Comment dompter le shell (bash)?

Shell, c'est quoi exactement ?

- Il s'agit d'une interface texte entre l'utilisateur et le système
  - Tout se fait au clavier
  - Pas de clic de souris
- L'utilisateur tape des commandes qui sont exécutées par le système
  - Le shell est donc un « interpréteur de commandes »
  - Chaque commande a une syntaxe particulière
  - Il existe des milliers de commandes différentes
- Les commandes peuvent aussi provenir d'un fichier
  - Le fichier contient les commandes à exécuter
  - L'utilisateur appel le fichier plutôt que de taper toutes les commandes
    - Utile pour les tâches répétitives
- Le shell reste le moyen le plus efficace pour contrôler le système
  - C'est aussi le plus utilisé sous Linux/Unix

Shell ou langage de programmation ?

- Le shell est un véritable environnement de programmation
  - Variables, boucles, structures de contrôle « if »
  - Programmes
- Les programmes écrits pour le shell sont interprétés au moment de l'exécution
  - Aucune compilation préalable n'est utile
  - Les performances n'égalent pas un programme en C
- Les programmes écrits pour le shell sont des « scripts » :

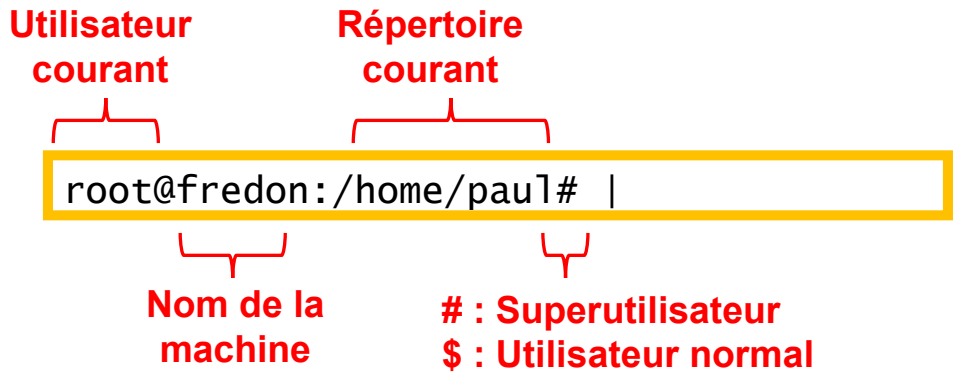
```
# Test de l'existence du fichier
if [ -f $logfile ]
then
  rm $logfile
  echo effacement de ipscan.log effectue
fi

echo scan des ip...
```

Les différents types de shell

- Il existe plusieurs types de shell
  - Bourne shell
  - Korn Shell
  - Bash (Bourne again shell)
  - Tcsh (Terminal C shell)
  - L'interpréteur de commande MS-DOS (Sous Windows)
  - PowerShell (Windows 2008 server)
- Sous Linux, on peut choisir son shell
  - Le shell bash domine le marché actuellement

Présentation de la ligne de commande (CLI)

- La ligne de commande se présente sous forme de texte ayant la signification suivante :

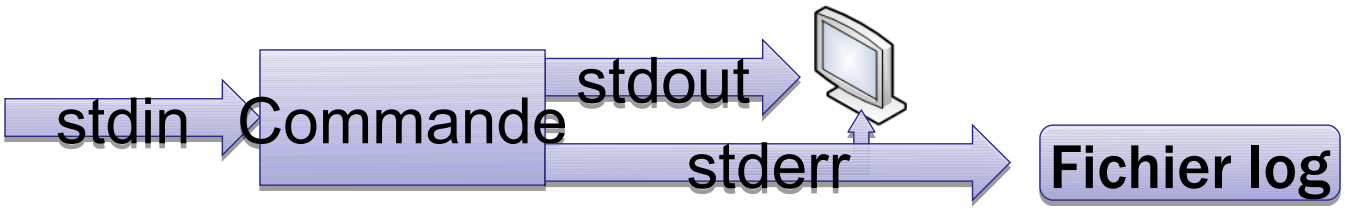


- Il suffit alors de taper une commande pour qu'elle soit exécutée
- Une commande peut être appelée :
  - En tapant son nom puis des arguments ou paramètres
  - Exemple permettant de rechercher dans le répertoire courant les fichiers dont la taille est supérieure à 2Mo

```
root@fredon:/home/paul# find ./ -size +2M
```

