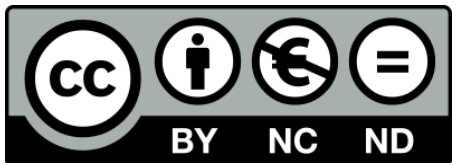
Programmation Shell sous UNIX et Linux

Jean-Pierre Messager (jp@xiasma.fr)

17 janvier 2023 – version 0.9a



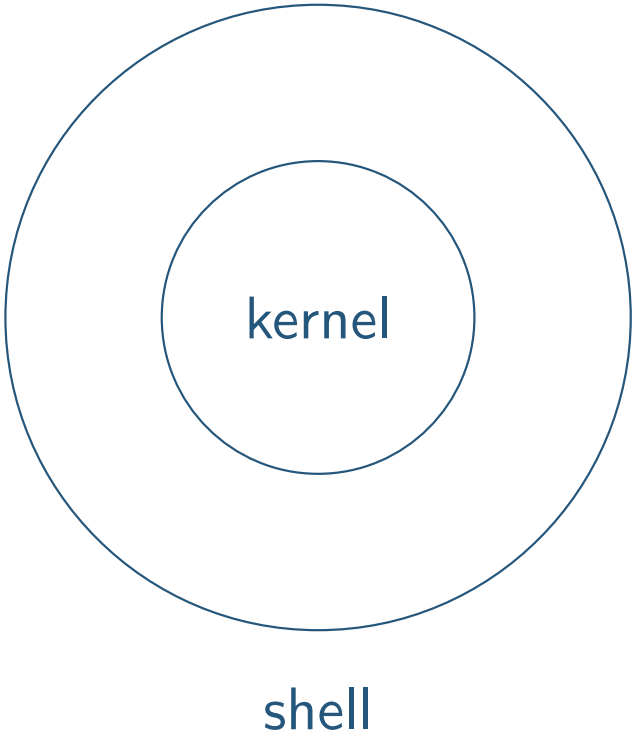
Utilisation commerciale interdite sans autorisation

Introduction

Architecture d'UNIX

- Le noyau « enrobe » le matériel (CPU, mémoire, périphériques)
- Le *Shell* (coquille) « entoure » le noyau et intéragit avec les utilisateurs sous formes de commandes
- Commandes et Shells invoquent des appels systèmes, par l'intermédiaire d'appels de bibliothèques, qui sont traités par le noyau

Arg



Bases de la ligne de commande

Le(s) Shell(s)

- En 1977 UNIX v7 (AT&T) est fourni avec le Shell Bourne
- Développé par Stephen Bourne
- En 1978 Bill Joy publie le c-shell
- David Korn publie le *Korn Shell* en 1983
- Le projet GNU développe le *Bourne Again Shell* : Bash
- Paus Falstad publie *zsh* en 1990

Les normes et bonnes pratiques

- En dehors de certains UNIX basé sur BSD le Shell par défaut est généralement de la famille Bourne :
  - Bash sur la plupart des distribution de GNU/Linux
  - Bash ou zsh sur macOS (selon la version)
- La norme POSIX inclut le support d'un Shell Bourne
- *Csh Programming Considered Harmful* : https://www-uxsup.csx.cam.ac.uk/misc/csh.html

La famille des *Shells Bourne*

Portabilité

- Les shells de la famille *Bourne* sont compatibles entre eux à 99%
- Nous préciserons les spécificités quand nous en rencontrerons

Fonctionnalités

- Bash, ksh et zsh ont largement amélioré le Shell Bourne historique
- Meilleure ergonomie en interactif
- Commande d'aide à la mise au point de scripts
- Types de données supplémentaires

Syntaxe

Interface en ligne de commande

- Le Shell présente une invite (*prompt*), généralement :
  - Qui vous êtes *(whoami)*
  - Sur quel système *(hostname)*
  - Quel est le répertoire de travail *(pwd)*
  - Êtes-vous *root (super user)* ? $ (non) ou # (oui)
- Il attends ensuite votre saisie :
  - Suite d'éléments séparés par espace
  - D'abord la commande
  - Puis ses éventuels arguments, toujours séparés par espace(s)

Dans le cours nous représenterons l'invite par le simple caractère $

```
$ whoami
$ uname
$ uname -a
```

Documentation en ligne !

