Introduction aux Scripts Shell

Prof. Dr. Merlec MPYANA

Sciences et Ingénierie informatiques
(Génie Logiciel)

Mars 2023

Pourquoi programmer en shell ?

- **Une connaissance fonctionnelle de la programmation shell est essentielle**
  - **À quiconque souhaite devenir efficace en administration système**
  - **Une compréhension détaillée des scripts d'administration est importante**
    - **Pour analyser le comportement du système, voire le modifier.**
  - **La seule façon pour vraiment apprendre la programmation des scripts est d'écrire des scripts.**

Quand ne pas programmer en shell ?

- **Pour des tâches demandant beaucoup de ressources ou beaucoup de rapidité.**
- **Pour une application complexe où une programmation structurée est nécessaire**
  - typage des variables, prototypage de fonctions, tableaux multidimensionnels, listes chaînées, arbres,…
- **Pour des situations où la sécurité est importante** (protection contre l'intrusion, le vandalisme).
- **Pour des applications qui accèdent directement au matériel.**
- **Pour des applications devant générer/utiliser une interface graphique utilisateur (G.U.I.).**
- **Pour des applications propriétaires (un script est forcément lisible par celui qui l'utilise).**

Introduction au Shell

- **Il existe deux moyens de « programmer » en Shell.**
  - **Le premier est dit en « direct »**
    - **L'utilisateur tape « directement » la ou les commandes qu'il veut lancer.**
    - **Si cette commande a une syntaxe qui l'oblige à être découpée en plusieurs lignes,**
      - le Shell indiquera par l'affichage d'un « prompt secondaire » que la commande attend une suite et n 'exécutera réellement la commande qu'à la fin de la dernière ligne.

```
Prompt> date
Tue Jan 16 17:26:50 NFT 2001
Prompt> pwd
/tmp
Prompt > if test 5 = 5
Prompt secondaire> then
Prompt secondaire> echo vrai
Prompt secondaire> fi
vrai
```

Introduction au Shell

- **Il existe deux moyens de « programmer » en Shell.**
  - **Le second est dit en « script », appelé aussi « batch » ou « source Shell ».**
    - **L'utilisateur crée un fichier texte par l'éditeur de son choix (par exemple : « vi »).**
    - **Il met dans ce script toutes les commandes qu'il voudra lui faire exécuter**
      - en respectant la règle de base de ne mettre qu'une seule commande par ligne.
      - Une fois ce script fini et sauvegardé, il le rend exécutable par l'adjonction du droit « x »

```
chmod u+x nom_fichier_script
```

Introduction au Shell

- **Il existe deux moyens de « programmer » en Shell.**
  - **Le second est dit en « script », appelé aussi « batch » ou « source Shell ».**
    - **L'utilisateur crée un fichier texte par l'éditeur de son choix (par exemple : « vi »).**
    - **Il met dans ce script toutes les commandes qu'il voudra lui faire exécuter**
      - en respectant la règle de base de ne mettre qu'une seule commande par ligne.
      - Une fois ce script fini et sauvegardé, il le rend exécutable par l'adjonction du droit « x »

Rendre le ficher exécutable: `chmod u+x nom_fichier_script.sh`

Exécuter le fichier script: `./nom_fichier_script.sh`

Introduction au Shell

- **Il existe deux moyens de « programmer » en Shell.**
  - **Le second est dit en « script », appelé aussi « batch » ou « source Shell ».**
    - **Exemple**

```
Lancement de l'éditeur
Prompt> vi prog
Mise à jour du droit d'exécution
Prompt> chmod a+x prog
Exécution du script pris dans le répertoire courant
Prompt> ./prog
Tue Jan 16 17:26:50 NFT 2001
/tmp
oui
```

Contenu du fichier «prog»

```
date
pwd
if test 5 = 5
then
    echo "vrai"
fi
```

Les commentaires

- **Un commentaire sert à améliorer la lisibilité du script.**
  - **Il est placé en le faisant précéder du caractère croisillon (« # »).**
  - **Ex: Ecrire un script shell qui affiche la date.**

```
#!/bin/sh

# Ce programme affiche la date

date # Cette ligne est la ligne qui affiche la date
```

Le débogueur

- **Afin de détecter l'erreur, le Shell offre un outil de débogage.**
  - **Il s'agit de l'instruction « *set* » agrémentée d'une ou plusieurs options suivantes :**
    - v : affichage de chaque instruction avant analyse => il affiche le nom des variables ;
    - x : affichage de chaque instruction après analyse => il affiche le contenu des variables ;
    - e : sortie immédiate sur erreur.
    - Chaque instruction « set -… » active l'outil demandé qui sera désactivé par l'instruction « set +… »

```
#!/bin/sh
set -x # Activation du débogage à partir de maintenant
date # Cette ligne est la ligne qui affiche la date
set +x # Désactivation du débogage à partir de maintenant
```

Les variables

- **Une variable sert à mémoriser une information afin de la réutiliser ultérieurement.**
  - **Elles sont créées par le programmeur au moment où il en a besoin.**
  - **Il n'a pas besoin de les déclarer d'un type particulier.**
  - **On peut en créer quand on veut, où on veut.**
  - **Leur nom est représenté par une suite de caractères**
    - commençant impérativement par une lettre ou le caractère _ (souligné ou underscore) et
    - comportant ensuite des lettres, des chiffres ou le caractère souligné.
    - Il ne doit pas correspondre à un des mots-clefs du Shell.

Les variables

- **Une variable sert à mémoriser une information afin de la réutiliser ultérieurement.**
  - **Leur contenu est interprété exclusivement comme du texte.**
  - **Il n'existe donc pas, en Bourne Shell,**
    - d'instruction d'opération sur des variables ou sur du nombre (addition, soustraction…).
  - **Il n'est pas non plus possible d'avoir des variables dimensionnées (tableaux).**
    - Mais cela est possible en Korn Shell et Bourne Again Shell (et shells descendants).
  - **Il est important de noter que le Shell reconnaît, pour toute variable, deux états :**
    - non définie (non-existence) : elle n'existe pas dans la mémoire ;
    - définie (existence) : elle existe dans la mémoire ; même si elle est vide.

