Partie III – Programmation shell avancée sous Linux

1. Les variables

1.1 Manipulation de variables

Une variable est un paramètre dont la valeur peut varier pendant l'exécution d'un programme. On utilise un nom pour repérer la valeur que représente ce paramètre. L'interpréteur de commandes distingue trois types de variables : les variables utilisateur, les variables système et les variables spéciales.

`env` : cette commande permet d'afficher la liste des variables utilisateur et système (leur nom et leur valeur)

Déclarer une variable

Une variable est déclarée et définie selon la syntaxe suivante :

```
$ nom=valeur
```

- Pas d'espace avant et après le symbole d'affectation **=**. L'élément `valeur` doit être encadré par des caractères de cittation (guillements doubles ou simples) s'il est composé de plusieurs mots séparés par des espaces ou s'il contient des caractères spéciaux.
- La variable existe dès qu'on lui affecte une valeur.
- Si aucune valeur n'est fournie, la variable reçoit une chaîne vide :

```
$ var=''
$ var=              (ces deux lignes sont équivalentes)
```

La valeur d'une variable peut contenir des caractères spéciaux. Dans ce cas, il faut déspécialiser les caractères spéciaux soit un par un, soit mettre toute la chaîne entre guillemets (simples ou doubles).

Accéder au contenu d'une variable

Pour accéder au contenu d'une variable, on place le signe **$** devant son nom :

```
$ var=bonjour
$ echo $var
bonjour

$ var=19
$ echo $var
19

$ var=deux mots
mots : commande introuvable
$ echo $var
(la commande a échoué : rien n'a été enregistré dans la variable)

$ var=bonjour\ le\ monde
echo $var
bonjour le monde
(→ déspécialisations individuelles)

$ var='bonjour le monde'
$ echo $var
bonjour le monde
(→ déspécialisation collective)

$ dir=~/Desktop/travail
$ cd $dir
$ ls $dir
```

```
…
$ cd $dir/cours/unix
$ pwd
/home/jdupont/Desktop/travail/cours/unix

$ a=bonjour
$ echo '$a le monde'
$a le monde
$ echo ''$a le monde''
bonjour le monde

$ echo ''bonjour ' le monde''
bonjour ' le monde
$ echo 'bonjour '' le monde'
bonjour '' le monde
(→ les guillemets simples et doubles se déspécialisent mutuellement)
```

On peut compléter le contenu d'une variable contenant une chaîne de caractères en utilisant la syntaxe suivante :

```
$ nom=$nom'valeur_ajoutee'
```

Exemples :

```
$ x=bonjour
$ echo $x
bonjour
$ x=$x' le monde'
$ echo $x
bonjour le monde
```

Pour distinguer le nom d'une variable d'une valeur ajoutée, on utilise les accolades : **`${nom}`**

```
$ ls
tp1
(On veut copier le fichiers 'tp1' en renommant la copie 'tp2')
$ base=tp
$ cp $base1 $base2
cp: opérande fichier manquant
Saisissez « cp --help » pour plus d'informations.
(un message d'erreur s'affiche car après le signe '$', les noms 'base1' et 'base2' sont
interprétés comme des noms de variables, au lieu de 'base' seulement)
$ cp ${base}1 ${base}2
équivaut à
$ cp tp1 tp2
```

Portée d'une variable

Une variable créée dans le shell courant n'est pas accessible à partir des sous-processus du shell ; elle est détruite à la fin de l'exécution du shell (à sa fermeture).

`export` : cette commande permet d'exporter une variable dans les shells et scripts autres que le shell courant.

```
$ export x=3
$ x=3 ; export x ;
(ces deux lignes sont équivalentes)
```

Supprimer une variable

`unset` : cette commande permet de supprimer une variable.

```
$ unset x base a dir var
```

`readonly` : cette commande permet de protéger une variable en écriture et contre sa suppression

```
$ x=jdupont
$ readonly x
$ x=kdurand
x: is read only
$ unset x
x: is read only
```

1.2 Tests sur les variables

Avant d'utiliser une variable, il convient de tester son existence et sa valeur. Le shell offre pour cela une syntaxe particulière qui permet éventuellement de lui attribuer un contenu par défaut selon certaines conditions :

```
${variable:<opérateur>remplacement}
```

