LA PROGRAMMATION "SHELL"

Licence de ce document

Permission est accordée de copier, distribuer et/ou modifier ce document selon les termes de la "Licence de Documentation Libre GNU" (GNU Free Documentation License), version 1.1 ou toute version ultérieure publiée par la Free Software Foundation.
En particulier le verbe "modifier" autorise tout lecteur éventuel à apporter au document ses propres contributions ou à corriger d'éventuelles erreurs et lui permet d'ajouter son propre copyright dans la page "Registre des éditions" mais lui interdit d'enlever les copyrights déjà présents et lui interdit de modifier les termes de cette licence.
Ce document ne peut être cédé, déposé ou distribué d'une autre manière que l'autorise la "Licence de Documentation Libre GNU" et toute tentative de ce type annule automatiquement les droits d'utiliser ce document sous cette licence. Toutefois, des tiers ayant reçu des copies de ce document ont le droit d'utiliser ces copies et continueront à bénéficier de ce droit tant qu'ils respecteront pleinement les conditions de la licence.

La version actuelle de ce document est la *2.0* et sa version la plus récente est disponible en téléchargement à l'adresse http://fr.lang.free.fr

Copyright © 2007 Frédéric Lang (fr.lang@free.fr)

Registre des éditions

Version	Date	Description des modifications	Auteur des modifications
1.0	30 juillet 2004	Mise en place du registre des éditions Corrections mineures (syntaxe, etc)	© Frédéric Lang (fr.lang@free.fr)
1.1	26 août 2004	Relecture – Corrections mineures	© Frédéric Lang (fr.lang@free.fr)
1.2	24 novembre 2004	Rajout du test de la numéricité d'une variable en "réel" Vérification et corrections majeures de tous les exercices	© Frédéric Lang (fr.lang@free.fr)
1.3	24 février 2005	Rectification sur le paragraphe des fonctions Corrections mineures	© Frédéric Lang (fr.lang@free.fr)
1.4	15 septembre 2005	Rajout de la variable "REPLY" dans le "select" et dans la liste des variables internes Corrections mineures	© Frédéric Lang (fr.lang@free.fr)
2.0	26 juin 2007	Rajout d'un paragraphe dans l'introduction Prise en compte de nouveaux shells Réécriture complète du chapitre des fonctions Correction sur les substitutions de variables Rajout de la variable "REPLY" dans le "read" Correction sur la commande "test" Création du chapitre sur les outils avancés Révision de l'instruction "set" Rajout de l'instruction "source" Rajout de l'instruction "trap" Rajout de l'instruction "let" Rajout des instructions "typeset" et "declare" Rajout des instructions "locale"" Révision de l'instruction "getopts" Correction sur la commande "expr" Correction sur la commande "exec" Rajout de la commande "seq" Rajout de la commande "xargs" Rajout de la commande "getopt" Révision majeure des exemples "division" et "résolution polynômes" Rajout de l'exemple "tours de Hanoï" Remplacement de l'adresse email "frederic-lang@fr.fm" par "fr.lang@free.fr"	© Frédéric Lang (fr.lang@free.fr)

SOMMAIRE

I) INTRODUCTION 7
1) POURQUOI PROGRAMMER EN SHELL ? 8
2) PRESENTATION 8
3) IMPERATIFS 10
4) CONVENTIONS DU SUPPORT DE COURS 10

II) ELEMENTS DE BASE DU LANGAGE 11
1) L'AFFICHAGE 11
2) LES COMMENTAIRES 11
3) LE DEBOGUEUR 11
4) QUI EXECUTE ? 12

III) LES VARIABLES 13
1) L'AFFECTATION – L'ACCES 13
a) Les variables simples 13
b) Les tableaux (shells évolués) 14
c) Remarque 14
2) LA SUBSTITUTION 15
a) Tous types de Shell 15
b) Uniquement en Korn Shell et Bourne Again Shell (et shells descendants) 16
c) Uniquement en Bourne Again Shell (et shells descendants) 16
d) Remarque 16
3) LA SAISIE EN INTERACTIF 17
4) LA PROTECTION 17
5) LA SUPPRESSION 18
6) LA VISIBILITE 18
7) LE TYPAGE ("KORN SHELL" ET "BOURNE AGAIN SHELL" ET SHELLS DESCENDANTS) 20
8) LES VARIABLES PREDEFINIES 21

IV) LA "SOUS-EXECUTION" 22

V) LES PARAMETRES 24
1) RECUPERATION DES PARAMETRES 24
2) DECALAGE DES PARAMETRES 25
3) RE AFFECTATION VOLONTAIRE DES PARAMETRES 26
4) LE SEPARATEUR DE CHAMPS INTERNES 26

