ÉCOLE D'INGÉNIEURS EN INFORMATIQUE

Piscine : journée 1 — TP

version nº 2

Assistants C/Unix 2014 <acu@epita.fr>

Copyright

Table des matières

*. https://acu.epita.fr

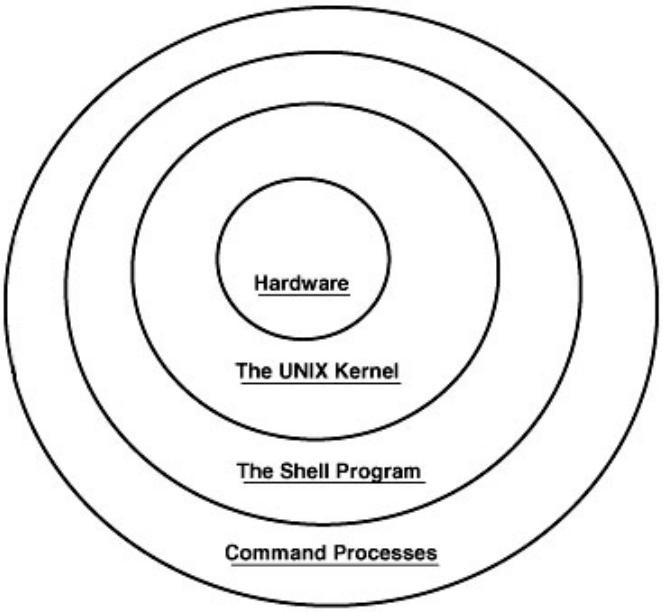
1 Cours

1.1 Qu'est-ce que le shell ?

1.1.1 Définition

Pour utiliser un ordinateur, vous passez par un système d'exploitation. Le cœur d'un tel système est son **noyau** (kernel). Le noyau est une entité chargée de traiter les requêtes des utilisateurs permettant ainsi d'accéder aux ressources matérielles et aux services fournis par votre système. Cependant, un noyau seul n'est pas suffisant pour utiliser un ordinateur. Il faut ajouter un programme proposant à l'utilisateur une interface simple lui permettant de transmettre ses requêtes au noyau du système. Grâce à ces commandes adressées au noyau, l'utilisateur pourra par exemple consulter la liste des fichiers sur son disque dur, démarrer ses applications, etc. C'est ce programme qu'on appelle un **shell**.

Les systèmes d'exploitation de la famille UNIX utilisent souvent une architecture en couches comme le montre la figure ci-dessous. Le mot anglais *shell*, a été choisi car le shell est une interface permettant d'accéder aux services du noyau. C'est une sorte de coquille autour du noyau.



1.2 Historique

L'histoire du shell sous sa forme actuelle se confond avec celle d'UNIX. Le shell des débuts était un programme plutôt rudimentaire : le Bourne Shell (du nom de son créateur : Stephen Bourne). Il s'agit d'un interpréteur en ligne de commande. Son originalité était de pouvoir chaîner les sorties de commandes aux entrées d'autres à l'aide de *pipes* (ou tubes).

Aujourd'hui le Bourne shell classique est dépassé et des évolutions comme **bash** se sont répandues dans les systèmes UNIX. Cependant, le Bourne shell a été standardisé par des spécifications comme la *SUS* (Single UNIX Specification) et on en trouve toujours un équivalent dans les systèmes UNIX modernes.

Les langages (informatiques) utilisés par les différents shells ne sont pas tous compatibles entre eux. Ainsi, si vous voulez écrire des scripts shells portables sur tous les systèmes UNIX, il est recommandé d'utiliser la syntaxe standardisée du Bourne shell et de n'utiliser que les commandes de base du monde UNIX.

1.3 Variantes

Le Bourne Shell est le shell historique des systèmes *UNIX*, et hérite des idées initiées avec *Multics* (fin des années 1960). Le C Shell (Csh) est le shell traditionnel des systèmes *BSD* (Berkeley Software Distribution). Le Csh utilise une syntaxe plus proche du C et innove avec les *Jobs Controls* ; cette fonctionnalité sera reprise par le Bourne Shell et ses descendants.

En 1988 apparaît le Ksh, qui s'inspire des deux précédents (la syntaxe du Bourne Shell et les *Jobs Controls* du Csh). Bash (écrit en 1987) est le shell de base d'une majorité d'UNIX (Linux, MacOSX, Solaris, etc). Il reprend les fonctionnalités de ses prédécesseurs et dépend du projet GNU.

Zsh est un shell plus récent, qui date de 1990 et s'inspire des précédents en ajoutant de nouvelles fonctionnalités telles que la complétion améliorée des commandes, des options et des arguments, l'historique partagé, la correction des fautes de typographie…

TCsh est le Shell des systèmes BSDs, il s'agit d'une amélioration de Csh.

2 Script Shell

2.1 Qu'est ce qu'un script shell ?

Vous avez déjà utilisé votre shell en entrant des commandes manuellement. Parfois, les traitements que vous aurez à effectuer seront complexes et pourront nécessiter plusieurs commandes. Si vous êtes amenés à réeffectuer plusieurs fois cette même opération, il deviendra très vite fastidieux de ressaisir manuellement les commandes à chaque fois.

Pour résoudre ce genre de problèmes, il existe une pratique très répandue dans le monde UNIX : les **scripts shell**. Un script shell n'est ni plus ni moins qu'un fichier dans lequel vous écrivez toutes les commandes nécessaires pour réaliser votre traitement. Par la suite, ce fichier pourra être relu par votre shell qui exécutera alors les commandes les unes à la suite des autres. On peut donc considérer un script shell comme un programme interprété.

Il faut tout de même préciser quel sera le programme qui sera chargé d'interpréter votre fichier. Sous un système UNIX, on utilise un **shebang** [1] pour cela. Un shebang est constitué des caractères `#!` suivi du chemin absolu vers l'interpréteur de commandes : ``/usr/bin/env ksh''. Le chemin de l'interpréteur peut être suivi des options de celui-ci.

Le corps du script shell est constitué d'une suite de commandes internes ou externes au shell. Il se termine en retournant une valeur de retour, traditionnellement 0 en cas de succès, une autre valeur en cas d'échec.

Pour lancer votre script shell, deux manières possibles :

- sh path/to/myscript.sh
  Ainsi le **shebang** sera ignoré et l'intepréteur sera le programme sh ;
- ./myscript.sh
  Dans ce cas, il est nécessaire que le script ait les droits d'exécution.

2.2 Entrée / Sortie

Il existe de nombreuses manières de communiquer avec un script shell.

2.2.1 Entrées

- La ligne de commande : C'est le moyen le plus pratique pour paramétrer l'exécution du script. L'inconvénient de ce système est qu'il nécessite une analyse complète de la ligne de commande pour vérifier la validité des arguments au démarrage, sans forcément l'ordre de ceux-ci. C'est donc assez fastidieux pour celui qui écrit le script.
- L'entrée standard (correspondant au FD [2] 0) : *le clavier* en général. C'est un moyen intuitif de communication entre l'utilisateur et le script, mais il n'est pas très utilisé car un script Shell n'a en général aucune interaction avec un humain vu qu'à l'origine, on souhaite automatiser l'exécution des commandes.
- Les fichiers : l'utilisation de fichiers de configuration pour utiliser un script Shell est facile, et recommandée pour un script qui tournera de manière automatique.