Les entrées/sorties

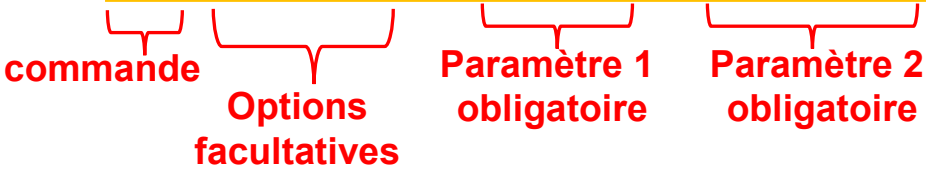
- Chaque commande dispose d'entrées/sorties :
  - Comme l'écran (sortie) ou le clavier (entrée)
- 3 types différents:
  - L'entrée standard définie par le symbole « **stdin** » et le descripteur **0**
    - Provenant du clavier par défaut
  - La sortie standard « **stdout** » et le descripteur **1**
    - Dirigée vers l'écran par défaut
  - La sortie d'erreurs « **stderr** » et le descripteur **2**
    - Egalement redirigée vers l'écran par défaut
- Possibilité de modifier le comportement par défaut
  - En utilisant les redirections d'entrées/sorties



Syntaxe d'une commande

- Chaque commande a une syntaxe particulière
  - Elle est composée d'options et de paramètres
  - Les paramètres permettent de fournir les données nécessaires à l'exécution de la commande
    - Ils sont souvent obligatoires
  - Les options permettent d'offrir des fonctionnalités supplémentaires qui s'adaptent à des besoins spécifiques
    - Exemple : Commande « ls -a » qui liste aussi les fichiers cachés
- Exemples de syntaxe (SYNOPSIS de la commande « cp »)

```
cp [OPTION]... SOURCE... DIRECTORY
```



- Les options sont précédées de signe « - » :

```
cp -r /etc/apache.conf apache.conf.bckp
```

Exemple de la commande « ls »

- « ls » liste les fichiers d'un répertoire donné
- SYNOPSIS :

```
ls [OPTION]... [FILE]...
```

- Quelques options utiles (Attention à la casse) :
  - a : Liste les fichiers cachés
  - l : Listing long (Plus d'infos sur les fichiers)
  - S : Classement par taille de fichiers
  - t : Classement par date de modification
- Exemples :
  - Listing long avec répertoires cachés et fichiers plus gros en 1er

```
ls -alS /etc
```

  - Listing long avec fichiers plus récents en 1er :

```
ls -lt
```

man : Aidez-moi s.v.p

- La commande « man » (Manual) permet d'obtenir de l'aide sur la syntaxe d'une commande
  - Très souvent en Anglais et rarement en français
  - L'aide est en générale très complète
    - Parfois difficile de trouver l'information manquante
- Une page de manuel se décompose en plusieurs parties distinctes
  - NAME : Nom et description rapide de la commande
  - SYNOPSIS : Syntaxe(s) de la commande
  - DESCRIPTION : Description complète de la commande
  - OPTIONS : La description des options
  - AUTHOR : Un mot sur l'auteur
  - BUGS : bugs connus
  - SEE ALSO : Autres commandes connexes à consulter également
  - … (Dépend des commandes)

man : Aidez-moi s.v.p (2)

- Syntaxe :

```
man commande
```

- Exemple : Obtenir l'aide en ligne pour la commande « cp »

```
~# man cp
```

```
CP(1)                                User Commands                                CP(1)

NAME

       cp - copy files and directories

SYNOPSIS

       cp [OPTION]... [-T] SOURCE DEST
       cp [OPTION]... SOURCE... DIRECTORY
       cp [OPTION]... -t DIRECTORY SOURCE...

DESCRIPTION

       Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.

       Mandatory  arguments  to  long  options are mandatory for short options
       too…
```

Redirections des entrées/sorties

- Capacité de rediriger les entrées/sorties d'une commande
  - « stdout » ou « stderr » vers un fichier plutôt qu'à l'écran
  - « stdin » depuis un fichier plutôt que le clavier
- Utilisation des opérateurs suivants :
  - > : Redirection de la sortie vers un fichier
  - >> : Redirection de la sortie à la fin du fichier (concaténation)
  - < : Redirection de l'entrée depuis un fichier
- Exemple de redirection de la sortie vers un fichier :

```
ls -l /etc  > listing-etc.txt
```

  - Le listing est écrit dans le fichier « listing-etc.txt »

- Redirection de l'entrée de la commande « wc » depuis un fichier
  - Compte le nb de lignes du fichier « listing-etc.txt »

```
wc -l  < listing-etc.txt
```

Redirections (2)

- 2 syntaxes supplémentaires possibles pour les redirections

```
n>&m ou
n>fichier
```

  - n : Numéro du descripteur à rediriger
  - m : Numéro du descripteur vers lequel on va rediriger
  - fichier : Fichier vers lequel s'effectuera la redirection
- Rappel : 0 (entrée standard), 1 (sortie standard), 2 (sortie erreur)
- Exemple : Rediriger la sortie d'erreur vers un fichier
  - Les messages d'erreurs seront écrits dans « erreurs.log »

```
cp /home/paul /home/jean 2>erreurs.log
```

