Compléments de cours

Système

Cours n°5

Isis Truck, UP8

Plan

- Généralités
- Variables
- Structures de contrôle (conditionnelles, boucles…)
- Tests
- Arithmétique
- Quelques commandes usuelles : echo, find, awk, cut, shift…
- Lancer et débugger un script

Généralités

- Types de shell:
  - sh : le premier (Bourne shell)
  - bash (Bourne again shell) : le shell standard GNU: puissant, souple, contenant beaucoup d'ajouts et d'extensions => notre choix
  - ksh, csh, … : syntaxe ressemblant au C ou bien convient pour des utilisateurs du monde Unix
- Pour changer de shell, on tape son nom.
  - Exemple: `ksh`
  - Dans ce cas, `ksh` est sous-shell du shell courant

Généralités

- Une précision sur les redirections des flux :
  - `commande 1>fic` : redirige la sortie standard sur le fichier `fic`
  - `commande 2>fic` : redirige la sortie erreur standard sur le fichier `fic`
  - **`commande >fic 2>&1`** : redirige la sortie standard **et** la sortie erreur standard sur le fichier `fic`

Le fichier `/etc/passwd`

- `nom_utilisateur:mot_de_passe:uuid:guid:commentaire:home:shell`
  - `nom_utilisateur` correspond au login de l'utilisateur.
  - `mot_de_passe` correspond au mot de passe de l'utilisateur remplacé par un x pour des raisons de sécurité.
  - `uuid` correspond à l'identifiant système de l'utilisateur.
  - `guid` correspond au groupe principal de l'utilisateur.
  - `commentaire` : commentaire au sujet de l'utilisateur : souvent son nom réel (Prénom et Nom).
  - `home` correspond au répertoire home de l'utilisateur sur ce système.
  - `shell` correspond à l'interpréteur shell par défaut de l'utilisateur.

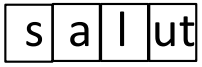
Les variables (1/4)

- **$0** : nom de la commande (du script qui s'exécute)
- **$1, $2, $3,** … **${10}** etc. : 1er argument de la commande, 2e argument, 3e argument, 10e argument, etc.
- **$#** : nombre d'arguments passés à la commande
- **$*** ou **$@** *ensemble des paramètres positionnels, équivalant à* **$1 $2 ... ${*n*}**
- **"$*"** *ensemble des paramètres positionnels, équivalant à* **"$1 $2 ... ${*n*}"**
- **"$@"** *ensemble des paramètres positionnels, équivalant à* **"$1" "$2" ... "${*n*}"**

Les variables (2/4)

- **var=val** ou **var="a b"** *affectation de la variable "var"* ***– Attention, pas d'espace autour du '=' pour les affectations***
- **$var** ou **${var}** *contenu de la variable "var"*
- **${#var}** *longueur de la variable "var"*
- **export var** *exportation de la variable "var" vers les shells fils*
- **set** *affichage de l'ensemble des variables définies dans le shell*
- **unset var** *suppression de la variable "var"*

TABLEAUX :

- **tab[0]=val** *affectation du premier enregistrement du tableau "tab"*
- **tab=(s a l ut)** *affectation du tableau "tab" :* 

- **${tab[0]}** ou **$tab** *contenu du premier enregistrement de "tab"*
- **${tab[11]}** *contenu du douzième enregistrement du tableau "tab"*
- **${tab[*]}** ou **${tab[@]}** *ensemble des enregistrements du tableau "tab"*
- **${#tab[11]}** *longueur du douzième enregistrement du tableau "tab"*
- **${#tab[*]}** *nombre d'enregistrements du tableau "tab"*

Les variables (3/4)

- **$$** *PID du shell courant*
- **$!** *PID du dernier travail lancé en arrière plan*
- **$?** *Valeur de retour de la dernière commande exécutée*

