KORN SHELL

Introduction au Korn Shell
Développement en Korn Shell

LIVRE DE COURS

1 - AVANT-PROPOS

1.1 - Public visé

Les participants à ce cours doivent disposer des notions de base UNIX. Le cours Utilisation d'Unix est un pré-requis suffisant à ce cours.
L'ensemble des points vus dans ce cours font référence à des commandes de base d'Unix, qu'il n'es matériellement pas possible d'intégrer à ce cours. La bonne connaissance d'un éditeur de textes et des mécanismes liés aux expressions régulières sont nécessaires.

1.2 - Contenu du cours

Ce cours est très pratique et intègre beaucoup d'exercices. Il présente la sémantique du Korn Shell, dans sa globalité. Il donne une réelle autonomie pour développer en Korn Shell.
A noter pour les utilisateurs de Linux :

- il existe un portage de Korn Shell disponible sous Linux, du nom de **pdksh** (« public domain Korn Shell »).
- Le shell par défaut de Linux est **bash** (« Bourne Again Shell »). Il intègre les notions les plus intéressantes de **sh** (« Bourne Shell ») et de **ksh**. La connaissance de ce cours intéressera donc les personnes ayant à utiliser **bash**.

Ce cours permettra aux participants d'écrire des shell-scripts Unix (équivalents des JCL) robustes et compatibles avec les standards de l'industrie. Un accent tout particulier est placé sur la portabilité : les mécanismes étudiés appartiennent donc au dénominateur commun Bourne-Shell / Korn-Shell.

1.3 - Typographie utilisée

Dans l'ensemble de ce document, quelques styles typographiques sont utilisés de façon à rendre plus aisée la lecture.

Les noms propres apparaissent en *italique*.

Vous verrez parfois des noms de **commandes** ou encore des **noms de variables** insérées dans le texte de ce document.

```
Ce style est utilisé pour les exemples de commandes passées au système.
```

1.4 - TABLE DES MATIERES

2 - Introduction au shell

Le shell est le programme qui est en cours d'exécution quand vous êtes connecté à une machine Unix ou Linux par un terminal ou une session telnet. C'est lui qui vous permet d'exécuter des commandes, et par là même d'utiliser le système, de créer d'autres programmes, et pourquoi pas un jour un que vous utiliserez vous-même comme shell. Le shell a pour rôle d'interpréter les commandes que vous lui passez.

Plusieurs shells différents existent. Parmi eux, le shell Korn Shell est **ksh** que nous allons étudier. Ce shell est un des plus performants, différent de **sh** (Bourne Shell), le shell d'origine.
Il existe aussi :
- **bash** (Bourne Again Shell), le shell de Linux, qu'il est possible d'installer sur toute machine Unix disposant d'un compilateur C.
- **csh** (C Shell), autre shell utilisé par les chercheurs.

2.1 - Variables d'environnement

Le shell positionne à la connexion plusieurs variables d'environnement. Nous avons vu dans le cours « Utiliser Unix » que vous pouvez vous aussi en positionner d'autres, soit globales, c'est à dire connues de tous les processus fils que vous lancerez, soit locales, c'est à dire connues uniquement par le shell en cours.

2.1.1 - Positionner la valeur d'une variable

2.1.1.1 - Les commandes set et export

On peut positionner la valeur d'une variable en la déclarant simplement, et par la commande **export**.
Syntaxe :
```
VARIABLE=« Texte »
```
La variable **VARIABLE** est alors connue dans le shell courant mais n'est pas connue dans les scripts que l'on appellerait depuis le shell courant.

Pour qu'elle le soit, il faut l'exporter :
Syntaxe :
```
export VARIABLE
```

Il est possible d'exporter une variable au moment où on lui affecte une valeur, grâce à la commande **export**.
Syntaxe :
```
export VARIABLE=« Texte »
```

La commande **set** permet de lister l'ensemble des variables déclarées.
Syntaxe :
```
set
```

2.1.1.2 - La commande env

La commande **env** permet de lister l'ensemble des variables exportées. Elle permet aussi d'appeler une commande en positionnant une variable de façon à ce que cette dernière soit connue lors de l'appel d'une commande.

Syntaxe :
env : liste les variables d'environnement
env VARIABLE=valeur test.sh : la variable **VARIABLE** est connue dans l'exécution de **test.sh**.

Cette dernière syntaxe permet de ne positionner une variable d'environnement que pour la durée d'exécution d'un programme, mais pas dans l'environnement appelant. Ceci est très utile pour tester des programmes en utilisant successivement des connexions vers des bases de données différentes, par exemple.

2.1.2 - Afficher la valeur d'une variable : echo

On peut afficher la valeur d'une variable ainsi que toute chaîne de caractères à l'aide de la commande **echo**.

Syntaxe :
echo [options] [« texte »] [$VARIABLE]

Pour connaître le contenu de la variable **VARIABLE**, il faut la faire précéder du signe **$**.

Exemple :
```
[fmi@serv-U fmi]$ echo "l'utilisateur conecte est : "$USER
l'utilisateur conecte est : fmi
```

La commande **echo** supporte plusieurs options, notamment pour le formatage du texte.

Option	Signification
\a	Affiche un caractère d'alerte
\b	Retour arrière
\c	Ne passe pas à la ligne
\f	Saute de page
\n	Passe à la ligne
\r	Retour charriot
\t	Tabulation
\v	Tabulation verticale
****	Backslash
\0num	Affiche le caractère de notation octale num.

2.1.3 - Vidage d'une variable : unset

On peut vider la valeur d'une variable à l'aide de la commande **unset**.
Même si celle-ci était exportée, elle est vidée et n'est plus connue.