Les variables simples

❑ L'affectation et l'accès aux variables simples.

◘ Syntaxe

```
variable = chaîne
```

◘ L'affectation d'une variable simple (il n'y a pas de possibilité de créer de tableau en Bourne Shell) se fait par la syntaxe « variable=chaîne ».

C'est la seule syntaxe du Shell qui ne veuille pas d'espace dans sa structure sous peine d'avoir une erreur lors de l'exécution.

Les variables simples

- **Dans le cas où on voudrait entrer une chaîne avec des espaces dans la variable**
  - il faut alors encadrer la chaîne par des guillemets simples ou doubles (la différence entre les deux sera vue plus tard).
  - À partir du moment où elle a été affectée, une variable se met à exister dans la mémoire, même si elle a été affectée avec « rien ».
  - L'accès au contenu de la variable s'obtient en faisant précéder le nom de la variable du ca ractère « $ ».

Les variables simples

- **Exemple**

```
Prompt> nom= "Merlec" # Affectation de « Merlec" dans la variable "nom"
Prompt> objet="voiture" # Affectation de "voiture" dans la variable "objet"
Prompt> coul=blanche # Affectation de "blanche" dans la variable "coul"
Prompt> echo "Il se nomme $nom" # Affichage d'un texte et d'une variable
Prompt> txt="$nom a une $objet $coul" # Mélange de variables et texte dans une variable
Prompt> echo txt # Attention à ne pas oublier le caractère "$"
Prompt>echo $txt # Affichage de la variable "txt"
```

Les tableaux (shells évolués)

- **Le Korn Shell, Bourne Again Shell (et shells descendants)**
  - permettent de créer des tableaux à une seule dimension.
  - L'affectation d'un élément d'un tableau se fait par la syntaxe « variable[n]=chaîne ».
    - Dans ce cas précis, les crochets ne signifient pas « élément facultatif », mais bien « crochets » et le pro grammeur doit les mettre dans sa syntaxe.
    - **Syntaxe:**

```
variable[n]=chaîne
variable=(chaîne1 chaîne2 ?)
```

      - L'indice « n » que l'on spécifie entre les crochets doit être impérativement positif ou nul, mais il n'a pas de limite maximale.

Les tableaux (shells évolués)

- **L'accès au contenu de la variable d'indice « n »**
  - **s'obtient en encadrant le nom de la variable indicée par des accolades « { } » et en faisant précéder le tout du caractère « $ ».**
  - **Si on remplace la valeur de l'indice par le caractère « * »,**
    - le Shell concaténera tous les éléments du tableau en une chaîne unique et renverra cette dernière.
  - **Et si on remplace la valeur de l'indice par le caractère « @ »**
    - le Shell transformera chaque élément du tableau en chaîne et renverra ensuite l'ensemble de toutes ces chaînes con caténées. Visuellement, il n'y a aucune différence dans le résultat entre l'utilisation des caractères « * » ou « @ ».
- **Dans des environnements de shells acceptant les tableaux,**
  - toute variable simple est automatiquement transformée en tableau à un seul élément d'indice « [0] ».

Les tableaux (shells évolués)

- **Exemple**

```
Prompt> nom[1]="Merlec" # Affectation de "Pierre" dans la variable "nom[1]"
Prompt> nom[5]="Ben" # Affectation de « Ben" dans la variable "nom[5]"
Prompt> nom="Jean" # Affectation de « Jean" dans la variable "nom[0]"
Prompt> i=5 # Affectation de "5" à la variable "i" (ou "i[0]")
Prompt>prenom=(Pim Pam Poum) # Affectation de 3 prénoms dans un tableau
Prompt> echo "Voici mes 3 noms: ${nom[0]}, ${nom[1]} et ${nom[$i]}"
Prompt> echo "Mon tableau de noms contient ${nom[*]}"
Prompt> echo "Mon tableau de prénoms contient ${prenom[@]}"
```

Tous types de Shell

Variable	Description
${var}	renvoie le contenu de « $var ». Il sert à isoler le nom de la variable par rapport au contexte de son utilisation. Ceci évite les confusions entre ce que l'utilisateur désire « ${prix}F » (variable « prix » suivie du caractère « F ») et ce que le Shell comprendrait si on écrivait simplement « $prixF » (variable « prixF »).
${var-texte}	renvoie le contenu de la variable « var » si celle-ci est définie (existe en mémoire) ; sinon renvoie le texte « texte ».
${var:-texte}	renvoie le contenu de la variable « var » si celle-ci est définie et non vide ; sinon renvoie le texte « texte »
${var+texte}	renvoie le texte « texte » si la variable « var » est définie ; sinon ne renvoie rien.
${var:+texte}	renvoie le texte « texte » si la variable « var » est définie et non vide ; sinon ne renvoie rien.
${var?texte}	renvoie le contenu de la variable « var » si celle-ci est définie ; sinon affiche le texte « texte » comme messa ge d'erreur (implique donc l'arrêt du script).
${var:?texte}	renvoie le contenu de la variable « var » si celle-ci est définie et non vide ; sinon affiche le texte « texte » (co mme « ${var:-texte} ») comme message d'erreur (implique donc l'arrêt du script).
${var=texte}	renvoie le contenu de la variable « var » si celle-ci est définie, sinon affecte le texte « texte » à la variable « v ar » avant de renvoyer son contenu.
${var:=texte}	renvoie le contenu de la variable « var » si celle-ci est définie et non vide, sinon affecte le texte « texte » à la variable « var » avant de renvoyer son contenu.

