Introduction au scripting Shell

Hugo Blanc - Université Lyon 1
LP ESSIR 2024-2025

October 03, 2024

« Two of the most famous products of Berkeley are LSD and Unix.
I don’t think that is a coincidence. »

– The UNIX HATERS Handbook

Table des matières

0.1. Pré-requis 5
1. Introduction 6
1.1. Qu'attendre de ce cours ? 6
2. Rappels sur le fonctionnement de Linux 8
2.1. *Linux is not UNIX* 8
2.2. Philosophie des systèmes UNIX 8
2.3. Hiérarchie du système de fichiers 9
2.4. `/dev` et `/proc` 10
3. Fonctionnalités de base du shell 12
3.1. Redirections d'E/S 12
3.2. Pipes 13
3.3. Pipes (*internals*) 13
3.3.1. Exemple 14
3.4. Exercices 14
4. Bases du scripting shell 15
4.1. Variables 15
4.2. Conditions 17
4.2.1. If/elif/else 17
4.2.2. Switch case 18
4.3. Comparaisons 20
4.4. Arithmétique 21
4.5. Boucles 21
4.5.1. While 21
4.5.2. For 22
4.6. Fonctions 23
4.7. Substitution de commande 24
4.8. Exercices 25
5. Concepts avancés 26
5.1. Manipulation de strings 26
5.1.1. Connaître la longueur d'une string 26
5.1.2. Extraction d'une *substring* 26
5.1.3. Suppression de *substring* - début 26
5.1.4. Suppression de *substring* - fin 27
5.1.5. Remplacement de *substring* 27
5.1.6. Note sur la performance 27
5.2. Substitution de processus 29
5.3. Tableaux 29
5.3.1. Récupérer la taille d'un tableau 31
5.3.2. Ajout d'un élément 31
5.3.3. Suppression d'un élément 31
5.4. Parsing d'options 31
5.4.1. Parsing d'options natif 32
5.4.2. `getopts` 33

6. Astuces et bonnes pratiques 35
6.1. Améliorer la fiabilité des scripts 35
6.2. Idempotence 35
6.3. Heredocs 36
6.4. Raccourcis clavier 37
7. Licence 38
Index des figures 39

0.1. Pré-requis

- Avoir une machine GNU/Linux en état de fonctionnement.
- Avoir un utilisateur différent de `root` appartenant au groupe `sudo`.
- Avoir une connexion à Internet.
- Avoir des connaissances de base sur le fonctionnement de Linux et du terminal.

1. Introduction

Les ordinateurs modernes ont de nombreuses interfaces qui permettent aux humains de leur envoyer des commandes: des interfaces graphiques, vocales, et bientôt des interfaces en VR/AR. Toutes ces interfaces sont très utiles pour un usage basique et simple de ces machines, mais elles atteignent leurs limites dès que les utilisateur.rices deviennent des *power users*. Une interface cependant permet d'aller plus loin que les autres: l'**interface textuelle**, le **shell**.

Le shell est à la fois:

- un **interpréteur de commandes**, qui permet d'envoyer des instructions au système hôte;
- un **langage de « programmation »**.

Un programme shell, appelé **script**, permet d'assembler des commandes/outils/programmes/appels systèmes… ensemble. Le shell met à disposition des commandes internes (appelées *builtins*) permettant de faire des structures complexes: des tests, des boucles, des branches, etc. Les *builtins* permettent également d'implémenter des fonctionnalités qui sont impossibles à mettre en œuvres avec d'autres programmes car ils manipulent le shell lui-même.

De part leur intégration avec l'écosystème Linux, les scripts shells sont particulièrement utiles pour l'administration système, les tâches répétitives de routine… C'est donc pour cela qu'une connaissance pratique du scripting shell est **essentielle** pour quiconque souhaite devenir raisonnablement compétent.e en administration système.

Au delà de l'administration système, réaliser des scripts shell permet également le prototypage rapide d'applications plus complexes (appels d'API, fonctionnalité de base…) avant de passer à un langage de programmation plus complexe.

Le tout premier shell UNIX, `sh`, créé par Ken Thompson (qui est notamment le créateur d'UNIX) aux Bell Labs. `sh` est déclaré en tant que « shell par défaut » dès la première version d'UNIX en 1971, et introduit de nombreux concepts et standards toujours utilisés aujourd'hui (pipes, etc.).

Cependant, ce shell a été remplacé en 1979 par le Bourne Shell, créé par Stephen Bourne aux Bell Labs. Il est « shell par défaut » dès UNIX 7.

> **Note**
>
> Dans la cadre de ce cours, nous utiliserons `bash` (Bourne-again Shell), standard de-facto sur la grande majorité des systèmes UNIX.

1.1. Qu'attendre de ce cours ?

Ce cours n'est pas:

- un cours d'introduction sur le fonctionnement de Linux. Nous ne reviendrons pas en détails sur l'architecture de l'OS, ainsi que ses grands principes de fonctionnement (kernel/userland, appels systèmes, stack réseau, etc.);
- un cours sur les commandes les plus utiles/populaires/importantes/etc.

En revanche, vous apprendrez comment:

- créer et exécuter un script Shell;
- utiliser les primitives Bash pour structurer et ajouter de l'intelligence à vos scripts;
- optimiser vos scripts et mettre en place des bonnes pratiques de programmation.

2. Rappels sur le fonctionnement de Linux

2.1. *Linux is not UNIX*

Même si l'introduction de ce document relate l'histoire de l'apparition du shell dans les premières versions d'UNIX, il est important de rappeler que ***Linux is not UNIX!***. En effet, un système utilisant MacOS (un dérivé de BSD) est, d'un point de vue généalogique, plus proche d'UNIX que ne l'est Linux.