- Exemple : Rediriger la sortie standard vers la sortie d'erreur

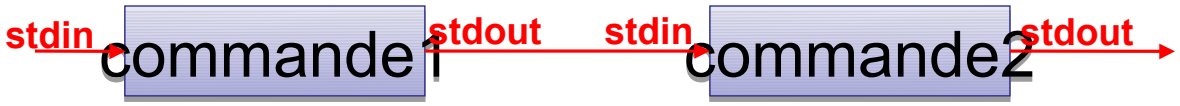
```
cp /home/paul /home/jean 1>&2
```

- Exemple : Rediriger stdout ET stderr vers un fichier

```
commande > fichier.txt 2>&1
```

Les tubes (pipes)

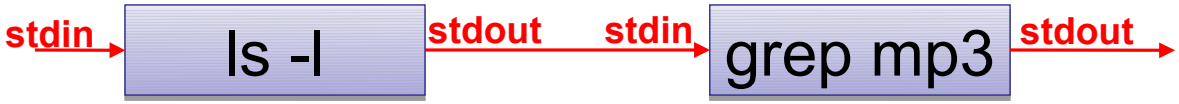
- Il s'agit de rediriger la sortie d'une commande vers l'entrée d'une autre avec l'opérateur « | »



- Élaborer des commandes complexes en une seule ligne
  - Exemple : Filtrer le résultat de la commande « ls » avec « grep »

```
ls -l /etc   | grep 'mp3'
```

    - On obtient la liste des fichiers contenant « mp3 »




```
 rout@fredon:~$ ls -l /etc   | grep 'mp3'
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau1.mp3
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau2.mp3
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau3.mp3
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau4.mp3
```

Exécution conditionnelle de commandes

- Pour exécuter une seule commande, rien de plus simple
  - Taper son nom au clavier
- Pour exécuter plusieurs commandes à la suite
  - L'exemple suivant crée un répertoire, s'y déplace et crée un fichier

```
mkdir toto ; cd toto ; touch alire.txt
```

- L'exécution conditionnelle de commandes est possible
  - Les commandes s'exécutent les unes après les autres sous condition
  - Utilisation des opérateurs && et ||
    - L'exemple suivant exécute commande1 et commande2 seulement si le résultat renvoyé par commande1 est égal à 0

```
commande1 && commande2
```

    - Même chose mais si le résultat de commande1 est différent de 0

```
commande1 || commande2
```

Ca marche ou ca marche pas ?

- Une commande renvoie toujours une valeur
  - 0 si la commande s'est exécutée correctement
  - 1 ou différent de zéro dans le cas contraire
- Exemple
  - La variable « $? » correspond à la valeur renvoyée par la dernière commande exécutée (Donc la commande « cd »)

```
# cd toto
Bash : cd toto : Le répertoire n'existe pas
# echo $?
1
```

- Cette valeur peut-être exploitée dans un script pour connaître le résultat d'une commande avant d'exécuter la suite

Exercices

- Copier le fichier « /etc/syslog.conf » vers son répertoire home

```
cp /etc/syslog.conf ~/
```

- A l'aide de la commande « cat », filtrer les lignes contenant le mot « internet » dans le fichier « toto.txt »

```
cat toto.txt | grep 'internet'
```

- Renommer le répertoire « rapports » vers « rapport2007 » et stocker les messages d'erreurs dans un fichier « mv-rapport.log »

```
mv rapport/ rapports2007 2>mv-rapport.log
```

- Créer le fichier « rap2009 dans le répertoire « rapports » seulement s'il existe

```
test -e rapports && touche rapports/rap2009
```

Caractères spéciaux

- Certains caractères ont une signification particulière
  - Interprétés par le shell
- Astérisque ou étoile : *
  - Interprété comme toute suite de caractères alphanumérique
  - Exemple : Effacer tous les fichiers commençant par « rapport »

```
rm rapport*
```

- Point d'interrogation : ?
  - Interprété comme un seul caractère alphanumérique
  - Exemple : Effacer certains fichiers commencant par « rapport?.doc »

```
rm rapport?.doc
```

    - « rapport1.doc » sera effacé mais pas « rapport12.doc »
- Point virgule : ;
  - Séparateur de commandes

```
cp bilan.txt bilan2007.txt ; rm bilan.txt
```

Caractères spéciaux (2)

- Les crochets : []
  - Remplace un caractère choisi parmi ceux énumérés entre les crochets
  - Exemple : Effacer les fichiers dont la 1ère lettre est « a » ou « b » et se terminant par « .txt »

```
rm [ab]*.txt
```

