Filtres et parcours récursifs

1 Généralités

Le langage de commandes d'un système d'exploitation est un langage interprété souvent qualifié de langage de scripts. De tels langages partagent plusieurs points communs :

- ce sont des langages interprétées orientés « ligne » : la ligne (notion logique) est l'unité d'interprétation ;
- ce sont des langages de programmation complets (parfois complexes) ;
- leur domaine d'usage est la programmation « dans le large » : les commandes et scripts sont la plupart du temps des traitements complexes sous forme de processus et portent sur des opérandes qui sont des objets de taille souvent importante, en l'occurrence des fichiers (programmes ou données). Les scripts permettent par exemple de décrire un traitement répétitif sur une arborescence de fichiers, d'assembler un ensemble de programmes pour qu'ils s'exécutent en parallèle et communiquent entre eux, de traiter des données dans des fichiers par des opérations de type « base de données » (par exemple, un tri).

2 Les langages de script de la famille shell

Les shells sont une famille de langages interprétés proposés comme langages de commandes (de scripts) pour le système Unix. Les différentes versions de langage shell ont des syntaxes souvent similaires mais différentes. La suite de ce document s'attache à présenter la syntaxe d'un shell que l'on retrouve sur les systèmes actuels de la famille Unix, en l'occurrence le `bash` (Bourne Again Shell) issu du projet `GNU`.

La syntaxe d'une commande simple est la suivante :

$$< commande > \quad [< options >] \quad [< arguments >]$$

Le nom d'une commande désigne soit une commande interne du langage (par exemple cd), soit un fichier dont le contenu doit alors être soit un script shell, soit un programme binaire exécutable [1]. Dans ce deuxième cas, l'exécution de la commande « externe » revient donc à exécuter un nouveau processus comme illustré dans la figure 1. S'il s'agit d'un script, le nouveau processus créé exécute l'interpréteur shell avec en entrée le fichier script <*commande*>. S'il s'agit d'un programme binaire exécutable, le processus exécute ce programme.

Cette approche assure une grande extensibilité au langage puisque tout script ou programme exécutable est potentiellement une nouvelle commande acceptable, exécutable pour l'interpréteur de commande. Mais c'est aussi un principe naturel puisqu'un interpréteur de commandes a pour rôle de base d'exécuter les programmes des usagers.

Pour l'interpréteur de commandes, une commande est en fait interprétée comme une simple suite de chaînes de caractères séparées par des blancs ou tabulations. Parmi elles, l'interpréteur reconnaît les opérateurs de contrôle et les opérateurs de redirection.

1. Dans les 2 cas, les droits d'accès au fichier doivent autoriser l'exécution au processus interpréteur courant

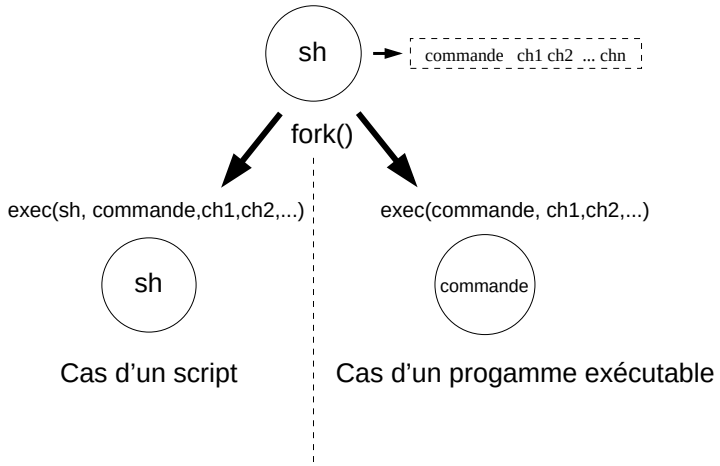


Figure 1 – Exécution d'une commande externe

2.1 Opérateurs de contrôle :

Les opérateurs de contrôle permettent de spécifier comment les commandes d'exécuteront : séquentiellement, parallèlement, en tâches de fond, conditionnellement, etc. On trouve :