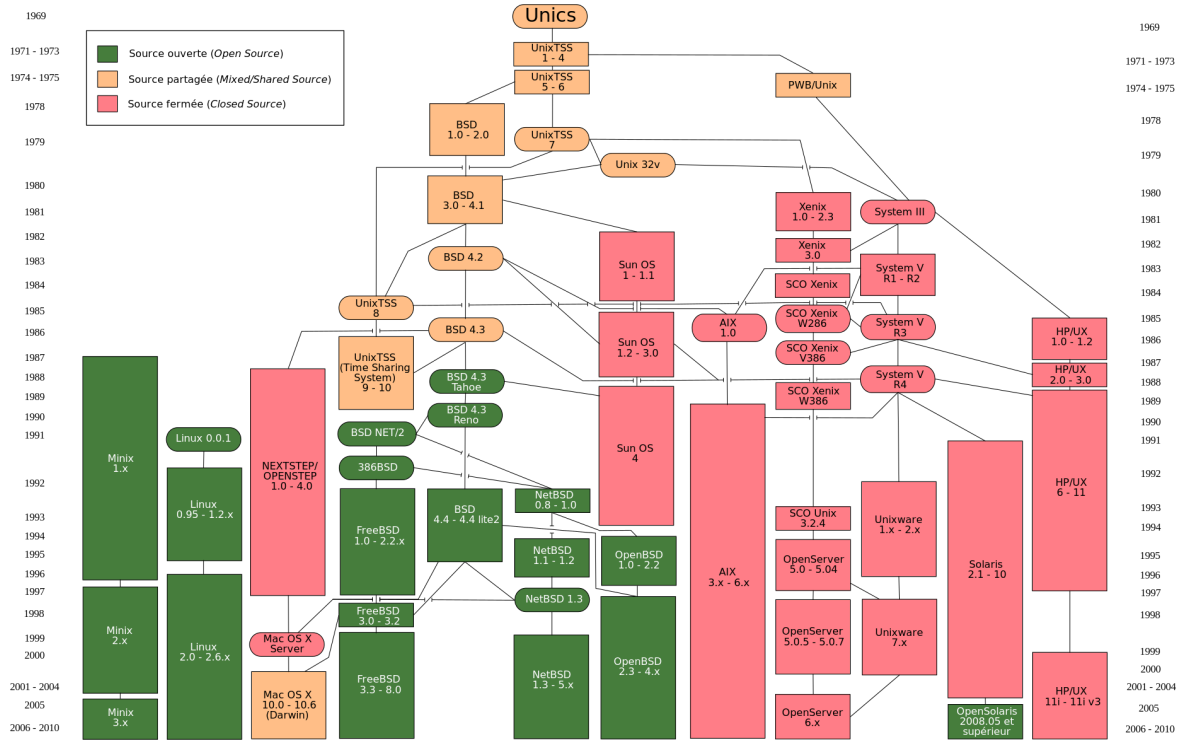

Fig. 1. – Principales familles de systèmes UNIX[1]

Ce que l'on appelle couramment « Linux » (ou GNU/Linux) est une famille d'OS **de type UNIX** (*UNIX-like*) reposant principalement sur le **kernel Linux** créé par Linus Torvalds en 1991.

> I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready.
> – Linux Torvalds, 25 Août 1991

2.2. Philosophie des systèmes UNIX

Les systèmes UNIX(-like) ont une philosophie qui leur est commune:

- ce qui est petit est beau;
- **chaque programme fait une chose et la fait bien**;

[1] https://fr.wikipedia.org/wiki/Unix°

- construire un prototype dès que possible;
- choisir la portabilité plutôt que l'efficacité;
- **enregistrer les données dans des fichiers plats**;
- utiliser le logiciel comme une force;
- utiliser les scripts shell pour accroître cette force;
- éviter les interfaces utilisateur captives;
- **faire de chaque programme un filtre**.

Un choix majeur fait par les systèmes UNIX-like est que **tout est fichier**. On peut donc traiter n'importe quelle entité du système comme un fichier plat, et manipuler son contenu uniquement via des flux de caractères.

Un fichier est une chaîne de caractères **non structurée** (du point de vue du système). À tout fichier est associé un **inode** qui contient:

- sa taille;
- l'adresse des blocs utilisés sur le disque pour ce fichier;
- l'identification de son propriétaire et ses droits d'accès;
- son type (fichier ordinaire ou spécial…);
- un compteur de ses références dans le système;
- un certain nombre de dates relatives aux principales opérations réalisables.

Il existe cependant différents types de fichiers:

- fichiers **ordinaires**: ils contiennent des programmes, des données (ASCII…);
- fichiers **spéciaux**: par exemple les entrées/sorties;
- **répertoires**.

2.3. Hiérarchie du système de fichiers

Le système de fichiers est construit en arborescence:

- `/` : C'est le répertoire racine. C'est là que l'arbre entier commence.
- `/bin` : Ce répertoire contient des programmes exécutables qui sont nécessaires en mode mono-utilisateur et pour mettre le système en marche ou le réparer.
- `/boot` : Contient les fichiers statiques pour le chargeur de démarrage. Ce répertoire contient uniquement les fichiers nécessaires au processus de démarrage.
- `/dev` : Les fichiers spéciaux ou fichiers de périphériques, qui font référence à des périphériques physiques.
- `/etc` : Contient les fichiers de configuration qui sont locaux à la machine.
- `/home` : Sur les machines avec des répertoires personnels pour les utilisateurs, ceux-ci se trouvent généralement sous ce répertoire, directement ou non.
- `/lib` : Ce répertoire doit contenir les bibliothèques partagées qui sont nécessaires pour démarrer le système et exécuter les commandes dans le système de fichiers racine.
- `/lost+found` : Ce répertoire contient les éléments perdus dans le système de fichiers.
- `/media` : Ce répertoire contient les points de montage pour les supports amovibles tels que les disques CD et DVD ou les clés USB.
- `/mnt` : Ce répertoire est un point de montage pour un système de fichiers monté temporairement.

- `/opt` : Ce répertoire doit contenir les paquets complémentaires qui contiennent des fichiers statiques.
- `/proc` : Il s'agit d'un point de montage pour le système de fichiers proc, qui fournit des informations sur les processus en cours et le kernel.
- `/root` : Ce répertoire est généralement le répertoire personnel de l'utilisateur `root`.
- `/run` : Ce répertoire contient des informations qui décrivent le système depuis son démarrage.
- `/sbin` : Comme `/bin`, ce répertoire contient les commandes nécessaires au démarrage du système, mais qui ne sont généralement pas exécutées par les utilisateurs normaux.
- `/tmp` : Ce répertoire contient des fichiers temporaires qui peuvent être supprimés sans préavis, par exemple par un travail régulier ou au démarrage du système. Peut être un FS dédié en RAM (`tmpfs`).
- `/var` : Ce répertoire contient les fichiers qui peuvent changer en taille, comme les logs.

