TP 2 – Programmation Shell (3h)

1. Scripts Shell

Le shell est un programme permettant d'interpréter et d'exécuter des commandes. Il existe des commandes « built-in » et des commandes qui sont des programmes binaires indépendants du shell. Il existe différents shell. Les plus connus sont Bourne Again Shell (bash), une version améliorée du Bourne Shell original de UNIX, et le TC Shell (tcsh), une version améliorée du C Shell original de BSD-UNIX.
Lorsqu'il est lancé, le shell lit des commandes sur l'entrée standard, les interprète et exécute les commandes dans des processus séparés.

Un script Shell est un fichier contenant un programme écrit en Shell. Ce fichier doit avoir les droits d'exécution. Des commentaires peuvent être insérés n'importe où en commençant par #.

La première ligne doit comporter un commentaire précisant le shell à utiliser.
On peut lancer un script shell en lui passant des arguments. Les arguments sont séparés par des espaces. Dans ce cas,
$0 est le nom de la commande
$1 le premier argument, $2 le deuxième....
$# le nombre d'arguments.
$@ l'ensemble de tous les arguments

Télécharger sur Moodle l'archive archive_exemples.zip et éditez le script affArg.sh. Ce script vérifie que le nombre d'arguments est 2 et les affiche.

Pour exécuter ce fichier, **pensez** à rendre le fichier exécutable par la commande :

```
chmod +x affArg.sh
```

Puis, pour exécuter ce fichier, **tapez dans une console :**

```
./affArg.sh toto titi
```

Les explications des instructions utilisées ainsi que la différence entre les simples quotes et les doubles quotes sont données dans les paragraphes suivants.

```
#!/bin/bash
#########################################################################
# @file     affArg.sh
# @author   Ph Lefebvre
# @version  1.0
# @date           01/01/2025
#
# Affiche les 2 premiers arguments
#########################################################################

# Test de présence de l'argument
if [ $# -ne 2 ]; then
  echo "Bad number of args !"
  echo "Usage : ./affArg.sh arg1 arg2"
  exit 1
fi
echo "Double quote : nom du programme : $0" $0
echo 'Simble quote argument 1 $1' $1
echo tous les arguments $@
```

Pour **debugger** un script on peut appeler le shell explicitement avec l'option -x :

```
bash -x  affArg.sh
```

2. Les variables shell

Les variables ne sont pas typées. Elles contiennent des chaînes de caractères. L'affectation se fait avec le signe =
Exemple : `NOM="toto"`

On accède au contenu par le signe $ Exemple : `echo $NOM`

Le shell évalue les expressions puis les exécute. Pour empêcher l'évaluation, on peut utiliser des guillemets.
Les doubles quotes " empêche l'évaluation des commandes mais pas des variables.
Les simples quotes ' empêche toute évaluation.

Les quotes inversés ` permettent l'évaluation et l'affectation du résultat à une variable. Ils sont similaires à $( )
Notez dans l'exemple suivant le point virgule pour séparer les commandes alors que l'espace sépare les arguments.

Essayez directement dans une console : (la commande *date* affiche la date courante)

```
NOM=toto
MAVAR='echo $NOM ; date' ; echo $MAVAR
MAVAR="echo $NOM ; date" ; echo $MAVAR
MAVAR=`echo $NOM ; date` ; echo $MAVAR
MAVAR=$(echo $NOM ; date) ; echo $MAVAR
MAVAR=$NOM".txt" ; echo $MAVAR
```

Les processus fils peuvent hériter des variables dites d'environnement. Pour transformer une variable en variable d'environnement on utilise la commande *export* . Pour lister les variables d'environnement on utilise la commande *env*.