Il y a beaucoup de commandes !

- UNIX est un environnement logiciel riche, il y a beaucoup d'outils puissants
- Il ne s'agit pas de retenir tous ces détails !

Manuel en ligne

- Un manuel est fourni : pages *man* (et parfois commande *info*)
- `man nom_de_page` : montre la première page de ce nom dans le manuel
- Noms de commandes, de fonctions système, de configuration, de concepts
- Le manuel est organisé en sections : `man man`
- On peut faire : `man section nom_de_page`
- Les commandes `apropos` et `whereis` permettent de faire des recherches

Types de commande et arguments

Types de commandes

- Commandes internes : exécutées directement par le Shell
  - Par nécessité : `cd`, `export`, `type`, `help`, ...
  - Pour l'optimisation ou la simplicité : `echo`, `pwd`, ...
- Commandes externes : lancement de programmes
  - Recherchées dans une liste de répertoires : `echo $PATH`
  - Gestion des fichiers et répertoires : `ls`, `mkdir`, ...
  - Accès aux données : `cat`, `less`, ...

Types d'arguments

- Le ou les objet concerné(s) : fichier, répertoire, nom de système, ...
  - Suite de caractères
  - On utilise `"..."` ou `'...'` si elle contient des espaces
- Options modifiant le comportement de la commande
  - Forme courte : tiret puis un ou plusieurs caractères
  - Forme longue : double tiret puis des termes

Exemples

Lister des fichier du répertoire de travail ou d'ailleurs

```
$ pwd
$ ls
$ ls /etc
$ ls -l
$ ls -a
$ ls --all
$ /usr/bin/ls -ld /etc
```

Chemins absolus et relatifs

Chemins absolus

- **Débutent par /**
  - Indiquent la voie à suivre **à partir de la racine**
  - Puis des noms séparés par /
- Puisque l'on part de la racine, ils désignent la même ressource quel que soit le contexte (répertoire de travail)

```
$ ls /etc
$ ls -l /etc/hosts
$ ls /home/joe/Documents
```

Exemples

- Traduire en chemin absolu : « le fichier nommé *file* situé dans le répertoire *bin* situé dans le répertoire *usr* situé à la racine »
- De même pour : « Le répertoire nommé *journal* situé dans le répertoire *log* situé dans le répertoire *var* situé à la racine »

Chemins absolus et relatifs

Chemins relatifs

- **Ne** débutent **pas** par /
- Généralement relatifs au répertoire de travail
  - Indiqué dans l'invite du Shell (~ représente « chez vous », votre répertoire de connexion, généralement dans `/home`)
  - On peut afficher le chemin absolu vers le répertoire de travail avec la commande `pwd` *(print working directory)*

Se déplacer dans l'arborescence

Comment changer de répertoire de travail ?

- Commande interne du Shell : `cd` *(change directory)*
- Sans argument vous ramène à votre répertoire de connexion
- Accepte comme argument un chemin absolu ou relatif

```
$ cd /home ; ls
$ cd /lib ; ls
$ cd modules ; ls
```

Se positionner dans le système de fichier

- Il est *fondamental* de toujours savoir « où l'on est »
- La même commande peut avoir des effets différents selon dans quel répertoire on se trouve

Répertoires spéciaux . et ..

- Dans chaque répertoire il existe deux entrées particulières
  - `.` : mène au répertoire lui-même
  - `..` : mène au répertoire de niveau supérieur (parent)
- Ainsi ces chemins *relatifs* sont possibles :
  - `../linus/Documents/proc.txt`
  - `../../etc/hosts`
  - et mènent bien à des fichiers ou non selon le contexte (répertoire de travail)

Savoir où l'on est...

Ceci est tout aussi important en interactif que dans un script Shell.

Un script :

- Peut se positionner dans un répertoire particulier
- Être écrit de façon à avoir un comportement indépendant de son répertoire de travail
- Être conçu pour agir en fonction de son répertoire de travail...

Tout dépend de l'objectif !