Toutes ces informations sont retrouvable directement dans un terminal, avec la commande:

```
$ man hier
```

2.4. `/dev` ET `/proc`

Le dossier `/dev` contient les périphériques physiques qui sont (ou ne sont pas) présents du point de vue *hardware*. Ils sont appelés **fichiers périphériques** (*devices* en anglais). Par exemple, nous pouvons y trouver les partitions disque:

```
$ df
Filesystem     1K-blocks     Used Available Use% Mounted on
devtmpfs            4096        0      4096   0% /dev
tmpfs            6069740    10408   6059332   1% /dev/shm
tmpfs            2427896     4344   2423552   1% /run
/dev/sda3      498533376 14561592 482917720   3% /
tmpfs            6069740     3148   6066592   1% /tmp
/dev/sda3      498533376 14561592 482917720   3% /home
/dev/sda2         996780   245220    682748  27% /boot
/dev/sda1         523244    17780    505464   4% /boot/efi
tmpfs            1213948       92   1213856   1% /run/user/1000
```

`/dev` contient entre autres des périphériques de loopback, comme `/dev/loop0`. Un périphérique de *loopback* est un artifice qui permet d'accéder à un fichier ordinaire comme s'il s'agissait d'un périphérique de bloc. Un tel fichier permet par exemple de monter un système de fichier entier dans un seul gros fichier:

```
SIZE=1000000 #1 meg

head -c "$SIZE" < /dev/zero > file
losetup /dev/loop0 file
mke2fs /dev/loop0
mount -o loop /dev/loop0 /mnt
```

> **Note**
>
> Un périphérique de bloc lit et/ou écrit des données par morceaux, ou blocs, contrairement à un périphérique de caractère, qui accède aux données par unités de caractère. Les disques durs, les lecteurs de CD-ROM et les lecteurs flash sont des exemples de périphériques de bloc. Les claviers, les modems et les cartes son sont des exemples de périphériques de caractères.

D'autres fichiers de `/dev` ont un rôle important, et seront détaillés plus loin dans ce cours.

Le dossier `/proc` est en réalité un pseudo système de fichiers: les fichiers de `/proc` reproduisent les processus système et kernel en cours d'exécution et contiennent des informations et des statistiques à leur sujet (`/proc/devices` , `/proc/interrupts` , `/proc/partitions` ...). Il est fréquent que des scripts shells aient à y extraire des données (niveau de batterie de la machine, informations sur le kernel...).

Le répertoire `/proc` contient des sous-répertoires portant des noms inhabituels (nombres). Chacun de ces noms correspond à l'ID d'un processus en cours d'exécution. Dans chacun de ces sous-répertoires, il y a un certain nombre de fichiers qui contiennent des informations utiles sur le processus correspondant (environnement chargé, avec le fichier `/proc/pid/environ` ...).

> **Avertissement**
>
> Il est fortement **déconseillé d'écrire** dans les fichiers présents dans `/proc` car cela pourrait corrompre le système de fichier et/ou faire crasher la machine.

3. Fonctionnalités de base du shell

3.1. Redirections d'E/S

Par défaut, tout processus possède:

- une entrée;
- deux sorties;

Ces E/S sont **distinctes** pour chaque processus. Pour les programmes interactifs, les entrées proviennent du clavier et les sorties s'affichent sur l'écran.

Ces E/S sont nommées:

- **STDIN** (*standard input*): ce qui est envoyé vers le processus.
- **STDOUT** (*standard output*): ce qui est envoyé par le processus.
- **STDERR** (*standard error*): les erreurs envoyées par le processus.

Fig. 2. – Schématisation des entrée/sorties d'un processus

Ces E/S sont associées à des **descripteurs de fichier** (*file descriptors*, *fd*):

- STDIN: fd 0;
- STDOUT: fd 1;
- STDERR: fd 2.

Les E/S peuvent être redirigées *de* et *vers* un fichier:

- `processus < fichier` : STDIN provient du fichier (et non plus du clavier).
- `processus > fichier` : STDOUT est écrit dans le fichier (et non plus sur le terminal).
- `processus >> fichier` : STDOUT est ajoutée **à la fin** du fichier.
- `processus 2> fichier` : STDERR est écrit dans le fichier (et non plus sur le terminal).

Plusieurs *devices* (fichiers dans `/dev`) peuvent être utilisés pour de la redirection d'E/S:

- `/dev/null` : trou noir annihiliant tout ce qui lui est envoyé.
- `/dev/zero` : envoie des zéros ad-vitam.
- `/dev/random` et `/dev/urandom` : fournisseurs officiels de « hasard ».
- `/dev/full` : se plaint toujours d'être plein.

Il est possible de coupler plusieurs redirections. Par exemple:

```
processus > fichier 2> /dev/null
```

redirige STDOUT dans le fichier et STDERR dans `/dev/null`.

On peut référencer un fd en utilisant le caractère `&`:

```
processus > /dev/null 2>&1
```

Exercice

Dans l'exemple ci-dessus, où va STDOUT ? STDERR ?

3.2. Pipes

Les pipes, représentés par le caractère `|` permettent d'envoyer la sortie d'un processus (STDOUT) à l'entrée d'un autre (STDIN):

```
processus1 | processus2 | processus3
```

Par exemple:

- `a | b` redirige STDOUT du programme `a` vers STDIN du programme `b`.
- `a 2>&1 | b` redirige STDERR de `a` vers STDOUT de `a` et redirige STDOUT de `a` vers STDIN de `b`.

Exercice

Faire réaliser le calcul $3 + \frac{2}{10}$ à l'utilitaire `bc` de deux manières différentes, en utilisant:

- une redirection avec des chevrons;
- un pipe.

3.3. Pipes (*internals*)

Note

Cette section va explorer plus en détail le fonctionnement des pipes. Elle n'est pas nécessaire pour la bonne compréhension de la suite du cours, mais est tout de même conseillée si vous souhaitez connaître plus en détails le fonctionnement du noyau Linux.

Les pipes sont implémentés dans le kernel Linux. Dans les grandes lignes:

- Linux a un module VFS (*virtual file system*) appelé `pipefs` qui est monté dans le kernel lors du boot.
- Ce FS est monté en parallèle de `/`, pas dedans. La racine du FS `pipefs` est `pipe:`.
- Contrairement aux autres FS, `pipefs` ne peut pas être examiné pas l'utilisateur.rice.
- Le point d'entrée de `pipefs` est le syscall `pipe(2)`.
- Le syscall `pipe(2)` utilisé par les programmes, les shells etc. créé un fichier dans ce FS et retourne deux fd: un pour la lecture et un pour l'écriture.
- `pipefs` est stocké en RAM.

Un pipe a une capacité limitée: depuis Linux 2.6.35, elle est de 65,536 octets. Quand un pipe est plein, un appel à `write(2)` sera bloquant (ou échouera si le flag `O_NONBLOCK` est précisé) jusqu'à ce qu'assez de données aient été lues pour libérer de l'espace.

De même, la tentative de lecture d'un pipe vide avec `read(2)` sera bloquant jusqu'à ce qu'il y ait des données de disponible. En revanche, si tous les fd pointant vers le côté écriture d'un pipe sont fermés, alors une lecture dans le pipe retournera EOF, et `write(2)` retournera 0.

