Base de Bash

Shell

Sous Linux, un shell est l'interpréteur de commandes qui fait office d'interface entre l'utilisateur et le système d'exploitation. Il s'agit d'interpréteurs, cela implique que chaque commande saisie par l'utilisateur et vérifiée puis exécutée. Nous avons déjà parlé des différents types de shell existants (csh, tcsh, sh, bash, ksh, dash, etc.). Nous travaillerons plus avec le bash (Bourne Again SHell).
Pour rappel, un shell possède deux principaux aspects :

- Un aspect environnement de travail
- Un aspect langage de programmation

L'environnement

L'environnement de travail d'un shell doit être agréable et puissant (rappel CLI GUI), bash permet entre autres choses de :

- Rappeler des commandes précédentes (historique)
- Modifier en ligne du texte de la commande courante (bi, emacs, nano)
- Gestion des travaux lancés en arrières-plan (jobs)
- Initialisation adéquate des variables de configuration (chaîne d'appel de l'interpréteur, chemins de recherche par défaut)

Pour illustrer ça, voyons que le shell permet d'exécuter une commande en mode interactif ou bien par l'intermédiaire de fichiers de commandes (scripts). En mode interactif, bash affiche à l'écran une chaîne d'appel (appelée prompt ou invite) qui se termine par défaut par le caractère # pour l'administrateur (root), et par le caractère $ pour les autres utilisateurs. Sous windows le prompt est souvent le nom du chemin où l'on se trouve, suivis de > , par exemple : " **C:\Windows\System32>**"

Un langage de programmation

Les shells ne sont pas uniquement des interpréteurs de commandes mais de véritables langages de programmation, un shell comme le bash intègre :

- les notions de variables
- les opérateurs arithmétiques
- les structures de contrôle
- les fonctions
- etc.

Par exemple :

```
$ mavariable=5 #=> affectation de la valeur 5 dans la variable qui a pour nom “mavariable”


$ echo $((mavariable +3)) #=> affiche la valeur de l’expression mavariable
8
```

Avantages et inconvénients des shells

L’étude d’un shell comme le bash dispose de plusieurs avantages :

- langage interprété, il est facile de trouver les erreurs et de les traiter
- modification rapide sans besoin de recompiler
- langage orienté chaîne de caractères (pas de pointeurs), moins de risque d’avoir des erreurs d’adressage.
- prototypage rapide d’application, il est facile de composer des programmes avec les commandes existantes et l’utilisation des tubes et substitution dans l’environnement Unix
- langage “glu”: on peut connecter des composants écrits dans des langages différents. Ils doivent juste respecter certains standards :
  - lire sur l’entrée standard
  - accepter des arguments et options éventuels,
  - écrire ses résultats sur la sortie standard
  - écrire les messages d’erreurs sur la sortie dédiée au erreur.

Voyons maintenant certains inconvénients des shells :

- La syntaxe est “ésotérique” et d’accès difficile pour les débutants
- Suivant le contexte, l’ajout ou l’oubli d’un espace peut provoquer des erreurs de syntaxe ou de traitement
- Plusieurs syntaxes pour implanter la même fonctionnalité (principalement à cause de la compatibilité ascendante avec le Bourne Shell
- Le sens de certains caractères spéciaux, comme les parenthèses, change avec le contexte, elle peuvent définir :
  - une liste de commandes
  - une définition de fonction
  - imposer un ordre d’évaluation
  - une expression arithmétique

Si vous voulez connaître le shell qui tourne actuellement, utilisez la commande ps, et si vous voulez connaître la version du bash :
`bash -- version`

Variables

Ils existent différents types de variables utilisables dans le shell. Elles sont identifiées par un nom (suite de lettres, chiffres, caractères espace

ou souligné ne commençant pas par un chiffre. la casse est gérée). On peut classer 3 groupes de variables :

- utilisateurs (ex: a, valeur)
- prédéfinies par le shell (ex; PS1, PATH, REPLY, IFS, HOME)
- prédéfinies pour les commandes unix (ex: TERM)

Pour affecter une variable, on peut utiliser l'opérateur = ou bien la commande interne read (pour demander une saisie utilisateur)

Les variables locales :

Elles ne sont disponibles que dans l'instance du shell dans lesquelles elles ont été créées. (Elles ne sont pas utilisées par les commandes dans ce shell). Par défaut une variable est créée en tant que variable locale, on utilise couramment des lettres minuscules pour nommer ses variables locales.
Exemple : `mavariablelocale = 1`