VI) NEUTRALISATION DES META CARACTERES 27
1) RAPPEL SUR LES META CARACTERES 27
2) LE "BACKSLASHE" 28
3) L'APOSTROPHE OU "QUOTTE SIMPLE" 28
4) LE DOUBLE GUILLEMET 28

VII) LES CONTROLES BOOLEENS 29
1) INTRODUCTION 29
2) LA COMMANDE "*TEST*" 30
a) Test simple sur les fichiers 30
b) D'autres test simple sur les fichiers ("Korn Shell" et "Bourne Again Shell" et shells descendants) 30
c) Test complexe sur les fichiers ("Korn Shell" et en "Bourne Again Shell" et shells descendants) 31
d) Test sur les longueurs de chaînes de caractères 31
e) Test sur les chaînes de caractères 31
f) Test sur les numériques 32
g) Connecteurs d'expression 32
3) LES COMMANDES "*TRUE*" ET "*FALSE*" 33
4) LA COMMANDE "*READ*" 33

VIII) LES STRUCTURES DE CONTROLES 34
1) INTRODUCTION 34

2) L'ALTERNATIVE SIMPLE 34
3) L'ALTERNATIVE COMPLEXE 35
4) LE BRANCHEMENT A CHOIX MULTIPLE 36
5) LA BOUCLE SUR CONDITION 37
6) LA BOUCLE SUR LISTE DE VALEURS 38
7) INTERRUPTION D'UNE OU PLUSIEURS BOUCLES 39
8) INTERRUPTION D'UN PROGRAMME 40
9) LE GENERATEUR DE MENUS EN BOUCLE (KORN SHELL ET BOURNE AGAIN SHELL ET SHELLS DESCENDANTS) 40

IX) LES FONCTIONS 41
1) INTRODUCTION 41
2) PASSAGE DE VALEURS 42
3) RETOUR DE FONCTION 43
4) RENVOI D'UNE VALEUR PAR UNE FONCTION 44
5) PORTEE DES VARIABLES 45
6) IMBRICATION DE FONCTIONS 46
7) LA TRACE DE L'APPELANT 48

X) LES INSTRUCTIONS AVANCEES 49
1) L'INCLUSION DE SCRIPT DANS UN SCRIPT 49
2) LA PROTECTION CONTRE LES SIGNAUX 50
3) TRANSFORMER UNE EXPRESSION EN ORDRE "SHELL" 51
4) ARITHMETIQUE SUR LES VARIABLES ("KORN SHELL" ET "BOURNE AGAIN SHELL" ET SHELLS DESCENDANTS) 51
5) CREER DE NOUVEAUX CANAUX D'ENTREES/SORTIES 52
6) GERER LES OPTIONS 53

XI) LES COMPLEMENTS 55
1) EVALUER DES EXPRESSIONS REGULIERES AVEC LA COMMANDE "EXPR" 55
a) Arithmétique 55
b) Comparaison 56
c) Travail sur chaînes de caractères 56
2) RECHERCHER DES CHAINES AVEC LA COMMANDE "GREP" 57
3) DECOUPER DES LIGNES AVEC LA COMMANDE "CUT" 58
4) TRIER LES INFORMATIONS AVEC LA COMMANDE "SORT" 58
5) FILTRER LES INFORMATIONS AVEC LA COMMANDE "SED" 59
6) TRANSFORMER LES INFORMATIONS AVEC LA COMMANDE "TR" 59
7) COMPTER LES OCTETS AVEC LA COMMANDE "WC" 60
8) AFFICHER UNE SEQUENCE DE NOMBRES AVEC LA COMMANDE "SEQ" 60
9) AFFICHER DES DONNEES FORMATEES AVEC LA COMMANDE "PRINTF" 60
10) DECOUPER LES NOMS AVEC LES COMMANDES "BASENAME" ET "DIRNAME" 61
11) FILTRER LES ARGUMENTS AVEC LA COMMANDE "XARGS" 61
12) MATHEMATIQUES EN VIRGULE FLOTTANTE AVEC LA COMMANDE "BC" 62
13) GERER LES OPTIONS AVEC LA COMMANDE "GETOPT" 62
14) GERER SON AFFICHAGE A L'ECRAN AVEC LES CODES "ESCAPE" 64