Syntaxe :
unset VARIABLE

3 - Rappels de quelques notions UNIX

3.1 - Processus séquentiels

Il est possible d'enchaîner l'exécution de plusieurs processus en les séparant par le signe « ; »

Exemple :

```
proc1
proc2
proc3
```

est équivlent à

```
proc1 ; proc2 ; proc3
```

3.2 - Processus en parallèles

Pour lancer des processus en tâche de fond, il faut terminer la ligne de commande de chaque processus par le signe « & »

Exemple :

```
proc1 &
proc2 &
proc3 &
```

Il est possible, notamment dans des shell scripts, le lancer plusieurs processus en parallèle sur la m^me ligne de commande. Dans ce cas, la syntaxe à utiliser est la suivante :

```
proc1 & proc2 & proc3 &
```

3.3 - Re-direction des entrées-sorties

Les entrées et les sorties des processus peuvent être redirigées :

- `<` L'entrée standard est lue à partir d'un fichier
- `>` La sortie standard est redirigée dans un fichier (RAZ du fichier)
- `>>` La sortie standard est redirigée dans un fichier (concaténation du fichier)
- `2>` Les erreurs sont redirigées dans un fichier
- `2>&1` Les erreurs sont redirigées dans le même fichier que la sortie standard

3.3.1 - Délimitation de l'entrée

Lors de la re-direction de l'entrée standard, il est possible de marquer la fin de la re-direction au moyen d'un délimiteur. Dans un shell script, par exemple, la ligne qui suit le délimiteur est considérée comme une commande suivante, et plus comme faisant partie de ce qu'il faut intégrer dans l'entrée standard.

Exemple :

```
Commande << [-] Marqueur
ligne1
ligne2
ligne3
...
Marqueur
```

Si le signe « - » est ajouté avant le délimiteur, les tabulations de début de ligne sont supprimées du document « lu » en entrée standard. Cela permet de rédiger du code incluant des marqueurs sans se priver de l'indentation, facilitant grandement la lecture du code.

3.3.2 - Re-direction de l'entrée et de la sortie

Plusieurs re-directions peuvent être utilisées dans la même ligne de commande.

Exemple :

```
cat > fichier << EOF
abc
def
EOF
```

3.4 - Les pipes

Les pipes servent à envoyer la sortie standard d'un processus vers l'entrée standard d'un second processus.

Exemple :

```
proc1  |  proc2
```

équivaut à :

```
proc1 > fich
proc2 < fich
```

Note : Les pipes ne créent pas de fichier temporaire. Pour que les deux exemples soient réellement similaires, il faudrait donc ajouter au second un effacement du fichier `fich`.

3.5 - Génération des noms de fichiers

Des caractères spéciaux permettent de « masquer » plusieurs noms de fichiers (du répertoire courant).

***** n'importe quelle chaîne de caractères

? n'importe quel caractère

[... **]** n'importe quel caractère décrit entre les crochets

3.6 - Exécution d'un Shell script

Pour pouvoir exécuter un shell script que vous avez écrit, celui-ci doit disposer de la permission le rendant exécutable et être dans un répertoire définit dans votre PATH si vous souhaitez simple l'invoquer par son nom.

Par contre, s'il ne dispose pas de la permission d'exécution, vous pouvez quand même l'exécuter, mais de la manière suivante :
ksh nom_fichier

Le plus simple est peut-être de faire un **chmod u+x nom_fichier** lors du premier test de votre script.

3.6.1 - En-tête d'un shell script

Attention, si celui-ci a la permission exécutable, il sera exécuté avec votre shell par défaut. Dans le cas où ce ne serait pas **KSH**, il faut forcer le passage par **KSH**. Pour forcer l'exécution du fichier en Korn Shell, le fichier doit commencer par une ligne contenant :
#!/bin/ksh.

3.6.2 - Options de ksh

Il existe quelques options utiles du shell ksh pour l'invocation de vos shell scripts.

Exemple	Signification
ksh **-n** nom_fichier	interprète les commandes sans les exécuter
ksh **-v** nom_fichier	imprime les lignes comme elles sont lues
ksh **-x** nom_fichier	imprime les lignes comme elles sont interprétées

4 - Personnalisation de la session

Pour personnaliser votre session, vous pouvez juste après votre connexion positionner quelques variables d'environnement, lancer des traitements au démarrage, etc.

Mais faire cela à chaque session est inenvisageable. Heureusement, il existe des fichiers permettant de personnaliser tous les shells.

4.1 - Les fichiers d'environnement

Les vôtres sont sur votre `HOME directory`, qui est celui sur lequel vous vous trouvez normalement au démarrage de votre session, et dont le chemin est indiqué par la variable `HOME`.

Sur ce répertoire, il y a un fichier nommé `.login` et un autre `.profile`. Dans ces fichiers, il faut positionner les commandes que vous souhaitez exécuter :
- à chaque nouvelle connexion (`.login`)
- à chaque lancement d'un shell (`.profile`).

Attention, en fonction du shell que vous utilisez, le fichier `.profile` peut porter un autre nom.

4.1.1 - Rôle des fichiers d'environnement

Ces fichiers peuvent contenir, par exemple, la personnalisation du `PATH`, pour prendre en compte des exécutables que vous auriez stocké dans un de vos répertoires, ils peuvent aussi positionner la variable `PS1`, qui définit ce qui est affiché à l'invite de commandes, ou réaliser toute autre action que vous souhaitez.

