Programmation Bourne Shell
(in a noix de coco !)

La programmation shell consiste à utiliser un interpréteur pour appeler les bonnes commandes UNIX. Le langage shell proprement dit ne sert que à manipuler les variables et les structures de test. Bien programmer en shell c'est avant tout connaître beaucoup de commandes Unix.
Tout peut se tester en lançant le bon shell et en entrant directement les commandes.

Le fichier programme :(coquillage)

- Il s'agit d'un fichier texte (vi coquillage)
- Sa première ligne est #!/bin/sh où sh est le programme shell choisi
- Ces droits sont rwxr-xr-x on les obtient avec chmod a+rx,u+w coquillage
- on teste avec ./coquillage

Les variables

- Les variables ne se déclarent pas, elles s'affectent :
  TOTO=Bonjour
  ESPACE=' '
  CHKOUN=Monsieur
- La valeur d'une variable s'exprime avec le $:
  echo $TOTO donne Bonjour
- La concaténation est naturelle :

  TUTU=$TOTO$ESPACE$CHKOUN
  echo $TUTU donne Bonjour Monsieur

- Les variables peuvent être transmises entre processus via l'environnement (export, env):

  TUTU=essai
  export TUTU
  env TUTU=essai

- Les variables se lisent au clavier avec read
  read TOTO

- Les variables sont case sensitive, toto et TOTO sont différents. C'est la meilleure source d'erreur que je connaisse.
- Il y a des variables définies par le père :

$*,$@	Tous les arguments
$#	Nombre d'argument
$$	Pid du shell courant
$?	Valeur de retour du dernier processus exécuté
$0,$1,$2,..	Nom du shell, argument 1, argument 2, ...

- Il y a des variables système :

PATH	Chemin de recherche des exécutables
PWD	Répertoire courant
PS1	Prompt du shell
HOSNAME	Nom de la machine
HOME	Répertoire de l'utilisateur
IFS	Valeur du séparateur du shell
CDPATH	Chemin de recherche des répertoires

Les expressions régulières (*,[,...)

Elles servent à désigner à l'interpréteur des éléments d'une sous liste contenant une expression :

- * designe n'importe quelle chaîne
- une expression entre [ ] désigne un caractère par exemple :
  [A-Z] les majuscules
  [aBC-F] les caractères a,B et de C aF
  [^p^P^R]* pas de p,P et R
  [^0-9] pas de chiffre
- les expressions se juxtaposent pour former les chaînes à reconnaître. Par exemple [a-b][0-9][0-7]* reconnaît a00 à b77.

C'est bien pratique et cela s'emploie de la façon : LISTE=`*.[ch]` pour constituer la liste des *.c *.h et le ranger dans une variable. La commande Unix grep a son propre jeu d'expression qu'il est bon de connaître.

Les quotes, l'échappement et les descentes de liste

Les quote permettent de modifier le comportement du shell. Il y a trois sortes de quote : ',",`

- ' (accent aigü) sert à bloquer toute interprétation du shell
  echo $TOTO afiche $TOTO
- " (double quote) permet de bloquer l'interprétation sauf pour $
  TOTO=pouet
  echo " pouet $TOTO"
  affiche pouet pouet
- ` (accent grave) permet d'exécuter la commande qu'il encadre et d'utiliser la sortie comme paramètre ou valeur :
  echo La date est `date +%D` affiche
  La date est 02/17/07

On utilise souvent les ` (grave) pour remplir des listes rapidement :

LFILE=`ls -R`

La redirection des entrées sorties

Les processus Unix ont trois flux de données prédéfinis, l'entrée standard, la sortie standard et la sortie d'erreur.

- <,<< redirige l'entrée standard depuis un fichier ou un pipe nommé
- >, >>redirige la sortie standard depuis un fichier ou un pipe nommé
- | redirige la sortie standard du processus a gauche vert l'entrée standard du processus à droite :
  ls | more
- 2> redirige la sortie d'erreur vers un fichier et 2>&1 vers la sortie standart.