2.2.2 Sorties

- La valeur de retour : un programme sous *UNIX* renvoie une valeur de retour. Cette valeur, comprise entre 0 et 255, permet de tester facilement si un programme a quitté normalement ou avec une erreur. Par convention, un programme ayant bien terminé retourne 0, tandis qu'une erreur provoque un code de retour différent de 0.
  De cette manière, si le code a été bien pensé, on peut facilement et rapidement identifier une erreur. La valeur de retour est stockée dans la variable spéciale du shell.
- La sortie standard (correspondant au FD 1) : *l'écran* en général. Elle permet d'afficher des informations pour l'utilisateur. Elle peut être redirigée dans un fichier pour être analysée plus tard, ou vers un programme qui s'en servira comme entrée.
- La sortie d'erreur (correspondant au FD 2) : *l'écran* en général. Il s'agit du deuxième canal de sortie, le premier étant plutot dédié aux sorties normales. La sortie d'erreur, comme son nom l'indique, est privilégiée quand il s'agit d'émettre un message d'erreur, ou d'information du système.
- Les fichiers : ils peuvent servir de moyen de stockage du résultat.

2.3 Redirections

Dans le monde *UNIX*, tout est fichier : l'écran, le disque dur, la mémoire du logiciel que vous utilisez, et bien sûr les fichiers présents sur le disque dur [3]. Il est donc normal de vouloir que le shell écrive dans des fichiers et non sur l'écran. Les redirections permettent de rediriger des flux d'entrée/sortie. Elles servent par exemple à faire en sorte que la sortie standard d'un programme soit redirigée vers un fichier : [4]

1. Shebang : de l'inexacte contraction de sharp bang ou haSH Bang
2. Sous *UNIX*, les fichiers ouverts sont identifiés par des numéros, appelés descripteurs de fichier (fd ou *file descriptor*). L'entrée standard est numérotée 0, la sortie standard 1 et la sortie d'erreur 2.
3. Les dossiers sont aussi représentés sous forme de fichiers spéciaux.
4. est un fichier spécial qui fait disparaitre tout ce qui y entre.

CMD > file : La sortie de CMD est écrite dans le fichier. Si ce fichier n'existe pas, il est créé. S'il existe déjà, son contenu est écrasé.

CMD >> file : Concatène la sortie de CMD au contenu du fichier. Si le fichier n'existe pas, il est créé.

CMD < file : Le contenu de file sert d'entrée standard à la commande.

On peut aller encore plus loin en jouant avec les descripteurs de fichiers, pour rediriger certaines entrées et/ou sorties, mais pas d'autres. Pour indiquer *à partir* de quel descripteur de fichier on veut rediriger, il suffit d'indiquer le numéro de celui-ci avant le symbole de redirection. Pour indiquer *vers* quel descripteur de fichier on veut rediriger le flux, il suffit d'indiquer le numéro du descripteur voulu suivi du symbole de redirection.

```
42sh$ echo foo > file
42sh$ cat file
foo
42sh$ echo bar >> file
42sh$ cat file
foo
bar
42sh$ cat < file
foo
bar
42sh$ #redirection de la sortie d'erreur dans /dev/null
42sh$ badcommand 2> /dev/null
42sh$ #redirection des sorties dans /dev/null
42sh$ badcommand &> /dev/null
42sh$ #redirection de la sortie d'erreur vers la sortie
#standard
42sh$ command 2>&1
42sh$ cat << EOF > file
ceci est un mini editeur de texte
    il est tres utile
pour ecrire des
    rapports dans
        un script shell.
EOF
42sh$ cat file
ceci est un mini editeur de texte
    il est tres utile
pour ecrire des
    rapports dans
        un script shell.
42sh$
```

Note : la redirection est une extension de Bash.

2.4 Variables

Les variables en shell sont des chaînes de caractères nommées dont le contenu peut changer au cours de l'exécution. On instancie une variable par var=value. **Attention** : il n'y a pas d'espace de part et d'autre du signe '='. L'identifiant doit commencer par un caractère alphabétique majuscule ou minuscule ou un underscore '_' suivi de 0 ou plusieurs caractères alphanumériques ou underscore '_'.

On accède au contenu de la variable par son identifiant préfixé du symbole '$' : $var

Il est possible d'utiliser la forme qui permet d'éviter des confusions :

Le shell possède des variables spéciales : ce sont des variables qui contiennent des informations sur le processus courant, les valeurs de retour, le nombre d'arguments d'un script ou d'une fonction, etc.

$?	Valeur de retour de la dernière commande exécutée.
$#	Nombre d'arguments passés au script ou à la fonction.
$0	Nom du script ou de la fonction. Dépend du shell.
$i	Contient la valeur du ième argument passé au script sur la ligne de commande ($1 <= i <= 9$).
$@	Liste de tous les arguments, sans aucune substitution.

$	Liste de tous les arguments, vue comme un seul mot.
$$	PID du shell courant.
$!	PID du dernier processus lancé en tâche de fond.

En shell, il existe deux types de variables :

— les variables internes au shell, qui ne sont utilisables que dans le shell courant et que vous pouvez créer comme expliqué précédemment ;

— les variables d'environnement, qui sont utilisables dans le shell courant ainsi que dans les éventuels sous-shells. En réalité elles pourront être utilisées par n'importe quel programme lancé à partir du shell où vous avez défini les variables en question.

Attention : Les sous-shells n'ont pas directement accès aux variables d'environnement du shell parent. Ils en possèdent une copie. Par conséquent, les sous-shells ne peuvent pas modifier les variables d'environnement du shell initial. Ils peuvent modifier leur copie, mais les changements ne seront visibles que dans le shell où la modification a été faite.

Pour créer des variables d'environnement, on procède de la même manière que pour créer une variable interne au shell. Cependant, il faudra utiliser la commande `export` pour demander au shell d'exporter cette variable dans l'environnement.

2.4.1 Variables usuelles

Dans les systèmes UNIX, un certain nombre de variables d'environnement ont une signification particulière et vous permettent de configurer ou d'obtenir des informations sur votre système. Votre shell vous fournit également certaines variables vous permettant d'interagir avec lui.

Voici une liste non exhaustive de ces variables et leur signification :

$PATH	Liste des différents répertoires où le shell est habilité à rechercher les exécutables si leur chemin absolu ou relatif n'est pas précisé. C'est grâce à ce mécanisme qu'on peut taper juste `ls` plutôt que `/bin/ls`.
$LD_LIBRARY_PATH	Liste des différents répertoires où le chargeur de bibliothèques est habilité à rechercher les bibliothèques dynamiques nécessaires au programme (cette variable a pour nom SHLIB_PATH sous HP-UX).
$USER	Nom de l'utilisateur courant.
$HOSTTYPE	Architecture du système.
$HOSTNAME	Nom de la machine courante.
$PS1	Configuration du prompt utilisateur.
$SECONDS	Temps en secondes depuis l'ouverture du shell ou la dernière remise à 0.
$OLDPWD	Chemin du dernier répertoire visité.
$EDITOR	Éditeur préféré de l'utilisateur.
$TERM	Type de terminal courant
$SHELL	Shell de l'utilisateur.
$SHLVL	*SHell LeVeL* : nombre de shells imbriqués au-dessus du shell courant.

Vous pouvez à tout moment afficher l'ensemble des variables d'environnement dans votre shell, à l'aide de la commande env.

2.5 Builtins

La plupart des commandes que vous utilisez sont des programmes à part. C'est le cas par exemple de ls ou cp. Les builtins sont des commandes intégrées au shell et executées par le shell lui-même. Ce sont en général des fonctionnalités bas niveau du shell comme cd ou des fonctions de gestion des variables d'environnement comme export.