4.1.2 - Fichier d'environnement personnel ou de groupe

Si on positionne la variable `ENV`, alors le fichier spécifié par son contenu (en général `$HOME/.kshrc` pour Korn Shell) est exécuté à chaque ouverture de session shell (on s'en sert pour définir les alias)

4.2 - Quelques personnalisations fréquentes

4.2.1 - Le prompt

Le **prompt** (*l'invite*) par défaut est le signe « **$** ». En positionnant dans votre **.profile** une ligne qui change la valeur de la variable **PS1**, on peut obtenir le prompt qui apparaît dans tout ce document dans les exemples.

Exemples de prompt :

```
$ export PS1='$PWD'"> "
/export/home/usr/fmi> export PS1="$LOGNAME:"'$PWD'"> "
fmi:/export/home/usr/fmi> export PS1="[$LOGNAME][" '$PWD'"] "
[fmi][/export/home/usr/fmi] export PS1="`uname -n`:"
Netfinity: export PS1="[$LOGNAME]["`uname -n`:'$PWD'"] "
[fmi][Netfinity:/export/home/usr/fmi] export PS1="[$LOGNAME]["`uname -n`:'${PWD##*/}'"] "
[fmi][Netfinity:fmi]
```

4.2.2 - La touche Backspace

Il arrive souvent que la touche **backspace** ne soit pas paramétrée correctement lorsque le'on se connecte sur une machine. Cela est du au paramétrage du terminal, et plus particulièrement au code identifiant le caractère de correction de la frappe.

Si lors de l'appui sur la touche **backspace** s'affichent des caractères « ^H », au lieu d'effacer la frappe précédente, il faut paramétrer le terminal pour que celui-ci interprète le caractère « ^H » comme étant le caractère d'effacement à gauche.

Pour cela, on utilise la commande **stty**. Le mot clef pour effacement à gauche est « **erase** ».
D'où la commande :

```
stty erase SUIVI D'UN APPUI SUR BACKSPACE
```

De la même manière, il est fréquent que le caractère effacé reste apparent à l'écran. Si cela vous gêne, vous pouvez positionner l'indicateur « **echoe** » par la commande **stty**. Là, il n'y a pas d'argument supplémentaire.

```
stty echoe
```

4.2.3 - L'éditeur de rappel de commandes

Il est possible de paramétrer le comportement du rappel de commandes.

set –o vi ➔ Pour vi
set –o emacs ➔ Pour emacs (ce dernier n'a pas besoin d'être installé)

Lorsque ce positionnement est fait, le Korn Shell adopte le fonctionnement de l'éditeur choisi lorsque vous rentrez en rappel de commande. Korn Shell simule en fait le comportement de ces éditeurs de textes. C'est la raison pour laquelle il n'y a pas besoin qu'ils soient installés pour que cela fonctionne.

5 - Les variables de Korn Shell

Nous avons vu que nous pouvons positionner des variables d'environnement pour nos propres shells. Nous avons aussi vu que l'on pouvait utiliser des variables positionnées par notre shell, telles que `LOGNAME` (nom d'utilisateur), `PWD` (répertoire courant), etc.

Il existe dans les différents shells fonctionnant sous Unix d'autres variables, qui doivent plus être perçues comme des variables systèmes du shell que comme des variables d'environnement.

5.1 - Variables d'environnement de Korn Shell

L'ensemble de ces paramètres peut être lu par `echo` en n'oubliant pas que ce sont des variables et qu'il faut faire précéder leur nom par le signe `$` pour les accéder.

Variable	Signification
`LINENO`	Numéro de la ligne à laquelle la commande est exécutée (très utile pour débugger)
`ERRNO`	Code de la dernière erreur
`HOME`	Le home directory (répertoire de par défaut)
`PATH`	Chemin de recherche pou l'exécution des commandes
`CDPATH`	Chemin de recherche pour la commande cd
`MAIL`	Chemin indiquant le répertoire du courrier
`PS1`	Prompt principal
`PS2`	Prompt de niveau 2
`PS3`	Prompt de niveau 3
`IFS`	Internal field separator
`SHELL`	Indique le shell de login

5.2 - Les variables en Korn shell

5.2.1 - Paramètres positionnels en le Korn Shell

Paramètre	Signification
#	Nombre de paramètres passés sur la ligne de commande
0—n	Paramètre numéro 0 à n. 0 = programme appelé, 1 = premier paramètre, etc…
*	Liste de tous les paramètres (arguments) : "$*" = "$1 $2 $3 "
@	Liste de tous les paramètres (arguments) : "$@" = "$1" "$2" "$3"
?	Code retour de la dernière commande
$	Numéro de processus (PID) du shell courant.
!	Numéro de processus (PID) de la dernière commande lancée en tache de fond.
_	Nom de complet de la commande lancée.

5.2.2 - La commande shift

Lors de la lecture des paramètres de la ligne de commande par un shell-script, il est possible de décaler vers la gauche les paramètres lus. La commande réalisant ceci est **shift**.

Exemple :
Dans le shell **params.ksh**, on lit le premier paramètre passé et on l'affiche
Puis on réalise le shift et on recommence l'opération.
Pour l'exemple le programme fait trois fois l'opération.

Shell script **params.ksh**

```
# !/bin/ksh
echo $1
shift
echo $1
shift
echo $1
shift
```

Testons l'appel du shell script :

```
$ ./params.ksh p1 p2 p3 p4 p5
p1
p2
p3
```

L'intérêt de la commande shift n'est pas évident dans cet exemple, on aurait pu se servir de $*, mais tous les paramètres auraient été affichés sur la même ligne, et les autres paramètres donnés lors de l'appel de notre shell script auraient été affichés.

