SCRIPTS SHELL

SHA-BANG

On commence un script Shell par un **sha-bang**, c'est non obligatoire mais cela indique à quelle sorte de bash on s'adresse

```
#!/bin/bash # Utilisation du shell bash
#!/bin/sh # Utilisation du shell sh
```

EXÉCUTION DE SCRIPT

Pour exécuter un script shell sans passer par son chemin absolu :

- Pouvoir exécuter le fichier (*chmod +x*)
- L'enregistrer dans un répertoire contenu dans la variable **$PATH**, ou y faire un lien symbolique, ou ajouter l'URL dans **$PATH**.

DÉBUGUER SON SCRIPT

Il y a deux possibilités :
Dans le script, près de la zone à débugguer, placer :

```
set -x
    # Code à débugguer
    echo $PS4 # La sortie débug est placée dans cette variable
set +x # Annule set -x
```

On pensera à enlever l'instruction une fois le bug trouvé (question de propreté de script).

Soit à l'appel du script en CLI :

```
bash -x ./script.sh
```

COMMENTAIRES

Un commentaire commence par **#**.

```
# Ceci est commentaire shell
```

DÉCLARATION ET ASSIGNATION DE VARIABLES

```
# Le nom d'une variable ne peut pas commencer par un chiffre
local variable # Dans une fonction, restreint la portée de la variable au scope de la fonction, sinon attention aux effets de bord
variable=«valeur» # On protège la valeur avec « »
{!variable} # Utilise le contenu de la variable comme nom de variable et la retourne. Ici, une nouvelle variable $valeur est créée

echo ${#variable} # Affiche la longueur de la variable
echo $variable # Affiche la valeur de la variable

nv_var=variable # Contient le nom d'une autre variable

# Référence directe
echo «nv_var = $nv_var» # Affiche nv_var = variable
# Référence indirecte
eval nv_var=\$$nv_var
echo «nv_var = $nv_var» # Affiche nv_var = valeur

pwd=`pwd` # Quotes obliques pr exécuter une commande
ls=$($(ls)) # 2e façon d'exécuter une commande, éviter le ls dans un script bash
# 3e façon, chq niveau de cmd est protégé par backslashes
crea_dirnfile=`cd ~\`mkdir DOSSIER\\\`touch file\\\`\``

action=«L'action demandée est $pwd» # Double quotes pour lecture de variable

env # Affiche toutes les variables d'environnement

readonly var # Verrouille la variable, ne peut pas être modifiée ou supprimée

unset var # Supprime la variable sauf si elle est protégée. Rend la variable à l'état de non défini (à ne pas confondre avec variable vide / nulle). L'espace mémoire alloué est libéré.

export var # Permet à un processus père faire connaître la variable aux processus fils découlant. La réciproque ne s'applique pas (des processus fils au processus père…). Sinon la variable est inconnue des autres environnements – zone mémoire vierge.
```

Variables numériques (seulement integers):

```
let « variable=3 »
echo $variable # Affiche 3

let « addition=3+4 »
echo $addition # Affiche le résultat : 7

# Relie une suite d'opérations par virgule, tout est évalué seul le dernier calcul est retourné
let « opé= (( a = 9, 15 / 3 )) » # Initialise a=9 et opé=15/3

let i++ # Incrémente (sinon ça bug)

seq 0 10 # Affiche les chiffres de 0 à 10
seq début pas fin # On peut indiquer un pas pr aller de … en …
{0..10} # Pareil que seq 0 10, fonctionne avec les lettres aussi {a..z}

echo «7 * 2 = $(( 7 * 2 )) # Affiche 7 * 2 = 14
```

Les opérations utilisables sont : `+` ; `-` ; `/` ; `%` *(modulo)* ; `**` *(puissance)* ; `+=` ; `-=` ; `*=` ; `/=` ; `%=`. Voir `bc` ou `awk` pour les nombres flottants.

