Polytech'Nice-Sophia **Année 2005-2006**
Département SI 1 Marc Gaëtano
Introduction à Unix

Programmation shell

Les exercices de cette feuille de TD illustrent la programmation de script shell en **sh** (le Bourne Shell). Les fichiers correspondants aux commandes que vous devez écrire doivent commencer par la ligne `#!/bin/sh` qui indique à Unix que le shell à utiliser est le Bourne Shell. Une `liste des principales commandes à utiliser` est indiquée en début de chaque exercice.

Exercice 1 `$#, $@`

Ecrivez la commande `args` qui prend un nombre quelconque d'arguments quelconques et qui, lorsqu'on l'exécute, affiche son nom, le nombre de ses arguments et ses arguments :

```
$ args fichier $USER 777 tutu
args est appelee avec 4 argument(s) : fichier gaetano 777 tutu
```

Exercice 2 `date, echo, if`

Ecrivez la commande `welcome`, commande sans argument qui affiche un message de bienvenue dépendant de l'heure de la journée :

```
$ welcome
Good morning gaetano, you're logged on wolfix and your current directory is /home/gaetano/
```

Le message est `Good morning` entre 0 heure et 12 heures, `Good afternoon` entre 13 heures et 18 heures et `Good evening` entre 19 heures et 23 heures.

Exercice 3 `cat, tr, sort, uniq`

Écrivez la commande `words` qui affiche chaque mot d'un fichier de texte précédé de son nombre d'occurrences. Les mots doivent apparaître triés par ordre alphabétique à raison d'un mot par ligne.

Exercice 4 `if, test`

Écrivez la commande `info` qui affiche les informations de l'unique fichier ou répertoire qui lui est passé en paramètre. Cette commande devra afficher par exemple :

```
$ info tutu                          $ info /usr/local/bin/java
tutu n'existe pas                    /usr/local/bin/java est un lien symbolique
$ info .zlogin                       $ info /local
.zlogin est un fichier (mode rw)     /local est un répertoire
```

Si la commande `info` est appelée avec zéro ou plus d'un paramètre, elle doit s'arrêter sur une erreur :

```
$ info tutu titi tata
usage: info FILE
```

Exercice 5 `for, expr`

Ecrivez la commande `index` qui prend au moins un argument et qui retourne la position de ce premier argument dans la liste des arguments qui suivent :

```
$ index X A B X C D
3
```

Si le premier argument n'apparaît plus parmi les autres arguments, `index` retourne 0, et s'il apparaît plusieurs fois parmi les autres arguments, `index` retourne l'indice de la première occurence :

```
$ index tutu titi tata toto    $ index onlyone    $ index aa xx aa yy zz aa ww aa
0                              0                  2
```

Exercice 6 `if, test, for`

Reprenez la commande `info` de l'exercice 4 afin qu'elle prenne maintenant un nombre quelconque d'arguments (au moins un), et qu'elle affiche les informations relatives à chacun de ses arguments.

Exercice 7 `if, test`

Écrivez la commande `xdiff` qui permet de visualiser les différences entre deux fichiers sous `xemacs` en utilisant le mode *ediff*. Sous `xemacs`, le mode `ediff` est lancé de façon interactive par la commande `M-x ediff` mais on peut aussi à la fois lancer `xemacs` et exécuter la commande `ediff` depuis le shell en invoquant la commande suivante:

```
xemacs -eval '(ediff "fichier1" "fichier2")'
```

Attention, les guillemets autour de *fichier1* et *fichier2* sont obligatoires ! Modifiez votre commande pour qu'elle puisse afficher les différences entre deux répertoires si les paramètres sont des répertoires. La différence de deux répertoires se fait avec la commande `xemacs -eval '(ediff-directories "repertoire1" "repertoire2" "")'`

Exercice 8 `if, grep, ps`

La commande **gnuclient** associée à un processus `gnuserv` lancé par `xemacs` avec l'option `-f gnuserv-start` permet d'ouvrir une nouvelle *frame*, c'est à dire une nouvelle fenêtre `xemacs` indépendante.

Ecrivez la commande `xframe` qui, si aucun processus `gnuserv` appartenant à l'utilisateur n'existe, lance un tel processus (avec la commande `xemacs -f gnuserv-start`) ou bien exécute la commande **gnuclient**. Faites apparaître ensuite une nouvelle icône sur la barre des tâches de votre KDE et liez-la à la commande `xframe`.

Exercice 9 `if, cat, case, zcat`

Écrivez la commande `viz` qui prend un fichier en argument et le visualise. Le programme de visualisation sera choisi en fonction de l'extension du fichier. Votre commande doit traîter les fichiers d'extension `.txt` (fichier de texte), `.ps` (fichier PostScript), `.pdf` (fichier au format PDF) et *ext*`.gz` (fichier compressé) avec *ext* une des trois extensions précédentes. Pour les fichiers compressés, considérez la commande **zcat**. Si le fichier n'a pas une des extensions précédentes, la commande `viz` termine sur une erreur. Un fichier de texte sera affiché par la commande `more` dans une nouvelle fenêtre (étudiez la commande Unix **xterm**).