    - « args1.txt » et « bilan.txt » seront effacés mais pas « comment.txt »
  - Exemple : Effacer les fichiers numérotés de 10 à 29
    - « rapport12.txt » mais pas « rapport3.txt »

```
rm rapport[12][0-9].txt
```

- L'espace
  - Utilisé comme séparateur de paramètres pour une commande
  - Exemple : Effacement de 2 fichiers passés en paramètres

```
rm rapport.doc rapport2008.txt
```

Caractères spéciaux (3)

- L'antislash : \
  - Inhibe le caractère spécial suivant
  - Exemple : Effacer un fichier contenant le caractère spécial espace

```
rm rapport\ .txt
```

- Autres caractères spéciaux : **! , ^, $, <, >, |**
  - **^** : Exprime la négation

```
rm [^r]*.txt
```

  - **$** : Utilisé pour les variables dans les scripts
  - **!** : Utilisé pour accéder à l'historique des commandes (! suivi du numéro de la commande dans l'historique. Voir la commande « history »)
  - **<,>** : Redirections
  - **|** : pipe (tube)

Chaînes de caractères

- Il existe plusieurs délimiteurs de chaînes de caractères
  - Apostrophe (simple quote) : *'texte'*
    - Le texte n'est pas du tout interprété
  - Guillemets (double quotes) : *"texte"*
    - Seuls les caractères \ , $ et ` sont interprétés
    - Utilisé pour du texte qui contient des variables ou des caractères spéciaux
  - Anti-quotes : *`texte`*
    - Le texte est interprété comme une commande à exécuter. Le résultat de cette commande sera substitué
    - Utilisé dans le but d'exploiter le résultat d'une commande
- Exemples

Chaînes de caractères : Exemples

- Exemples
  - Rechercher la chaîne « toto » dans tous les fichiers du répertoire « /home/paul »

```
grep 'toto' /home/paul/*
```

  - Rechercher les fichiers contenant la date d'aujourd'hui
    - 1) Obtenir la date d'aujourd'hui au format JJMMAA

```
root@fredon:~# date +%d%m%Y
28082008
```

    - 2) Exploiter ce résultat pour rechercher cette date dans les fichiers

```
root@fredon:~# grep `date +%d%m%Y` /home/paul/*
```

  - Créer un fichier « alire.txt » dans le répertoire home de l'utilisateur
    - La variable $HOME sera remplacée par sa valeur

```
root@fredon:~# touch "$HOME"/alire.txt
```

Exercices

- Déplacer tous les fichiers commençant par « bilan » vers le répertoire « bilans » qu'il faut créer avant

```
mkdir bilans ; mv bilan*.* bilans/
```

- Afficher la liste des fichiers dont le nom contient « bilan2006 » et « bilan2007 »

```
ls  | grep "bilan" | grep "200[67]"
```

- Afficher la liste des fichiers dont le nom commence par une majuscule suivie d'une minuscule suivie d'un chiffre.

```
ls | grep "[A-Z][a-z][0-9]"
```

- Créer un fichier qui sera nommé « backup_JJMMAA.dat » avec la date du jour

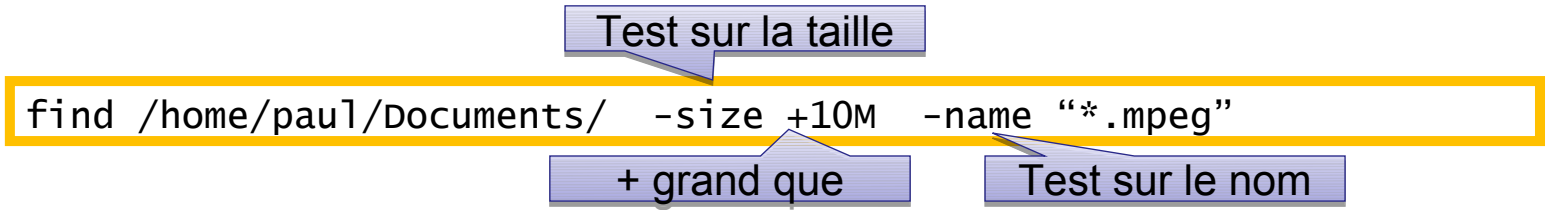
```
touch backup_`date +%d%m%Y`.dat
```

Rechercher des fichiers avec « find »

- Recherche multicritères
  - Par la date, la taille, le nom, …
- Syntaxe :

```
find [-H] [-L] [-P] [path...] [expression]
```

  - « path » : Chemin où chercher
  - « expression » : Expression permettant de définir des critères de recherche
- L'expression est construite autour d'options et de tests
  - Exemple d'option : « maxdepth » limite la profondeur de recherche
  - Exemple de test : « name » recherche par le nom du fichier
- Rechercher les vidéos mpeg dont la taille est supérieure à 10Mo