XII) EXEMPLES DIVERS 66
1) AFFICHER UNE PHRASE SANS QUE LE CURSEUR PASSE A LA LIGNE SUIVANTE 66
2) VERIFIER L'EXISTENCE D'UN FICHIER QUEL QUE SOIT SON TYPE 66
3) VERIFIER LA NUMERICITE D'UNE VARIABLE EN TERME DE "NOMBRE ENTIER" 66
4) VERIFIER LA NUMERICITE D'UNE VARIABLE EN TERME DE "NOMBRE REEL" 67
5) NOM DE BASE DU REPERTOIRE COURANT 67
6) VERIFICATION DE L'AUTORISATION D'ACCES D'UN UTILISATEUR QUELCONQUE 68
7) MEMBRES D'UN GROUPE 69
8) SERVEURS GERES PAR "INETD" 70
9) RECHERCHE D'UNE COMMANDE 71
10) ARBORESCENCE D'UN REPERTOIRE 72
11) FACTORIELLE D'UN NOMBRE 74
12) PGCD DE DEUX NOMBRES 76
13) DIVISION EN NOMBRES REELS 78
14) RESOLUTION DE POLYNOME DU SECOND DEGRE : $Ax^2 + Bx + C = 0$ 83
15) TOURS DE HANOÏ 86

INDEX ____ **88**

I) INTRODUCTION

Le Shell est un "**INTERPRETEUR DE COMMANDE**". Il ne fait pas partie du système d'exploitation UNIX; et c'est la raison pour laquelle il porte ce nom "coquille", qui indique son état détaché du "noyau" Unix. Son rôle est d'analyser la commande tapée afin de faire réagir le système pour qu'il réponde aux besoins de l'utilisateur. C'est le premier langage de commandes développé sur Unix par Steve Bourne.
Une comparaison rapide pourrait l'apparenter au DOS (Disc Operating System) développé par Microsoft, mais cette comparaison n'est là que pour illustrer le rôle du Shell par rapport à Unix.

De par sa qualité de "programme externe", il n'est pas unique. En effet, rien n'empêche n'importe quel programmeur de programmer une boucle qui attend une chaîne au clavier, analyse cette chaîne et appelle ensuite le système pour exécuter l'ordre demandé. C'est la raison pour laquelle il existe plusieurs shells. Nous trouvons entre autres (liste non-exhaustive) :

- ✓ le Bourne Shell ("/bin/sh")
- ✓ le Korn Shell ("/bin/ksh")
- ✓ le c Shell ("/bin/csh") pour les utilisateurs préférant un langage apparenté au "C"
- ✓ le Job Shell ("/bin/jsh")
- ✓ le Shell réseau ("/bin/rsh")
- ✓ le Bourne Again Shell ("/bin/bash") qui a repris le Bourne Shell mais qui l'a agrémenté de nouvelles fonctionnalités (rappel de commandes, complétion automatique, etc.)
- ✓ le c Shell amélioré ("/bin/tcsh") améliorant le c Shell tout en lui restant compatible
- ✓ l'Almquist Shell ("/bin/ash") améliorant le Bourne Shell tout en étant plus compact que le Bourne Again Shell
- ✓ le z Shell ("/bin/zsh")
- ✓ et d'autres encore à venir…

C'est un langage de commandes mais aussi un langage de programmation. Il permet donc :

- ✓ l'utilisation de variables
- ✓ la mise en séquence de commandes
- ✓ l'exécution conditionnelle de commandes
- ✓ la répétition de commandes

Un programme shell appelé aussi "script" est un outil facile à utiliser pour construire des applications en regroupant des appels systèmes, outils, utilitaires et programmes compilés. Concrètement, le répertoire entier des commandes Unix, des utilitaires et des outils est disponible à partir d'un script shell. Les scripts shell conviennent parfaitement pour des tâches d'administration du système et pour d'autres routines répétitives ne réclamant pas les particularités d'un langage de programmation structuré.

1) Pourquoi programmer en shell ?

Aucun langage de programmation n'est parfait. Il n'existe même pas un langage meilleur que d'autres ; il n'y a que des langages en adéquation ou peu conseillés pour des buts particuliers. (Herbert Mayer).

Une connaissance fonctionnelle de la programmation shell est essentielle à quiconque souhaite devenir efficace en administration système car toute l'administration dans Unix est écrite en shell. Une compréhension détaillée des scripts d'administration est importante pour analyser le comportement du système, voire le modifier. De plus, la seule façon pour vraiment apprendre la programmation des scripts est d'écrire des scritpts.

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 multi dimensionels, listes chaînées, arbres, etc.)
- ✓ 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 qui devront générer ou utiliser une interface graphique utilisateur (G.U.I.)
- ✓ Pour des applications propriétaires (un script est forcément lisible par celui qui l'utilise)

Pour toutes ces situations, Unix offre une gamme de langages de scripts plus puissants comme le Perl, Tcl, Python, Ruby ; voire des langages compilés de haut niveau comme le C et le C++.

