Unix : concepts et outils avancés
TD no 3 : scripts shell, suite

Olivier Aumage
LIP, bureau 302
email : `Olivier.Aumage@ens-lyon.fr`

10 octobre 2000

1 Redirections

En fonctionnement interactif normal, l'entrée et la sortie standard d'une commande sont *reliées* à la console (l'entrée est reliée au clavier et la sortie est reliée à l'écran). Le shell `sh` dispose de plusieurs opérateurs permettant de *rediriger* les entrées et sorties standards des commandes, c'est-à-dire de les *brancher* sur des flux de données (ou des fichiers) différents de ceux par défaut.

Entrée standard L'opérateur `<fichier` redirige l'entrée standard de la commande sur le fichier `fichier`. En conséquence, la commande correspondante lira le contenu du fichier en lieu et place du clavier. Exemple :

```
% sort <file.txt
```

Une variante un peu particulière de cet opérateur est l'opérateur *here-document* dont l'utilisation (généralement dans un script) est la suivante (ici avec la commande `cat`) :

```
% cat <<[-]['']mot['']
texte1
texte2
texte3
mot
```

Le texte compris entre la première ligne suivant la première occurrence de `mot` et la ligne précédent la seconde occurrence de `mot` est considéré comme un fichier et est redirigé sur l'entrée standard de la commande (ici `cat`). Le caractère `'-'` optionnel après l'opérateur entraîne la suppression des tabulations en début de chaque ligne. Les guillemets simples optionnels autour de `mot` indiquent que les symboles spéciaux contenus dans le texte du document ne doivent pas être interprétés. De plus, ils activent un fonctionnement spécial du caractère *backslash* lorsqu'il précède un retour à la ligne : le retour à la ligne est supprimé et les deux lignes sont fusionnées. En l'absence des guillemets, ce comportement est désactivé.

Sortie standard L'opérateur `>fichier` redirige la sortie standard sur le fichier `fichier`. Tous les messages préalablement destinés à être affichés sur l'écran seront écrits dans le fichier spécifié. Exemple :

```
% echo 10 >num.txt
```

L'opérateur `>>fichier` redirige également la sortie standard sur le fichier `fichier`. En revanche, et contrairement à >, si le fichier existe déja il n'est pas détruit et les messages sont simplement *ajoutés* à son contenu. Exemple :

```
% echo 20 >>num.txt
```

Tubes L'opérateur `commande1|commande2` (aussi appelé *tube* ou *pipe*) redirige la sortie de la commande `commande1` sur l'entrée de la commande `commande2`. Exemples :

```
% ls | wc -l
% ps -aef | grep netscape
```

Questions

– Essayez les exemples ci-dessus et constatez les effets de chacun des opérateurs de redirection. Que font les deux exemples d'utilisation des tubes ?
– Que constatez-vous si vous exécutez `ls >/dev/null`. Qu'en déduisez-vous sur la propriété du fichier `/dev/null` et sur l'usage des redirections vers `/dev/null` ?
– Quel est le résultat de `cat </dev/null >fichier.txt`. Qu'en déduisez-vous sur un usage possible de cette commande ?
  Attention : ne pas confondre `/dev/null` et `/dev/zero` ! Essayez de déduire la propriété de `/dev/zero` (et pourquoi ce n'est pas une bonne idée de l'utiliser dans l'exemple précédent) en testant les deux commandes suivantes (vous pouvez aussi les essayer sur `/dev/null` pour voir) :

  ```
  % cat </dev/zero
  % cat </dev/zero | tr '\000' '0'
  ```

  Utilisez `man tr` pour les détails sur la commande `tr`.
– Observez le résultat de `ls >ls.txt`. Que constatez-vous ? Qu'en déduisez-vous sur le fonctionnement du shell ?
– Les commandes `head` et `tail` affichent respectivement le début et la fin d'un fichier (cf. `man` pour les options). Ecrivez un script `milieu <debut> <fin> <fichier>` qui met à profit ces deux commandes pour afficher les lignes de `debut` à `fin` incluses du fichier `fichier`.