Pour vérifier si une commande est une builtin ou non, on peut utiliser la builtin : builtin. Voici une liste non-exhaustive des builtins les plus utiles, la liste complète des builtins disponibles pour un CLI donné est disponible dans le manuel de ce shell.

exit	Quitte le shell courant avec pour valeur de retour la valeur donnée en argument. Si cette valeur est omise, retourne par défaut.
cd	Change de répertoire et se déplace dans le dossier indiqué en argument. (cd = Change Directory)
echo	Affiche sur STDOUT (la sortie standard) les arguments.
source	Exécute un script Shell dans le shell courant.
shift	Enlève les N premiers arguments de la ligne de commande (par défaut N=1).
eval	Permet de construire une commande à partir des arguments passés à eval.

2.6 Quoting

Le quoting permet de modifier la manière dont le shell va interpréter certaines expressions. Pour cela, il suffit d'entourer l'expression par des guillemets simples, doubles ou inversés. Chacun de ces symboles a un comportement différent.

2.6.1 Les double quotes (``)

- Inhibent le globbing du shell.
- Assemblent plusieurs **mots** en un seul argument [5].
- Permettent l'expansion des variables, c'est-à-dire le remplacement du nom de la variable par sa valeur.

Il est fortement conseillé d'utiliser les doubles quotes lors de l'utilisation de variables contenant une chaîne de caractères.

2.6.2 Les simple quotes (')

Les simple quotes ont le même comportement que les double quotes, mais inhibent en plus l'expansion des variables.

2.6.3 Les backquotes (`)

Les backquotes ont un comportement différent des deux quotings précédents. Elles permettent de lancer une commande dans un sous-shell. La commande entre backquotes est substituée par la sortie standard de cette commande. Ainsi il est possible de récupérer cette sortie standard dans une variable en utilisant `` `cmd` ``. La valeur de retour de la commande est stockée dans la variable `$ ?`. Il est possible de cumuler différents types de *quoting*.

```
42sh$ echo "Bonjour $USER, nous sommes le `date`."
Bonjour grand_x, nous sommes le Sun Sep 15 17:41:28 CEST 2013.
```

3 Script Shell avancé

3.1 Test et Calcul

3.1.1 Test

Le shell intègre une builtin de test nommée test. Cette builtin permet de tester une condition sur des variables, des valeurs numériques et/ou des fichiers. Elle retourne 0 si le test est vérifié, 1 sinon. Il existe un équivalent plus court à test : `[`, cette builtin se termine par un `]`. Voici une liste rapide des options de les plus courantes [6] :

-n string	Vérifie que la chaîne n'est pas vide, ou inexistante.
-z string	Vérifie que la chaîne est vide ou inexistante.
string1 = string2	Teste si les deux chaînes sont identiques.
string1 != string2	Teste si les deux chaînes sont différentes.

5. Utile si l'argument comprend des espaces.
6. string = chaîne de caractères, integer = valeur numérique entière, file = nom de fichier

integer1 -eq integer2	Teste si les deux entiers ont la même valeur.
integer1 -lt integer2	Teste si *integer1* est inférieur à *integer2*.
integer1 -ge integer2	Teste si *integer1* est supérieur ou égale à *integer2*.
-e file	Vérifie l'existence d'un fichier.
-d file	Vérifie si le fichier est un répertoire.
-x file	Vérifie si le fichier existe et est exécutable (ou traversable s'il s'agit d'un dossier).

3.1.2 Calcul

Le shell possède un évaluateur d'expressions arithmétiques, `$((exp))` où `exp` est une expression arithmétique qui accepte les opérations classiques : addition, soustraction, multiplication, division, modulo, et aussi avec les nombres relatifs mais pas décimaux.

```
42sh$ x=3
42sh$ y=6
42sh$ echo $(( x + (4 - y) % 3))
1
42sh$ echo $(( $x + (4 - $y) % 3))
1
```

3.2 Structures de contrôle

Le shell permet d'utiliser les structures de contrôle comme le if ou les boucles. La condition de contrôle de ces commandes est la valeur de retour de la commande de condition : 0 = vrai, n'importe quelle autre valeur = faux.

3.2.1 if then else

Les mots clefs if, then, else permettent de tester la validité d'une condition et de brancher sur la portion de script intéressante. Une condition se termine toujours par un fi. Une condition est dite validée si la valeur de retour du dernier programme exécuté est nulle.

```
if CONDITION; then
 COMMANDES
elif CONDITION; then
 COMMANDES
else
 COMMANDES
fi;
```

3.2.2 Boucles

Il existe deux façons de faire des boucles en Shell. Elles ont une syntaxe similaire :
while : Tant que la condition est vérifiée, la boucle tourne. until : Tant que la condition n'est pas vérifiée, la boucle tourne.
La détermination de la validité de la condition est identique à celle de if.
Il existe un troisième type de boucle, la boucle for. Sa syntaxe est la suivante :
for VARIABLE in LIST ; do COMMANDES done
Ce type de boucle permet de parcourir une liste. Tant que la liste n'est pas finie, VARIABLE prend à chaque tour de boucle une nouvelle valeur de LIST. LIST est une suite de mot séparés par l'IFS courant.

3.2.3 switch ... case

Le shell possède également une structure de contrôle de type *switch case* utilisable de la manière suivante :

```
case WORD in
   PATTERN1 )
    EXPRESSION ;;
   PATTERN2 | PATTERN3 )
    EXPRESSION ;;
```

```
   * )
    DEFAULT EXPRESSION;;
esac
```

3.3 Globbing

Parfois vous aurez besoin d'appliquer le même traitement sur un grand nombre de fichiers. Il serait très fastidieux de lister tous les fichiers à la main. Le globbing vous permet d'exprimer des noms de fichiers sous la forme de *patterns*. Un *pattern* (ou motif en français) est une chaîne de caractères qui décrit certaines caractéristiques d'un nom de fichier. Ainsi, on peut exprimer que les noms de fichiers à matcher comportent 8 lettres, que la deuxième lettre est un 't' et que les lettres 4 et 6 sont des voyelles. Pour cela, on se sert des *caractères jokers*. Ce sont des caractères spéciaux qui, dans le contexte du globbing, représentent un ou plusieurs caractères.

Voici une liste des équivalences des caractères et leur signification.

?	1 caractère
*	0 ou plusieurs caractères
[aeiouy]	1 des caractères compris entre les crochets
[a-t]	1 des caractères compris entre et selon l'ordre *ASCII* (pour plus de précision).
[^abc] ou [!abc]	1 caractère différent de ceux mentionnés.

```
42sh\$ echo /[a-e]??
/bin /dev /etc
42sh\$ echo /h*e
/home
```

3.4 Enchaînement de commandes

Le shell dispose d'un certain nombre d'opérateurs permettant de contrôler l'enchaînement de commandes…

`CMD1 ; CMD2` : Exécute `CMD2` une fois que `CMD1` a fini. `CMD1 && CMD2` : Exécute `CMD2` si et seulement si `CMD1` a fini avec succès. `CMD1 || CMD2` : Exécute `CMD2` si et seulement si `CMD1` a échoué. `CMD1 | CMD2` : Exécute `CMD1` et redirige sa sortie standard vers l'entrée standard de `CMD2`. `CMD1 & CMD2` : Exécute `CMD1` en arrière-plan et `CMD2` en avant-plan.

```
42sh$ true && echo "ca marche"
ca marche
42sh$ false && echo "ca marche"
42sh$ cat file.txt | grep toto
ici toto mange des artichauts
toto aime les artichauts
42sh$
42sh$ sleep 3 & echo 'je suis ici'
[1] 423
je suis ici
42sh$
  ## 3 secondes plus tard:
42sh$
[1]+  Done                    sleep 3
```

3.5 Sous-shell, source et exec

3.5.1 Sous-shells

Les seules informations qu'on peut récupérer d'un sous-shell sont sa sortie standard et son code de retour. On ne peut pas récupérer la sortie d'erreur, et les variables définies dans le sous-shell ne sont pas visibles dans le shell parent. Les sous-shells sont donc des moyens d'exécuter des commandes qui interagissent très peu avec le shell courant. Si on veut interagir plus avec avec le shell courant, il faut utiliser une autre solution.

