Le Shell Bash

pour Unix/Linux

Remarque 1. Le Shell Bash de mac os X présente des différences avec celui traité ici : certaines commandes, notamment, ne sont pas les mêmes.

Sources :

1. MOOC « maîtriser le Shell Bash » proposé par l'Université de la Réunion : https://www.fun-mooc.fr/courses/course-v1:univ-reunion+128001+session01/info.

Table des matières

1. VOCABULAIRE

Répertoire — autre mot pour « dossier ».

Langage interprété — Le shell est un langage *interprété* : il faut un programme externe (bash) qui interprète le code source pour qu'il soit exécuté.

Un exemple de langage non interprété serait le *C*, qui est un langage compilé : il faut un compilateur pour transformer le code source en langage machine et ensuite on peut exécuter le programme compilé sans l'aide d'un programme externe.

Langage typé — qiu oblige à déclarer le type des variables en début de programme ; Bash n'est pas du tout typé.

2. QUELQUES COMMANDES UTILES

2.1. Commandes internes au bash

Les commandes *internes* sont fournies avec le bash et ne dépendent pas du système d'exploitation.

`type` — indique si une commande est interne ou externe.

`echo` — affiche du texte.

`man commande` — donne le manuel relatif à `commande`.

`date` — indique la date sous un format à choisir ou effectue des conversions de temps.

`kill` — fait cesser un processus en cours.

`cd` — se positionne dans le répertoire choisi.

Les autres commandes sont dites *externes*, les lignes de code qui les définissent sont dans `/bin`. En fait, il n'y a pas que `/bin` : un petit ensemble de répertoires est en fait consulté lors de la recherche du programme permettant l'exécution de la commande. Ces répertoires sont définis dans une variable d'environnement appelée `$PATH`. Si la commande n'est pas trouvée dans l'un des répertoires listés dans `$PATH`, l'exécution se solde par un message d'erreur du type `command not found`.

2.2. Commandes liées à du texte

`cat fichier.txt` — affiche le contenu de `fichier.txt`.

`rev fichier.txt` — affiche `fichier.txt` en ordre inversé.

`wc -l` (words count) — introduit une sous-ligne de commande : on tape des choses puis CTRL D pour sortir, et ça compte le nombre de lignes des choses qu'on a tapées.

`tac` — renvoie un texte en inversant les lignes. Subtilité, pour modifier le séparateur (par défaut le saut de ligne), mettre en option `--separator=";"` ou `-s ";"` par exemple si l'on choisit le point-virgule.

2.3. Divers

`source fichier.sh` — exécute le script `fichier.sh`.

`sleep temps` — interrompt tout pendant le temps indiqué.

`time commande` — effectue la commande puis affiche le temps utilisé.

`date` — la commande `date +%s` donne un *timestamp unix*, i.e. la date sous format de nombre (secondes depuis 01/01/1970), tandis que `date -d @1000214040` convertit le *timestamp* en date usuelle.

`bash` — qui curieusement n'est pas interne ! crée un « sous-bash »

2.4. Empreintes

`echo "jkljkl" | md5sum` donne l'empreinte de l'argument "jkljkl", est utilisé dans ce MOOC pour vérifier si l'on a tapé la bonne instruction, sans être obligé de donner la réponse.

2.5. Alias

Exemple: `lsd='ls -l'` permet d'appeler `ls -l` en tapant seulement `lsd`.

3. Trouver de l'aide

`man abcd` ou `abcd --help` ou `abcd -h` donne l'aide relative à la commande `abcd`.
Lorsqu'on fait, sciemment ou pas, un erreur d'optio : `abcd -jkdsqljkdl`, l'aide s'affiche aussi.
`apropos encodage` renvoie la liste des instructions dont l'aide contient le mot `encodage`.

aide générale : `man man`, on se déplace page à page avec ctrl F ou F (foreward) et ctrl B ou B (backward).

Les aides fournies par `man` sont stockées dans `/usr/share/man/`. Elles sont écrites dans un format particulier qui contient des macros `roff` permettant une mise en forme basique.

Sous Linux, `/usr/share/doc/commande` contient aussi bien souvent de la documentation, souvent sous la forme d'un simple fichier texte README, parfois en pdf ou en html.

Il y a aussi le site http://man7.org/linux/man-pages/ qui donne tous les `man` en ligne.

4. Raccourcis clavier et gestion du Shell

4.1. Raccourcis clavier

- `ctrl A` début de ligne ;
- `ctrl E` fin de ligne ;
- `ctrl U` copie la ligne actuelle depuis le curseur jusqu'au début ;
- `ctrl K` copie la ligne actuelle depuis le curseur jusqu'à la fin ;
- `ctrl Y` pour coller ;
- ctrl L ou `clear` pour effacer la fenêtre du terminal ;
- alt supprimer À gauche = supprimer le mot qui précède le curseur ;
- alt D = supprimer le mot qui suit le curseur ;
- ctrl R pour rechercher dans l'historique ;
- autocomplétion : undébutdemot suivi de ⇥ ou undébutdemot suivi de ⇥⇥ s'il y a ambigüité.

4.2. Manipulation de la fenêtre du shell

`clear` efface l'écran.
`exit` ou ctrl D ferme la session.

4.3. Gestion de l'historique

Commandes passées

`history` affiche la liste numérotée des commandes déjà saisies.
`history -c` permet d'effacer l'historique.
`history -w fichier.txt` permet de sauver l'historique courant dans `fichier.txt`.
`history -r fichier.txt` permet d'ajouter les commandes contenues dans `fichier.txt` à l'historique courant.

Raccourcis

`!!` reprend la dernière commande, équivalent à un appui sur la flèche vers le haut, mais pratique pour faire par exemple `sudo !!`.

Exemple, taper `cd`↵ puis `!!`↵ puis la flèche vers le haut : cela permet de bien comprendre.

`!hi`↵ permet de rappeler la dernière commande commençant par `hi` (par exemple ça peut être `history`).

Pour vérifier quelle était cette commande mais sans l'exécuter faire `!hi:p`.

`!2` rappelle la commande numérotée 2 dans l'historique.

`!-4` pour rappeller la 4e commande en partant de la fin dans l'historique.

`!?to` permet de rappeler la dernière commande contenant ce nom (par exemple ça peut être `history`)

Arguments passés

`!*` rappelle tous les arguments de la dernière commande.

`!^` rappelle uniquement le premier argument de la dernière commande, et :

`!$` uniquement le dernier.

`!!:5` rappelle le 5ème argument et `!!:2-4` tous les arguments du 2e au 4e.

Substitutions

`!!:s/1/2/` rappelle la dernière commande mais en remplaçant le premier 1 éventuel par un 2.

Exemple `cat fichier1.txt`↵ puis `!!:s/1/2/` va renvoyer `cat fichier2.txt`.

`!!:gs/1/2/` rappelle la dernière commande mais en remplaçant tous les 1 éventuels par des 2.

Par exemple si vous souhaitez exécuter `wc fichier2.txt` dupliquer `fichier2.txt` en `fichier3.txt`, vous pouvez taper `wc fichier2.txt` puis `cp !^ !^:s/2/3/`.

5. Arborescence

5.1. Les dossiers spéciaux

À la racine du disque dur, il y a :

- `bin` : contient les commandes ;
- `etc` ;
- `home` (sur mac os X c'est "users") ;
- `temp` : vidé à chaque redémarrage ;
- `usr`.

Attention, `/usr` est le dossier `usr` à la racine, tandis que `usr` n'est qu'un nom relatif désignant un éventuel dossier `usr` dans le dossier courant.

5.2. Commandes pour naviguer dans l'arborescence

`pwd` retourne le chemin absolu du répertoire courant ;

`ls` liste les élements du dossier courant, et avec `ls -p` les dossiers sont suivis d'un `/` pour mieux les distinguer des fichiers ;

`cd` tout seul ramène dans ∼ ;

`cd` ∼ amène dans le répertoire personnel ;

`cd ..` remonte d'un cran dans l'arborescence ;

`cd ../..` remonte de deux crans

`mkdir toto` crée le dossier `toto` dans le dossier courant ;

`touch k.txt` crée le fichier `k.txt` ;

`rm k.txt` supprime le fichier `k.txt` ;

`rmdir toto` supprime le dossier `toto` ;

`cp nomactuel nomdufichierdupliqué` copie le dossier ou le fichier ;

`mv nom_ou_cheminactuel nomoucheminchoisi` déplace un dossier ou un fichier et permet soit de renommer soit de déplacer ;

6. Utilisateurs et droits

6.1. Les utilisateurs

`id` donne des infos compliquées sur l'utilisateur

Il y a un `root` et des `users`.

Chaque fichier possède des droits attribués à trois types d'utilisateurs, qui sont :

- le propriétaire (appelé `u`) ;
- les membres (appelés `g`) du même groupe que le propriétaire ;
- et les autres (appelés `o`) ;
- pour raccourcir, on peut utiliser la lettre `a` qui signifie `ugo`.

On appelle GID le numéro d'identification du groupe principal de l'utilisateur.