- **Quelques variables d'environnement**
  - **$HOME** *chemin du répertoire personnel de l'utilisateur*
  - **$OLDPWD** *chemin du répertoire précédent*
  - **$PATH** *liste des chemins de recherche des commandes exécutables*
  - **$PPID** *PID du processus père du shell*
  - **$PS1** *invite principale du shell*
  - **$PWD** *chemin du répertoire courant*
  - **$RANDOM** *nombre entier aléatoire compris entre 0 et 32767*
  - **$SECONDS** *nombre de secondes écoulées depuis le lancement du shell*

Les variables (4/4)

- Cas d'une variable bash ayant son nom (ou une partie de son nom) stocké dans une autre variable :

```
var1=hello; var2=salut; var3=coucou; i=2
mavar=var$i
echo ${!mavar}
```

affichera `salut`

  - Pour accéder à ce type de variable, il faut utiliser la syntaxe :
    `${!<nom_variable>}` où `<nom_variable>` est une variable comportant le nom de la variable à laquelle on souhaite accéder.

Substitution de chaîne

- Un mécanisme de *search and replace* est natif en bash, selon 4 syntaxes :

```
${PARAMETER/PATTERN/STRING}
${PARAMETER//PATTERN/STRING}
${PARAMETER/PATTERN}
${PARAMETER//PATTERN}
```

  - La 1ere syntaxe (avec 1 seul slash) substitue seulement la 1ere occurrence du pattern, la seconde (deux slashes) substitue toutes les occurrences du pattern

  - Exemple :

```
array=(This is a text)
echo "${array[@]/t/d}"
    ⇒ This is a dext
echo "${array[@]//t/d}"
    ⇒ This is a dexd
```

Manipulation de chaînes

- Ajouter un espace entre chaque lettre dans une string :

```
$ echo '123hello!' | sed 's/./& /g'
1 2 3 h e l l o !
```

C'est une regex !

- OU BIEN : utiliser le ':' en utilisant **l'expansion** de string :

```
string="123hello!"
for ((i=0; i<${#string}; i++)); do
     string_new+="${string:$i:1} "  # on extrait chaque caractère et on ajoute un espace
done
```

- Pourquoi ça marche ?
  - parce que : `${string:position}` extrait la sous-chaîne de `string` à la position `position`.
  - Si `string` vaut "*" ou "@", cela extrait les paramètres positionnels, en commençant à la position `position`.
  - `${string:position:length}` extrait `$length` caractères depuis la chaîne `$string` à partir de la position `$position`.

- Plus de choses encore à cette adresse : https://tldp.org/LDP/abs/html/string-manipulation.html

Séquence avec `seq`

La commande `seq` permet de générer une séquence (sur la sortie standard) en fonction des arguments passés :

```
seq [OPTION]... DERNIER
seq [OPTION]... PREMIER DERNIER
seq [OPTION]... PREMIER INCREMENT DERNIER
```

Afficher les nombres du PREMIER jusqu'au DERNIER, selon l'INCREMENT.

`-f, --format`=`FORMAT` utiliser le style de `FORMAT` en virgule flottante de printf

`-s, --separator`=`CHAINE` utiliser la `CHAINE` pour séparer les nombres (par défaut : \n)

`-w, --equal-width` équilibrer les largeurs en insérant des zéros en tête

Si PREMIER ou INCRÉMENT est omis, il prend par défaut la valeur 1.

`seq` : exemples

```
 #!/bin/bash
for i in $(seq 1 10); do
   echo $i
done
```

affichera :

1

2

3

4

…

10

`seq` : exemples

```
seq -w 01 10
```

affichera :

01

02

03

04

…

10

`seq` : exemples

```
seq -s - 8
```

affichera :

1-2-3-4-5-6-7-8

```
seq -f "%g/08/2024" 11
```

affichera :

01/08/2024

02/08/2024

03/08/2024

…

11/08/2024

Lecture (d'un fichier)

```
read a
```

=> lit l'entrée standard et stocke la chaîne lue dans la variable a

```
IFS=';'
while read a b c ; do
    …
done < fichier.txt
```

