SVM Mac 160 et 161

# Flash-back sur le Shell.

**Faisons un « break » dans notre parcours initiatique et attardons nous sur les symboles spéciaux du shell le temps d'une page. Nous nous limiterons à la portion congrue tant la richesse des interpréteurs de commandes Unix est grande en la matière.**

Yannick Cadin

```
$ cd dossier ; ls
```
Le point-virgule permet simplement l'exécution de commandes en séquence.

```
$ ls ~/?rticles/SVMMac/*.pdf
```
Lister tous (*) les fichiers au format PDF (tout au moins ceux dont le nom se termine par la séquence de caractères « .pdf ») dans le sous-dossier SVMMac de tout dossier débutant par n'importe quel caractère (?) lui même situé dans mon répertoire de départ (symbolisé par ~).

```
$ ls ~/?rticles/SVMMac/*199[6-8].pdf
```
Idem mais nous limitons notre recherche aux seuls documents mentionnant les séquences 1996, 1997 ou 1998 avant le suffixe .pdf. ATTENTION, l'intervalle défini de cette façon porte sur des caractères et NON sur des chiffres, encore moins sur des nombres. Ainsi l'expression `*19[96-98].pdf` est erronée. Plus exactement elle est remaniée en `*19[967898].pdf` puis simplifiée et comprise comme `*19[6-9].pdf` par notre interpréteur de commandes.

```
$ ls ~/?rticles/SVMMac/*{1998,1999,2000,2001}.pdf
```
Cette fois, nous listons les fichiers pour chacune des années mentionnées entre accolades.

```
$ mv *.{gif,jpg,png} dossierImages
```
Autre exemple d'utilisation des accolades pour déplacer tous les fichiers possédant un suffixe .gif ou .jpg ou encore .png dans un sous-dossier.

```
$ grep -i title *.[Hh][Tt][Mm]
```
Unix étant un système qualifié de « case sensitive », il nous faut faire preuve de vigilance si nous ne voulons pas oublier des documents en cours de route. Ici la recherche du texte title se fera dans tous les fichiers d'extension HTM qu'elle soit écrite .htm, .htM, .hTm, .hTM, .Htm, .HtM, .HTm ou encore .HTM.

```
$ mv 'avec espaces.txt' sans_espace.txt
$ mv "second avec espaces.txt" second_sans_espace.txt
$ mv ultime\ avec\ espaces.txt ultime_sans_espace.txt
```
`'' "" \` interdisent (parfois limitent seulement) l'interprétation de caractères spéciaux aux yeux du shell. Soit en les plaçant à l'intérieur de chaînes délimitées par des apostrophes ou des guillemets, soit en les faisant immédiatement précéder par une barre de fraction inverse (\, le backslash). Il existe une différence subtile entre guillemets et apostrophes qui réside dans l'interprétation ou non (respectivement) des variables introduites par leur caractère $ à l'intérieur des chaînes ainsi délimitées.

```
$ expr 3 + 9
$ expr 3 \* 9
```
Calcule les expressions 3+9 puis 3x9. Je vous propose un contre-exemple pour bien montrer les subtilités d'interprétation des caractères spéciaux du shell. Essayez
```
$ expr 3 * 9
```
puis
```
$ mkdir bizarre ; cd bizarre ; touch + ; expr 3 * 9
```
Je vous laisse cogiter et comprendre comment le shell a compris la dernière commande.

```
$ cat premierFichier secondFichier > unSeulFichier
```
ConCATénation de deux fichiers dont le résultat est dirigé (RE-dirigé devrais-je dire) dans unSeulFichier à l'aide de l'opérateur de redirection en sortie (>). En l'absence de cette redirection, cat, à l'instar d'un grand nombre de commandes Unix, vous affiche le résultat directement dans votre fenêtre Terminal (qui constitue sa « sortie standard » par défaut).

```
$ tr '[a-z]' '[b-za]' < fichier > fichier_confidentiel
```
Certaines commandes, comme tr, vont lire et écrire les données à traiter exclusivement sur leurs entrée et sortie standard (respectivement ce que vous saisissez au clavier et votre fenêtre Terminal par défaut), ce qui nous oblige à employer des caractères tels que < et > pour rediriger les flux de données en provenance et à destination de fichiers.
Vous pouvez essayer sans pour vous en rendre compte (vous pouvez interrompre le traitement avec la combinaison de touches Ctrl-d qui symbolise une fin de fichier.).
Vous pourrez décrypter votre document avec… (Pas d'impatience ! La suite le mois prochain.)

```
$ tr '[b-za]' '[a-z]' < fichier_confidentiel
```
À noter : les expressions [b-za] et [a-z] sont interprétées par tr et NON par le shell ! Vous pouvez remarquer qu'elles sont entourées d'apostrophes pour les « protéger ».

```
$ find / -name fichier_perdu 2> /dev/null
```
De même que les résultats de traitements, les messages d'erreur sont affichés sur votre fenêtre Terminal qui constitue la « sortie d'erreur standard » par défaut. Ici nous indiquons au shell qu'il doit envoyer les messages d'erreur (2>) générés par les actions de la commande find vers le néant (représenté par le fichier spécial /dev/null).
Remarque : cette syntaxe n'est pas valide avec le shell (tcsh) fourni par défaut dans les versions précédant Panther.

```
$ echo Titres des documents HTML : > liste_titres.txt
$ grep -i title *.[Hh][Tt][Mm] >> liste_titres.txt
```
Nous créons un fichier liste_titres.txt débutant par un entête avec une commande echo dont nous redirigeons (>) l'affichage. Nous y ajoutons ensuite (>>) le résultat d'une commande grep.

```
$ ls *.pdf | wc -l
```
Dénombre les fichiers au format PDF dans le dossier courant. Toujours le principe des redirections mais cette fois la destination est l'entrée standard d'un programme (wc). Pour lever l'ambiguïté et ne pas créer un fichier wc contenant la liste des fichiers PDF, nous devons utiliser la barre verticale (|) au lieu de >. Ce mode d'acheminement des données entre deux programmes est appelé tube de communication (pipe en anglais).

