GB4 BIMB **Cours n°4** 2^{nd} semestre 2025-2026
Introduction à l'Administration Système et Réseau Gilles Bernot

Exposés :
- 5 minutes par exposé
- note sur 3 critères : Intérêt de la question / Difficulté de réalisation / Pédagogie de l'exposé
- 50% note promo et 50% note prof

1 Shell scripts et programmation en shell

Pour gérer un système, on se rend vite compte que l'écriture de shell scripts permet de gagner beaucoup de temps. Un shell script est une simple fichier qui contient des commandes successives comme on les écrirait au clavier. On peut naturellement l'utiliser en le donnant en entrée standard de la commande `bash` mais le plus convivial est de procéder comme suit :
- créer un répertoire pour placer tous ses shell scripts (généralement `$HOME/bin`) et l'ajouter à la variable `$PATH`,
- faire commencer chaque fichier de shell script par la ligne « `#! /bin/bash -` » afin que le système sache qu'il doit interpréter ce fichier en utilisant le shell `bash`[1],
- rendre le fichier exécutable avec `chmod`.

Le shell script a alors automatiquement accès à des variables « numérotées » un peu spéciales :
- `$0` est l'adresse absolue du shell script lui-même
- `$1` est le premier argument donné au shell script par l'utilisateur qui l'a appelé, `$2` est le second argument, *etc.* Par exemple si l'utilisateur tape « `cmd -t toto tutu` » alors pour le shell script `cmd`, la variable `$1` vaut `"-t"`, `$2` vaut `"toto"` et `$3` vaut `"tutu"` ; les variables suivantes (`$4` *etc*) sont vides.
- `$#` est le nombre d'arguments du shell script (3 dans l'exemple précédent)
- `$*` est égal à `"$1 $2 $3 ..."`
- la commande `shift` dans le shell script décale vers la gauche les variables `$1`, `$2`, `$3`, *etc.* Par conséquent, la valeur de `$1` est perdue et `$#` diminue de 1 (pratique pour analyser les arguments les uns après les autres avec un `while` si on n'en connaît pas le nombre *a priori* : `while [ $# -gt 0 ] ; do` *blablabla avec* `$1 ; shift ; done`).

Le shell offre également les primitives de contrôle habituelles :

```
if [ ... ]
  then ...
  else ...
fi
#
# un "#" indique que la suite de la ligne est un commentaire
#
while [ ... ]
do
  ...
done
#
for variable in liste
do
   ...
done
#
case expression in
  pattern1) # expressions régulières au sens du shell, donc "simplifiée"
    ...
    ;;
  pattern2)
    ...
    ;;
  etc)
esac
```

1. Au passage, ça marche aussi avec python *via* « `#! /bin/python` » et on écrit ensuite un programme en python...

La syntaxe des conditions entre crochets (`if` et `while`) est définie dans `man test`. En réalité, le crochet ouvrant est une abréviation pour `test` et plus généralement ou peut donner en argument du `if` ou du `while` n'importe quelle commande. La condition est considérée comme « vraie » si la commande n'échoue pas, et comme « fausse » si la commande échoue. `test` est donc simplement programmé pour échouer si la condition qu'on lui donne en arguments est fausse.

Les redirections comme « | » (sortie standard de la première commande en entrée standard de la suivante), « < » (fichier en entrée standard d'une commande), « > » (sortie standard dans un fichier), *etc.* ou les backquotes `‘...‘` (qui transforment la sortie standard d'une commande en chaîne de caractères) fonctionnent aussi dans un shell script.

