TRAVAUX PRATIQUES 3
Introduction aux scripts Linux

L'objectif de ce TP est d'aborder la réalisation de programmes scripts exécutés l'interpréteur de commandes (ou shell) disponible sous les systèmes d'exploitation de type UNIX et permettant d'automatiser un certains nombre de traitements.
Une fois votre session ouverte sous un système d'exploitation de type UNIX, ouvrez un terminal pour entrer les commandes décrites dans l'énoncé. Dans la suite, le prompt du terminal sera symbolisé par « `$>` ».

EXERCICE 1

Dans cet exercice nous allons présenter divers éditeurs de texte permettant d'ouvrir et modifier des fichiers en mode texte à l'aide d'un terminal. Un éditeur de texte permet de taper du texte brut dans un fichier, contrairement un outil de traitement de texte qui permet de mettre en page du texte. Cet outil est par exemple utilisé pour taper des notes, ou encore taper du code source dans un langage de programmation. Vous aurez à l'utiliser dans la suite de ce TP afin de réaliser des scripts qui vous seront demandés.
Il existe beaucoup d'éditeurs de texte différents, des éditeurs simples possédant une interface graphique comme `gedit` et `kwrite` (qui sont très intuitifs et assez proches du bloc-notes sous Windows) et d'autres disposant de fonctionnalité plus puissantes comme `emacs` ou `vim`.
Nous allons nous intéresser aux deux derniers éditeurs cités et voir leur principales fonctionnalités, car ils sont disponibles sur la majorité des systèmes de type UNIX quelque soit le système de fenêtrage (interface graphique) disponible.

L'éditeur *Emacs*

Emacs est un éditeur de texte possédant des fonctionnalités supplémentaires comme lire des news, envoyer/recevoir des e-mails ou jouer à des jeux de société par exemple. Cet outil a été écrit par Richard M. Stallman qui est l'initiateur des logiciels libres (projet *GNU*) et de la licence *GPL* (*General Public Licence*), dont fait partie l'outil *Emacs* ou encore le compilateur *gcc*. *Emacs* est disponible pour les systèmes de type UNIX, mais également sous Windows.
Toutes les commandes de cet outil utilisent la touche `<CTRL>` associée à une autre touche du clavier, par exemple `<CTRL>-h` et abrégé `C-h` pour un appui simultané sur les touches `<CTRL>` et `h` et permettant d'obtenir de l'aide.

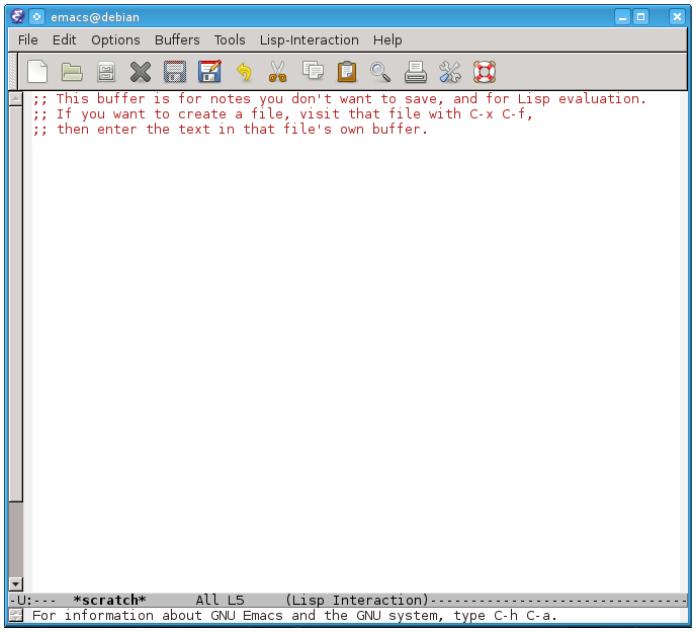
Pour lancer l'éditeur *Emacs* il suffit de taper la commande ci-dessous :

```
$> emacs
```

Pour ouvrir l'éditeur afin de travailler sur un fichier particulier il faut indiquer le nom du fichier en paramètre comme ci-dessous (le fichier sera créé s'il n'existe pas) :

```
$> emacs nom_fichier
```

Si vous avez un système de fenêtrage disponible, alors l'éditeur s'affichera dans une fenêtre graphique (comme dans l'image ci-dessous) et vous aurez accès à un système de menus à l'aide de votre souris.