Opérateur	Signification
`${a:-remplacement}`	Si la variable `a` n'existe pas ou si elle est indéfinie, son contenu est remplacé par le texte `remplacement` mais la variable ne prend pas pour valeur le texte de remplacement. Sinon, elle garde sa valeur initiale.
`${a:=remplacement}`	Si la variable `a` n'existe pas ou si elle est indéfinie, son contenu est remplacé par le texte `remplacement` et la variable prendre pour valeur ce texte. Sinon, elle garde sa valeur initiale.
`${a:+remplacement}`	Si la variable `a` existe et est définie, son contenu est remplacé par le texte `remplacement` mais la variable ne prend pas pour valeur le texte de remplacement. Sinon, son contenu est remplacé par une chaine vide.
`${a:?remplacement}`	Si la variable `a` n'existe pas ou si elle est indéfinie, le script est interrompu et le message `remplacement` s'affiche. Si aucun texte `remplacement` n'est fourni, un message d'erreur standard est affiché.

```
$ x=
$ echo ${x:-"le"}
le
(la variable existe mais elle est vide, le texte "le" est affiché à la place)
$ echo $x

(la variable 'x' n'a toujours pas de valeur)

$ x=
$ echo ${x:="le"}
le
(la variable existe mais elle est vide, le texte "le" est affiché à la place)
$ echo $x
le
(la variable x a pris pour valeur le texte de remplacement)

$ x=bonjour
$ echo ${x:="le"}
bonjour
(la variable existe et n'est pas vide, le texte "le" est affiché à la place)
```

```
$ x=bonjour
$ echo ${x:+"le"}
le
(la variable x existe et est définie, le texte "le" est affiché à la place)
$ echo $x
bonjour
(mais la variable garde sa valeur initiale)

$ x=
$ echo ${x:+"le"}

(la variable x existe mais elle est vide, une chaîne vide est affichée à la place)
$ echo $x

(la variable x n'a toujours pas de valeur)

$ unset x
$ echo ${x:?Erreur, variable inexistante ou indéfinie}
bash: x:  Erreur, variable inexistante ou indéfinie

$ unset x
$ echo ${x:?}
bash: x : paramètre vide ou non défini
(aucun message d'erreur n'a été défini par l'utilisateur après le signe '?', un message
d'erreur est affiché la place)
```

On peut obtenir la longueur d'une variable contenant une chaîne de caractères avec le **#** :

```
echo ${#nom_variable}
```

```
$ x=bonjour
$ echo ${#x}
7
```

1.3 Variables utilisateur

Valeurs numériques

`typeset -i` : cette commande permet de typer une variable en entier (*integer*). Il devient alors possible de faire des calculs et des comparaisons avec la variable en utilisant des opérateurs spécifiques aux valeurs numériques.

```
$ typeset -i sum
$ sum=1+1
$ echo $sum
2
```

Tableaux

Un tableau peut être déclaré de deux manières :

```
$ nom=(element1 element2 element3)
(ou)
$ set -a nom element1 element2 element3
```

Pour accéder à un élément du tableau, on utilise les crochets **[]** (à savoir : une case de tableau non initialisée est vide) :

```
$ nom=(bonjour le monde)
$ nom[0]="bonjour"
$ nom[1]=" le"
$ nom[2]=" monde"
$ echo ${nom[0]}"_"${nom[1]}"_"${nom[2]}
bonjour_ le_ monde
```

Si l'indice de l'élément auquel on veut accéder est stocké dans une variable, on ne place pas le `$` devant son nom :

```
$ i=1
$ echo ${nom[i]}
 le
```

Pour afficher tous les éléments du tableau, on utilise le caractère ***** :

```
$ echo ${nom[*]}
bonjour le monde
```

Pour obtenir la taille du tableau, on applique le caractère **#** à `nom[*]`, la liste des éléments du tableau (les indices commencent à 0) :

```
$ echo ${#nom[*]}
3
```

Pour obtenir la longueur d'un élément du tableau :

```
$ echo ${#nom[0]}
7            (longueur de 'bonjour')
```

1.4 Variables système

Les variables système (aussi appelées « variables d'environnement ») sont des variables qui sont définies dans l'environnement de l'interpréteur de commandes. Ces variables contiennent des informations qui sont utilisées par l'interpréteur de commandes ou par des commandes lancées à partir de celui-ci.

Lister les variables système

`set` : cette commande affiche la liste des variables connues du shell en cours

`env` : cette commande affiche la liste des variables connues du shell en cours et qui sont exportées

Les principales variables système

HOME : contient le chemin absolu du répertoire personnel de l'utilisateur connecté. Il est préférable de ne pas changer sa valeur.