Paramètrage du Shell

Fichiers lus au démarrage

- Shells Bourne ici (Bash, zsh, ksh, ...)
- Quand un Shell se lance il parcourt (il « source ») plusieurs fichiers scripts
- Selon le Shell les noms de ces fichiers diffèrent (`.profile` à l'origine du Shell Bourne)
- Nous allons donner les détails pour *Bash*

- `/etc/bash...` : pour tous les utilisateurs
- `~/.bash_login` : pour un Shell de connexion
- `~/.bashrc` : pour tout Shell (Terminal...)

Personnaliser notre configuration

- Ouvrez votre `.bashrc` avec un éditeur de texte
- Ajoutez les lignes suivantes :

```
PATH=~/bin:$PATH
alias ll='ls -l'
alias la='ls -a'
echo "Bonjour $USER !"
```

- Créez un répertoire `bin` dans votre répertoire de connexion.
- Fermez votre terminal et lancez en un nouveau
- Affichez la valeur de la variable `PATH`

- Consultez le manuel de *Bash* pour en savoir plus sur les possibilités de personalisation... Vaste sujet !
- Les distributions de GNU/Linux fournissent un environnement initial bien pensé qu'il est peu nécessaire d'adapter

Scripts simples

Un script est un fichier texte

- Contenant des commandes Shell
- Bonne pratique : débuter en indiquant le Shell concerné (ligne *shebang*)
- On peut y créer des variables quelconques
- Qui seront d'environnement (héritées) si vous utiliser `export`

```
#!/usr/bin/env bash
cours="Programmation Shell"
echo "Formation $cours"
echo "Vous êtes connecté à $HOSTNAME !"
read -p "Votre prénom : " prenom
echo "Bonjour $prenom !"
```

```
$ ./bonjour
```

Installation de notre script

- Copiez le script `bonjour` dans votre répertoire `~/bin`
- Verifiez que vous pouvez l'exécuter de n'importe où en saisissant simplement son nom

```
$ cd /tmp
$ bonjour
```

Redirection des sorties

Shells et Entrées/Sorties

- Le Shell communique à travers un terminal
- Il y a trois flux de communications définis :
  - *stdin* ou 0 : entrée standard
  - *stdout* ou 1 : sortie standard
  - *stderr* ou 2 : sortie pour messages d'erreur ou avertissements
- Quand vous exécutez une commande elle hérite de ces trois canaux et les utilise pour communiquer (lire et écrire)

Exemple : `cat` lit les fichiers passés en arguments s'il y en a et *sinon* lit l'entrée standard (qui se termine au clavier par CTRL+D)

```
$ cat /etc/networks /etc/hosts
$ cat
Hello
Hello
^D
```

Redirection de l'entrée et des sorties

Redirection entrée standard

```
$ cat < /etc/hosts
```

Redirection de la sortie standard ou d'erreur

```
$ ls -l > liste
$ cat liste
$ ls -l nothere 2> erreur
$ cat erreur
```

Par défaut le fichier est vidé si il existe

Redirection en ajout à la fin de la sortie

```
$ ls -l /usr /bin /lib >> liste
$ cat liste
```

Pour faire disparaître une des sorties : redirigez-la vers le fichier spécial de périphérique `/dev/null` !

À vous. . .

Reportez-vous à l'exercice 1 du manuel

Substitution d'arguments : jokers

Traiter plusieurs fichiers à la fois

- Le Shell propose des caractères spéciaux (« jokers ») pour évoquer plusieurs noms de fichiers (ou chemins) existants
- `*` : n'importe quelle séquence de caractère
- `?` : un caractère quelconque
- `[aeiou]` : un caractère dans une liste
- `[a-z]` : un caractère dans un intervalle
- `[!...]` : un caractère pas dans la liste ou l'intervalle

```
$ ls -l *.txt
$ ls -ld /usr/[a-m]*
$ ls -ld /usr/[!a-m]*
$ ls -ld /usr/[n-z]*
$ file *
```

Application à la manipulation de fichiers

Reportez-vous à l'exercice 2 du manuel.