Dans le cas contraire, celui-ci s'exécutera dans le terminal ce qui peut être pratique lorsque vous désirez vous connecter sur une machine distante. Si vous avez un système de fenêtrage, vous pouvez également forcer l'exécution dans votre terminal en tapant la commande suivante : `$> emacs -nw`

L'affichage est décomposé en trois parties :

- en haut : vous avez accès un menu standard,
- au centre : le contenu du fichier est affiché,
- en bas : un champ texte permet à l'éditeur d'afficher des informations importantes à l'utilisateur.

Lorsque vous éditez un fichier l'éditeur va utiliser l'extension de celui-ci afin d'adapter l'affichage (appelé *mode*) au contenu (système de coloration, d'indentation automatique …). Dans le cas où le contenu du fichier ne peut être déterminé, le mode standard (ou *fondamental*) est utilisé.

Emacs utilise la notion de « *buffer* » pour éditer un fichier, les modifications réalisées sur un fichier ne sont pas directement effectuées sur celui-ci. En effet, tant que vous n'aurez pas sauvegardé les modifications elles seront mises en mémoire dans le « *buffer* » et transférées dans le fichier ouvert lors de la sauvegarde.

Vous pouvez modifier un « *buffer* » en vous déplaçant dans celui-ci à l'aide des flèches et en tapant du texte directement au clavier. Vous pouvez sauvegarder les modifications apportées au « *buffer* » avec la combinaison de touches `C-x` puis `C-s`. Si vous voulez enregistrer les modifications sans avoir ouvert un fichier, *Emacs* vous demande d'entrée le nom du fichier à créer ainsi que sont chemin d'accès (à partir de la racine de votre compte utilisateur).

Pour quittez l'éditeur, vous devez utiliser la combinaison de touches `C-x` puis `C-c`. Si un ou plusieurs « *buffer* » ont été modifiés sans avoir été sauvegardés, *Emacs* vous demandera s'il doit les sauvegarder.

Voici ci-dessous les principales combinaisons de touches permettant d'accéder aux diverses fonctionnalités de cet éditeur :

- Quitter *Emacs*

Commande	Description
`C-z`	Suspendre emacs
`C-x C-c`	Quitter emacs

- Aide

Commande	Action
`C-h`	Aide d'emacs
`C-h t`	Lance le tutorial d'emacs

- Manipuler les fichiers et « *buffers* »

Commande	Description
`C-x C-f`	Ouvrir un (nouveau) fichier
`C-x C-s`	Sauvegarder le buffer courant
`C-x s`	Sauvegarder tous les buffers en cours d'édition
`C-x C-b`	Avoir la liste de tous les buffers.
`C-x b`	Changer de buffer
`C-x o`	Passer à une autre fenêtre
`C-x 1`	Faire disparaître toutes les fenêtres sauf la fenêtre courante
`C-x 2`	Partage la fenêtre courante en 2, horizontalement
`C-x 3`	Partage la fenêtre courante en 2, verticalement

- Déplacer du texte

Bouger d'un(e)...	Vers l'avant	Vers l'arrière
caractère	`C-b`	`C-f`
mot	`M-b`	`M-f`
ligne	`C-p`	`C-n`
début/fin de ligne	`C-a`	`C-e`
phrase	`M-a`	`M-e`
paragraphe	`M-{`	`M-}`
buffer	`M-<`	`M->`

- Effacer du texte

Commande	Action
`C-d`	Efface le caractère sur lequel est le curseur.
`M-d`	Efface le mot à partir du curseur.

M-backspace	Efface le mot précédent.
C-k	Efface la ligne à partir du curseur
_	Efface le paragraphe à partir du curseur.

- Sélectionner du texte

Commande	Action
M-@ <n>	Sélectionne <n> mots à partir de la position du curseur
M-h	Sélectionner tout le paragraphe
C-x h	Sélectionner le buffer entier

- Couper, copier et coller

Commande	Action
C-w	Couper la sélection
M-w	Copier la sélection
C-y	Coller

- Chercher et remplacer

Commande	Action
C-s	Recherche simple vers la fin du fichier
C-r	Recherche simple vers le début du fichier
M-%	Remplacer

Plus de raccourci et de détails sur cet éditeur sont disponibles dans le lien suivant :

http://www.tuteurs.ens.fr/unix/editeurs/emacs.html

Question 1 – Lancez l'éditeur *Emacs*, puis entrez du texte dans l'éditeur et sauvegardez le fichier.