PATH : contient une liste ordonnée des répertoires que le shell parcourt lorsqu'une commande est lancée pour trouver les binaires correspondants (voir la Partie I – 1.3 Interprétation des commandes par le shell pour un exemple).

LDPATH : contient une liste ordonnée des répertoires que l'éditeur de liens dynamiques parcourt à la recherche de bibliothèques partagées (on étudiera cette variable plus en détails dans les prochaines séances)

MANPATH : contient une liste ordonnée des répertoires que le shell parcourt à la recherche de pages de manuel.

PWD : contient le chemin absolu du répertoire dans lequel se trouve actuellement l'utilisateur. Sa valeur est mise à jour par l'interpréteur de commandes à chaque fois que l'utilisateur change de position dans l'arborescence.

OLDPWD : contient le chemin absolu du répertoire visité précédemment par l'utilisateur ; cette variable correspond au caractère − utilisé avec la commande `cd`

PS1 : contient la liste des informations qui sont affichées dans l'invite de commandes principale

`\u` le nom d'utilisateur (login)

`\h` le nom de la machine

`\w` le répertoire courant

PS2 : contient la liste des informations qui sont affichées dans l'invite de commandes secondaire qui s'affiche lorsque le shell attend la suite d'une commande.

LANG : (*language*) contient la définition de la langue à utiliser et le jeu de caractères (par exemple, `fr_FR.UTF-8` pour utiliser la langue française et l'encodage UTF-8).

LOGNAME : contient le nom d'utilisaeur de l'utilisateur connecté

USER : contient le nom du login utilisé par l'utilisateur lors de sa connexion

Exportation de variables

On distingue trois catégories de variables d'environnement selon les processus par lesquels elles sont utilisées :

- les variables utilisées uniquement par le shell (par exemple `PS1` ET `PS2`),
- les variables qui sont utilisées des commandes et parfois par le shell (par exemple `PATH`)
- et les variables qui sont utilisées par une commande précise (par exemple `EXINIT` qui est utilisée par `vi`)

Les variables qui peuvent être utilisées par d'autres processus que le shell courant (d'autres shells, par exemple) doivent être exportées pour leur être transmises. Il s'agit des variables système suivantes : `PATH`, `PWD`, `HOME`, `LOGNAME` et `TERM`.

1.5 Variables spéciales

Les variables spéciales sont des variables prédéfinies par le shell et qui peuvent être référencées par l'utilisateur dans les scripts.

Variable	Contenu
`$?`	Contient le code retour de la dernière commande exécutée
`$$`	Contient le PID du shell actif
`$!`	Contient le PID du dernier processus lancé en arrière-plan
`$-`	Contient les options du shell

```
$ ls *.t
ls: impossible d'accéder à *.t: Aucun fichier ou dossier de ce type
$ echo $?
2

$ ls *s
…
$ echo $?
0
```

```
$ echo $$
35789
$ echo $!

(aucun processus n'a été lancé en arrière-plan depuis ce terminal)
$ echo $-
```

Exercice 1 (16)

1. Placez le chemin de votre répertoire en cours dans une variable nommée `monrep` puis listez son contenu en utilisant cette variable.

   Fermez votre terminal texte courant et ouvrez-en un nouveau (ou ouvrez un nouvel onglet dans le même terminal). Dans le nouveau terminal (ou le nouvel onglet) affichez le contenu de la variable `monrep` de la question 3. Que se passe-t-il ? Pourquoi ?

2. Créez 3 fichiers vides `exo1.c`, `exo2.c` et `exo3.c`. Créez ensuite une variable `base=exo`.

   Affichez maintenant la liste des fichiers dont le nom commence par `'exo'` et se termine par `'.c'`, en utilisant la variable `$base`.

   Copiez `exo1.c` en `exo4.c` en utilisant la variable `base` ?

3. Protégez la variable `base` de la question 5 précédente en écriture. Essayez ensuite de la supprimer.

   Affichez le code retour de la dernière commande.

4. Déclarez une variable `a` de type entier et assignez-lui la valeur 101. Multipliez-la par elle-même et stockez le résultat dans une variable `mult`. Affichez le résultat.

5. Déclarez une variable de type tableau qui s'appelle `tab` et qui contient la liste suivante :

   `lundi mardi mercredi jeudi vendredi samedi dimanche.`

   Proposez une deuxième manière de déclarer le même tableau.