Uniquement en Korn Shell et Bourne Again Shell (et shells descendants)

Variable	Description
${var[n]}	renvoie le contenu du nie élément du tableau « var ».
${var[*]}	concatène tous les éléments présents dans le tableau « var » en une chaîne unique et renvoie cette dernière.
${var[@]}	transforme individuellement chaque élément présent dans le tableau « var » en une chaîne et renvoie la concaténation de toutes les chaînes.
${var#texte}	si « texte » contient un métacaractère, alors il sera étendu jusqu'à la plus petite correspondance avec le contenu de « var » pris à partir du début. Si cette correspondance est trouvée, elle est alors supprimée du début de « var »
${var##texte}	si « texte » contient un métacaractère, alors il sera étendu jusqu'à sa plus grande correspondance avec le contenu de « var » pris à partir du début. Si cette correspondance est trouvée, elle est alors supprimée du début de « var ».
${var%texte}	si « texte » contient un métacaractère, alors il sera étendu jusqu'à sa plus petite correspondance avec le contenu de « var » pris à partir de la fin. Si cette correspondance est trouvée, elle est alors supprimée de la fin de « var ».
${var%%texte}	si « texte » contient un métacaractère, alors il sera étendu jusqu'à sa plus grande correspondance avec le contenu de « var » pris à partir de la fin. Si cette correspondance est trouvée, elle est alors supprimée de la fin de « var ».
${#var}	envoie le nombre de caractères contenus dans la variable « var ». Si la variable est un tableau, renvoie alors le nombre d'éléments du tableau.
$((expression))	renvoie la valeur numérique de l'expression demandée.

Uniquement en Bourne Again Shell (et shells descendants)

Variable	Description
${!var}	utilise le contenu de la variable « var » comme un nom de variable et renvoie le contenu de cette dernière (permet donc de simuler un pointeur).
${var:x:y}	renvoie les « y » caractères de la variable « var » à partir du caractère n° « x » (attention, le premier cara ctère d'une variable porte le n° « 0 »). Si la variable est un tableau, renvoie alors les « y » éléments du ta bleau « var » à partir de l'élément n° « x ».
${var:x}	renvoie la fin de la variable « var » à partir du caractère n° « x » (attention, le premier caractère d'une vari able porte le n° « 0 »). Si la variable est un tableau, renvoie alors les derniers éléments du tableau « var » à partir de l'élément n° « x ».
${var/texte1/texte2}	envoie le contenu de « var », mais en lui remplaçant la première occurrence de la chaîne « texte1 » par la chaîne « texte2 ».
${var//texte1/texte2}	renvoie le contenu de « var », mais en lui remplaçant chaque occurrence de la chaîne « texte1 » par la chaîne « texte2 ».

- **L'imbrication de séquenceurs est possible.**
  - Ainsi, la syntaxe « ${var1:-${var2:-texte}} » renvoie le contenu de la variable « var1 » si celle-ci est définie et non nulle ; sinon, renvoie le contenu de la variable « var2 » si celle-ci est définie et non nulle ; sinon renvoie le texte « texte ».

Les variables prédéfinies

Variable	Signification
$HOME	Répertoire personnel de l'utilisateur
$PWD	Répertoire courant (uniquement en « Korn Shell » ou « Bourne Again Shell » et shells descendants)
$OLDPWD	Répertoire dans lequel on était avant notre dernier changement de répertoire (uniquement en « Korn Shell » ou « Bourne Again Shell » et shells descendants)
$LOGNAME	Nom de login
$PATH	Chemins de recherche des commandes
$CDPATH	Chemins de recherche du répertoire demandé par la commande « **cd** »
$PS1	Prompt principal (invite à taper une commande)
$PS2	Prompt secondaire (indique que la commande n'est pas terminée)
$PS3	Prompt utilisé par la commande « **select** » (uniquement en « Korn Shell » et « Bourne Again Shell » et shells descendants)
$PS4	Prompt affiché lors de l'utilisation du mode débogueur « **set -x** »
$TERM	Type de terminal utilisé
$REPLY	Chaîne saisie par l'utilisateur si la commande « **read** » a été employée sans argument (uniquement en « Korn Shell » et « Bourne Again Shell » et shells descendants). Numéro choisi par l'utilisateur dans la commande « **select** » (uniquement en « Korn Shell » et « Bourne Again Shell » et shells descendants)
$IFS	Séparateur de champs internes
$SHELL	Nom du Shell qui sera lancé chaque fois qu'on demandera l'ouverture d'un Shell dans une application interactive (« **vi** », « **ftp** »…)
$RANDOM	Nombre aléatoire entre 0 et 32 767 (uniquement en « Korn Shell » et « Bourne Again Shell » et shells descendants)
$$	Numéro du processus courant
$!	Numéro du dernier processus lancé en arrière-plan
$?	Statut (état final) de la dernière commande

La saisie en interactif

- **Cette action est nécessaire lorsque le programmeur désire demander une information ponctuelle à celui qui utilise le programme.**
  - Syntaxe:

```
read [var1 var2 ?]
```

  - À l'exécution de la commande, le programme attendra du fichier standard d'entrée une chaîne terminée par la touche « Entrée » ou « fin de ligne ».
  - Une fois la saisie validée, chaque mot (séparé par un « espace ») sera stocké dans chaque variable (« var1 »,« var2 »…).
  - En cas d'excédent, celui-ci sera stocké dans la dernière variable.
  - En cas de manque, les variables non remplies seront automatiquement définies, mais vides.

La saisie en interactif

❑ Si aucune variable n'est demandée

- la chaîne saisie sera stockée dans la variable interne « $REPLY » (uniquement en Korn Shell, Bourne Again Shell et shells descendants).

❑ Le Korn Shell et le Bourne Again Shell (et les shells descendants)

- permettent d'affecter automatiquement chaque mot en provenance du fichier standard d'entrée dans les éléments d'un tableau.
- **Syntaxe:**

```
read -a tableau # Korn Shell et Bourne Again Shell (et shells descendants)
```

- Le premier mot ira dans le tableau d'indice « 0 », le second dans le tableau d'indice « 1 »…
- Cette syntaxe remplace tout le tableau par les seules chaînes provenant de l'entrée standard.
- L'ancien éventuel contenu disparaît alors pour être remplacé par le nouveau.