Cet exemple est un des archétypes de la philosophie Unix. D'un côté, une commande qui ne sait que lister des fichiers, mais qui le fait le mieux possible. De l'autre, une commande qui ne sait que dénombrer. On compartimente dans un nombre important de commandes distinctes des traitements aux effets limités. Entre elles, un flux de données acheminé par un mécanisme de « pipeline ».

```
$ (echo Titres des documents HTML : ; grep -i title
*.[Hh][Tt][Mm]) > liste_titres.txt
```

Une réécriture d'un exemple précédent. Les parenthèses permettent de combiner les flux de données de différents processus. Ainsi ici, ce seront les résultats de la commande echo puis de la commande grep qui seront dirigés vers le fichier liste_titres.txt

```
$ destination=/chemin/vers/un/dossier/dont/le/nom/est/tres/long
$ cp fichier.txt $destination ; ls $destination
```

Le caractère $ sert à préfixer les noms des variables. Inutile d'être programmeur pour comprendre l'intérêt qu'elles offrent dans les manipulations de tous les jours.
Note : ici encore, l'affectation d'une variable requiert une syntaxe différente dans le shell livré avant Panther.

```
$ for fichier in *.pdf ; do mv $fichier ${fichier}_ANCIEN ; done
```

Autre usage des accolades, elles peuvent servir à délimiter le nom d'une variable. Si nous les omettions dans l'expression ci-dessus, le shell considérerait alors `fichier_ANCIEN` comme étant le nom de la variable.
Re-note : encore désolé, il faut préciser que la structure de contrôle for ne porte pas le même nom et n'adopte pas la même syntaxe sous tcsh (le shell fourni avant Panther).

```
$ echo Nous sommes le : `date +%d/%m/%Y`
```

Caractères précieux, les apostrophes inverses demandent l'évaluation par l'interpréteur de l'expression qu'elles délimitent.
Par opposition, vous verrez immédiatement la différence de comportement en exécutant

```
$ echo Nous sommes le : date +%d/%m/%Y
```

```
$ for fichier in *.pdf ; do mv $fichier `echo $fichier | tr ':'
'_'` ; done
```

Nous renommons (mv) chaque fichier (boucle for) d'extension .pdf de sorte à remplacer tous les caractères deux-points que son nom pourrait contenir en soulignés à l'aide de la commande tr appliquée sur son nom que lui communique la commande echo. Remarquez l'emploi combiné des caractères spéciaux `* $ `` | '' et ;`