6. Affichez le contenu du tableau et sa taille au format suivant :

   `<taille>:(element1 element2 … elementN)`

7. Comment peut-on obtenir la longueur d'une variable déclarée ainsi : `a='bonjour le monde'` ?

8. Si une variable `a` est définie, qu'affiche la commande suivante : `${a:+"salut"}`. La valeur de `a` a-t-elle changé ?

9. Affichez la liste des variables d'environnement qui sont connues du shell courant et exportées.

10. Affichez le contenu des variables d'environnement suivantes et précisez ce qu'elles contiennent :

    `HOME`, `PATH`, `PWD`, `OLDPWD`, `LANG`, `LOGNAME` et `USER`.

11. Modifier la valeur de la variable d'environnement `HOME` sans exporter la variable. Ouvrez une nouvelle fenêtre de terminal et affichez-y le contenu de `HOME`. A-t-il changé ?

12. Affichez le contenu de la variable PS1. Que signifient les caractères suivants dans sa définition :

    a. `\u`

    b. `\h`

    c. `\w`

    d. `\a`

13. Définissez et testez les invites de commande principales suivantes :

    a. `$`

    b. (rien du tout)

    c. `<utilisateur>`

d. `<répertoire_courant> *`

e. `<utilisateur>, entrez une commande >`

14. Modifiez la variable d'environnement `TMOUT` pour que le shell se termine après 7 secondes d'inactivité.

15. Affichez le contenu de la variable d'environnement `HISTFILE`. Quel est son contenu ? Modifiez la variable `HISTSIZE` pour fixer la taille de l'historique du shell (en nombre de lignes) à 50000.

Affichez le contenu des fichiers `~/.bash_history`. Que contient-il ?

16. Que contient le fichier `~/.bash_logout` ?

2. Les structures de contrôle

2.1 Sélection : `if..then..else..fi`

```
if condition
then
      ...
fi
```

```
if condition ; then
      ...
fi
```

```
if condition
then
      ...
else
      ...
fi
```

```
if condition
then
      ...
elif condition
then
      ...
else
      ...
fi
```

Exemple :

```
$ a="b"
$ if [ $a = "b" ]  (les deux chaînes sont égales)
> then
> echo "vrai"
> elif [ $a = "c" ]
> then
> echo "faux"
> fi
vrai
```

2.2 Sélection : `case..esac`

```
case variable in
      modele1)
            ... ;;
      modele2)
            ... ;;
      modele3 | modele4 | modele5 )
            ... ;;
      *)
            action par défaut ;;
esac
```

La structure `case` permet des tests sur les différentes valeurs possibles d'une même variable. Elle est dans ce cas plus lisible qu'une structure `if...elif` équivalente.

Un modèle peut être un simple caractère ou une chaîne de caractères ou un motif contenant des caractères spéciaux. Le dernier modèle (`*`) est l'action par défaut si aucun des modèles précédents n'est vérifié. Ce dernier modèle est facultatif. Chaque modèle est suivi d'une suite de commandes qui est terminée par deux points-virgules : `;;`. Ces deux `;;` indiquent que la fin du traitement de la suite de commandes et permettent de sortir de la structure `case`. Le caractère `|` exprime l'alternative entre plusieurs modèles qui sont tous associés à la même suite de commandes.

Exemple :

```
$ a=5
$ case $a in
> 1) echo "faux";;              # si la valeur de 'a' est 1
> 5) echo "vrai";;              # si la valeur de 'a' est 5
> *) echo "nul";;               # sinon
> esac
```

```
vrai
```

Exemple :

```
$ a=bonjour
$ case $a in
> b*) echo "vrai";;       # si la valeur de 'a' commence par la lettre 'b'
> c*) echo "faux";;       # si la valeur de 'a' commence par la lettre 'c'
> *) echo "null";         # sinon
> esac
vrai
```

2.3 Itération : `for..do..done`

La boucle `for` permet de traiter une liste de valeurs (nombres, caractères, noms de fichiers, …) représentée ainsi : `val1 val2 val3 … valn`. La liste de valeurs peut être donnée telle quelle :

```
for var in val1 val2 val3 ... valn
do
      ...
done
```

ou à l'aide d'une variable contenant une liste d'éléments :

```
for var in $variable                    for elem in ${tab[*]}
do                                      do
      ...                                     ...
done                                    done
(variable simple)                       (tableau)
```

ou par substitution d'une commande qu'on place entre apostrophes inversées ` ` et dont le résultat est une liste :

```
for var in `commande`
do
      ...
done
```

ou à l'aide d'un motif représentant une liste de valeurs :

```
for var in *.c
do
      ...
done
```

ou par substitution de variable :

```
for var in $*
do
      ...
done
```

ou la liste de valeurs est implicite : `$*` dans un script contient la liste de ses paramètres :

```
for params
do
      ...
done
(dans un script, les deux derniers exemples sont équivalentes ; $* est la liste des
arguments en ligne de commande du script)
```