5.2.3 - La délimitation de variables

Korn Shell sait reconnaître des formes dans les chaînes de caractères et intervenir sur leur apparition lors de l'expression de ces variables. Nous avons utilisé pour le dernier prompt (voir « Personnalisation de l'environnement ») cette fonctionnalité en lui demandant de supprimer tout sauf ce qui suit le dernier « **/** » de notre variable **PWD**.

Pour pouvoir utiliser cette fonctionnalité, il est nécessaire de délimiter les variables par des accolades (« **{** » et « **}** »). Cela se fait de la manière suivante :
Au lieu de faire :
```
echo $VARIABLE
```
il faut faire :
```
echo ${VARIABLE}
```

De plus, si vous souhaitez afficher la variable **ABC** suivie du caractère **D**, vous devez absolument utiliser cela si vous ne souhaitez pas mettre le **D** entre guillemets.

Exemple :
```
ABC=toto
```
`echo $ABCD` ➔ Ne fonctionnera pas, il n'y a pas de variable ABCD de définie
`echo ${ABC}D` ➔ Affichera **TOTOD**

5.2.4 - Les « modifiers »

Les modificateurs (« modifiers »), sont des opérateurs permettant de jouer sur l'interprétation des variables. En voici la liste

Dans le tableau ci-dessous, **VAR** est une variable à laquelle on applique l'opérateur (avec la valeur **ABCD**).

Modifier	Signification
`${VAR:-ABCD}`	Si **VAR** est positionnée et non vide, affiche la valeur de **VAR**, sinon la chaine **ABCD**
`${VAR:+ABCD}`	Si **VAR** est positionnée et non vide, alors **ABCD** est affiché, sinon rien n'est affiché
`${VAR:=ABCD}`	Si **VAR** est positionnée et non vide, **VAR** est affichée et garde sa valeur d'origine. Sinon, elle prend la valeur **ABCD** et est affichée.
`${VAR:?}`	Si **VAR** est positionnée et non vide, alors elle est affichée. Sinon s'affiche un message d'erreur « parameter null or not set ». Une erreur est générée et le shell-script s'arrête.
`${VAR:?message}` ou `${VAR:?$MSG}`	Idem, mais on peut faire un afficher un autre message en le mettant après le signe **?** Dans la seconde syntaxe, c'est la valeur de la variable MSG qui serait affichée.

Le signe « **:** » donne plus de lisibilité, mais il peut être omis.

Attention :
Sans le signe « : », la condition n'est plus tout à fait la même.
Au lieu de « *est positionnée et non vide* », elle devient « *est positionnée* ».

5.2.5 - Les patterns de substitution

Les patterns permettent aussi de jouer sur l'apparition des variables, mais sont à considérer comme des manières d'en filtrer le contenu.

Pattern	Signification
`${#} ou ${#}` ou `${#*} ou ${#@}`	Nombre de paramètres positionnels passés au shell script
`${#VAR}`	Longueur de la chaîne `VAR`.
`${#VAR[*]}` ou `${#VAR[@]}`	Nombre d'éléments dans le tableau `VAR`.
`${VAR#*ABCD}` et `${VAR##*ABCD}`	Si `VAR` contient `ABCD` (`*ABCD`), alors tout jusqu'à `ABCD` est retiré de l'affichage. `#` : la plus petite correspondance est retirée. `##`, la plus grande correspondance est retirée.
`${VAR%ABCD*}` et `${VAR%%ABCD*}`	Idem, mais en partant de la droite.

Note :
Dans les deux derniers patterns exposés (`#` et `%`), le signe `*` précédant ou suivant la chaîne `ABCD` n'est pas obligatoire, mais sert à repérer n'importe quelle chaîne de caractère. Le signe `?` aurait pu être utilisé pour repérer n'importe quel caractère. L'utilisation de ces patterns a beaucoup moins d'intérêt si l'on utilise pas de caractères génériques de type `*` ou `?`.

5.3 - Banalisation de caractères

Certains caractères ne doivent pas être interprétés car ils serviront par exemple dans les variables où ils figurent, ou encore ils doivent tout simplement être affichés être affichés.

Exemple : afficher un `$`, et une `*` avant le la chaîne de caractères « `ABCD` ».
L'exemple suivant ne réussira pas…
```
echo $*ABCD
```
Celui-ci non plus, mais il affichera néanmoins l'`*`, précédée du numéro de processus.
```
echo $$*ABCD
```
Alors que l'exemple ci-dessous n'affiche plus que le numéro de processus:
```
echo $$$*ABCD
```

Nous pourrions continuer longtemps, ou imaginer de passer par une variable contenant le signe `$` et le signe `*`. Ceci ne serait pas très élégant d'un point de vue programmation, et nécessiterait quand même d'arriver à mettre notre `$` et notre `*` dans la variable en question, ce qui n'est pas gagné d'avance.

C'est pour cela qu'il existe des méthodes permettant de banaliser certains caractères.

Signe	Signification
`\`	banalise le caractère suivant : `echo \$\*` affiche un `$` et une `*`
`" ... "`	banalise tous les caractères sauf `\`, `$` et `` ` `` : `echo "$*"` n'affiche plus le `$` et l'`*`
`' ... '`	banalise tous les caractères : `echo '$*'` affiche un `$` et une `*`
`` ` ... ` ``	substitution de commande : `` echo `ls` `` affiche le résultat de la commande `ls`.

6 - Les tests Korn shell

Les ordres de programmation structurée du Korn Shell évaluent des conditions. Ces conditions sont le code retour d'une commande réalisant le test que l'on exprime.

