Architecture et système

Bash

C. Nguyen, J. Razik

Département d'informatique
Université de Toulon

Motivations

Ce langage de script permet de configurer et d'administrer un ordinateur par l'intermédiaire de lignes de commande à destination du système d'exploitation.

Depuis 2016, le shell Bash [1] est intégré dans les trois OS Windows, MacOS et Linux et dans de multiples architectures (smartphone, tablette, nano-ordinateur, etc.).

Ce type de programmation s'appuie sur le paradigme « diviser pour régner » et consiste principalement à chaîner des utilitaires : « *Chaque commande fait une seule chose et le fait bien* ».

Un shell est portable, économe en ressource, proche du système (et de sa logique) par définition et donc efficace. Il est en réalité bien plus souple d'utilisation qu'une interface graphique.

1. Bourne Again Shell

Aide en ligne

La plupart des commandes du Bash proposent une aide, affichée dans le terminal, via une option (le plus souvent `--help` ou `-h`).

On dispose aussi d'un manuel via la commande `man`. Par exemple, pour accéder au manuel de la commande `date`, on saisira la commande `man date`.

Remarque : le système peut être configuré en anglais, mais on a la possibilité d'afficher l'aide en français (si elle a été installée) avec l'option `-L fr`.

Le manuel du `man` est subdivisé en sections pour lever les ambiguïtés, dont seule la première concerne les commandes. Exemple : `man 1 printf` et `man 3 printf`.

Aide en ligne

La commande `help` propose de la documentation pour les commandes internes (*inline*) de Bash (par exemple, la commande `cd`).

La commande `info` est concurrente de `man` et interprète la documentation au format texinfo (format officiel de GNU [2]), elle permet la navigation en hypertexte.

La commande `apropos` permet de lister les manuels dont la description comprend un des mots passés en paramètre.

2. acronyme récursif « GNU's Not UNIX »

Plan

Fonctionnement

Le shell est interprété et son fonctionnement suit ces étapes :

1. saisie d'une ligne de commande (à la suite de l'invite),
2. édition évoluée de la saisie (correction, répétition, etc.),
3. analyse à la confirmation : identification de la commande et de ses arguments,
4. interprétation (avec substitution éventuelle),
5. définition de l'entrée et de la sortie de la commande,
6. exécution de la commande avec ses arguments substitués.

Interaction

L'interaction est indispensable dans le cadre de l'utilisation de l'entrée clavier, afin de gérer les oublis, erreurs de frappe et répétitions de façon efficace.

Elle se fait via des raccourcis clavier.

déplacements		couper / coller	
CTRL+a	début de ligne	CTRL+k	coupe (fin de ligne)
CTRL+e	fin de ligne	CTRL+u	coupe (début de ligne)
CTRL+b	recule d'un car.	CTRL+w	coupe (début de mot)
CTRL+f	avance d'un car.	ALT+d	coupe (fin de mot)
ALT+b	recule d'un mot	CTRL+y	colle
ALT+f	avance d'un mot		

modification			
ALT+c	capital	ALT+l	minuscule
CTRL+_	annulation	ALT+u	majuscule

Historique

Lors d'une session de travail, on est amené à utiliser plusieurs fois les mêmes commandes.

Bash conserve l'historique des 500 dernières commandes exécutées et fournit plusieurs moyens de le consulter.

La commande `history` liste les commandes exécutées de la plus ancienne à la plus récente (`history n` affiche les *n* dernières commandes).

Déplacements et rappels :

- ↑ et ↓ permettent de naviguer dans l'historique,
- !! permettent de rappeler la dernière commande,
- ! suivi d'une chaîne rappelle la commande la plus récente qui commence par cette chaîne (exemple : `!cd:p`),
- en encadrant la chaîne de ? on recherche la commande la plus récente qui *contient* cette chaîne (exemple : `!?to?`),

Historique

Rappels, recherches, réutilisations et modifications :

- ! suivi d'un numéro permet de rappeler la commande correspondante (exemple : `!217`),
- `CTRL+r` effectue une recherche dans l'historique,
- pour réutiliser les arguments de la dernière commande : `!*` correspond à tous les argument, `!^` au premier argument et `!$` au dernier,
- pour modifier les arguments on utilise l'option `:s` (exemple : on a exécuté la commande `wc prg1.py`, `!!:s/1/2` permet d'obtenir la commande `wc prg2.py`) ; avec l'option g la substitution est globale.