2.4 Itération : `while..do..done`

La boucle `while` permet d'exécuter un bloc de commandes « tant que » la condition est réalisée. Dès que la condition devient fausse, la boucle est terminée. Deux syntaxes peuvent être utilisées :

```
while condition
do
      ...
done
```

ou

```
while
      bloc de commandes représentant une condition
do
      ...
done
```

Exemple : pour demander à l'utilisateur d'entrer une nouvelle valeur tant que la valeur saisie est invalide

```
while
      echo "Entrez votre choix :"
      read str
      [ -z $str ]
do
      echo "Entrez une chaîne non vide"
done ; echo "votre choix est : "$str ;
```

Exemple : pour parcourir le contenu d'un fichier ligne par ligne

```
while read line
do
      echo $line
done < fichier.txt
```

```
cat fichier.txt | while read line
do
      echo $line
done < fichier.txt
```

2.5 Saut : `break` et `continue`

`break`

Cette commande est utilisée pour arrêter l'exécution d'une boucle prématurément dès qu'une certaine condition est vérifiée.

```
while true
do
      if condition
      then
            break
      fi
done
```

`continue`

Cette commande est utilisée pour refaire un tour de boucle en repartant du `do`.

```
while condition
do
      if condition
      then
            then continue
      fi
done
```

Exemple :

```
while true
do
      echo "Entrez votre login :"
      read lg

      # si la chaîne est vide, la boucle continue :
      [ -z $a ] && { echo 'Recommencez \ !' ; continue ; }

      # sinon elle termine
      break ;
done
```

Exercice 2 (9)

1. Écrivez une boucle `for` qui affiche les nombres de 1 à 11.
2. Écrivez une boucle `for` qui parcourt la liste des entrées du répertoire courant dont le nom se termine par `'.c'` et applique à chacune la commande `ls -l`.
3. Écrivez une structure `case` qui parcourt la liste des lignes du ficher `/etc/passwd` et qui affiche :

`root`	si la ligne concerne `root`
`le shell par défaut est sh`	si la ligne se termine par `/bin/sh`
`le shell par défaut est bash`	si la ligne se termine par `/bin/bash`
`compte verrouillé`	si la ligne se termine par `/bin/nologin`
`accès au shell interdite`	si la ligne se termine par `/bin/false`
`inconu`	dans tous les autres cas

4. Déclarez une variable de type tableau qui s'appelle `tab` et qui contient la liste suivante :

   `lundi mardi mercredi jeudi vendredi samedi dimanche.`

   Affichez le contenu du tableau ainsi, où `taille` est la longueur de l'élément avant le signe `:` (en nombre de caractères) :

   ```
   lundi:<taille>
   mardi:<taille>
   ...
   dimanche:<taille>
   ```

5. Utilisez une boucle `while` avec la commande `sort` dans un tube de commandes pour trier les lignes du fichier `/etc/passwd` par ordre numérique croissant des `UID`, et afficher les lignes dont la longueur est inférieure ou égale à 15 caractères.
6. Que font les commandes `true` et `false` ?
7. Que fait la boucle suivante :

```
$ for a in {5..100..5}; do echo $a; done
```

8. Utilisez une boucle `for` qui affiche l'alphabet en minuscules sans en utilisant l'intervalle `a..z`.
9. Avec la même syntaxe, affichez 10 fois la date à un intervalle de 2 secondes.

3. Les tests

3.1 Syntaxe

La commande `test`

Cette commande a deux syntaxes :

`test expression` : teste l'`expression` et renvoie 0 (vrai) ou 1 (faux). Son code retour se trouve dans **`$?`**

`[ expression ]` : les crochets doivent être séparés de leur contenu par un espace. Cette syntaxe est la plus utilisée

Les commande `[[ ]]`

Cette commande est une version enrichie de la commande `test`. Elle permet d'utiliser des opérateurs supplémentaireset de faire des tests sur une chaîne de caractères sans utiliser les guillemets (voir la section 3.3 plus bas).

