Introduction au scripting Shell

Programmation pour l'administration

Hugo Blanc

Université Lyon 1 - LP ESSIR 2024-2025

Table of contents

Note

Cette présentation est un complément au fascicule du cours qui lui est plus complet. Il est important de se baser sur le fascicule pour les révisions et avoir des explications plus détaillées.

Durée: 8h de cours + 1h de contrôle + 1h d'introduction au TP noté.

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.

Introduction

Introduction

Le shell est à la fois:

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

Introduction

Les shells ont été présents dès la naissance d'UNIX:

- UNIX 1 livré avec `sh` de Ken Thompson comme Shell par défaut (1971)
- UNIX 7 voit `sh` être remplacé par `bash` de Stephen Bourne (1976)

Programme shell:

Programme shell: **script**.

Programme shell: **script**.

Le shell met à disposition des ***builtins***, des fonctions et commandes permettant de faire des opérations complexes (boucles, conditions, ...).

Programme shell: **script**.

Le shell met à disposition des ***builtins***, des fonctions et commandes permettant de faire des opérations complexes (boucles, conditions, ...).

Grande intégration avec l'environnement Linux: très utile pour du prototypage rapide et de l'administration système.

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

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.

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.

Rappels sur le fonctionnement de Linux

Linux is not UNIX

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.

Linux is not UNIX

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 »

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**;
- 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**.

Philosophie des systèmes UNIX

Sous UNIX, **tout est fichier**.

Sous UNIX, **tout est fichier**.

Votre souris, votre clavier, voter écran, vos connexions TCP… Tout est traité comme étant des fichiers.

Philosophie des systèmes UNIX

Sous UNIX, **tout est fichier**.

Votre souris, votre clavier, voter écran, vos connexions TCP… Tout est traité comme étant des fichiers.

Il est donc possible de manipuler tous les composants du système via de 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**.

Hiérarchie du système de fichiers

En environnement Linux, le système de fichier est sous forme d'arbre, dont la racine est `/`.

```
$ ls -l /
total 124
lrwxrwxrwx   1 root root     7 août  14  2023 bin -> usr/bin
drwxr-xr-x   5 root root  4096 juil. 29 13:51 boot
drwxr-xr-x   2 root root  4096 août  14  2023 cdrom
drwxr-xr-x  23 root root  5540 juil. 31 10:58 dev
drwxr-xr-x 159 root root 12288 juil. 29 13:51 etc
drwxr-xr-x   3 root root  4096 août  14  2023 home
lrwxrwxrwx   1 root root     7 août  14  2023 lib -> usr/lib
lrwxrwxrwx   1 root root     9 août  14  2023 lib32 -> usr/lib32
lrwxrwxrwx   1 root root     9 août  14  2023 lib64 -> usr/lib64
lrwxrwxrwx   1 root root    10 août  14  2023 libx32 -> usr/libx32
drwx------   2 root root 16384 août  14  2023 lost+found
[...]
```

Hiérarchie du système de fichiers

Chaque dossier sous `/` est une branche avec du contenu spécifique (dossiers utilisateur.rices, librairies, exécutables, …).

Hiérarchie du système de fichiers

Chaque dossier sous `/` est une branche avec du contenu spécifique (dossiers utilisateur.rices, librairies, exécutables, ...).

Des informations complètes sur le rôle de chaque branche sont disponibles avec la commande:

```
$ man hier
```

/dev et /proc

Le dossier `/dev` contient les périphériques physiques qui sont présents du point de vue *hardware*. Ils sont appelés **fichiers périphériques** (*devices* en anglais).

Le dossier `/dev` contient les périphériques physiques qui sont présents du point de vue *hardware*. Ils sont appelés **fichiers périphériques** (*devices* en anglais).

```
$ 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 et /proc

Le dossier `/proc` est 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`...).

/dev et /proc

Le répertoire `/proc` contient des sous-répertoires correspondant à l'ID d'un processus en cours d'exécution.

Dans chacun de ces sous-répertoires, sont stockées des informations utiles sur le processus correspondant (environnement chargé, avec le fichier `/proc/pid/environ`...).

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.

Fonctionnalités de base du Shell

Redirections d'E/S

Par défaut, tout processus possède:

- une entrée;
- deux sorties;

Redirections d'E/S

Par défaut, tout processus possède:

- une entrée;
- deux sorties;

Ces E/S sont **distinctes** pour chaque processus.

Redirections d'E/S

- **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. 1. – Schématisation des entrée/sorties d'un processus

Redirections d'E/S

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).

Redirections d'E/S

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.

Redirections d'E/S

Il est possible de coupler plusieurs redirections:

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

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

Redirections d'E/S

On peut référencer un fd en utilisant le caractère &:

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

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

Pipes

Les pipes permettent d'envoyer la sortie d'un processus (STDOUT) à l'entrée d'un autre (STDIN):