Afficher « Bonjour » si la variable A est égale à la variable B ne se fait pas comme ci-dessous
`If a=b ;then… echo Bonjour; fi`

En effet, ceci positionne à la valeur "**b**" la variable **a**, ce qui réussit à chaque fois, et donc la commande émet un code retour signifiant qu'elle a réussi. La signification de la ligne précédente est :
`a=b ; echo Bonjour… (ou a=b & echo Bonjour)`

La condition toujours vraie, ce qui suit le mot clé **then** sera donc toujours exécuté, (et au passage, on a écrasé l'ancienne valeur de a).

Voyons donc avant toute chose la manière dont sont faits les tests en Shell.

6.1 - Les tests

Un test est une commande qui peut être directement exécutée sur la ligne de commande. L'affichage de la variable **$?** juste après son exécution indiquera l'issue du test :
- **0** : le test est vrai
- **1** : le test n'est pas vrai

6.1.1 - Syntaxe de la commande test

La commande **test** sert à tester des expressions. On appelle donc **condition** le **test d'une expression**.

Exemple s:
```
test "a" = "a"
echo $?

test "a" = "b"
echo $?

test 1 -eq 0
echo $?

test 1 -eq1
echo $?
```

Nous remarquons que :
- le test d'égalité d'expressions de type chaîne de caractères ne se fait pas de la même manière que le test d'égalité d'expressions entières.
- Test retourne :
  - 0 quand l'égalité est vérifiée (quand le test est vrai)
  - 1 quand l'égalité n'est pas vraie (quand le test n'est pas vrai)

La commande **test** peut être utilisée comme nous venons de le voir dans les shell scripts. Mais il existe une sémantique plus agréable permettant une meilleure lisibilité dans les scripts.

Cette sémantique consiste à encadrer l'expression à tester entre des crochets (**[expr]**).

Attention, il faut absolument un espace :
- après le crochet ouvrant
- de part et d'autre de l'opérateur de comparaison
- avant le crochet fermant.

Exemple
```
test "a" = "a"
```

est équivalent à

```
[ "a" = "b" ]
```

6.1.2 - Opérateurs de test

6.1.2.1 - Opérateurs liés aux fichiers

Opérateur	Signification
`-r fichier`	vrai si le fichier existe et est accessible en lecture (R)
`-w fichier`	vrai si le fichier existe et est accessible en écriture (W)
`-x fichier`	vrai si le fichier existe et est exécutable (X)
`-f fichier`	vrai si le fichier existe et est un fichier régulier
`-d fichier`	vrai si le fichier existe et est un répertoire
`-s fichier`	vrai si le fichier existe et a une taille non nulle
`-L fichier`	vrai si le fichier existe et est un lien symbolique

6.1.2.2 - Opérateurs liés aux chaînes de caractères (ou expressions régulières)

Opérateur	Signification
`s1 = s2`	vrai si les deux expressions sont égales
`s1 != s2`	vrai si les deux expressions sont différentes
`s1`	vrai si s1 n'est pas la chaîne nulle (`test "a"` est vrai, `test ""` est faux.)

6.1.2.3 - Opérateurs mathématiques

Opérateur	Signification
`e1 -eq e2`	vrai si les deux entiers `e1` et `e2` sont algébriquement égaux
`e1 -ne e2`	vrai si les deux entiers `e1` et `e2` sont différents
`e1 -gt e2`	vrai si `e1` est plus grand que `e2`
`e1 -ge e2`	vrai si `e1` est plus supérieur ou égal à `e2`
`e1 -lt e2`	vrai si `e1` est plus petit que `e2`
`e1 -le e2`	vrai si `e1` est inférieur ou égal à `e2`

6.1.2.4 - Opérateurs logiques

Opérateur	Signification
`!`	négation unaire
`-a`	opération binaire `ET`
`-o`	opération binaire `OU`
`(expr)`	vrai si `expr` est vraie. Permet de regrouper des expressions.
`expr1 && expr2`	vrai si `expr1` et `expr2` sont vraies.
`expr1 || expr2`	vrai si `expr1` ou `expr2` est vraie.

7 - Les structures de contrôle

7.1 - Les tests conditionnels : Si condition alors

Syntaxe :

```
if condition
        then
                commandes
        elif condition
                then
                        commandes
        else commandes
fi
```

Un bloc de commandes peut lui aussi contenir des tests conditionnels.

Exemples :
Si le fichier dont le nom est donné en paramètre existe, alors affichage du type de son contenu.

```
if test -f $1
then
        file $1
else
        echo " le fichier $1 n'existe pas "
fi
```

Il est possible d'utiliser les statuts de n'importe quelle commande Unix dans une condition **if**.
En effet, la commande **grep** retourne **VRAI** quand elle trouve, et **FAUX** quand elle ne trouve pas.

```
if grep jean personnel
then
        echo jean >> disponible
elif grep pierre personnel
        then
                echo pierre >> disponible
        else
                echo vide >> disponible
fi
```

Note :
Dans des shell scripts, il est fréquent que les programmeurs positionnent le mot clé **then** sur la même ligne que le mot clé **if**, pour gagner en lisibilité. Cela oblige à séparer la condition **if** (ou **elif**) du mot **then** par un point virgule (c'est le séparateur de commandes).

Exemple de syntaxe :

```
if condition ; then
   commandes
elif condition ; then
   commandes
else
   commandes
fi
```

On peut aussi écrire comme ceci …rien ne l'interdit…

```
if condition ; then commandes; elif condition ; then commandes; else commandes; fi
```

…sauf le respect pour les lecteurs de vos shells scripts.

7.2 - Les tests conditionnels : La structure Case

La structure case est un moyen à utiliser lorsque l'on souhaiter de tester un trop grand nombre de clauses **elif** (ELSE IF) et que beaucoup de valeurs différentes sont possibles pour une même expression.