3.3.1. Exemple

Prenons une suite de commandes liées entres-elles par des pipes:

```
ls -l | sort | less
```

- Bash va créer deux pipes, un pour piper `ls` vers `sort` et un pour piper `sort` vers `less`.
- Ensuite, Bash va se forker (`fork(2)`) lui-même 3 fois: 1 processus parent et 3 processus enfants pour chaque commande.
- Le processus enfant 1 (`ls`) va configurer son fd de STDOUT pour qu'il écrive dans le pipe A.
- Le processus enfant 2 (`sort`) va configurer son fd de STDIN pour qu'il lise dans le pipe A (pour lire la sortie de `ls`).
- Le processus enfant 2 (`sort`) va configurer son fd de STDOUT pour qu'il écrive dans le pipe B.
- Le processus enfant 3 (`less`) va configurer son fd de STDIN pour qu'il lise dans le pipe B (la sortie de `sort`).
- Chaque processus enfant va lancer sa commande.

Ainsi, toutes les commandes qui sont pipées entres-elles s'exécutent en parallèle.

3.4. Exercices

1. Utiliser la commande `cat` pour écrire quelques caractères dans le fichier `/tmp/foobar`.
2. Utiliser la commande `cat` pour afficher le contenu du fichier `/etc/hosts` de deux manières : en passant le nom du fichier en argument, puis en utilisant une redirection.
3. Combien de dossiers y-a-t-il dans `/etc` ?
4. Créer un fichier appartenant à l'utilisateur et au groupe `root`. Grâce aux mécanismes vus précédemment, écrire le contenu de `/etc/passwd` dans ce fichier. Quel est le problème ? Comment le contourner ?

4. Bases du scripting shell

Un script shell dans sa forme la plus simple est un fichier contenant des instructions à exécuter. Il existe deux moyens de lancer un script shell:

1. en le donnant explicitement en paramètre à l'interpréteur:

```
$ bash script.sh
```

2. en précisant un **shebang**, en rendant exécutable le fichier puis en l'exécutant:

```
$ chmod +x script.sh
$ ./script.sh
```

> **À retenir**
>
> Un **shebang** est une ligne au début de votre fichier qui indique quel interpréteur utiliser lors de l'exécution de votre script.

Un shebang est composé des caractères `#!` suivis par le chemin vers l'interpréteur à utiliser. La syntaxe générale est:

```
#! interpreter [optional-arg]
```

Donc, pour utiliser Bash comme interpréteur:

```
#!/bin/bash
```

Nous pouvons donc créer un petit script shell qui affiche « Hello World », comme ceci:

```
#!/bin/bash
echo "Hello World"
```

Les autres lignes commençant par le caractère `#` sont considérées comme étant des **commentaires**: elles ne seront pas interprétées.

> **Exercice**
>
> Écrire un script qui, lors de son invocation, affiche **l'heure** et **la date**, liste tous les **utilisateurs connectés** à la machine, et affiche la durée depuis laquelle le **système est allumé**. Le script inscrira ensuite ces informations dans un fichier.

4.1. Variables

Les variables sont des éléments dans lesquels le système, le shell, l'utilisateur ou des programmes peuvent stocker des données. L'environnement dans le lequel tourne le système contient déjà de nombreuses variables sur son état:

```
#!/bin/bash

echo "Your shell is: $SHELL"
echo "Your user is: $USER"
echo "Your user's home directory is: $HOME"
```

À retenir

Lors de la substitution (récupération de la valeur), la variable doit être précédée par `$`.

Il est également évidemment possible de créer des nouvelles variables nous-même:

```
#!/bin/bash

age=42
echo "Your age is $age"
```

Avertissement

Une mauvaise configuration de certaines variables de l'environnement peut créer des failles de sécurité importantes (exécution de malware...).

Par exemple, lorsque vous tapez une commande dans votre terminal, le shell parcours tous les chemins présents dans `$PATH` à la recherche d'un exécutable du même nom.

```
$ cat << EOF > /tmp/ls
> #!/bin/bash
> echo "pwned"
> EOF
$ chmod +x /tmp/ls
$ export PATH=/tmp:$PATH
$ ls
pwned
```

Le shell met à disposition un certain nombre de variables spéciales:

- `$?` : cette variable contient le code de retour de la commande, de la fonction ou du script qui vient de s'exécuter. Un code de retour égal à 0 signifie, en général, que tout s'est bien déroulé.
- `$$` : cette variable contient le PID du script/programme dans lequel elle est invoquée.
- `$#` : cette variable contient le nombre de paramètres passés au script:

```
$ cat script.sh
#!/bin/bash

echo "$#"
$ ./script.sh
0
$ ./script.sh foo bar
2
```

- `$0`, `$1`, `$2`, … : ces variables contiennent les paramètres passés au script:

```
$ cat script.sh
#!/bin/bash

echo "$1"
$ ./script.sh

$ ./script.sh foo
foo
```

- `$@` : contient tous les paramètres passés au script sous forme de tableau:

```
$ cat script.sh
#!/bin/bash

echo "$@"
$ ./script.sh hello
hello
$ ./script.sh hello world
hello world
```

- `$RANDOM` : retourne un entier non signé aléatoire codé sur 16 bits (0 - 32767):

```
$ echo "$RANDOM"
23750
$ echo "$RANDOM"
12076
```

Attention cependant à ne pas l'utiliser pour des applications de sécurité !

Il existe encore plus de variables managées par Bash. Elles sont listées dans la page de manuel de Bash (`man bash`).

4.2. Conditions

4.2.1. If/elif/else

La construction if/then/else/fi permet d'effectuer des branchements conditionnels:

- en fonction de la valeur de sortie de commandes:

```
#!/bin/bash

target="192.168.0.254"
echo -n "$target is"
if ping -c1 "$target" > /dev/null 2>&1; then
    echo " reachable"
else
    echo " unreachable"
fi
```

- en fonction de la valeur de variables:

```
#!/bin/bash

if [ "$USER" == "root" ]; then
    echo "hi boss"
else
    echo "you're a basic user, meh"
fi
```

Il est possible de cascader les if/else avec le mot clé elif:

```
#!/bin/bash

if [ "$USER" == "root" ]; then
    echo "hi boss"
elif [ "$USER" == "alice" ]; then
    echo "time is a cruel master"
else
    echo "you're a basic user, meh"
fi
```

4.2.2. Switch case

Le switch case est une alternative à la construction if/then/else/fi:

```
#!/bin/bash

command="$1"
name="$2"

case "$command" in
    "create")
        echo "creating file $name"
        touch "$name"
        ;;
    "delete")
        echo "deleting file $name"
        rm "$name"
        ;;
    *)
        echo "unknown command"
        ;;
esac
```

Ce qui donne, si l'on exécute ce script:

```
$ ./script.sh create foo
creating file foo
$ ls
foo script.sh
$ ./script.sh delete foo
deleting file foo
$ ls
script.sh
```

> **Note**
>
> Rajouter le `*)` en tant que dernier case du switch case permet d'intercepter toutes les valeurs qui ne sont pas explicitement définies, et d'afficher une erreur/aide/...