2) Présentation

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.

Exemple :

```
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
```

Le second, est dit en "script" ; appelé aussi "batch" ou "source Shell". L'utilisateur crée un fichier texte par l'éditeur de son choix (ex : "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" (*cf. Gestion des fichiers*). Il peut ensuite lancer l'exécution de ce fichier comme n'importe quelle autre commande Unix (attention donc au contenu de la variable "PATH" qui indique où aller chercher une commande lorsqu'on la lance sans préciser où elle se trouve).
Toutes les commandes inscrites dans le fichier texte seront exécutées séquentiellement.

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
```

Remarque :
L'adjonction du droit "x" n'est pas obligatoire, mais l'utilisateur devra alors demander spécifiquement à un Shell quelconque d'interpréter le "script".

Exemple : L'utilisateur demande d'interpréter le script par l'intermédiaire du "*Bourne Shell*"

```
Prompt> sh prog
Tue Jan 16 17:26:50 NFT 2001
/tmp
oui
```

Exemple : L'utilisateur demande d'interpréter le script par l'intermédiaire du "*Korn Shell*"

```
Prompt> ksh prog
Tue Jan 16 17:26:50 NFT 2001
/tmp
oui
```

3) Impératifs

Comme il l'a été dit, le Shell est un "interpréteur". C'est à dire que chaque ligne est analysée, vérifiée et exécutée. Afin de ne pas trop limiter la rapidité d'exécution, il y a très peu de règles d'analyse. Cela implique un **grande rigidité d'écriture** de la part du programmeur. Une majuscule n'est pas une minuscule ; et surtout, **deux éléments distincts sont toujours séparés par un espace**… à une exception près qui sera vue plus tard.

Enfin, le premier mot de chaque ligne, si la ligne n'est pas mise en commentaire, doit être une instruction Shell "correcte". On appelle "instruction" soit

- ✓ l'appel d'une commande "Unix" (ex : date)
- ✓ l'appel d'un autre script "Shell" (ex : ./prog)
- ✓ une commande interne du langage (ex : cd…)

Exemple :

```
Prompt> echoBonjour
sh: echoBonjour: not found
Prompt> echo Bonjour
Bonjour
```

4) Conventions du support de cours

Il est difficile d'écrire un support de cours en essayant de faire ressortir des points importants par l'utilisation de caractères spécifiques (guillemets, parenthèses, etc.) sur un langage utilisant certains caractères comme instructions spécifiques (guillemets, parenthèses, etc.). De plus, le lecteur ne sait pas forcément distinguer les éléments provenant de l'ordinateur des éléments qu'il doit taper au clavier. C'est pourquoi, il est nécessaire de définir des conventions

✓ La chaîne "***Prompt***>" ou "***Prompt2***>" présente dans les exemples de ce cours est une "invite" ; appelée aussi "prompt". C'est une chaîne affichée par l'interpréteur dans lequel travaille l'utilisateur afin de lui indiquer qu'il attend une instruction. Cette chaîne n'est pas forcément la même pour des utilisateurs différents. Dans ce support, sa présence signifie que l'exemple proposé peut être tapé directement en ligne, sans obligatoirement passer par un fichier script.

✓ La chaîne "***#!/bin/sh***" présente dans les exemples de ce cours est expliquée plus tard. Dans ce support, sa présence signifie qu'il vaut mieux écrire l'exemple proposé dans un fichier "script" afin de pouvoir exécuter l'exemple plusieurs fois pour mieux analyser son fonctionnement.

✓ Les syntaxes présentes en début de paragraphe donnent la syntaxe exacte d'une commande. Elles sont donc présentées sans prompt ni ligne "#!/bin/sh".

✓ Des éléments mis entre crochets "***[elem]***" signifient qu'ils sont facultatifs. L'utilisateur peut les utiliser ou pas, mais ne doit en aucun cas mettre de crochets dans sa frappe ou son programme.

✓ Le caractère ***n*** employé dans la syntaxe signifie "nombre entier quelconque positif ou, parfois, nul".

✓ Le caractère "|" employé dans la syntaxe signifie "choix entre l'élément situé à gauche ou à droite du |".

✓ Les points de suspension "**…**" signifient que l'élément placé juste avant eux peut être répété autant de fois que l'on désire.

II) ELEMENTS DE BASE DU LANGAGE

1) L'affichage

L'affichage est la première commande qu'un programmeur débutant pourra être tenté de faire. Cela lui permet en effet de visualiser directement à l'écran le résultat de ses actions.
L'affichage en Shell se fait avec la commande "***echo***".

Exemple :