Un shell est un langage interprété donc les « shell scripts » sont des fichiers lisibles, aisément vérifiables et modifiables. On peut ensuite programmer exactement ce qu'on taperait pour faire la commande en mode interactif. De plus, on peut programmer des « fonctions » en shell. Dans une session en bash, les fonctions s'écrivent sous la forme :

```
nomfonction () {
  ...
  corps de la fonction
  ...
}
```

et les variables `$1`, `$2`, ... deviennent les arguments *de la fonction*. Ainsi dans une fonction à l'intérieur d'une commande écrite en shell, les $1, $2, etc. sont les arguments *de la fonction* et on n'a donc plus accès directement aux arguments de la commande. Par exemple, si « `prevision` » est le shell-script suivant :

```
#! /bin/bash
compose () {
  ## ici $1 et $2 sont les arguments d'appel de compose, pas de prevision :
  if [ "$1" = "$2" ]
    then echo "$1"
    else echo "$1-$2"
  fi
}
## ici par contre le "$#" est le nbre d'arguments de prevision :
case "$#" in
  0|1) echo "Donner plusieurs arguments !!"
       exit 1 ;;
  *) echo "Le prefixe sera \"$1\""
     prefixe="$1"
     shift
esac
## Par definition : $*="$1 $2 $3 $4 ..."
for suffixe in $*
do
  nom=‘compose $prefixe $suffixe‘
  if [ -f $nom ] #c'est un nom de fichier
    then echo $nom
  fi
done
```

alors la commande « `prevision old truc machin chouette` » imprimera à l'écran les noms de fichiers qui existent parmi `old-truc`, `old-machin` et `old-chouette`.

Pour accéder à l'intérieur d'une fonction aux arguments du shell-script, il faut créer une variable intermédiaire avant d'appeler la fonction. C'est ce que fait la ligne « `prefixe="$1"` » dans l'exemple précédent.

La syntaxe de définition de fonctions est trompeuse : dans l'exemple, bien qu'on écrive « `compose ()` », la fonction `compose` accepte deux arguments, mais le nombre d'arguments est complètement implicite.

Enfin, notons que la fonction est toujours locale à la commande (elle n'est pas « exportée » vers les processus fils) et ne peut donc être utilisée que dans le shell script où elle a été déclarée.
En particulier, l'option `-exec` de `find` (qu'on verra plus tard) ne peut donc pas faire appel à une fonction définie dans le shell script...

`man bash` fournit tous les détails (trop !) à propos du language `bash` : à utiliser par survol et en extrayant ponctuellement les informations utiles au problème précis du moment.

2 Les expressions régulières sous Unix/Linux

Comme on le verra plus loin, un bon nombre de commandes utiles pour développer des shell-scripts utilisent des *expressions régulières*. Il y a sous Unix deux sortes d'expressions régulières : l'une, dite « simplifiée » dans la suite, sert principalement à la complétion de noms de fichiers, et l'autre est en revanche très universelle. On ne présente ici que les aspects majeurs de ces expressions régulières, fortements utiles pour la suite.

Pour le shell, on utilise des expressions régulières simplifiées, où :
— « `*` » remplace n'importe quelle suite de caractères dans les noms de fichiers (ou dans les patterns des `case` comme on le verra plus loin).
— « `?` » remplace n'importe quel caractère.
— et enfin on peut limiter le remplacement à une liste de caractères en les mettant entre crochets.

```
$ ls
agaga    glop    tag    tatoo    tatu    toto    tutu
$ ls t*
tag    tatoo    tatu    toto    tutu
$ ls t?t?
tatu    toto    tutu
$ ls *g*
agaga    glop    tag
$ ls t[oai]*
tag    tatoo    tatu    toto
```

Les expressions régulières « standard » ne suivent pas ces règles simples du shell. Les commandes les plus communes qui utilisent des expressions régulières standard sont par exemple `ed`, `sed`, `expr`, *etc.*

Pour une définition complète des expressions régulières sous Unix, faire `man ed`. Nous donnons ici seulement les constructions les plus souvent utiles.