Les constantes :

Une constante est une variable en lecture seul d'une certaine manière, elle n'a pas pour but d'être modifié dans le programme (d'où son nom). Pour créer une constante, vous pouvez utiliser la commande declare -r.
Exemple: `declare -r pi=3.14159`

Les variables d'environnement :

Les variables d'environnement existent dans le shell pour lequel elles sont créées, mais aussi pour toutes les commandes qui sont utilisées dans ce shell. On utilise couramment des majuscules pour nommer ses variables d'environnement.
Exemple1 :

```
#Transformer une variable
ENVVAR=10 #Création d'une variable locale
export ENVVAR #Transforme la variable locale en variable d'environnement
```

Exemple 2:

```
# Créer une variable d'environnement
export ENVVAR2 = 11     # Première solution
declare -x ENVVAR3 = 12 # Deuxième solution
typeset -x ENVVAR4 = 13 # Troisième solution
```

Commandes utiles pour les variables :

- echo : Vous pouvez utiliser la commande echo si vous souhaitez connaître le contenu d'une variable.

Exemple : `echo $PATH` permettra d'afficher le contenu de la variable d'environnement PATH qui contient les chemins de fichier de
commande dans le shell.

```
$ set param1 param2
$ echo $1
param1
$ set --
$ echo $1
$ #on a perdu les valeurs
```

Pour connaître le nombre de variables de position, il existe une variable spéciale $#

- shift : permet de décaler les variables de position (sans toucher au $0)

```
$ set a b c d e f g h i j # param 1 2 3 4 5 6 7 8 9
$ echo $1 $2 $#
a b 9
$ shift 2 # variable deviennent c d e f g h i j
$ echo $1 $2 $#
c d 7
```

L'utilisation du shift sans argument équivaut à faire un `shift 1`

- unset : permet de supprimer une variable

```
$ set myvar=1
$ echo $myvar
1
$ unset myvar
$ echo $myvar
$
```

Mon premier bash

```
#!/bin/bash
echo “Nom du programme : $0”
echo “Nombre d'arguments : $#”
echo “Source : $1”
```

```
echo “Destination $2”
cp $1 $2
```

```
$ chmod u+x monpremierbash.sh
$
$ monpremierbash.sh /etc/passwd /root/copiepasswd
Nom du programme : ./monpremierbash.sh
Nb d’arguments : 2
Source : /etc/passwd
Destination : /root/copiepasswd
$
```

```
$ set un deux trois quatre
$
$ echo $* # affiche tous les arguments
un deux trois quatre
$ echo $@ # affiche tous les arguments
un deux trois quatre
$ set un “deux trois” quatre # testons avec 3 paramètres et des guillemets
$ set “$*” # équivalent à set “un deux trois quatre”
$ echo $#
1
$ echo $1
un deux trois quatre
$ set un “deux trois” quatre # testons $@ avec 3 paramètres et des guillemets
$ set “$@” # équivalent à set “bonjour” “deux trois” “quatre”
$ echo $#
3
$ echo $2
deux trois
```

Si dans un bash on souhaite supprimer les ambiguïtés d’interprétation des paramètres de position, on utilise le ${paramètre}, comme dans l’exemple suivant.
Exemple :

```
$ x=bon
$ x1=jour
$ echo $x1
jour
$ echo ${x}1
```

```
bon1
$ set un deux trois quatre cinq six sept huit neuf dix onze douze
$ echo $11
un1
$ echo ${11}
onze
```

Indirections

Bash offre la possibilité d'obtenir la valeur d'une variable v1 dont le nom est contenu "v1" dans une autre variable mavar. Il suffit pour cela d'utiliser la syntaxe de substitution : ${!mavar}.
Exemple :

```
$ var=v1
$ v1=un
$
$ echo ${!var}
un
$
```

Ce mécanisme, appelé indirection,permet d'accéder de manière indirecte et par conséquent de façon plus souple, à la valeur d'un
deuxième objet. Voyons un autre exemple d'utilisation :
Exemple d'un fichier indir:

```
#!/bin/bash
agePierre=10
ageJean=20
read -p "Quel âge (Pierre ou Jean) voulez-vous connaître ? " prenom
rep=age$prenom #construction du nom de la variable
echo ${!rep}
$ indir
Quel âge (Pierre ou Jean) voulez-vous connaître ? Pierre
10
$ indir
Quel âge (Pierre ou Jean) voulez-vous connaître ? Jean
20
$
```

Ce mécanisme s'applique également aux deux autres types de paramètres : les paramètres de position et les paramètres spéciaux ($1, $2, , ...)