L'éditeur *Vi* ou *Vim*

L'éditeur *Vi* est l'un des éditeurs les plus utilisés et issu des premiers éditeurs sous les systèmes de type UNIX.
Cet éditeur dispose de deux principaux modes, le mode *normal* pour entrer des commandes et le mode *insertion* pour modifier le contenu d'un fichier ouvert.
Le mode normal est celui utilisé par défaut lorsque l'éditeur est lancé, il est ainsi possible d'entrer des commandes indiquées par une lettre préfixée par le symbole « : ». Pour entrer en mode insertion, il faut taper la touche `i` au clavier il sera possible ensuite de naviguer avec les flèches du clavier et modifier le fichier édité. La combinaison de touche <CTRL>+C permet de revenir au mode *normal*.

Voici ci-dessous les principales combinaisons de touches permettant d'accéder aux diverses fonctionnalités de cet éditeur :

- Sauver, charger et quitter

Commande	Effet
`:w`	sauve le fichier en cours d'édition
`:w fichier`	écrit le texte dans le fichier indiqué
`:sav fichier`	sauve le fichier sous un nouveau nom
`:ed fichier`	édite un nouveau fichier
`:q`	quitte
`:x`	sauve si nécessaire et quitte
`:wq`	sauve et quitte
`:q!`	quitte en mode forcé (sans sauvegarder)

- Autres principales commandes

Commande	Sens	Type	Effet
`i`	insert		passe en mode insertion
`a`	append		passe en mode insertion en faisant avancer le curseur
`o`	open		passe en mode insertion en créant une nouvelle ligne sous le curseur
`O`	open		passe en mode insertion en créant une nouvelle ligne au dessus du curseur
`R`	replace		passe en mode remplacement
`A`			passe en mode insertion en allant à la fin de la ligne
`I`			passe en mode insertion en allant au début de la ligne
`fx`	forward	déplacement	avance jusqu'au prochain *x*
`Fx`		déplacement	recule jusqu'au *x* précédent
`gg`		déplacement	va au début du texte
`G`		déplacement	va à la fin du texte
`$`		déplacement	va à la fin de la ligne
`0`		déplacement	va au début de la ligne
`{`		déplacement	va au début du paragraphe
`}`		déplacement	va à la fin du paragraphe
`n`	next	déplacement	continue une recherche
`N`		déplacement	continue une recherche, en sens inverse
`u`	undo		annule la dernière action
`.`	redo		répète la dernière opération
`rx`	replace		remplace un caractère par *x*
`p`	paste		colle un registre après le curseur
`P`			colle un registre avant le curseur
`/`			fais une recherche
`?`			fais une recherche vers l'arrière
`d`	delete	action	supprime
`y`	yank	action	copie dans un registre
`=`		action	met en forme un programme

Plus de raccourci et de détails sur cet éditeur sont disponibles dans le lien suivant :
http://www.tuteurs.ens.fr/unix/editeurs/vim.html

Question 2 – Lancez l'éditeur *Vi*, puis entrez du texte dans l'éditeur et sauvegardez le fichier.

EXERCICE 2

Dans cet exercice, nous allons étudier l'outil Make permettant d'automatiser la chaine de compilation de programme en cours et lors du précédent TP.

L'outil Make

L'outil `make` exploite les dépendances existantes entre les modules entrant en jeu dans la construction d'un programme exécutable pour ne lancer que les opérations de compilations et éditions de liens nécessaires, lorsque ce programme exécutable doit être reconstruit suite à une modification intervenue dans les modules sources.
Pour cela, `make` utilise deux sources d'informations : un fichier de description appelé le *Makefile* qui contient la description des dépendances entre les modules, et les noms et les dates de dernières modifications des modules.