Essayez :

```
env
```

Repérez la variable PATH. **A qui sert-elle ?**

Modifiez la variable PS1. **A quoi sert-elle ?**

Essayez (la commande *ps -f* affiche la liste des processus lancés par le shell) :

```
export MAVAR
env
bash
ps -f
echo $MAVAR
echo $NOM
```

tapez `[CTRL] [D]` pour sortir du shell fils

Le fichier ~/.bashrc , écrit en langage shell, est exécuté au lancement d'un nouveau shell.
Modifier ce fichier afin :

- d'afficher « « Bienvenue » à chaque démarrage de nouveau shell
- modifier la variable PATH afin que le repertoire courant « . » soit inclus

3. Les caractères spéciaux :

? dit au shell de générer une liste de fichiers en remplaçant le ? par n'importe quel autre caractère.

Ex : `ls tot?` # liste tous les fichiers dont le nom fait 4 lettres et commence par tot

***** comme ? mais remplace soit 0 soit plusieurs caractères.

Ex : `ls tot*` # liste tous les fichiers dont le nom commence par tot

`ls *` # liste tous les fichiers sauf ceux commençant par un point (fichiers cachés)

4. Opérations sur les variables :

${variable:n:m} permet d'extraire *m* caractères de la variable depuis le caractère *n*
${variable#motif} enlève de la valeur de la variable la chaîne initiale la plus courte qui correspond au motif. Le caractère « # » permet de travailler sur le préfixe des variables.
${variable##motif} sert à éliminer le plus long préfixe
{variable%motif} et ${variable%%motif} correspondent au contenu de la variable indiquée, qui est débarrassée, respectivement, du plus court et du plus long suffixe correspondant au motif transmis.

Exemple :

```
var=fichier.txt
echo ${var:1:3}          → ich
echo ${var%\.txt}        → fichier
echo ${var#fichier}      → .txt
```

5. Les redirections d'entrée/sortie

Le shell autorise les redirections d'entrée / sortie des processus.

>	sortie redirigée vers un nouveau fichier
>>	sortie redirigée en ajout.
2>	sortie erreur redirigée vers un nouveau fichier
<	entrée est prise à pTitre principalartir d'un fichier

`<< EOF` l'entrée reste « stdin » mais la commande se termine lorsqu'elle rencontre la ligne contenant les trois caractères « EOF »

`commande1 | commande2` Avec le caractère | (pipe), la sortie de commande1 est redirigée sur l'entrée de commande 2.

Essayez :

```
ls /tmp > /tmp/fichier_tmp
cat /tmp/fichier_tmp
```

Aidez vous du manuel `man` pour **expliquer** ce que font :

```
ls /etc/init.d | wc
echo -e "Robert:0567\nJuan:5359" |cut -d":" -f2
```

cut est une commande qui va découper chaque ligne de son entrée standard en champs. Chaque champs est repéré par un délimiteur spécifié par l'option -d. Les champs sont numérotés et l'affichage d'un champ spécifique se fait avec l'option -f.

6. L'instruction for : parcours de liste

Syntaxe : **`for`** *`index`* **`in`** *`liste ;`* **`do`** *`commandes ;`* **`done`**

L'exemple suivant affiche tous les mots d'une chaîne de caractère. Le séparateur des éléments d'une liste est l'espace. Notez les « ; » qui séparent les instructions lors de la saisie directement en ligne de commande.

```
$ phrase="O fortuna, velut luna, statu variabilis"
$ for i in $phrase ; do echo $i ; done
```

Le script suivant affiche tous les mots d'un texte dont le nom est passé en argument :

```
#!/bin/bash
for i in `cat $1`
do
    echo $i
done
```

Écrire un script qui renomme tous les fichiers d'un répertoire en ajoutant « _old » au nom du fichier. Le nom du répertoire sera passé en argument du script. La commande pour renommer est « mv ».

Remarque : Ce n'est pas demandé, mais analyser la chaîne de caractères afin de concaténer « _old » au nom et non à l'extension requiert l'utilisation de la commande expr.

7. L'instruction if et la commande test ou [ ]

Syntaxe :

```
if test-command
then
        commands
elif test-command
then
        commands
…
else
        commands
fi
```

Les parties elif et else sont optionnelles.