Résultats, Code de retour et opérateur sur les code de retour

Il ne faut pas confondre le résultat d'une commande et son code de retour : le résultat correspond à ce qui est écrit sur sa sortie standard; le code de retour indique uniquement si l'exécution de la commande s'est bien effectuée ou non. Parfois, on est intéressé uniquement par le code de retour d'une commande et non par les résultats qu'elle produit sur la sortie standard ou la sortie d'erreur.
Exemple :

```
$ grep toto pass > /dev/null 2>&1 #=> ou bien : grep toto pass &>/dev/null
$
$ echo $?
1 #=> on en déduit que la chaîne toto n'est pas présente dans pass
```

Les opérateurs && et || autorisent l'exécution conditionnelle d'une commande cmd suivant la valeur qu'a pris le code de retour de la dernière commande précédemment exécutée.

Exemple pour && :

```
$ grep toto pass > /dev/null 2>&1 #=> ou bien : grep toto pass &>/dev/null
$
$ echo $?
1 #=> on en déduit que la chaîne toto n'est pas présente dans pass
```

La chaîne de caractères daemon est présente dans le fichier pass, le code de retour renvoyé par l'exécution de grep est 0; par conséquent, la commande echo est exécutée.
Exemple pour || :

```
$ ls pass tutu
ls : impossible d'accéder à tutu: Aucun fichier ou dossier de ce type pass
$ rm tutu || echo tutu non effacé
rm : impossible de supprimer tutu: Aucun fichier ou dossier de ce type
tutu non effacé
$
```

Le fichier tutu n'existant pas, la commande rm tutu affiche un message d'erreur et produit un code de retour différent de 0; la commande interne echo est exécutée.
Exemple combiné || et || :

```
$ ls pass || ls tutu || echo fin aussi
pass
$
```

Le code de retour ls pass est égal à 0 car pass existe, la commande ls tutu ne sera pas exécutée. D'autre part le code de retour de l'ensemble ls pass || ls tutu est le code de retour de la dernière commande exécutée, c'est-à-dire 0 (ls pass). donc echo fini aussi n'est pas exécutée.
Exemple combiné && et || :

```
$ ls pass || ls tutu || echo suite et && echo fin
pass
fin
$
```

la commande ls pass a un code de retour égal à 0, donc la commande ls tutu ne sera pas exécutée; le code de retour de l'ensemble ls pass || ls tutu sera donc égal à 0. la commande echo suite et n'est pas exécutée donc le code de retour de l'ensemble reste 0 echo fin sera donc exécutée.

Boucles et structure de contrôle

case, structure de choix multiple

Syntaxe :

```
case mot in
[modele [ | modele] ...) suite de commandes ;; ] ...
esac
```

Le shell va étudier la valeur de mot puis la comparer séquentiellement à chaque modèle. Quand un modèle correspond à mot, la suite de commandes associée est exécutée, terminant l'exécution de la commande interne case. Les mots case et esac sont des mots-clés; ils doivent être le premier mot d'une commande. suite de commandes doit se terminer par 2 caractères ; collés de manière à ce qu'il n'y ait pas d'ambiguïté avec l'enchaînement séquentiel de commande cmd1; cmd2; etc. Quand au modèle , il peut-être construit à l'aide de caractères et expressions génériques de bash (*, . , [], etc.). Dans ce contexte le symbole | signifiera OU. Pour indiquer le cas par défaut (si aucun des autres ne survient) on utilisera le modèle *. il doit être placé en dernier modèle. Le code de retour de la commande composée case est égal à 0, si aucun modèle n'a pu correspondre à la valeur de mot. Sinon, c'est celui de la dernière commande exécutée de suite de commandes.