&	&&	(	)	{	}	\n	;;	;	|	||

Plus précisément, la sémantique de ces opérateurs est détaillée dans le tableau suivant :

com &	Exécution de la commande en processus de fond (non interactif)
com1 ; com2	Exécution séquentielle des commandes quel que soit leur terminaison.
com1 com2	Exécution séquentielle de com1 puis de com2
com1 && com2	Exécution séquentielle des commandes jusqu'à ce qu'une commande s'exécute mal.
com1 | | com2	Exécution séquentielle des commandes jusqu'à ce qu'une commande s'exécute bien.
(liste de commandes)	Exécution de la liste de commandes comme un bloc. Attention, les variables définies par affectation dans le bloc sont locales
{ liste de commandes }	Exécution de la liste de commandes comme un bloc. Attention : les variables référencées ou affectées sont globales
com1 | com2	Exécution en parallèle de com1 et com2 avec connexion par pipe

Environnement d'exécution d'une commande Une commande étant un processus (créé pour exécuter la commande ou, pour les commandes internes, le processus shell lui-même), celle-ci s'exécute avec un environnement standard de communication par flots de données, en l'occurrence :

- `stdin` : flot d'entrée standard associé au canal numéro 0 et connecté par défaut à la ressource clavier ;
- `stdout` : flot de sortie standard associé au canal 1 et connecté par défaut à la fenêtre de lancement de la commande ;
- `stderr` : flot de sortie standard associé au canal 2 et connecté par défaut à la fenêtre de lancement de la commande.

Ces connexions implicites peuvent être modifiées par le mécanisme de redirection (voir 2.2).

Par ailleurs, un ensemble de variables globales de l'interpréteur shell peut intervenir dans le déroulement d'une commande. À titre d'exemple, la variable `PATH` contient une suite de noms de répertoires séparés par le caractère `:`. Cette variable intervient lors de la recherche même du fichier correspondant à la commande puisque l'ensemble des répertoires présents dans la liste spécifiée par la variable `PATH` peut être parcouru pour y trouver finalement le fichier demandé.

Ces variables dites d'environnement peuvent varier d'un shell à l'autre. En `bash`, les principales sont :

- `HOME` : spécifie le chemin du répertoire personnel de l'utilisateur
- `PATH` : spécifie la liste des chemins de recherche des commandes exécutables
- `PPID` : spécifie le PID du processus père du shell
- `PS1` à `PS4` : spécifie les invites du shell (principal, secondaire, ...
- `PWD` : spéficie le chemin du répertoire courant
- `UID` : spécifie le numéro de l'usager associé au processus shell

Le résultat d'une commande À l'exécution de toute commande est associé un code de terminaison. En effet, une commande peut se terminer par un appel superviseur de fin normale ou anormale (primitive exit) ou à la suite d'une exception : interruption ou déroutement. Pour l'interpréteur shell, seule une commande se terminant par l'appel de la primitive de terminaison (exit) **avec en opérande la valeur 0** est considérée comme une terminaison correcte et interprétée par une valeur booléenne vraie. Dans tous les autres cas, l'interpréteur considère que la commande a eu une terminaison incorrecte et interprète sa terminaison par une valeur booléenne fausse. Toute exécution de commande engendre donc une valeur booléenne utilisable, en particulier pouvant être testée (voir structures de contrôle). Les commandes internes (built-in) ont aussi un résultat booléen.

Il est de plus possible de préfixer toute commande par l'opérateur booléen **non** dénoté par le caractère `!`. Par exemple, l'interpréteur évalue la ligne de commande `! <com>` à vrai si la terminaison de la commande est incorrecte et inversement à faux si la termaison de la commande est correcte.

2.2 Opérateurs de redirection :

Les opérateurs de redirection modifient l'environnement d'exécution de la commande (≡ du processus associé à la commande). Les flots de données d'entrée et de sortie peuvent être connectés à des ressources spécifiques (pipes par exemple).

< << > >>

Sémantique des opérateurs de redirection

- `commande <fichier` : provoque la redirection de l'entrée standard sur le fichier. Toute lecture de l'entrée standard équivaudra donc à lire le fichier spécifié.
- `commande <<`*mot* : permet de lire des lignes de données sur l'entrée standard jusqu'à une ligne égale à la chaîne de caractère spécifiée *mot*. La lecture de cette ligne « finale » provoque la fin du flux d'entrée mais n'en fait pas partie.
- `commande >fichier` : provoque la redirection de la sortie standard sur le fichier. Toute écriture sur la sortie standard équivaudra donc à écrire dans le fichier spécifié. Si le fichier existe déjà, son contenu sera écrasé par les nouvelles écritures.