6.2. Les droits

Les trois droits sont :

- `r` (lire) ;
- `w` (écrire) ;
- et `x` (s'y positionner si c'est un répertoire, le lancer si c'est un programme).

Exemples de situations :

- un technicien qui m'installe un logiciel et qui ne veut pas que j'aie accès au code source peut mettre dans mon ~ un exécutable en restant propriétaire, et en ne donnant, au groupe et aux « autres », que le droit `x` (on l'exécute mais on ne le lit pas) ;
- `w` permet de supprimer un fichier ; attention si on a le droit `w` sur un dossier on peut supprimer ce dossier même si on n'a pas les droits `w` sur les éléments individuels du dossier.

Pour lister les droits des éléments d'un dossier, faire `ls -l`. On obtient alors :

- pour chaque fichier, une information du type `-rwxrwxrwx` qui doit se lire comme :
  - `-` (c'est un fichier). S'il y a un `d` à la place du `-`, c'est un dossier.
  - (rwx)(rwx)(rwx) (droits pour `u,g,o`) ;
- `drwxrwxrwx` pour les dossiers (le `d` signifiant *directory*, soit « répertoire »).

6.3. Pour changer les droits

`chmod g+w,o-rx nomdudossier` signifie « on ajoute le droit `w` au groupe (`g`) et on enlève les droits `r` et `x` au autres (`o`) ».

exemple `chmod o-rwx nomdufichier` pour retirer tout droit aux "autres".

`chmod a+x fichier` donne à tout le monde le droit d'éxécuter le fichier.

Changements groupés

On utilise les chiffres de 1 à 8, avec : 4 = `r`, 2 = `w`, 1 = `x`.

Ainsi, `chmod 754 nomdufichier` donnera :

- le droit 7 = 4 + 2 + 1 = `rwx` au propriétaire ;
- le droit 5 = 4 + 1 = `rx` au groupe ;
- le droit 4 = `r` aux autres.

Pour voir les droits de chaque utilisateur, lire le fichier `passwd` dans le dossier `/etc`.

Résumé

- Le `sudo` peut faire `chown` (changer le propriétaire), `chgrp` (changer le grope), `chmod` (changer les droits) ;
- le propriétaire peut faire `chmod` et `chgrp` s'il appartient au groupe ayant les droits sur le dit fichier ; il ne peut pas faire `chown`.

7. FICHIERS DE TEXTE

7.1. Généralités

Noms à respecter : a-z A-Z 0-9 ainsi que les deux traits d'union - et _, cependant jamais le - au début du nom.

`cat fichier.txt` pour afficher dans la console le contenu d'un petit `fichier.txt`.

`cat -b fichier.txt` la même chose mais avec les lignes numérotées à l'affichage.

`cat fichier1 fichier2` affiche tout à la suite, pratique pour concaténer par `cat fichier1 fichier2 > fichiertotal`.

`head` et `tail` pour les 10 premières (dernières) lignes du fichier.

`head -20` et `tail -20` pour les 20 premières (dernières) lignes du fichier.

si `fichier1` contient

```
a
b
c
```

et `fichier2` contient

```
d
e
f
```

alors `paste -d : fichier1 fichier2` contiendra

```
a:d
b:e
c:f
```

`sort` permet d'ordonner les lignes alphabétiquement, exemple `cat fichier | sort -nk 2` trie numériquement suivant la colonne 2 et `cat fichier | sort -nk 2 -kr 3` trie, en second paramètre, suivant la colonne 3 mais à l'envers. L'option `-t` permet de spécifier le séparateur de colonne.

`-k` veut dire « key » et désigne la colonne à choisir.

`split -l 30 fichier kf1` sépare le fichier en fichiers `kf1aa,kf1ab,...` on remplace `kf1` par ce que l'on veut.

La commande suivante effectue la meˆme division mais en se´parant en fichiers ayant le meˆme nombre de lignes, dans notre cas 30, avec l'option -l 30.

`tail -f` affiche en temps réel les dernières lignes qui s'ajoutent petit à petit si le fichier est modifié extérieurement ; on quitte en faisant ctrl C.

`tr "[A-Z]" "[a-z]" < texte.txt` transforme les majuscules en minuscules tandis que `tr "AR" "ar" < texte.txt` minusculise seulement les A et les R.

`wc -l` donne le nombre de lignes du fichier.

7.2. La commande `less`

Pour des fichiers plus longs, on utilisera : `less fichier.txt` ou `less -N fichier.txt`.

`less -S` tronque les lignes qui dépassent la largeur de l'écran.

Une fois cette commande lancée :

- on peut utiliser les flèches pour se déplacer ;

- ou taper un chiffre comme 10 puis la flèche haut ou bas pour se déplacer d'autant de lignes ;
- ou taper `/lo` pour afficher la première ligne contenant `lo` entre la position courante et la fin ;
- ou taper `?lo` pour la première ligne contenant `lo` entre la position courante et le début ;
- ou taper `&lo` pour toutes les lignes contenant `lo` ;
- ou F (forward) ou B (back) pour avancer (ou reculer) page par page ;
- ou G pour amener le curseur à la fin du fichier ;
- ou 12G pour se rendre à la 12e ligne.

7.3. Le programme `vi`

`vi fichier.txt`, puis on navigue comme avec `less`.

On tape I (la touche « i ») pour insérer et esc quand on a fini, ou A pour insérer après le curseur.

Résumé : `vi fichier.txt`, touche I (touche « i »), taper ce que l'on veut, esc puis `:wq`↵ pour revenir à la ligne de commande.

On tape `dNd` pour supprimer n lignes à partir de la ligne courante. Si vous ne spécifiez pas de valeur pour n, une seule ligne sera supprimée.

Raccourcis spécifiques à `i` :
0 placer le curseur en début de ligne
$ placer le curseur en fin de ligne
W placer le curseur sur le mot suivant ; peut être préfixée par n
B placer le curseur sur mot précédent ; peut être préfixée par n
G placer le curseur à la fin du fichier.
12G placer le curseur sur la n-ième ligne du fichier
ctrl F (forward) placer le curseur sur la page suivante
ctrl B (backward) placer le curseur sur la page précédente
. répéter la commande précédente
U annuler la commande précédente
⇧J fusionner une ligne et la ligne suivante sur une même ligne. Cette commande peut être préfixé par n.

`/motif` Rechercher et placer le curseur sur la première (ou n-ième) occurrence de "motif" entre la position courante et la fin du fichier.
`:%s/ancienmotif/nouveaumotif/g` fait rechercher remplacer partout
`:s/ancienmotif/nouveaumotif/g` fait rechercher remplacer une fois.

Instructions particulières :
`:h` aide
`:q` quitter l'aide
`:w` sauver
`:q` quitter vi
`:wq` sauver puis quitter
`:x` ou `:q!` quitter sans enregistrer.
`:set number` préfixer chaque ligne par son numéro.

Copier-coller
Lorsque vous supprimez du texte, celui-ci est copié dans un presse papier. Vous pouvez coller le contenu de ce presse papier grâce à la commande P . Le contenu du presse papier sera inséré après le curseur.

`yw` Copier un mot. Peut être préfixée par n.

`yNy` Copier n lignes à partir de la ligne courante. Si vous ne spécifiez pas de valeur de n, une seule ligne sera copiée.

Il y a 26 presse-papiers nommés de `a` à `z`. L'enchaînement des touches `"` `F` `Y` `3` `Y` permet de copier 3 lignes vers `Y`.

8. SUBSTITUTIONS = GLOBBING

8.1. Globbing vs expressions rationnelles

Le *globbing* c'est lorsque l'on fait référence à des fichiers :

`*`	un ou plusieurs caractères
`?`	un caractère exactement
`[...]`	un caractère dans la liste (peut être un intervalle)
`[!...]`	aucun caractère de la liste

Par exemple `[A-Z]?2*.pdf` produit la liste des fichiers (et répertoires) dont le nom commence par une majuscule, suivi d'un caractère, suivi d'un 2, suivi ce que l'on veut, suivi d'un point et suivi de pdf.

Les *expressions régulières*, elles, sont utilisées par les utilitaires comme `grep`, `sed`, `vi` ou `awk` pour faire référence à des chaînes à l'intérieur d'un fichier texte. La syntaxe est beaucoup plus riche que celle du globbing. Pour compliquer la chose, il existe plusieurs sortes d'expressions régulières :

- les basiques (BRE) qui sont la bases de toutes les autres ;
- les étendues (ERE) qui sont enrichies, utilisées par exemple par grep -E (aka egrep) ;
- celles qui proviennent de Perl (PCRE) et que l'on retrouve dans la plupart des langages de programmation.

Les *expressions rationnelles* sont évoquées dans le 13.6 avec la commande `sed`.

8.2. Répertoires usuels

`.` représente le répertoire courant, `..` le répertoire parent, ∼ le répertoire *home* de l'utilisateur.

