Interpréteurs (shells)

D. Bogdaniuk – P. Portejoie

Samuel LE BERRE	**25/11/16**
Groupe A2	

Remarque préliminaire : tout en présentant quelques nouvelles notions, ce TP fait essentiellement la synthèse de nombreux concepts déjà étudiés lors des séances précédente, en les généralisant. Prenez le temps de bien lire ce document et faites appel à votre mémoire…

Le shell est l'interface entre l'utilisateur et le système d'exploitation. C'est un interpréteur de commandes qui possède un mécanisme d'expansion de métacaractères et de substitution de variables. C'est aussi un interpréteur de fichiers de commandes (shell-scripts).

Il existe plusieurs interpréteurs de commandes : le Bourne-shell, le C-shell, le Korn-shell et d'autres encore…

- le Bourne-shell (sh) qui est un standard sous Unix est le plus ancien. Il sert actuellement essentiellement à l'écriture de scripts
- le C-shell (csh) possède une syntaxe proche du C, peut cohabiter avec sh et se montre plus convivial au terminal
- le Korn-shell (ksh) possède une compatibilité ascendante avec sh et reprend la convivialité de csh

NB : bash est une variante du Bourne Shell, ou plutôt du Korn-Shell adaptée à Linux.

Les caractères spéciaux

`\`	banalise le caractère suivant
`"…"`	banalise tous les caractères sauf `\` , `$` et `` ` ``
`'…'`	banalise tous les caractères
`` `…` ``	autorise la substitution de commande

Le login

sh	ksh	csh	

`.bash_profile`
`.bash_login`
`.profile`
`.bash_logout`
	`.profile`	`.login`
		
		`.logout`
			fichiers exécutés au moment de la connexion

`.bashrc`	`.kshrc`	`.cshrc` `.tcshrc`	fichiers exécutés à chaque nouveau shell

Redirection des entrées/sorties

sh	ksh	csh	
`<` `0<`		`<`	redirection de l'entrée standard à partir d'un fichier
`<<`*`marqueur`*			redirection de l'entrée standard avec un marqueur(1)
`>` `1>`		`>`	redirection de la sortie standard
`>>` `1>>`		`>>`	redirection de la sortie standard en concaténation
`2>`			redirection de la sortie erreur
`2>&1`		`>&` `>!`	redirection des sorties standard et erreur redirection des sorties standard et erreur avec inhibition des contrôles

(1) : envoie sur l'entrée standard tout ce qui est saisi depuis la ligne suivant la commande jusqu'à la ligne contenant *marqueur* (n'existe pas en csh)

Exemple :

```
> cat > Fichier <<FINI
  blablabla
  et encore blablabla
  FINI
> more Fichier
  blablabla
  et encore blablabla
>
```

Variables

sh et ksh, contrairement à csh, ne distinguent pas fonctionnellement de variables locales et globales. Les variables définies restent locales tant qu'elles ne sont pas exportées par la commande `export`.

sh	ksh	csh	
`HOME`		`HOME` `home`	le répertoire d'accueil
`PATH`		`PATH` `path`	chemins de recherche pour l'exécution des commandes
`USER`		`USER` `user`	nom de login
`GROUP`		`GROUP` `group`	nom de groupe
`HOSTNAME`		`HOSTNAME`	nom de machine
`CDPATH`		`cdpath`	chemin de recherche pour la commande `cd`
`MAIL`		`MAIL`	chemin indiquant le répertoire du courrier
`PS1`		`prompt`	prompt principal
`PS2`		`prompt2` `prompt3`	prompts secondaires
`SHELL`		`SHELL` `shell`	shell de connexion
`HISTSIZE`		`HISTSIZE` `history`	nombre de commandes mémorisées dans l'historique
`HISTFILESIZE`		`HISTFILESIZE` `savehist`	nombre de commandes mémorisées dans l'historique après une déconnexion
`TERM`		`TERM` `term`	le terminal de connexion
`DISPLAY`		`DISPLAY`	définit le serveur X
`PWD`		`PWD` `cwd`	répertoire courant
`IGNOREEOF`		`ignoreeof`	si elle n'est pas initialisée, la déconnexion par `^D` est impossible
`$?` `$status`		`$?` `$status`	statut de la dernière commande exécutée (0 si elle s'est bien exécutée)
`IFS`			séparateur interne de champ
		`autologout`	le temps de déconnexion automatique d'un shell (en minutes)

Variables spéciales

sh	ksh	csh	
`$$`			numéro du shell parent
`read variable`		`set variable=$<`	lecture au clavier

Définition de variables

sh	ksh	csh	
`variable=valeur`		`set variable=valeur` `setenv variable valeur`	affectation
`$variable` `${variable}`			accès au contenu de la variable
`[ "$variable" ]`		`$?variable`	la variable est-elle initialisée ?