La protection et La suppression

❑ Protection

```
readonly var1 [var2 ?]
readonly
```

- Employée sans argument, l'instruction « readonly » donne la liste de toutes les variables protégées.
  - Une fois verrouillée, la variable ne disparaîtra qu'à la mort du processus qui l'utilise.
  - Cette commande, lorsqu'elle est employée sur une variable, la verrouille contre toute modification et/ou suppression, volontaire ou accidentelle.

❑ Suppression

```
unset var1 [var2 ?]
```

- Le mot « suppression » rend la variable à l'état de « non défini » ou « non existant ».
- Il y a libération de l'espace mémoire affecté à la variable ciblée.
- Il ne faut donc pas confondre « variable supprimée » et « variable vide ».

La visibilité des Variables

❑ Sans exportation

- Action : affectation de « var »

```
Prompt> var=Bonjour
```

- Action : affichage de « var »

```
Prompt> echo $var
Bonjour
```

- Résultat : « var » est bien créée.
- Action : lancement du script « prog »

```
Prompt> ./prog
Contenu de var :[]
Contenu de var :[Salut]
```

- Résultat : « prog » ne connaît pas « var » (ou « var » n'existe pas dans « prog »).
- Puis « var » est créée et ensuite, elle est affichée.

La visibilité des Variables

❑ Sans exportation

- Action : affichage de « var »

```
Prompt> echo $var
Bonjour
```

- Résultat : malgré la modification faite dans le script, « var » n'a pas changé.

❑ Avec exportation

- Action : affichage de « var »

```
Prompt> var=Bonjour
Prompt> export var
```

- Action : affichage de « var »

```
Prompt> echo $var
Bonjour
```

- Résultat : « var » est bien créée.

La visibilité des Variables

❑ Avec exportation

- Action : lancement du script « prog »

```
Prompt> ./prog
Contenu de var :[Bonjour]
Contenu de var :[Salut]
```

- Résultat : ici, « prog » connaît « var » puis « var » est modifiée et ensuite, elle est affichée.

- Action : affichage de « var »

```
Prompt> echo $var
Bonjour
```

- Résultat : malgré la modification faite dans le script et bien qu'il y ait un export, « var » n'a toujours pas changé.

Le typage

- **Syntaxe**

```
typeset [-a] [-i] [-r] [-x] var1 [var2 ?]
typeset
```

- **Les shells évolués** (Korn Shell, Bourne Again Shell et autres descendants)
  - Permettent de restreindre les propriétés des variables et correspondent à une certaine forme de typage « simpliste ».
    - `typeset -a var`: la variable sera traitée comme un tableau.
    - `typeset -i var`: la variable sera traitée comme un entier et peut être incluse dans des opérations arithmétiques.
    - `typeset -r var`: la variable sera mise en « lecture seule » (équivalent de « readonly »).
    - `typeset -x var`: la variable sera exportée automatiquement dans les processus fils (équivalent de « export »).
  - À noter : l'instruction « *declare* » accessible uniquement en Bourne Again Shell (et autres descendants) est un synonyme de l'instruction « *typeset* ».

Les paramètres

- **Un paramètre, appelé aussi « argument »**
  - **Il est un élément (chaîne de caractères) situé entre le nom du programme et la touche « Entrée » qui active le programme.**
  - **Il s'agit en général d'éléments que le programme ne connaît pas à l'avance**
    - Mais dont il a évidemment besoin pour travailler.
  - **Ces éléments peuvent être nécessaires au programme pour son bon fonctionnement**

```
Prompt> cp fic1 fic2 # Commande "cp", argument1 "fic1",
argument2 "fic2"
Prompt> ls -l # Commande "ls", argument1 "-l"
Prompt> cd # Commande "cd" sans argument
```

Les paramètres

- **Un paramètre, appelé aussi « argument »**
  - **Il est un élément (chaîne de caractères) situé entre le nom du programme et la touche « Entrée » qui active le programme.**
  - **Il s'agit en général d'éléments que le programme ne connaît pas à l'avance**
    - Mais dont il a évidemment besoin pour travailler.
  - **Ces éléments peuvent être nécessaires au programme pour son bon fonctionnement**

```
Prompt> cp fic1 fic2 # Commande "cp", argument1 "fic1",
argument2 "fic2"
Prompt> ls -l # Commande "ls", argument1 "-l"
Prompt> cd # Commande "cd" sans argument
```

Récupération des paramètres

- **Dans un script, les paramètres ou arguments, positionnés par l'utilisateur exécutant le script, sont automatiquement et toujours stockés dans des « *variables automatiques* » (remplies automatiquement par le Shell).**
  - **Ces variables sont :**
  - $0 : nom du script. Le contenu de cette variable est invariable.
  - $1, $2, $3, …, $9 : argument placé en première, seconde, … neuvième position derrière le nom du scrip;
  - $# : nombre d'arguments passés au script ;
  - $* : liste de tous les arguments (sauf $0) concaténés en une chaîne unique ;
  - $@ : liste de tous les arguments (sauf $0) transformés individuellement en chaîne.

    Visuellement, il n'y a pas de différence entre « $* » et « $@ ».

Récupération des paramètres

Exemple d'un script « prog »

```
#!/bin/sh
echo $0 # Affichage nom du script
echo $1 # Affichage argument n° 1
echo $2 # Affichage argument n° 2
echo $5 # Affichage argument n° 5
echo $# # Affichage du nombre d'arguments
echo $* # Affichage de tous les arguments
```

Récupération des paramètres

Exemple d'un script « prog »

```
#!/bin/sh
echo $0 # Affichage nom du script
echo $1 # Affichage argument n° 1
echo $2 # Affichage argument n° 2
echo $5 # Affichage argument n° 5
echo $# # Affichage du nombre d'arguments
echo $* # Affichage de tous les arguments
```

Résultat de l'exécution

```
Prompt> ./prog
./prog
0
Prompt> ./prog a b c d e f g h i j k l m
./prog
a
b
e
13
a b c d e f g h i j k l m
```

Décalage des paramètres

- **Comme on peut le remarquer, le programmeur n'a accès de façon individuelle qu'aux variables « $1 » à « $9 ».**
  - **Si le nombre de paramètres dépasse neuf, ils sont pris en compte par le script Shell.**
    - mais le programmeur n'y a pas accès de manière individuelle.
  - **Il peut y accéder en passant par la variable « $* »**
    - mais il devra alors se livrer à des manipulations difficiles d'extraction de chaîne.
    - Ainsi, la commande « echo $10 » produira l'affichage de la variable « $1 » suivi du caractère « 0 ».