Il est possible d'utiliser le caractère spécial `|` pour regrouper plusieurs motifs dans un même « cas »:

```
#!/bin/bash

command="$1"
name="$2"

case "$command" in
    "create"|"c")
        echo "creating file $name"
        touch "$name"
        ;;
    "delete"|"d")
        echo "deleting file $name"
        rm "$name"
        ;;
    *)
        echo "unknown command"
        ;;
esac
```

4.3. Comparaisons

La commande `test` (ou `[]`, voir `man [`) permet de comparer des valeurs entre elles. On peut faire des comparaisons numériques:

- `eq` : égal (*equal*)
- `ne` : différent (*not equal*)
- `gt` : plus grand que (*greater than*)
- `ge` : plus grand que ou égal à (*greater or equal*)
- `lt` : plus petit que (*less than*)
- `le` : plus petit ou égal à (*less or equal*)

On peut aussi faire des comparaisons de chaînes:

- `==` : égal (chaînes identiques)
- `!=` : différent
- `<` : précède (tri alphabétique)
- `>` : suit (tri alphabétique)

`test` a (entre autres) les options `-f`, `-d`, `-z`, `-p`, `-x`, qui permettent respectivement de:

- `-f` : vérifier qu'un fichier existe.
- `-d` : vérifier qu'un dossier existe.
- `-z` : vérifier qu'une string a un longueur de 0.
- `-p` : vérifier qu'un pipe existe.
- `-x` : vérifier qu'un fichier existe et qu'il est exécutable.

```
#!/bin/bash

conf_file="/etc/foobar.conf"
if [ ! -f "$conf_file" ]; then
    echo "my config" > "$conf_file"
fi
```

Exercice

Écrire un script qui attend au plus 2 paramètres et dans le cas où:

- il en a 0, affiche le dossier courant;
- il en a 1, teste s'il s'agit d'un fichier auquel cas il en affiche le contenu à l'écran et, s'il s'agit d'un dossier, donne la liste des fichiers et dossiers qu'il contient;
- enfin s'il en a deux, vérifie qu'il s'agit de deux fichiers ordinaires et fait une copie du premier dans le deuxième s'il n'existe pas déjà.

Tout autre cas entraîne l'affichage d'un message sur la sortie d'erreur standard.

4.4. Arithmétique

L'arithmétique est un élément essentiel dans la programmation, notamment pour la réalisation de calculs. Bash offre plusieurs principales façons de réaliser des calculs. Par exemple; pour réaliser l'opération $10 \times 20 + 2$:

- la commande `expr` suivie d'une expression:

```
valeur=`expr 10 \* 20`
valeur=`expr $valeur + 2`
```

Note

`expr` n'est pas une primitive du shell, mais bien un programme à part. Voir `type expr`.

- la construction `$[expression]`:

```
valeur=$[10 * 20]
valeur=$[$valeur + 2]
```

- la construction `$((expression))`:

```
valeur=$((10*20))
valeur=$(($valeur+2))
```

- l'utilisation d'outils comme `bc` via des pipes. Cette solution est la plus consommatrice en termes de ressources car elle impliquer de créer des processus.

De manière générale, la construction en `$((expression))` est la plus couramment utilisée.

4.5. Boucles

Il existe deux manières de réaliser des boucles avec Bash, avec deux mots-clés distincts: `while` et `for`.

4.5.1. While

La construction while condition/do/done permet de boucler tant que la condition est **vraie**:

```
#!/bin/bash

num=1
while [ "$num" -le 5 ]; do
    echo "$num"
    num=$(($num+1))
done

host="192.168.1.110"
while ping -c1 "$host" > /dev/null 2>&1; do
    echo "$host is reachable"
    sleep 1
done
echo "$host is unreachable"
```

Dans l'exemple ci-dessus, nous avons deux boucles `while` :

- La première affiche la valeur de `$num` puis l'incrémente tant que `$num` est plus petit ou égal à 5.
- La seconde boucle `while` envoie une requête ICMP_Request toutes les secondes tant que l'hôte est joignable.

Il est possible de créer des boucles infinies avec la construction `while true; do` :

```
#!/bin/bash

while true; do
    echo "Hello ESSIR"
    sleep 1
done
```

Ce script affichera la chaîne de caractères « Hello ESSIR » toutes les secondes jusqu'à son interruption.

Exercice

Écrire un script qui ajoute une chaîne de caractères fixée (le premier paramètre) dans un fichier temporaire nommé `/tmp/f$$` tant que la taille du fichier est inférieure à la valeur du second paramètre. À la fin du programme, la taille sera affichée.

Toute erreur générée sera écrite sur STDERR.

4.5.2. For

La construction en `for` permet d'itérer (de boucler) autour d'une liste d'éléments:

```
for student in Alice Bob Charlie Ed; do
    echo "Hello $student"
done
```

Dans cet exemple, `student` prendra successivement les valeurs de « Alice », « Bob », « Charlie » et « Ed ».

Cette construction peut aussi être utilisée pour boucler autour d'une série d'entiers:

```
for i in {0..10}; do
    echo "current index is $i"
done
```

À chaque itération, `i` sera incrémenté de 1 et ira de 0 à 10.

Par défaut, l'itération autour d'une série d'entier utilise un incrément de 1. Il est cependant possible de spécifier un incrément particulier:

```
echo "I can count from 5 to 5!"
for i in {0..50..5}; do
    echo "$i"
done
echo "see?"
```

Ici, l'incrément est de 5.

Comme dans la plupart des langages, il est possible d'utiliser les mots-clés `continue` et `break` pour contrôler l'exécution de la boucle `for` :

- `continue` stoppe l'itération courante de la boucle et repart au début de la boucle `for` pour la prochaine itération;
- `break` stoppe l'itération courante de la boucle et quitte la boucle `for` .

4.6. Fonctions

Une fonction est une sous-routine, un bloc de code qui implémente un ensemble d'opérations, une « boîte noire » qui exécute une tâche spécifique.

Lorsqu'il y a du **code répétitif**, lorsqu'une tâche se répète avec seulement de légères variations dans la procédure, pensez à utiliser une fonction.

Il existe deux formes pour définir des fonctions:

```
function first_function() {
    # function body...
}

second_function() {
    # function body...
}
```

Une fonction ne doit pas forcément être sur plusieurs lignes, il est possible de la compacter en une seule ligne en utilisant le point-virgule pour terminer les commandes:

```
function greeting() { echo "Hello there :)"; }
```

Dans ce format là (comme lorsque l'on utilise plusieurs commandes en une seule ligne), il faut penser à mettre un `;` après chaque commande, même si il n'y a **qu'une seule commande**.