Saisie interactive :

```
read -p « Message à afficher par prompteur » var1 var2
# -p -> Active le prompteur
# -a -> Les variables seront des arrays
# -s -> Active le mode silence, la frappe ne s'affiche pas, utile pour faire saisir un mdp
# -r -> N'autorise pas les backslashes pour échapper les car
# -n -> Suivi d'un chiffre. Limite la saisie à n caractères

read -p « Message prompteur » variable ; echo
# L'ajout de la commande echo permet d'ajouter un \n à la fin du message du prompteur, sinon la saisie se fait à la suite du message.
```

Avec `read`, on peut assigner plusieurs variables à la fois, ou une seule, ou aucune ! Si aucune variable n'est demandée, alors la saisie sera stockée dans `$REPLY`.
Le programme `read` attend de **stdin** (entrée standard) une chaîne terminée par **Entrée** ou **Fin de Ligne** (EOL). Une fois la saisie validée, chaque mot séparé par une espace sera stocké dans chaque variable passée à la commande. En cas d'excédent, celui-ci ira dans la dernière variable. En cas de manque, les variables non remplies sont définies, mais vides.

Ksh & *bash* permettent d'affecter dans un **tableau** chaque mot provenant de **stdin**. Le 1er mot ira à l'indice **0**, le 2nd à l'indice 1, etc. Cette syntaxe remplace tout le tableau, il est réinitialisé en écrasant les données précédentes.

TYPAGE DE VARIABLE

```
typeset [-airx] var1 [var2]...
# -a : tableau indexé
# -i : entier, peut être utilisé dans les opé arithmétiques
# -r : équivalent de readonly
# -x : équivalent de export
# -f : fonction
```

declare est un synonyme de **typeset** mais spécifique à Bash.

TABLEAUX

Non POSIX

Ici, l'indice continu (0, 1, 2…) est facultatif, il peut manquer l'indice 4 entre l'indice 3 et l'indice 5. On lira le tableau dans une boucle for.

```
tableau=(«valeur0» «valeur1» «valeur2») # Déclaration
tableau=([ind0]=val0 [ind1]=val1 [ind2]=val2) # Autre façon de déclarer un tableau
tableau[5]=«valeur5» # On peut directement affecter un indice
tableau3=(«${tableau1[@]}» «$tableau2[@]}») # Concatène 2 tableaux en 1

${tableau[0]} # Appelle l'indice 0 du tableau renvoie la valeur
${!tableau[0]} # Affiche l'indice 0
${!tableau[@]} # Affiche tous les indices, très utile dans une boucle for avec indices discontinus

${tableau[*]} # Concatène tous les elt d'un tableau en une chaîne unique et la renvoie
${tableau[@]} # Transforme individuellement chq elt présent dans le tableau en une chaîne et renvoie la concaténation de toutes les chaînes

${#tableau} # Renvoie le nb d'elts de tableau
${#tableau[*]}# Renvoie le nb d'elts de tableau
${#tableau[@]}# Renvoie le nb d'elts de tableau

${tableau:x:y} # Renvoie les y elts à partir de x
```

CONDITIONS

If

```
if [[ condition1 ]] && [[ condition2 ]]
# Si condition1 VRAI, exécuter condition2.
then

    # Différentes actions à effectuer si c1 et c2 Vrai

elif [[ condition3 ]] || [[ condition4 ]]
# Si condition3 FAUX, exécuter condition4
then

    # Actions à mener si c1 ou c2 FAUX et c3 ou c4 VRAI

else

    # Actions si c1 ou c2 FAUX et c3 ET c4 FAUX

fi # fin de condition if
```

Case : tester plusieurs conditions à la fois *(ne gère pas les regex !, seulement la classe [0-9])*

```
case $1 in
    «valeur1»)
        # Actions si $1 = valeur1
        ;;
    «valeur2» | «autre»)
        # Actions si $1 = valeur2 ou autre
        ;;
    *)

        # Actions par défaut si les conditions ci-dessus ne sont
pas vérifiées
        ;;
esac
```

Select : proposer à l'utilisateur un menu numéroté commençant à 1. À chaque n° est associée une chaîne prise séquentiellement dans la liste. Pour modifier le message de prompt, modifier la variable **$PS3**. Le n° du menu choisi sera stocké **$REPLY** et sa valeur dans **$option**. Il faudra sans doute (menu complexe) évaluer l'une des deux variables avec une structure if ou case. On peut imbriquer un menu dans un menu. On sort d'un menu avec l'instruction **break**. S'il y a 2 menus imbriqués l'un dans l'autre, break fera sortir du 2e menu et break 2 remontera au menu parent (bien placer ses break pour cela…)

```
PS3=«Enter a number»

select option in opt1 opt2 opt3 # menu parent
do
    case $option in
```

```
        «opt1»)
            select sub_option in s_o1 s_o2
            do
                if [[ ! $s_o1 ]]; then
                    # Si ce n'est pas s_o1
                    break 2 # retour menu parent
                else
                    break # retour menu sub_option
                fi
            done
            ;;
        «opt2» | «opt3»)
            # Diverses actions
            break 2 # Sort du menu tout court
            ;;
    esac

    echo «Option Choisie : $option»
    echo «N° Option Choisie : $REPLY»

    # Penser à faire une condition pour sortir du menu, car là
comme ça, il va boucler sans fin, instruction break à ce niveau
du menu suffisante pour sortir

done
```