— `commande >>fichier` comme précédemment mais si le fichier existe déjà, les écritures se feront après la fin du fichier existant (concaténation).

Si un numéro est précisé avant le symbole de redirection, c'est le flot d'entrée ou sortie correspondant qui est redirigé.

On peut aussi spécifier, au lieu d'un fichier, un numéro de flot : il faut alors faire précéder le numéro de flot par un &.

Exemples

— `commande 2>>fichier` : provoque la redirection de la sortie standard des erreurs (flot n°2) vers le fichier spécifié.

— `commande 2>> &1` : provoque la redirection de la sortie standard des erreurs (flot n°2) sur la sortie standard (flot n°1).

2.3 Les variables

Comme tout langage de programmation, le langage shell permet l'usage de variables. Il n'existe qu'un seul type de variables : le type chaîne de caractères. Cependant, la valeur de cette chaîne, dans le contexte d'usage d'une commande pourra être interprétée différemment : comme un nom de fichier, comme un nombre entier ("1234"), comme un booléen ("0" est considéré comme la valeur vraie par exemple, etc)

Syntaxe de base de l'affectation : <nom de variable>=<valeur> [2].

Exemples : `x=12 ; f=projet.tex ; liste="a b c d e"`

L'utilisation des guillemets est nécessaire pour que les espaces ne soient par interprétés comme des fins de chaîne.

Syntaxe d'une référence : $<nom de variable> [3]

Exemples : `echo $x ; ls -l $f ; x=$x$liste ; echo $x`

Cette séquence de commandes aura pour résultat visible, compte tenu des affectations des exemples précédents :

```
12
-rw-r--r--  1 toto  staff  30  5 Apr 15:55 projet.tex
12a b c d e
```

2. Attention : pas d'espace à gauche et à droite de l'opérateur =
3. De nombreuses variantes existent.

2.4 Mots clés

L'interpréteur reconnaît aussi un ensemble de mots clés (ou réservés) auxquels il donne une signification particulière s'ils sont en début de ligne ou après un caractère de contrôle.

!	{	}	case	esac
if	then	elif	else	fi
for	until	while	do	done

On peut remarquer que les mots réservés correspondent essentiellement aux mots clés utilisés pour les structures de contrôle du langage. En shell, les structures de contrôle sont traitées comme des commandes internes et les mots clés correspondant doivent donc apparaître toujours en début de ligne ou après un opérateur de contrôle.

Exemple

```
if [ -f $quelfich ]
then echo le fichier $quelfich existe
else touch $quelfich
fi
```

2.5 L'interprétation des lignes de commandes

L'interpréteur de commande propose toujours un ensemble de fonctions destinées à faciliter la saisie des lignes de commande : historique, complétion automatique des noms de fichiers, abréviation des noms de fichiers grâce aux chemins relatifs, possibilité de définir un ensemble de noms de fichiers par un motif (métacaractères)... Les lignes de commande saisies vont dont être interprétées, c'est-à-dire

- analysées afin de déterminer les différents opérandes et opérateurs de composition figurant dans la ligne, et de traiter successivement chacune des commandes élémentaires ;
- transformées afin de remplacer les formes/noms abrégés par des formes/noms complets.

L'interprétation d'une ligne de commandes s'exécute comme une suite séquentielle de passes d'expansion consistant à substituer des caractères. En particulier, les méta-caractères vont être interprétés lors de ces transformations. Après une identification des opérateurs de contrôle conduisant à une décomposition en commandes élémentaires de la ligne de commande(s), des passes d'expansion sont donc appliquées à chaque commande élémentaire dans l'ordre suivant :

1. expansion des accolades, puis du tilde (`~`),
2. substitution des paramètres et des variables,
3. substitution des commandes entre anti-apostrophes (`` ` ``) ,
4. évaluation arithmétique,
5. évaluation des méta-caractères `*`, `?`, `[` et `]` pour l'expansion des chemins d'accès,
6. élimination des occurrences des caractères \, `'` et `"` qui n'ont pas été engendrés par les expansions précédentes.

Après quoi, les redirections sont traitées.

L'expansion des accolades

Les accolades permettent d'engendrer des mots à partir d'une suite de mots séparés par des virgules.

