plusieurs valeurs (séparées chacune par un espace) à partir de l'entrée standard. Exemple : `read VAR1 VAR2`, puis utilisation de `$VAR1`, `$VAR2`.
- Le script utilisera une boucle infinie (`while true; do ...; done`) avec rupture de boucle.
- Les opérations s'effectueront sur des entiers uniquement.

Exemples :

```
$ ./calc.sh
Entrez une opération (q pour sortir) : 1 + 1
1 + 1 = 2
Entrez une opération (q pour sortir) : 5 x 4
5 x 4 = 20
Entrez une opération (q pour sortir) : q
$ echo $?
0
$ ./calc.sh
Entrez une opération (q pour sortir) : 10 / 3
10 / 3 = 3
Entrez une opération (q pour sortir) : 2
Erreur de syntaxe
Entrez une opération (q pour sortir) : 2 +
Erreur de syntaxe
Entrez une opération (q pour sortir) : 2 M 3
2 M 3 = Erreur de syntaxe
Entrez une opération (q pour sortir) : q
$ echo $?
1
```

Exercice 6 : lecture d'un fichier, variable `IFS`, tableau associatif, test de la version du bash

1. Le fichier `airports-FR.csv` contient des informations sur tous les aéroports de France. Chaque ligne représente les informations d'un aéroport, et le format d'une ligne est :

   $\langle IDENT \rangle ; \langle TYPE \rangle ; \langle NOM \rangle ; \langle LAT \rangle ; \langle LONG \rangle ; \langle ALT \rangle ; \langle REGION \rangle ; \langle VILLE \rangle ; \langle IATA \rangle$

   L'élément séparateur entre les champs est le `;`.

   Créer un script `airports.sh` qui affiche le nombre d'aéroport de chaque région.

   Remarques :

   - la lecture de toutes les lignes d'un fichier se réalise à l'aide d'une structure :

```
while read ⟨Liste des champs à lire⟩
do
  ⟨Instructions⟩
done < fichier
```

- on pourra utiliser un tableau associatif pour stocker les réultats, avec en index la région de l'aéroport et en valeur le nombre.
- la variable `IFS` sera positionnée à `;` juste avant le `while`.

Exemple :

```
$ ./airports.sh
Il y a 16 aéroport(s) dans la région FR-OCC
Il y a  6 aéroport(s) dans la région FR-CVL
...
```

PROGRAMMATION BASIQUE (CORRECTION)

testfic.sh

```
#!/bin/bash

# Test d'un élément passé en paramètre pour savoir :
# - si c'est un répertoire
# - si c'est un fichier régulier
# - si c'est autre chose
# - si l'élément n'existe pas

# Test sur nombre d'argument
if [[ "$#" != "1" ]]        # ou [[ $# -ne 1 ]]
then
    echo "Il faut un (seul) argument." >&2
    exit 1
fi

if [[ ! -e "$1" ]]            # Test si element existe
then
    echo "L'element $1 n'existe pas"
    exit 1
fi

if [[ -d "$1" ]]            # Test si repertoire
then
    echo "L'element $1 est un repertoire"
    exit 0
fi

if [[ -f "$1" ]]            # Test si fichier regulier
then
    echo "L'element $1 est un fichier regulier"
    exit 0
fi

echo "L'element $1 n'est ni un rertoire ni un fichier regulier"

exit 0
```

lsmin.sh

```
#!/bin/bash

# lsmin <seuil> :
# affiche uniquement les fichiers réguliers d'une taille
# supérieur ou égale au seuil (en nombre de caractères).

# gestion de la valeur du seuil
if (($#==0))    # ou if [[ $# -eq 0 ]]
then
    seuil=1024    # seuil par défaut
else
    seuil=$1    # seuil passé en argument
fi

echo "Elements de taille >= a $seuil caracteres :"
for elem in ./*    # Boucle sur tous les elements du rep courant
do
    if [[ -f "$elem" ]]
    then
        size=$(wc -c < "$elem")
        if [[ "$size" -gt "$seuil" ]]
        then
            echo "$elem (taille : $size)"
        fi
    fi
done

exit 0
```

