Bash

SX32/CNAM-IM, Laurent-Stéphane Didier, Jean-Marc ROBERT

30 mai 2023

Les exercices 2 et 3 sont tirés de Brunet *et al.*, SupTelecom.

1 Quelques manipulations d'affichage et de fichiers

1.1 quelques préliminaires

1. écrire un script qui affiche "Hello World !"
2. écrire un script qui crée un fichier nommé d'après vos initiales et contenant le texte "Hello !"
3. écrire un script qui affiche le nombre d'arguments et les arguments passés au script (un argument par ligne)
4. écrire un script qui crée un fichier nommé d'après le nom donné en premier argument ; ce fichier contiendra le texte passé dans le deuxième argument. Ce script devra aussi gérer le manque d'argument en indiquant à l'utilisateur le bon usage de la commande.
5. écrire un script qui crée 10 fichiers nommés d'après le nom donné en argument et suffixés par un indice. Comme précédemment, ces fichiers contiendront le texte passé en deuxième argument et un compteur correspondant à l'indice du fichier.

1.2 Traitement de fichier texte

Découper des lignes : Le but de cet exercice est d'écrire un petit programme capable de découper des lignes, c'est-à-dire d'extraire certains mots séparés par des blancs.

1. Pour commencer, nous allons vérifer que les arguments du script ne sont pas invalides. Les arguments sont invalides si :
   — le nombre d'arguments n'est pas égal à 2,
   — ou si le premier argument ne correspond pas à un fichier existant (test `[ ! -e fic ]`),
   — ou si le premier argument correspond à un répertoire (test `[ -d fic ]`).
   En cas d'erreur, le script doit quitter en affichant un message adéquat et en renvoyant faux (valeur 1), sinon, il doit quitter en renvoyant vrai (valeur 0).
2. Complétez votre script pour qu'il lise, ligne à ligne, le fichier passé en argument (premier argument) et écrive ces lignes sur la sortie standard. On vous rappelle que pour lire un fichier ligne à ligne à partir du chier fichier, il faut utiliser la construction `while read line; do ... done <fichier`.
3. Modifiez votre script pour qu'il itère sur chaque mot de chaque ligne non vide (indiqué par `[ -n "$line" ]`), et affiche le mot sur une ligne séparée. Pensez à utiliser une boucle `for` pour itérer sur les mots de la ligne précédemment lue.

4. Modifiez votre script pour qu'il n'affiche que le $n^{\text{ième}}$ mot de chaque ligne, où n est le second argument du script. Pour mettre en œuvre cet algorithme, nous vous conseillons d'utiliser une variable annexe, par exemple *num*, que vous initialiserez à zéro avant d'itérer sur les mots, et que vous incrémenterez à l'aide de la commande `expr` à chaque itération de la boucle sur les mots. Vous pourrez commencer par consulter la page de manuel de la commande `expr` et testez-la.

   Remarque : L'algorithme proposé ne traite pas le cas dans lequel une ligne contient moins de n mots. Dans ce cas, l'algorithme va simplement ignorer la ligne et ne rien afficher.

5. Modifier votre script pour tenir compte de la remarque précédente.

1.3 Duplication de flux

Le but de cet exercice est d'écrire un script qui prend en argument un fichier. Votre script doit lire les lignes provenant de l'entrée standard, et les écrire à la fois dans le fichier passé en argument et sur la sortie standard.

1. Dans un premier temps, nous allons traiter le cas où les arguments du script sont invalides. Les arguments sont invalides si (i) il n'y a pas un et un seul argument ou (ii) si l'unique argument est un chemin vers un répertoire existant (le test correspondant est `[ -d chemin ]`). Écrivez un script nommé `tee.sh` qui renvoie faux (valeur 1) après avoir affiché un message d'erreur adéquat si les arguments sont invalides, et qui renvoie vrai (valeur 0) sinon.
2. Modifiez votre script de façon (i) à lire l'entrée standard ligne à ligne et (ii) à afficher chaque ligne lue sur la sortie standard. On vous rappelle que pour lire l'entrée standard ligne à ligne, il faut utiliser la construction `while`.