Auto-complétion

Une troisième forme d'aide à la saisie prend la forme de l'auto-complétion.

Elle s'effectue au moyen de la touche de tabulation après avoir saisie les premiers caractères.

S'il n'y a qu'une seule complétion possible, elle est affichée, sinon il faut presser deux fois la touche de tabulation pour afficher la liste des complétions possibles.

Plan

Principes

Dans les langages de commandes, une syntaxe particulière déclenche une étape de substitution (le remplacement d'un mot par l'interprétation de ce mot).

⚠ ces constructions sont interprétées **avant** que la commande s'exécute.

Il y a plusieurs types de substitution (*expansion*) :

- de caractères spéciaux,
- de variables,
- de commandes.

Le mécanisme de substitution est contrôlé par l'utilisation éventuelle du mécanisme d'inhibition.

Substitution (*globbing*)

Le Bash interprète par défaut certains caractères, ils sont dits spéciaux. Une phase d'analyse syntaxique a lieu avant exécution.

On a déjà rencontré des caractères spéciaux pour désigner des répertoires (., .. et ~).

Pour désigner plusieurs fichiers par une seule chaîne de caractères, on utilisera avec profit les caractères ? et * :

- `?` permet de désigner n'importe quel caractère[3],
- `*` permet de désigner n'importe quelle séquence de caractères[3].

Exemple : `ls p*` (ou `ls prg?.py`) ⇒ `prg1.py prg2.py`

⚠ `rm *` ⇒ suppression de tous les fichiers (dont les « cachés ») !

3. sauf le point comme premier caractère.

Substitution (*globbing*)

Il y a aussi la possibilité de déclarer un ensemble de caractères, celui-ci doit être entre crochets.

Exemple : `ls [A-Z]*[0-9]` affiche tous les fichiers dont le nom commence par une majuscule et se termine par un chiffre.

Classes de caractères[4] : elles sont représentées par un nom qui décrit l'ensemble des caractères encadré par `[:` et `:]` :

- `[:alnum:]` : alphabétiques et numériques,
- `[:alpha:]` : lettres,
- `[:digit:]` : chiffres,
- `[:lower:]` : lettres minuscules,
- `[:space:]` : blancs (espace, tabulation, ...),
- `[:upper:]` : lettres majuscules.

4. norme POSIX (Portable Operating System Interface UniX).

Substitution de variable

Une variable est un espace de stockage en mémoire identifié par un nom.

En Bash une variable n'a pas à être déclarée, et ne peut prendre qu'un type de valeur : la chaîne de caractères.

Le nom d'une variable doit être formé uniquement de caractères alphanumériques (A-Z, a-z et 0-9) et du blanc souligné (_).

L'affectation suit la syntaxe :

```
nom_var=valeur
```

⚠ notez l'absence d'espace autour du caractère égal.

Substitution de variable

Pour retrouver la valeur d'une variable, il faut préfixer son nom par `$`.

L'accès à une variable sans affectation préalable ne produit pas d'erreur et retourne le valeur vide (*null value*).

Exemple : `echo :$truc:` ⇒ `::`

Remarque : la commande (interne) `set` définit des options du Bash, par exemple `set -u` traite les variables non définies comme des erreurs lors de la substitution.

La commande (interne) `unset` supprime une variable.

La forme complète de substitution utilise les accolades : `${var}`.

Exemple : `agent=00 ; echo ${agent}7` ⇒ 007

Substitution de variable

Un certain nombre d'opérations sont possibles sur le contenu des variables (qui sont des chaînes de caractères).

- `${#var}` : longueur de la chaîne `var`,
- `${var:`p`:`l`}` : extraction d'une sous-chaîne à partir de la position p et de longueur l,
- `${var/`ch_1`/`ch_2`}` : remplacement de la sous-chaîne ch_1 par la sous-chaîne ch_2,
- `${var/`ch`}` : suppression de la sous-chaîne ch,
- `${var^^}` : conversion en majuscule,
- `${var,,}` : conversion en minuscule,

Substitution de commande

Elle consiste à remplacer une commande par le résultat de celle-ci.