`test-command` est une expression qui retourne un boolen et qui peut être écrit avec la commande **`test`** ou dans sa forme condensée avec les **`[ ]`**. ***attention aux espaces autour des crochets et du signe = .***
Les plus usuelles sont (*cf. man*):

[*exp1* **`=`** *exp2*] vrai si *exp1* = *exp2* (exp1 et exp2 sont des chaînes de caractères)
[*exp1* **`-a`** *exp2*] vrai si *exp1* ET *exp2* sont vraies
[*exp1* **`-o`** *exp2*] vrai si *exp1* OU *exp2* est vrai
[**`!`** *exp*] vrai si *exp* est faux
[*exp1* **`-eq`** *exp2*] vrai si *exp1* == *exp2* (exp1 et exp2 sont des nombres)
[*exp1* **`-lt`** *exp2*] vrai si *exp1* < *exp2* (exp1 et exp2 sont des nombres)
[*exp1* **`-gt`** *exp2*] vrai si *exp1* > *exp2* (exp1 et exp2 sont des nombres)
[**`-e`** *exp*] vrai si le fichier de nom *exp* existe.
[**`-d`** *exp*] vrai si le répertoire de nom *exp* existe.

L'exemple suivant compare 3 phrases lues sur l'entrée standard. Notez les guillemets autour des variables permettant de comparer des phrases contenants des espaces.
La commande « `read mot` » retourne vrai tant qu'il y a des choses à lire sur l'entrée standard et place chaque ligne lue dans la variable *mot*.

```
#!/bin/bash
echo -n "word 1: "
read word1
echo -n "word 2: "
read word2
echo -n "word 3: "
read word3
if [ "$word1" = "$word2" -a "$word2" = "$word3" ] ; then
      echo "Match: words 1, 2, & 3"
elif [ "$word1" = "$word2" ] ; then
      echo "Match: words 1 & 2"
elif [ "$word1" = "$word3" ] ; then
      echo "Match: words 1 & 3"
elif [ "$word2" = "$word3" ] ; then
      echo "Match: words 2 & 3"
else
      echo "No match"
fi
```

Écrire un script qui affiche « Yesss » si le premier argument est « bravo ».
Écrire un script qui affiche « Yesss » si le fichier annuaire.txt existe.

8. L'instruction while

Syntaxe : **`while`** *`test-commande`* **`;`** **`do`** *`commandes`* **`;`** **`done`**

Le script suivant affiche les chiffres de 0 à 10. Vous noterez l'utilisation de (( )) pour évaluer une expression numérique. Notez également l'utilisation de *-n* pour éviter le retour à la ligne.

```
#!/bin/bash
number=0
while [ "$number" -lt 10 ]
do
      echo -n "$number"
      ((number +=1))
done
```

Écrire un script sans utiliser « wc » qui compte le nombre de lignes d'un ficher texte (utilisation de while, cat et read). Remarque, lors de l'utilisation du pipe « | » il y a lancement d'un nouveau shell. Les modifications de variables ne sont donc pas vues par le shell principal.
Pour réaliser ce script l'une des solutions est d'utiliser la commande cat dont on redirigera la sortie vers un script shell qui lira ligne par ligne avec la commande read. Pour grouper un ensemble de commandes on utilisera les accolades { } :
`cat $1` **`|`** **`{`** *`suite du script`* **`}`**

Comme on l'a vu au paragraphe précédent, read lit l'entrée standard ligne par ligne et retourne *true* tant qu'il y a des choses à lire.

9. L'instruction case

Syntaxe :

```
case
 test-string
in
pattern-1)
      commands-1
;;
pattern-2)
      commands-2
;;
. . .
esac
```

Le script suivant exécute « ps » si l'argument est 1, « ls » si l'argument est 2 ou 3 et affiche message « choix invalide » sinon. Notez le caractère * pour le choix par défaut et le caractère | pour une combinaison de choix.

```
#!/bin/bash
case "$1" in
1)
      ps
;;
2|3)
      ls
;;
*)
echo "choix invalide"
;;
esac
```

10. Les fonctions

Elles se déclarent comme ceci :

nom_Fonction( ) {
commandes ;
}Sous-titre

Les fonctions s'exécutent plus rapidement qu'un script stocké sur disque. De plus les fonctions s'exécutent dans le même script.