Si je m'appelle Alice et si un autre utilisateur de cet ordinateur s'appelle Bob, alors on a aussi le raccourci ∼`bob` équivalent à ∼`/../Bob` c'est-à-dire le répertoire *home* de Bob.

8.3. Caractères de substitution ? et *

`ls ????` liste les fichiers et dossiers contenant quatre caractères, c'est-à-dire les fichiers nommés par exemple `a.txt` ou les dossiers nommés `abcd`, tandis que `ls a?????` liste les fichiers dont le nom est formé d'un `a` suivi d'exactement cinq caractères, exemple `ab.doc`.

`ls a*` liste les fichiers dont le nom commence par « a », peu importe le nombre de caractères que leur nom comporte.

`ls *.png` affiche tous les fichiers ayant l'extension png.

`ls *.*` liste tous les fichiers (dont le nom possède une extension), tandis que `ls .*` liste tous les fichiers invisibles, et `ls * .*` (attention à l'espace) liste tous les fichiers visibles puis invisibles.

Attention, `cd /bin ; ls *.*` renvoie une liste vide car le dossier un peu spécial ne contient que des fichiers sans extensions (les commandes externes du bash).

`ls nomdundossier` liste le contenu du dossier choisi, équivalent à `cd nomdundossier ; ls`.

`ls *` affiche la liste de tous les dossiers et de leur contenu. En effet, cette instruction est interprétée par le shell comme « pour tous les noms * (donc tous les noms) de dossier, fais ls nomdudossier ». De même, `ls /*` liste les dossiers de la racine et leur contenu, et `ls ∼/*` fait de même pour les dossiers à la racine du *home* de l'utilisateur.

Partout, on peut aussi remplacer `ls` par `echo` et cela donnera une liste non tabulée.

8.4. Caractères de substitution avec [...]

`ls [aef]*` liste les fichiers et dossiers dont le nom comment par a,e, ou f.
`ls [!A-Z]*` affiche tous les fichiers qui ne commencent pas par une lettre majuscule.

On peut aussi utiliser les instructions suivantes :

instruction	équivalent	explication
`[:alnum:]`	`[A-Za-z0-9]`	caractères alphabétiques et numériques
`[:alpha:]`	`[A-Za-z]`	lettres
`[:blank:]`		espace ou tabulation
`[:cntrl:]`		caractères de contrôle
`[:digit:]`	`[0-9]`	chiffres
`[:graph:]`		caractères compris entre $33 \leqslant \text{ASCII} \leqslant 126$ =caractères graphiques affichables hormis l'espace
`[:lower:]`	`[a-z]`	lettres minuscules
`[:print:]`		caractères compris entre $32 \leqslant \text{ASCII} \leqslant 126$ = [:graph:] avec l'espace en plus.
`[:space:]`		blancs (espace, tabulation, passage à la ligne, retour chariot, saut de page, tabulation verticale)
`[:upper:]`	`[A-Z]`	lettres majuscules
`[:xdigit:]`	`[A-Fa-f0-9]`	chiffres hexadécimaux

8.5. Extglob

Taper `shopt -s extglob` pour avoir accès à ces instructions.

`A?(a|b)` désigne les noms de fichier `A`, `Aa`, `Ab` : aucun des motifs ou bien un seul des motifs et une seule fois

`A*(a|b)` désigne les noms de fichier `A`, `Aa`, `Aaa`, `Aaaa`, `Ab`, etc : aucun des motifs ou bien un seul des motifs autant de fois qu'on veut.

`A+(a|b)` désigne les noms de fichier `Aa`, `Aaa`, `Aaaa`, `Ab`, etc : un seul des motifs autant de fois qu'on veut.

`A?(a|b)` désigne les noms de fichier `Aa`, `Ab` : un seul des motifs et une seule fois

`A!(a|b)` désigne tout sauf `A?(a|b)`.

Exemple : « à l'aide de la commande echo, vous devez afficher uniquement les noms des images commençant par "ab" ou bien par "xyz" et dont l'extension n'est ni ".jpg", ni ".gif" ».

Réponse : `echo +(ab|xyz)!(*.gif|*.jpg)`.

9. VARIABLES

9.1. Généralités

9.1.1. Utilisation courante

On tape `i=2` puis dans une instruction quelconque on peut appeler la variable `i` par `$i`.

Exemple : `echo $i` renvoie la valeur de `i`.

Attention, pas d'espaces ! `a = 1` ne marchera pas !!!!!!!!!

9.1.2. Utilisation avec une commande

`$(pwd)` renvoie la valeur qu'aurait `pwd` si on le faisait. Pratique pour taper par exemple `chemin=$(pwd)`. De même, `mot=$(cat fichier)`.

9.2. Variables, inhibiteurs et caractères de substitution

Le `''` inhibe tous les caractères de substitution, en particulier les `$`.

Ainsi, `echo '$(pwd)'` renvoie juste la chaîne `$(pwd)` telle quelle : on dit que le `''` a *inhibé* le `$`.

Si `a=*`, alors :

- `echo $a` équivaut à `echo *` donc renvoie la liste des fichiers du répertoire courant ;

- `echo '$a'` renvoie `$a` tel quel ;
- `echo "$a"` renvoie `*` : le `""` inhibe le `*` mais pas le `$`.
- `echo \$a` renvoie `$a` car le `\` inhibe seulement le caractère qui suit, quel qu'il soit, donc ici en l'occurence le `$`.

Si $a =$ "toto", pour afficher « tototo », `echo $ato` ne marche pas car il cherche une variable de nom ato, on utilise alors les accolades : `echo ${a}to`.

Divers exemples :

```
AGE=45 ; NOM=Henri ; echo "Monsieur $NOM a $AGE ans"
AGE=45 ; NOM=Henri ; PHRASE="$NOMa$AGEans" ; echo $PHRASE
```

renvoie une chaîne vide car `$NOMa` et `$AGEans` sont interprétées comme des variables inconnues, donc vides. C'est la preuve que le langage Bash ne nécessite pas de déclarer les variables.

`AGE=45 ; NOM=Henri ; PHRASE="$NOM a $AGE ans" ; echo $PHRASE` marche bien.

9.2.1. Inhibition de l'espace et de la fin de ligne

- Le `\` est pratique pour imposer plusieurs espaces : `echo b\ \ \ \ \ a` renvoie « b a » séparés de 5 espaces alors que `echo b a` renvoie « b a » : le bash, par défaut, ignore les espaces multiples.
- L'inhibition de fin de ligne est utile :

  ```
  echo \
  ab
  ```

  équivaut à `echo ab`, pratique pour clarifier de longues instructions.

9.2.2. Caractères spéciaux

`\a` bip
`\b` espacement arrière
`\e` échappement
`\f` saut de page (le nom anglais de ce caractère est `form feed`)
`\n` saut de ligne
`\r` retour chariot
la différence entre `\r` *et* `\n` *est une histoire d'encodages et d'OS.*
`\t` tabulation
`\v` tabulation verticale
`\\` anti-slash (inhibition du `\`)
`\'` apostrophe (inhibition du `'`)
`234` le caractère 8 bits dont la valeur en octal est 234
`\x5F` le caractère 8 bits dont la valeur en hexadécimal est HH
`\cx` le caractère contrôle-X.

9.2.3. Attention aux noms de variables longs :

`$ab_` cherche une variable qui s'appellerait `ab_`. De même, `echo $a1` peut être ambigü pour l'utilisateur car on ne sait pas si c'est la variable `a1` (ce que bash va interpréter en fait), ou la variable `a` suivie du chiffre `1`. Pour lever ces ambigüités, on peut faire `echo ${a1}` ou `echo ${a1}` suivant ce que l'on désire.

9.2.4. Variable contenant une commande

si `p=ls`, alors `$p1` va juste renvoyer la chaîne `ls`.

`$ eval \${$p1}` va par contre exécuter la commande `ls` : ici, le bash travaille en deux temps :

- d'abord, il interprète`\${$p1}` qui donne `ls` ;
- ensuite, il actionne `ls`.

Il semble qu'il y ait quelque chose à comprendre sur cette discussion.

`echo "x vaut $x"` donne ce qu'on attend tandis que

`echo 'x vaut $x'` renvoie le `$s` tel quel.

10. Processus et leur déroulement

10.1. Parents, enfants

Un processus, c'est une commande en train de s'éxécuter.

Un processus est toujours le fils d'un autre processus, exemple lorsqu'on tape `ls` dans le bash, le processus parent « bash » lance le processus enfant « ls ». Lorsque ce dernier est terminé, on revient au parent (l'invite du bash donc).

10.2. Historique des processus

`echo $?` après un processus renvoie 0 s'il s'est déroulé correctement et 1 ou un autre entier non nul s'il y a eu une erreur : on appelle ce chiffre le *code retour*.

`ps -f` liste les processus exécutés précedemment et affiche leurs caractéristiques :

UID (propriétaire), le PID (processus fils), le PPID (processus parent), les CMD (commande, arguments...).