BOUCLES

While (*tant que VRAI, jusqu'à ce que FAUX, = 0*)

```
while [[ test ]]
do
    # Actions à mener
done

# Boucles sur sortie de commandes
cmd | while read -r ; do ... done # 1
while read -r ; do ... done< <(cmd) # 2
while read -r, do ... done <<< «$(cmd)» # 3
while read -r ; do ... done < /path/file # Remplace cat en
script
```

Comme **read** renvoie **VRAI** si **stdin** lue et **FAUX** si entrée vide, on programme une boucle de lecture pour traiter un flot d'information. D'ailleurs pour ne pas utiliser **cat** dans un script *(proscrit, trop gourmand en ressources, comme ls)*, on passe le fichier à lire en sortie de boucle, tant qu'il y aura une ligne dans le fichier, elle serait transmise à **stdin**, qui la passera à **read**, qui renverra **VRAI** à

while et on bouclera tant que le fichier n'est pas fini. On utilise **read -r** pour ne pas modifier ce qui est passé en lecture.

Until (*jusqu'à ce que VRAI*, *tant que FAUX*, $\neq 0$)

```
until [[ test ]]
do
    # Actions à mener
done
```

For

Syntaxe *in valeur1* est optionnelle. Si omise, les valeurs sont prises dans **$***. Ne pas exécuter de commandes (**seq**, **ls**, **cat**…) après **in**, boucler avec **while** sur une sortie de commande. *Ne pas initialiser de variable dans une boucle for ! (var=0)*

```
for variable in «valeur 1» «valeur 2» «valeur 3» # Syntaxe 1
for (( var=0 ; var <= 10 ; var++ )) # Syntaxe 2
do
    # Action à mener avec $variable
done
```

TESTS

Si le test réussit, la valeur de retour est **VRAI** ou **0**. On récupère la valeur de retour avec **$?**. On protège au miximum ses opérandes avec la syntaxe **[[ ]]** et on pense bien à laisser des espaces autour des crochets à l'intérieur du test.

3 types de test possibles :

String :

```
test $chaine1 == $chaine2 # Test si chaîne1 = chaîne2

# Si une variable n'est pas définie, elle est considérée comme vide. Très utile pour vérifier la présence de paramètres (script ou fonction)
if [ -z $1 ] ; then # Vérifie si 1er paramètre est une chaîne vide. Attention aux espaces après les crochets !

[[ $string1 =~ $string2 ]] # Test sur regex (pas obligé que string, num ok)
! [[ $string1 =~ $string2 ]] # Seule façon de tester une négation sur une regex
```

Integer :

```
if [[ $num1 -eq $num2 ]] ; then # Vérifie si num1 = num2
```

Files :

```
[[ -s $file ]] # Teste si file existe et n'est pas vide
[[ ! -e $file ]] # ! permet d'obtenir la négation du test utilisé, ici = « si $file n'existe pas »
[[ $file1 -nt $file2 ]] # Teste si file1 est plus récent que file2
```

cf fiche Bash Test pour plus de précisions sur les autres tests possibles (négations, regex…).

FONCTIONS

En Shell, une fonction se déclare toujours avant de l'appeler ! Il ne faut pas confondre, en Shell, retour de fonction et valeur retournée par la fonction comme en C. La valeur retournée est simulée.

```
nom_fonction () # Syntaxe 1
{
    # Actions
    return [0-255] # Code de retour de la fonction, de 0 à 255
    exit 0 # Sort du script (fonctionne en niveau)
    exit 1 # Sort de la fonction mais pas du script
}

function nom_fonction () # Les parenthèses sont facultatives
{
    local variable
    variable=$1
    # Actions
    $# # Contient le nb de paramètres de la fonction
    $* # Contient la valeur des paramètres en une seule chaîne
    $@ # Contient la valeur des paramètres considéré chacun comme une chaîne
    $- # Donne la liste de options (flags)
    $0 # Contient le nom du script exécuté
    $FUNCNAME # Contient le nom de la fonction quand elle est exécutée
}

nom_fonction variable_paramètre1 # Appel de la fonction
# $1 => $variable_paramètre1 => nom de la variable
# ${!1} => ${!variable_paramètre1} => valeur de la variable
```