Pour ces expressions régulières :
— Un point remplace n'importe quelle lettre.
Par exemple le pattern « `t.t.` » filtre aussi bien `toto` que `tutu` ou `toti`, mais aussi `twtr`.
— Pour se limiter à certaines lettres, il faut les énumérer entre crochets.
Par exemple « `t[aiou]t[aiou]` » filtre les 16 possibilités de `tata` à `tutu` en passant par `toti`.
— Avec un tiret, on peut énumérer une séquence de lettres entre crochets.
Par exemple « [A-Z] » filtre toute lettre majuscule, ou encore « `[A-Z0-9]` » filtre tout caractère qui est une majuscule ou un chiffre.
Si l'on souhaite un véritable tiret dans l'énumération de caractères, il faut commencer ou finir par lui (comme dans « `[-xy]` »). De même, si l'on souhaite un crochet fermant il faut commencer par lui.
— On peut également indiquer les caractères qu'on ne veut pas. Dans ce cas on commence par un accent circonflexe pour indiquer une négation.
Par exemple « `[^0-9]` » signifie « tout caractère qui n'est pas un chiffre ».
— Pour les suites de caractères, une étoile indique une répétition d'un pattern un nombre quelconque de fois (même 0 fois).
Par exemple « `Gr*oum*f` » filtre `Groumf` ou encore `Grrrroummmf`, mais aussi `Gouf` ou `Grouf`. Si l'on veut au moins un `r` et un `m`, on écrit « `Grr*oumm*f` ».
— Le caractère backslash est un caractère d'échappement. Ainsi « `.*\.jpg` » filtre `toto.jpg` mais pas `totoxjpg`. Si l'on veut un vrai backslash, il faut le doubler (parfois tripler ou quadrupler car backslash est aussi un caractère d'échappement du shell).
— Il y a une exception. Pour beaucoup de commandes de substitution, la forme spéciale « `debut\(milieu\)fin` » permet de ne retenir que le milieu.
Par exemple le pattern « `toto\(.*\)\.jpg` », lorsqu'il est appliqué à `toto123.jpg` retourne la chaîne de caractères `123`.
La commande `expr` sert à calculer toutes sortes de choses. Faire `man expr`. Parmi ses possibilités, il y a la gestion de pattern matching ; dans ce cas, on l'utilise sous la forme :

`expr "`*chaine*`" ':' '`*pattern*`'`

Par exemple la commande « `expr "toto123.jpg" ':' 'toto\([0-9]*\)\.jpg'` » donne la chaîne `123`.

Si le pattern n'est pas reconnu, `expr` retourne soit une chaîne vide soit un `0` (selon la version installée sur le système) et un code de retour 1 (qui signale un échec de la commande appelée) au lieu de 0 (qui signale un succès).

La commande `sed` (comme « stream editor ») est également très puissante pour manipuler des chaînes de caractères, ligne par ligne. On l'utilise beaucoup pour transformer la sortie standard d'une commande afin, par exemple, de l'adapter à l'entrée standard d'une autre, ou encore afin d'en extraire les informations qui nous intéressent. Dans son usage le plus courant, on utilise les *substitutions* :

```
.... | sed -e 's/pattern1/remplacement1/g' -e 's/pattern2/remplacement2/g' |
....
```

On peut également appliquer des commandes génériques à toutes les lignes qui répondent à un motif donné, comme par exemple `d` pour supprimer (delete) la ligne :

```
sed -e '/pattern/ d' < fichier | ....
```

Les `patterns` sont les expressions régulières et dans les `remplacements`, on peut utiliser des « `\1` », « `\2` », *etc.* pour récupérer les portions mémorisées dans les « `\(...\)` » du pattern. De plus, n'importe quel autre caractère que « `/` » est utilisable pour séparer les patterns et remplacements (pratique si le pattern ou le remplacement en contient un). Enfin le « `g` » final signifie *global* c'est-à-dire que le remplacement sera effectué partout où il est possible. Un « `1` » à la place du « `g` » signifierait *seulement la première occurrence*... là encore, faire `man sed`.