Filtrer avec « grep » ou « egrep »

- Commande puissante permettant de filtrer les lignes d'une sortie de texte
  - Affiche uniquement les lignes correspondant aux critères de filtrage
  - Très utilisée pour rechercher à l'intérieur des fichiers
- Syntaxes :

```
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]
```

  - « Pattern » : Expression régulière agissant comme filtre
  - « File » : Fichier ou répertoire où est débutée la recherche
- Options intéressantes
  - -r : Permet de recherche dans les sous-répertoires (Peut-être long)
  - -n : Connaître le n° de la ligne et donc la position de l'occurrence trouvée dans le fichier
  - -A : Ajouter à la sortie les lignes situées après la ligne filtrée
  - -B : Ajouter à la sortie les lignes situées avant la ligne filtrée

Exemples d'utilisation de « grep » ou « egrep »

- Quelques exemples d'utilisation de « grep »

Commande	Signification
`grep "[Mm]icrosoft" /etc`	Recherche « Microsoft » ou « microsoft » dans le répertoire « /etc »
`grep "rap200[0-9]" texte.txt`	Recherche « rap2001 », « rap2002 », etc jusqu'à 9
`egrep "[cC]hat|[cC]hien" texte.txt`	Recherche « chien », « Chien », « chat » ou « Chat »
`ls -l | egrep "mpeg|mpg$"`	Liste tous les fichiers dont l'extension est « mpeg » ou « mpg ». Le « $ » indique que la chaîne doit se trouver à la fin de la ligne. Le 1er « | » indique l'opérateur « pipe » alors que le 2ème indique l'opérateur « ou »

Les commandes rwrite et talk

rwrite envoie des messages qui apparaissent dans la console du destinataire (il faut donc qu'il soit logué).
Exemple d'utilisation :

bireme ~ $ rwrite toto@drakkar
Salut, tu manges au pot?
^D

De son côté, Toto voit apparaître ceci dans sa console :

Message from titi@corvette on /dev/pts/4 at 15:12 ...
Salut, tu manges au pot?

talk (ou mieux : ytalk) est une conversation interactive entre deux personnes ou plus. La demande de talk se fait de la même façon que pour écrire un rwrite :

bireme ~ $ ytalk toto@drakkar

Toto reçoit un message dans sa console, et répond par

drakkar ~ $ ytalk titi@corvette

On quitte la conversation en tapant ^C.

Les scripts

- Un script est un fichier contenant un ensemble de commandes exécutées séquentiellement
  - Sous forme de fichier texte contenant les commandes
- Le script ne peut être exécuté que par un interpréteur
  - « /bin/sh » pour le shell bash
- Le langage de script shell est un langage évolué offrant de nombreuses possibilités
  - Boucles, variables, tests avec if, création de fonctions, …
- Dans quels cas utilise-t-on les scripts ?
  - Pour effectuer un travail répétitif
  - Pour des tâches d'administration système
  - Pour installer des programmes
  - Au démarrage du système pour démarrer les services et applis
  - Bref : Tout le temps !!!

Exécuter un script

- Pour exécuter le script, il faut appeler l'interpréteur

```
root@fredon:~# sh monscript
```

- Possibilité de simplifier l'appel en ajoutant la ligne suivante en tête du script

```
#!/bin/sh
```

  - L'appel est alors plus simple

```
root@fredon:~# ./monscript
```

- L'utilisateur courant doit posséder le droit « x » pour le fichier
  - Exemple : Seul, l'utilisateur « paul » pourra exécuter le script

```
-rwxr--r-- 1 paul compta 7406 2008-08-15 14:44 script.sh
```

  - Pour autoriser les membres du groupe « compta »

```
chmod g+x script.sh
```

Format de script

- Il s'agit d'un fichier texte au format suivant :

```
#! /bin/sh

# Commentaires : Fonction du script
# Auteur : toto
# Version : 1.0 - Oct 2008

# Début du script
…
```

- Un script peut accepter des arguments
  - Il faut donc vérifier ces arguments avant de commencer le traitement
  - Rappeler le fonctionnement du script par un message d'erreur

```
Usage de la commande : script.sh arg1 arg2 arg3

arg1 : 1er argument , arg2 : 2ème  argument, arg3 : 3ème  argument
```

Les variables d'environnement

- Ces variables sont définies à l'ouverture de session
- Leurs valeurs dépendent de l'utilisateur connecté
- Exemple : Variable PATH
  - Défini les différents chemins où chercher les commandes

```
root@fredon:~# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/X11R6/bin
```

- La commande « export » permet de créer/modifier une variable
  - Cette modification est temporaire

```
root@fredon:~# export VAR=VALEUR
```