=> lit ligne à ligne fichier.txt et stocke les 2 premiers champs (séparateur ';') dans a et b, puis le reste de la ligne (sans le séparateur) dans c

Les structures conditionnelles

- **SI ALORS , SI ALORS SINON, SI ALORS SINON SI**

```
if condition
then
    commandes si condition vraie
fi
```

```
if condition
then
    commandes si condition vraie
else
    commandes si condition fausse
fi
```

```
if condition1
then
    commandes si condition1 vraie
elif condition2
then
    commandes si condition1
     fausse et condition2 vraie
else
    commandes si condition1 et
     condition2 fausses
fi
```

Les boucles

- **while do done**

```
while condition
do
   commandes tant que condition est vraie
done
```

```
while read ligne
do
   traitement de $ligne qui contient successivement chaque ligne du fichier "fic"
done < fic
```

- **until do done**

```
until condition
do
   commandes tant que condition est fausse
done
```

- **for in do done**

```
for var in liste de valeurs
do
   traitement de $var qui prend successivement pour valeur chaque argument de la liste
done
```

```
for var in *
do
   traitement de $var qui prend successivement pour valeur chaque nom de fichier présent dans le répertoire courant
done
```

```
for var      # équivalent à  for var in $*
do
   traitement de $var qui prend les valeurs des paramètres positionnels
done
```

Structures de contrôle (suite)

- **case in esac**

  case $var **in**

  a) traitement si $var contient "a" **;;**

  b|c) traitement si $var contient "b" ou "c"**;;**

  d*) traitement si $var commence par "d"**;;**

  ***)** traitement par défaut**;;**

  esac
- **continue**
  - La commande **continue** permet de passer à l'itération suivante dans une structure "`do ... done`"
- **break**
  - La commande **break** permet de sortir d'une structure "do...done"
  - Il est possible de sortir de plusieurs boucles imbriquées en spécifiant leur nombre en argument de la commande **break**
- **exit** permet de sortir d'un script de façon explicite. On pourra ajouter un code de sortie (exemple : exit 1)

Les tests (1/6)

- `test` ou bien `[ ]` (avec les espaces) : calcule un test et renvoie une valeur binaire. Quelques exemples :

```
if [ $# -eq 0 ]   # Si aucun paramètre fourni
then
    echo "Aucun argument reçu !"
fi
```

```
if test -f "$1"    # ou bien: if [ -f "$1" ]
then
   echo Le fichier \"$1\" est bien présent.
   echo "Voulez vous lire le fichier $1 (o pour oui) ?"
   read reponse
   if [ $reponse = "o" ]
   then
        less "$1"
   else
        echo  "J'ai noté que vous ne vouliez pas lire le fichier"
   fi
else
    echo "Fichier \"$1\" absent"
fi
```

Les tests (2/6)

- **`if` sans crochets ne fonctionne pas** car il faut une **commande qui demande l'évaluation !**
- `=` permet un test d'égalité mais fait une comparaison lexicale (chaîne de caractères), alors que `-eq` fait une comparaison numérique
  - En fait, tout est considéré comme chaînes. `-eq` dit à bash d'interpréter les chaînes comme des entiers (ce qui a pour effet de renvoyer un 0 sans warning si une chaîne n'est pas numérique).

- `-a` : norme POSIX pour *logical and* (devenu obsolète) => on préférera `&&`
- `-o` : idem pour *logical or* (devenu obsolète) => on préférera `||`

- *Pour d'autres tests de ce type, voir http://wiki.bash-hackers.org/commands/classictest*

Les tests (3/6)

- Quelques unes des principales conditions de test :
  - Sur fichiers

    -f : fichier normal.

    -s : fichier non vide.

    -d : répertoire.

    -r : fichier accessible en lecture.

    -w : fichier accessible en écriture.
  - Sur variables

    -z : variable vide.