Exercice 10 `cal, case, date, if, set, while`

Ecrivez la commande `cal` qui améliore la commande `/usr/bin/cal` en autorisant de qualifier les mois à l'aide de l'un des mots `jan`, `feb`, `mar`, `apr`, `may`, `jun`, `jul`, `aug`, `sep`, `oct`, `nov` et `dec`, et ce indépendemment de la casse (i.e. `jan`, `Jan`, `JAN`, `jAn`, etc. sont équivalents). La nouvelle commande `cal` doit se comporter comme `/usr/bin/cal` et en particulier avoir les mêmes options. Quand le seul argument est le mois en lettres, l'année est l'année en cours :

```
$ cal dec
December 2005
Su Mo Tu We Th Fr Sa
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```

Exercice 11 `case, expr, read, test`

Écrivez la commande `myhead`. Cette commande devra simuler le fonctionnement de la commande **head** de Unix. Comme la commande Unix, votre programme devra accepter de façon optionnelle le nombre de lignes à afficher ou bien lire ce nombre dans la variable d'environnment `MYHEAD_DEFAULT`.

Exercice 12 `expr, read, while`

Écrivez la commande `myuniq` qui recopie sur le fichier standard de sortie les lignes lues sur le fichier standard d'entrée en supprimant les lignes consécutives identiques. Chaque ligne écrite sur le fichier standard de sortie sera précédée par son nombre d'apparitions (i.e. votre commande doit fournir un résultat identique à celui produit par la commande **uniq** avec l'option *-c*).

Exercice 13 `column, cut, ls, tr, while`

Ecrivez la commande `size` qui prend en arguments des fichiers et qui, pour chacun, affiche sa taille en (kilo-)octets suivie de son nom. La commande `size` affiche `???` à la place de la taille en kilo-octet si le fichier n'est pas une fichier régulier :

```
$ size *
???   psdir
???   scriptdir
8     sujet.aux
14K   sujet.dvi
3.4K  sujet.log
70K   sujet.ps
14K   sujet.tex
475   unix.sty
```

Exercice 14 `cut, find, tr, while`

Écrivez la commande `chklnk` qui prend en paramètre un répertoire et qui affiche tous les liens symboliques invalides de ce répertoire (et récursivement de ses sous-répertoires). Un lien symbolique invalide est un lien qui ne pointe pas sur un fichier ou un répertoire existant.

Exercice 15 `grep, if, ls, nm, read, while`

Ecrivez la commande `afind` qui prend en argument un répertoire et un nom et qui cherche dans quelles bibliothèques C appartenants au répertoire, le nom est défini. Les bibliothèques sont des fichiers d'extension `.a` et on trouve en général les bibliothèques standards sous `/usr/lib`. On utilisera la commande `nm` pour connaître les noms externes définis par une bibliothèque.

Exercice 16 `grep, if, ls, read, while`

Ecrivez la commande `hfind` qui prend en argument un répertoire et un nom, et qui cherche récursivement dans le tout le répertoire, dans quels fichiers profil C le nom est défini. Les fichiers profils ont l'extension `.h`. La plupart des fichiers profil C de la librairie standard sont sous le répertoire `/usr/include` (et sous certains de ses sous-répertoires)

```
$ hfind /usr/include compare
/usr/include/arpa/nameser.h
/usr/include/asm/system.h
/usr/include/c++/3.2.2/bits/stl_alloc.h
....
```

Exercice 17 `basename, case, for, if, ls, mv, read`

Ecrivez la commande `chext` qui prend en arguments deux extensions `ext1` et `ext2` et un nom de répertoire, et qui renomme tous les fichiers de nom `nom.ext1` du répertoire en `nom.ext2`. Si le nom du répertoire n'est pas fourni, la commande s'applique au répertoire courant. Modifiez ensuite la commande `chext` pour qu'elle admette l'option `-i` et demande confirmation avant chaque renommage.

Exercice 18 `eval, mv, tr, while`

Ecrivez la commande `nospace` qui prend en paramètre des noms de fichiers (au moins un) et qui, quand le fichier existe et quand son nom contient le caractère *espace*, renomme le fichier en remplaçant toute les occurences du caractère *espace* par le caractère *souligné* (`'_'`) :