Quelques exemples :

avant expansion	après expansion
`em{port,ball}er`	`emporter emballer`
`fich{a{1,2},b}`	`ficha1 ficha2 fichb`

L'expansion du ~

Le caractère `~` est remplacé par le contenu de la variable `HOME`.

Substitution des paramètres et variables

Le méta-caractère `$` est interprété comme un ordre de substitution du numéro de paramètre ou du nom de variable qui suit par sa valeur.

Substitution des commandes entre anti-apostrophes (`)

Le fait de placer une (suite de) commandes entre anti-apostrophes provoque leur exécution. Le résultat produit sur la sortie standard remplace la chaîne origine. Quelques exemples :

avant exécution	après exécution
`` `pwd` ``	*résultat de l'exécution de la commande, par exemple* `/Users/...`
`` x=`ls` ``	x=*chaîne contenant les noms de fichiers du répertoire courant séparés par un espace*

En `bash`, la syntaxe suivante est équivalente : `$(`<*commande*>`)`. Par exemple, `$(pwd)` ≡ `` `pwd` ``.

L'évaluation arithmétique

En `bash`, il est possible de faire quelques opérations élémentaires sur des entiers grâce à la syntaxe suivante : `$((`<*expression*>`))`.

avant évaluation	après évaluation
`a=$((1+2))`	`a=3`
`echo $(($a-4))`	`echo -1`

La commande utilitaire `expr` (voir 5.2) permet aussi de faire des calculs élémentaires.

Traitement des méta-caractères * et ?

La plupart des commandes comportent un ou plusieurs noms de fichiers. Les méta-caractère `*` et `?` permettent de construire des filtres pour choisir un sous-ensemble de noms de fichiers dans un répertoire. Le méta-caractère `*` est substituable par n'importe quelle sous-chaîne (éventuellement vide) et le caractère `?` par n'importe quel caractère (réel).

Exemples

avant expansion	après expansion
`ls *`	*liste des noms du répertoire courant*
`ls ./projet/??*.doc`	*liste des noms du répertoire* `./projet/` *d'au moins 2 lettres et de suffixe* `.doc`

Suppression de l'interprétation des méta-caractères

L'interprétation des méta-caractères peut être inhibée, supprimée en particulier lorsque des valeurs de paramètres ou variables doivent contenir des maté-caractères. L'exemple de la commande `eval` est un exemple d'une telle situation. La caractère \ permet d'empêcher l'interprétation du méta-caractère suivant.

Plus globalement, une sous-chaîne contenant des méta-caractères peut être mise entre apostrophes `'` ou guillemets `"`. Une différence essentielle entre les deux solutions : les apostrophes supprime l'interprétation du méta-caractère `$` contrairement aux guillemets. Par exemple :

`a=valeur ; echo '$a'` a pour résultat `$a`
`a=valeur ; echo "$a"` a pour résultat `valeur`

3 Liste des commandes internes

Un certain nombre de mots « clés » désignent les commandes internes de l'interpréteur. Les commandes internes sont celles exécutées directement par l'interpréteur lui-même, donc sans création de processus fils (commandes externes). Ce sont des commandes de base telles que celle permettant de changer de répertoire courant (`cd`), de faire l'écho d'une chaîne de caractères ou d'afficher le répertoire courant (`pwd` sur la sortie standard, mais aussi les constituants des structures de contrôle tels que `if`, `then`, `while`, `do`, etc

La liste complète pour le `bash` est la suivante :

`alias, bg, break, case, cd, chdir, command, continue, do, done, echo, elif, else, esac, eval, exec, exit, export, false, fc, fg, fi, for, getopts, hash, if, jobs, login, logout, nice, popd, printf, pwd, read, readonly, rehash, repeat, set, setvar, shift, switch, test, then, trap, true, type, ulimit, umask, unalias, unset, until, wait, while`

4 Parcours d'une arborescence : find

Le parcours d'une arborescence de fichiers s'appuie sur la commande find.

find $< chemin\ d'accès > ...$ $< expression >$

Une expression est construite à partir de termes élémentaires de la forme - option [param] ayant un résultat booléen et composables via les opérateurs logiques classiques notés -or (ou), -and (et) et -not ou ! pour le non. Des parenthèses peuvent être aussi utilisées pour fixer l'ordre d'évaluation des opérations [4].