Exemple :

```
#!/bin/bash
affiche(){
   t=$1
   echo Voici "\$1 = $1"
   echo Voici "\$2 = $2"
   echo Voici "\$t = $t"
   echo Voici "le nb de parametres = $#"
}
affiche Le monde
echo "valeur de t = $t"
```

Décrivez ce que fait le script /etc/init.d/network-manager

11. Le filtre grep et les expressions régulières

grep est une commande qui recherche un motif et affiche toutes les lignes comportant ce motif. Par exemple

```
$ grep -E "ete" resultat
```

affiche les lignes du fichier *resultat* contenant la chaine "ete". Le premier paramètre est le motif et le deuxième le nom du fichier (ou l'entrée standard si omis).

Le motif est exprimé par une grammaire, appelée expression régulière contenant des caractères spéciaux (méta-caractères).

- **.** : correspond à un caractère quelconque sauf le retour à la ligne (de code ASCII 0x0D ou 0x0A)
- ***** : n'importe quel nombre d'occurrences du caractère précédent l'astérisque
- **+** : une occurrence ou plus de l'expression régulière qui précède ;
- **?** : zéro ou une occurrence de l'expression régulière qui précède ;

[...] : défini un ensemble de caractères possibles pour l'occurrence. On utilise un tiret pour définir un intervalle de caractères. **[^...]** défini un ensemble de caractères exclus pour l'occurrence.
^ : en première position d'une expression précise que l'expression doit se trouver en début de ligne.
$: en dernière position d'une expression précise que l'expression doit se trouver en fin de ligne.
{n} : le motif qui précède correspond exactement n fois ;
{n,} : le motif qui précède correspond n fois ou plus ;
{n,m} : le motif qui précède correspond n fois au mois mais pas plus que m fois ;
**** : protection du caractère qui suit, pour spécifier un méta-caractère comme caractère ;

| : OU logique ; le motif correspond soit à l'expression qui précède, soit à celle qui suit ;
( ) : permet de regrouper des expressions régulières ;

Si l'option **-E** n'est pas utilisée, les caractères **( ) { } | + ?** sont interprétés comme des caractères normaux.

L'option **-R** permet de parcourir l'ensemble des fichiers et sous répertoires d'une arborescence.
D'une manière générale l'option -r des commandes Linux permet souvent de parcourir de manière *récursive* l'ensemble d'un répertoire et des sous-répertoires comme le très dangereux « rm -rf ».

L'option **-o** permet d'afficher le motif plutôt que la ligne complète contenant le motif.

Exécutez la suite de commandes suivantes et interprétez à chaque étape ce que vous observez :

Vous aurez besoin d'un fichier nommé *resultat* qui pourra être :

```
texte 1
texte 2
separateur
4 texte 1
5 texte 2 blabla
```

```
$ grep " 2" resultat
$ grep -n " 2" resultat
$ grep -v sepa resultat # exclut les lignes avec "sepa".
$ grep "e.t" < resultat # le point représente n'importe quel caractère.
$ grep "e.*t" resultat # * répète le caractère précédant n fois (avec n>=0).
$ grep "t" resultat
$ grep "^t" resultat # lignes débutant par "t".
$ grep "2$" < resultat # lignes finissant par "2".
$ grep "[a-z]$" < resultat # lignes finissant par une lettre (donc pas un numéro).
$ grep -e "1" -e "2" resultat # lignes contenant 1 ou 2.
$ grep -E "1|2" resultat
$ echo "12 livres" >> resultat ; echo "10 kg" >> resultat
$ grep -E "^((0[1-9])|(10))" resultat # lignes commençant 01, 02..., 10
$ ls -l
$ ls -l | grep "tat"
$ echo "retrouver le motif suivant 4X05 avec grep" | grep -o -E '[0-9]+X[0-9]+'
```