Exemple 1: Programme shell oui affichant OUI si l'utilisateur a saisi le caractère o ou O

```
#!/bin/bash
read -p "Entrez votre réponse : " rep
case $rep in
o|O ) echo OUI ;;
* ) echo Indefini
esac
```

Exemple 2 : Programme shell nombre prenant une chaîne de caractères en argument, et qui affiche cette chaîne si elle est constituée d'une suite de chiffres. ([:digit:] , [:upper:], [:lower:], [:alnum:]

```
#!/bin/bash
# on autorise l'utilisation des expressions générique
shopt -s extglob
case $1 in
+([[:digit:]]) ) echo ″$1 est une suite de chiffres″ ;;
esac
```

Exemple 3 : Si l'on souhaite ignorer la casse on peut modifier le flash de shopt

```
#!/bin/bash
read -p "Entrez un mot : " mot
shopt -s nocasematch
case $mot in
oui ) echo ″Vous avez écrit oui" ;;
* ) echo "$mot n'est pas le mot oui" ;;
esac
```

While

La commande interne while correspond à l'itération 'faire - tant que' présente dans de nombreux langage de programmation.
Syntaxe :

```
while suite_cmd1
do
suite_cmd2
done
```

La suite de commandes suite_cmd1 est exécutée, si son code de retour est égal à 0, alors la suite de commande suite_cmd2 est exécutée, puis suite_cmd1 est re-exécutée. Si son code de retour est différent de 0, la suite se termine. L'originalité de cette méthode est que le test ne porte pas sur une condition booléenne, mais sur le code de retour issu de l'exécution d'une suite de commandes. Une commande while, comme toutes commandes internes, peut être écrite directement sur la ligne de commande.
Exemple :

```
$ while who | grep root> /dev/null
> do
```

```
> echo “Utilisateur root est connecté”
> sleep 5
> done
Utilisateur root est connecté
Utilisateur root est connecté
Utilisateur root est connecté
^C
$
```

Commande interne while est :
La commande interne : associée à une itération while compose rapidement un serveur (démon) rudimentaire.
Exemple :

```
$ while : #=> Boucle infinie
> do
> who | cut -d’ ‘-f1 > fic #=> Traitement à effectuer
> sleep 300 #=> Temporiser
> done &
[1] 1123 #=> pour arrêter l’exécution kill -15 1123
$
```

On peut parfois utilisée la commande while pour lire un fichier texte. La lecture se fait alors ligne par ligne. Pour cela, il suffit de :

- placer la commande read dans suite_cmd1
- de placer les commandes de traitement de la ligne courante dans suite_cmd2
- de rediriger l’entrée standard de la commande while avec le fichier lire

Syntaxe :

```
while read [var1 ...]
do
Commandes de traitements
done < fichier à lire
```

Exemple :

```
#!/bin/bash
who > tmp
while read nom divers
do
```

```
echo $nom
done < tmp
rm tmp
```

Notez que l'utilisation du while pour lire un fichier n'est pas très performante. On préférera en général utiliser une suite de filtre pour obtenir les résultats voulus (cut, awk, ...)

Modificateur de chaîne

Échappement

Différents caractères particuliers servent en shell pour effectuer ses propres traitements ($ pour la substitution, > pour la redirection, * en joker). Pour utiliser ces caractères particuliers en tant que simple caractère, il faut les échapper en les précédant du caractère \
Exemple :

```
$ ls
tata toto
$ echo *
tata toto
$ echo \*
*
$ echo \\
\
$
```

Autre particularité, le caractère \ peut aussi échapper les retours à la ligne. On peut donc aller à la ligne sans exécuter la commande.
Comme nous l'avons déjà vu, les caractères " et ' permettre une protection partielle, ou total (') d'une chaîne de caractères.
Exemple :

```
$ echo "< * $PWD ' >"
< * /root ' >
$
$ echo "\"$PS2\""
"> "
$ echo '< * $PWD " >'
< * $PWD " >'
$ echo c'est lundi
>
> '
```

```
cest lundi
$ echo c\’est lundi
c’est lundi
$ echo “c’est lundi”
c’est lundi
```

Chaîne de caractères longueur

Syntaxe : `${#paramètre}`
Cette syntaxe permet d’obtenir la longueur d’une chaîne de caractères.
Exemple :

```
$ echo $PWD
/root
$ echo ${#PWD}
5
$ set “Bonjour à tous”
$ echo ${#1}
14
$ ls > /dev/null
$
$ echo ${#?}
1 #=> la longueur du code de retour (0) est de 1 caractère
```

Chaîne de caractères modificateur

On peut modifier les chaîne de caractères directement :
Syntaxe : ${paramètre#modèle} pour supprimer la plus **courte** sous-chaîne à gauche
Exemple :

```
$ echo $PWD
/home/christophe
$ echo ${PWD#*/}
home/christophe #=> le premier caractère / a été supprimé
$
$ set “25b75b”
$
$ echo ${1#*b}
75b #=> Suppression de la sous-chaîne 25b
```