L'expression débute par le premier argument trouvé commençant par -, ou ! ou (. La commande find exécute un parcours récursif des arborescences de fichiers dont la ou les racines sont spécifiées par leur chemin d'accès en cherchant les fichiers qui vérifient l'expression booléenne.

Principales options

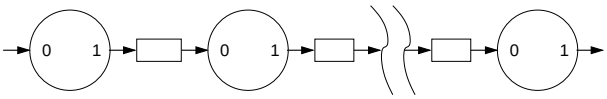
-name $<motif>$ vrai si le fichier opérande courant vérifie le motif ;
-print toujours vrai et provoque l'affichage sur stdout du nom de fichier opérande ;
-exec <*commande*> <*argument*> ... \ ; vrai si l'exécution de la commande réussit. Le fichier opérande peut être désigné dans la liste d'argument par {} ;
-depth toujours vrai et provoque un parcours en profondeur d'abord ;
-newer <*chemin d'accès*> vrai si le fichier opérande est plus récent que le fichier spécifié par son chemin d'accès ;
-size [+]n[c] vrai si la taille du fichier opérande est de n blocs (de 512 octets) ou n caractères (option **c** ; la présence d'un + indique que l'on cherche un fichier de taille supérieure à n blocs ou caractères.
-type t vrai si le fichier opérande est t :
- b fichier spécial d'échange par blocs ;
- c fichier spécial d'échange par caractères ;
- d fichier répertoire ;
- f fichier ordinaire ;
- l lien symbolique.

Exemple : find . -name "*.tex" -size +0 -print : provoque le parcours de toute l'arborescence de fichiers ayant pour racine le répertoire courant (dénoté par .) et liste tous les chemins d'accès aux fichiers non vides ayant le suffixe tex.

5 Filtrage

La notion de filtrage consiste à soumettre un flot de données d'entrée à une succession de traitements exécutés par des processus parallèles selon un schéma de type pipeline. Chaque traitement élémentaire consomme un flot d'entrée lu, par défaut, sur son entrée standard et produit un flot de sortie, par défaut, sur sa sortie standard. La liaison entre 2 traitements consécutifs est assurée par pipe.

4. Attention : ces parenthèses doivent rester non interprétées pour être transmises à la commande find en tant qu'arguments par l'usage du symbole \. À titre d'exemple, on peut écrire : find . \(-size 0 -or -name "*.c" \)



Exemple : cat f1 f2 f3 | sort : le contenu des fichiers f1,f2,f3 est produit en sortie par la commande cat et ce flot de lignes est trié par la commande sort. Le résultat est donc un flot de sortie contenant les lignes triées des 3 fichiers.

5.1 Quelques commandes de filtrage

• Concaténation et listage de fichiers

```
cat   [-]   [-nbstv]   [fichier...]
```

Options

- `-` lecture sur l'entrée standard
- `n` listage avec numérotation des lignes
- `b` listage avec numérotation des lignes sauf les lignes vides
- `s` listage avec remplacement d'une suite de lignes vides par une seule
- `v` listage avec affichage des caractères de contrôle sous la forme `^X` sauf `TAB` et `NEWLINE`.

Exemples

`cat f1 f2`	listage des fichiers `f1` et `f2` sur la sortie standard
`cat -n f1 f2 >f3`	listage des fichiers `f1` et `f2` dans `f3` avec numérotation des lignes
`cat f1 f2 f3 >f1`	équivalent à `cat f2 f3 >f1` (écrase le fichier `f1`)
`cat - >f`	création du fichier `f` qui contiendra les lignes frappées sur l'entrée standard jusqu'à la frappe d'un caractère «fin de fichier» (`CTRL-D`)

• Tri de fichier(s)

```
sort   [-bdfr] [-k POS1[,POS2]]  fichier...
```

Par défaut, le tri se fait sur le contenu global de la liste de fichiers en arguments et sur les lignes complètes. Une zone clé peut être spécifiée via l'option `k`.

Options

`b`	ignorer les espaces en début de ligne
`d`	trier selon l'ordre lexicographique
`f`	ignorer la distinction entre majuscules et minuscules
`r`	trier dans l'ordre inverse ;
`k POS1[,POS2]`	trier selon la zone située dans la ligne entre les positions `POS1` et `POS2`.