Décalage des paramètres

- **Il existe néanmoins en « Bourne Shell » un moyen d'accéder aux arguments supérieurs à neuf : il s'agit de l'instruction « shift [n] ».**
  - **« *n* » étant facultativement positionné à « 1 » s'il n'est pas renseigné.**

```
Syntaxe:
shift [n]
```

  - **Cette instruction produit un décalage des paramètres vers la gauche de « *n* » positions.**

- Dans le cas de « shift » ou « shift 1 », le contenu de « $1 » disparaît pour être remplacé par celui de « $2 »;
- Celui de « $2 » fait de même pour recevoir le contenu de « $3 »… jusqu'à « $9 » qui reçoit le contenu du dixi ème argument.

Décalage des paramètres

❑ Remarque

- **L'instruction « *shift 0* » ne décale pas les paramètres, mais elle est autorisée afin de ne pas générer d'erreur dans un programme**
  - si par exemple la valeur qui suit le « shift » est issue d'un calcul, il sera inutile d'aller vérifier que ce calcul ne vaut pas « 0 ».

Décalage des paramètres

- **Exemple: Script qui récupère et affiche le 1er, 2e, 12e et 14e paramètres :**

```
#!/bin/sh
# Ce script récupère et affiche le 1er, 2e, 12e et 14e paramètre

# Récupération des deux premiers paramètres qui seront perdus
après le "shift"
prem=$1
sec=$2

# Décalage de 11 positions pour pouvoir accéder aux 12e et 14e
paramètres
shift 11

# Affichage des paramètres demandés (le 12e et le 14e ont été
amenés en position 1 et 3 par le "shift")
echo "Les paramètres sont $prem, $sec, $1, $3"
```

Réaffectation volontaire des paramètres

- **L'instruction « set [--] valeur1 [valeur2 …] »**
  - **(qui sert à activer des options du Shell comme le debug)**
  - **Permet aussi de remplir les variables « $1 », « $2 »…, « $9 », au mépris de leur ancien éventuel contenu, avec les valeurs indiquées.**

❖ Il y a d'abord effacement de toutes les variables puis remplissage avec les valeurs provenant du « set ».

```
Syntaxe:
set [--] valeur1 [valeur2 ?]
```

❖ Les variables « $# », « $* » et « $@ » sont aussi modifiées pour correspondre à la nouvelle réalité. Comme toujours, la variable « $0 » n'est pas modifiée.

La commande test

- « test » renvoie donc un statut vrai ou faux.
  - Mais cette commande n'affiche rien à l'écran.
    - Il faut donc, pour connaître le résultat d'un test, vérifier le contenu de la variable « $? ».
  - La commande « test » a pour but de vérifier (tester) la validité de l'expression demandée.
    - en fonction des options choisies. Elle permet ainsi de vérifier l'état des fichiers, comparer des variables…

```
Syntaxe:
test option "fichier"
```

La commande test

Option	Signification
-s	fichier « non vide »
-f	fichier « ordinaire »
-d	fichier « répertoire »
-b	fichier « spécial » mode « bloc »
-c	fichier « spécial » mode « caractère »
-p	fichier « tube »
-L	fichier « lien symbolique »
-h	fichier « lien symbolique » (identique à « *-L* »)
-r	fichier a le droit en lecture
-w	fichier a le droit en écriture
-x	fichier a le droit en exécution
-u	fichier a le « setuid »
-g	fichier a le « setgid »
-k	fichier a le « sticky bit »
-t [n]	fichier n° « n » est associé à un terminal (par défaut, « n » vaut « 1 »)

Test complexe sur plusieurs fichiers

Syntaxe:

```
test "fichier1" option "fichier2"
```

Option	Signification
-nt	fichier1 plus récent que fichier2 (date de modification)
-ot	fichier1 plus vieux que fichier 2 (date de modification)
-ef	fichier1 lié à fichier2 (même numéro d'inode sur même système de fichiers)

Test sur les chaînes de caractères

Syntaxe:

```
test "chaîne1" option "chaîne2"
```

Opérateurs

Opérateur	Signification
=	chaîne1 identique à chaîne2
!=	chaîne1 différente de chaîne2

⚠ L'emploi des doubles-guillemets dans les syntaxes faisant intervenir des chaînes est important surtout lorsque ces chaînes sont prises à partir de variables. En effet, il est possible d'écrire l'expression sans double-guillemet, mais si la variable est vide ou inexistante, l'expression reçue par le Shell sera bancale et ne correspondra pas au schéma attendu dans la commande « test ».

Test sur les chaînes de caractères

- **Exemple d'un test sur les chaînes de caractères.**

```
test $a = bonjour # Si a est vide, le shell voit test = bonjour (incorrect)
test "$a" = "bonjour" # Si a est vide, le shell voit test "" = "bonjour" (correct)
```

Test sur les chaînes de caractères

Remarques

ksh 88 et bash proposent aussi la syntaxe des « doubles-crochets » qui est une version étendue de la commande test. Cette syntaxe permet une plus grande souplesse au niveau de la manipulation de chaînes de caractères. Par exemple, il n'est plus nécessaire d'encadrer ses variables avec des doubles-guillemets lorsque l'on souhaite faire une comparaison de chaînes.