```
$ ls
Le\ fichier\ de\ tata Le\ fichier\ de\ toto titi tutu
$ nospace *
Le_fichier_de_tata Le_fichier_de_toto titi tutu
```

Exercice 19 `case, for, grep, kill, ps, read, tr`

Ecrivez la commande `kproc` qui prend en argument des mots et qui tue interactivement tous les processus dont le nom contient un de ces mots. Pour chacun des processus, la commande `kproc` affiche toutes les informations (celles affichées par la commande Unix **ps**) et demande confirmation à l'utilisateur.

Exercice 20 `basename, case, if, read, test, while`

Écrivez la commande `move`, commande strictement identique à la commande `mv`, mais n'admettant **aucune** option. Cette première version de `move` prendra exactement **deux** arguments.

Exercice 21 `basename, case, if, read, test, while`

Modifiez la commande `move` de l'exercice précédent pour qu'elle prenne maintenant deux ou plusieurs arguments.

Exercice 22 `basename, dirname, case, test, for`

Le but de cet exercice est d'écrire un jeu de scripts permettant de gérer simplement des *sauvegardes* de fichiers.

Ecrivez la commande `backup` qui prend comme arguments des fichiers ou des répertoires, et qui en crée des copies de nom `xxx.bak` si le nom du fichier ou du répertoire est `xxx`. Ecrivez la commande symmétrique `restore` qui prend comme des fichiers ou des répertoires, et qui les remplace par leur copie. Après restauration, les copies sont effacées.

Afin d'éviter les sauvegardes de sauvegardes de sauvegardes,..., on **interdit** de sauvegarder un fichier de sauvegarde avec la commande `backup`. A l'inverse, pour faciliter l'utilisation de la commande `restore`, on peut l'appeler indifféremment avec le nom du fichier à restaurer ou avec le nom du fichier de sauvegarde. Autrement dit, `restore xxx` et `restore xxx.bak` sont équivalents.

Modifiez les commandes `backup` et `restore` pour qu'elles admettent maintenant l'option `-r` (comme `r`écursive). Si cette option est présente, les arguments qui sont des répertoires sont traîtés récursivement et les fichiers qu'ils contiennent sont sauvegardés/restaurés.

Exercice 23 `basename, case, date, read, test, while`

Concevez et écrivez un ensemble de scripts comprenant au minimum les commandes `trash`, `intrash`, `untrash` et `cleantrash` qui permettent de mettre des fichiers à la poubelle, et de pouvoir éventuellement les récupérer. Le système ne se limite pas à déplacer les fichiers dans un répertoire spécial au lieu de les effacer. Par exemple, on peut mettre à la poubelle consécutivement deux fichiers de même nom (relatif ou absolu) et on désire les conserver tous les deux dans la poubelle.

La commande `trash` se comporte comme `rm` mais déplace les fichiers vers un répertoire *poubelle* au lieu de les effacer. Vous pouvez vous limiter dans un premier temps aux deux options les plus utiles, `-i` et `-r` et ajouter ensuite l'option `-f` (voir `man rm`).

La commande `intrash` permet de lister de manière lisible et pertinente le contenu de la poubelle.

La commande `untrash` permet de récupérer simplement un fichier contenu dans la poubelle à partir des informations fournies par `intrash`.

La commande `cleantrash` permet de vider tout ou partie de la poubelle en fonction de certains paramètres. Une version simple de `cleantrash` consiste à effacer (définitivement) de la poubelle tous les fichiers qui y sont depuis plus d'un certain temps (par exemple un mois). Cette commande peut être invoquée au début de chaque login.

L'emplacement du répertoire poubelle peut être contenu dans la variable d'environnement `TRASHDIR`.

Exercice 24 `basename, case, date, read, test, while`

Certaines commandes Unix modifient leur environnement, comme `rm`, `cp` ou `mv` pour ne citer que les plus communes. Il serait utile de disposer d'un mécanisme d'*annulation* d'une telle commande qui permette de revenir à l'état initial. C'est ce que nous vous proposons de réaliser via la commande `undo`.

Le principe de la commande `undo` est le suivant : par exemple, si on veut pouvoir annuler l'effet de la commande `'mv xxx yyy'`, il faut que la commande `mv` *elle-même* produise sans l'exécuter la commande *annulante* `'mv yyy xxx'`. La commande annulante peut être placée dans un script standard qu'on nommera judicieusement `undo`. Le chemin absolu de ce script doit être la valeur d'une variable d'environnement. Ainsi, les effets de la commande `'mv xxx yyy'` peuvent être annulés par la commande `'mv yyy xxx'` (exécutée via la commande `undo`), à condition que le fichier `yyy` n'existe pas au moment de l'exécution de la commande `mv`. Dans le cas contraire, la commande `'mv xxx yyy'` doit effectuer une sauvegarde du fichier `yyy`, par exemple avec une commande du type `'mv yyy ~/tmp/yyy'`, et les commandes annulantes sont `'mv yyy xxx'` suivie de `'mv ~/tmp/yyy yyy'`. Pour résumer, on dira que c'est la commande à (éventuellement) annuler qui, connaissant ses propres paramètres, est à même de produire les commandes suceptibles d'annuler ses effets !

En utilisant la méthode décrite précédemment, écrivez les commandes `uXXX`, versions *annulables* des commandes `XXX` où `XXX` prend au moins les valeurs `rm`, `mv` et `cp`.