   Remarque : Vous pouvez tester votre script de façon interactive en écrivant des lignes sur le terminal. Dans ce cas, vous pouvez fermer le flux associé à l'entrée standard avec la combinaison de touches `control-d` . Vous pouvez aussi tester votre script en redirigeant l'entrée standard à partir d'un fichier, par exemple avec `./tee.sh fic < /etc/passwd`.
3. Nous allons maintenant dupliquer la sortie vers le fichier passé en argument. Ouvrez un flux associé au fichier passé en argument au début du corps du script. Vous devez ouvrir le flux en écriture et en écrasement. Modifiez aussi votre boucle de façon à écrire chaque ligne lue à la fois sur la sortie standard (commande `echo ligne`) et dans le flux ouvert (commande `echo ligne >&...`).

1.4 Lignes alternées

Le but de cet exercice est d'écrire les lignes provenant de deux fichiers de façon alternée sur la sortie standard.

1. Dans un premier temps, nous allons traiter le cas où les arguments du script sont invalides. Les arguments sont invalides si (i) il n'y a pas deux arguments ou (ii) si l'un des deux arguments n'est pas chemin vers un chier existant (le test correspondant est `[ -f chemin ]`). Écrivez un script nommé `paste.sh` qui renvoie faux (valeur 1) après avoir affiché un message d'erreur adéquat si les arguments sont invalides, et qui renvoie vrai (valeur 0) sinon.
2. Votre script doit maintenant ouvrir deux ux en lecture : l'un associé au fichier passé en premier argument et l'autre associé au fichier passé en second argument.

3. Pour simplifier, nous supposons pour le moment que les deux fichiers ont le même nombre de lignes. Affichez chacune des lignes des deux fichiers en les alternant. Pour vous guider, vous devez écrire une boucle qui affiche une ligne de chaque fichier, et ceci tant qu'il reste des lignes à lire dans le premier fichier et dans le second. L'algorithme est donc le suivant :

```
Tant que lecture premier flux renvoie vrai et (opérateur &&)
lecture second flux renvoie vrai
    Écrit la ligne lue à partir du premier flux sur la sortie standard
    Écrit la ligne lue à partir du second flux sur la sortie standard
```

Testez votre script avec :

```
$ ./paste.sh ./paste.sh ./paste.sh
```

4. Nous allons maintenant gérer des fichiers de tailles différentes. Il faut commencer par transformer votre boucle en boucle infinie (boucle `while`). Dans votre boucle, commencez par lire une ligne du premier fichier :
   — si la lecture dans le premier fichier renvoie vrai, lisez une ligne du second fichier
     — si la lecture dans le second fichier renvoie vrai, affichez les lignes des premier et second fichiers
     — sinon, afficher la ligne du premier fichier
   — sinon (c.-à-d., la lecture sur le premier fichier renvoie faux) lisez une ligne du second fichier
     — si la lecture dans le second fichier renvoie vrai, affichez la ligne du second fichier
     — sinon, quittez votre script avec un `exit 0`

     Testez votre script avec :
     `$ ./paste.sh ./paste.sh /etc/passwd`

2 Une calculette

2.1 Calculette uni-opération

Dans cette première phase, il s'agit de réaliser un script shell qui permette de réaliser une addition, une soustraction, une multiplication ou une division sur l'ensemble des opérandes donnés en ligne de commande. Voici un exemple de résultat d'exécution attendu (les cinq derniers appels doivent être considérés par votre script comme incorrects et générer des messages d'erreurs) :

```
$./calculette_uni_operation.sh add 1 2 3
1 + 2 + 3 = 6
$./calculette_uni_operation.sh supp 6 4
6 - 4 = 2
$./calculette_uni_operation.sh mult 2 4 3
2 * 4 * 3 = 24
$./calculette_uni_operation.sh div 66 11 3 3
66 / 11 / 3 / 3 = 0
$$
$./calculette_uni_operation.sh
Opération manquante
```

```
$./calculette_uni_operation.sh add
Opérande manquant
$./calculette_uni_operation.sh ad 1 2 3
Opération inconnue
$./calculette_uni_operation.sh 1 2 3
Opération inconnue
$./calculette_uni_operation.sh div 6 2 0
6 / 2 / 0 : Division par 0 interdite
$
```