Pour lancer un sous-shell, il existe deux méthodes : `` `COMMANDE` `` ou `$(COMMANDE)`

3.5.2 Source

`.` est une builtin qui permet de faire exécuter un script shell par le shell courant. On utilise cette fonctionnalité pour mettre en place des variables, des fonctions ou des alias.

3.5.3 exec

Le principal intérêt de la builtin `exec` est de remplacer le shell par la commande en argument. Cette builtin est aussi utilisée pour faire des redirections vers de nouveaux descripteurs de fichier pour le shell courant si aucune commande n'est spécifiée.

```
42sh$ exec 3>/tmp/file.txt
42sh$ echo "Le shell c'est bien
> mais avec exec
> ca va très loin" >&3
42sh$ echo "Hé ouais" >&3
42sh$ cat /tmp/file.txt
Le shell c'est bien mais avec exec ca va très loin
Hé ouais
```

3.6 Fonctions

Les fonctions sont comme des mini-scripts shell. On peut accéder aux arguments d'une fonction *via* les variables spéciales `$1`, ..., `$9`. Une fonction possède une valeur de retour.

```
42sh$ bigger() { du $1/* | sort -n; }
42sh$ cd test; bigger .
4   ./AUTHORS
4   ./doc
4   ./src
12  ./README
28  ./TODO
42sh$
```

- `du` : estime l'espace disque utilisé pour un fichier.
- `sort` : trie.

3.7 Entrées/sorties avancées

Il est intéressant pour certains scripts shells de pouvoir interroger l'utilisateur et lui demander de rentrer des informations pendant le déroulement du script. Pour cela, on utilise la builtin `read` qui lit sur l'entrée standard et enregistre les mots dans les variables dont les noms sont les arguments de `read`. S'il y a plus de mots que d'arguments passés à `read`, la commande stocke le surplus dans la dernière variable.

Pour séparer les mots, `read` utilise l'IFS (*Internal Field Separator*). L'*IFS* détermine la façon dont le shell reconnaît les champs ou les limites de mots lorsqu'il interprète des chaînes de caractères. Par défaut l'IFS vaut à la fois l'espace, la tabulation et le retour à la ligne, mais on peut le changer et, par exemple, lui donner la valeur `:` pour extraire des informations du `$PATH` par exemple.

```
42sh$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/opt/bin/:/u/a1/login_l/mbin
42sh$ echo $PATH > PATH.file
42sh$ IFS=':'
42sh$ read p1 p2 p3 p4 < PATH.file
42sh$ echo $p1
/usr/local/bin
42sh$ echo $p2
/usr/bin
42sh$ echo $p3 $p4
/bin /opt/bin /u/a1/login_l/mbin
```

4 Divers

4.1 Commandes utiles

tr	Remplace ou supprime un caractère.
head	Donne les premières lignes d'un fichier.
tail	Donne les dernières lignes d'un fichier.
cat	Affiche le contenu d'un fichier.
time	Donne le temps d'exécution d'un programme.
grep	Cherche un motif dans un fichier
cut	Coupe une ligne en champs en fonction d'un délimiteur.
paste	Colle une ligne de champs séparés par un délimiteur.
diff	Affiche les différences entre deux fichiers.
kill	Tue un processus.
sort	Trie un fichier.
wc	Compte le nombre de lignes, mots, caractères d'un fichier.
uniq	Supprime les doublons.
man	Manuel de toutes les applications.
date	Donne la date courante.
screen	Terminal virtuel, détachable.
bc	Évaluateur d'expression arithmétique, extensible aux réels avec l'option .
xargs	Construit et exécute des lignes de commande à partir de l'entrée standard.
dirname	Ne conserve que la partie répertoire d'un chemin d'accès.
basename	Élimine le chemin d'accès et le suffixe d'un nom de fichier.

4.2 Liens utiles

`http://www.shellunix.com`
`http://www.livefirelabs.com/unix_tip_trick_shell_script/unix_tip_archive.htm`
`http://cli.asyd.net`

4.3 Références

`http://www.multicians.org/shell.html`	The Origin of the Shell
`http://www.opengroup.org/onlinepubs/009695399/`	Single UNIX Spécification
`http://www.UNIX.org/version3/ieee_std.html`	POSIX
	Manuel en ligne de sh
ou	Manuel en ligne de Bash

5 Rappels

Pour écrire un script , il faut rajouter cette ligne en haut de tout fichier afin de spécifier au système que nous allons utiliser l'interpreteur et la syntaxe associée ().

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

Ensuite, il faut mettre le fichier en mode exécutable avec la commande :

```
42sh$ chmod +x script.sh
```

On pourra alors appeler le programme de la façon suivante :

```
42sh$ ./script.sh
```

Nous rappellons que nous utilisons le shell sh, qui n'est pas votre shell par défaut, vous devez lancer la commande sh pour pouvoir tester interactivement.

Pour les exercices suivants, vous avez le droit à toutes les builtins. Pour savoir si une commande que vous utilisez est une builtin ou non, servez-vous de la commande type qui, elle, est une builtin (commande intégrée au shell).

```
42sh$ type echo
echo is a shell builtin
42sh$ type type
type is a shell builtin
42sh$ type cd
cd is a shell builtin
42sh$ type ls
ls is /bin/ls
42sh$ type ps
ps is /bin/ps
```

Pour chaque exercice, il vous sera spécifié les commandes auxquelles vous avez le droit. Pour plus d'informations sur ces commandes, reportez-vous aux man.

Le shell peut vous donner un tas d'informations sur l'exécution qui peuvent être utiles si vous souhaitez << debugguer >> vos scripts. Pour les activer, passez le paramètre à :

```
42sh$ sh -x monscript.sh
...

OU
#! /bin/sh -x
#
# Next part of the script
...

OU
#! /bin/sh
# No need to debug
set -x
# Debug mode on
set +x
# No need to debug
```

Si vous avez des questions sur une commande, vérifiez que la réponse ne se trouve pas dans le man de la commande. Si vous recherchez un comportement précis, utilisez la commande pour connaitre les pages de évoquant ce comportement.

6 Expressions rationnelles

Les **expressions rationnelles**, en anglais ***regular expressions*** (la traduction « expressions règulières » est inexacte), « regexp » ou « regex » en abrégé, sont un support essentiel pour de nombreux utilitaires UNIX dont `grep` et `sed`, que nous allons étudier.

Une expression rationnelle est un motif qui permet de décrire un ensemble de chaînes de caractères. Les expressions rationnelles sont construites comme des opérations arithmétiques : elles utilisent différents opérateurs pour combiner des expressions plus petites.

Il existe plusieurs types d'expressions rationnelles : les **simples** et les **étendues**. La plupart des utilitaires proposent l'une ou l'autre forme grâce à des options en ligne de commande. La différence entre ces deux formes est principalement la nécessité ou non de préfixer certaines constructions par un **backslash** (\), comme nous le verrons plus loin.

Une expression rationnelle est mise en correspondance avec une chaîne caractère par caractère.

Note : Dans les exemples, on utilisera l'opérateur `=~` (c'est la notation utilisée en Perl et Ruby), qui signifie un test de correspondance (*match* en anglais), l'expression de gauche étant la chaîne à tester et celle de droite l'expression rationnelle.

6.1 Expression rationnelle simple

Les briques élémentaires sont les expressions rationnelles correspondant à un seul caractère.

```
ABCD =~ ABCD            => correspondance
abcd =~ ABCD            => non correspondance
xyzABCDxyz =~ ABCD      => correspondance (avec la sous-chaîne)
```

Certains caractères ont une signification spéciale : on les appelle des métacaractères. Pour inhiber leur signification spéciale, on les précède d'un *backslash* \.