3.2 Tests sur les fichiers

Tests sur l'existence

`-a fichier` : vrai si `fichier` existe

`-e fichier` : (*exist*) vrai si `fichier` existe

`-s fichier` : (*size*) vrai si `fichier` n'est pas vide

Tests sur le type

`-f fichier` : (*file*) vrai si `fichier` est un fichier ordinaire

`-d fichier` : (*directory*) vrai si `fichier` est un répertoire

`-h fichier` : (*hard link*) vrai si `fichier` est un lien physique

`-L fichier` : vrai si `fichier` est un lien symbolique

`-b fichier` : (*bloc*) vrai si `fichier` est de type bloc

`-c fichier` : (*character*) vrai si `fichier` est de type spécial caractère

`-p fichier` : (*pipe*) vrai si `fichier` est de type tube nommé

`-S fichier` : (*socket*) vrai si `fichier` est de type socket

Tests sur les permissions

`-r fichier` : (*read*) vrai si `fichier` est accessible en lecture

`-w fichier` : (*write*) vrai si `fichier` est accessible en écriture

`-x fichier` : (*execute*) vrai si `fichier` peut être exécuté

`-u fichier` : (*SetUID*) vrai si `fichier` a le droit `SUID`

`-g fichier` : (*SetUID*) vrai si `fichier` a le droit `SGID`

`-O fichier` : (*owner*) vrai si l'utilisateur est propriétaire de `fichier`

`-G fichier` : (*group*) vrai si l'utilisateur est membre du groupe propriétaire de `fichier`

Comparaisons

`f1 -nt f2` : (*newer than*) vrai si `f1` est plus récent que `f2`

`f1 -ot f2` : (*older than*) vrai si `f1` est plus ancien que `f2`

`f1 -ef f2` : (*equal file*) vrai si les blocs de données de `f1` sont les mêmes que ceux de `f2` (les deux noms de fichiers pointent sur le même inode)

3.3 Tests sur les chaînes de caractères

Lors d'un test sur une variable contenant une chaine de caractères, le shell remplace le nom de la variable par son contenu. Si la variable est vide (i.e., contient une chaine de longueur nulle), l'opérateur de test utilisé a pour argument... du vide ! Et la commande `test` affichera un message indiquant une erreur de syntaxe. Pour cette raison, il est recommandé de mettre le nom de la variable entre guillemets doubles : `"$chaine"`. Les guillemets permettent ainsi d'éventuellement représenter la chaîne comme une chaîne vide et non comme du vide.

`-z chaine` : (*zero*) vrai si `chaine` est de longueur nulle

`-n chaine` : (non zero) vrai si `chaine` est de longueur non nulle

`ch1 = ch2` : vrai si `ch1` et `ch2` sont égales

`ch1 != ch2` : vrai si `ch1` et `ch2` sont différentes

`chaine` : vrai si `chaine` est vide

```
$ chaine=bonjour
$ test -z "$chaine"
$ echo $?
1
(chaine n'est pas vide → le résultat du test est faux)

$ chaine=bonjour
$ test -n "$chaine"
$ echo $?
0
(chaine n'est pas vide → le résultat du test est vrai)

$ chaine=
$ test -z "$chaine"
$ echo $?
0
(chaine est vide → le résultat du test est vrai)
```

Les opérateurs supplémentaires de la commande `[[ ]]`

`ch = motif` : vrai si `ch` correspond à `motif`, un motif formé des caractères génériques vus en Partie II – 2.2 Caractères génériques.

`ch != motif`: vrai si `ch` ne correspond pas à `motif`

`ch1 > ch2` : vrai si `ch1` est supérieure à `ch2` dans l'ordre lexicographique

`ch1 < ch2` : vrai si `ch1` est inférieure à `ch2` dans l'ordre lexicographique

3.4 Tests sur les valeurs numériques

`a -eq b` : (*equal*) vrai si `a` est égal à `b`

`a -ne b` : (*not equal*) vrai si `a` est différent de `b`

`a -lt b` : (*lesser than*) vrai si `a` est strictement inférieur à `b`

`a -le b` : (*lesser or equal*) vrai si `a` est inférieur ou égal à `b`

`a -gt b` : (*greater than*) vrai si `a` est strictement supérieur à `b`

`a -ge b` : (*greater or equal*) vrai si `a` est supérieur ou égal à `b`

3.5 Opérateurs logiques

`!` : (*not*) négation

`-a` : (*and*) ET

`&&` : (*and*) ET ; remplace `-a` dans la commande `[[ ]]` (dans laquelle il ne fonctionne pas)