Les tests (if then else), test et [

Le principe de l'instruction if est de prendre en argument une commande. Si celle-ci retourne 0, l'instruction suivant le then est exécutée. Autrement si il y a un else, l'instruction qui le suit est exécutée. Cela correspond à :

```
if Commande_test
  then
    commande_si_zero
  else
    commande_si_autre
fi
```

La commande test classique est ***test*** qui a un synonyme **[**. L'expression du test se constitue à l'aide de lettres et d'opérateurs dont :

- -x,-w,-r filename , fichier filename respectivement executable, writable, readable
- -d dirname dirname est un répertoire
- -ot, file1 -nt file2 plus ancien plus récent
- -b, -c device block ou caractère
- -z chaîne, vraie si la chaîne est vide
- ! expression, vraie si l'expression est fausse
- =,-eq, égalité, !=, -ne différent
- -lt,>,-le,>= plus petit, plus petit ou égal
- -gt, >, -ge,<= plus grand, plus grand ou égal
- -a, -o et et ou logique

L'expression se termine par] et il y a des **espaces entre chaque opérande** car ce sont des paramètres distincts d'une même commande. Il faut aussi mettre des '' autour des variables de façon à éviter une erreur de syntaxe si elles sont vides. Par exemple :

```
if [ ! -z ''$VIDE'' ]; then
  echo ''Fonctionne meme si VIDE n'existe pas''
fi
```

Un emploi typique est :

```
if [ -f ''/usr/bin/firefox'' ]; then; echo 'il y a firefox'; fi
```

L'aiguillage case :

Il correspond au synopsis suivant :

case word in [pattern [| pattern] ...) list ;;] ... esac

Par exemple :

```
case ''$VAR'' in
  UN) echo ''VAR vaut 1''
      ;;
  DEUX|TROIS) echo ''VAR vaut 2 oubien 3''
      ;;
  *) echo ''Cas non prévu''
     ;;
esac
```

UN et DEUX peuvent être des variables.

Les boucles (for, while, until)

Pour while et until le synopsis est :

```
while cmd_test; do cmds; done
until cmd_test; do cmds; done
```

Par exemple cela donne :

```
while [ -z ''$FIN'']; do
   echo ''$FIN''; FIN='1'
done
```

Pour for le synopsis est :

```
for VAR in EL1 EL2 EL3 ... ; do cmds; done
```

Voici un exemple à méditer :

```
LISTE=`echo *.c`
for I in $LISTE; do
  gcc -c -g $I
done
```

L'instruction for est aussi fort pratique en ligne de commande.

Les inclusions et les fonctions :

On peut inclure un script shell dans un autre en utilisant un point simple suivi du nom du script. Par exemple :

```
 .     /etc/sysconfig/network
```

exécute le script network dans le shell courant. Dans ce cas, les définitions ajoutées par network seront persistantes et le script courant continuera après la fin de network. Pour appeler un autre shell sans revenir il faut utiliser exec :

```
exec autre_script
```

à la sortie de autre_script le shell courant s'arrête. Pour lancer une exécution en batch il faut utiliser &.

La définition de fonction en shell se fait comme suit:

```
get_file_number()
{
instruction; instruction; ...
}
```

les paramètres de la fonction s'appellent $1 $2 ..., l'appel des fonctions se fait comme suit :

```
get_file_number titi toto tutu
```

Il est possible de constituer des librairies en utilisant l'inclusion de scripts. Le fichier .profile est inclus à chaque démarrage de shell.

Les commandes Unix utiles :

Les options de ces commandes s'obtiennent avec –-help ou man.

grep	Filtre les lignes contenant un mot
cat	Écrit un fichier dans la sortie standard
cut	Coupe des colonnes fixes ou avec séparateur cut -d'';'' -f 1
sort	Trie, en ordre numérique avec -n
uniq	Enlève les lignes en double
tr	tr ''[a-z]'' ''[A-Z]'' passe en majuscule, marche avec les expression régulières
expr	expr 1 + 3 donne 4, attention aux espaces
sed	sed -e s!avant!apres!g remplace après en avant dans l'entrée standard
tail	Coupe la fin d'un fichier, voir l'option -f
head	Coupe le début d'un fichier
date	Donne la date, voir les options +y, -y
wc	Compte les éléments d'un fichiers

C. Gouinaud/2007/ISIMA