10.3. Processus en parallèle, arrière-plan et premier plan

On peut obliger un processus à se faire en arrière-plan et ainsi exécuter plusieurs processus en parallèle par `commande &`. Pour voir alors la liste des processus en train de courir, on tape `jobs`.

cela ne correspond pas à ce que donne le *moniteur d'activité*

Chaque processus en cours est numéroté par un entier ($n = 1,2,..$) appelé *numéro de tâche*, et aussi par un plus grand nombre appelé *numéro de processus*.

En tapant `fg %n`, on force le processus numéroté n à revenir au premier plan.

Inversement, `CTRL Z` interrompt le processus qui est au premier plan, alors en tapant `bg` on relance ce processus tout en l'envoyant à l'arrière-plan.

Pour clore définitivement un processus qui court en arrière-plan ou qui est interrompu, on fait `kill+numéro_du_processus`.

11. Entrées, Sorties

11.1. Principe général

Par défaut les commandes du shell bash prennent en entrée ce que l'on tape au clavier sur la ligne de commande, cela est appelé *entrée standard*, ou *entrée 0* ou *stdin*.

Par défaut, elle envoie en sortie sur l'écran, par un affichage sur les lignes qui suivent la ligne de commande où l'on a écrit les instructions, cela est appelé *sortie standard* ou *sortie 1* ou *stdout*.

Enfin, mais ça se voit moins quand on débute, il y a une sortie appelée *sortie erreurs* ou *sortie 2* ou *stderr* où vont les messages d'erreur. Dans la pratique usuelle, cette sortie paraît être au même endroit que la sortie 1 (sur l'écran, dans les lignes suivant la ligne de commande).

11.2. Détournement de la sortie 1

`ls > fichier.txt` va envoyer le résultat de `ls` non plus sur l'écran mais dans un fichier `fichier.txt` existant ou pas;

De même, `ls >> fichier.txt` va rajouter le résultat de `ls` à la fin de `fichier.txt` (ou le créer s'il n'existe pas encore).

Parfois on a besoin juste d'ignorer le résultat d'une commande alors on l'envoie dans le trou noir : `ls > /dev/null`

On peut s'amuser (ce qui *a priori* ne sert à rien) à détourner plusieurs fois la sortie 1 :

`echo "to" > ici.txt > non_la.txt > ohbenfinalementici.txt`

ce code crée deux fichiers vides `ici.txt` et `nonla.txt`, et un fichier `ohbenfinalementici.txt` qui contient la chaîne `to`.

Attention, on ne peut pas rediriger vers un fichier une commande concernant un fichier. Exemple, `cat truc.txt>truc.txt` va d'abord ouvrir `truc.txt` et le vider pour se préparer à y mettre le résultat, puis faire `cat truc.txt` qui renverra donc l'ensemble vide. Ainsi, `cat truc.txt>truc.txt` revient à vider `truc.txt` ce qui est donc équivalent à `>truc.txt`

11.3. Détournement de la sortie 2 (sortie des erreurs)

On peut avoir envie d'ignorer les messages d'erreurs alors on les redirige vers le trou noir en tapant `commande arguments 2> /dev/null`.

Autre situation : le `man time` indique `time writes a message to standard error` ce qui veut dire que le résultat de `time` part sur `stderr`. Logique, on fait `time commande truc` et le résultat de `commande` part sur `stdout`, alors il faut bien que le résultat de `time` parte ailleurs, il ne reste que `stderr`. Ainsi on peut faire `time ls > fichier.txt 2> temps.txt`.

11.4. Détournement vers la sortie 2

`echo "erreur !" >&2`

voir dans le chapitre « tests »

11.5. Détournement de l'entrée 0

`wc -l < unfichier.txt` affiche le nombre de lignes du fichier (prend le fichier comme entrée à la place de `stdin`). Presque équivalent à `wc -l unfichier.txt`.

`tac` prend en entrée soit l'argument qui suit soit, (si l'argument est `-` ou s'il n'y a aucun argument) l'entrée `stdin`. Ainsi, pour que `tac` agisse sur `truc` puis sur `machin` on peut faire au choix :

- `tac truc machin`
- `tac truc ; tac machin`
- `echo truc | tac - machin`
- `echo machin | tac truc -`

11.6. Pipes

`history | wc -l` envoie le résultat de `history` en entrée au processus `wc`. Cela revient au même que `history > fichier.txt; wc -l < fichier.txt` qui cependant nécessiterait de créer `fichier.txt`.

11.7. Divers à creuser

Apparemment, on peut détourner les deux sorties en même temps par quelque chose comme :

`time ls 1>ici_ls.txt 2>&1` mais sur mon terminal ça ne marche pas.

Apparemment, les sorties sont répertoriées là : `/dev/fd` mais n'y sont pas, il y a quelque chose à comprendre avec *wl* et avec `lsof` : *list open files*, et avec `ls -l /proc/self/fd`.

Compiler les résultats du devoir de la séquence 2.

12. VARIABLES D'ENVIRONNEMENT

12.1. Généralités

12.1.1. Liste des variables

L'instruction `printenv` permet de lister les variables réservées dites variables *d'environnement*. Elle est équivalente à `env`.

`set` permet de lister aussi les variables créées par l'utilisateur.
`declare` aussi mais en différenciant variables exportées et variables locales.

12.1.2. Exemple de variables d'environnement

`EDITOR=/usr/bin/vi` définit l'éditeur de texte par défaut de l'utilisateur.
`HOME` définit le répertoire par défaut donné par `cd` ainsi que celui donné par ∼.
Remarque : la variable ∼ est une variable comme une autre sauf qu'elle s'appelle par ∼ sans `$` devant. Elle contient le répertoire par défaut (i.e. le contenu de `HOME`).
Attention, `HOME=..` donnerait un `cd` différent à chaque fois !
`LOGNAME` est le nom de l'utilisateur à la connexion.
`MANPAGER` contient (par défaut `less -R`) la commande que `man` utilisera pour afficher les manuels.
`PATH` désigne les répertoires (comme `/usr/bin`) où le shell va chercher la définition des commandes externes qu'on lui demande d'exécuter. En général, `/usr/local/bin` contient les commandes personnelles et `usr/bin` contient les commandes externes fournies par défaut. L'ordre compte : si plusieurs commandes `truc` sont dans plusieurs dossiers différents, le premier rencontré dans par le `PATH` l'emporte.
voir ici https://fr.wikipedia.org/wiki/Variable_d%27environnement#Sous_Mac_OS_X
`RANDOM` renvoie, à chaque consultation de sa valeur, un nombre différent aléatoire compris entre 0 et 32767.
`SHELL` désigne le shell utilisé par défaut.

12.1.3. Variables liées à l'appareil ou à ses utilisateurs

Les fichiers suivants sont exécutés dans cet ordre :

- `/etc/profile` s'exécute à chaque démarrage et contient les variables d'environnement communes à tous les utilisateurs et à tous les shells ;
- `/etc/bashrc` s'exécute aussi à chaque démarrage et contient les variables d'environnement communes à tous les utilisateurs mais pour le shell bash uniquement.
  Sous mac os X, il contient la valeur de `PS1`, voir 12.3.1 ci-dessous.
  où est le fichier qui contient $PATH ?
- ∼`/.profile` s'éxécute à l'ouverture de session et charge les variables liées à l'utilisateur (ou modifie celles déjà chargées par les deux autres).
  N'existe pas sous mac os X.
- ∼`/.bashrc` semble fait pour exécuter `/etc/bashrc` ?
  N'existe pas sous mac os X.

12.2. Modifications des variables d'environnement

12.2.1. Modifier pour une seule ligne

- `HOME=∼/toto cd`
- `MANPAGER=cat man date`

Le principe est donc :
`VARIABLE=truc instruction_utilisant_VARIABLE`
ou
`env VARIABLE=truc instruction_utilisant_VARIABLE`

12.2.2. Modifier pour toute la session

- La modification donnée par `$HOME=truc` est valable pendant toute la session.
- `MANPAGER=cat ; man date`
  Le pointillé rend la modification valable pour toute la session.

12.2.3. Modification permanente

La liste des variables d'environnement et des variables exportées est stockée dans les fichiers `/etc/.bashrc` (spécifique à bash) et `~/.profile` (plus général).

On édite l'un ou l'autre et on rajoute les deux lignes :
```
ici=$HOME/ici
export ici
```
puis on exécute le fichier par `source .bashrc`
Pour pouvoir modifier de façon plus lisible, on va dans `~/.profile` et on y tape :
```
variable=truc
export variable
```

12.2.4. `tmux`

Permet de lancer un sous-shell avec la possibilité de contrôler sa position (arrière-plan, avant-plan). Nécessite d'être installée au préalable. Utile pour modifier des variables d'environnement de manière « locale ».

12.3. La variable `PS1`

12.3.1. Principe

`PS1` définit le prompt, exemple `PS1="\u dans \w >"` où `\u` désigne le nom de l'utilisateur et `\w` désigne le répertoire courant. On peut aussi utiliser `\t` qui donne le temps.

On la modifie par `PS1="ce qu'on veut"`, et la modification durera le temps de la session.