Exemples

`sort f1 f2`	tri du contenu global des fichiers `f1` et `f2` sur la sortie standard ;
`sort -r f1 f2`	tri des fichiers `f1` et `f2` ;
`sort -k 10,20 f1`	tri du fichier `f1` en utilisant pour clé la zone de `POS1` à `POS2`.

• Filtrer des lignes de textes

`grep motif [fichier]`

La commande `grep` lit un flot de lignes sur l'entrée standard ou dans le fichier spécifié et extrait les lignes qui satisfont le motif spécifié sous forme d'expression régulière. Les lignes extraites sont envoyées sur la sortie standard.

Quelques éléments sur la syntaxe des expressions régulières utilisables Un ensemble de méta-caractères sont reconnus :

`.`	caractère "joker"
`^`	début de ligne ou "sauf" après [
$	fin de ligne
`[`<*chaîne*>`]`	un caractère parmi ceux de la chaîne spécifiée

Facteurs de répétition Chaque métacaractère peut être suivi d'un facteur de répétition :

`*`	répétition de 0 à un nombre indéfini de fois
$\backslash\{m,n\backslash\}$	répétition de m à n de fois

Exemples

motif	signification
`^abc`	extrait toutes les lignes commençant par la chaîne `abc`
`^abc$`	extrait toutes les lignes réduites à la chaîne de 3 caractères `abc`
`a.*aa`	extrait toutes les lignes qui contiennent une sous-chaîne commençant par un `a` et se terminant par `aa`
`[a-z][^:;]*`	extrait toutes les lignes qui contiennent une sous-chaîne commençant par une lettre minuscule suivie de n'importe quel caractère sauf `:` et `;`

• Listage des dernières lignes d'un fichier

`tail [-r][-b|-c|-n [+|-]`α`] [fichier]`

Options

`+`α	la position α est comptée à partir du début
`-`α	la position α est comptée à partir de la fin
`-n`	α désigne un nombre de lignes
`-b`	α désigne un nombre de blocs de 512 octets
`-c`	α désigne un nombre d'octets
`-r`	listage des lignes dans l'ordre inverse

Le signe par défaut est −

La position par défaut est `-n 10`

Exemples

`tail`	`f1`	listage des 10 dernières lignes du fichier `f1`
`tail`	`-n -1 f1`	listage de la dernière ligne du fichier `f1`
`tail`	`-n +2 f1`	listage du fichier `f1` sauf sa première ligne (numérotée 1)
`tail`	`-c 100 f1`	listage des 100 derniers octets du fichier `f1`
`tail`	`-r f1`	listage du fichier `f1` avec les lignes dans l'ordre inverse
`tail`	`-rn 10 f1`	listage des 10 dernières lignes du fichier `f1` dans l'ordre inverse

• Listage du début d'un fichier

`head [-c|-n` α`] [fichier...]`

Options

`-n`	α désigne un nombre de lignes
`-c`	α désigne un nombre d'octets

La position par défaut est `-n 10`.

Exemples

`head`	`f1`	listage des 10 premières lignes (si elles existent) du fichier `f1`
`head`	`-n 100 f1 f2`	listage des 100 premières lignes de chaque fichier `f1` et `f2` (si elles existent). Le listage commence par une ligne d'en-tête de la forme ==> nom du fichier <==

• Fusion de lignes de fichiers

`paste [-d list] fichier...`

Par défaut, `paste` fusionne les fichiers spécifiés ligne par ligne. Autrement dit, la ième ligne produite sur la sortie standard est la concaténation de la ième ligne de chaque fichier avec le caractère `NEWLINE` de chacune de ces lignes remplacé par une tabulation (`TAB`) excepté pour la ligne du dernier fichier. Si un fichier n'a pas de ième ligne, une ligne vide est introduite réduite donc à son séparateur (`TAB` par défaut).

Option

`-d list`	permet de spécifier le(s) séparateur(s) de ligne qui remplace(nt) les `NEWLINE` les caractères spécifiés sont pris circulairement comme séparateurs

Exemple

`paste -d : f1 f2 f3` produit des lignes issues de la fusion des lignes des 3 fichiers avec des deux points comme séparateur.