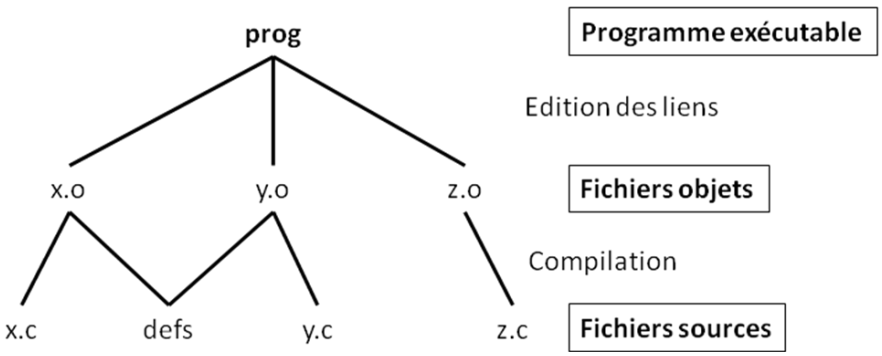
Format du fichier Makefile

Le fichier *Makefile* décrit les dépendances existantes entre les modules intervenant dans la construction d'un exécutable. Il traduit sous forme de règles le graphe de dépendance du programme exécutable à construire et indique pour chacune de ces dépendances, l'action qui lui est associée.
Une règle dans le fichier *Makefile* est de la forme :

```
module cible :  dépendances
<tab> commande pour construire le module cible
```

Remarque : il y a une tabulation avant la ou les commandes associées à une règle.

Prenons comme exemple le cas suivant, le programme exécutable `prog` est construit à partir d'une étape d'édition des liens prenant en compte les trois modules objets `x.o`, `y.o` et `z.o`. Le module objet `z.o` est issu de la compilation d'un programme source `z.c`. Les modules `x.o` et `y.o` sont à leur tour issus de la compilation respective des modules source `x.c` et `y.c`. Ces deux derniers modules utilisent un module `defs` par le biais d'un ordre d'inclusion `#include`.
Le graphe de dépendance du programme `prog` est donné sur la figure ci-dessous.



Le fichier *Makefile* résultant est :

```
prog : x.o y.o z.o
     gcc -o prog x.o y.o z.o
x.o : defs x.c
     gcc -c x.c
y.o : defs y.c
     gcc -c y.c
z.o : z.c
     gcc -c z.c
```

La première règle stipule la dépendance associée au programme exécutable `prog` qui est donc construit à partir des modules `x.o`, `y.o` et `z.o`. La commande permettant la construction du programme exécutable `prog` à partir de ces trois modules objets est la commande d'édition des liens `gcc -o prog x.o y.o z.o`.
La seconde et la troisième règle stipulent que le fichier objet `x.o` (respectivement `y.o`) dépend à la fois du fichier `x.c` (respectivement `y.c`) et du fichier `defs`. Les fichiers `y.o` ou `x.o` sont construits par compilation des fichiers sources `.c` correspondants.
Enfin, la dernière règle spécifie que le fichier `z.o` dépend uniquement de la compilation du fichier `z.c`.

Fonctionnement de l'utilitaire Make

L'outil `make` utilise le fichier *Makefile* et les dates de dernières modifications des modules pour déterminer si un module est à jour.
Un module est à jour si le module existe et si sa date de dernière modification est plus récente ou égale aux dates de dernière modification de tous les modules dont il dépend. Si un module n'est pas à jour, la commande associée à ses dépendances est exécutée pour reconstruire le module.
Considérons par exemple que le module `z.c` soit modifié. L'utilitaire `make` va détecter que ce module est devenu plus récent que le module objet `z.o`. Il va donc lancer la commande associée à la règle de dépendance du module `z.o`, soit la commande de compilation `gcc -c z.c`. L'exécution de cette commande va à son tour générer un module `z.o` plus récent que le programme exécutable `prog`. En conséquence, l'utilitaire `make` va reconstruire le programme exécutable `prog` en lançant la commande d'édition des liens `gcc -o prog x.o y.o z.o`. D'une façon similaire, toute modification au sein du module `defs` entraînera la reconstruction des modules `y.o` et `x.o`, par le biais de deux opérations de compilation, puis la reconstruction du programme exécutable `prog`. Dans ces deux cas, seules les opérations de compilation ou d'édition des liens nécessaires sont exécutées.
L'utilitaire `make` est appelé au moyen de la commande `$> make prog` qui suppose qu'un fichier *Makefile* est présent dans le répertoire où la commande est lancée.

Question 1 – En suivant les indications données ci-dessus et en reprenant les sources du TP précédent, réalisez un *Makefile* contenant les règles de compilation et d'édition des liens afin d'obtenir le programme exécutable `prog_exo1`. Le programme `prog_exo1` est généré à partir des modules objets `prog_exo1.o` et `lib_func.o`, obtenus respectivement à partir de la compilation des fichiers sources `prog_exo1.c` et `lib_func.c`.