6.1.1 Le point .

Le métacaractère point (`.`) représente **une** instance de **n'importe quel** caractère :

```
AxB =~ A.B              => correspondance
A B =~ A.B              => correspondance
A123B =~ A.B            => non correspondance
A =~ A.                 => non correspondance
AxB =~ A\.B             => non correspondance
A.B =~ A\.B             => correspondance
```

6.1.2 Début ^ et fin $ de chaîne

Les deux symboles spéciaux `^` et `$` représentent les début et fin de chaîne. Il ne sont mis en correspondance avec aucun caractère véritable mais plutôt avec une chaîne vide. Par exemple `^a` signifie « n'importe quelle chaîne commençant par `a` » ; de même `b$` signifie « n'importe quelle chaîne terminée par `b` ».

```
abcde =~ ^a             => correspondance
bacde =~ ^a             => non correspondance
abcde =~ e$             => correspondance
abcde =~ ^a$            => non correspondance
a =~ ^a$                => correspondance
```

Attention : Les symboles `^` et `$` reprennent leur signification littérale s'ils ne sont pas en début (respectivement fin) de l'expression rationnelle :

```
A$B =~ A$B              => correspondance
A^ =~ A^                => correspondance
(chaîne vide) =~ ^$     => correspondance
^$ =~ ^$                => non correspondance
b$ =~ b$                => non correspondance
b$ =~ b\$               => correspondance
```

6.1.3 Alternatives \|

Le caractère `|`, précédé d'un backslash \, représente une alternative entre deux éléments :

```
abd =~ ab\|cd           => correspondance
acd =~ ab\|cd           => correspondance
axd =~ ab\|cd           => non correspondance
aed =~ ab\|c\|ed        => correspondance
```

6.1.4 Listes []

Les listes sont des raccourcis pour les alternatives : au lieu d'écrire `` `a\|b\|c' ``, on peut écrire `` `[abc]' ``.

```
abd =~ a[bce]d          => correspondance
acd =~ a[bce]d          => correspondance
axd =~ a[bce]d          => non correspondance
```

Attention : À l'intérieur des listes les métacaractères reprennent leur signification littérale.

```
\ =~ [.*\]              => correspondance
[ =~ []abc[]            => correspondance
] =~ []abc[]            => correspondance
a =~ []abc[]            => correspondance
```

Si le premier caractère de la liste est ^ la signification de la liste est inversée.

```
a =~ [^abc]             => non correspondance
e =~ [^abc]             => correspondance
```

6.1.5 Intervalles

Les intervalles sont des raccourcis pour décrire des caractères successifs au sein d'une liste.

```
z3 =~ [a-z][0-9]        => correspondance
zz =~ [a-z][0-9]        => non correspondance
a4 =~ a[1-35]           => non correspondance
a5 =~ a[1-35]           => correspondance
a6 =~ a[1-35-7]         => correspondance
```

Pour pouvoir utiliser le caractère – dans une liste, il faut le placer en premier.

```
- =~ [-A-C]             => correspondance
B =~ [-A-C]             => correspondance
```

Attention : L'ordre des caractères est déterminé par le code ASCII.

6.1.6 Classes

Les classes sont des intervalles prédéfinis, qui peuvent être paramétrés par la variable d'environnement `LC_ALL`, afin de respecter les différents encodages de caractères. L'encodage qui nous intéressera sera ISO-8859-1, obtenu grâce à `LC_ALL=fr_FR`.

La syntaxe des différentes classes est : `[ :nom de la classe:]` à l'intérieur d'une liste.

Nom	Signification	ASCII	ISO-8859-1
alpha	Lettres alphabétiques dans la localisation en cours	`[A-Za-z]`	`[A-Za-zÁÀÄ...ùûý]`
digit	Chiffres décimaux	`[0-9]`	idem ASCII
xdigit	Chiffres hexadécimaux	`[0-9A-Fa-f]`	idem ASCII
alnum	Chiffres ou lettres alphabétiques	`[[ :alpha:][ :digit:]]`	`[[ :alpha:][ :digit:]]`
lower	Lettres minuscules dans la localisation en cours	`[a-z]`	`[a-záàäâ...üý]`
upper	Lettres majuscules dans la localisation en cours	`[A-Z]`	`[A-ZÁÀÄÂ...üý]`
blank	Caractères blancs	`[ \t]`	idem ASCII
space	Caractères d'espacement	`[ \t\n\f\r]`	idem ASCII
punct	Signes de ponctuation	`[]-!"#$%&'()*./:+;<=>?@\^_\{|}~[]``	idem ASCII
graph	Symboles ayant une représentation graphique	`[[ :alnum:][ :punct:]]`	`[[ :alnum:][ :punct:]]`
print	Caractères imprimables	`[[ :graph:]]`	`[[ :graph:]]`
cntrl	Caractères de contrôle	Code ASCII inférieur à 31 et caractère de code 127	idem ASCII

6.1.7 Répétitions

Pour décrire des répétitions on dispose des métacaractères suivants :

— `*` zéro ou plusieurs occurrences de l'élément précédent ;
— `\+` une ou plusieurs occurrences de l'élément précédent ;
— `\?` zéro ou une occurrence de l'élément précédent ;
— `\{n,m\}` au moins `n' et au plus `m' occurrences de l'élément précédent ;
— `\{n,\}` au moins `n' occurrences de l'élément précédent ;
— `\{n\}` exactement `n' occurrences de l'élément précédent.

```
ac =~ ab\+c             => non correspondance
abc =~ ab\+c            => correspondance
abbbc =~ ab\+c          => correspondance

ac =~ ab*c             => correspondance
abc =~ ab*c            => correspondance
abbbc =~ ab*c          => correspondance

ac =~ ab\?c             => correspondance
abc =~ ab\?c            => correspondance
abbbc =~ ab\?c          => non correspondance

abc =~ ab\{2,3\}c     => non correspondance
abbc =~ ab\{2,3\}c    => correspondance
abbbc =~ ab\{2,3\}c   => correspondance
abbbbc =~ ab\{2,3\}c  => non correspondance
```

À noter que les caractères de répétition combinés à des métacaractères représentant des alternatives n'attendent pas plusieurs occurrences du même caractère :

```
a12345c =~ a.\{5\}c   => correspondance
axyzxc =~ a[xyz]\+c     => correspondance
```

6.1.8 Groupements `\(...\)`

On peut regrouper différents caractères en un seul élément grâce à la forme `\(\)`. Cette construction s'avère très utile, surtout en combinaison avec d'autres métacaractères :

```
123 =~ \(123\)\{2\}     => non correspondance
123123 =~ \(123\)\{2\}  => correspondance

abc =~ \(abc\)\|\(def\)  => correspondance
def =~ \(abc\)\|\(def\)  => correspondance
xyz =~ \(abc\)\|\(def\)  => non correspondance

AxBAyB =~ \(A.B\)\{2\}  => correspondance
```

6.1.9 Référence arrière `\NUM`

La valeur mise en correspondance par un groupement peut être réutilisée dans la suite d'une expression rationnelle grâce à la construction `\NUM`, avec `NUM` un numéro représentant le `NUM`-ième groupement :

```
axbcyabc =~ \(.\)x\(..\)y\1\2   => correspondance
axbcyabd =~ \(.\)x\(..\)y\1\2   => non correspondance
```

6.2 Expressions rationnelles étendues

Les expressions rationnelles étendues ont la même puissance que les expressions rationnelles simples, seule la syntaxe est différente : on omet le caractère `\` pour certaines constructions :