3 Les chaînes de caractères et l'évaluation des expressions en shell

Le langage utilisé par les shells, en particulier celui de `sh` ou `bash`, est antique, conçu avant que la théorie des langages soit bien développée en informatique. Il en résulte un langage objectivement maladroit ; par exemple :

- Il n'y a pas de types de données en shell. Toutes les données sont des chaînes de caractères (qu'elles soient ou non écrites entre guillemets) et il n'y a même ni entiers ni booléens qui sont pourtant des types indispensables...
- Par conséquent, il faut faire appel à une commande, comme `expr` ou `bc` par exemple, pour effectuer un calcul, afin que ces commandes interprètent les arguments qu'on leur donne (qui sont donc des chaînes de caractères) comme des données de types plus élaborés.
  ```
  --> expr 1 + 2
  3
  ```
- Alors que dans presque tous les langages de programmation modernes un nom de variable réfère implicitement à la valeur de cette variable, ici, du fait que tout est chaîne de caractères dans un shell-script, il faut faire explicitement précéder d'un `$` le nom de la variable pour avoir sa valeur...
- Les instructions de contrôle comme `if` ou `while` n'ayant pas de booléens à disposition prennent en fait en « argument » une commande Unix et le `then` (ou l'entrée dans la boucle) est suivi si la commande réussit, et le `else` (ou la sortie du `while`) si la commande échoue...
- Ainsi, pour « simuler » des booléens, il y a une commande `test` (faire `man test`) qui évalue ses arguments comme s'ils constituaient une expression booléenne, et échoue si l'évaluation résulte sur « False ». Pour faciliter la lecture, « `test` » porte également le nom « `[` », et lorsqu'il est appelé sous ce nom il exige un dernier argument égal à « `]` ». Par exemple
  ```
  if [ "$var1" = "$var2" ]
    then ...
  fi
  ```
  équivaut à
  ```
  if test "$var1" = "$var2"
    then ...
  fi
  ```
  et on passe dans le `then` si les variables `var1` et `var2` contiennent la même chaîne de caractères.

En shell, on simule un type « liste » par une chaîne de caractères où les espaces séparent les éléments de la liste. Par exemple, on peut écrire :

```
maliste="un deux trois quatre cinq"
for elem in $maliste
do
  ...
done
```

On peut ainsi exploiter le fait que les expressions régulières simplifiées du shell fournissent comme résultat toutes les chaînes qui « matchent » séparées par des espaces. Par exemple :

```
for fichier in *.txt
do
  ...
done
```

La difficulté de cette convention est qu'il est difficile de gérer des éléments de liste qui contiennent des espaces ! On peut parfois s'en sortir avec des simples ou doubles quotes :

```
for name in Dupont Dupond "Le Gall" Smith "Mac Mahon" "Que d'eau !"
do
  ...
done
```

La structure des « tableaux » est présente en shell (par défaut, ils sont similaires à des dictionnaires en Python où les éléments ne pourraient être que des entiers positifs). Les tableaux se manipulent comme dans l'exemple suivant :

```
mois[1]="janvier"
mois[2]="février"
mois[3]="mars"
...
mois[12]="décembre"
i=5
echo ${mois[$i]}
```

et l'on obtient « `mai` » en sortie standard. Remarquez les accolades qui servent à délimiter l'appel à une valeur du tableau ; si l'on écrivait seulement « `$mois[$i]` », le shell chercherait d'abord la valeur de la variable « `$mois` » puis, seulement dans un deuxième temps, chercherait à interpréter le « `[$i]` » restant.
Pour obtenir la puissance des dictionnaires de Python, il faut préalablement déclarer le « dictionnaire » comme un *tableau associatif* :

```
declare -A Type
Type["jpg"]="image"
Type["png"]="image"
Type["mp4"]="vidéo"
Type["mkv"]="vidéo"
Type["mp3"]="audio"
...
```

De plus, pour tous les tableaux : `${!tab[@]}` fournit la liste des clefs du tableau `tab`, `${tab[@]}` sa liste de valeurs (dans un ordre aléatoire, sans supprimer les répétitions), et `${#tab[@]}` sa taille.