  - Pour rendre l'effet permanent, il faut ajouter « export » au fichier .bashrc
    - .bashrc est un fichier caché
- Pour modifier la variable PATH sans effacer son contenu

```
root@fredon:~# export PATH=$PATH:/nouveau/repertoire
```

Variables de substitution

- Elles sont définies implicitement et peuvent être utilisées à tout moment dans le script
- Quelques variables utiles
  - $0 : Nom du script (Utile lorsqu'on renomme le script)
  - $1 à $9 : Argument 1 à 9 passés au script
  - $# : Nombre d'arguments passés au script
  - $? : Résultat de la commande précédente
  - Exemple

```
#! /bin/sh
cp $1 tata.txt
echo $?
```

  - Exécution

```
root@fredon:~/Documents/cours-shell# ./script.sh file.txt
cp: ne peut évaluer `file.txt': Aucun fichier ou dossier de ce type
1
```

EXEMPLES

1. Essayer les exemples d'affectation des variables donnés ci hautet expliquer les résultats.
2. Dans un shell bash taper les commandes suivantes. Justifier les résultats obtenues de chaque commande.

```
>X1=3
>Y1=10
>Z1=4
>export Y1
>env |grep X1=
>echo $X1
>echo $x1
>env |grep Y1=
>unset Y1
>export X1
>bash
>env |grep X1=
>echo $Z1
>exit
>echo $Z1
```

Évaluation, guillemets et quotes

Avant évaluation (interprétation) d'un texte, le shell substitue les valeurs des variables.
On peut utiliser les guillemets (") et les quotes (') pour modifier l'évaluation :

les guillemets permettent de grouper des mots, sans supprimer le remplacement des variables.
Par exemple, la commande suivante ne fonctionne pas :

```
$ x=Hauru no
bash: no: command not found
```

Avec des guillemets c'est bon :

```
$ x="Hauru no"
```

On peut utiliser une variable entre guillemets :

```
$ y="Titre: $x Ugoku Shiro"
$ echo $y
Titre: Hauru no Ugoku Shiro
```

les quotes (apostrophes) groupes les mots et suppriment toute évaluation :

```
$ z='Titre: $x Ugoku Shiro'
$ echo $z
Titre: $x Ugoku Shiro
```

Parler c'est bien, écouter c'est mieux

Un script qui écoute : la commande read

Jusqu'ici, votre programme est capable de parler, de dire bonjour, mais il n'est même pas capable de vous appeler par votre nom, tout juste par votre login, ce qui n'est pas très intime...

1) Nous allons donc lui donner la faculté d'écouter, grâce à la commande read.
Prenons le script suivant :

```
#!/bin/sh
# Fichier "mon-nom"
echo "Bonjour... Comment vous-appelez-vous ?"
read nom
echo "Je vous souhaite, $nom, de passer une bonne journée."
```

2) Vous connaissez déjà la commande echo. La commande read permet de lire des variables.

Si vous exécutez ce script, après avoir affiché la ligne

```
Bonjour... Comment vous-appelez-vous ?
```

le shell va attendre que vous tapiez votre nom. Tapez par exemple Toto, puis appuyez sur Entrée, et vous verrez :

```
Bonjour... Comment vous-appelez-vous ?
Toto
Je vous souhaite, Toto, de passer une bonne journée.
```

La commande read doit être suivie du seul nom de la variable, non précédé du signe dollar.
Le signe dollar ne doit précéder le nom de la variable que lorsque l'on cite son contenu.

Lire plusieurs variables

La commande read permet également de lire plusieurs variables. Il suffit pour cela d'indiquer à la suite les noms des différentes variables. Exemple :

```
#!/bin/sh
# Fichier "administration"
echo "Écrivez votre nom puis votre prénom :"
read nom prenom
echo "Nom : $nom"
echo "Prénom : $prenom"
```

2) Vous aurez :

```
Écrivez votre nom puis votre prénom :
Hugo Victor
Nom : Hugo
Prénom : Victor
```

Appuyer sur Entrée pour continuer

Nous avons vu comment utiliser read avec un seul argument et avec plusieurs arguments ;
il reste à voir l'usage de read sans argument.
Oui, c'est possible ! Cela équivaut simplement à attendre un réaction de l'utilisateur,
mais sans mémoriser ce qu'il tape.