Environnement

Une commande s'exécute dans un *environnement*

- Un ensemble de variables ayant des valeurs
- Paramètre le comportement du Shell et de bien d'autres commandes et applications
- Permet de récuperer des informations précieuses : nom du système, nature du système d'exploitation, identifiant ulisateur, ...

```
$ env
$ echo $LANG
$ man ls
$ LANG=C man ls
$ man ls
$ export LANG=C
$ man ls
```

Documents « en place »

Spécifier le contenu de l'entrée d'une commande

```
$ rot13 <<EOT
> bonjour
> au revoir
> EOT
obawbhe
nh eribve
```

- L'entrée de la commande sera le texte jusqu'au marqueur de fin spécifié librement
- On utilise communément EOT ou EOF *(End Of Text, End Of File)*
- Surtout utilisé dans des scripts

Tubes et filtres

Nous avons vu comment rediriger la sortie standard vers un fichier et comment lire un fichier via l'entrée standard

- Les Shells UNIX proposent une possibilité encore plus puissante !
- Envoyer la sortie standard d'une commande sur l'entrée d'une autre
- Et ainsi de suite : tube *(pipe)* et *pipelines* !

```
$ getent passwd | grep /home
$ getent group | grep $USER
```

La commande `grep` est un *filtre* qui ne passe en sortie que les lignes contenant, *ici*, `/home` ou votre identifiant

Commandes grep et sed

grep est un filtre

- Filtre : commande qui lit (sauf exception) son entrée standard et envoye le résultat sur la sortie standard
- Les filtres sont conçus pour être utilisés avec des tubes

```
grep motif [fichier ...]
```

Un autre filtre : sed

Aussi expressif que `vi` ! Par exemple pour rechercher remplacer :

```
sed -e 's/motif/remplacement/[gi...]'
```

Afficher les sous-répertoires de `/usr` ainsi :

```
$ ls -d /usr/* | sed -e 's/\/usr\//Répertoire /'
```

Beaucoup d'outils, d'applications et langages de programmation acceptent la même syntaxe pour les *motifs*. Nous les illustrerons avec `grep`.

Expressions régulières

- C'est la syntaxe pour les motifs utilisable avec `grep` et autres
- Principe similaire aux *jokers* du Shell mais beaucoup plus puissant
- Utile en traitement de données, administration système, programmation, linguistique, ...
- La syntaxe étendue (moderne) est utilisable avec `grep -E` et est préférable

Caractères spéciaux d'expressions régulières

- `.` : n'importe quel caractère
- `[...]` : caractère dans une liste ou un intervalle
- `[^...]` : caractère *pas* dans une liste ou intervalle
- `*` : l'expression précédente répétée zero fois ou plus
- `+` : l'expression précédente répétée une fois ou plus
- `?` : l'expression précédente présente une fois ou absente
- `\` : le caractère spécial qui suit doit être considéré littéralement

Expressions régulières

- ^ et $: ancres (débuts et fin de lignes)
- (...) : regroupement et étiquetage
- (chat|chien) : alternative

```
$ getent passwd | grep '^d.*'
# Liste les groupes dont vous êtes l'unique membre
# (hors groupe primaire) :
$ getent group | grep ':votrelogin$'
$ grep 'host$' /etc/hosts
# " " au lieu de ' ' pour évaluer $LOGIN
$ who | grep "^$LOGIN"
```

À vous...

Reportez-vous à l'exercice 3 du manuel.

Autres outils et langages

Outre `grep` et `sed` les expressions régulières profitent à beaucoup d'outils et langages :

- *find*
- *awk*
- *Perl* et *Python*
- *Javascript*, *Java*, *C#*, ...
- Microsoft Office et LibreOffice
- Bases de données (PostgreSQL par exemple)

Autres filtres et outils

Avant de réinventer la roue. . .

. . . il vaut mieux trouver le bon outil !

Une liste (non exhaustive) à considérer :

- `sort` : trier
- `tr` : transformer/translater des caractères
- `uniq` : supprimer des doublonss
- `cut` : découper des lignes
- `column` : mettre en forme des tableaux
- `tee` : réplique un flux vers un fichier
- `cmp` et `diff` : comparent deux fichiers ou ensembles de fichiers
- . . .