    -n : variable non vide.
- Comparaison d'une variable à un nombre

  -eq : égal (*equal*)

  -ne : différent (*not equal*)

  -lt : < (*lesser than*)

  -gt : > (*greater than*)

  -le : <= (*lesser or equal*)

  -ge : >= (*greater or equal*)

Les tests (4/6)

```
test -w $1
echo "Le fichier $1 est modifiable si 0 est affiché : $?"
```

```
[ -d "../shell" ]
if [ $? -eq 0 ]
then
    echo "le repertoire shell existe"
fi
```

```
L=25
test "$L" -eq 25 && echo "L=25"
test "$L" -ge 15 && echo "L>15"
```

```
if [ -z "$mathieu" ]
then
    echo "la variable mathieu est vide »
fi
```

*NB : Il faut **toujours** placer les variables dont on n'a pas le contrôle absolu **entre guillemets doubles**. En effet, si la variable est vide, non définie ou si elle contient un espace, le test ne fonctionnera pas.*

Les tests (5/6)

- Commande de test **étendue**
  - Depuis la version 2.02, Bash a introduit les doubles crochets `[[ ... ]]` (commande étendue de test)
  - `[[ ... ]]` réalise les comparaisons de façon similaire aux autres langages usuels. Par ex., `==` est permis et est un alias de `=`
  - Utiliser `[[ ... ]]` à la place de `[ ... ]` peut donc aider à éviter les erreurs de logique dans les scripts
  - *Exple* : les opérateurs `&&`, `||`, `<` et `>` fonctionnent avec `[[ ]]` alors qu'ils génèrent une erreur avec `[ ]`

Les tests (6/6)

- Commande de test **étendue** (suite)
  - On peut également tester une variable en passant par les **expressions régulières** : il faut utiliser l'opérateur **=~**
  - Exemple : On cherche tous les clients dont le nom commence par « Du »
    - Le paramètre positionnel n°1 (ou 1er argument de la commande) contient la chaîne « Du » (donc `$1` contient « `Du` »)
    - Dans le script, la variable `client` contient successivement tous les noms des différents clients
    - `if [[ $client =~ ^$1 ]]` permettra d'obtenir "Durand", "Dupond", "Dugenou", etc.

Inversion d'un test

- Opérateur ***logical not*** : **!**
- Syntaxes possibles (ici, test de l'existence du répertoire `$rep`) :

```
if test ! -d $rep ; then
    echo "$rep n'existe pas"
fi
```

  - ou bien

```
if [ ! -d $rep ] ; then ...
```

  - ou bien

```
if ! [ -d $rep ] ; then ...
```

Arithmétique

- `(( … ))` ou bien `let …`
  - permet de demander l'évaluation d'un calcul (expansion et évaluation)
  - Exemple :
    - `a=$(( 4+2 ))` ou bien `let "a=4+2"`
- NB: `(( … ))` sera préféré car plus portable (dans différents shells)

Substitution de commande

- Il y a 2 syntaxes pour opérer la substitution d'une commande :
  - avec **$(**`commande`**)** ou bien
  - avec des *backticks* (backquotes)
- Exemple :
  - On suppose que `fic` contient un nom de fichier texte, par exemple : "rapportMoral.txt »
  - `nomFSE` veut dire : nom du **f**ichier **s**ans **e**xtension
  - On substitue la commande `basename` grâce à `$()` pour que `nomFSE` reçoive le résultat (sur la sortie standard) de la commande
  - `nomFSE=$(basename $fic .txt)`
  - `nomFSE` contiendra donc "`rapportMoral`", ds l'exemple

Substitution de commande (fin)

- On peut aussi écrire : `nomFSE=`basename $fic .txt``

- Le style `$()` fonctionne ds tous les shells conformes à POSIX.
- Le style *old-fashion* `` ` `` peut être utile pour les autres shells.