Nous pouvons accéder aux arguments de la fonction de la même manière que pour accéder aux arguments d'un script, avec les variables `$0`, `$1`, `$2`, `$n` ...

Elles sont surchargées localement lors de l'appel de la fonction, **seulement dans le bloc de la fonction**. Par exemple le script ci-dessous:

```
#!/bin/bash

function greeting() {
    echo "Hello $1"
}

echo "$1"
greeting "ESSIR"
```

donne la sortie suivante:

```
$ ./script.sh foobar
foobar
Hello ESSIR
```

Les fonctions peuvent être définies n'importe où dans le script, mais forcément avant leur premier appel:

```
#!/bin/bash

if [ "$USER" == "bozo" ]: then
    function hello_bozo() {
        echo "Hi Bozo"
    }
fi

hello_bozo
```

Les fonctions peuvent avoir des noms exotiques:

```
_(){ for i in {1..10}; do echo -n "$FUNCNAME"; done; echo; }
```

ou encore:

```
:(){:|:&};:
```

4.7. Substitution de commande

La substitution de commande permet de remplacer une commande par sa sortie. Il existe deux syntaxes pour réaliser une substitution de commande:

```
$(command)
```

ou, plus ancienne:

```
`command`
```

Bash réalise l'expansion en exécutant la commande dans un sous-shell, et remplace la substitution de commande par **la sortie standard** de la commande. Bash tronque automatiquement les retours à la ligne vide.

```
$ echo "my public IP is $(curl -4 ifconfig.co 2>/dev/null)"
my public IP is 69.42.13.37
```

4.8. Exercices

1. Écrire une boucle for qui affiche 30 nombres aléatoires.
2. Écrire un script `rename.sh` qui change l'extension des fichiers finissants par `.jpeg` en `.jpg`.
3. Écrire un script `ex.sh` qui extrait automatiquement une archive avec le bon outil en fonction de l'extension du fichier donné en paramètre. Les extensions supportées devront être: `.zip`, `.tar`, `.gz`, `.bz2`, `.tar.gz`, `.tar.bz2`, `.7z`.
4. Écrire un script `analyze.sh` qui affiche le nombre de paramètres, le nom du script, le troisième paramètre, et la liste de tous les paramètres.

5. Concepts avancés

5.1. Manipulation de strings

Avec Bash, toutes les variables sont manipulées comme étant des chaînes de caractères. Il est donc essentiel de savoir les manipuler.

5.1.1. Connaître la longueur d'une string

- `${#string}`
- `expr length $string`

```
$ cat length.sh
#!/bin/bash

first_var=1337
second_var="foobar"

echo "${#first_var}"
echo "$(expr length $second_var)"
$ ./length.sh
4
6
```

5.1.2. Extraction d'une *substring*

- `${string:position}` : extrait toute la string à partir de la position.
- `${string:position:length}` : extrait une string de longueur donnée à partir de la position.

```
$ cat extract.sh
#!/bin/bash

var=abcABC123ABCabc

echo "${var:0}"
echo "${var:1}"
echo "${var:7:3}"
$ ./extract.sh
abcABC123ABCabc
bcABC123ABCabc
23A
```

5.1.3. Suppression de *substring* - début

- `${string#substring}` : supprime **la plus courte** `substring` du **début** de `string`.
- `${string##substring}` : supprime **la plus longue** `substring` du **début** de `string`.

```
$ cat script.sh
#!/bin/bash

var=abcABC123ABCabc

echo "${var#a*C}"
echo "${var##a*C}"
$ ./script.sh
123ABCabc
abc
```

5.1.4. Suppression de *substring* - fin

- `${string%substring}` : supprime **la plus courte** `substring` de la **fin** de `string`.
- `${string%%substring}` : supprime **la plus longue** `substring` de la **fin** de `string`.

```
$ cat script.sh
#!/bin/bash

var=abcABC123ABCabc

echo "${var%b*c}"
echo "${var%%b*c}"
$ ./script.sh
abcABC123ABCa
a
```

5.1.5. Remplacement de *substring*

- `${string/substring/repl}` : remplace la **première occurrence** de `substring` par `repl`
- `${string//substring/repl}` : remplace **toutes les occurrences** de `substring` par `repl`

```
$ cat script.sh
#!/bin/bash

var=abcABC123ABCabc

echo "${var/abc/foo}"
echo "${var//ABC/bar}"
$ ./script.sh
fooABC123ABCabc
abcbar123barabc
```

5.1.6. Note sur la performance

Toutes ces opérations sur des chaînes de caractères sont réalisables avec un outil très puissant: `sed`. Cependant, l'appel à `sed` dans un script nécessite la création d'un nouveau processus, ce qui peut être coûteux en termes de performance:

```
$ var=abcABC123ABCabc
$ strace -c echo "${var/abc/foo}"
fooABC123ABCabc
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 35.71    0.000045           5         9           mmap
 15.87    0.000020           6         3           mprotect
 10.32    0.000013          13         1           munmap
  6.35    0.000008           1         5           close
  4.76    0.000006           6         1           write
  4.76    0.000006           2         3           brk
  4.76    0.000006           2         3           openat
  4.76    0.000006           1         4           newfstatat
  2.38    0.000003           1         2           pread64
  2.38    0.000003           3         1           getrandom
  1.59    0.000002           1         2         1 arch_prctl
  1.59    0.000002           2         1           set_tid_address
  1.59    0.000002           2         1           set_robust_list
  1.59    0.000002           2         1           prlimit64
  1.59    0.000002           2         1           rseq
  0.00    0.000000           0         1           read
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
------ ----------- ----------- --------- --------- ----------------
100.00    0.000126           3        41         2 total
```

```
$ strace -c sed 's/abc/foo/' <<< "$var"
fooABC123ABCabc
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 32.52    0.000187           6        29           mmap
 27.65    0.000159         159         1           execve
  9.39    0.000054           6         8           openat
  6.61    0.000038           5         7           mprotect
  5.22    0.000030           3        10           newfstatat
  3.83    0.000022           3         7           read
  2.96    0.000017           1         9           close
  2.26    0.000013          13         1           munmap
  1.91    0.000011           5         2           statfs
  1.57    0.000009           3         3           brk
  1.39    0.000008           4         2         1 access
  0.87    0.000005           5         1           write
  0.70    0.000004           2         2           pread64
  0.70    0.000004           2         2         1 arch_prctl
  0.52    0.000003           3         1           futex
  0.52    0.000003           3         1           getrandom
  0.35    0.000002           2         1           set_tid_address
  0.35    0.000002           2         1           set_robust_list
  0.35    0.000002           2         1           prlimit64
  0.35    0.000002           2         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.000575           6        90         2 total
```

On voit ici qu'il y a un peu plus de 2x plus d'appels systèmes lors de l'invocation de `sed` que lors de l'utilisation de la substitution de string native à Bash, pour un tamt d'exécution allant de 0.126ms avec Bash à 0.575ms avec `sed`.