1. Supposez que l'appel à votre script est correct et traitez le cas de la somme. Pour vous aider à débuter, voici le pseudo code du script shell attendu :

   ```
   #! /bin/sh
   # déclaration d'une variable res dans laquelle le résultat sera accumulé,
   initialisée avec le deuxième argument de la ligne
   # si le premier argument est égal à "add" alors
   # décalage de deux arguments puisqu'on a traité l'opérateur et le premier opérande
   # pour chaque argument suivant
   # res=res+argument courant
   # affichage de l'opération effectuée
   #
   # affichage du résultat
   ```

   Attention : *res* ne doit pas être égal à la chaîne de caractère $< res{+}argumentcourant >$, mais le résultat de l'évaluation de cette commande. On vous rappelle donc que, comme vu en cours, vous pouvez a ecter une variable à la sortie d'une commande avec la construction bash suivante :

   ```
   var=$(cmd arg1 arg2...)
   ```

   Par exemple, `x=$(expr 1 + 2)` affectera la valeur 3 à la variable x.
2. Mettez en place un schéma alternatif afin de traiter les différents opérateurs. Attention à protéger correctement le caractère spécial '*' dans l'évaluation de la multiplication.
3. Traitez le cas de la division par zéro.
4. À présent, traitons les cas d'erreurs. Voici le pseudo code du script attendu :

   ```
   #! /bin/sh
   #si le nombre d'arguments est égal à 0# affichage "Opération manquante"
   # sortie du script
   #sinon
   # si le nombre d'arguments est égal à 1
   #   affichage "Opérande manquant"
   #   sortie du script
   # sinon
   #   res=deuxième argument de la ligne
   #   affichage
   #   si le premier argument est égal à
   #       "add" :
   #           décalage de deux arguments puisqu'on a traité l'opérateur et le premier opé
   ```

```
#           pour chaque argument suivant
#           res=res+argument courant
#   affichage de l'opération effectuée
#       "supp" :
#           ...
#       "mult" :
#               ...
#       "div" :
#           si operande = 0
#               affichage " / 0 : Division par 0 interdite"
#           sinon
#               ...
#       autre cas :
#           affichage "Operation non traitée"
#           sortie du script
# affichage du resultat
```

2.2 Calculette multi-opérations sans priorité

À présent, nous vous proposons de réaliser une calculette qui permet d'appliquer les opérations arithmétiques dans l'ordre où elles sont écrites, i.e. sans tenir compte des priorités sur les opérateurs. Voici un exemple de résultat d'exécution attendu :

```
$./calculette_sans_priorite.sh 1 + 2 + 3
1 + 2 + 3 = 6
$./calculette_sans_priorité.sh 6 - 4
6 - 4 = 2
$./calculette_sans_priorité.sh 2 x 4 + 3
2 * 4 + 3 = 11
$./calculette_sans_priorite.sh 3 + 2 x 4
3 + 2 * 4 = 20
$./calculette_sans_priorite.sh 3 + 2 x 4 / 2
3 + 2 * 4 / 2 = 10
$
$
$./calculette_sans_priorite.shOpérande manquant
$./calculette_sans_priorite.sh 1 +2
1 +2 -> Opération non traitée
$./calculette_sans_priorite.sh 1 / 0
1 /0 -> Division par 0 interdite
$
```

Remarque : Afin de pas à se soucier de l'interprétation du symbole ' * ' en tant qu'opérateur de multiplication de notre script shell, utilisez le caractère "x" à la place.