Concrètement, cela est très utile après un message « Appuyez sur Entrée pour continuer. » Exemple :

```
#!/bin/sh
# Fichier "continuer"
echo "Quelle est la différence entre un canard ?"
echo "(Appuyez sur Entrée pour avoir la réponse)"
read
echo "Les pattes, surtout la gauche."
```

Structure de contrôle « if » (1)

- Permet d'effectuer une exécution conditionnelle

```
if [ expression ]
then
    # commandes à exécuter si la cond. est vraie
else
    # commandes à exécuter si la cond. est faux
fi
```

- L'expression est constituée d'opérateurs
  - Liste des opérateurs numériques
    - -eq : Egalité (Equals)
    - -ne : Non égalité (Non Equals)
    - -lt : Infériorité (Less than)
    - -le : Infériorité ou égalité (Less than)
    - -gt : Supériorité (Greater then)
    - -ge : Supériorité ou égalité (Greater equals)

Structure de contrôle « if » (2)

- Liste des opérateurs sur chaîne de caractère
  - -z : Chaîne vide
  - -n : Chaîne non vide
  - = : Egalité de chaîne
  - != : Non égalité de chaîne
- Liste des opérateurs sur fichiers
  - -L : Lien symbolique
  - -d , -f : Répertoire, Fichier
  - -s : Fichier vide
  - -r, -w, -x : Droits qui s'appliquent (Lecteur, Ecriture, Exécution)
  - -nt : Plus récent (Newer than)
  - -ot : Plus vieux (Older than)

Structure de contrôle « if » (3)

- Liste des opérateurs logiques
  - ! : Négation
  - -a : Et (And)
  - -o : Ou (Or)

Description

Robert le jardinier programme un hachoir en shell, et veut en interdire l'accès à tout autre utilisateur que lui-même, afin de protéger le petit Émile de ce dangereux instrument.
Le programme hachoir se pose la question suivante : est-ce que $USER vaut « Robert » ?

->Si oui, alors le hachoir doit se mettre en marche ;
->si non, alors le hachoir doit refuser de se mettre en marche.

C'est pour les structures de type « si... alors... ou sinon », la commande if a été faite. Elle est constituée de cinq termes, dont trois sont obligatoires :

if (si), qui marque la condition à remplir ;
then (alors), qui marque les conséquences si la condition est remplie ;
elif (contraction de else if, qui signifie « sinon, si... »), qui est facultatif et marque une condition alternative ;
else (sinon), qui est facultatif et marque le comportement à adopter si la condition n'est pas remplie ;
fi (if à l'envers), qui marque la fin de la structure de branchement.

Les autres caractères spéciaux

#	Introduit un commentaire
$	Introduit un nom de variable
&	Termine une commande lancée en arrière plan
;	Sépare des commandes se trouvant sur une même ligne
< > << >> |	Caractères de redirection
` `	Exécute une commande
' '	Délimite une chaîne de caractères où les caractères spéciaux perdent leur signification sauf lui-même
" "	Délimite une chaîne de caractères où les caractères spéciaux perdent leur signification sauf \ $ ` "
\	Déspécilise le caractère qui le suit
( )	Exécuter une commande dans un shell fils
{ }	Regroupe des commandes

La commande test

test expression ou [expression]
Cette commande renvoie un code de retour égal à 0 si l'expression est vraie et une valeur différente de 0 dans cas contraire.

[-a fichier **]** existence du fichier
[-s fichier **]** existence du fichier + taille non nulle
[-d fichier **]** existence + état de répertoire
[-f fichier **]** existence + état de fichier
[-r fichier **]** existence + autorisation de lecture
[-w fichier **]** existence + autorisation d'écriture
[-x fichier **]** existence + autorisation d'exécution
[-n chaîne **]** chaîne non nulle
[-z chaîne **]** chaîne nulle

Structure de contrôle « if » (3)

On peut donc taper :

```
if commande ; then commandes ; else commandes ; fi
```

ou bien (car le point et virgule équivaut à un saut de ligne) :

```
if condition
    then commandes
elif condition
    then commandes
else commandes
fi
```

Par exemple, pour le hachoir de Robert le jardinier, on aura :

```
if test $USER = Robert
    then echo "Le hachoir va se mettre en marche."
    mettre en marche le hachoir
else echo "Quand tu seras grand, $USER."
    echo "Et fais bien attention en traversant la rue."
fi
```

Structure de contrôle « if » (3)

Comme vous pouvez le constater, à la suite des commandes **then** et **else**, vous pouvez mettre autant de commandes que vous le voulez. Le **shell** exécute tout ce qu'il trouve jusqu'à la prochaine instruction.

Par exemple, si la condition indiquée par **if** est remplie, le **shell** va exécuter tout ce qui suit **then** jusqu'à ce qu'il trouve l'une de ces instructions :

elif,
else ou
fi.

```
#!/bin/sh
# Fichier "mon-pwd"
if test $PWD = $HOME
    then echo "Vous êtes dans votre répertoire d'accueil."
elif test $PWD = $HOME/bin
    then echo "Vous êtes dans votre répertoire d'exécutables."
elif test $PWD = $HOME/bin/solaris
    then echo "Vous êtes dans votre répertoire d'exécutables pour Solaris."
else echo 'Vous êtes dans un répertoire quelconque, qui n'est ni $HOME,'
    echo 'ni $HOME/bin, ni $HOME/bin/solaris.  "Vous êtes dans $PWD."
fi
```

Structure de contrôle « case »

Choix multiple case

Syntaxe :

case mot **in**
[modèle [| modèle] ...) suite_de_commandes ;;] ...
esac

Le shell évalue la valeur de mot puis compare séquentiellement cette valeur à chaque modèle. Dès qu'un modèle correspond à la valeur de mot, la suite de commandes associée est exécutée, terminant l'exécution de la commande interne composée case.