En outre, les versions 3 et supérieures de bash proposent l'opérateur =~ qui permet de faire des tests sur expressions régulières, en utilisant non pas la commande test, mais les doubles-crochets.

Test sur les chaînes de caractères

Exemple

```
prompt> var="1A"
prompt> [[ $var =~ ^[0-9]*$ ]] # renvoie faux
prompt> [[ $var =~ ^[0-9] ]] # renvoie vrai
prompt> [[ $var =~ [A-Z]{1} ]] # renvoie vrai
prompt> [[ $var =~ ^[0-1]{1}[A-Z]{1}$ ]] # renvoie vrai
```

Test sur les longueurs de chaînes de caractères

Syntaxe:

```
test "fichier1" option "fichier2"
```

Option	Signification
-z	chaîne de longueur nulle
-n	chaîne de longueur non nulle

Tests sur les valeurs numériques

Syntaxe:

```
test nb1 option nb2
```

Option	Signification
-eq	nb1 égal à nb2 (*equal*)
-ne	nb1 différent de nb2 (*non equal*)
-lt	nb1 inférieur à nb2 (*less than*)
-le	nb1 inférieur ou égal à nb2 (*less or equal*)
-gt	nb1 supérieur à nb2 (*greater than*)
-ge	nb1 supérieur ou égal à nb2 (*greater or equal*)

⚠ Utiliser « = » ou « *!=* » à la place de « *-eq* » ou « *-ne* » peut produire des résultats erronés.

Tests sur les valeurs numériques

Syntaxe:

```
test nb1 option nb2
```

Exemple

```
test "5" = "05" # Renvoie "faux" (ce qui est mathématiquement
incorrect)
test "5" -eq "05" # Renvoie "vrai" (correct)
```

Connecteurs d'expression

- **Les connecteurs permettent de composer des expressions plus complexes.**

Option	Signification
-a	« ET » logique
-o	« OU » logique
!	« NOT » logique
(...)	Groupement d'expressions (doit être protégé de backslashes pour ne pas que le shell l'interprète comme une demande de création de sous-shell)

Connecteurs d'expression

❑ Exemple

◘ Mais cette commande n'affiche rien à l'écran.

❖ Vérifie si l'année courante est bissextile (divisible par 4, mais pas par 100 ; ou divisible par 400)

Sélectionnez

```
y=`date '+%Y'` # Récupère l'année courante dans la variable "y"
test \( `expr $y % 4` -eq 0 ?a `expr $y % 100` -ne 0 \) ?o `expr $y %
400` -eq 0
```

Remarque

Il n'y a aucune optimisation faite par la commande « **test** ». Ainsi, lors d'un « et », même si la première partie du « et » est fausse, la seconde partie sera inutilement vérifiée. De même, lors d'un « ou », même si la première partie est vraie, la seconde partie sera tout aussi inutilement vérifiée.

Les commandes « true » et « false »

- **Les commandes « true » et « false » n'ont d'autre but que de renvoyer un état respectivement à « vrai » ou « faux ».**

Syntaxe

```
true
false
```

Exemple

```
Prompt> true # La commande renvoie la valeur "vrai"
Prompt> echo $?
0
Prompt> false # La commande renvoie la valeur "faux"
Prompt> echo $?
1
```

La commande « read »

- **Permet de lire l'entrée standard et de remplir la (ou les) variables demandées avec la saisie du clavier.**
  - **Cette commande renvoie un état « vrai » quand elle a lu l'entrée standard ou « faux » quand l'entrée standard est vide.**
    - Il faut donc, pour connaître le résultat d'un test, vérifier le contenu de la variable « $? ».

Syntaxe

```
read [var1 var2 ?]
```

Les structures de contrôles

- **Comme tout langage évolué, le Shell permet des structures de contrôles.**
- **Ces structures sont :**
  - **l'alternative simple (« *&&…* », « ||… ») ;**
  - **l'alternative complexe (« *if*… ») ;**
  - **le branchement sur cas multiples (« *case*… ») ;**
  - **la boucle (« *while*… », « *until*… », « *for*… »)..**

L'alternative simple

Syntaxe

```
cde1 && cde2
cde1 || cde2
```

- **1ere syntaxe correspond à un « commande1 ET commande2 »**
  - se traduit par « exécuter la commande n° 1 ET (sous-entendu « si celle-ci est « vrai » donc s'est exécutée entièrement ») exécuter la commande n° 2 »..
- **2eme syntaxe correspond à un « commande1 OU commande2 »**
  - se traduit par « exécuter la commande n° 1 OU (sous-entendu « si celle-ci est « faux » donc ne s' est pas exécutée entièrement ») exécuter la commande n° 2 ».

L'alternative simple

- **Exemple:** Écrire un script affichant si on lui a passé zéro, un ou plusieurs paramètres. Ensuite il devra afficher les paramètres reçus.

```
#!/bin/sh
# Script affichant si on lui passe zéro, un ou plusieurs
paramètres
# Ensuite il affiche ces paramètres

# Test sur aucun paramètre
test $# -eq 0 && echo "$0 n'a reçu aucun paramètre"

# Test sur un paramètre
test $# -eq 1 && echo "$0 a reçu un paramètre qui est $1"

# Test sur plusieurs paramètres
test $# -gt 1 && echo "$0 a reçu $# paramètres qui sont $*"
```

L'alternative simple

Remarque

Il est possible d'enchaîner les alternatives par la syntaxe « **cde1 && cde2 || cde3** ». L'inconvénient de cette syntaxe est qu'on ne peut placer qu'une commande en exécution de l'alternative, ou alors, si on désire placer plusieurs commandes, on est obligé de les grouper avec des parenthèses.