Syntaxe : $(commande)

La substitution de commande s'utilise pour donner comme argument à une commande le résultat d'une autre commande.

Exemple : `echo chemin $(pwd)` ⇒ `chemin /home/alice`

Inhibitions (*quoting*)

On peut contrôler le mécanisme de substitution via trois formes d'inhibitions :

- \ : inhibition d'un caractère,
- ' ' : inhibition totale,
- " " : inhibition partielle, la substitution de variable, de commande et de caractère spécial est maintenue.

Exemple d'inhibition de caractère : `echo \$a` ⇒ `$a`

Exemple d'inhibition totale : `echo '* $(pwd)'` ⇒ `* $(pwd)`

Exemple d'inhibition partielle : `a=* ; echo "$a"` ⇒ `*`

Ordre de priorité

L'interprétation de la ligne de commande se fait dans l'ordre suivant :

1. substitution de variable,
2. substitution de commande,
3. substitution de noms de fichier.

Si l'on veut changer cela, la commande `eval` permet de relancer la phase d'interprétation, le résultat de la première interprétation sert à constituer la ligne de commande pour la seconde interprétation.

Exemple :
```
x=10 ; p=x
echo $p ⇒ x
echo \$$p ⇒ $x
eval echo \$$p ⇒ 10
```

Plan

Exécution des commandes

Il faut distinguer les commandes internes des commandes externes :

- une commande interne est propre au shell,
- une commande externe est en réalité un fichier exécutable ayant comme nom, le nom de la commande.

L'exécution d'une commande externe entraîne la création d'un processus exécutant le programme correspondant à la commande.

Processus

Un programme est un fichier exécutable. Un processus est un programme en cours d'exécution.

Un programme peut donner naissance à un nombre indéterminé (quoique limité [5]) de processus, qui sont autant d'instances de la commande en cours d'exécution et qui sont gérés par l'OS.

Chaque processus est décrit par un contexte d'exécution, dans lequel est pris en compte :

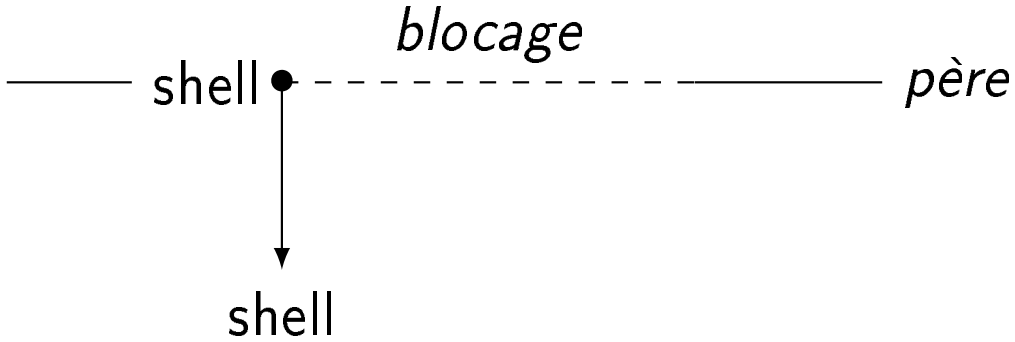
- l'état d'avancement de l'exécution,
- les ressources utilisées,
- les droits associés,
- les attributs.

5. cf. `ulimit -u`

Création

La création d'un processus se fait par un autre processus (sauf `systemd`).

Le shell est lui-même un processus avec lequel l'utilisateur interagit. Lorsqu'une commande est exécutée, le programme est recherché[6]. Puis, le shell crée un clone de lui-même et demande à l'OS de le remplacer par le programme (mais conserve les fichiers standards 0, 1, 2 ouverts).