Lorsqu'on doit gérer des données qui peuvent contenir des espaces, il est plus facile de les mémoriser dans un tableau que de les mémoriser dans une liste.

Le shell présente une gestion assez « délicate » des chaînes de caractères :
— Bien que `toto` soit équivalent à `"toto"` puisque, de toutes façons, tout est chaîne de caractères en shell, les guillemets sont utiles pour passer en argument d'une commande une chaîne de caractères contenant un espace. En effet, « `cmd toto tutu` » signifie que l'on passe deux arguments « `toto` » et « `tutu` » à la commande « `cmd` », alors que « `cmd "toto tutu"` » signifie que l'on passe un seul argument « `toto tutu` ».

— On obtient le même résultat avec « `cmd 'toto tutu'` » mais la différence entre `"..."` (double quotes) et `'...'` (quotes) est que cette seconde forme n'évalue rien dans la chaîne qu'elle entoure, tandis que la première évalue les valeurs de variables avec le `$` :

```
--> echo '$HOME'
$HOME
--> echo "$HOME"
/home/bioinfo/bernot
```

— Lorsqu'on lance une commande, elle fournit *a priori* un résultat sur sa sortie standard (par exemple en écrivant dans la fenêtre de terminal si l'on n'a pas redirigé la sortie standard). Si l'on souhaite exploiter ce résultat dans un programme comme une chaîne de caractère, il faut entourer la commande et ses arguments avec un backquote `` `...` `` :

```
--> pwd
/home/bioinfo/bernot/bin
--> if [ `pwd` = "$HOME" ] ; then echo "maison" ; else echo "ailleurs" ; fi
ailleurs
```

On peut ainsi mettre dans une variable le résultat d'une commande :

```
--> repertoireEquipe=`dirname $HOME`
```

et si l'on veut bien gérer les commandes dont le résultat pourrait contenir un espace :

```
--> repertoireEquipe="`dirname $HOME`"
```

Au passage cela permet de préciser que `"..."` interprète non seulement les « `$` » mais aussi les « `` ` `` » alors que `'...'` ne le fait pas.

```
--> echo $repertoireEquipe
/home/bioinfo
--> repertoireEquipe='`dirname $HOME`'
--> echo $repertoireEquipe
`dirname $HOME`
```

Un point fort en revanche de cette approche « tout est chaîne de caractères » est qu'il est facile de *calculer* une commande puis de l'exécuter :

```
if [ -d "$adresse" ] ##si l'adresse est un répertoire
  then voir='ls $adresse'
elif [ -f "$adresse" ] ##un fichier plat
  then voir='less $adresse'
  else voir='echo $adresse est un fichier spécial !'
fi
resultat="`$voir`"
```

On peut également utiliser « `eval` » : cette commande prend en argument une chaîne de caractères et l'évalue d'une manière similaire à `` `...` `` mais fournit le résultat sur la sortie standard au lieu d'une chaîne de caractère. Par exemple si `$adresse` vaut `/dev/null` dans ce qui précède alors

```
--> eval $voir
/dev/null est un fichier spécial !
```

Et pour finir :

— La commande « `source` *`nom/de/fichier`* » dans un shell script permet d'exécuter les lignes du fichier en question comme si elles étaient présente dans le script appelant.

— Si l'on ajoute un `&` après une commande et ses arguments alors *pendant* que cette commande s'exécute, la commande qui suit commence déjà sans attendre la fin de la première commande : la première commande tourne alors en « background ». La commande `jobs` de bash donne la liste des processus fils qui tournent en background.

4 Présentation des projets

Voir feuilles distribuées.