Syntaxe: `${paramètre##modèle}` pour supprimer la plus **longue** sous-chaîne à gauche
Exemple :

```
$ echo $PWD
/home/christophe
$ echo ${PWD##*/}
christophe #=> suppression jusqu'au dernier caractère /
$
$ set "25b75b"
$
$ echo ${1##*b}
b
```

Pour la suppression par la droite, c'est la même chose en utilise le caractère % comme contrôle
Syntaxe :
`${paramètre%modèle}` pour supprimer la plus courte sous-chaîne à droite
`${paramètre%%modèle}` pour supprimer la plus longue sous-chaîne à droite

On peut extraire une sous-chaîne également :
Syntaxe:
`${paramètre:ind}` : extrait la valeur de paramètre de la sous-chaîne débutant à l'indice ind.
`${paramètre:ind:nb}` : extrait nb caractères à partir de l'indice ind
Exemple :

```
$ lettres="abcdefghijklmnopqrstuvwxyz"
$
$ echo {$lettre:20}
uvwxyz
$ echo {$lettre:3:4}
defg
$
```

Remplacement dans une sous-chaîne
Syntaxe:
`${paramètre/mod/ch}`
bash recherchera dans paramètre la plus longue sous-chaîne satisfaisant le modèle mod puis remplacera cette sous-chaîne par
la chaîne ch. Seule la première sous-chaîne trouvée sera remplacée. mod peut être des caractères ou expressions génériques.
`${paramètre//mod/ch}` :
Pour replacer toutes les occurrences et pas seulement la première
`${paramètre/mod/}` :
`${paramètre//mod/}` :
Supprime au lieu de remplacer
Exemple :

```
$ v=totito
$ echo {$v/to/lo}
lotito
$ echo {$v//to/lo}
lotilo
```

Structure de contrôle for et if

Itération et for

Syntaxe 1:

```
for var
do
suite_de_commandes
done
Syntaxe 2:
for var in liste_mots
do
suite_de_commandes
done
```

Dans la première forme, la variable var prend successivement la valeur de chaque paramètre de position initialisé
Exemple :

```
$ cat for_args.sh
#!/bin/bash
for i
do
echo $i
echo “next ...”
done

$ ./for_args.sh first second third
first
next ...
second
next ...
third
```

```
next ...
```

Exemple 2:

```
$ cat for_set.sh
#!/bin/bash
set $(date)
for i
do
echo $i
done

$ ./for_set.sh
samedi
29
Juin
2019,
12:09:21
(UTC+0200)
$
```

La deuxième syntaxe permet à var de prendre successivement la valeur de chaque mot de liste_mots.
Exemple :

```
$ cat for_liste.sh
#!/bin/bash
for a in toto tata
do
echo $a
done
```

Si liste_mots contient des substitutions, elles sont préalablement traitées par bash
Exemple 2:

```
$ cat affiche_ls.sh
#!/bin/bash
for i in /tmp ${pwd}
do
echo “ --- $i ---”
ls $i
```

```
done

$ ./affiche_ls.sh
--- /tmp ---
toto tutu
--- /home/christophe
for_liste.sh affiche_ls.sh alpha tmp
```

If et le choix

La commande interne if implante le choix alternatif
Syntaxe :

```
if suite_commande1
then
suite_commande2
[elif suite_de_commandes; then suite_de_commande] ...
[else suite_de_commandes]
fi
```

Le principe de fonctionnement est le même que pour le for, on test la valeur de retour d'une commande plutôt qu'une valeur booléenne simple. Donc dans notre exemple, suite_commande2 est exécuté, si suite_commande1 renvoi 0 (pas d'erreur). Sinon c'est elif ou bien else qui sera exécuté.
Exemple :

```
$ cat rm1.sh
#!/bin/bash
if rm "$1" 2> /dev/null
then echo $1 a été supprimé
else echo $1 n\'a pas été supprimé
fi

$ >toto
$ rm1 toto
toto a été supprimé
$
$ rm1 toto
toto n'a pas été supprimé
$
```

Lorsque rm1 toto est lancé, si le fichier toto est effaçable, il le sera, et la commande rm renvoi 0.
Notez qu'il est possible d'imbriquer les if ensembles
Exemple :