Tâches en avant- et arrière- plan

Plusieurs programmes ?

- UNIX et Linux sont intrinsinquement multitâches
- Beaucoup de processus tournent en parallèle sur votre système
  - Gestion des périphériques
  - Services réseaux, journaux d'activité, ... (« Démons »)
  - Composants du bureau Gnome ou autre

Shell et tâches

- Lorsque le Shell exécute une commande externe :
  - Il crée une copie de lui-même *(fork)*
  - Le « parent » se met en attente
  - Le « fils » exécute la commande *(exec)* et se termine
  - Le « parent » reprend le contrôle du terminal
- Il est possible de contrôler ce comportement

Exécution et « sourçage »

Quand on exécute un script Shell il se passe la même chose :

- Le Shell de départ crée une copie de lui même
- Si le script est un script Bash (cf. ligne « shebang ») ce descendant exécute le script puis se termine
- Il est donc impossible ainsi d'agir sur l'état du Shell parent : environnement, répertoire de travail, fonctions et alias définis, etc.

source

- La commande interne « . » (point) ou `source` (Bash) demande à ce que ce soit le Shell courant qui exécute le script
- Ainsi on peut agir sur l'état du Shell courant
- C'est ainsi que les scripts de personnalisation fonctionnent (bashrc, bash_login, etc.)
- Permet de concevoir des bibliothèques de fonctions « sourcées » par d'autres scripts

Avant-plan et arrière-plan

Contrôle des tâches

- Une commande est par défaut en avant-plan et le Shell est en attente
- En terminant la commande par & on demande au Shell de lancer une tâche en arrière-plan
- Une tâche qui veut contrôler le terminal se mettrait en attente...
- On peut contrôler plusieurs tâches :
  - `jobs` : liste des tâches du terminal courant
  - `fg` : passe une tâche au premier-plan *(foreground)*
  - `bg` : passe une tâche en arrière-plan *(background)* où elle s'exécute (si possible...)
  - ctrl+Z : « stoppe » la tâche en avant-plan

```
$ xeyes &
$ xeyes &
$ jobs
```

Processus

Tâches en cours et processus

- Les tâches contrôlées par un Shell ne sont qu'une partie des processus en cours gérées par le noyau UNIX ou Linux
- Des commandes permettent de les examiner tous ou sélectivement

```
$ ps
$ ps aux # ou ps -elf
$ pstree
$ top # q pour quitter
```

Envoi de signaux

Communiquer avec un processus ?

- UNIX prévoit l'envoi de *signaux* aux processus (ctrl-C, ...)
- Utile pour leur demander de relire leur configuration, s'arrêter, s'endormir, les « tuer » ...
- Liste des signaux : `kill -l`
- Envoyer un signal à un processus : `kill -TERM 88742` ou `kill -TERM %1` ou `pkill`

Principaux signaux

- INT, TERM, QUIT : fin « gracieuse » si non bloqué
- HUP : souvent signifie « relire sa configuration »
- KILL (9) : fin sans condition (dangereux !)
- STOP et CONT : endormir et réveiller

```
$ ps aux | grep xeyes
$ pgrep xeyes
$ pkill xeyes # par défault : TERM
```

Interception de signaux

- Un processus peut demander à ce qu'un traitement spécifique soit executé lors de la réception d'un signal.
- C'est possible dans un shell avec la commande interne `trap`
- La commande exécutée peut faire du ménage avant de se terminer
- La commande exécutée peut être vide (blocage du signal)

clean

```
#!/usr/bin/env bash
touch .clean_lock
trap 'rm -f .clean_lock' 'INT'

sleep 60
rm -f .clean_lock
```

Scripts Shells

Automatiser des tâches complexes

- Le Shell nous permet d'automatiser des tâches complexes
- Déclencher certaines opérations sélectivement
- Vérifier le bon fonctionnement d'une opération
- Tester des propriétés de fichiers
- Réaliser des traitements répétitifs

Code de retour

- Quand un processus se termine il renvoie un *code*
- 0 signifie « ok », autre chose qu'il y a un problème
- Vos propres scripts peuvent se comporter ainsi
  - Terminer par `exit 0` si tout va bien
  - Une autre valeur s'il y a eu un problème