À première vue, cette différence peut sembler négligeable; et dans la plupart des cas elle l'est. Cependant, lors du traitement de gros volumes de données, utiliser la manipulation native à Bash peut apporter un gain de temps considérable.

Exercice

Ré-écrire le script `rename.sh` du chapitre précédent avec les nouvelles notions vues.

5.2. Substitution de processus

Jusqu'à présent, nous avons vu comment piper STDOUT d'un seul processus vers un autre. Avec la substitution de processus, nous pouvons envoyer **STDOUT de plusieurs commandes** vers STDIN d'un processus:

Les commandes doivent êtres enfermées entre parenthèses:

- `<(commands)`
- `>(commands)`

⚠ **Avertissement**

Il n'y a pas d'espace entre le chevron et la parenthèse.

```
$ ls
bar/    foo/
$ diff <(ls bar/) <(ls foo/)
```

La substitution de processus utilise les fichiers `/dev/fd/<n>` pour envoyer les résultats du ou des processus entre parenthèses à un autre processus:

```
$ echo >(true)
/dev/fd/63
```

5.3. Tableaux

Une variable de type *array*, soit un tableau unidimensionnel, peut être initialisé de deux façons différentes:

- avec la notation `array[n]` où `array` est le nom de notre tableau et `n` est un index de celui-ci;
- avec l'instruction explicite `declare -a array` où `declare` est un *builtin* Bash et `array` le nom de notre tableau.

Pour **déréférencer** (récupérer le contenu) d'un élément du tableau, il faut utiliser la notation: `${array[n]}` où `array` est le nom de notre tableau et `n` l'index de l'élément à déréférencer.

Les éléments d'un tableau ne doivent pas forcément être contiguës (c'est à dire être à la suite les uns des autres), ni forcément du même « type »:

```
declare -a area

area[1]=3.1415926536
area[13]=12
area[51]=UFO

echo "$area[51]"
# Output: UFO
```

Il existe d'autres méthodes pour déclarer des variables d'un tableau:

```
#!/bin/bash

arr1=( foo bar baz )
echo "${arr1[0]}"
echo "${arr1[2]}"

arr2=([10]=toor [20]=plop)
echo "${arr2[10]}"
```

donne:

```
$ ./script.sh
foo
baz
toor
```

Bash permet de réaliser des opérations de tableaux sur des variables, **même si les variables ne sont pas explicitement déclarées comme des tableaux**. Le script:

```
#!/bin/bash

var=abcABC123ABCabc
echo "${var[@]}"
echo "${var[*]}"
echo "${var[0]}"
```

a pour sortie:

```
$ ./script.sh
abcABC123ABCabc
abcABC123ABCabc
abcABC123ABCabc
```

Exercice

Quel est le résultat des deux instructions suivantes ? Pourquoi ?

```
#!/bin/bash

var=abcABC123ABCabc
echo "${var[1]}"
```

5.3.1. Récupérer la taille d'un tableau

- `${#array[@]}`
- `${#array[*]}`

```
if [ "${#array[@]}" -eq 0 ]; then
    echo "the array is empty"
fi
```

5.3.2. Ajout d'un élément

- `arr+=("element")`
- `arr[${#arr[@]}]="element"`
- `arr=(${arr[@]}"element")`

```
students=("alice" "bob")
students+=("carol")
echo "${students[@]}" # Output: alice bob carol
```

5.3.3. Suppression d'un élément

- `${arr[@]/element}`

```
students=("alice" "bob" "carol")
del="carol"
students=("${students[@]/$del}")
echo "${students[@]}" # Output: alice bob
```

5.4. Parsing d'options

Pour créer des scripts portables et configurables, il est nécessaire de pouvoir réaliser du *parsing* d'options (*flags*). Les flags sont tous les éléments passés aux programmes qui sont précédés par un ou plusieurs « - ». Par exemple:

```
$ ls -l --color=auto
```

Dans cet exemple, nous passons le flag court `-l` et le flag long `--color=auto` au programme `ls`.

Avec Bash, il existe deux façons principales pour récupérer des options:

- commandes `bash` « natives »;
- `getopts` (à ne pas confondre avec `getopt`).

5.4.1. Parsing d'options natif

Cette méthode se base sur la construction switch case pour récupérer les flags. Ceux-ci sont traités comme des paramètres passés au script, et sont donc récupérés via les variables associées (`$1`, `$2`, `$n`).

```
#!/bin/bash

while [ "$#" -gt 0 ]; do
    case "$1" in
        "-t"|"--target")
            target="$2"
            shift
            ;;
        "-p"|"--port")
            port="$2"
            shift
            ;;
        *)
            echo "unknown parameter passed: $1"
            exit 1
            ;;
    esac
    shift
done

echo "Target is: $target"
echo "Port to use is: $port"
```

```
$ ./script.sh --target plop -p 1337
Target is: plop
Port to use is: 1337
```

L'astuce ici est d'utiliser le *builtin* `shift`, qui décale les paramètres passés au script vers la gauche: `$1` prend la valeur de `$2`, `$2` prend la valeur de `$3`… Et nous répétons l'opération tant que `$#` est plus grand que 0, soit tant qu'il reste des paramètres.

Les avantages de cette méthode:

- explicite;
- simple et rapide à écrire:
- *it works*™.

Mais l'inconvénient principal:

- on ne peut passer que des flags au script.

C'est là que `getopts` rentre en jeu.

5.4.2. `getopts`

`getopts` (à ne pas confondre avec `getopt`) est une primitive shell permettant de *parser* des flags **courts** donnés en paramètre au script.

```
#!/bin/bash

while getopts "abc" opt; do
    case "$opt" in
        "a")
            echo "the flag a has been set"
            ;;
        "b")
            echo "the flag b has been set"
            ;;
        "c")
            echo "the flag c has been set"
            ;;
    esac
done
```

```
$ ./script.sh -a -c
the flag a has been set
the flag c has been set
```

`getopts` met à disposition deux variables réservées:

- `$OPTIND` : contient l'indice de la prochaine option à traiter.
- `$OPTARG` : contient l'argument associé à l'option.

Pour indiquer qu'un flag attend une valeur, il suffit d'ajouter un `:` après sa définition:

```
while getopts "a:" opt; do
    ...
done
```

Ici, nous précisons que `-a` attend une valeur.