Correction : Voici ci-dessous un exemple de *Makefile* répondant à la question

```
prog_exo1 : prog_exo1.o lib_func.o
     gcc -o prog_exo1 prog_exo1.o lib_func.o

prog_exo1.o : prog_exo1.c
```

```
	gcc -c prog_exo1.c

lib_func.o : lib_func.c
	gcc -c lib_func.c
```

Question 2 – Utilisez un éditeur de texte afin de créer le *Makefile* donné à la question précédente. Vous sauvegarderez ce fichier sous le nom `Makefile` afin que l'outil *Make* puisse l'utiliser.
Exécutez votre fichier *Makefile* en exécutant la commande suivante dans le répertoire contenant les fichiers sources .c : `$> make prog_exo1`
Listez le contenu du répertoire dans lequel vous avez exécuté votre Makefile et dites si vous obtenez le résultat attendu.

Question 3 – On désire rajouter une règle nommée `clean` qui a pour objectif de supprimer le fichier exécutable `prog_exo1` ainsi que tous les fichiers objets (fichiers `prog_exo1.o` et `lib_func.o`).
Indiquez ce qu'il faut rajouter dans le fichier Makefile pour réaliser cette action.

Correction : Voici ci-dessous un exemple de *Makefile* répondant à la question

```
prog_exo1 : prog_exo1.o lib_func.o
	gcc -o prog_exo1 prog_exo1.o lib_func.o

prog_exo1.o : prog_exo1.c
	gcc -c prog_exo1.c

lib_func.o : lib_func.c
	gcc -c lib_func.c

clean : prog_exo1 prog_exo1.o lib_func.o
	rm prog_exo1
	rm prog_exo1.o
	rm lib_func.o
```

Question 4 – Exécutez cette règle en tapant la commande suivante : `$> make clean`
Listez le contenu du répertoire contenant les fichiers `prog_exo1.c` et `lib_func.c`, et dites si vous obtenez bien le résultat attendu.

Question 5 – Même question mais en utilisant obligatoirement une expression régulière pour la commande associée à la règle `clean`.

Correction : Voici ci-dessous un exemple de *Makefile* répondant à la question

```
prog_exo1 : prog_exo1.o lib_func.o
	gcc -o prog_exo1 prog_exo1.o lib_func.o

prog_exo1.o : prog_exo1.c
	gcc -c prog_exo1.c

lib_func.o : lib_func.c
	gcc -c lib_func.c

clean : prog_exo1 prog_exo1.o lib_func.o
```

```
rm prog_exo1
rm *.o
```

EXERCICE 3

Dans cet exercice, nous allons aller beaucoup plus loin dans l'automatisation de traitements en réalisant de petites scripts assez simples, dits « *scripts shell* ».
Un script shell est un fichier texte qui est interprété par un terminal (ou « *shell* »). Cela signifie que le terminal lit et exécute en séquence les lignes contenues dans le script.
Il existe divers types de terminaux : *Bourne Shell* (`sh`), *Bourne Again Shell* (`bash`), *Korn Shell* (`ksh`) ou encore *C Shell* (`csh`). Les plus répandus sont les deux premiers types cités, dans la suite nous utiliserons le premier cas.

Pour réaliser un script, vous devez utiliser un éditeur de texte afin d'enregistrer son contenu. Vous êtes libre d'utiliser l'éditeur de texte qui vous est le plus simple à utiliser, sachez que les éditeurs *Vim* et *Emacs* proposent des fonctionnalités (coloration syntaxique …) assez utiles pour développer des scripts shell.

Un script est un fichier texte contenant une succession de commandes exécutées par le shell. Tout script shell contient la ligne ci-dessous indiquant le type de shell et son chemin d'accès pour exécuter le script :

`#!/bin/sh` ou `#!/bin/bash`

Il est important de rendre lisible votre code en l'aérant avec des sauts de ligne, en l'indentant ou en mettant des commentaires. Toute ligne commençant par le symbole # sera interprété comme une ligne de commentaire. N'hésitez pas à mettre des commentaires dans vos programmes cela permettra de les rendre plus simple à comprendre pour vous d'autres personnes afin qu'ils puissent être repris plus tard.
D'autre part afin de pouvoir exécuter tout script il faut que le droit en exécution soit activé, pour cela vous pouvez utiliser la commande suivante vu dans les précédent TP :

```
$> chmod u+x nom_script
```

Par défaut, votre shell ne connaît pas le chemin d'accès à votre script pour l'exécuter (excepté si celui-ci se trouve dans le répertoire courant dans lequel vous lancer le script). Vous devez donc indiquer le chemin d'accès à votre script ou ajouter son chemin d'accès dans la variable d'environnement PATH.