```
processus1 | processus2 | processus3
```

Pipes

- `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`.

Pipes

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.

Pipes (*internals*)

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

Pipes (*internals*)

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.

Pipes (*internals*)

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`.

Exemple

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

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

Pipes (*internals*)

- 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.

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 ?

Bases du scripting Shell

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
```

Un shebang est les 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 »:

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

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

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.

Variables

Les variables permettent de…

Variables

Les variables permettent de… **stocker de la donnée**.

Variables

De nombreuses variables sont déjà configurées par l'environnement:

```
#!/bin/bash

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

Voir la commande env.

Variables

Il est bien sûr possible de créer de nouvelles variables:

```
#!/bin/bash

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


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

Variables

⚠

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

Voir le PoC dans le cours.

Variables

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

- `$?` : contient le code de retour de la commande, de la fonction ou du script qui vient de s'exécuter.
- `$$` : contient le PID du script/programme dans lequel elle est invoquée.
- `$#` : contient le nombre de paramètres passés au script.
- `$0`, `$1`, `$2`, … : contiennent les paramètres passés au script.
- `$@` : contient tous les paramètres passés au script sous forme de tableau.
- `$RANDOM` : retourne un entier non signé aléatoire codé sur 16 bits (0 - 32767). Attention, ce « générateur » n'est pas cryptographiquement sûr !
- et d'autres encore (voir `man bash`).

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
```

Conditions

- 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
```

Conditions

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
```

Switch case

Alternative à la construction if/then/else/fi. Elle teste plusieurs « cas » de manière séquentielle, et exécute le code défini dans le premier qui matche.

Conditions

```
#!/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
```

> *i*
>
> 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/...

Conditions

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

Conditions

```
#!/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
```

Comparaisons

La commande `test` (ou `[]`, voir `man [`) permet de comparer des valeurs entre elles.

Comparaisons

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*)

Comparaisons

Des comparaisons de chaînes:

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

Comparaisons

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

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

Comparaisons

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.

Arithmétique

Bash propose plusieurs façons de réaliser des calculs.

Arithmétique

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

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

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

Arithmétique

- la construction `$[expression]`:

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

Arithmétique

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

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

- l'utilisation d'outils comme `bc` via des pipes (perte de performance car création de processus).

Arithmétique

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

Boucles

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

While

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

Boucles

```
#!/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"
```

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.

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.

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.

Il est cependant possible de spécifier un incrément particulier, par exemple 5:

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

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`.

Fonctions

Une fonction est…

Fonctions

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

Fonctions

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

Elles sont utilisées lorsqu'il y a du **code répétitif**.

Fonctions

Il existe deux formes pour définir des fonctions:

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

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

Il est possible de compacter une fonction sur une seule ligne en utilisant le point-virgule pour terminer les commandes:

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

Les arguments de la fonction peuvent être récupérés comme des variables: `$0`, `$1`, `$2`, `$n`...

Elles sont surchargées localement lors de l'appel de la fonction !

Fonctions

Par exemple:

```
#!/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
```

Fonctions

Les fonctions peuvent avoir des noms exotiques:

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

ou encore:

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

Substitution de commande

La substitution de commande permet de remplacer une commande par sa sortie.

Il existe deux syntaxes:

```
$(command)
```

ou, plus ancienne:

```
`command`
```

Substitution de commande

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.

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

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.

Concepts avancés

Manipulation de strings

Avec Bash, toutes les variables sont manipulées comme étant des chaînes de caractères.

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
```

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
```

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
```

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
```

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
```

Note sur la performance

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 (voir benchmarks dans le fascicule).

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

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.