```
Prompt> echo Hello World
```

2) Les commentaires

Un commentaire sert à améliorer la lisibilité du script. Il est placé en le faisant précéder du caractère ***dièse*** ("***#***"). Tout ce qui suit ce dièse est ignoré jusqu'à la fin de la ligne ; ce qui permet de mettre un commentaire sur une ligne d'instructions. Il ne faut pas oublier alors l'espace séparant la fin de l'instruction et le début du commentaire.

Exemple :

```
#!/bin/sh
# Ce programme affiche la date
date              # Cette ligne est la ligne qui affiche la date
```

3) Le débogueur

Une procédure longue et difficile peut ne pas réussir du premier coup. 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 +…*". On peut ainsi activer le "traqueur" sur une portion précise du programme source.

Exemple :

```
#!/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
```

Remarque : Compte tenu du flot important d'informations produit par ces outils, il peut être avantageux de lui préférer un affichage des variables pouvant causer l'erreur (commande "*echo*").

4) Qui exécute ?

Rappelons qu'il existe plusieurs shells et que chacun d'eux possède des caractéristiques différentes. De plus, chaque utilisateur Unix peut demander à travailler dans le Shell de sa convenance. Il s'ensuit qu'un script écrit par un utilisateur travaillant en Bourne Shell ne fonctionnera pas forcément s'il est exécuté par un utilisateur travaillant en C-Shell.

L'utilisateur peut visualiser le Shell dans lequel il travaille en tapant la commande suivante:

```
Prompt> echo $0
```

Cependant, Unix cherche à assurer la portabilité de ses programmes. Il est donc nécessaire qu'un script écrit par un utilisateur travaillant en Bourne Shell puisse être exécuté par tous les utilisateurs, quels que soient leurs shells de travail.
Pour cela, il convient d'indiquer dans le script quel interpréteur utiliser pour exécuter ledit script. Si cette indication n'existe pas, le système utilisera l'interpréteur de travail affecté à l'utilisateur pour analyser et exécuter le script en question, interpréteur pas forcément compatible avec la syntaxe du script.
Cette indication se porte en début de script avec la ligne "***#!interpréteur***".

Exemple en Bourne Shell :

```
#!/bin/sh       # Ce script sera traité par le programme "/bin/sh" (Bourne Shell)
date            # Demande à l'interpréteur "/bin/sh" de lancer le programme "date"
```

Exemple en Korn Shell :

```
#!/bin/ksh      # Ce script sera traité par le programme "/bin/ksh" (Korn Shell)
date            # Demande à l'interpréteur "/bin/ksh" de lancer le programme "date"
```

Ici, cas particulier, le caractère "*#*" (*dièse*) de la première ligne combiné au caractère "*!*" (*point d'exclamation*) n'est plus pris comme un commentaire mais comme une instruction indiquant quel programme a la charge d'analyser le script.

Remarque:
Ce cours proposera globalement une syntaxe Bourne Shell avec quelques éléments spécifiques aux shells de même famille dits "plus évolués" (Korn Shell ou Bourne Again Shell) qui seront indiqués au moment où cela sera nécessaire. Le Bourne Shell étant intégralement supporté par le Korn Shell et le Bourne Again Shell (et leurs descendants) ; cela permet une compatibilité ascendante (un script en Bourne Shell fonctionnera toujours en environnements Korn Shell ou Bourne Again Shell ou shells descendants).

Autre exemple (ne fonctionne qu'une seule fois) :

```
#!/bin/rm       # Ce script sera traité par le programme "/bin/rm" (effacement)
# Le script s'efface donc lui-même - Ici le script est déjà effacé
# Quoi que l'on mette ici, cela ne sera pas exécuté, cela n'existe déjà plus…
```

III) LES VARIABLES

Il n'est pas de langage sans variable. 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 et peut en créer quand il veut, où il 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.
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, etc.). 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

1) L'affectation – L'accès

a) Les 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.

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). A 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 caractère "***$***".
Exemples :

```
Prompt> nom="Pierre"                    # Affectation de "Pierre" dans la variable "nom"
Prompt> objet="voiture"                 # Affectation de "voiture" dans la variable "objet"
Prompt> coul=bleue                      # Affectation de"bleu" 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"
```

b) Les tableaux (shells évolués)

Syntaxe :