$1 contient le 1er paramètre passé à la fonction/script, on peut aller jusqu'à 9 paramètres comme cela (**$1** à **$9**), voire plus. C'est différent des **options** (*flags*) que l'on gère avec **set** et

getopt/**getopts**. On les appelle les **paramètres positionnels** et on les configure avec **set**.
Une variable ne peut pas être référencée par un chiffre. On ne peut pas faire **2=«-»** ni **$2=«-»** ni **${!2}=«-»**. Par conséquent, on ne peut pas modifier la valeur des paramètres positionnels comme cela *(logiquement, on n'est pas censé le faire : effet de bord…)*. Je n'ai trouvé que 2 façons de le faire :

Avec **set** :

```
set [-option(s)] [valeur$1 [valeur$2] [...] ] ]

set # Donne toutes les options activées (environnement, script en cours…)

set arbre cerise fleur
echo «$1, $2, $3» # Affiche arbre, cerise, fleur
```

Ce n'est pas très pratique surtout si on souhaite ne travailler que sur **$2**, il faudra également affecter une valeur à **$1**, à moins de combiner avec **shift**. Attention, **set** réinitialise tous les arguments existants.

Avec **read** et une redirection de flux :

```
read $2 <<< «nouvelle valeur de $2»
```

shift

Cette commande permet de décaler la liste des arguments d'une ou plusieurs positions vers la gauche. À l'appel, sans arguments, **$2** devient **$1** (sauvegarder la valeur de **$1** avant de lancer **shift** !), **$3** devient **$2**, etc.

```
valeur_positionnelle_1=$1

shift

echo $1 # A pris la valeur de $2

shift 2 # n doit être inf ou égal à $#

echo $1 # A pris la valeur de $4 si existant

set `date`

echo $1 # Retourne le jour
echo $2 # Retourne le mois
echo $3 # Retourne l'année
echo $4 # Retourne l'heure
```

```
echo $* # Retourne Thu Aug 20 21:42:25 CEST 2020
echo $# # Affiche 6 (paramètres positionnels)
```

caller

```
caller [n]
```

Instruction à placer dans le corps d'une fonction. Donne des infos sur l'appelant de la fonction (nom du script, fonction appelante, n° lg d'appel), **n** représente l'incrément, pour remonter de niveau (cf **break**).

VARIABLES PRÉDÉFINIES

```
$HOME # Répertoire personnel utilisateur
$PATH # Chemin de recherche des commandes
$PS1 # Prompt Shell principal
$PWD # Chemin courant

$IFS # Séparateur de champ interne. Très importante pour
l'analyse syntaxique du Shell. Sauvegarder son contenu avant
toute modification
    chaine= «toto:titi/tata»; old=$IFS; IFS=:/
    set $chaine # Déconcaténation de la chaîne suivant IFS
    IFS=old # Récup de l'ancienne valeur d'IFS
    echo $3 # Affiche tata

$TERM # Type de terminal utilisé
$REPLY # Saisie utilisateur
$RANDOM # Nb aléatoire entre 0 et 32 767
$$ # n° processus courant
$? # Statut de la dernière commande
$! # n° dernier processus lancé en bg
```

DOCUMENTER SON SCRIPT

Il possible d'ajouter des options à un script et de demander des valeurs pour ces options. Par exemple :

```
ls -l *sh
```

-l est une **option**, ***sh** un **paramètre positionnel**.

```
cut -c 2-5 filename
```

-c est l'**option**, **2-5** sa **valeur** et **filename** le **paramètre positionnel**. On utilise **getopt** ou **getopts** pour gérer les options. **getopts** ne gère pas les options longues et **getopt** ne fonctionne pas avec des options longues sur MAC *(paquet gnu-getopt pour gérer les options longues)*.

```
function args()
{
    get_opt=`getopt -o abc: -l lettrea, lettreb, lettrec:` --
«$@»

    eval set -- «$get_opt»

    while true
    do
        case «$1» in
            -a | --lettrea)
                # Actions
                ;;
            -b | --lettreb)
                # Actions
                ;;
            -c | --lettrec)
                # Actions si besoin sur «lettrec»
                shift # L'argument de l'option lettrec est $1
maintenant
                # Actions sur l'argument de lettrec
                ;;
            --)
                shift
                break
                ;;
        esac
        shift # $2 devient $1
    done
}

args $0 « $@ »
```