12.3.2. Caractères de substitution

On peut puiser, entre autres, parmi les variables suivantes :

`\u` : login de l'utilisateur courant
`\w` : répertoire courant, avec $HOME abrégé par un tilde
`\t` : Heure courante au format 24-heures HH:MM:SS
`\s` : Le nom du shell (aussi retourné par la variable $0)
`\A` : Heure courante au format in 24-heures HH:MM
`\H` : Nom de la machine
`\W` : Nom de base du répertoire courant, avec $HOME abrégé par un tilde
`\!` : Le numéro historique de la commande
`\#` : Le Numéro de la commande
`\n` : Passage à la ligne

12.3.3. Modification durable sous mac os X

Il faut modifier `/etc/bashrc` :

- par défaut il a les droits `-rwxr-xr-x=755`, lui faire donc `sudo chmod 777 bashrc` ;
- faire `vi bashrc` et modifier la valeur de `PS1` ;
- faire `source bashrc` pour rendre le changement durable ;
- remettre les droits d'avant on ne sait jamais : `sudo chmod 755 bashrc`.
- Voir 12.2.3 ci-dessous si ce n'est pas contradictoire.

12.4. Variables perso

On peut créer des variables à partir des variables d'environnement. Exemple : `ici=$HOME/undossier`.

Les variables d'environnement sont accessibles depuis les sous-shells (quand on tape `bash`), mais pas les variables perso, sauf à les exporter par `export variable`.

Les variables ainsi exportées apparaissent elles aussi dans le `printenv`.

`unset variable` vide le contenu de `variable`.

12.5. Options du shell

`shopt` affiche les options en cours (suivies chacune d'un `off` ou d'un `on`). On modifie chaque option par `shopt -s optionchoisie` pour activer ou `shopt -u optionchoisie` pour désactiver.

13. Filtres puissants

13.1. Les commandes natives du Bash

Voir ici https://man.developpez.com/man1/bash/#L14.3.

13.2. La commande `cut`

`cut -f numero -d "separateur" fichier` permet de prendre le champ « numéro » de chaque ligne du fichier avec le séparateur choisi.

Exemples :

`ls 2018,02* | cut -f 1 -d "." > dates.txt`

`ls 2018,02* | cut -f1 -d. > dates.txt`

permettent de lister tous les fichiers dont le nom commence par `2018,02` et de placer la liste de ces fichiers sans leur extension dans le fichier `dates.txt`.

`cut -f 1 -d " " .bash_history`

permet de lister seulement les commandes du fichier d'historique.

`cat .bash_history | cut -f 1 -d " " | sort`

donne toutes les instructions par ordre alphabétique, tandis que :

`cat .bash_history | cut -f 1 -d " " | sort | uniq -c`

donne la même chose sans les doublons.

`cat .bash_history | cut -f 1 -d " " | sort | uniq -c | sort -r | head`

pour afficher les dix instructions les plus utilisées !

`cut -d "," -f2,5` donne les colonnes 2 à 5 tandis que `cut -c2,5` donne les caractères 2 à 5. Attention, le séparateur par défaut est le tab.

`echo "a.b" | cut -f1 -d.` renvoie `a`

`echo "a.b.c.d" | cut -f2- -d.` renvoie `b.c.d` (tous les champs à partir du 2nd).

13.3. La commande `find`

`find . -iname "*.mkv" -size +1G -mtime -7 -o iname "*.epub" -user alice -mtime -7`

permet de trouver dans `.` et dans les sous-dossiers de `.` :

- les fichiers .mkv de taille >1Go et dont la date de dernière modification est $\leqslant$ 7 jours ; le `-iname` désensibilise à la casse, alors que `-name` y est sensible
- ou les .epub dont je (=Alice) suis le propriétaire.

`find . -iname "*.mkv" -size +1G -mtime -7 -exec mv {} /tmp/. ';'`

permet de déplacer les premiers vers le répertoire temporaire `/tmp`

Options courantes :

- `-type d` liste seulement les répertoires ;
- `find /ici /la "truc"` cherche dans les deux répertoires indiqués ;
- `-mtime -7` signifie « modifiés il y a moins de 7 jours » tandis que `-mtime +7` signifie « modifiés il y a plus de 7 jours » ;
- `find $HOME "truc"` est équivalent à `find ~ "truc"` ;
- `find $HOME "truc" ! -user $LOGNAME` liste dans ~ les trucs dont je ne suis pas le propriétaire ;
- `find . -name ‘‘*.py’’ -o -name ‘‘*.cpp’’` : ici le `-o` veut dire « ou » ;

- `find . -iname "*.txt" -exec wc -l {} ';' -exec cp {} {}.bak ';'`
  trouve dans `.` les fichiers .txt et les affiche un par un précédés de leur nombre de lignes ; ensuite les duplique dans un fichier qui sera du coup un .txt.bak
- `-maxdepth` auquel on donne l'argument d'option 1 pour que `find` ne recherche que dans le répertoire courant

13.4. La commande `grep`

`grep truc fichier.txt`
affiche toutes les lignes de fichier.txt contenant le mot « truc ».
`cat fichier1.txt fichier2.txt | grep motif` idem avec deux fichiers
`grep -v truc fichier.txt` ici le `-v` joue le rôle qu'on attendrait d'un `!`.
`grep -n truc fichier.txt` affiche les lignes numérotées
`grep -c truc fichier.txt` affiche seulement le nombre de lignes trouvées
`grep ^truc fichier.txt` affiche seulement les lignes commençant par « truc »
`grep truc$ fichier.txt` affiche seulement les lignes finissant par « truc »
`grep '^[KU]' fichier.txt` les lignes commençant par K ou U (il faut mettre les '') ;
`-i` désensibilise à la casse
`grep '^[KU]' fichier.txt`
`grep "truc\|machin" .bashhistory` pour les lignes contenant « truc » ou contenant « machine »
`-w` pour une recherche par mot entier
`-C2` affichera les deux lignes autour de chaque ligne concernée (`-A2` et `-B2` les deux lignes après ou avant)
`grep "[A-Z].[v]" fichier.txt` liste les lignes contenant **Liv**e ou **V**i**v**ien
pour lister (`grep`) les lignes (`^`) commençant par mkv, suivi de n'importe quels caractères (`.*`) et finissant par l'extension mkv (le `$` sert à quoi ?) dans le fichier `.bashhistory`.
`find -iname "actress.csv" -exec cat {} ';' | grep Aniston`
cherche tous les fichiers actress.csv, et en affiche seulement les lignes contenant Aniston

13.5. La commande `awk`

Exemple du document-compagnon :
si le fichier `wcs.txt` contient :
Ada LOVELACE 1815 1852
Annie EASLEY 1933 2011
Grace HOPPER 1906 1992
alors

```
awk ' BEGIN {print "Les informaticiennes :"}
{print $2 " " $1 " " $3 "-" $4 " age: " $4 - $3}
END {print "Il y en a : " NR "."}' wcs.txt
```

donnera :
Les informaticiennes :
LOVELACE Ada 1815-1852 age: 37
EASLEY Annie 1933-2011 age: 78
HOPPER Grace 1906-1992 age: 86
Il y en a : 3.
`BEGIN` et `END` sont facultatifs s'il n'y a pas de boucle à traiter. Ces deux mots-clés créent une boucle équivalente à « pour chaque ligne fais ceci... ».
`NR` affiche le numéro de la ligne en cours.
`$(NF)` donne le nombre total de lignes.

Attention, dans le script suivant :

```
fichier=${1:?"Vous devez fournir un path"}
echo $fichier
awk '{print $(NF)}' $fichier
```

la commande `echo` va traiter le nom du fichier alors que `awk` va traiter chaque ligne (= le contenu) du fichier.

13.6. La commande `sed` et les expressions rationnelles

Expressions rationnelles = pattern matching.

À comprendre :

`*` permet de « multiplier » un caractère :

`L*` désigne un motif contenant zéro ou plusieurs `L`.

`.` désigne un caractère n'importe lequel et `.*` désigne un motif contenant rien ou plusieurs caractères quelconques.

reprend la syntaxe de `grep` :

`sed -i '/^mpl.*mkv$/d' .bashhistory`

pour modifier le fichier dans le fichier lui-même (`-i`) en supprimant (`d`) les lignes concernées par le filtre. Ces lignes concernées sont celles qui commencent (`^`) par mpl et qui finissent (`$`) par mkv.

`sed -i 's/truc/machin/g' fichier1.txt fichier2.txt` remplace tous les « truc » par « machin » dans les deux fichiers. Sans le `-g`, seule la première occurence est traitée.

`sed 's/a//' fichier.txt` affiche le contenu du fichier en supprimant tous les « a ».

`sed '/a/d' fichier.txt` affiche le contenu du fichier en supprimant toutes les lignes contenant un « a ».

Sans le `-i`, la modification est envoyée sur la sortie standard.

Attention ; sur certains systèmes il faut remplacer `-i` par `-i ''`.