```
variable[n]=chaîne

variable=(chaîne1 chaîne2 …)
```

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 programmeur doit les mettre dans sa syntaxe.
L'indice "n" que l'on spécifie entre les crochets doit être impérativement positif ou nul; mais il n'a pas de limite maximale. De plus, rien n'oblige le programmeur à remplir le $n^{ème}$ élément avant d'aller remplir le $(n + 1)^{ème}$.
Il est possible de remplir en une seule instruction les "n" premiers éléments du tableau en utilisant la syntaxe "***variable=(chaîne1 chaîne2 …)***". Dans cette syntaxe où l'utilisation de parenthèses "***( )***" est obligatoire, le texte "*chaîne1*" ira dans la variable "*variable[0]*"; le texte "*chaîne2*" ira dans la variable "*variable[1]*"; etc. Cette syntaxe remplace tout le tableau par les seules chaînes situées entre parenthèses. L'ancien contenu éventuel disparaît alors pour être remplacé par le nouveau.

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 concatènera 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 concaténées. 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]".

Exemple :

```
Prompt> nom[1]="Pierre"                 # Affectation de "Pierre" dans la variable "nom[1]"
Prompt> nom[5]="Paul"                   # Affectation de "Paul" dans la variable "nom[5]"
Prompt> nom="Jacques"                   # Affectation de "Jacques" 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[@]}"
```

c) Remarque

Même si cela est possible de façon limitée (*cf. commande eval*), il est difficile et déconseillé de vouloir manipuler des "variables de variable" (connues dans d'autres langages sous le nom "pointeurs"). Autrement dit, la syntaxe "$$var" qui *pourrait* vouloir dire "contenu du contenu de la variable *var*" signifie en réalité "contenu de la variable *$$*" (sera vue ultérieurement) suivi de la chaîne "*var*".

2) La substitution

On peut utiliser des séquenceurs spéciaux pour modifier la manière dont le Shell va renvoyer le contenu de ou des variables demandées.

a) Tous types de Shell

✓ ${var} : renvoie le contenu de "*$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 "*${prix}F*" (variable "*prix*" suivi du caractère "*F*") et ce que le Shell comprendrait si on écrivait simplement "*$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 message d'erreur (implique donc l'arrêt du script).

✓ ${var:?texte} : renvoie le contenu de la variable "*var*" si celle-ci est définie et non-vide ; sinon affiche le texte "*texte*" (comme "*${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 "*var*" 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.

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

✓ ${var[n]} : renvoie le contenu du n^{ième} 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 méta caractère, 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 méta caractère, 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 méta caractère, 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 méta caractère, 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} : renvoie 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.

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

✓ ${!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 caractère d'une variable porte le n° "0"). Si la variable est un tableau, renvoie alors les "*y*" éléments du tableau "*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 variable 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} : renvoie 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*".

d) Remarque

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*".

3) La saisie en interactif

Syntaxe :

```
read [var1 var2 …]
```

Cette action est nécessaire lorsque le programmeur désire demander une information ponctuelle à celui qui utilise le programme. A l'exécution de la commande, le programme attendra du fichier standard d'entrée (*cf. Gestion des processus*) 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", etc.)*. 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.
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).

Syntaxe :

```
read –a tableau                # Korn Shell et 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 (*cf. Gestion des processus*) dans les éléments d'un tableau. Le premier mot ira dans le tableau d'indice "*0*"; le second dans le tableau d'indice "*1*"; etc. 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.

4) La protection

Syntaxe :

```
readonly var1 [var2 …]

readonly
```

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

5) La suppression

Syntaxe :

```
unset var1 [var2 …]
```

Cette instruction supprime la variable sur laquelle elle est appliquée à condition que cette dernière n'ait pas été protégée par l'instruction "***readonly***".
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*".

6) La visibilité

Syntaxe :

```
export var1 [var2 …]

export
```

Lorsque un script Shell est lancé depuis l'environnement d'un utilisateur, ce script commence son exécution (*cf. Gestion des processus*) avec une zone mémoire vierge qui lui est propre. Il ne connaît donc, par défaut, aucune des variables de l'environnement qui lui a donné naissance (environnement "père").
Pour qu'un processus "père" puisse faire connaître une variable à un processus "fils", il doit l'exporter avec la commande "***export var***". Ceci fait, la variable exportée depuis un environnement particulier sera connue de tous les processus "fils" ; et de tous les processus "fils" des "fils", etc.
Cependant, modifier le contenu d'une variable dans un processus quelconque ne reporte pas cette modification dans les environnements supérieurs. Dans la même optique, il n'y a aucun moyen simple pour renvoyer une variable quelconque d'un processus vers un processus parent.
Employée seule, la commande "***export***" donne la liste de toutes les variables qui ont été exportées.