Question 1 – Créez un premier script shell de nom `hello.sh` permettant d'afficher le message « Hello world ! » et exécutez le dans votre terminal pour voir le résultat.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "Hello world!"
```

Remarque : par défaut la commande `echo` affiche un message avec un retour chariot en fin de celui-ci, pour ne pas ajouter de retour chariot vous pouvez utiliser l'option `-n`.

Question 2 – Comment modifier ce premier script afin d'afficher votre login d'utilisateur en fin de phrase. Modifiez ce script en conséquence et exécutez le pour voir le résultat.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "Hello world $USER!"
```

La commande `read` permet de récupérer une chaine de caractères ou un nombre entré suivit par la touche `<entrée>` au clavier par l'utilisateur. Cette commande s'utiliser comme suit :

```
read var
```

où `var` est le nom d'une variable dans laquelle sera stocké le contenu lu au clavier.

Question 3 – Modifiez le script précédent afin de demander à l'utilisateur d'entrer son nom puis d'afficher le message « Hello world John! » si l'utilisateur a entré *John* au clavier.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "Entrez votre nom :"
read user
echo "Hello world $user!"
```

Remarque : la commande read peut prendre plusieurs variables en paramètre (le symbole d'espacement permet de séparer le contenu affecté à chaque variable) ou aucune (dans ce dernier cas cela permet d'attendre que l'utilisateur appuie sur une touche).

La commande `grep` permet d'afficher en sortie les lignes d'un flux de données (à partir d'un fichier ou résultat d'une autre commande) contenant une chaine de caractères indiquées en paramètre. Par exemple, la commande suivante recherche et affiche toutes les lignes du livre.txt contenant le mot « Lola » : `$> grep Lola livre.txt`

Question 4 – Créez un script listant le contenu du répertoire courant et affichant le nombre de fichiers d'extension .sh, vous combinerez plusieurs commandes et utiliserez les redirections de flux.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "Le nombre de scripts contenus dans le repertoire courant est :"
ls | grep .sh | wc -l
```

Question 5 – Même question mais comptant le nombre de fichier qui ne sont pas des scripts.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "Le nombre de fichiers autres que des scripts contenus dans le repertoire courant est :"
ls | grep -v .sh | wc -l
```

Question 6 – Modifiez le script de la question précédente de sorte à compter le nombre de sous-répertoires contenus dans le répertoire courant. Pour vous aider, vous utiliserez les expressions régulières et la commande avec les options comme suit : `ls -al`

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "Le nombre de sous-repertoires contenus dans le repertoire courant est :"
ls -al | grep ^d | wc -l
```

Il est possible de passer des arguments lors du lancement d'un script, cela permet de fournir des informations au script dès son lancement.
Plusieurs variables réservées sont automatiquement affectées comme indiquées ci-dessous :

Nom variable	Descriptif
`$#`	Indique le nombre d'arguments passés en paramètre
`$0`	Correspond au nom du script
`$1, $2, … ${10}`	Correspond à la valeur du premier, deuxième, …, dixième argument
`$*`	Liste tous les arguments

Question 7 – Modifiez le script précédent pour afficher tous fichiers du répertoire courant dont l'extension correspond à l'extension passé en paramètre par l'utilisateur.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "Les fichiers avec l'extension $1 dans le repertoire courant sont :"
ls -al | grep $1$
```

Question 8 – Modifiez le script précédent pour afficher également la liste des arguments passés en paramètres au script.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
#!/bin/sh

echo "La liste des parametres sont :"
echo "$*"
```

```
        echo "Les fichiers avec l'extension $1 dans le repertoire
courant sont :"
        ls -al | grep $1$
```

Question 9 – Réalisez un script permettant de créer une archive avec le nom indiqué par l'utilisateur avec le contenu du répertoire courant. Attention, n'oubliez pas de changer de répertoire courant avant de l'archiver.

Correction : Voici ci-dessous un exemple de script répondant à la question

```
        #!/bin/sh

        echo "Demarrage de l'archivage du répertoire : $PWD"
        rep=$PWD
        cd ..
        tar -cvzf $1 $rep
        echo "Archivage terminé, fichier $1 créé dans le
répertoire $PWD"
```