Substitution de processus

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

Substitution de processus

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
```

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.

Tableaux

Les éléments d'un tableau ne doivent pas forcément être contiguës, 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
```

Tableaux

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

Tableaux

Le script:

```
#!/bin/bash

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

a pour sortie:

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

Quel est le résultat des deux instructions suivantes ? Pourquoi ?

```
#!/bin/bash

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

Récupérer la taille d'un tableau

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

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

Ajout d'un élément

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

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

Suppression d'un élément

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

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

Parsing d'options

Les flags sont tous les éléments passés aux programmes qui sont précédés par un ou plusieurs « - »:

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

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

Parsing d'options

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

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

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`).

Parsing d'options

```
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
```

Le *builtin* `shift`, qui décale les paramètres passés au script vers la gauche.

Nous répétons l'opération tant que `$#` est plus grand que 0, soit tant qu'il reste des paramètres.

Parsing d'options

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.

`getopts`

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

Parsing d'options

```
#!/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
```

Parsing d'options

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

Parsing d'options

`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.

Parsing d'options

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

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

Parsing d'options

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
```

Parsing d'options

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

Parsing d'options

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.

Parsing d'options

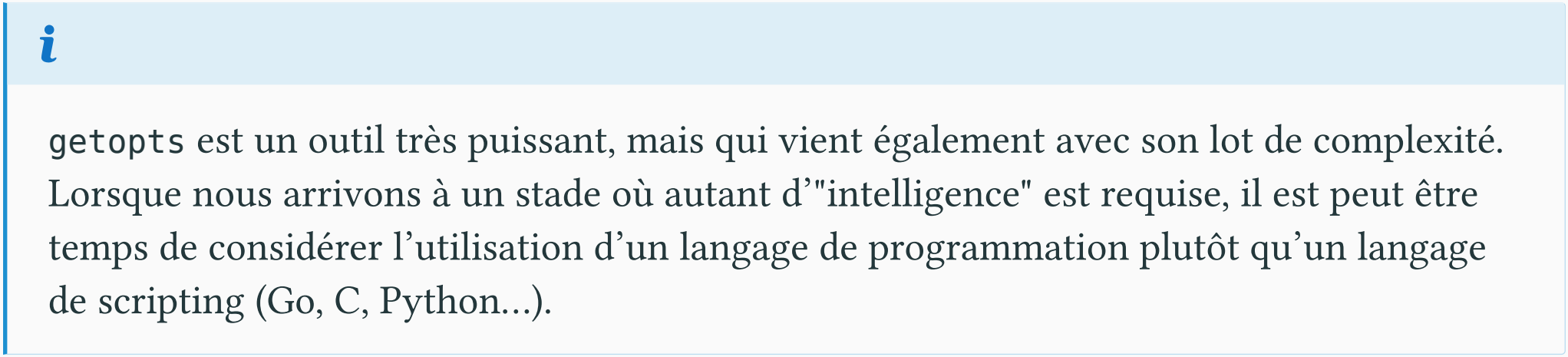
```
#!/bin/bash

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"
```

Parsing d'options

```
$ ./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
```

Parsing d'options

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



`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…).

Astuces et bonnes pratiques

Améliorer la fiabilité des scripts

Bash est très permissif:

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

$
```

Améliorer la fiabilité des scripts

Il est possible de le rendre plus exigeant avec `set`:

- `set -u`
- `set -e`
- ...

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

# start of the script...
```

i

Parfois, il est « normal » qu'un programme retourne un code d'erreur supérieur à 0:

```
$ grep "foobarbaz" >bashrc
$ echo "$?"
1
```

On peut désactiver la protection avec l'instruction `set +e`.

Idempotence

« En mathématiques et en informatique, l'idempotence signifie qu'une opération a le même effet qu'on l'applique une ou plusieurs fois.[1] »

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

Idempotence

Il est important de réaliser des scripts qui soient le **plus idempotents possible**

- `touch` est par défaut idempotent. Pourquoi ?
- `mkdir -p`
- `rm -f`
- ...

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.

Heredocs

```
#!/bin/bash

cat << EOF

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

EOF
echo "Welcome!"
```

donne:

Heredocs

```
$ ./banner.sh

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

Welcome!
```

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.

Ils sont détaillés dans le fascicule du cours.

Licence

Licence