Que fait ce script :

```
#!/bin/bash
for i in `cat $1`
    do echo $i
done >  /tmp/toto
grep -E "^([1-9]|(0[1-9])|([12][0-9])|(30)|(31))/" /tmp/toto | wc -l
rm /tmp/toto
```

Que fait ce script :

```
curl -s https://www.ensicaen.fr |grep -i IMG |grep -o -E  'src *= *\"[^\"]*\"'
```

Exécutez la liste des commandes suivantes

```
cd ~
mkdir tmp2 ; cd tmp2
mkdir tmp3 ; cd tmp3
ln -s ../../tmp2 lien_vers_tmp2
ll
echo "Bonjour les SATE" > fichier.txt
echo "Hello the SATE" >> fichier.txt
cd ~/tmp2
grep -R "SATE"
```

- **A quoi** sert la commande ln ?
- **Pourquoi** grep affiche un avertissement concernant des répertoires récursifs ?
- **Écrivez** une ligne de commande qui affiche les mots contenant 2 lettres « a » et pas de lettre « x »

12. Commande find

La commande find recherche un fichier dans un dossier et ses sous-dossiers en appliquant l'expression donnée. Essayez les commandes suivantes pour vous familiariser avec celle-ci :

```
$ find . -name "*.txt" # noms des fichiers se terminant par txt.
$ find ~ -type d # les répertoires à la racine de votre HOME directory
$ find . -name "*.c" -o -type d # -o pour faire un OU.
$ find  -name "*.c" -o -name "r*" # se terminant par c OU commençant par un r.
$ find . -perm 755 # fichiers dont les droits sont positionnés à 755.
$ find . -name "*.c" -exec ls -l {} \; exécute la commande ls sur tous les fichiers
trouvés
$ for i in `find /doc -name "*.py"` ; do grep -l -s "dict" $i ; done
    commande qui recherche le mot «dict» dans tous les fichiers «*.py» du
    répertoire /doc et ses sous-répertoires et affiche les noms des fichiers trouvés.
    (utilisation de l'option -l de grep)
```

Écrivez

- une commande qui recherche tous les fichiers « .o » d'une arborescence et les efface. (attention, dangereux !!)
- une commande qui recherche le mot « scanf » dans tous les fichiers « *.c » d'une arborescence.

13. Stream Editor : sed

sed est une commande permettant d'analyser et de modifier un texte. Pour des manipulations complexes on préférera la commande ***awk.***
D'une manière générale, sed recherche un motif, puis peut le supprimer ou le remplacer. La commande est passée par l'option ***-e***. Les commandes usuelles sont :

p : pour afficher (souvent utilisé avec -n pour éviter l'affichage en double)
d : pour supprimer
a : pour ajouter
s/regexp/replacement/g : pour substituer. **regexp** est le motif recherché, exprimé sous forme d'expression régulière (à utiliser avec l'option ***-r***). ***replacement*** est le motif de remplacement. ***g*** veut dire de manière globale (toutes les occurrences pour chaque ligne). Sinon, le remplacement ne se fait que sur la première occurrence de chaque ligne.
Si l'on spécifie un chiffre, sed remplace la n^{ième} occurrence.

Ces commandes peuvent être préfixées par 1 nombre ou 2 nombres séparées par une virgule pour spécifier les numéros de ligne où exécuter la commande. Par exemple « 12,23 » signifie entre les lignes de 12 à 23. Le caractère $ est utilisé pour spécifier la dernière ligne.
On peut aussi préfixer par des expressions régulières séparées par une virgule et encadrées par / pour spécifier un intervalle où appliquer la commande.
On peut également donner un motif sous forme d'expression régulière pour faire l'opération de suppression, affichage ou substitution uniquement sur les lignes correspondant à un motif.

```
$ sed -r -e '1,2d' annuaire.txt # supprime la 2ième ligne du fichier resultat.txt
$ sed -r -e '1atoto' annuaire.txt # ajoute une ligne contenant « toto » après la 1ière ligne.
$ sed -r -e 's/om/ame/1' annuaire.txt >> annuaire2.txt # remplace la première occurrence de « om » par « ame »
$ sed -r -e 's/om/ame/g' annuaire.txt # remplace toutes les occurrences de « om » par « ame »
$ sed -r -e '/regexp/ s/om/ame/g' annuaire.txt # remplace toutes les occurrences de « om » par « ame » si la ligne correspond à « regexp »

$ route -n | sed -n -e '2,/^0\.0\.0\.0/p'# Affiche de la ligne 2 à la ligne commençant 0.0.0.0. Notez l'utilisation du backslash pour éviter l'interprétation du point.
```