Arguments de la ligne de commande

sh	ksh	csh	
`$0`		`$0` `$argv[0]`	nom de la commande
`$n`		`$n` `$argv[n]`	n^{ième} paramètre
`$#`		`$#` `$#argv`	nombre de paramètres

`$*`
 	
`$@`
		`$*`
		
		`$argv`
			ensemble des paramètres
			
			ensemble des paramètres sous forme de chaînes de caractères

Exécution de scripts

sh	ksh	csh	
`./script`			le script est exécuté par un shell sh à moins que la première ligne ne débute par `#!` et ne désigne le shell dans lequel l'exécution doit se faire
`sh` *`script`*	`ksh` *`script`*	`csh` *`script`*	le script est exécuté par le shell indiqué
`.` *`script`*	`.` *`./script`*	`source` *`script`*	le script est exécuté par le shell courant (en ksh le script doit se trouver dans un des répertoires du PATH)
`sh -n`	`ksh -n`	`csh -n`	l'option `-n` sert à vérifier la syntaxe
`sh -v`	`ksh -v`	`csh -v`	l'option `-v` sert à afficher chaque ligne avant son exécution (variable `verbose`)
`sh -x`	`ksh -x`	`csh -x`	l'option `-x` sert à afficher chaque ligne avant son exécution, mais après interprétation des variables et des métacaractères (variable `echo`)

Alias

sh	ksh	csh	
`alias` *`chaîne='cmd'`*		`alias` *`chaîne 'cmd'`*	définition d'un alias
`unalias` *`chaîne`*			suppression d'un alias

Commandes d'édition de la ligne de commande

sh	ksh	csh	
`set -o vi`		`bindkey -v`	autorise l'utilisation des commandes vi dans la ligne de commande (touches K, J, H, L pour haut, bas, gauche, droite)
`set -o vi-tabcomplete`			autorise le remplissage automatique dans la ligne de commande (touche tab)

Enfin, de nouveaux interpréteurs plus spécialisés ont fait leur apparition ces dernières années : Perl, Tcl/Tk et Python.

Perl est un langage de programmation interprété semblable au langage C mais comportant de nombreuses fonctionnalités Unix telles sed, awk ou tr… Il est considéré comme un bon choix pour l'écriture des CGI (Common Gateway Interface) puisqu'il permet la manipulation de texte avec une grande facilité, de même que celle de fichiers binaires. En général, Perl est facile à apprendre et ses programmes sont plus rapides à écrire que ceux de langages plus structurés et compilés comme C ou C++.

Tcl est un langage de script interprété. Il est généralement associé à TK (Tool Kit) pour la création d'interfaces graphiques. TclBlend est une version de Tcl qui peut accéder à certaines fonctions java.

Python est un langage de programmation objet interprété semblable au langage C. Il est particulièrement utilisé pour l'écriture de scripts et permet l'écriture de puissants programmes en rivalise avec le langage Java dans sa version Jpython.

TP

NB : vous trouverez les scripts nécessaires dans le forum prof habituel. Si, malgré ce qui vous a été conseillé vous ne l'avez pas fait avant de venir en TP, **lisez attentivement** la page 1 (vous aurez également besoin de consulter par la suite les différents shells)

Exercice 1

1/ Récupérez les fichiers `version` et `version1` sur le forum habituel (`version1` ne diffère de `version` que par sa première ligne). Si nécessaire rendez les fichiers exécutables