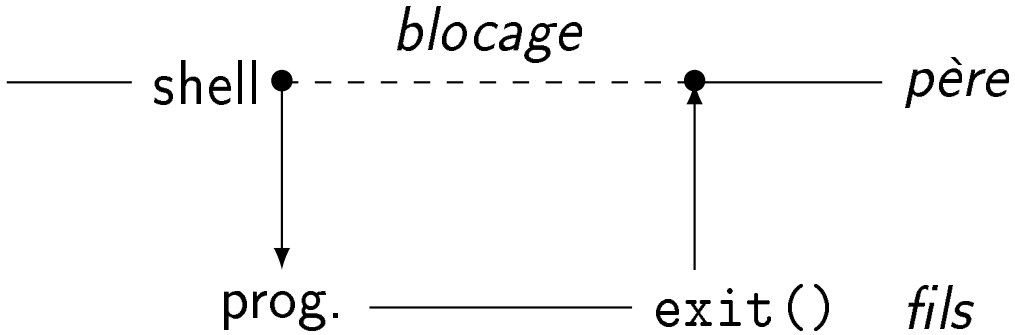


6. dans les répertoires enregistrés par la variable d'environnement `PATH`

Création

La création d'un processus se fait par un autre processus (sauf `systemd`).

Le shell est lui-même un processus avec lequel l'utilisateur interagit. Lorsqu'une commande est exécutée, le programme est recherché[6]. Puis, le shell crée un clone de lui-même et demande à l'OS de le remplacer par le programme (mais conserve les fichiers standards 0, 1, 2 ouverts).



6. dans les répertoires enregistrés par la variable d'environnement `PATH`

Retour

Pourquoi le père n'exécute pas la commande lui-même ?

Quand un processus se termine, il renvoit un code retour qui indique au processus parent comment s'est déroulé son exécution. Ce code est mémorisé dans la variable ?.

Exemple : `ls /home ; echo $?` ⇒ 0
`ls /xyz ; echo $?` ⇒ 2

Retour

💬 Pourquoi le père n'exécute pas la commande lui-même ?

👉 Car ainsi le père n'est pas affecté par une erreur et peut la contrôler.

Quand un processus se termine, il renvoit un code retour qui indique au processus parent comment s'est déroulé son exécution. Ce code est mémorisé dans la variable ?.

Exemple : `ls /home ; echo $?` ⇒ 0
`ls /xyz ; echo $?` ⇒ 2

Attributs

Un processus est caractérisé par plusieurs paramètres.

champ	signification	rôle
`PID`	*process id*	numéro unique
TTY	TeleTYpewriter	terminal d'attachement
TIME		temps CPU cumulé
CMD	CoMmanD	commande et ses arguments
UID	User ID	identifiant du propriétaire
PPID	Parent PID	PID du processus parent
PRI	PRIority	0 = priorité maximum
STAT	STATe	état (R)un, S)leep, Z)ombie, ...)

Commande : `ps -l`

Exécution

Par défaut, le shell exécute les tâches (*jobs*) en séquence, on doit attendre la fin d'une commande pour lancer la suivante.

Son fonctionnement en multi-tâches, qui consiste à libérer le terminal au lancement d'une commande, autorise un traitement « parallèle » [7].

En pratique, on lance une commande en arrière-plan (*background*) :

- soit par l'ajout du caractère `&` en fin de commande,
- soit en suspendant la commande par `CTRL-z` puis en l'envoyant en arrière-plan par la commande `bg`.

Pour ramener une commande en avant-plan on utilise la commande `fg` (*foreground*).

7. pas réellement pour un CPU mono-processeur, mono-cœur

Exécution

Exemple

```
$ sleep 10 &
[1] 13869
$ sleep 12 &
[2] 13870
$ jobs
[1]-  En cours d'execution   sleep 10 &
[2]+  En cours d'execution   sleep 12 &
$ jobs
[1]-  Fini                   sleep 10
[2]+  En cours d'execution   sleep 12 &
```

Interruption

Il est parfois utile d'interrompre l'exécution d'un processus (afin de libérer les ressources associées).

Si l'on connait le numéro du processus (PID), on peut lui envoyer un signal via la commande `kill`.

Il existe des dizaines de signaux, dont `SIGTERM` (15) par défaut, `SIGSTOP` (17), `SIGINT` (2) ou `SIGKILL` (9) pour interrompre un processus.

Si le processus est en avant-plan, il suffit d'utiliser la combinaison de touches `CTRL-z` (SIGSTOP), `CTRL-c` (SIGINT) ou `CTRL-\` (SIGQUIT)[8].

8. à ne pas confondre avec `CTRL-d` EOF

Interruption

Exemple

```
$ sleep 10 &
[1] 14618
$ kill -2 14618
$ jobs
[1]+  Interrompre                 sleep 10
```

Lorsque le numéro de processus n'est pas connu, on peut utiliser la commande `killall` suivi du nom du processus.