`cat fichier.txt | sed -e 's/a/A/g' -e 's/TA/ta/g'`

transforme les a en A puis, chronologiquement, les TA en ta. Ainsi Table devient table.

les `-e` sont nécessaires avant chaque instruction lorsqu'il y a plusieurs.

`sed '1,5s/UNIX/*nix/g' fichier.txt` transforme les UNIX en *nix mais seulement sur les lignes 1 à 5.

`sed -e '/^C/s/a/A/g' fichier.txt` transforme les a en A seulement sur les lignes commençant par C.

`sed -e '/C/s/a/A/g' fichier.txt` transforme les a en A seulement sur les lignes contenant un C.

`sed -e '/C$/s/a/A/g' fichier.txt` transforme les a en A seulement sur les lignes finissant par C.

`sed -e 's/L.*s/A/g' fichier.txt` transforme toute occurence de Louis ou de Luzernes en la simple lettre A.

Pour supprimer toutes les extensions dans la `ls` d'un répertoire qui contient des `fichier.toto.txt` par exemple, deux stratégies possibles :

`sed 's/\..*//'` supprime tout ce qui est derrière le premier point ;

`sed 's/\([^.]*\)\..*/\1/'` garde tout ce qui est devant le 1er point

`sed -e 's/a.a/b&b/' fichier.txt` permet de

`sed -e 's/.*/Je dis: &/' slogan.txt` préfixe chaque ligne par « Je dis : ». Ici le `&` signifie « les occurences trouvées par `.*` ».

Si un fichier `fichier.txt` contient des lignes du style `2017-01-21`, la commande :

`sed -e 's/\(.*\)-\(.*\)-\(.*\)/date: \3 \2 \1/' fichier.txt` renverra :

`date: 21 01 2017`.

Pour remplacer tous les a****a par des i****i, faire :

`sed 's,a\(.\)a,i\1i,'`

Compliqué, pour supprimer tous les retours chariots (ou sauts de ligne) avec `sed` :

`nom_nouveau=$(sed ':a;N;$!ba;s/\n//g' nom.txt)`

Explication : l'astuce c'est d'ajouter toutes les lignes dans le *pattern space* avec « `N` » (qui ajoute un saut de ligne entre elles) et de n'exécuter la substitution que pour la dernière ligne.

En effet, `sed` lit une ligne seulement à la fois, donc s'arrête de lire dès qu'il rencontre un retour à la ligne. Ce qu'il a lu est écrit dans son "espace de travail" (sans le caractère de retour à la ligne), puis sed effectue le traitement sur ce qui est dans son "espace de travail", puis l'affiche en ajoutant un retour à la ligne, avant de passer à la ligne suivante. `sed` ne travaille pas sur tout le texte en entier directement, ce qui fait qu'il ne rencontre *a priori* aucun caractère de retour à la ligne parmi ceux qu'il traite.

14. EXPRESSIONS MATHÉMATIQUES

14.1. `expr`, commande archaïque

`expr 2 + 3` renvoie 5, tandis que `expr 2+3` renvoie « 2+3 ». et `a=$($a+1)` incrémente.

Utiliser `\(` et `\)` pour les parenthèses, et `\*` pour le produit, les autres symboles étant usuels, notamment le `%` pour le modulo.

14.2. la syntaxe `$((...))`

Tout simplement, `$((2*a+1))` renvoie $2a+1$.

Les opérateurs logiques fonctionnent aussi.

On peut écrire `a=$((2+1))`.

14.3. la syntaxe `((...))`

`((a=1+2))` fonctionne même si a n'a pas été définie avant, ce qui n'était pas le cas avec `$((...))`.

C'est équivalent à `let "a=1+2"`.

Après exécution d'un tel code, la variable standard `$?` renvoie :

- 0 ou 1 suivant si le calcul, s'il est numérique, a généré une erreur ou non ;
- 0 ou 1 pour vrai ou faux si le calcul est booléen.

`$((truc))` renvoie une valeur, contrairement à `((truc))` qui exécute une instruction : on peut écrire `a=$((b+2))` mais si on enlève le `$` ça n'a plus de sens en bash. Donc :

théorème : `a=$((b+2))` est équivalent à `((a=b+2))`.

14.4. La déclaration de variable

`declare -i a=2` ou `declare -i a` puis on peut manipuler des expressions type `a=2` sans les `((...))`.

`typeset` est équivalent à `declare`.

`a="toto" ; echo $a` renverra 0 si `a` a été déclarée.

14.5. La commande `bc`

Penser aussi à `awk` qui permet de faire des calculs mathématiques.

`bc` ou `bc -q` permet de rentrer en mode « maths ».

`quit` pour sortir.

avec un pipe : `b=10;echo "scale=10;$b/9" | bc` renvoie 1,1111111111

si un fichier contient des lignes contenant chacune un nombre, on peut avoir leur somme par :

`echo $(cat number.txt | tr '\n' +)0 |bc`

le 0 sert à ne pas terminer par un +.

14.6. Divers

`echo $(($RANDOM % 50))` renvoie un aléatoire entier dans $[|0;50|[$.

15. ARCHIVER ET COMPRIMER

Archiver : créer un fichier contenant l'information complète de tout un répertoire.

syntaxe : `tar cvf archive.tar dossier/`

Pour extraire : `tar xvf archive.tar`

Remarque : on peut faire `tar cvf archive.tar dossier1/ dossier2/ dossier3/...`

Attention : Le chemin vers le répertoire à archiver ne doit commencer ni par `/` ni par `.` ni par `..` afin que l'on puisse choisir la destination lors de l'extraction.

Lister le contenu d'une archive : `tar tvf archive.tar` ou `tar tvf archive.tar dossier1/`

Extraire une partie d'une archive : `tar xvf archive.tar dossier1/`

Attention, `f` est suivi du nom de l'archive, donc `tar -cfv X` crée l'archive nommée « v ».

Compresser : créer un fichier de taille réduite contenant la même information qu'un fichier donné.
`gzip archive.tar` efface `archive.tar` et le remplace par `archive.tar.gz`
`gunzip` a exactement l'effet inverse.
Pour compresser sans effacer le fichier original : `gzip -c monFichier > monFichier.gz`.
`zcat` est équivalent à `gzip -c`, ça envoie le fichier décompressé sur la sortie standard.

Comme on préfère éviter d'avoir deux points dans le nom du fichier, on utilise plutôt :
`tar cvfz archive.tgz dossier/` ce qui créera directement un archive.tgz équivalent au archive.tar.gz ci-dessus.
Pour extraire : `tar xvfz archive.tgz`

Binaire <-> Ascii :
`uuencode monFichier.xls > monFichier.txt`
et pour l'opération inverse : `uudecode`

16. Transferts

À approfondir :
`scp` pour transférer via SSH.
`wget` pour télécharger : `wget http://www.example.com/archive.tgz`
`ssh` pour ouvrir une session à distance.

17. Scripts shell

17.1. Méthode

Taper un script avec un éditeur de texte brut.
La première ligne doit toujours être `#!/bin/bash` et les remarques commencent par `#`
Vérifier qu'on a les droits dessus : `chmod +x fichier.sh`
L'éxécuter en tapant nom pas `fichier.sh` mais `./fichier.sh`
Pour que `fichier.sh` fonctionne dans spécifier son chemin, il faut que `.` soit dans la variable `PATH`
On peut aussi l'exécuter en faisant `bash fichier.sh`
Autre méthode : taper `/ici/la/fichier.sh` (en donnant donc le chemin absolu) permet d'exécuter la commande.

17.2. Alias

Taper `truc=fichier.sh` permet d'exécuter le fichier en tapant `./$truc`.
On peut aussi taper `truc=./fichier.sh` et l'exécuter en tapant `$truc`.
Le mieux est quand même d'utiliser le chemin absolu : `truc=/ici/la/fichier.sh` ce qui permet de l'exécuter par `$dada` où que l'on soit dans l'arborescence.
Attention, la commande `$dada` ne sera plus reconnue si l'on ferme puis ouvre une nouvelle session.

17.3. $PATH

Quand on affiche le $PATH par `echo $PATH`, on voit que `/usr/local/bin` y figure, cela veut dire que si on place `fichier.bash` dans ce dossier, on pourra, après avoir fait `cd /usr/local/bin` puis `chmod +x fichier.bash`, l'exécuter par `fichier.bash` sans avoir besoin du `./` devant.
Attention, ceci n'est valable que pour les fichiers directement dans `/usr/local/bin` (ou autre dossier du $PATH), pas dans un sous-dossier.

17.4. Créer une commande

17.5. Variables locales ou pas

Les variables restent **locales** à l'intérieur du programme : les commandes d'un script sont traitées dans un contexte d'exécution différent de celui du shell du terminal sur lequel il a été lancé. Et à la fin de l'exécution du script, son contexte d'exécution est supprimé.

17.5.1. « Globaliser » les variables

Pour rendre les variables du programme encore valides après l'exécution du script, on le lance par `source ./fichier.sh`

Par cette méthode, le script n'est pas exécuté comme un processus de plus, mais en quelques sortes « à l'intérieur » du processus bash lui-même.