Tester une valeur de retour

Pour des opérations simples

- Opérateurs logiques « court-circuit » : && (et) ; || (ou)
- `comm1 && comm2` : comm2 s'exécute si comm1 a réussi
- `comm1 || comm2` : comm2 s'exécute si comm1 a échoué

```
$ ls /tmp/test || echo "/tmp/test absent"
$ ls /tmp/test && echo "/tmp/test présent"
$ touch /tmp/test # et recommencer
```

Opérateur `test`

- La commande (interne ou externe) `test` permet de tester, entre autres, des propriétés de fichiers
- `test -f ...` : fichier normal existe
- `-r`, `-w`, `-x` : fichier lisible, écrivable, exécutable
- `help test` ou `man test`

À vous...

Reportez-vous à l'exercice 4 du manuel.

Autre syntaxe d'exécution conditionnelle

Pour des opérations plus complexes. . .

S'il y a plus qu'une simple commande à exécuter ou non une autre forme est préférable (la clause *else* n'est pas obligatoire) :

```
if test -f /tmp/truc
then
  echo "truc est présent dans /tmp"
else
  echo "truc est absent dans /tmp"
fi
```

Il existe aussi une close `elif` pour simplifier les tests en cascade.

test, [, [[et ((

Avec cette syntaxe la notation `[ ... ]` (équivalente à `test`) est plus lisible :

```
if [ -f /tmp/truc ]
...
```

Une extension spécifique à Bash

La forme `[[` est une extension propre à Bash et ksh (non compatible avec les Shells Bourne génériques) qui résoud certains problèmes délicats de syntaxe.

Réécrivons les scripts précédents avec cette forme.

Pour les tests numériques utilisez `(( ... ))` : `if (( $n > 2 ))`

Boucle `for` et substitution

`for` permet de faire parcourir une liste de valeurs à une variable :

```
for name in Ada Alan Brian Linus
do
  echo "$name est célèbre en informatique."
done
```

Substitution

Deux formes de substitution sont particulièrement utiles ici (et fonctionnent aussi avec d'autres commandes que `for`) :

- `$*` ou (mieux) `"$@"` pour la liste des arguments reçus par notre script (note : on peut aussi y accéder un par un : `$1`, `$2`, ...)
- `$( commande ... )` : la sortie de la commande devient une liste d'arguments

Substitution en liste et boucle for

```
for arg in "$@"
do
  echo "Param : $arg"
done
```

```
# que fait le who | ... cut ... ?
for user in $( who | cut -d' ' -f1 )
do
  echo "$user est connecté"
done
```

```
for pid in $( pgrep xeyes )
do
  kill -TERM $pid
done
```

À vous...

Reportez-vous à l'exercice 5 du manuel.

Boucle `while`

Boucler sur une condition

- Exécute répétivement une séquence de commandes
- Même type de condition que `if` (état de sortie, commande `test`, `[` ou `[[`, etc.)

```
while [ ! -f /tmp/chose ]
do
  echo "Pas de chose dans /tmp"
  sleep 1 # dort une seconde
done
echo "chose trouvée dans /tmp !"
```

Lecture de l'entrée standard

Comment lire l'entrée ligne à ligne ?

- Avec la boucle `while`
- En récupérant chaque ligne dans une variable
- `read` échouera en fin de flux : sortie de boucle

```
# Suppression des espaces

while read line
do
  echo $line | tr -d ' '
done
```

À vous...

Référez-vous à l'exercice 6 du manuel.

Substitutions

Nous avons vu quelques exemples de substitutions d'arguments :

- `$name` (ou mieux `"${name}"`) pour une variable
- `$( commande ... )` pour la sortie d'une commande
- `*`, `?` et `[...]` pour des noms et chemins d'accès à des fichiers ou des répertoires

Ces substitutions ont lieux aussi dans une chaîne encadrée par des guillemets (`"..."`), mais pas par des apostrophes (`'...'`).

Autres substitutions

- Ensembles d'éléments : `{a,b,c}` ou `{0..9}`
- Produit cartésien d'ensembles : `{P,F}{P,F}`, `{1..6}{1..6}`
- Calculs numériques ; `$(( 42 + 12*7 ))`

À vous. . .