`-o` : (*or*) OU

`||` : (*or*) OU remplace `-o` dans la commande `[[ ]]` (dans laquelle il ne fonctionne pas)

`\(...\)` : caractères de regroupement permettant d'imposer un ordre d'évaluation des opérateurs

Exercice 3 (11)

1. Utilisez les commandes `ls` et `while` dans un tube pour afficher les entrées du répertoire `/dev` ainsi :

`dir: <nom>`	si la ligne est un fichier de type `d`
`char: <nom>`	si la ligne est un fichier de type `c`
`link: <nom>`	si la ligne est un fichier de type `l`
`char: <nom>`	si la ligne est un fichier de type `b`
`???: <nom>`	dans tous les autres cas

2. Comment savoir si deux fichiers sont des liens physiques qui pointent vers les mêmes blocs de données ?

3. Créez un fichier vide `fic` dans le répertoire courant. Supprimez le droit d'exécution sur ce fichier à tous les utilisateurs. Essayer maintenant de le supprimer. Utilisez une structure `if` pour afficher `ok` si la commande a réussi ou `ko` si elle a échoué. Indice : utilisez les variables spéciales du shell vues au Chapitre 1 – 1.3 Variables spéciales.

4. Écrire une commande qui demande à l'utilisateur d'entrer un nombre supérieur à 0 et inférieur ou égal à 100 et qui lui demande une nouvelle valeur tant que la valeur saisie est invalide.

5. Réécrivez le test suivant en utilisant la commande de test `[[ ]]` à la place de `[ ]` :

```
[ -f $f -a \( -e $f -o -l $f \) ]
```

6. En utilisant les commandes `while` et `if`, parcourez les lignes du fichier `/etc/passwd` et affichez :

`root`	si la ligne concerne `root`
`shell par défaut : sh`	si la ligne se termine par `/bin/sh`
`shell par défaut : bash`	si la ligne se termine par `/bin/bash`
`compte verrouillé`	si la ligne se termine par `/bin/nologin`
`accès au shell interdit`	si la ligne se termine par `/bin/false`
`inconu`	dans tous les autres cas

10. Que fait la commande suivante ? Que représente `IFS` ?

```
$ while IFS=: read -r c1 c2 c3 c4 c5 c6 c7
> do
>   echo $c1"-"$c2"-"$c3"-"$c4"-"$c5"-"$c6"-"$c7
> done < /etc/passwd
```

11. Utilisez la même syntaxe pour extraire les champs `login`, `UID` et `GID` de chaque utilisateur (pour des détails sur le format de ce fichier, reportez-vous à la Partie II – Exercice 3 – Question 15). Enregistrez le résultat dans un fichier `passwd.ext`.

4. L'arithmétique

4.1 Syntaxe

La commande expr

Cette commande évalue la valeur d'une expression et l'affiche. Elle peut être utilisée avec des expressions artihmétiques, pour comparer des valeurs et pour la recherche de motif dans une chaîne de caractères.

Les espaces autour de l'opérateur sont nécessaires. Sinon, l'expression est considérée comme un seul argument.

Les caractères de regroupement `\( \)` dans un motif permetent d'extraire la sous-chaîne qui correspond à la partie du motif qui est entourée par les parenthèses.

```
expr nombre1 operateur_math nombre2
expr variable1 operateur_comparaison variable2
expr chaine : motif
```

Exemples :

```
$ expr 7 + 3
10
$ expr 7+3
7+3

$ a=5; b=5; c=6;
$ expr $a =5
1
$ expr $a = $b
1
$ expr $a \> $c
0

$ expr $a = $b \& $a \< $c
1
$ expr $a = $b \& $a \> $c
0

$ a=gh57KlM78oix
$ expr "$a" : '[a-zA-Z]*[0-9]*[a-zA-Z]*[0-9]*'
9
(les 9 premiers caractères de $a correspondent au motif)
$ expr "$a" : '[a-zA-Z]*[0-9]*'
4
(les 4 premiers caractères de $a correspondent au motif)
$ expr "$a" : '\([a-z]*\)[0-9]*'
gh
```

La commande (( ))

Cette commande a des avantages sur `expr` :

- les espaces ne sont pas nécessaires autour des opérateurs : `((a=a+5))`
- les variables peuvent être utilisées sans le `$` : `((a=a+c))`
- les caractères spéciaux du shell n'ont donc pas besoin d'être déspécialisés : `((a*5))`
- son exécution est plus rapide