Entrée/Sortie

Habituellement, un processus lit des données en entrée et écrit ses résultats en sortie.

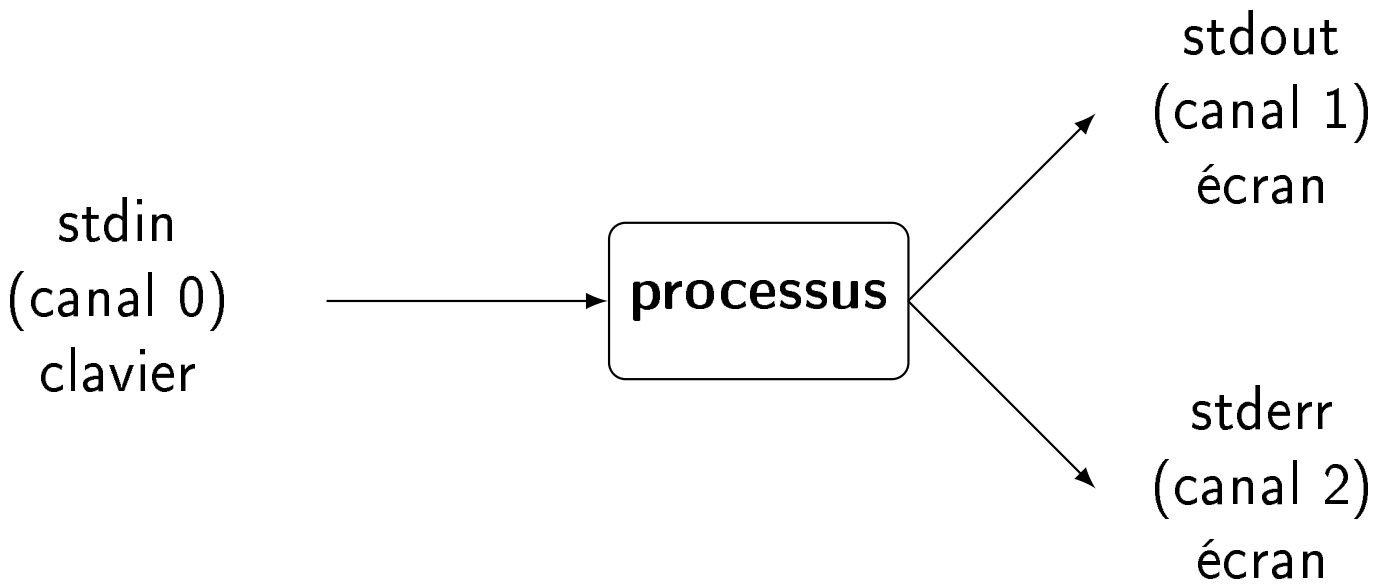
Par défaut, l'entrée dite standard (`stdin`) est le clavier et la sortie standard (`stdout`) l'écran [9]. Pour l'OS ce sont de simples fichiers.

Il est possible de rediriger aussi bien les entrées que les sorties des processus : c'est le mécanisme de redirection.

Cette redirection peut se faire aussi bien depuis ou vers un fichier, que depuis ou vers un autre processus.

9. il y a un 3ème canal qui est la sortie standard d'erreur (`stderr`)

Entrée/Sortie



Redirection

Le mécanisme de redirection des E/S permet d'utiliser des fichiers « classiques » en lieu est place des E/S standards.

Pour rediriger la sortie, on utilisera les caractères spéciaux > (création) ou >> (ajout).

Exemple : `ls > liste.txt`

Pour rediriger les messages d'erreur, on utilisera la suite de caractères 2> (création) ou 2>> (ajout), le « 2 » fait référence au canal associé aux erreurs.

Remarque : on peut ignorer les messages d'erreur en redirigeant la sortie d'erreur vers le fichier spécial `/dev/null`.

Il est aussi possible de rediriger conjointement la sortie et les messages d'erreur par &>.

Redirection

La redirection de l'entrée trouve son utilité dans l'automatisation des tâches.

Elle se fait grâce au symbole < (la double redirection d'une entrée n'a pas de sens).

Exemple : `wc < liste.txt`

Branchement

Les résultats d'une commande peuvent être transmis en entrée d'une autre commande sans passer par un fichier intermédiaire.