Signification	Symbole pour expression rationnelle simple	Symbole pour expression rationnelle étendue

Caractère générique	`.`	`.`
Début de ligne	`^`	`^`
Fin de ligne	`$`	`$`
Alternative	`|`	`|`
Liste de caractères	`[]`	`[]`
Classe de caractère	`[ :classe:]`	`[ :classe:]`
Zéro ou plusieurs occurrences de l'élément précédent	`*`	`*`
Une ou plusieurs occurrences de l'élément précédent	`\+`	`+`
Zéro ou une occurrence de l'élément précédent	`\?`	`?`
Au moins n et au plus m occurrences de l'élément précédent	`\{n,m\}`	`{n,m}`
Au moins n occurrences de l'élément précédent	`\{n,\}`	`{n,}`
Au plus m occurrences de l'élément précédent	`\{0,m\}`	`{0,m}`
Exactement n occurrences de l'élément précédent	`\{n\}`	`{n}`
Groupement de caractères	`\(\)`	`()`
Référence arrière au num-ième regroupement	`\num`	`\num`
Préfixe d'un caractère spécial pour prendre sa valeur littérale	`\`	`\`

7 Grep

7.0.1 Version basique de grep

`grep` est un outil qui recherche dans les fichiers indiqués en ligne de commande (ou depuis l'entrée standard si aucun fichier n'est fourni, ou si le nom – est indiqué) les lignes correspondant à une certaine expression rationnelle. Par défaut, `grep` affiche les lignes qui correspondent à l'expression rationnelle simple donnée en paramètre.

Voici les options courantes de `grep` :

- `-E` utilise les expressions rationnelles étendues ;
- `-v` affiche les lignes ne contenant pas la chaine ;
- `-i` ne tient pas compte de la casse (majuscules/minuscules) ;
- `-c` compte le nombre de lignes contenant la chaîne ;
- `-n` numérote chaque ligne contenant la chaîne ;
- `-x` ligne correspondant exactement à la chaîne ;
- `-l` affiche le nom des fichiers qui contiennent la chaîne ;
- `-R` effectue une recherche récursive à partir du dossier ;
- `-A NUM` Affiche `NUM` lignes avant la ligne détectée ;
- `-B NUM` Affiche `NUM` lignes après la ligne détectée ;
- `-C NUM` Affiche `NUM` lignes avant et après la ligne détectée.

Voici quelques exemples simples d'utilisation avec le fichier `/etc/passwd` :

- Comment obtenir toutes les lignes commençant par le caractère `x` ou par la chaîne `nguyen_o` ?

```
42sh$ grep "^x\|^nguyen_o" /etc/passwd
nguyen_o:x:62166:2012:tony nguyen:/home/nguyen_o/:/bin/tcsh
xavier:x:7005:7000:xavier mangeard:/home/xavier/:/bin/tcsh
xercav_s:x:24036:24000:sarah xercavins:/home/xercav_s/:/bin/tcsh
xie_h:x:74710:32014:huai xie:/home/xie_h/:/bin/tcsh
xie_m:x:16371:2006:mu xie:/home/xie_m/:/bin/tcsh
xie_t:x:73639:32013:tingchi xie:/home/xie_t/:/bin/tcsh
xodo_a:x:77212:32016:antoine xodo:/home/xodo_a/:/bin/tcsh
xoual_o:x:93115:52013:olivier xoual:/home/xoual_o/:/bin/tcsh
xu_a:x:72089:32012:anthony xu:/home/xu_a/:/bin/tcsh
```

```
xu_d:x:22891:21000:deauma xu:/home/xu_d/:/bin/tcsh
xu_l:x:74949:32014:long-long xu:/home/xu_l/:/bin/tcsh
```

— Comme la question précédente, mais en utilisant les expressions rationnelles étendues.

```
42sh$ grep -E "^x|^nguyen_o" /etc/passwd
nguyen_o:x:62166:2012:tony nguyen:/home/nguyen_o/:/bin/tcsh
xavier:x:7005:7000:xavier mangeard:/home/xavier/:/bin/tcsh
xercav_s:x:24036:24000:sarah xercavins:/home/xercav_s/:/bin/tcsh
xie_h:x:74710:32014:huai xie:/home/xie_h/:/bin/tcsh
xie_m:x:16371:2006:mu xie:/home/xie_m/:/bin/tcsh
xie_t:x:73639:32013:tingchi xie:/home/xie_t/:/bin/tcsh
xodo_a:x:77212:32016:antoine xodo:/home/xodo_a/:/bin/tcsh
xoual_o:x:93115:52013:olivier xoual:/home/xoual_o/:/bin/tcsh
xu_a:x:72089:32012:anthony xu:/home/xu_a/:/bin/tcsh
xu_d:x:22891:21000:deauma xu:/home/xu_d/:/bin/tcsh
xu_l:x:74949:32014:long-long xu:/home/xu_l/:/bin/tcsh
```

— Comment afficher les lignes ne contenant pas le motif `sh` ?

```
42sh$ grep -v sh /etc/passwd
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
usbmuxd:x:113:113:usbmuxd user:/:/sbin/nologin
avahi-autoipd:x:499:499:avahi-autoipd:/var/lib/avahi-autoipd:/sbin/nologin
vcsa:x:69:498:virtual console memory owner:/dev:/sbin/nologin
rtkit:x:498:497:RealtimeKit:/proc:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
abrt:x:497:496::/etc/abrt:/sbin/nologin
tcpdump:x:72:72::/:/sbin/nologin
avahi:x:496:493:avahi-daemon:/var/run/avahi-daemon:/sbin/nologin
haldaemon:x:68:492:HAL daemon:/:/sbin/nologin
openvpn:x:495:491:OpenVPN:/etc/openvpn:/sbin/nologin
apache:x:48:490:Apache:/var/www:/sbin/nologin
saslauth:x:494:489:''Saslauthd user'':/var/empty/saslauth:/sbin/nologin
mailnull:x:47:488::/var/spool/mqueue:/sbin/nologin
smmsp:x:51:487::/var/spool/mqueue:/sbin/nologin
ntp:x:38:38::/etc/ntp:/sbin/nologin
nm-openconnect:x:493:486:NetworkManager user for OpenConnect:/:/sbin/nologin
pulse:x:491:483:PulseAudio System Daemon:/var/run/pulse:/sbin/nologin
gdm:x:42:481::/var/lib/gdm:/sbin/nologin
qemu:x:107:107:qemu user:/:/sbin/nologin
```

— Comment afficher le nombre de lignes qui contiennent le mot `root` ?

```
42sh$ grep -c root /etc/passwd
3
```

— Comment connaître le numéro des lignes qui contenant le mot `root` ?

```
42sh$ grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
```

```
11:operator:x:11:0:operator:/root:/sbin/nologin
12129:rootme:x:812:800:rootme lse:/home/rootme/:/bin/tcsh
```

Attention : N'oubliez pas que le shell **étend** les arguments avant d'exécuter la commande.
Prenez l'habitude de protéger vos expressions rationnelles par des *quotes*.

7.0.2 Versions alternatives de `grep`

— **`fgrep`**, pour des recherches avec des motifs fixés.
— **`zgrep`**, pour une recherche dans un fichier compressé avec gzip.

8 Sed

`sed` (*stream editor*) est un éditeur de flux non interactif : il lit les lignes d'un fichier (ou de l'entrée standard) une par une, leur applique des commandes d'édition et renvoie leur résultat sur la sortie standard. Il ne modifie pas le fichier traité mais écrit tout sur la sortie standard. Le plus gros atout de `sed` est de pouvoir être utilisé au sein d'un pipeline.