Exemple, si `fichier/sh` contient `echo $x ; x=10 ; echo $x` on peut alors faire la différence entre :

`x=20`
`./fichier.sh`
qui affichera : *rien* puis 10
et
`x=20`
`source ./fichier.sh`
qui affichera 20 10

Le `source` est très utile si l'on souhaite par exemple un script qui modifie les variables d'environnement.

17.5.2. « Envoyer » une variable du shell vers le processus avec `export`

On peut vouloir donner une variable au script avant d'exécuter celui-ci et faire :
`export x=20`
`./fichier.sh`
qui affichera 20 10 mais par contre une fois le script terminé un `echo $x` affichera 20.

17.5.3. « Envoyer » une variable du shell vers le processus avec des arguments

On peut aussi transmettre au script des arguments en les ajoutant sur la ligne d'appel du script :
`./fichier.sh a b c`
puis utiliser dans le script :
`$2` vaut ici `b` ;
`$#` le nombre d'arguments donnés, ici 3 ;
`$@` la liste de tous les arguments.

Attention au guillemets

Si l'on appelle des variables, si par exemple `mot` contient l'expression « un arbre », alors la commande `./fichier.sh $mot` sera interprétée comme `./fichier un arbre` et donc la variable `$#` va renvoyer 2. Il faut alors mettre `./fichier.sh "$mot"` et bash l'interprète comme une seule variable envoyée au processus du script.

Valeurs par défaut

Au lieu d'utiliser la variable `$1` (premier argument), on peut écrire `A=$1` et utiliser `$A`, ce qui ne sert qu'à faire joli, mais cette idée permet `A=${1:-quelquechose}` qui, si le premier argument est omis, affecte néammoins la valeur « quelquechose » à la variable `a`.

On peut aussi, dans le même ordre d'idées, écrire :

`A=${1:?"Vous devez fournir un argument"}` qui, en cas d'argument n°1 oublié :

- renverra un code retour d'erreur (variable `$?`) ;
- affichera un message ;
- stoppera l'exécution du programme.

Ou bien prendre une variable fantôme pour imposer deux arguments :

```
IS_OK=${2:?"Vous devez fournir 2 valeurs"}
```

Convention

On a l'habitude d'utiliser des noms majuscules pour les variables correspondant à des arguments :

```
NOMBRE=$1
```

17.5.4. Variable retour

Exemple :

```
./fichier.sh
echo $?
```

Explication : si le script est exécuté comme un processus enfant du bash (donc si l'on ne fait pas `source`), alors la variable `$?` donne le code retour (0 si tout s'est bien passé, $\neq 0$ si erreur).

Pour contrôler le code retour, faire `exit 6` pour interrompre et renvoyer le code retour 6.

17.5.5. Exécution en mode débogage

Faire :

`bash -x fichier.sh` affiche tout ce qu'il fait
`bash -x -v fichier.sh` affiche tout ce qu'il doit faire puis tout ce qu'il fait
`bash -n fichier.sh` affiche les éventuelles erreurs de syntaxe

18. Tests

18.1. Syntaxe

Pour tester si $a > 10$, on peut faire `test $a -gt 10` ou `[ $a -gt 10 ]` (dans ce dernier cas, attention aux espaces contre les crochets !) qui vont envoyer le résultat du test dans la variable code retour, soit `$?` avec 0 pour vrai et $\neq 0$ pour faux.

18.2. Tests mathématiques

syntaxe `bash`	notation mathématique
`-eq`	$=$
`-ne`	$\neq$
`-gt`	$>$ (*greater than*)
`-lt`	$<$ (*lower than*)
`-ge`	$\geqslant$
`-le`	$\leqslant$

18.3. Tests sur des chaînes

syntaxe `bash`	interprétation
`-n`	chaîne non vide
`-z`	chaîne vide
`=` ou `==`	chaînes identiques
`!=`	chaînes différentes
`\<`	chaîne inférieure (dans l'ordre alphabétique)
`\>`	

pour les deux dernières lignes :

- il faut inhiber le comportement naturel de $>$ et $<$;
- l'ordre est celui de ASCII :
  - chiffres = $\{48; ...; 57\}$;
  - majuscules = $\{65; ...; 90\}$, minuscules = $\{97; ...; 122\}$.

Attention aux chaînes :

- `[ 32 \> 200 ]` renvoie 0 (vrai) car 32 et 200 sont pris par défaut comme des chaînes ;
- si `a=$((32))` et `b=$((200))` alors `[ $a \> \b ]` renvoie 1 (faux) ;
- `[ $mot1 \> $mot2 ]` peut générer une erreur si l'une des deux variables n'existe pas, ce qu'on peut contourner par `[ "$mot1" \> "$mot2" ]` : la variable qui n'existe pas sera substituée par une chaîne vide.

18.4. Tests sur des fichiers

Faire `man test` pour avoir l'exaustivité

syntaxe `bash`	interprétation
`-s`	fichier non vide
`-f`	est un fichier (pas un répertoire)
`-d`	est un répertoire
`-e`	existe (fichier ou répertoire)
`-r`, `-w`, `-x`	teste les droits correspondants (pour le propriétaire)
`-nt`	fichier plus récent qu'un autre
`-ot`	fichier plus ancien qu'un autre

18.5. Opérateurs logiques

syntaxe `bash`	interprétation
`!`	négation de ce qui suit
`-a`	et
`-o`	ou

dans l'ordre des priorités, on a d'abord `!`, puis `-a`, puis `-o`.

On peut penser aux correspondances mathématiques suivantes :

`!`	$-$
`-a`	$\times$
`-o`	$+$

18.6. Tests étendus

On remplace `[ a \> b ]` par `[[ a > b ]]` :

- plus besoin d'inhiber le $>$ ou le $<$;
- on peut (il faut ?) utiliser `&&` et `||` respectivement pour `-a` et `-o` ;
- on peut évaluer les expressions arithmétiques sans avoir besoin de les calculer avant ;
- dans la comparaison `==` ou `!=` de chaînes, la seconde variable peut utiliser le *pattern matching*.

Note : le code retour de `((...))` peut être utilisé mais c'est délicat.

18.7. && et ||

on fait `[ test ] && commande` et la commande ne sera exécutée que si le $? du test est 0.
on fait `[ test ] || commande` et la commande ne sera exécutée que si le $? du test est $\neq 0$.

Exemple :

```
[ -f toto.txt ] && cat toto.txt || echo "le fichier toto n'existe pas" >&2
```

si le résultat du [-f] est 0 alors le cat s'éxécute sans erreur et s'il est $\neq 0$ alors le cat ne s'exécute pas et on enchaîne sur le echo.

18.8. if

la syntaxe est :

```
if [ ici un test... ]
then
    commandes
else
    commandes
fi
```

l'indentation des commandes semble purement esthétique.

L'exemple avec toto.txt ci-dessus s'écrirait alors :

```
if [ -f toto.txt ]
then
    cat toto.txt
else
    echo "le fichier toto n'existe pas" >&2
fi
```

Le mot-clé `elif` existe, comme en Python.

18.9. case

```
case a in
   motif1)
      jdkslqjdksq ;;
   motif2 | motif 3)
      dsqjkdsqkjl ;;
   esac
```

le | signifie « ou ».

19. Boucles

19.1. While

Syntaxe :

```
while [ ... ];
do
   commandes
done
```

Exemple :

```
#!/bin/bash
MOT=${1:?"Vous devez indiquer un mot"}
NB=${2:-10}
i=0
while[$i-lt $NB];do
   echo $MOT
   i=$(( $i + 1 ))
done
```

19.2. While avec read

On peut lire un fichier ligne par ligne avec `read` en prenant comme argument le `$1` en le faisant « rentrer » sur la boucle `while` au niveau du `done` comme suit :

```
#!/bin/bash
```

```
# nommons ce script "read.sh"
while read ll; do
   echo -n "Ma ligne : "
   echo $ll
done < $1
```

Lançons le script `read.sh` avec comme paramètre le fichier `read.sh`.
Nous obtenons :

```
Ma ligne : #!/bin/bash
Ma ligne : # nommons ce script "read.sh"
Ma ligne : while read ll; do
Ma ligne : echo -n "Ma ligne : "
Ma ligne : echo $ll
Ma ligne : done < $1
```

19.3. break et continue

`break` arrête la boucle
`continue` ne finit pas la boucle et passe à la suivante.

19.4. Boucles `for`

Exemple qui va afficher tous les arguments un par un :

```
#!/bin/bash
NUM=1
for V in $@;do
  echo "argument $NUM = $V"
  NUM=$(( $NUM + 1 ))
done
```

Boucle qui utilise `seq`
`seq 1 5` renvoie `1 2 3 4 5`.

```
#!/bin/bash
MOT=${1:?"Vous devez indiquer un mot"}
NB=${2:-10}
for i in $(seq 1 $NB);do
   echo $MOT
done
```

Syntaxe inspirée du *C* :

```
#!/bin/bash
for ((i=1; i<=4; i++)); do
   echo "Nombre $i"
done
```

20. FONCTIONS

20.1. Fonctionnement général

On écrit :

```
fonctiontruc() {
...
return chose
```

```
}
#corps du programme
...
fonctiontruc arguments
echo $?   # renverra chose si 0<=chose<256
...
```

Le code retour de la fonction est utilisé comme « résultat » de la fonction. Attention, un code retour est toujours entre 0 et 255 !!!! Deux contournements possibles :

- `echo $(fonctiontruc arguments)` ;
- ou bien utiliser une variable (elles sont globales par défaut).