- Mise à part le point de vue technique, le style `` ` `` a également un inconvénient visuel :
  - Pas facilement repérable dans le code
  - On peut le confondre avec les quotes
  - Pas si simple à trouver sur le clavier

Lectures spéciales

- `<<` est une structure ***here-document*** : On indique au programme quel sera le texte de fin, et quand le délimiteur est lu, le programme lit comme une entrée tout le flux passé, et s'exécute. Exemple :

```
$ wc << EOF
un deux trois
quatre cinq
EOF
 2 5 26
```

  C'est la même chose que lancer `wc`, taper des mots et appuyer sur `Ctrl + D`

- `<<<` est une structure ***here-string*** et appelée ***process substitution***. Au lieu de taper du texte, on donne une chaîne pré-fabriquée à un programme. Par exemple, on peut écrire

  `bc <<< 5*4` qui est l'équivalent de `echo '5*4' | bc`

  Ou encore : `wc <<< date` : ici, c'est la chaine "date" qui est considérée

  `wc <<< $(date)` : ici, c'est la **commande** `date` qui est considérée

  Le "triple inférieur" permet de substituer une commande et de la donner en entrée à une autre commande

Lecture d'une variable multiligne

- `<<<` peut être utilisé pour lire une variable multiligne. Exemple :

```
var=$(ls)   #var est une variable multiligne
while read ligne; do
  echo "LIGNE: '${ligne}'"
done <<< "$var"
```

affichera

```
LIGNE: 'Applications'
LIGNE: 'Desktop'
LIGNE: 'Documents'
LIGNE: 'Downloads'
LIGNE: 'Library'
LIGNE: 'Music'
LIGNE: 'Pictures'
LIGNE: 'Public'
```

- `<<<` est compris par bash et par sh. Sous bash (mais pas sous sh), il peut être remplacé par la syntaxe `<(…)` , par exemple `< <(echo $var)`

Commandes d'affichage

- `echo` : affiche les arguments passés sur la sortie standard
- `printf` : assez similaire à `echo`, mais permet d'utiliser les règles de formatage
- Exemple :

```
echo "Bonjour"
echo
echo  "Voici…"
```

```
printf "Bonjour\n\n"
printf "Voici…"
```

`find` : chercher des fichiers

- **Synopsis : `find chemin [expression]`**
  - `chemin` : endroit où chercher
  - `expression` : composée de `primaires` et d'`opérandes`
- Exemples de `primaires`:
  - `-amin` *n* : dernier accès au fichier il y a *n* minutes
  - `-atime` *n* : dernier accès au fichier il y a *n**24 heures
  - `-empty` : fichier vide
  - `-delete` : détruit les fichiers (ou répertoires) trouvés, correspondant au motif
- `-exec utility [argument ...] {}` : renvoie vrai si le programme `utility` renvoie 0 comme valeur de retour (NB: 0 signifie vrai, toute autre valeur signifie faux). Des arguments optionnels peuvent être passés à `utility`. L'expression doit se terminer par ``;''. Si l'on invoque `find` depuis un shell, il faut *quoter* le ';' (ou le *backslasher*) sinon le shell le traiterait comme un opérateur de contrôle. Si la chaîne ``{}'' apparaît, elle est remplacée par le chemin complet du fichier courant.

  Exemple : `find . -name '*rc.conf' -exec chmod o+r {} \;`
- Les options :
  - `-name 'toto'` : spécifie que l'on cherche les fichiers dont le nom contient "toto"
  - `-iname 'toto'` : spécifie que l'on cherche les fichiers dont le nom contient "toto", indépendamment de la casse (donc Toto, tOtO, TOTO, totO, etc.)
  - `-regex '^ba[bn]'` : spécifie que l'on cherche les fichiers dont le nom commence par "baba", "banane", etc.

`awk` : effectuer des actions dans un fichier

- `awk [-Fs] [-v variable] [-f fichier de commandes] 'program' fichier`
  - `-F` Spécifie les séparateurs de champs
  - `-v` Définit une variable utilisée à l'intérieur du programme
  - `-f` Les commandes sont lues à partir d'un fichier
  - `program` Peut contenir l'instruction print qui affiche du texte

  Les variables que se construit `awk` sont : $1, $2, $3, etc. qui correspondent au champ de l'enregistrement courant. Le changement de champ se fait grâce au séparateur défini par –F

  $0, quant à lui, représente l'enregistrement en entier.