Exemple d'un script "prog" affichant et modifiant une variable qu'il n'a pas créée :

```
#!/bin/sh
echo "Contenu de var : [$var]"      # Affichage de la variable "var"
var=Salut                           # Modification de la variable dans le script pour voir si elle remonte
echo "Contenu de var : [$var]"      # De nouveau affichage de la variable "var"
```

Exemple 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

Action : Affichage de "var"

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

Résultat : Malgré la modification faite dans le script, "var" n'a pas changé

Exemple avec exportation :

Action : Affectation et exportation de "var"

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

Action : Affichage de "var"

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

Résultat : "var" est bien créée

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

7) Le typage ("Korn Shell" et "Bourne Again Shell" et shells descendants)

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 correspond 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*")

A noter : L'instruction "***declare***" accessible uniquement en Bourne Again Shell (et autres descendants) est un synonyme de l'instruction "***typeset***".

8) Les variables prédéfinies

Un utilisateur possède lors de sa connexion plusieurs variables automatiquement définies par le système. Certaines sont modifiables, certaines ne le sont pas, mais toutes sont utiles.
Quelques variables prises parmi les plus courantes…

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***", etc.)
$RANDOM	Nombre aléatoire entre 0 et 32767 (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

IV) LA "SOUS-EXECUTION"

La "sous-exécution" de commande est un des mécanismes les plus importants en Shell. Il permet de remplacer, lors de l'exécution du programme, une partie du script par **le texte qu'une commande affiche normalement à l'écran**.

Ce mécanisme s'obtient en encadrant la commande à remplacer par des "***accents grave***" (caractère "`" obtenu en pressant la touche "*7*" sur un clavier "Azerty" tout en maintenant la touche "*Alt Gr*" enfoncée).

Ce mécanisme peut être utilisé en coordination avec beaucoup d'autres qui seront vus ultérieurement. Mais l'utilisation la plus courante est l'affectation de variables.

Exemples utiles :

```
Prompt> var1=`echo Salut`   # Ce qu'affiche la commande "echo Salut" ira dans "var1"
Prompt> var2=`date`         # Ce qu'affiche la commande "date" ira dans "var2"
Prompt> var3=`ls –l`        # Ce qu'affiche la commande "ls -l" ira dans "var3"
Prompt> var4=`pwd`          # Ce qu'affiche la commande "pwd" ira dans "var4"
```

Exemples inutiles :

```
Prompt> var5=`cd /`         # Ce qu'affiche la commande "cd /" (donc rien) ira dans "var5"
Prompt> var6=pwd            # La chaîne "pwd" ira dans "var6" (oubli des accents grave)
```

Bien entendu, lorsque le programmeur veut exécuter "normalement" une commande (sans avoir besoin de récupérer ce que la commande affiche), il ne met pas d'accent grave.

Exemple :

```
Prompt> date
```

Résultat : Exécution normale de la commande "*date*" (comme avant, depuis le début du cours)

```
Prompt> `date`
```

Résultat : Erreur de syntaxe. La commande "*date*" est sous-exécutée et son résultat est alors interprété par le Shell dans le contexte où la sous-exécution s'est produite. Or le premier mot d'une ligne en Shell doit être une commande Unix et il est peu probable que le premier mot affiché par la commande "date" soit une commande Unix .

```
Prompt> `echo date`
```

Résultat : Inutile. La commande "*echo date*" est sous-exécutée et son résultat (en l'occurrence le mot "date") est alors interprété par le Shell dans le contexte ou la sous-exécution s'est produite. Ici, la chaîne "*date*" produite par la sous-exécution du "echo" correspond à une commande Unix valide et comme son contexte la place en début de ligne, le Shell interprètera cette chaîne comme une instruction et exécutera cette dernière. On aura donc comme résultat final l'exécution de la commande "*date*". Cette remarque sera valable chaque fois que le programmeur sera tenté de sous-exécuter la commande "*echo*".

Remarques :

✓ Il est tout à fait possible d'imbriquer des "niveaux" de sous-exécution. Mais chaque niveau doit être protégé du supérieur par l'emploi de ***backslashes*** ("\"). Et ces backslashes doivent eux-mêmes être protégés du niveau supérieur par des ***backslashes***. Ainsi; une affectation avec 3 niveaux de sous-exécution (par exemple) s'écrira "*var=`cmd1 \`cmd2 \\\`cmd3\\\`\``*. Ce mécanisme de protection sera vu ultérieurement plus en détail (*cf. Neutralisation des méta caractères*) mais il conduit rapidement à une écriture assez lourde qui devient vite illisible si on commence à empiler les sous-niveaux (le plus bas niveau d'une commande à "*n*" imbrications s'écrira avec "$2^{n-1} - 1$" backslashes). Il vaut mieux dans ce cas décomposer les sous-exécutions en plusieurs étapes utilisant des variables.