somarg.sh

```
#!/bin/bash
# Somme de tous les nombres entiers passés en paramètre.
# Code de retour :
# - 1 si erreur sur nombre de paramètre,
# - 0 sinon.

# Gestion du nombre des paramètres
if [[ $# -eq 0 ]]; then
    echo "Erreur : mauvais nombre de paramètre."
    exit 1
fi

declare -i som=0

for n in "$@"            # ou for n
do
    let som+=n
done

# Affichage du résultat
echo "Somme de tous les arguments : $som."

exit 0
```

devine.sh

```
#!/bin/bash
#
# Deviner un nombre entre 1 et un nombre max passé en paramètre
# (ou 10 si pas de paramètre).

# Code de retour :
# - 1 si erreur sur nombre de paramètre,
# - 0 sinon.

# Gestion du nombre des paramètres
if [[ $# -gt 1 ]]
then
    echo "Erreur : mauvais nombre de paramètre."
    exit 1
fi

# Initialisation de la variable max
if [[ -z "$1" ]]
then
    max=10
else
    max=$1
fi

# Nombre à trouver
nombre=$(($RANDOM%$max + 1))

echo "Devinez un nombre entre 1 et $max :"

declare -i nb
nb=1
while true
do
    echo -n "Tentative n°$nb : "
    read choix
    if (( nombre == choix )) # ou if [[ $nombre -eq $choix ]]
    then
        echo Trouvé \!
        break
    fi
    echo -n Essai encore...
    if (( nombre > choix ))    # ou if [[ $nombre -gt $choix ]]
    then
        echo " (c'est plus)"
    else
        echo " (c'est moins)"
    fi
    let nb+=1    # ou let nb++ ou nb=$(($nb + 1))
done

exit 0
```

calc.sh

```
#!/bin/bash

while true
do
    echo -n "Entrez une opération (q pour sortir) : "
    read OPER1 OP OPER2
    if [[ "$OPER1" = "q" ]]
    then
        break
    fi
    if [[ "$OPER2" = "" ]]
    then
        echo Erreur de syntaxe
        CODE_RET=1
        continue
    fi
    echo -n "$OPER1 $OP $OPER2 = "
    case $OP in
        + | - | /)
            echo $(($OPER1 $OP $OPER2))
            CODE_RET=0
            ;;
        x)
            echo $(($OPER1 * $OPER2))
            CODE_RET=0
            ;;
        *)
            echo Erreur de syntaxe
            CODE_RET=1
            ;;
    esac
done

exit $CODE_RET
```

nb_airports.sh

```
#!/bin/bash

# Compte le nombre d'aéroports par région
# Fichier en entrée : airports-FR.csv
# Éléments séparés par ";"

DATA="./airports-FR.csv"

# Tableau associatif des aéroports avec :
# en index : la région
# en valeur : le nombre
declare -A TabAero

# Le séparateur de champs d'une ligne est le ;
IFS=";"
# Lecture de tous les champs d'une ligne
while read IDENT TYPE NOM LAT LONG ALT REGION VILLE IATA
do
    # Incrémentation du nombre pour la région concernée
    (( TabAero["$REGION"]+=1 ))
done < <(tail -n +2 "$DATA")    # Permet de commencer à la deuxième ligne

# À partir de bash 5.1, utilisation possible de l'opérateur K
# pour afficher les paires Index / Valeur d'un tableau associatif
# Comparaison uniquement sur les 3 premiers caractères
[[ ${BASH_VERSION:0:3} > "5.0" ]] && echo ${TabAero[@]@K}

# Boucle sur toutes les régions (les index du tableau associatif)
for reg in "${!TabAero[@]}"
do
    # Affichage formaté
    printf "Il y a %2d aéroport(s) dans la région %s\n" ${TabAero[$reg]} $reg
done

exit 0
```