Exercices :

1. **Remplacer** tous les « : » du fichier annuaire.txt par des saut de ligne.
2. **(optionnel) Écrire** une commande qui affiche uniquement le bloc de lignes faisant référence à l'interface ethernet (en0, eth0 ou enp0s…) de la commande **ip add**.
3. **(optionnel) Écrire** une commande qui affiche l'adresse ip correspondant à l'interface ethernet.

Le fichier texte suivant est un fichier permettant d'importer des événements dans un calendrier type Google calendar. C'est par exemple le fichier que peut générer ADE. Les dates des événements (DTSTAMP, DTSTART, DTEND) sont notées au format local ou UTC. Dans le fichier ci-dessous, elles sont au format UTC car elles se terminent par 'Z'.

4. **Écrire une** commande qui supprime le Z à la fin des dates, sans supprimer les autres Z.

```
BEGIN:VCALENDAR
METHOD:REQUEST
PRODID:-//ADE/version 6.0.Z1
VERSION:2.0
CALSCALE:GREGORIAN
BEGIN:VEVENT
DTSTAMP:20191009T074209Z
DTSTART:20200106T120000Z
DTEND:20200106T130000Z
SUMMARY:CTP réseaux_s2
LOCATION:Salle Licence pro
DESCRIPTION:\n\nGRP_INF312TP06 Z3102:Services réseaux\nTP3.2\nLEFEBVRE Ph
 ilippe\n(Exported :09/10/2019 09:42)\n
UID:ADE60323031392d323032302d34313738352d302d30
CREATED:19700101T000000Z
LAST-MODIFIED:20191009T074209Z
SEQUENCE:1928234929
END:VEVENT
BEGIN:VEVENT
DTSTAMP:20191009T074209Z
DTSTART:20191212T090000Z
DTEND:20191212T110000Z
SUMMARY:TP réseaux_s46
LOCATION:Salle TP Réseaux
DESCRIPTION:\n\nGRP_INF312TP05 Z3102:Services réseaux\nTP3.1\nLEFEBVRE Ph
 ilippe\n(Exported :09/10/2019 09:42)\n
UID:ADE60323031392d323032302d36393036382d302d34
CREATED:19700101T000000Z
LAST-MODIFIED:20191009T074209Z
SEQUENCE:1928234929
END:VEVENT
BEGIN:VEVENT
DTSTAMP:20191009T074209Z
DTSTART:20191018T150000Z
DTEND:20191018T170000Z
SUMMARY:TP Programmation Réseau en C_s42
LOCATION:Salle TP Réseaux
DESCRIPTION:\n\nGRP_INF311TP02 Z3101:Systèmes d'Exploit.\nTP1.2\nLEFEBVRE
  Philippe\n(Exported :09/10/2019 09:42)\n
UID:ADE60323031392d323032302d35353934392d302d30
CREATED:19700101T000000Z
LAST-MODIFIED:20191009T074209Z
SEQUENCE:1928234929
END:VEVENT
END:VCALENDAR
```

5. Compteur de mots (Extrait du support de S. Fourey)

Écrire un script Shell permettant d'afficher la fréquence d'apparition des mots d'un fichier texte donné en argument. Une trace d'exécution pourra être de la forme :

```
$ ./compteur_mots.sh fichier.txt
85
12 #
7 les
6 #+
4 fichier
4 de
```

Astuce : Ce résultat est obtenu plus facilement pour un fichier dans lequel il n'y a qu'un mot par ligne.
Commandes utiles : sort, uniq et tr.