La syntaxe d'invocation de `sed` est la suivante :

```
sed -e 'programme sed' fichier-a-traiter
```

ou :

```
sed -f fichier-programme fichier-a-traiter
```

Voici plusieurs exemples d'invocation de `sed` :

```
42sh$ sed "s/foo/bar/; s/toto/titi/g" fichier
42sh$ sed -e 's/foo/bar/' -e 's/titi/toto/' fichier
42sh$ echo 's/foo/bar/' > /tmp/prog.sed &&
     sed -f /tmp/prog.sed -e 's/titi/toto/' fichier
42sh$ < fichier sed "s/foo/bar/; s/toto/titi/g"
```

Il est aussi possible de faire des scripts `sed`, pour cela nous remplaçons le traditionnel *shebang* `#!/bin/sh` des scripts shell par `#!/bin/sed -f`.

Le mécanisme de `sed` est le suivant :

1. Lecture d'une ligne sur le flux d'entrée (jusqu'à la rencontre d'un caractère de saut de ligne).
2. Traitement de cette ligne en la soumettant à **toutes** les commandes recontrées dans le fichier script.
3. Affichage de la ligne résultante sur la sortie standard, sauf si `sed` est invoqué avec l'option `-n`.

Il est possible aussi d'appeler `sed` avec l'option `-i`. Dans ce cas, les commandes modifient directement le fichier donné en argument plutôt que l'afficher sur la sortie standard. Certains arguments peuvent être ajoutés à `-i` en suffixe. (Cette option ne fonctionne pas avec toutes les versions de `sed`.)

Chaque instruction d'un fichier `sed` est composée d'une commande ou d'un bloc de commandes (ajout d'une ligne, suppression d'une ligne, remplacement de chaîne, etc.) et peut être précédée d'une adresse permettant d'appliquer la commande à un sous ensemble de lignes.

Exemple de script `sed` :

```
#!/bin/sed -rf

# This is a comment
/[a-z]+[a-z0-9]*/ {
  y/0123456789/1234567890/
  p
}

d
```

Par défaut, `sed` utilise les expressions rationnelles simples. Pour utiliser les étendues, comme avec `grep`, on invoquera `sed` avec l'option `-r`.

8.1 Commandes simples

8.1.1 La commande de substitution : s

La fonction de substitution `s` permet de changer la première ou toutes les occurrences d'une chaîne par une autre.
C'est de loin la commande la plus utilisée dans les scripts `sed`. À elle seule, elle donne un intérêt à l'outil `sed`.
La syntaxe générale est la suivante :

```
s/motif/remplacement/options
```

- `motif` est une expression rationnelle (simple ou étendue selon l'option donnée à l'invocation de sed, simple par défaut) ;
- `remplacement` est une chaîne qui remplace le `motif' dans le résultat. Cette chaîne peut contenir deux métacaractères :
  - `&` sera remplacé par la chaîne de caractères complète qui a été mise en correspondance avec le `motif',
  - `\num`, où `num` étant un nombre entre 1 et 9, est remplacé par le `num`-ième groupement dans le `motif`. Un groupement correspond à ce qui se trouve entre parenthèses dans le motif ;
- `options` disponibles :
  - `g` : Remplacer tous les motifs rencontrés dans la ligne en cours ;
  - `n` : Ne remplacer que la `n'-ième occurrence du motif dans la ligne ;
  - `i` : Être non sensible à la case ;
  - `p` : Afficher la ligne sur la sortie standard si une substitution est réalisée ;
  - `w fichier` : Écrire la ligne dans `fichier' si une substitution est réalisée.

Il peut y avoir plusieurs options en même temps.
Voici quelques exemples :

```
# change la premiere occurrence de la chaine
# `toto' par `TOTO'  (la premiere chaine `toto'
# rencontree dans le texte uniquement)
sed 's/toto/TOTO/' fichier

# va changer la troisieme occurrence de la
# chaine `toto' par `TOTO' (la troisieme chaine
# `toto' rencontree dans le texte uniquement)
sed 's/toto/TOTO/3' fichier

# va changer toutes les occurrences de la chaine
# `toto' par `TOTO' (toutes les chaines toto
# rencontrees sont changees)
sed 's/toto/TOTO/g' fichier

# en cas de substitution la ligne en entree est
# inscrite dans un fichier `resultat'
sed 's/toto/TOTO/w resultat' fichier
```

La fonction de substitution peut évidemment être utilisée avec une expression rationnelle.

```
# substitue toutes les chaines
# `Fraise' ou `fraise' par `FRAISE'
sed -e 's/[Ff]raise/FRAISE/g'

# substitue toutes les chaines
# `Fraise' ou `fraise' par `_Fraise_' ou `_fraise_'
sed -e 's/[Ff]raise/_&_/g' fichier

# enleve les `<>' entourant les chaines
sed -e 's/<\(.*\)>/\1/g' fichier
```

Note : On peut utiliser n'importe quel caratère de séparation (pas seulement `/'). Pour pouvoir lui redonner son sens usuel dans les motifs, il faut le précéder d'un `\'. Il en est de même pour la commande `y'.

```
42sh$ sed -e 's,titi,/titi/,g'
42sh$ sed -e 's,;,\,,g'
42sh$
```

8.1.2 La commande de suppression : `d'

La commande de suppression `d' supprime la ligne en cours et commence immédiatement le prochain cycle sans exécuter les autres commandes.

8.1.3 La commande de remplacement de caractères : `y'

La commande `y/src-chars/dest-chars' transpose tous les caractères de `src-chars' par leur equivalent dans `dest-chars', comme le fait la commande Unix `tr'.

8.1.4 Les commandes d'affichage : `p', `l', et `='

La commande `p' (**print**) affiche la ligne sélectionnée sur la sortie standard. Indispensable quand `sed` est invoqué avec l'option `-n'.

La commande `l' (**list**) affiche la ligne sélectionnée sur la sortie standard, avec en plus les caractères de contrôle en clair (``\t'' au lieu d'une tabulation, etc.) et le code ASCII (deux chiffres en octal) pour les caractères non imprimables.

```
42sh$ echo -e 'te\tst'| sed -n 'p'
te      st
42sh$ echo -e 'te\tst'| sed -n 'l'
te\tst\n$
42sh$
```

La commande `=' donne le numéro de la ligne sélectionnée sur la sortie standard.

8.2 Adresses

La plupart des commandes peuvent être précédées d'une adresse, dont le rôle est d'indiquer à quelles lignes des données la commande concernée s'applique. Les différents formats d'adresses utilisables sont les suivants :

`NUMBER' un nombre `n' précise que seule la nième ligne de données est concernée par la commande (les lignes de données sont numérotées à partir de 1) ;

`$' correspond à la dernière ligne du dernier fichier de données ;

`/expression/' sélectionne toutes les lignes contenant l'expression rationnelle `expression' ;

`\%expression%' a le même effet que le format d'adresse précédent, mais offre la possibilité d'encadrer l'expression rationnelle à l'aide de caractères différents du slash `/' (n'importe quel caractère peut être utilisé en lieu et place des signes `%') ;

`adresse1,adresse2' sélectionne toutes les lignes à partir de la première sélectionnée par `adresse1' et jusqu'à la première sélectionnée par `adresse2', inclusivement ; cependant, la recherche de `adresse2' ne commencera qu'à la ligne suivant `adresse1' si `adresse2' est une expression rationnelle ;

`adresse!' élimine toutes les lignes qui auraient été sélectionnées par adresse, et sélectionne les autres.