Attention à bien séparer vos arguments par des espaces.

1. Dans un nouveau script shell, abordez le problème sans vous soucier des cas d'erreurs. Pour vous aider, voici le pseudo code du script shell attendu :

```
#! /bin/sh
#
#res=premier argument
#affichage
#décalage
#
#tant qu'il reste des arguments
# si l'argument courant est égal à
#   + :
#       res=res+ argument qui suit
#       affichage
#   - :
#        ...
#   x :
#       ...
#   / :
#       si argument qui suit = 0, affichage "Division par 0 interdite"
# décalage
#
# affichage du résultat
```

2. Traitez les cas d'erreurs.

2.3 Calculette multi-opération avec priorité

En vous basant sur le script précédent, faites en sorte de prendre en compte les priorités qu'imposent les opérateurs arithmétiques, et ce, sans parenthèse.

La multiplication et la division sont prioritaires sur l'addition et la soustraction ; que ce soit dans une suite d'additions et de soustractions ou dans une suite de multiplications et de divisions, on effectue les calculs dans l'ordre d'écriture (c'est-à-dire de manière similaire à la question précédente). Ainsi, le challenge réside dans la différenciation des opérations selon leur priorité.

Pour vous aider, à chaque fois que vous rencontrez un opérateur non prioritaire, il faut que vous détectiez si l'opération suivante doit être évaluée avant. Plusieurs opérations prioritaires pouvant se suivre, cette vérification + évaluation doit être faite au sein d'une boucle qui prend fin lorsque l'opérateur rencontré n'est ni un opérateur de multiplication ni un opérateur de division.

3 Tri sans `sort`

On se donne un fichier texte dont le contenu est de la forme :

```
ARZALIER;ALEXANDRE;16;4;14;19
BERGERON;ALEXANDRE;8;7;17;12
DARMOISE;ALEXIS;16;9;3;7
DAUSQUE;ANGELO;18;18;17;4
DE JESUS;ANTOINE;18;14;8;10
DUCLOS;BASTIEN;13;13;9;2
FELIX;HANANE;13;2;5;5
GARY;JEREMIE;7;7;20;18
```

```
GRALL;KILLIAN;3;14;3;15
KEREBEL;LOGAN;8;4;11;14
LAURORA;LOUIS;0;9;15;18
MARTIN;MAELIS;12;13;12;12
MICHEL;MATHIEU;19;3;2;0
NASRI;MATHIS;16;13;18;17
RAKOWSKI;NELLOU;14;20;6;15
RICHARD;QUENTIN;9;20;5;18
RODAT;RAPHAEL;15;0;11;6
ROUX;RICHARD;11;1;16;15
SCHALCKENS;VALENTINE;10;11;0;6
SEIGNOBOS;VINCENT;3;11;19;13
TREMOUREUX;YAEL;2;13;6;20
VINCENT--RENARD;ZACHARIE;17;4;19;1
KEREBEL;ALEXANDRE;18;15;6;11
```

On souhaite trier ce fichier sans utiliser la commande `sort`.

3.1 Sens de tri

Proposez un script qui va trier ce fichier selon la première colonne. Le sens dans lequel sera effectué le tri sera passé en paramètre du script. `-up` pour le sens croissant et `-down` pour le sens décroissant.

3.2 Choix de la colonne

Adaptez le script précédent de façon à ce que la colonne prise en compte dans le tri soit passée en paramètre du script. `-0` pour la première colonne, `-1` pour la deuxième, etc.

3.3 Choix par défaut

Adaptez le script précédent de façon à ce que par défaut votre script trie le fichier selon la première colonne dans le sens croissant.

4 Calcul de moyenne

On considère un fichier ayant le même format que celui de la question précédente.
Ecrivez un script qui affiche le nom, prénom et la moyenne des 4 valeurs.