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

Tests

Dans les bash, vous retrouverez souvent une notation de commande interne [[souvent utilisé avec le if. Elle permet l'évaluation de d'expressions conditionnelles portant sur des objets aussi différents que les permissions sur une entrée, la valeur d'une chaîne de caractères ou encore l'état d'une option de la commande interne set.
Syntaxe: `[[ expr_conditionelle ]]`
Les deux caractères crochets doivent être collés et un caractère séparateur doit être présent de part et d'autre de expr_conditionelle. Les mots [[et]] sont des mots-clés. On a vu que le if fonctionne selon la valeur de retour d'une commande, et pas d'un booléen, cette syntaxe permet "d'exécuter un test" qui renverra 1 si le test est vrai, 0 sinon. Si l'expression contient des erreurs syntaxique une autre valeur sera retournée. La commande interne [[offre de nombreuses expressions conditionnelles, c'est pourquoi seules les principales formes de exp_conditionnelle seront présentées, regroupées par catégories.

Permission

-r entrée vraie si entrée existe et est accessible en lecture par le processus courant
-w entrée vraie si entrée existe et est accessible en écriture par le processus courant
-x entrée vraie si entrée existe et est accessible en exécutable par le processus courant ou si le répertoire entrée existe et

le processus courant possède la permission de passage
Exemple :

```
$ echo coucou > toto
$ chmod 200 toto
$ ls -l toto
--w- --- --- 1 christophe christophe 29 juin 1 14:04 toto
$
$ if [[ -r toto ]]
> then cat toto
```

```
> fi
$ => rien ne ce passe
$ echo $?
0 => code de retour de la commande interne if
$
MAIS
$ [[ -r toto ]]
$ echo $?
1 => code de retour de la commande interne [[
$
```

Type d’une entrée

-f entrée vraie si entrée existe et est un fichier ordinaire
-d entrée vraie si entrée existe et est un répertoire
Exemple :

```
$ cat afficher.sh
#!/bin/bash
if [[ -f “$1” ]]
then
echo “$1” : fichier ordinaire
cat “$1”
elif [[ -d “$1” ]]
then
echo “$1” : répertoire
ls “$1”
else
echo “$1” : type non traité
fi

$ ./afficher
. : répertoire
afficher.sh test.sh toto alpha rm1.sh
$
```

Renseignement divers sur une entrée
-a entrée vraie si entrée existe
-s entrée vraie si entrée existe et sa taille est différente de 0 (un répertoire vide > 0)
entrée 1 -nt entrée 2 vraie si entrée 1 existe et sa date de modification est plus récente que celle de entrée2
entrée1 -ot entrée 2 vraie si entrée1 existe et sa date de modification est plus ancienne que celle

de entrée2
Exemple :

```
$ > err
$
$ ls -l err
-rw-rw-r-- 1 christophe christophe 0 juin 29 14:30 err
$ if [[ -a err ]]
> then echo err existe
> fi
err existe
$ if [[ -s err ]]
> then echo err n'est pas vide
> else err est vide
> fi
err est vide
$
```

Longueur d'une chaîne de caractère

-Z ch vraie si la longueur de ch est égale à 0
ch ou (-n ch) vraie si la longueur de ch est différente de 0
ch1 < ch2 vraie si ch1 précède ch2
ch1 > ch2 vraie si ch1 suit ch2
ch == mod vraie si la chaîne ch correspond au modèle mod
ch != mod vraie si la chaîne ch ne correspond pas au modèle mod
-o opt vraie si l'option interne opt est sur on

> Important : il existe un opérateur =~ qui permet de mettre en correspondance une chaîne de caractères ch avec une expression
> régulière.

Exemple :

```
$a=01/01/2010
$[[ $a =~ [0-9]{2}\/[0-9]{2}\/[0-9]{2,4} ]]
$ echo $?
0
$ a=45/54/1
$[[ $a =~ [0-9]{2}\/[0-9]{2}\/[0-9]{2,4} ]]
$ echo $?
```

```
1
```

Expressions conditionnelles

(cond) vraie si cond est vraie
! cond vraie si cond est fausse
cond1 && cond 2 vraie si cond1 et 2 sont vraie, l’évaluation s’arrête si cond1 est fausse
cond1 || cond2 vraie si cond1 ou 2 sont vraie.

Exemple :

```
$ ls -l /etc/at.deny
-rw-r----- 1 root daemon 144 oct. 25 2018 /etc/at.deny
$
$ if [[ ! ( -w /etc/at.deny || -r /etc/at.deny ) ]]
> then
> echo OUI
> else
> echo NON
> fi
OUI
```

Revision #1
Created 31 October 2019 13:02:19 by Cécile
Updated 31 October 2019 13:02:57 by Cécile