Syntaxe :

```
((expression_arithmétique))
```

Exemples :

```
$ x=50
$ ((x = $x + 40))
$ echo $x
90
$ (( x = $x + 10 ))
$ echo $x
100
$ ((x=$x+10))
$ echo $x
110
$ ((x=$x+5))
$ echo $x
115

$ a=5; b=5;
$ (( $a == $b )) && echo 'vrai'
vrai
$ (( ($a > 1) && ($b > 1) )) && echo 'vrai'
vrai

$ a=5; b=5;
$ echo $(( ! $a == $b ))
0
$ echo $(( $a == $b ))
1
$ echo $(( a == b ))
1
```

La commande let

Cette commande est équivalente à la précédente

```
let expression
((expression))
```

Exemples :

```
$ let "a = 5 * 7"
$ let "b = $a * 5"
$ echo $a $b
35 175
```

4.2 Opérateurs

Opérateurs artihmétiques

dans `expr`	**dans** `(( ))`	**signification**
+	+	addition
-	-	soustraction
*	*	multiplication (doit être déspécialisé ainsi * pour indiquer au shell qu'il ne s'agit pas d'un caractère générique)

dans `expr`	**dans** `(( ))`	**signification**
`/`	`/`	division
`%`	`%`	modulo

Opérateurs de comparaison

dans `expr`	**dans** `(( ))`	**signification**
`!=`	`!=`	différent : vaut 1 si les valeurs sont différentes, 0 sinon
`=`	`==`	égal : vaut 1 si les valeurs sont égales, 0 sinon
`\<`	`<`	Inférieur : vaut 1 si les valeurs sont différentes, 0 sinon
`\<=`	`<=`	Inférieur ou égal : vaut 1 si inférieures ou égales, 0 sinon
`\>`	`>`	Supérieur : vaut 1 si les valeurs sont différentes, 0 sinon
`\>=`	`>=`	Supérieur ou égal : vaut 1 si supérieures égales, 0 sinon

Opérateurs logiques

dans `expr`	**dans** `(( ))`	**signification**
`\&`	`&&`	ET, vaut 1 si les expressions sont vraies, 0 sinon
`|`	`||`	OU,
	`!`	!

Autres

dans `expr`	**dans** `(( ))`	**signification**
`-nombre`	`-nombre`	Opposé de `nombre`
`\(expression\)`	`(expression)`	Regroupement
	`=`	Assignation
`:`		Recherche d'un motif dans une chaîne de caractères

Remarque : La commande **:** renvoie toujours la valeur 0 (réussite). Elle est en cela équivalente à la commande `true`. Placée devant une autre commande, elle

4.3 Substitution d'expressions arithmétiques

$(( ))

La commande `(( ))` effectue un calcul seulement, elle n'affiche rien. Donc, l'utilisation d'un opérateur de substitution (par exemple `` `(( ))` ``, voir la Partie II – 2.3 Les caractères de substitution de commandes) pour afficher son résultat n'a pas de sens.

```
commande $((expression_arithmétique))
(la commande 'commande' prend pour argument le résultat de l'expression arithmétique)
```

Exemples :

```
$ a=5; ((a+3)); ((a=a*6)); echo $((a=a*6))
180
```

Exercice 4 (6)

1. Quelle est la différence entre les deux lignes de commandes suivantes ?

```
$ A=5; B=7; A=$((A+B))
$ A=5; B=7; typeset -i A; typeset -i B; A=A+B;
```

2. Initialisez un tableau avec les valeurs suivantes : `45 67 33 2 5 0 83`.

   Recherchez les indices des valeurs minimum et maximum dans le tableau.

   Calculez la somme des éléments du tableau.

3. Affichez les tables de multiplication de 1 à 11 sans stocker le résultat de chaque multiplication dans une variable (i.e., le résultat de la multiplication est calculé pendant la commande d'affichage). Indice : utilisez l'opérateur de substitution de commandes arithmétiques lors de l'affichage.

4. Que fait la suite de commandes suivante ?

```
$ a=2; r=$a;
$ for (( c=1; c<=10; c++ ))
> do
>   (( r = $r * $a ))
>    echo $c" : "$r
> done;
```

5. Que renvoie la commande suivante ? Expliquez son fonctionnement avec cet exemple.

```
$ expr Bonjour : ".*"
```

6. Expliquez le fonctionnement de la boucle suivante ?

```
$ while :
> do
>   echo "Bonjour \ !"
> done
```

   Écrivez une boucle équivalente en remplaçant `':'` par une autre commande.