Pour cela, on combine les commandes par l'intermédiaire du symbole | (tube ou *pipe*).

Exemple : `history | wc -l`
ce qui est équivalent à :
`history > combien ; wc -l < combien`.

Remarque : on n'est pas limité à deux commandes associées par un tube.

Plan

Introduction

Il est indispensable de pouvoir adapter un outil à ses besoins. Cela facilite son apprentissage puis son usage.

Le bash dispose pour cela de variables[10] d'environnement et de commandes associées.

Il faut distinguer les variables de l'utilisateur des variables propres au bash. Ces dernières ont des noms prédéfinis et sont utilisées par le shell et d'autres programmes.

Elles sont propres à chaque utilisateur et à chaque session. Leur modification influe sur le comportement du bash.

10. association de noms et de valeurs

Variables d'environnement

Il existe de nombreuses variables, mais l'une d'elles est plus importante, il s'agit de `PATH`.

Cette variable contient les chemins d'accès aux commandes du système, elle représente une règle de recherche systématique et *ordonné* pour le shell.

Exemple : `PATH=/home/nguyen/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin`

Exemple : `cd` est en fait traduit en `cd $HOME`.

Les options du shell permettent de modifier son comportement, voir la commande `shopt`.

Environnement de travail

Adapter l'environnement de travail du bash passe par des variables associées à une session.

La commande `env` liste les variables courantes.

La commande `set` définit ou invalide des options et des paramètres du shell.

La commande `unset` permet de supprimer une variable.

La commande `export` est associée à la notion de sous-shell, elle permet de promouvoir des variables définies par l'utilisateur en variables d'environnement.

Environnement de travail

Exemple

```
$ x=1
$ echo $x
1
$ unset x
$ echo $x

$ x=1
$ bash
$ echo $x

$ exit
$ export x=1
$ bash
$ echo $x
1
```

Fichiers de configuration

Les variables d'environnement sont mémorisées dans des fichiers ce qui permet de les créer dès la connexion de l'utilisateur.

Les variables communes à tous les utilisateurs et à plusieurs shells (sh, ksh, bash, ...) sont définies dans le fichier `/etc/profile`.

Les variables propres à chaque utilisateur sont dans le fichier `$HOME/.profile`, qui inclut pour le bash le fichier `$HOME/.bashrc`[11] s'il existe.

11. *bash running commands*

L'invite de commande

Les informations affichées dans l'invite sont spécifiées dans la variable `PS1` (*prompt shell*).
Commandes (liste non exhaustive) :

- \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 : nom du shell

Exemple

```
$ PS1="\u > "
alice >
alice > PS1="\u :\w > "
alice :~> cd /tmp
alice :/tmp >
```

Plan

Introduction

Les commandes qui lisent et écrivent sur les E/S sont appelées des filtres.

Un filtre est un moyen de simplifier, séparer ou clarifier un flux de données. Un flux de données est souvent un texte (fichier) ou une chaîne de caractères (sortie d'une commande).

Les filtres sont souvent utilisés en branchement de commande (en réalisant un tube à l'aide du caractère |).

Découpage d'un fichier (head, tail, split, sort)

Les commandes `head` et `tail` permettent de n'afficher que les premières ou dernières lignes (10 par défaut) d'un fichier ou de l'entrée standard.

Exemple : `head ~/.bashrc`.

La commande `split` permet de découper un fichier en plusieurs morceaux.

Exemple : `split -l 100 /var/log/syslog log_`.

La commande `sort` permet de trier les lignes d'un fichier avant affichage. Options : -a alphabétique, -n numérique, -k numéro de colonne, ...

Exemple : `cat annuaire.txt | sort -nk 3`.

Extraction d'informations (cut)

La commande `cut` permet d'extraire des colonnes d'un fichier :

- -d : définition du séparateur (TAB par défaut)
- -c : extraction de caractères (exemple : -c1-2)
- -f : extraction de colonnes (exemple : -f2,5)

Exemple (csv : comma-separated values)

```
$ head -2 annuaire.csv
Sexe,Prénom,Année de naissance
M,Alphonse,1932
$ cat annuaire.csv | cut -d ',' -f 1,3 | head -2
Sexe,Année de naissance
M,1932
```

Assemblages (cat, paste, join)

La commande `cat` permet en réalité de concaténer plusieurs fichiers les uns à la suite des autres et d'afficher le résultat sur `stdout`.