Les arguments sont gérés par les classiques `$1,$2..` et `$#` (nombre d'arguments) et `$@` (liste des arguments) et enfin `$*` (chaîne contenant tous les arguments).

Toute variable définie à l'intérieur d'une fonction est globale par défaut. En cas de gros programme contenant beaucoup de fonctions, pour rendre « muette » une variable, on fait :
`local $a` ou `local $a=2`.

La commande `return` joue pour les fonctions le rôle de la commande `exit` pour les scripts : elle interrompt la fonction (pas le script qui appelle la fonction évidemment) et renvoie un code.

Si l'on fait `$(foo)` et que dans la définition de foo il y a `echo $R` alors `$(foo)` renverra la valeur de `$r`.

20.2. Alias

Dans les fichiers `.bashrc` ou `.profile` sont définis des alias

Un alias est défini par `alias nomalias="commande"` ou `alias nomalias='commande'`.

Pour supprimer un alias on fait `unalias nomalias`.

Pour avoir des alias avec paramètre (qui s'appellent *fonctions interactives*) on fait :

```
mkd() {
mkdir -p "$@" && cd "$@"
}
```

qui crée la commande `mkd` ayant pour effet de créer un répertoire avec l'argument donné et de s'y placer, le `-p` autorisant les sous répertoires par exemple `mkd /truc/machin` crée cela et s'y place.

Rappel : quand on modifie `.bashrc`, il faut lui faire `source` pour actualiser.

Ces alias ne peuvent s'utiliser que dans la ligne de commande, pas dans un script. À vérifier.

Une bonne pratique est de séparer les fichiers contenant les alias et les fonctions en créant des fichiers `.alias` et `.func`, et de rajouter dans le fichier `.bashrc` les lignes suivantes :

```
source .alias
source .func
```

21. EXEMPLES DE SCRIPTS ET BOUTS DE CODE

21.1. Renommage fichier par fichier

Renomme le fichier pris en argument en lui rajoutant `_alice` à la fin du nom, exemple `a.truc.txt` devient `a_alice.truc.txt` :

```
fichier=${1:?"Erreur : vous devez indiquer le nom du fichier a renommer"} n
nom=$(echo $fichier|cut -f1 -d.)
extensions=$(echo $fichier|cut -f2- -d.)
```

```
nouveau=$(echo $nom _alice. $extensions)
nouveau=$(echo $nouveau | sed 's/ //g')
mv $fichier $nouveau
```

Attention dans la ligne :
`extensions=$(echo $fichier|cut -f2- -d.)`
à ne pas mettre d'espace avant le = sinon il cherche à interpréter `extensions` comme une commande.

Utilise le script précédent pour renommer en bloc :
`find -name "*truc*" -printf %f'\n' | while read f ; do tagalice.sh $f ; done`

21.2. Droits

Teste si le fichier en argument est bien un fichier (pas un dossier) et est protégé en écriture (pour le propriétaire)

```
#!/bin/bash
a=$1
[ -f "$a" -a ! -w "$a" ]
```

21.3. Gestion des fichiers TEXMACS

```
#!/bin/bash
fichier=${1:?"Vous devez fournir un path"}
#je mets le / à la fin des variables contenant un path
lepdf=/Users/vincentdouce/.TeXmacs/system/tmp/preview.pdf
#l'emplacement du pdf tel que par défaut by TeXmacs
lepath=${fichier%/*}/
cd $lepath
lenom=${fichier##*/}
lenomsanslextension=${lenom%.*}
lesimages=${lenomsanslextension}!images/
if [ -e ${lepdf} ]
then
#apparemment le then doit être à la ligne sinon erreur...
mv -f ${lepdf} ${lenomsanslextension}.pdf
#prend le pdf preview dans /tmp et le renomme puis le met à côté du tm
else
echo "----------------------------"
echo "attention, le pdf n'y est pas" >&2
#le fait de mettre >&2 change quoi ?
echo "----------------------------"
fi
if [ -e ${lenomsanslextension}.zip ]
then
rm ${lenomsanslextension}.zip
#vire l'archive existante
fi
zip -r ${lenomsanslextension}.zip \
        ${lenomsanslextension}.tm \
        ${lesimages} \
```

21.4. Divers

Calcule le discriminant

```
#!/bin/bash
## fichier : delta.sh ###
  A=${1:?"Erreur : vous devez indiquer la valeur de A"}
  B=${2:?"Erreur : vous devez indiquer la valeur de B"}
  C=${3:?"Erreur : vous devez indiquer la valeur de C"}
  echo $((B*B-(4*A*C)))
###
```

Teste si le mot contient un `X` et un `Y`.

```
#!/bin/bash
a=$1
[[ "$a" == *X*Y* || "$a" == *Y*X* ]]
```

ensuite on fait `./programme.sh mot ; echo $?`, si c'est 0 le test est bon, si c'est 1, le test est négatif, si c'est 2 ou autre, il y a une erreur dans l'écriture ou dans l'exécution du test.

Écrit dans un fichier :

```
#/bin/bash
if [ -f message -a -w message ]
then
    echo "I love Bash" >> message
elif [ ! -e message ]
then
    echo "I love Bash" > message
elif [ -f message -a ! -w message ]
then
    chmod u+w message
    echo "I love Bash !" >> message
else
    echo "message existe mais n'est pas un fichier ordinaire" >&2
fi
```

Affiche le type d'un fichier en fonction de son extension :

```
#!/bin/bash
if [ $# -ne 1 ]
then
    echo "Erreur : nombre d'argument incorrect" >&2
else
case $1 in
*.c) echo "Code source en langage c";;
*.sh) echo "Script bash";;
*.jpeg | *.jpg | *.png) echo "Image";;
*) echo "Type de fichier non reconnu";;
esac
fi
```

Pose une question et réagit à la réponse

```
 #!/bin/bash
"Etes vous d'accord avec la loi (oui ou non) ?"
read rep
case $rep in
[oO] | [oO][uU][iI])
    echo "Ok, merci";;
[nN] | [nN][oO][nN])
    echo "Tres bien, je respecte votre choix";;
*)
    echo "Desole, je ne comprends pas votre reponse";;
```

```
esac
fi
```

Même genre de truc sans utiliser de `if`

```
#!/bin/bash
[ $# -eq 0 ] && echo "Vous n'avez pas donne votre reponse" && exit 3
[ $# -gt 1 ] && echo "Donnez une seule reponse" && exit 4
case $1 in
[oO])
  echo "oui" ;;
[nN])
 echo "non" ; exit 1 ;;
*)
  echo "Pas compris" ; exit 5 ;;
esac
```

teste un caractère d'entrée

```
#!/bin/bash
if [ $# -ne 1 ]
then
  echo "Donner exactement un argument"
  exit 90;
fi
case "$1" in
  [a-z]) echo Minuscule; exit 10;;
  [A-Z]) echo Majuscule; exit 11;;
  [0-9]) echo Chiffre; exit 12;;
  ??*) echo "Donner un seul caractere"; exit 91;;
  *) echo Autre; exit 20;;
esac
```

Récupérer le résultat d'une fonction :

```
#!/bin/bash
eholecho () {
(( a=200+$1 )) # variable globale
echo 300       # uniquement pour être récupéré par le $()
return 300
}
echo "1) avec \$?"
eholecho
echo $?        # marchera pas car 300>255
echo "2) avec \$()"
echo $(eholecho 100)           # marche très bien
gg=$(eholecho 100) ; echo $gg  # marche très bien aussi
echo "3) avec variable globale"
eholecho 100
echo $a                        # marche très bien
```

Modifier les noms des fichiers :

```
for i in $(ls); do fichier=$(echo $i | sed 's/E//g'| cut -d'_' -f2,3 | sed "s/
_//g"); mv $i $fichier 2>/dev/null ; done
```

22. LIENS VERS DE LA DOCUMENTATION

TLDP

TLDP aussi

beaucoup de liens sur la page de conclusion du Mooc.

Eh bien, pour pratiquer de manière agréable, je vous invite à vous attaquer aux défis de CodinGame (https://www.codingame.com).

Guide avancé d'écriture des scripts Bash: Une exploration en profondeur de l'art de la programmation shell Mendel Cooper 5.1.04 Publié le 26 décembre 2007

http://aral.iut-rodez.fr/fr/sanchis/enseignement/bash/

http://wiki.bash-hackers.org/start

pour pouvoir faire du Bash dans le bus, en métro, entre 2 rendez-vous et de façon ludique, 2 applications (payantes): Mimo et SoloLearn