L'alternative complexe

Syntaxe

```
if liste de commandes
then
    commande1
    [ commande2 ?]
[ else
    commande3
    [ commande4 ?] ]
fi
```

Exemple avec des « if imbriqués »

```
#!/bin/sh
echo "Entrez un nombre"
read nb
if test $nb -eq 0 # if n°1
then
    echo "C'était zéro"
else
    if test $nb -eq 1 # if n°2
    then
        echo "C'était un"
    else
        echo "Autre chose"
    fi # fi n°2
fi # fi n°1
```

Exemple avec des « elif »

```
#!/bin/sh
echo "Entrez un nombre"
read nb
if test $nb -eq 0 # if n°1
then
    echo "C'était zéro"
elif test $nb -eq 1 # Sinon si
then
    echo "C'était un"
else
    echo "Autre chose"
fi # fi n°1
```

Le branchement à choix multiples

- **La structure « case…esac » évalue la chaîne en fonction des différents choix proposés.**
  - À la première valeur trouvée, les instructions correspondantes sont exécutées.
  - Le double « point-virgule » indique que le bloc correspondant à la valeur testée se termine. Il est donc obligatoire… sauf si ce bloc est le dernier à être évalué.
  - La chaîne et/ou les valeurs de choix peuvent être construites à partir de variables ou de sous-exécutions de commandes.

Syntaxe

```
case chaine in
    val1)
        commande1
        [ commande2 ?]
        ;;
    [val2)
        commande3
        [ commande4 ?]
        ;;]
esac
```

Le branchement à choix multiples

- **les valeurs de choix peuvent utiliser les constructions suivantes :**

Construction	Signification
[x-y]	La valeur correspond à tout caractère compris entre « *x* » et « *y* »
[xy]	La valeur testée correspond à « *x* » ou « *y* »
xx|yy	La valeur correspond à deux caractères « *xx* » ou « *yy* »
?	La valeur testée correspond à un caractère quelconque
*	La valeur testée correspond à toute chaîne de caractères (cas « autres cas »)

Le branchement à choix multiples

- **Exemple:** Script qui fait saisir un nombre et qui évalue ensuite s'il est pair, impair, compris entre 10 et 100 ou autre chose.

```
#!/bin/sh
# Script de saisie et d'évaluation simple du nombre saisi

# Saisie du nombre
echo "Entrez un nombre"
read nb

# Évaluation du nombre
case $nb in
    0) echo "$nb vaut zéro";;
    1|3|5|7|9) echo "$nb est impair";;
    2|4|6|8) echo "$nb est pair";;
    [1-9][0-9]) echo "$nb est supérieur ou égal à 10 et
inférieur à 100";;
    *) echo "$nb est un nombre trop grand pour être
évalué"
esac
```

La boucle sur conditions

- **La boucle « while do…done »**
  - Exécute une séquence de commandes tant que la dernière commande de la « liste de commandes » est « vrai » (statut égal à zéro).
- **La boucle « until do…done »**
  - exécute une séquence de commandes tant que la dernière commande de la « liste de commandes » est « faux » (statut différent de zéro).

Syntaxe

```
while liste de commandes
do
    commande1
    [ commande2 ?]
done
until liste de commandes
do
    commande1
    [ commande2 ?]
done
```

La boucle sur conditions

- **Exemple:** Script qui affiche tous les fichiers du répertoire courant et qui, pour chaque fichier, indique si c'est un fichier de type « répertoire », de type « ordinaire » ou d'un autre type.

```
#!/bin/sh
# Script d'affichage d'informations sur les fichiers du répertoire courant

# La commande "read" lit l'entrée standard. Mais cette entrée peut être redirigée d'un pipe
# De plus, "read" renvoie "vrai" quand elle a lu et "faux" quand il n'y a plus rien à lire
# On peut donc programmer une boucle de lecture pour traiter un flot d'informations
ls | while read fic # Tant que le "read" peut lire des infos provenant du "ls"
do
    # Évaluation du fichier traité
    if test -d "$fic"
    then
        echo "$fic est un répertoire"
    elif test -f "$fic"
    then
        echo "$fic est un fichier ordinaire"
    else
        echo "$fic est un fichier spécial ou lien symbolique ou pipe ou socket"
    fi
done
```

La boucle sur liste de valeurs

- **La boucle « for… do…done »**
  - va boucler autant de fois qu'il existe de valeurs dans la liste.
  - À chaque tour, la variable « $var » prendra séquentiellement comme contenu la valeur suivante de la liste.
  - Les valeurs de la liste peuvent être obtenues de différentes façons (variables, sous-exécutions…).
  - La syntaxe « in valeur1 … » est optionnelle. Dans le cas où elle est omise, les valeurs sont prises dans la variable « $* » contenant les arguments passés au programme.

Syntaxe

```
for var in valeur1 [valeur2 ?]
do
    commande1
    [ commande2 ?]
done
```

La boucle sur liste de valeurs

- **La boucle « for… do…done »**
  - Dans le cas où une valeur contient un métacaractère de génération de nom de fichier (« *étoile* », « *point d'interrogation* »…),
    - le Shell examinera alors les fichiers présents dans le répertoire demandé au moment de l'exécution du script et remplacera le métacaractère par le ou les fichiers dont le nom correspond au métacaractère.

Syntaxe

```
for var in valeur1 [valeur2 ?]
do
    commande1
    [ commande2 ?]
done
```

La boucle sur liste de valeurs

- **Exemple:** Même script que dans l'exemple précédent, qui affiche tous les fichiers du répertoire courant et qui, pour chaque fichier, indique si c'est un fichier de type « répertoire », de type « ordinaire » ou d'un autre type, mais en utilisant une boucle « for ».

La boucle sur liste de valeurs

```
#!/bin/sh
# Script d'affichage d'informations sur les fichiers du
répertoire courant
for fic in `ls` # Boucle sur chaque fichier affiché par la
commande "ls"
do
    # Évaluation du fichier traité
    if test -d "$fic"
    then
        echo "$fic est un répertoire"
    elif test -f "$fic"
    then
        echo "$fic est un fichier ordinaire"
    else
        echo "$fic est un fichier spécial ou lien
symbolique ou pipe ou socket"
    fi
done
```

La boucle sur liste de valeurs

Remarques

Ce script présente un léger « bogue » dû à l'emploi de la boucle « *for* ». En effet, le « *for* » utilise l'espace pour séparer ses éléments les uns des autres. Il s'ensuit que si un fichier possède un espace dans son nom, le « for » séparera ce nom en deux parties qu'il traitera dans deux itérations distinctes et la variable « fic » prendra alors comme valeurs successives les deux parties du nom.