  – Exemples :

  `awk –F: '{ $2 = "" ; print $0 }' /etc/passwd`
  - imprime chaque ligne du fichier `/etc/passwd` après avoir effacé le deuxième champ

  `awk '{print $0}' fichier`
  - affiche toutes les lignes de *fichier* (idem que `cat fichier`)

`cut` **et** `awk`

- `cut` permet de récupérer des caractères ou des champs d'une ligne
- L'option **-d** permet d'indiquer le caractère séparateur de champ. Le caractère séparateur par défaut est la tabulation.
- Exemples :
  - Tronquer les 2 premiers chiffres d'un code postal :

    ```
    echo 93200 | cut -c1-2
    ```

    affiche : 93
  - renvoie la liste des utilisateurs :

    ```
    cut -d: -f1 /etc/passwd
    ```

    idem que : `awk -F: '{print $1}' /etc/passwd`

`diff` : comparaison de fichiers

- `diff [options] source cible`
- `diff` compare ligne à ligne la source et la cible
- Lorsque les 2 fichiers sont identiques, `diff` n'affiche rien
- Quelques options:

  -b ignore les différences dues à des espaces blancs -B ignore les différences dues à des lignes blanches -i ignore les différences minuscules/MAJUSCULES -q indique seulement si les fichiers sont différents et n'affiche pas les différences elles-mêmes -s indique lorsque deux fichiers sont identiques
- Explications détaillées ici : https://fog.ccsf.edu/~gboyd/cs260a/online/adminbasics1/diff.html

diff : exemple

- **cat fic1**

```
1 bonjour
2 hello
4 guten Tag
```

- **cat fic2**

```
1 bonjour
2 hello
3 good morning
4 guten Tag
```

Le ‘a’ signifie ‘add’ (ajout d’une ligne)
Le ‘d’ signifie ‘delete’ (suppression d’une ligne)
Le ‘c’ signifie ‘change’ (changement d’une ligne)

Ainsi, ‘2a3’ signifie ‘la ligne n°3 dans le fichier cible doit être ajoutée après la ligne 2 dans le fichier source pour que les 2 fichiers soient semblables’

Ensuite ‘> 3 good morning’ indique que la chaîne ‘3 good morning’ doit être ajoutée dans le fichier source.

- **diff fic1 fic2** 2a3 > 3 good morning

- **diff fic1 fic2** 3d2 < 3 good morning

shift

- La **commande `shift`** déplace à gauche les valeurs actuelles des paramètres de position.
- Exemple:
  - si les valeurs des paramètres de position actuels sont :

```
$1 = -r   $2 = file1   $3 = file2
```

  - après avoir exécuté `shift`, les paramètres sont :

```
$1 = file1   $2 = file2
```

- Commande très utile lorsqu'on souhaite analyser les arguments de la commande.
- NB : on peut également paramétrer `shift` pour déplacer plus d'une valeur à la fois.
- Dans l'exemple ci-dessus, `shift 2` donnerait :

```
$1 = file2
```

Lancer un script

- Pour lancer un script, soit
  - On l'exécute dans un sous-shell (fils du shell courant) et on tape
    - `mon-script.sh` si PATH contient le répertoire '.'
    - `./mon-script.sh` sinon
    - (il faut que le *user* ait les droits en exécution sur le script)
  - On l'exécute directement dans le shell courant et on tape
    - `source mon-script.sh` ou bien
    - `. mon-script.sh`
    - (car `source` est équivalent à la commande '.')
    - il est inutile que le script ait les droits en exécution dans ce cas

Débugger un script

- L'option `-x` de la commande `bash` permet de lancer le mode debug :
- `bash -x mon-script.sh`
- (à tester en TP !)

- NB : il est utile de supprimer les variables locales avant de quitter le script : utilisation de `unset`
- Exemple :

```
dupond@machine$ cat mon-script.sh
…
for i in … ; do … ; done
unset i
dupond@machine$
```