Reportez-vous à l'exercice 7 du manuel

Récupération des arguments

Variables spéciales

- `$#`: nombre d'arguments
- `$0` : nom du script lui-même
- `$1`, `$2`, ... : arguments individuels
- `$*` ou (mieux) `"$@"` : liste des arguments

La commande `shift` supprime `$1` et décale la suite d'argument

Commande `getopts`

- Permet d'analyser les arguments reçus dans une boucle `while`
- Attend une chaîne de spécification indiquant les options attendues.
- Suivi de `:` si l'option attend une valeur, récupérable dans `OPTARG`
- Produit une erreur si une option inconnue est passé, sauf en cas de `:` initial

Exemple avec getopts

```
verbose=0
while getopts "i:o:v" option
do
  case "${option}" in
    i)
      infile="${OPTARG}"
      ;;
    o)
      outfile="${OPTARG}"
      ;;
    v)
      verbose=1
  esac
done
```

Exemple avec getopts (suite)

```
[ -z "${infile}" -o -z "${outfile}" ] \
  && echo "Usage : $0 -i infile -o outfile [-v]" \
  && exit 1

[ "${verbose}" == "1" ] \
  && echo "Envoi de ${infile} dans ${outfile}"

tr '[:lower:]' '[:upper:]' < "${infile}" \
  > "${outfile}"
```

Fonctions

Programmation structurée

- On peut définir des fonctions facilement, elles sont appelées comme des commandes par la suite.
- On peut y manipuler des variables locales ou globales

```
add() {
  local res
  res=$(( $1 + $2 ))
  echo $res
}
add 42 12
```

- Mot clé `function` optionnel
- Les arguments se récupèrent comme pour un script : `$n`, `"$@"`, `shift`, ...
- La commande `return` permet de spécifier un code de retour

À vous. . .

Reportez-vous à l'exercice 8 du manuel.

Compléments sur les variables

Variables Shells et environnement

- Une variable *exportée* devient une variable d'environnement
- Elle alors visible par les processus fils
- Commande interne `export`

Valeurs par défauts

- `${name:-joe}` : vaut `$name` si name existe non vide sinon *joe*
- `${name:=joe}` : *name* vaudra joe si name est inexistant ou vide
- `${name:?Erreur: name inexistant ou vide}` : erreur et arrêt du script si *name* n'existe pas ou vide

Tableaux

Extension de Bash

```
Food=('spam', 'ham', 'egg')
echo "${Food[0]}" "${Food[2]}"
for food in "${Food[@]}"
do
  echo ${food}
done
```

Beaucoup de manipulation sont possibles, voir l'annexe du manuel d'exercice.

AWK

Un autre langage

- Langage créé par Alfred Aho, Peter Weinberger et Brian Kernighan en 1977
- Pour pallier les difficultés à traiter des données en Shell
- Réimplémenté par le projet GNU : *gawk*

Orienté traitement de données

- Les enregistrement sont des lignes
- Les colonnes sont des champs automatiquement extraits
- On spécifie le séparateur
- Syntaxe inspirée du Shell Bourne et du C

Exemple de script awk

S'utilise souvent en *one liner* :

```
$ getent passwd | awk -F: "{ print $0, $6}"
```

Ou dans un script

Un bloc de code est précédé d'un critère d'exécution (BEGIN, END, expression régulière, test)

```
#!/usr/bin/awk -f

BEGIN { FS=":";  print "Utilisateurs\tShell" }

/home/ { print $1, "\t", $NF }

END { print "Fin" }
```

```
$ getent passwd | ./exemple_awk
```

Au délà de Bash et awk...

awk aussi a montré ses limites...

- Le langage *Perl* a été conçu pour dépasser les limites de awk dans les années 1990
- Il a longtemps dominé l'administration système UNIX/Linux, voire MS Windows et l'analyse de données (projet du génome humain par exemple)

Python

- Python a grignoté la position de *Perl* par la suite
- Très lisible et facile à apprendre
- Beaucoup de modules simplifiant la programmation système
- Leader incontesté de l'analyse de données et de l'IA