2/ Vérifiez si `sh` lance directement un shell posix (ie répondant au standard IEEE1003) ou s'il s'agit seulement d'un lien symbolique vers un shell *bash.* Pour ce faire, lancez la première commande donnée ci-dessous afin d'identifier le(s) chemin(s) conduisant à `sh`, puis lancez la deuxième en y adaptant le premier chemin trouvé afin de répondre à la question

3/ `>whereis sh`
`sh: ...`
`>ls -l /?premier_chemin_trouvé?/sh`

```
[e1604902@ens-iutva-0385 TP9]$ sh

sh-4.3$ whereis sh

sh: /usr/bin/sh /usr/share/man/man1p/sh.1p.gz /usr/share/man/man1/sh.1.gz

sh-4.3$ ls -l /usr/share/man/man1/sh.1.gz

lrwxrwxrwx. 1 root root 9 30 sept. 13:09 /usr/share/man/man1/sh.1.gz -> bash.1.gz
```

4/ En exécutant les instructions suivantes indiquez à chaque fois (en justifiant) quel est le nom de l'interpréteur utilisé (n'oubliez pas que *sh* → *bash*) :

	Interpréteur utilisé (seulement son nom, pas sa version)
`./version`	*sh*
`./version1`	*csh*
`csh`	*Lancement d'un nouveau shell*
`./version`	*sh*
`source version`	csh
`bash version`	sh
`bash version1`	sh
`ksh version`	ksh
`ksh version1`	Ksh
`csh version`	csh
`bash`	*Lancement d'un nouveau shell*
`./version`	ksh
`./version1`	*ksh*
`. version`	sh
`. version1`	sh
`ksh`	*Lancement d'un nouveau shell*
`. ./version`	ksh
`. ./version1`	ksh

5/ Comparez le rôle du point (".") pour `". version"` de *bash* et `". ./version"` de *ksh* avec `"source version"` de *csh*

« ./ » le script est exécuté par un shell sh à moins que la première ligne ne débute par `#!` et ne désigne le shell dans lequel l'exécution doit se faire

« . » et « source » Le script est exécuter par le shell courant

6/ Que faut-il modifier dans le script `version` pour que son exécution par `./version` lance par défaut un interpréteur C-shell ?

Il faut rajouter une ligne au debut du script avec un csh dessus pour que le script s'éxecute en temps que csh

Exercice 2

Analysez et précisez le rôle du script `ques` donné ci-après et disponible dans le forum habituel (essentiellement pour en déduire les *arguments* nécessaires à son exécution), puis testez-le par `./ques arguments`

```
#!/bin/csh
# ROLE
# ?
# ?
echo "Debut du script $0"
mkdir $1
mkdir $1/bin $1/src
cp $0 $1/bin
cd $1/bin
chmod $2 $0
ls -al ../*
cd ../src
echo "fin du script $0"
```

Remarque : l'interpréteur découpe la ligne de commande avec l'espace comme séparateur de champ ; ainsi $0 est le premier champ, donc le nom de la commande, $1 est le premier paramètre, etc...

Il y a 2 arguments
Ce script affiche « Debut du script argument 1 », ensuite il crée un répertoire, et dans ce repertoire crée 2 autres répertoires src et bin,il copie ques dans le dossier crée a l'intérieur de bin, il nous envoie ensuite dans bin du répertoire créé. Il change ensuite les droits de ques par le parametre 2. il liste tous avec les droits des fichiers.
Il affiche a la fin « fin du script argument 1 »
Les 2 arguments sont donc le nom d'un dossier et les droits qu'on va donner a ques

Exercice 3

Ecrivez un script en C-shell qui rend l'adresse électronique de la personne dont l'identifiant (login) est récupéré par saisie au clavier

#!/bin/csh
 # Trouvé l'adresse mail d'un utilisateur
 # ?

```
echo "adresse mail de l'utilisateur $1"
ypcat passwd | egrep $1 | cut -d ',' -f2 | cut -d ' ' -f2
```

```
[e1604902@ens-iutva-0385 TP9]$ ./script e1604902
adresse mail de l'utilisateur e1604902
le-berre.e1604902@etud.univ-ubs.fr
```