Voici trois exemples qui appellent la commande `p' sur les lignes sélectionnées :

```
/test/p  # affiche les lignes contenant ``test''
4,8p     # affiche les lignes 4 a 8
$!p      # affiche toutes les lignes sauf la derniere
```

8.2.1 Blocs de commandes : `{}'

On peut regrouper plusieurs commandes en une seule grâce aux blocs de commandes, en entourant les commandes par des accolades `{}', et en séparant chaque commande par un saut de ligne ou par un ` ;'.

Cela sert principalement quand de nombreuses commandes doivent s'appliquer aux mêmes lignes.

8.2.2 Remplacement de bloc : `c'

La commande `c' s'utilise de la façon suivante :

```
adresse1,adresse2 c\
texte
```

Elle permet de remplacer la ligne courante par le texte donné. Par exemple :

```
/imbécile/c\
  Ligne censurée
```

Cette commande remplacera toutes les lignes contenant ``imbécile'' par le texte ``Ligne censurée''.

8.3 Labels

Une autre caractéristique intéressante de `sed`, permettant par exemple de réaliser des boucles et des tests, est l'utilisation de labels. Un label peut être défini à l'aide de la syntaxe :

```
:label
```

Deux commandes mettent une telle déclaration à profit :

- la commande `b label' cause un saut immédiat à la position du script où label est défini (si label est omis, `sed` saute au début du script, affiche la ligne en cours de traitement si l'option `-n' n'a pas été donnée, et lit une nouvelle ligne de données) ;
- la commande `t label' cause un saut à la position du script où label est défini à condition qu'une commande `s' au minimum ait causé une substitution depuis la lecture de la dernière ligne ou le dernier saut à une commande `t'.

8.4 Commandes avancées

Il existe d'autres commandes, qui ne seront pas abordées ici, faisant intervenir un espace mémoire de stockage et permettant d'effectuer de nombreuses tâches avancées, en particulier on pourra travailler sur plusieurs lignes en même temps.

De plus il existe de nombreuses extensions GNU des différentes commandes présentées, ainsi que de nouvelles commandes. Bien que souvent très utiles, il est recommandé de ne pas les utiliser afin de conserver les scripts portables.

9 Exercices

9.1 count_files.sh

Commandes autorisées : Built-ins, cut, grep
Écrire les commandes permettant de répondre aux questions suivantes :

- Combien y-a-t'il de fichiers dans le répertoire */dev/*.
- Lister les groupes présent dans */etc/group*
- Lister les fichiers contenant le shebang *#!/bin/sh*

9.2 salutation.sh

Nom du fichier : salutation.sh Commande autorisée : built-ins, date, cut

Réaliser un shell script qui, en fonction de l'heure courante, affiche :

- ``Good morning !'' de 0h à 12h ;
- ``Good afternoon !'' de 12h à 17h ;
- ``Good evening !'' de 17h à 24h.

9.3 squeeze.sh

Nom du fichier : squeeze.sh Commande autorisée : built-ins, tr

Écrire un script qui remplace une suite d'espaces en un seul à l'argument qui lui est donné sur la ligne de commande, une étoile () par un point (). Note : ** devient ``..''

```
42sh$ ./squeeze.sh 'my    source      file     *c'
my_source_file_.c
```

Veillez à bien lire le man, une option de pourra vous être très utile.

9.4 get_line.sh

Nom du fichier : get_line.sh Commandes autorisées : sed, head, tail
Écrire un script qui permet de récupérer la -ième ligne d'un fichier.
Le premier argument passé au script sera le nom du fichier et le deuxième le numéro de la ligne à récupérer.
Si le numéro de ligne donné en argument est trop grand, on affichera la dernière ligne.

```
42sh$ cat file.txt
Ligne numero 1
Ligne numero 2
Ligne numero 3
Ligne numero 4
Ligne numero 5
Ligne numero 6
42sh$ ./get_line file.txt 3
Ligne numero 3
```

9.5 log.sh

Nom du fichier : log.sh Commande autorisée : Built-ins, tee

Vous devez écrire un script qui prend en entrée une commande et ses arguments et l'exécute. Vous devez :

- rediriger sa sortie standard dans le fichier *log.out* ;
- afficher sa sortie standard sur la sortie standard ;
- rediriger sa sortie d'erreur dans le fichier *log.err* ;
- rien ne doit être affiché sur la sortie d'erreur.

```
42sh$ ./log.sh /bin/echo toto
toto
42sh$ cat log.err
42sh$ cat log.out
toto
42sh$ ./log.sh ls unknown
42sh$ cat log.out
42sh$ cat log.err
ls: unknown: No such file or directory
```

9.6 sum.sh

Nom du fichier : sum.sh Commande autorisée : Built-ins, sed, tr

Écrire un script qui calcule la somme de tous ses arguments, 0 s'il n'en a aucun. On admettra que tous les arguments sont des entiers valides. On affichera le détail de l'opération suivi du résultat.

```
42sh$ ./sum.sh 3 4 5
3 + 4 + 5 = 12
42sh$ ./sum.sh
0 = 0
42sh$ ./sum.sh 3
3 = 3
```

9.7 Correcteur orthographique

Le fichier `/usr/share/dict/words' contient un dictionnaire pour la langue anglaise. On veut vérifier qu'un texte ne contient pas de fautes. Écrivez un script qui affiche dans l'ordre alphabétique l'ensemble des mots du fichier donné en argument qui ne sont pas présents dans le dictionnaire. Il est évident que la vérification ne doit pas tenir compte de la casse.

Quelques conseils :

- Le script devrait pouvoir tenir en 3 lignes. Vous pouvez en utiliser plus si vous voulez, mais ne partez surtout pas dans du code long et compliqué.
- Il n'est pas utile d'utiliser des programmes autres que tr, sort et grep.
- Lire les ``man'' est toujours une bonne idée.

9.8 Exercices en Sed

Pour ces quelques exercices, vous devez programmer en `Sed', exclusivement. Vous pouvez mettre les commandes soit dans un fichier `.sed', soit directement en ligne de commande de votre Shell.

9.8.1 Trailing whitespace

On souhaite supprimer les espaces à la fin des lignes d'un fichier.
Écrivez un script pour accomplir cette tâche. Pensez aussi à supprimer les tabulations à la fin des lignes.

```
42sh$ cat -e test
Ligne 1$
Ligne 2    $
ligne     3 $
42sh$ sed -f deletetw.sed test | cat -e
Ligne 1$
Ligne 2$
ligne     3$
42sh$
```

9.8.2 Grep

On souhaite utiliser `sed' à la place de `grep'. Écrivez un script shell appelant sed qui affiche uniquement les lignes correspondant à l'expression rationnelle de votre choix. Modifiez ensuite votre script pour simuler l'option `-v' de `grep'.

9.8.3 Head

On souhaite simuler la commande `head' à l'aide de `sed'. Écrivez un script `sed' qui affiche les 5 premières lignes de l'entrée.

9.8.4 Head | Tail

On souhaite maintenant afficher quelques lignes au milieu d'un fichier (c'est équivalent à appeler `head' puis `tail'). Écrivez un script qui affiche uniquement les lignes 25 à 30.

9.9 Compter en binaire

On souhaite réaliser un script qui compte en nombres binaires. Votre script doit lire un nombre donné (sur l'entrée standard) et afficher son successeur.
Astuce : pour gérer la retenue, il peut être utile de remplacer les 1 de la fin par un autre caractère, temporairement.

```
42sh$ echo '10' | ./binary.sed
11
42sh$ echo '11' | ./binary.sed
100
42sh$ echo '101001' | ./binary.sed
101010
42sh$
```

Proposer une modification du script pour qu'il affiche tous les nombres binaires de 0 à 1100.

```
42sh$ echo '0' | ./bcompteur.sed
0
1
10
11
100
101
110
111
1000
1001
1010
1011
1100
42sh$
```

Why so sleepy?