Syntaxe :

```
case paramètre in
        choix1[|choix2] ... ) commandes ;;
        choix3 ) commandes ;;
        * ) commandes ;;
esac
```

Chaque valeur du paramètre est terminée par le signe «) ».
Si plusieurs valeurs sont à traiter de la même manière, on les sépare par le signe | :
Exemple :

```
choix1 |choix2 | choix3 )
```

Chaque bloc de commandes se situant dans une des clauses du **case** doit se terminer par deux points-virgule.

La clause par défaut se marque par "***)**"

Exemple :

```
case $1 in
        -d | -r ) # option -d ou -r
                  echo "Vous avez demandé le mode "R"écursif ou "D"irectory
                  mode=R
                  ;;
        -o )      # option -o
                  echo "Vous souhaitez répondre automatiquement par Oui à toute question"
                  OUI=1
                  ;;
        * )       # option inconnue
                  echo "argument incorrect "
                  ;;
esac
```

Note :
Fréquemment, les deux points-virgule sont sur une ligne à part, pour gagner en lisibilité et faciliter l'insertion de lignes de codes. La syntaxe est alors comme dans l'exemple ci-dessus.
De plus, et dans le même but, on termine fréquemment la première ligne, juste après la parenthèse, avec du commentaire (introduit par un **#**).

Cela permet de palier à l'absence de mots clés **begin** et **end** ou d'accolades **{** et **}**.

7.3 - Les boucles Tant que

La boucle tant-que exécute un bloc de commandes tant qu'une condition est remplie.

Syntaxe :

```
while condition
do
        commandes
done
```

Exemple :
Ce shell script concatène dans le fichier **result** l'ensemble des fichiers dont les noms sont donnés en argument.

```
while [ -r " $1 " ]
do
        cat $1 >> result
        shift
done
```

Note :
Comme pour les tests conditionnels, il est possible de mettre le mot clé **do** sur la même ligne que le mot clé **while**, à condition de séparer l'expression testée et le mot **do** par un point-virgule.

7.4 - Les boucles Répéter jusqu'à

La boucle répéter-jusqu'à exécute une commande jusqu'à ce qu'une condition soit remplie.

Syntaxe :

```
until condition
do
        commandes
done
```

Le même exemple que précédemment, mais avec une structure **until**.

```
until [ ! -r " $1 " ]
do
         cat $1 >> concat
         shift
done
```

Note :
Dans cet exemple, notez la négation du test grâce à l'opérateur unaire de négation (**!**)

7.5 - Les boucles For

La boucle For permet en Korn Shell d'exécuter bloc de commandes de manière « pour chaque », et non pas un nombre N de fois, comme dans d'autres langages.

```
for param [in liste]
do
        commandes
done
```

La variable **param** prend successivement les valeurs de la liste.
Si la liste est omise, **param** prend les valeurs passées en paramètres du script.

Exemples : Recopie chaque fichier du répertorie courant dans /tmp.

```
for i in `ls`
do
        cp $i /tmp/$i
        echo "$i copié "
done
```

En effet, `ls`signifie « résultat de l'exécution de ls ».
Les fichiers résultants sont séparés par un blanc, et forment donc une liste valide.
➔ Vérifier cela par **echo `ls`**

Autre exemple : Compte le nombre de fichiers dans les répertoires d'une liste

```
for dir in /dev /usr /users/bin /lib
do
        num=`ls $dir|wc -w`
        echo "$num fichiers dans $dir "
done
```

Autre exemple : Liste les paramètres d'appels du shell script

```
for i
do
        echo $i
done
```

7.6 - L'ordre Select

Syntaxe :

```
select Identificateur [in liste]
do
        commande
        ...
done
```

La commande **select** écrit sur la sortie d'erreur la liste des choix, chacun étant précédé d'un numéro. Si **in liste** n'est pas spécifié, les paramètres positionnels sont utilisés.

Le contenu de la variable **PS3** s'affiche et l'entrée standard est lu. Si le numéro d'un des mots listés est saisi1, le paramètre **Identificateur** prend la valeur de ce mot.

Si la ligne est vide, la liste s'affiche de nouveau. Sinon, la valeur du paramètre **Identificateur** est "". Le contenu de la ligne lue à partir de l'entrée standard est sauvegardé dans le paramètre **REPLY**. La liste est exécutée pour chaque sélection jusqu'à un caractère d'interruption ou de fin de fichier.

L'exemple suivant est la partie « saisie de la réponse » d'un menu.

```
PS3=" votre choix "
select chx in "choix 1"  "choix 2"  "fin"
do
        case $chx in
                "choix 1")# Choix 1
                .... ;;
                "choix 2")# Choix 2
                 ... ;;
                "fin")# Sortie du menu
                 exit 0 ;;
                "") # Réponse invalide
                 echo "$REPLY est une réponse invalide "
        esac
done
```

7.7 - Le calcul, l'évaluation de variables

Le calcul arithmétique sont réalisable grâce à la commande **let**.

7.7.1 - La commande let

Syntaxe :
```
let expression
```

ou aussi

```
((expression))
```

Exemples :
```
let VARIABLE=1
let VARIABLE++
```

Attention :
Ne pas confondre l'expression que l'on emploie avec la commande **let**, avec les expressions régulières, ni avec les expressions dont ont parle dans la commande **test**.

7.7.2 - Les opérateurs arithmétiques

On retrouve en Korn shell l'ensemble des opérateurs du **Langage C**.