Exercice 4

1/ *Vous trouverez* `compilateur` *et* `progc.c` *dans le forum habituel. Analysez et complétez le script* `compilateur` *(cf pages 3 et 4). Puis, dans un premier temps, testez-le sans paramètre, et enfin avec le paramètre* `prog` . Lorsque nécessaire, vous serez amené à corriger `prog.c`

```
#!/bin/csh
if ( )  then
  echo "Usage : compilateur <fichier>"
  echo " "
  echo "Compile le programme C passe en paramètre"
  echo "et edite le fichier en cas d'erreur."
  exit(0)
endif

while (1)
  cc $1.c -o $1 >& .erreur
  if (  $status == 0  ) then
    echo "Le programme $1.c ne comporte pas d'erreur "
    echo "Execution de $1 :"
    echo ""
    ./$1
    exit(0)
 else

  echo "Le programme $1.c contient des erreurs"
  echo "voulez-vous l'editer ?"
  set reponse=$<
  if ( reponse != o ) then
    echo "fin de traitement";
    exit(0)
  endif
  emacs .erreur $1.c
end
```

```
[e1604902@ens-iutva-0385 TP9]$ ./compilateur prog
Le programme prog.c contient des erreurs
voulez-vous l'editer ?
o
fin de traitement
[e1604902@ens-iutva-0385 TP9]$ ./compilateur
Le programme .c contient des erreurs
voulez-vous l'editer ?
o
fin de traitement
```

2/

3/ Modifiez le programme pour que le nom du compilateur choisi (`cc` dans l'exemple) soit aussi passé en paramètre lors de l'appel à `compilateur`

```
#!/bin/csh
if ( )  then
  echo "Usage : compilateur <fichier>"
  echo " "
  echo "Compile le programme C passe en paramètre"
  echo "et edite le fichier en cas d'erreur."
```

```
    exit(0)
  endif

  while (1)
    $2 $1.c -o $1 >& .erreur
    if (  $status == 0  ) then
      echo "Le programme $1.c ne comporte pas d'erreur "
      echo "Execution de $1 :"
      echo ""
      ./$1
      exit(0)
   else

    echo "Le programme $1.c contient des erreurs"
    echo "voulez-vous l'editer ?"
    set reponse=$<
    if ( reponse != o ) then
      echo "fin de traitement";
      exit(0)
    endif
    emacs .erreur $1.c
end
```

Exercice 5

Dans cet exercice vous allez utiliser l'éditeur en ligne `sed` pourc constituer un tableau HTML. L'annexe de ce document contient un exemple de fichier HTML et un exemple d'utilisation de `sed`. La solution vous est quasiment fournie en annexe, mais vous devez chercher à en comprendre la mise en œuvre.

Ecrivez un script en bash qui produit un fichier HTML contenant les noms et prénoms des étudiants dont le groupe UNIX est passé en paramètre. Le nom du fichier HTML est également un paramètre du script.

ANNEXE

```
#!/bin/bash
echo "<HTML>
        <HEAD>
                <TITLE> Exercice 5 </TITLE>
        </HEAD>
        <BODY>
                 <H1>Liste des utilisateurs</H1>
                <H3>`date "+%A %d %B %Y"`</H3>
                <TABLE border=1>
                <TR><TH><B>Nom</B></TH><TH><B>Pr&eacute;nom</B></TH></TR>" > $2.html
                ypcat passwd  | ... | … | … | sed "s/.../.../" >> $2.html
echo "          </TABLE>
        </BODY>
        </HTML>" >> $2.html
```

Commande sed avec son expression régulière pour la constitution du tableau des prénoms et noms :

```
sed "s/\(.*\) \(.*\)/<TR><TD>\2<\/TD><TD>\1<\/TD><\/TR>/"
```

Explications du sed :

`s/chaîne1/chaîne2/` : arguments de sed pour la substitution de chaîne1 par chaîne2 ; les chaînes peuvent être littérales ou être des expressions régulières

`\(.*\) \(.*\)` : expression régulière qui identifie la ligne à 2 champs séparés par un espace, ces champs sont ensuite reconnus par `\1` et `\2`

`<TR><TD>\2` : expression régulière qui ouvre la ligne et construit la colonne contenant le nom (champ identifié 2)

`<\/TD><TD>\1` : expression régulière qui construit la colonne contenant le prénom (champ identifié 1)

`<\/TD><\/TR>` : chaîne qui ferme la ligne