Voici un exemple dans lequel on gère les options avec **optget** par une fonction qu'on appelle juste après dans le script pour lire les options passées par l'utilisateur, *on pourrait se passe de la fonction et de son appel*. **-o** sert à déclarer des options courtes, **-l** sert à déclarer des options longues. **:** *(colon)* indique que l'option le précédant nécessite une valeur obligatoire avec **::** (spécial GNU) l'argument est optionnel. **--** arrête l'analyse des options, **$@** est un paramètre n'étant pas une option, il permet d'indiquer à **getopt** de considérer chaque argument comment une chaîne distincte. Si on avait mis « **$*** », il aurait considéré qu'il n'y avait qu'un seul argument **$1**, alors qu'il peut y en avoir jusqu'à 3, **$1**, **$2** et **$3**. Il vaut mieux utiliser **getopts** dans un script, car c'est un built-in de bash (pas de **man**, pour consulter l'aide de l'option : **help getopts**). **optget** est externe à bash. Il souvent nécessaire d'inclure un **eval** pour que les espaces et les guillemets soient traités correctement.

getopts s'utilise un peu différemment :

```
getopts options var_stock_opt [arg]

while getopts «ab:» OPTNAME; do {
    echo « getopts a trouvé l'option «$OPTNAME»
    case « $OPTNAME » in
        a)
            # Actions sur l'option a
            echo «Indice prochaine option à traiter : $OPTIND»
            ;;
        b)
            # Actions sur l'option b
            echo «Liste des arguments à traiter : $OPTARG»
            ;;
    esac
}
done
shift $(( OPTIND - 1 )) # Gère les arguments supplémentaires
n'étant pas des options
exit 0
```

: derrière une option, comme pour **getopt**, indique que le flag prend une valeur. La variable **$OPTNAME** permet de récupérer l'option à traiter, **$OPTIND** donne l'indice de la prochaine option (donc **$(($OPTIND - 1))** donne l'indice de l'option courante) et **$OPTARG** donne la valeur de l'argument associé à l'option.
Il y a deux façons de gérer les erreurs et options invalides avec **getopts** :

1. **getopts** détecte une option invalide, la variable **$OPTNAME** est initialisée avec le caractère **?** et un message d'erreur est

affiché à l'écran et **$OPTARG** est « unset ». Si c'est un argument d'option qui est manquant, alors **$OPTNAME=?**, **$OPTARG** est « unset » et un message de diagnostique d'erreur est affiché. C'est la façon « *traditionnelle* ». Il faut alors traiter **?** dans le **case** de la boucle **while**, et il faut l'échapper pour que le shell ne l'interprète pas littéralement (**\?**).

2. Si les options passées à **getopts** commencent par **:**, comme dans le cas **«:ab:»**, alors le mode *silence* est activé et aucun message d'erreur ne s'affiche. Si une option invalide est trouvée, elle est placée dans **$OPTARG** et **$OPTNAME** prend la valeur **?**. Si c'est un argument d'option qui manque, **$OPTNAME** prend la valeur **:** et place l'option correspondante dans **$OPTARG**.

SOURCE

```
source path/fichier
```

Importe un fichier où **source** est exécutée, le fichier importé remplaçant la ligne de la commande équivalent de *#include* (C) ou *include()*/*require()* (PHP).

LES COULEURS

```
echo -e '\033[effet;couleur_texte;couleur_bgm texte \033[0m'
```

	Effet	Texte	Arrière-plan
Normal	0		
Gras	1		
Non-gras	21		
Sombre	2		
Non-sombre	22		
Italique	3		
Non-italique	23		
Souligné	4		
Non-souligné	24		
Clignotant	5		
Non-clignotant	25		
Inversé	7		
Non-inversé	27		
Invisible	8		
Non-invisible	28		
~~Barré~~	9		
Non-barré			
Noir		30	40
Rouge		31	41
Vert		32	42
Jaune		33	43
Bleu		34	44
Magenta		35	45
Cyan		36	46
Blanc		37	47

La syntaxe pour donner une couleur est en vert. Ne pas oublier de fermer avec un m les paramètres et de clôturer avec un retour à la normale (on remet en normal avec \033[0m).