Il est possible de détecter les flags non supportés en ajoutant un case pour `\?` :

```
#!/bin/bash

while getopts "a:" opt; do
    case "$opt" in
        "a")
            echo "the 'a' value is $OPTARG"
            ;;
        \?)
            echo "option not supported, exiting..."
            exit 1
            ;;
    esac
done
```

```
$ ./script.sh -a foo -b
the 'a' value is foo
script.sh: illegal option -- b
option not supported, exiting...
```

Comme pour tout script, les options sont stockées dans les paramètres positionnels `$1`, `$2`, `$n`. Afin de pouvoir récupérer des arguments **après** les options, il faut ajouter la ligne suivante après le `while` :

```
shift "$((OPTIND-1))"
```

L'expression `OPTIND-1` représente le nombre d'options analysées, donc la valeur du décalage à réaliser pour enlever tous les flags des paramètres du script.

```
#!/bin/bash

while getopts "t:p:" opt; do
    case "$opt" in
        "t")
            echo "the target is: $OPTARG"
            ;;
        "p")
            echo "the port is: $OPTARG"
            ;;
        \?)
            echo "option not supported, exiting..."
            exit 1
            ;;
    esac
done
shift "$((OPTIND-1))"

echo "the command is: $1"
```

```
$ ./script.sh -t scanme.nmap.org -p 1337 fast_scan
the target is: scanme.nmap.org
the port is: 1337
the command is: fast_scan
```

Le principal avantage de `getopts` est qu'il permet de passer **à la fois des options et des paramètres** au script. Cependant, il ne support que les **options courtes**.

> **Note**
>
> `getopts` est un outil très puissant, mais qui vient également avec son lot de complexité. Lorsque nous arrivons à un stade où autant d'"intelligence" est requise, il est peut être temps de considérer l'utilisation d'un langage de programmation plutôt qu'un langage de scripting (Go, C, Python...).

6. Astuces et bonnes pratiques

6.1. Améliorer la fiabilité des scripts

De base, Bash est très permissif. Par exemple, il est possible par défaut de faire des opérations sur des variables non-initialisées:

```
$ foobarbaz="Hello World"
$ echo "$foobazbar"

$
```

Afin de se prémunir de ce problème et de générer une erreur lors de l'exécution de nos scripts, il faut impérativement ajouter `set -u` au début de nos scripts.

De la même manière, nous pouvons ajouter l'option `set -e` afin de faire quitter notre script si une commande retourne un *exit code* différent de 0, signifiant qu'il y a eu une erreur. Ainsi, nos scripts doivent impérativement commencer par:

```
#!/bin/bash
set -eu

# start of the script...
```

> **Note**
>
> Parfois, il est « normal » qu'un programme retourne un code d'erreur supérieur à 0. Par exemple, si nous souhaitons savoir si un pattern existe dans un fichier, nous allons très probablement utiliser `grep` :
>
> ```
> $ grep "foobarbaz" >bashrc
> $ echo "$?"
> 1
> ```
>
> Nous voyons que `grep` a retourné un code d'erreur de 1. Cependant, il est possible que nous traitions ce cas dans notre script et que nous ne voulions pas qu'il se termine. Il faut alors penser à désactiver la protection avec l'instruction `set +e` .

6.2. Idempotence

> En mathématiques et en informatique, l'idempotence signifie qu'une opération a le même effet qu'on l'applique une ou plusieurs fois.[2]

[2] https://fr.wikipedia.org/wiki/Idempotence

De part leur nature (automatisation, tâches répétitives…), il est important de réaliser des scripts qui soient le **plus idempotents possible**. Il existe plusieurs techniques et bonnes pratiques pour s'en assurer. Quelques exemples:

- `touch` est par défaut idempotent. Pourquoi ?
- `mkdir` en tant que tel n'est pas idempotent: l'appeler génère une erreur sur le répertoire existe déjà). On préférera `mkdir -p` qui ne

génère pas d'erreur si le dossier existe déjà.

- Ajouter le flag `-f` (`--force`) à `rm` permet d'ignorer les fichiers non-existants. **Attention**: cette action peut être **dangereuse** lorsqu'elle est couplée au flag `-r` !
- Parfois les scripts ajoutent des lignes à des fichiers (`echo "foobar" >> file.txt`). Pour éviter de rajouter la ligne plusieurs fois en cas de multiples exécutions du script, on peut par exemple vérifier la présence de la ligne en utilisant `grep`.
- Ne pas hésiter à tester l'existence des fichiers/dossiers/variables avec des `if`.

6.3. Heredocs

Les *Here Documents* sont un type de redirection particulier qui indiquent au shell de lire de l'entrée de la source jusqu'à une ligne qui contient un certain token. Toutes les lignes lues sont redirigées sur l'entrée standard d'une commande.

Le format générique d'un *Here Document* est:

```
<<[-]word
    here-document
delimiter
```

où `word` est le mot en question qui sert de délimiteur. Par exemple:

```
#!/bin/bash

cat << EOF
 _    ___   ___  ___  ___  _  ___
| |  | . \ | __>/ __>/ __>| || . \
| |_ |  _/ | _> \__ \\__ \| ||   /
|___||_|   |___><___/<___/|_||_\_\

EOF
echo "Welcome!"
```

donne:

```
$ ./banner.sh
 _    ___   ___  ___  ___  _  ___
| |  | . \ | __>/ __>/ __>| || . \
| |_ |  _/ | _> \__ \\__ \| ||   /
|___||_|   |___><___/<___/|_||_\_\

Welcome!
```

6.4. Raccourcis clavier

Comme vous allez passer une grande partie de votre temps dans un terminal, apprendre les raccourcis clavier de votre shell est la meilleure façon de devenir plus productif.ve. Voilà les principaux *shortcuts* de Bash:

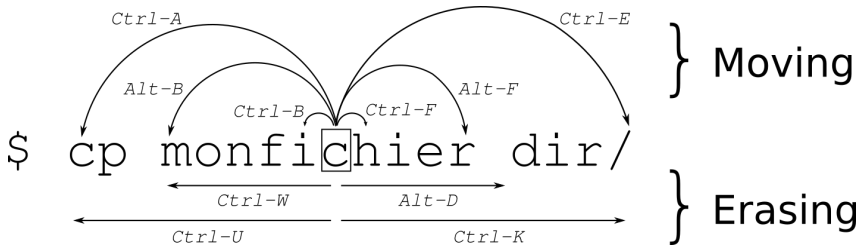

Fig. 3. – Principaux raccourcis claviers de Bash[3]

[3] https://gist.github.com/tuxfight3r/60051ac67c5f0445efee°

7. Licence

Index des figures

Fig. 1: Principales familles de systèmes UNIX[4] 8
Fig. 2: Schématisation des entrée/sorties d'un processus 12
Fig. 3: Principaux raccourcis claviers de Bash[5] 37

[4] https://fr.wikipedia.org/wiki/Unix°
[5] https://gist.github.com/tuxfight3r/60051ac67c5f0445efee°