7.7.2.1 - Opérateurs unaires

Ces opérateurs sont évalués comme suit :

Opérateur	Signification
+ VALEUR	VALEUR
- VALEUR	Opposé de VALEUR
! ARG	NON logique : vaut 1 su ARG=0, 0 sinon.
~ ARG	NON bit à bit : chaque bit de l'octet subit un NON logique.
++ARG	Incrément préfixé : incrémente ARG de 1 et retourne sa nouvelle valeur
ARG++	Incrément postfixé : retourne ARG et incrémente sa valeur de 1
--ARG	Décrément préfixé
ARG--	Décrément postfixé

7.7.2.2 - Opérateurs d'affectation

Opérateur	Signification
VAR=VALEUR	Affectation
VAR *=x	multiplie VAR par x
VAR */x	divise VAR par x
VAR %=x	met dans VAR le résultat de VAR modulo x
VAR +=x	ajoute x à VAR
VAR -=x	soustrait x à VAR
VAR <<=x	Décalage à gauche bit à bit de x bits.
VAR >>=x	Décalage à droite bit à bit de x bits
VAR &=x	ET logique
VAR ^=x	OU exclusif
VAR |=x	OU logique

7.7.2.3 - Opérateurs binaires

Opérateur	Signification
`||`	OU logique
`|`	OU arithmétique (bit à bit)
`&&`	ET logique
`^`	OU exclusif arithmétique
`&`	ET arithmétique
`==`	Egalité
`!=`	Non égalité
`<`	Inférieur à
`>`	Supérieur à
`<=`	Inférieur ou égal à
`>=`	Supérieur ou égal à
`<<`	Décalage à gauche bit à bit
`>>`	Décalage à droite bit à bit
`+, -, *, /, %`	Addition, soustraction, multiplication, division, modulo
`arg1?arg2:arg3`	Si arg1 n'est pas nul, vaut arg2, sinon vaut arg3

7.7.2.4 - Regroupements

Le groupement d'expressions se fait entre parenthèses.

Exemple :

```
VAR=(a+b)*c
```

7.8 - Les fonctions et Librairies en Korn Shell

Il est possible de déclarer des fonctions en Korn Shell
On peut développer des fonctions récursives.

7.8.1 - Déclaration d'une fonction

Syntaxe :

```
function Nom {
LIGNES DE CODE ;
}
```

7.8.2 - Déclaration des variables

`typeset var`	déclaration d'une chaîne de caractères
`integer var` `typeset -i var`	déclaration d'un entier
`typeset -r var=valeur` `readonly var=valeur`	définition d'une constante

On peut utiliser des tableaux qui ne sont déclarés que lors de leur assignation:
`tab[100]=toto`

7.8.3 - Récupération des résultats de la fonction

On positionne le code retour de la fonction par `return num`
Dans le code shell script appelant , il est récupéré par `$?`

On renvoie une valeur par l'intermédiaire de la commande **echo**.
Celle-ci est alors récupérée par :
`VAR=`fonction` ou VAR=$(fonction)`

7.8.4 - Les librairies

Il est possible d'émuler le système de librairies dynamiques, dans lesquelles le Korn Shell saura aller charger des fonctions (à condition que l'on lui indique de le faire).
Pour cela, on crée un répertoire dans lequel on stocke les fichiers contenant les fonctions (une fonction par fichier, et le nom du fichier doit être le même que le nom de la fonction).

Pour utiliser cette librairie, positionner la variable `FPATH`, puis importer les fonctions (parle mot clé `autoload` (qui est un alias pour `typeset -fu`) ou par l'instruction `typeset -fu`).

Exemple :

```
FPATH=$HOME/lib/rep1:$HOME/lib/rep2
tyteset -fu nom_fonction
```

On peut définir des fonctions dans le fichiers définie par `ENV` mais elles ne seront visibles que du shell interactif.
Pour les rendre toujours visible il faut les exporter par **typeset -fx nom_fonction**. Cela doit être fait pour chaque fonction.

7.9 - Commandes diverses

Opérateur	Signification
#	Introduit un commentaires
(cmd1 ; cmd2)	exécute la commande dans un sous-shell
read VAR	lecture d'une sur l'entrée standard et positionnement dans **VAR**
exit **num**	Sort avec le code retour **num** du shell-script. (On choisira en général 0 si la commande s'est bien exécutée)
return num	Sort avec le code retour **num** d'une fonction.
cmd1 && **cmd2**	Séparateur conditionnel (**cmd2** sera exécuté si **cmd1** a un code retour de 0)
cmd1 || **cmd2**	Séparateur conditionnel (**cmd2** sera exécuté si **cmd1** a un code retour différent de 0)
readonly var	empêche la modification d'une variable

8 - AWK

8.1 - Introduction

Le langage `AWK` est prévu pour traiter des fichiers texte. Si `prog.awk` contient le programme, et si on veut traiter le fichier `demo.txt`, on écrit :

```
awk -fprog.awk demo.txt
```

8.1.1 - Principe

Il y a ouverture du fichier, parcours ligne par ligne et action éventuelle, puis fermeture du fichier.

La ligne courante a pour nom fixé `$0`.
Elle contient `NF` mots et le *ième* mot s'appelle `$i`.
L'affichage se fait par `print` ou `printf` (**f** pour **formatté**).

Le traitement de chaque ligne se fait entre accolades `{` et `}`.
On délimite les instructions par un `;`.
Les commentaires commencent par `#`.
On distingue majuscule et minuscule.

La syntaxe de base est la même que celle du langage C, sauf qu'on ne déclare pas les variables.