• Sélection de zones d'une ligne de fichier

```
cut -[b|c|f] pos,... [-d list] fichier
```

La commande `cut` sélectionne des champs dans chaque ligne pour les lister sur la sortie standard. Une zone est repérée par un numéro α ou un intervalle α-β. La numérotation commence à 1.

Options

`-b`	sélection de zones en octets
`-c`	sélection de zones en caractères
`-f`	sélection de zones par champs
`-d list`	permet de spécifier le(s) séparateur(s) de ligne qui remplace(nt) les `NEWLINE`

Exemples

`cut -f1,3-5 f1`	sélectionne les champs 1, 3,4 et 5 de chaque ligne du fichier `f1`
`cut -c10-20 f1`	sélectionne les caractères de 10 à 20 de chaque ligne du fichier `f1` (s'ils existent).

• Substitution de caractères

```
tr [-dcs] chaîne1 [chaîne2]
```

Par défaut, `tr` recopie sur la sortie standard ce qui est lu sur l'entrée standard en substituant chaque caractère présent dans la première chaîne par le caractère de la seconde chaîne présent dans la même position. Les caractères de la première chaîne qui n'ont pas de correspondants dans la seconde chaîne (celle-ci est plus courte que la première) sont substitués par le dernier caractère de la seconde chaîne.

Abréviation : la chaîne `a-d` équivaut à la chaîne `abcd`

Options

`-d`	supprimer toutes les occurrences de caractères de la chaîne «chaîne1»
`-c`	remplacer tous les caractères qui **ne sont pas** dans la chaîne «chaîne1»
`-s`	remplacer les suites d'occurrences du même caractère par un seul

Exemples

`cat f1 | tr aeiouy 123456`	listage du contenu du fichier `f1` avec toutes les voyelles remplacées par des chiffres
`cat f1 | tr aeiouy '.'`	listage du contenu du fichier `f1` avec toutes les voyelles remplacées par des points
`cat f1 | tr -dc aeiouy`	listage du contenu du fichier `f1` réduit aux voyelles qu'il contenait
`cat f1 | tr -cs 'e\n' '.'`	listage du contenu du fichier `f1` avec chaque suite de caractères autres que 'e' et `NEWLINE` remplacée par un '.'
`find . -print | tr -dc '/\012'`	listage de lignes contenant autant de caractères slash que dans les chemins d'accès produits par `find`

• **Comptage de caractères, octets, mots, lignes**

`wc [-clmw] [fichier ...]`

La commande `grep` lit par défaut un flot de lignes sur l'entrée standard ou dans les fichiers spécifiés. Elle produit sur la sortie standard le nombre de lignes, mots, caractères ou octets lus selon les options (par défaut, les lignes, les mots et les caractères).

5.2 Quelques commandes utiles

• **La commande `read`**

Cette commande permet de lire une (seule) ligne sur l'entrée standard. La ou les variables reçoivent les valeurs des mots lus. S'il en manque, les variables finales prennent pour valeur la chaîne vide.

`read` *variable ...*

Exemple :

```
>read x y z
premier second
>echo $x$z$y
premiersecond
```

• **Tests divers : sur les attributs d'un fichier, sur des variables**

`test expression` ou `[ expression ]`

Cette commande permet de tester diverses conditions sur les valeurs des variables et sur les fichiers. Les différents éléments de l'expression doivent être séparés par des blancs. La commande renvoie vrai (c'est-à-dire un code de retour égal à 0) si la condition est vérifiée.

Une expression est une suite de termes séparés soit par des opérateurs logiques : non noté !, le ou noté $-o$, le et noté $-a$, soit par des parenthèses notées \(et \).

Les différents termes d'une expression peuvent être :

chaîne-1 $\left[\begin{array}{c} = \\ != \end{array} \right]$ chaîne-2	comparaison de chaînes
nombre-1 $\left[\begin{array}{c} -eq \\ -ne \\ \cdots \end{array} \right]$ nombre-2	comparaison de chaînes représentant des entiers
$\left[\begin{array}{c} -r \\ -w \\ -x \end{array} \right]$ fichier	vrai si le fichier est accessible pour le droit d'accès spécifié
$\left[\begin{array}{c} -f \\ -d \end{array} \right]$ fichier	vrai si le fichier est ordinaire (f) ou un répertoire (d)
$\left[\begin{array}{c} -z \\ -n \end{array} \right]$ chaîne	vrai si la chaîne est vide (z) ou non (n)

Exemples

`[ -r $f -o -z $x ]`	vrai si le fichier désigné par `f` est lisible ou si la variable `x` est vide
`[ $x -eq 1 ]`	vrai si la valeur de la variable `x` est égale à 1

• Evaluer une expression, extraire une sous-chaîne,. . .