Ce bogue n'existe pas avec l'emploi de la structure « *ls | while read fic… »*, car le « *read* » lit la valeur jusqu'à la « fin de ligne ».

Par ailleurs, dans le cas de la commande ls ou encore du parcours de fichiers dans un script, il est préférable de privilégier l'utilisation le métacaractère « * » (encore appelé « wildcard »).

La boucle sur liste de valeurs

- **Exemple:** Reprenons l'exemple précédent :

```
#!/bin/sh
# Script d'affichage d'informations sur les fichiers du
répertoire courant
for fic in *
do
    # Évaluation du fichier traité
    if [ -d "$fic" ]
    then
        echo "$fic est un répertoire"
    elif [ -f "$fic" ]
    then
        echo "$fic est un fichier ordinaire"
    else
        echo "$fic est un fichier spécial ou lien
symbolique ou pipe ou socket"
    fi
done
```

La boucle sur liste de valeurs

Il est aussi possible de parcourir des valeurs itératives comme dans la plupart des langages de programmation à l'aide de la boucle for. Pour cela, on peut utiliser la commande seq comme suit :

```
# parcours et affichage des valeurs allant de 0 à 10
for i in `seq 0 10`
do
    echo $i
done
```

La boucle sur liste de valeurs

Dans les langages shell dits « évolués », il est également permis d'utiliser ces types de syntaxe dont on retrouve des équivalences dans d'autres langages de programmation courants :

```
# parcours et affichage des valeurs allant de 0 à 10
for (( i=0 ; i <= 10 ; i++ ))
do
    echo $i
done

# Autre syntaxe possible
for i in {0..10}
do
    echo $i
done
```

Interruption d'une ou plusieurs boucles

- **L'instruction « break [n] »**
  - va faire sortir le programme de la boucle numéro « *n* » (« *1* » par défaut).
  - L'instruction passera directement après le « done » correspondant à cette boucle.

Syntaxe

```
break [n]
continue [n]
```

- **L'instruction « continue [n] »**
  - va faire repasser le programme à l'itération suivante de la boucle numéro « n » (« 1 » par défaut).
  - Dans le cas d'une boucle « while » ou « until », le programme repassera à l'évaluation de la condition.
  - Dans le cas d'une boucle « for », le programme passera à la valeur suivante.

Interruption d'une ou plusieurs boucles

- **L'instruction « break [n] »**
  - La numérotation des boucles s'effectue à partir de la boucle la plus proche de l'ins truction « *break* » ou « *continue* », qu'on numérote « 1 ».
  - Chaque boucle englobant la précédente se voit affecter un numéro incrémental (2, 3…).
  - Le programmeur peut choisir de sauter directement sur la boucle numérotée « n » en mettant la valeur « *n* » derrière l'instruction « *break* » ou « *continue* ».

Syntaxe

```
break [n]
continue [n]
```

Interruption d'une ou plusieurs boucles

Remarques

- L'utilisation de ces instructions est contraire à la philosophie de la « *programmation structurée* ». Il incombe donc à chaque programmeur de toujours réfléchir au bien-fondé de leurs mises en application.
- Contrairement aux croyances populaires, la structure « **if... fi** » n'est pas une boucle.

Exemple

Interruption d'une ou plusieurs boucles

- **Exemple:** Script qui fait saisir un nom et un âge. Mais il contrôle que l'âge soit celui d'un majeur et soit valide (entre 18 et 200 ans). Ensuite, il inscrit ces informations dans un fichier. La saisie s'arrête sur un nom vide où un âge à « 0 ».

Interruption d'une ou plusieurs boucles

```
#!/bin/sh
# Script de saisie; de contrôle et d'enregistrement d'un
nom et d'un âge

while true # Boucle infinie
do
    # Saisie du nom et sortie sur nom vide
    echo "Entrez un nom : "; read nom
    test -z "$nom" && break # Sortie de la boucle infinie
si nom vide

    # Saisie et contrôle de l'âge
    while true # Saisie en boucle infinie
    do
        echo "Entrez un âge : "; read age
        test $age -eq 0 && break 2 # Sortie de la boucle
infinie si age = 0
        test $age -ge 18 -a $age -lt 200 && break # Sortie
de la boucle de saisie si age correct
    done

    # Enregistrement des informations dans un fichier
"infos.dat"
    echo "Nom: $nom; Age: $age" >>infos.dat
done
```

Interruption d'un programme

- **L'instruction « exit [n] » met immédiatement fin au Shell dans lequel cette instruction est exécutée.»**
  - Le paramètre « n » facultatif (qui vaut « 0 » par défaut) ne peut pas dépasser « 255 ».
  - Ce paramètre sera récupéré dans la variable « $? » du processus ayant appelé ce script (processus père).
  - Cette instruction « exit » peut donc rendre un script « vrai » ou « faux » selon les conventions du Shell.

Remarque

Même sans instruction « **exit** », un script Shell renvoie toujours au processus père un état qui est la valeur de la variable « *$?* » lorsque le script se termine (état de la dernière commande du script).

Le générateur de menus en boucle

- **La structure « select… do… done »**
  - proposera à l'utilisateur un menu prénuméroté commençant à « 1 ».
  - Chaque boucle englobant la précédente se voit affecter un numéro i ncrémental (2, 3…).
  - Le programmeur peut choisir de sauter directement sur la boucle nu mérotée « n » en mettant la valeur « *n* » derrière l'instruction « *brea k* » ou « *continue* ».

Syntaxe

```
select var in chaîne1 [chaîne2 ?]
do
    commande1
    [ commande2 ?]
done
```

Q & R

Contact: mlec.academia@gmail.com