La commande `paste` permet de fusionner les colonnes contenues dans plusieurs fichiers.

La commande `join` permet de joindre des fichiers en fonction de mots clés.

Exemple

```
$ paste -d \| annuaire.csv bash.csv | head -2
Sexe,Prénom,Année de naissance|Prénom,Niveau
M,Alphonse,1932|Alphonse,bash newbie
$ join -t ',' -1 2 -2 1 annuaire.csv bash.csv
Prénom,Sexe,Année de naissance,Niveau
Alphonse,M,1932,bash newbie
```

Modification (uniq, tr)

La commande `uniq` permet de supprimer des lignes *adjacentes* identiques dans un fichier.

Exemple

```
$ cat toto.txt
AAA
AAA
BBB
AAA
$ uniq toto.txt
AAA
BBB
AAA
```

La commande `tr` (*translate*) remplace une liste de caractères par une autre.

Exemple : `tr '[A-Z]' '[a-z]' < annuaire.csv`.

Information (wc)

La commande `wc` (*word count*) affiche le nombre de lignes, de mots et d'octets d'un fichier ou depuis l'entrée standard.

Exemple : `wc annuaire.csv`
retourne `4 6 83 annuaire.csv`.

Recherche (find)

La commande `find` permet de retrouver dans une arborescence un fichier (ou un ensemble de fichiers) satisfaisant une expression.
Options :

- `-name` (-iname) suivi du nom (et caractères spéciaux) du fichier à trouver
- `-perm` suivi des permissions (en octal)
- `-size` suivi de la taille du fichier
- `-exec` suivi de la commande à exécuter

Exemples :
```
find . -iname "a*" -type d
find $HOME ! -user $LOGNAME
```

Recherche (find)

Une suite de critères est considérée comme une intersection (« et »).

Exemple : `find . -size +100M -size -1G`

L'option `-o` permet une union (« ou ») de critères.

Exemple : `find . -name "*.py" -o -name "*.c"`

L'option `-exec` permet d'exécuter une commande sur les chemins trouvés qui sont symbolisés par `{}`, elle doit se terminer par `';'`.

💬 Pourquoi les quotes ?

Exemple :
```
find . -iname "*.txt" -exec wc -l {} ';'
       -exec cp {} {}.bak ';'
```

Recherche (find)

Une suite de critères est considérée comme une intersection (« et »).

Exemple : `find . -size +100M -size -1G`

L'option `-o` permet une union (« ou ») de critères.

Exemple : `find . -name "*.py" -o -name "*.c"`

L'option `-exec` permet d'exécuter une commande sur les chemins trouvés qui sont symbolisés par `{}`, elle doit se terminer par `';'`.

💬 Pourquoi les quotes ?

👍 Car le caractère spécial ; est le séparateur de commandes.

Exemple :
```
find . -iname "*.txt" -exec wc -l {} ';'
       -exec cp {} {}.bak ';'
```

Recherche dans des fichiers (grep)

La commande grep affiche toutes les lignes des fichiers (ou de l'entrée standard) contenant une chaîne de caractères.

Exemple : `grep 'print(' *.py`

Les motifs de recherche peuvent être des expressions régulières (un ensemble de chaînes de caractères possibles).

Exemple :
```
grep -i '^[ab].*e$' bash.csv
Alphonse,bash newbie.
```

En Bash, une « regexp » peut être employée depuis la version 3 après l'opérateur de comparaison `=~`.

Syntaxe des expressions régulières

- le point . : n'importe quel caractère,
- les crochets [] : ensemble de caractères,
  - le standard POSIX[12] 1003.2 définit les classes de caractères sous la forme [:nom_classe :], exemple : `[:lower:]`,
  - le caractère **!** (ou ^) placé juste après le caractère **[** indique la négation.
- les quantificateurs (nombre de répétitions) : * n'importe quelle occurence, ? 0 ou 1 occurence, + au moins une occurence,
- l'accent circonflexe ^ : début de ligne,
- le dollar $: fin de ligne,
- le pipe | : alternative,
- les parenthèses () : capture une séquence.