2 Commande `read`

La commande `read` permet de lire une ligne sur l'entrée standard et de la stocker dans une variable. Elle s'utilise généralement de la manière suivante :

```
while read a
do
    ... travail sur le contenu de a ...
done
```

La boucle s'arrête lorsque la *fin du fichier* est atteinte si l'entrée standard est redirigée sur un fichier. Dans le cas contraire la boucle s'arrête lorsque `read` lit le *caractère de fin de fichier* (`CTRL-d` au clavier).

Questions

- Écrivez un script `filtre_ls` tel que la commande `ls -l * | ./filtre_ls.sh` affiche la somme de la taille des fichiers listés par `ls -l *`.
- Pourquoi le caractère '`*`' est-il nécessaire ici ?
- Proposez une correction à ce problème.

3 Construction `case`

La construction `case` permet à un script shell de comparer une chaîne de caractères à une série d'expressions régulières décrivant des motifs (*patterns* en anglais) et d'associer une action particulière à la correspondance de la chaîne avec l'un de ces motifs. L'utilisation en est la suivante :

```
case <word> in
    pattern1) echo '1' ;;
    pattern2|pattern3) echo '2'; echo 'ou'; echo '3' ;;
    ...
esac
```

Si le mot `word` correspond au motif `pattern1` le script affiche `1` (**et non `1 2 ou 3` contrairement au C**). Si `word` correspond à `pattern2` ou à `pattern3`, le script affiche `'2 ou 3'` (sur 3 lignes pour être précis). Chaque série de commandes est terminée par `;;`. Les motifs peuvent utiliser les caractères spéciaux `*`, `?` et `[ ]`.

Question

- Écrivez un script `undup.sh` qui lit l'entrée standard ligne par ligne et écrit chaque ligne sur la sortie standard (c'est ce que l'on appelle un *filtre*), après avoir supprimé tous les mots redondants de la ligne (i.e. pour chaque ligne, on ne garde que la première occurrence de chaque mot). Vous aurez peut-être besoin des commandes étendues de substitution de paramètre décrites dans le `man` du `sh` de Solaris à la section *Parameter Substitution*. On suppose que les mots sont séparés par des espaces.

4 Fonctions

Comme de nombreux autres langages, les scripts `sh` ont une notion de fonction dont le but est de faciliter la structuration des scripts et la réutilisation de code. Une fonction `sh` est définie de la manière suivante :

```
my_f() {
    ... code de la fonction
}
```

L'appel d'une fonction `sh` est identique à un appel de commande, à ceci-près qu'un appel de fonction ne crée pas de processus supplémentaire. À l'appel de la fonction, la liste d'arguments courante (`$*`) est sauvée et remplacée par les arguments de la fonction. La liste d'arguments est restaurée à la sortie de la fonction. Il est possible de forcer un retour de fonction avant la fin de l'exécution de son code à l'aide de la commande `return`. On peut spécifier un entier optionnel comme paramètre de `return`. Cet entier constitue dans ce cas le *code d'erreur* (ou *code de retour*) de la fonction.

Attention ! Les variables des scripts `sh` sont toutes globales exceptée la liste d'arguments. Il faut tenir compte de cette particularité, notamment lors d'appels récursifs ou concurrents.

Questions

- écrivez un script `moyenne.sh` qui analyse un fichier `notes` dont chaque ligne est composée d'un nom et d'un prénom d'élève et d'une série de notes (de longueur arbitraire, mais non nulle), et liste la moyenne de chaque élève (ainsi que son identité).
- modifiez le script `moyenne.sh` pour que le listing produit soit ordonné par ordre décroissant des moyennes et que chaque entrée soit listée dans l'ordre `nom`, `prenom`, `moyenne`. La commande `sort` pourra avantageusement être utilisée pour la phase de tri (utilisez les options `-n` et `-r`, cf. `man sort`).
- modifiez `moyenne` pour afficher la moyenne avec un chiffre après la virgule.
- écrivez un script `semblable.sh` qui prend deux répertoires en arguments et compare les deux arborescences : il doit indiquer si les deux arborescences sont identiques ou non, au niveau des noms de fichiers et de leur contenu. Le script terminera sa recherche dès la découverte d'une différence. Vous pourrez utiliser la commande `cmp` (cf. `man cmp`) pour comparer deux fichiers.

— ❃ —