8.1.2 - Exemples

Avec le programme :

```
{ print $1 " moyenne " ($2 +$3)/2 }
```

et le fichier :

```
BOBY                    18 5
ZELYNOU                 6 11
ANTIN                   8 4
BOB                     16 8 15
IZEL                    16 18 12
```

on obtient comme affichage :

```
BOBY moyenne 11.5
ZELYNOU moyenne 8.5
ANTIN moyenne 6
BOB moyenne 12
IZEL moyenne 17
```

De même, avec le programme

```
{   print $1
    printf(" moyenne %5.2f\n",($2+$3)/2 )
} # fin de traitement de la ligne courante
```

et le fichier :

```
BOBY                    18 5
ZELYNOU                 6 11
ANTIN               8 4
BOB                16 8 15
IZEL                16 18 12
```

on obtient comme affichage :

```
BOBY
moyenne  11.50
ZELYNOU
moyenne   8.50
ANTIN
moyenne   6.50
BOB
moyenne  12.00
IZEL
moyenne  17.00
```

8.2 - Variables Et Fonctions

Awk met à la disposition du programmeur :

Des variables

NR	**numéro global de l'enregistrement**
FNR	**numéro local de l'enregistrement**
FILENAME	**nom du fichier ouvert**
ARGV	**vecteur des arguments passés**

Et des fonctions

length(H)	**donne la longueur de H**
substr(C,D,F)	**renvoie F caractères de C à partir de D**
index(T,C)	**donne la position de C dans T ou 0**
system(D)	**exécute la commande dos D**
close(F)	**ferme la fichier F**
gsub(W,S,T)	**remplace R par S dans T**

Les éléments de tableaux se notent entre crochets.

Ainsi **`T[i]`** désigne l'élément en position **`i`** (**`i`** peut être numérique ou caractère).

8.3 - Instructions

Les instructions classiques :

=	**affectation**
if	**conditionnelle**
for	**boucle pour**
while	**boucle tant que**
++	**incrémentation de 1**
+=	**incrémentation variable**

Exemple :

```
while( i*i < n ) { i++ }
for (i=a;i<=b;i+=2) {
    if (i*i==n) { print "trouvé" }
       else     { print i n i*i  }
} # finpour i
```

Awk initialise automatiquement les variables. Ainsi { **`print i+1`** } met automatiquement **`0`** dans **`i`** puis affiche **`1`**.

8.4 - Expressions Régulières

8.4.1 - Filtrage

AWK est surtout intéressant pour son mécanisme de filtrage et ses expressions régulières.
Le traitement d'une ligne courante se fait par et `{ action(s) }`, avec comme cas particuliers :

Pas de filtrage, on traite toutes les lignes, la syntaxe est alors :
`{ instruction(s) }`

Pas d'action, on affiche la ligne en cours.
Filtrage d'un champ : `{ print $0 }`

8.4.2 - Expressions régulières

8.4.2.1 - Forme

Le filtrage se réalise aussi par

```
/ [expression régulière] /

/ [expression régulière] / ~ [nom de variable]

( condition ) .
```

Une expression régulière est une chaîne de caractère définie par des caractères, des répétitions de caractères ou des positions de caractères.

8.4.2.2 - Quantification

Après les caractères, on peut mettre un critère de quantification :

- `*` pour indiquer la répétition 0 fois ou plus
- `+` pour indiquer la répétition 1 fois ou plus
- `?` pour indiquer la répétition 0 fois ou 1 fois

Le critère s'applique au caractère qui le précède.

Exemples :
`A*B` correspond à `B` ou `AB` ou `AAB` ou `AAAB` etc.. .

`0+,?1` correspond à `01` ou à `0,1` ou à `001` ou à `00,1` etc…

8.4.2.3 - Positionnement

On dispose des symboles de positionnement suivants :

- `[]` pour grouper des éléments
- `^` pour indiquer le début de chaîne
- `$` pour indiquer la fin de chaîne
- `|` pour indiquer un choix
- `.` pour indiquer n'importe quel caractère (le « `.` » se note `\.`)

8.4.3 - Exemples

8.4.3.1 - Quelques expressions régulières simples

`^B`	chaîne qui commence par B
`G$`	chaîne qui finit par G
`^.$`	chaîne à un seul caractère
`[AEIOU]`	chaîne avec une seule voyelle majusucle
`^[ABC]`	chaîne qui commence par A ou B ou C
`^[^a-z]$`	chaîne à un seul caractère qui n'est pas une minuscule

8.4.3.2 - Quelques expressions régulières techniques

`^[0-9]+$`	chaîne non nulle avec seulement des chiffres
`^(\+|-)?[0-9]+\.?[0-0]*$`	nombre réel avec éventuellement, signe, point et des décimales

8.4.4 - Compléments

Certains caractères ne peuvent pas être utilisés directement dans les expressions régulières. On met un `\` devant. Ainsi, `/\\/` désigne les chaînes qui contiennent `/` et `/\./` celles qui contiennent un `.` .

On peut exécuter des instructions avant l'ouverture du fichier grâce à **`BEGIN { }`** ou après la fermeture du fichier grâce à **`END { }`**.

Attention : **`FILENAME`** n'est pas disponible dans la partie **`BEGIN`**.

Si on lance un programme **`AWK`** sur plusieurs fichiers, par exemple par **`awk -fprog.awk *.c`**,
Alors :
- **`NR`** est le numéro de ligne global (comme si tous les fichiers étaient concaténés),
- **`FNR`** est le numéro de ligne pour le seul fichier en cours.

Pour les tableaux, le fait de donner un indice crée cet indice. Le tableau est automatiquement trié par ordre croissant d'indice, qu'il soit numérique ou chaîne. On extrait les indices par la boucle **`for .. in`**.

9 - Annexes

9.1 - Index Alphabétique