12. Portable Operating System Interface UniX.

Filtre programmable (awk)

La commande `awk` (Aho, Kernighan, Weinberger) est un outil de sélection et de manipulation de texte.

awk traite les lignes d'un fichier une par une et de manière séquentielle, chaque ligne est composée de champs ($0, $1, ...) séparés par des espaces.

awk dispose de variables et de commandes qui lui sont propres, il peut faire des calculs, effectuer des tests, afficher des résultats, ...

awk permet aussi de spécifier des blocs de pré-traitement (exécuté avant le traitement de la 1ère ligne, BEGIN) et post-traitement (exécuté après le traitement de la dernière ligne, END).

Filtre programmable (awk)

Exemple

```
$ cat informaticiennes.txt
Ada LOVELACE 1815 1852
Annie EASLEY 1933 2011
Grace HOPPER 1906 1992

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

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
```

Édition (sed)

La commande `sed` (*stream editor*) est un éditeur de texte non-interactif (substitution, ajout, insertion, modification).

L'utilisation principale de la commande sed est la substitution d'une expression régulière par une chaîne de caractères, syntaxe :

`s/expression-reguliere/chaîne/g`

L'option /g indique que toutes lignes validant l'expression régulière seront traitées, sinon le traitement s'arrêtera après la première occurrence trouvée.

Exemples :

```
sed 's/alice/bob/g' fichier1 fichier2
cat fichier | sed -e 's/a/A/g' -e 's/.*/# &/g'
sed '/^A.*e$/d' bash.csv
echo 1789-07-14 | sed -e 's/\(.*\)-\(.*\)-\(.*\)/
                          date: \3 \2 \1/'
```

Plan

Introduction

Le shell traite toutes les valeurs comme des données de type chaîne de caractères.

Exemple

```
$ i=1
$ i=$i+1
$ echo $i
1+1
```

Pour effectuer des calculs numériques, il y eu initialement la commande `expr` (portable) puis le bash a intégré la commande `let` (spécifique).

Commande externe expr

La commande expr prend en arguments les termes d'une expression arithmétique.

Exemple : `expr 1 + 2`

⚠ les espaces entre opérateur et opérandes sont indispensables.

Opérateurs dans l'ordre décroissant de priorité dans l'évaluation de l'expression :

Opérateurs	Opérations
\(expression \)	parenthésage
*	multiplication
/	quotient de la division entière
%	reste de la division entière
+	addition
-	soustraction

Substitution arithmétique

Cela consiste à évaluer une expression arithmétique et à la substituer par le résultat de cette évaluation. Syntaxe :

```
$(( expression ))
```

C'est un mécanisme interne au bash, plus simple que la commande `expr` (pas d'espace, pas de substitution explicite, pas de caractères spéciaux).

Exemple :

```
$ a=1 ; b=2
$ c=$((a+b))
$ echo $c
3
```

Substitution arithmétique

Opérateurs arithmétiques dans l'ordre de priorité décroissant d'évaluation (idem langage C) :

Opérateurs	Opérations
(expression)	parenthésage
++, −−	incrémentation, décrémentation
!	négation logique
**	puissance
*, /, %	multiplication, quotient, reste
+, −	addition, soustraction
==, !=, <, >, <=, >=	comparaison
&&	et logique
||	ou logique
=, +=, -=, *=, /=, %=	affectation

La commande (( )) (ou `let`)

Cette commande évalue une expression arithmétique mais ne retourne que le code d'erreur (💬 qui est ?).

Exemple

```
$ i=1
$ ((i=i+1))
$ let i+=1
$ ((i++))
$ let 'i<1'
```

Cette commande est plus simple à utiliser que la commande `expr` car elle reprend la syntaxe de nombreux langages de programmation.

⚠ ne pas confondre la commande (( )) avec la substitution arithmétique $(( )).

Calculs de nombres décimaux

Le calcul scientifique nécessite des commandes spécifiques : `awk` et `bc` (*basic calculator*). Cette dernière est l'équivalent d'une calculatrice programmable.

Par défaut, `bc` s'exécute en mode interactif. Pour effectuer des calculs à partir du shell, il faut utiliser `bc` comme un filtre.

Exemple

```
$ a=10
$ echo "scale=5 ; $a/3" | bc
3.33333
```