Les mots **case** et **esac** sont des mots-clé ce qui signifie que chacun d'eux doit être le premier mot d'une commande.

suite_de_commandes doit se terminer par deux caractères point-virgule collés, de manière à ce qu'il n'y ait pas d'ambiguïté avec l'enchaînement séquentiel de commandes **cmd1** ; **cmd2**.

Un modèle peut être construit à l'aide des caractères et expressions génériques de bash [cf. § Caractères et expressions génériques]. Dans ce contexte, le symbole **|** signifie OU.

Pour indiquer le cas par défaut, on utilise le modèle ***.** Ce modèle doit être placé à la fin de la structure de contrôle case.

Le code de retour de la commande composée case est celui de la dernière commande exécutée de suite_de_commandes.

Structure de contrôle « case »

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

```
#!/bin/bash
#@(#)oui
read –p ″Entrez votre réponse : ″  rep
case $rep in
o|O ) echo OUI ;;
*)    echo Indefini
esac
```

Rq : il n'est pas obligatoire de terminer par ;; la dernière suite_de_commandes

Structure de contrôle « if » (3)

- Exemples

```
# Teste si le paramètre $1 est egal à 2
if [ $1 -eq 2 ]
then
   # Commande
fi
```

```
# Teste si le paramètre $1 est un fichier OU un répertoire
if [ -d $1 -o -f $2 ]
then
   # Commande
fi
```

```
# Teste si le fichier existe
if [ ! -f "/etc/toto.conf«  ]
then
   # Commande exécutée si le fichier n'existe pas
fi
```

Exercices

- Tester si le paramètre 2 est un fichier

```
if [ -f $2 ]; then ; fi
```

- Tester si le paramètre 1 égale à toto ET paramètre 2 supérieur à 3

```
if [ $1="toto" -a $2 -gt 3 ]; then ; fi
```

- Tester si le paramètre 1 n'est pas un répertoire ET s'il a le droit d'écriture

```
if [ ! -d $1 -a -w $1 ]; then ; fi
```

- Script capable de créer le répertoire passé en argument et de vérifier que sa création n'a pas provoqué d'erreurs

```
if [ ! -d $1 ]; then
   mkdir $1
   if [ $? -eq 0 ]; then
      echo "Erreur lors de la création de $1«
      exit 1
   fi
fi
```

```
echo -n "Entrez un nombre compris entre 1 et 3 inclus >"
read nombre
if ["$nombre" = "1"]; then
    echo "Vous avez entré 1"
else
    if ["$nombre" = "2"]; then
        echo "Vous avez entré 2"
    else
        if ["$nombre" = "3"]; then
            echo "Vous avez entré 3"
        else
            echo "Vous n'avez pas entré de nombre"
            echo "entre 1 et 3"
        fi
    fi
fi
```

case, peut être utilisée pour construire un programme équivalent :

```
# !/bin/bash

echo -n "Entrez un nombre compris entre 1 et 3 compris >"
read nombre
case $nombre in
    1 ) echo "Vous avez entré 1"
  ;   ;
    2) echo "Vous avez entré 2"
        ;;

    3) echo "Vous avez entré 3"
        ;;
    *) echo "Vous n'avez pas entré de nombre"
       echo "entre 1 et 3"
esac
```

La commande case s'utilise de la façon suivante :

```
case mot in

    motifs ) instructions ;;

esac
```

case exécute sélectivement les instructions qui correspondent au motif.
Vous pouvez avoir une quantité quelconque de motifs et d'instructions.
Les motifs peuvent contenir du texte ou des caractères de remplacement.
Vous pouvez avoir de multiples motifs séparés par le caractère "|" qui veut dire ou logique.

Exemple :

```
echo -n "Entrez un nombre ou une lettre >"
read caractere
case $caractere in
        # Vérification pour une lettre
    [a-z] | [A-Z] ) echo "Vous avez entré la lettre $caractere"
        ;;

        # Vérification pour un nombre
    [0-9]  )       echo "Vous avez entré le nombre $caractere"
        ;;

        #Vérification pour tout autre chose
    * )            echo "Vous n'avez pas entré de nombre ni de lettre"
esac
```