`expr expression`

Les expressions prennent des opérandes et produisent des résultats de type chaîne de caractères. Mais, pour les opérations arithmétiques, il faut naturellement que ces chaînes représentent des entiers. Les différentes formes d'une expression peuvent être :

`expr1 | expr2`	produit sur la sortie standard le résultat de l'exécution de `expr1` si celui-ci n'est pas égal à une chaîne vide ni à zéro. Sinon, produit le résultat de l'exécution de `expr2`.
`expr1 & expr2`	produit sur la sortie standard le résultat de l'exécution de `expr1` si aucune des 2 expressions n'est égale ni à une chaîne vide ni à zéro.
`expr1 [+|-|*|%|/] expr2`	opérations arithmétiques si les opérandes sont des nombres
chaîne `: expr1`	recherche la plus longue sous-chaîne commençant en début de *chaîne* et appariée à l'expression régulière `expr1`. Si `expr1` contient un sous-motif entre parenthèses le résultat est la sous-chaîne appariée au sous-motif sinon le résultat est la longueur de la sous-chaîne appariée.

Exemples

`a=‘expr $a + 1‘`	addition de 1 à la variable `a`
`expr $a : '.*/\(.*\)'`	si `a` contient un chemin d'accès `d1/d2/fich`, renvoie `fich`
`expr $a : '.*'`	renvoie le nombre de caractères contenus dans `a`

Rappel L'interpréteur shell traite les valeurs des variables comme des chaînes de caractères. La commande précédente expr est une possibilité pour faire quelques calculs arithmétiques simples.

En bash, on peut aussi effectuer des calculs sur des nombres entiers en utilisant la syntaxe $((...)) pour délimiter les expressions arithmétiques (voir 2.5). On notera qu'il n'y a pas besoin d'espace entre opérandes contrairement au cas des opérandes d'une expression dans une commande `expr`.

• La commande `eval`

Cette commande permet de préparer une ligne de commande dans une variable (en tant que valeur de cette variable) et de provoquer une passe d'évaluation complète (expansion des accolades, des $, etc) en référençant cette variable en opérande de la commande `eval`.

`eval $`*`variable`*

Exemple

`a=ls ; x='$a *.php' ;` alors `eval $x` ≡ `ls *.php`

Les méta-caractères contenus dans la chaîne représentant la valeur de la variable `x` sont interprétés après substitution de la référence à la variable par sa valeur. Sans la commande `eval`, la référence à `x` aurait bien été substituée par sa valeur mais sans passe ultérieure d'interprétation de ce résultat.

6 Exercices

1. Afficher le nom de tous les fichiers présents dans la sous-arborescence du répertoire courant possédant le suffixe .o.
2. Compter le nombre de fichiers avec l'extension .o présents dans la sous-arborescence du répertoire courant.
3. Supprimer tous les fichiers core non vides présents dans la sous-arborescence du répertoire courant.
4. Calculer la somme du nombre de caractères présents dans tous les fichiers ordinaires de l'arborescence issue du répertoire courant.
5. Compter le nombre de lignes où apparaît une chaîne donnée dans tous les fichiers du répertoire courant.
6. Compter le nombre de lignes contenant `bash` parmi les 20 premières lignes du fichier `/etc/passwd`
7. Lister les caractéristiques du troisième objet (objet = fichier ou répertoire) du répertoire courant dans l'ordre chronologique de modification.
8. Compter le nombre de voyelles dans un fichier donné.
9. Compter le nombre d'occurrences d'une chaîne (sans espace) dans un fichier. On ne compte qu'une occurrence du mot recherché par mot du fichier (ba dans "baba babababa ba" apparaît trois fois), et il peut se trouver plusieurs occurrences du mot recherché sur une même ligne. Le délimiteur entre les mots du fichier est l'espace.
10. Lister pour tous les répertoires contenus dans le répertoire courant les informations suivantes : nom du propriétaire, nom du répertoire, date de dernière modification.
11. Calculer la longueur de la plus longue ligne d'un fichier donné.