✓ La sous-exécution peut être aussi obtenue en "Korn Shell" et en "Bourne Again Shell" (et shells descendants) par la syntaxe "*$(commande*)". Cette syntaxe permet une écriture plus simple et plus lisible des sous-niveaux "*$($(commande))*" mais n'est pas compatible avec le "Bourne Shell".

Ce mécanisme permet au Shell de déléguer à d'autres programmes tout ce qu'il ne sait pas faire par lui-même (calculs, travail sur chaines, recherches dans fichier, etc) et de récupérer le résultat dudit programme. Si le programme n'existe pas, il suffit de l'écrire (en n'importe quel langage) et lui faire sortir ses valeurs sur l'écran pour pouvoir récupérer ensuite lesdites valeurs dans le script.
Il permet donc une imbrication totale de tout programme Unix, quel qu'il soit, dans un script shell.

L'excellente compréhension de ce mécanisme est primordiale dans la compréhension de la programmation en Shell.

V) LES PARAMETRES

Un paramètre, appelé aussi "argument", 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 et dont il a évidemment besoin pour travailler. Ces éléments peuvent être nécessaires au programme pour son bon fonctionnement ("*cp fic1 fic2*"), ou facultatifs pour lui demander une variation dans son travail, c'est à dire un travail "optionnel" ("*ls –l*").
Ils constituent généralement un substitut plus avantageux qu'une saisie en "interactif" car le programme n'a alors pas besoin, durant son traitement, de la présence d'un utilisateur qui répondra à ses questions. Celles-ci sont déjà prévues par les "arguments" que le programme connaît, et qu'il utilisera au moment opportun.

Exemple :

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

1) 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. Il peut être considéré comme étant "à part" du reste des arguments.

✓ $1, $2, $3,… $9 : argument placé en première, seconde, troisième… neuvième position derrière le nom du script etc.

✓ $# : nombre d'arguments passés au script

✓ $* : liste de tous les arguments (sauf *$0*) concaténés en une chaîne unique.

✓ $@ : liste de tous les arguments (sauf *$0*) transformés individuellement en chaîne. Visuellement; il n'y pas de différence entre "*$**" et "*$@*".

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 nb d'arguments
echo $*         # Affichage tous 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
```

2) Décalage des paramètres

Syntaxe :

```
shift [n]
```

Comme on peut le remarquer, le programmeur n'a accès de façon individuelle qu'aux variables "*$1*" à "*$9*". Si le nombre de paramètres dépasse neuf, ils sont pris en compte par le script Shell mais le programmeur n'y a pas accès de manière individuelle. Bien entendu, il peut y accéder en passant par la variable "*$**" 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*".

Remarque : La syntaxe "***echo ${10}***" fonctionne en "Korn Shell" et "Bourne Again Shell" (et shells descendants) mais pas en "Bourne Shell".

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. 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*", etc. jusqu'à "*$9*" qui reçoit le contenu du dixième argument. Ainsi un décalage de "*n*" paramètres fait disparaître les "*n*" premiers mais récupère les "*n*" suivants "*$9*".
De plus, les variables "*$#*", "*$**" et "*$@*" sont aussi modifiées pour correspondre à la nouvelle réalité. Seule la variable "*$0*" reste invariante. Ainsi, une simple boucle testant la valeur décroissante de "*$#*" en utilisant l'instruction "*shift*" permet d'accéder individuellement à tous les arguments.

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").

Exemple : Script qui récupère et affiche le 1er, 2°, 12° et 14° paramètres

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

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

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

# Affichage des paramètres demandés (le 12° et le 1 4° ont été amenés en position 1 et 3 par le "shift" )
echo "Les paramètres sont $prem, $sec, $1, $3"
```

3) Ré affectation volontaire des paramètres

Syntaxe :

```
set [--] valeur1 [valeur2 …]
```

L'instruction "***set [--] valeur1 [valeur2 …]***" (qui sert à activer des options du Shell comme le débug) permet aussi de remplir les variables "*$1*", "*$2*", …, "*$9*", au mépris de leur ancien éventuel contenu, avec les valeurs indiquées. Il y a d'abord effacement de toutes les variables puis remplissages avec les valeurs provenant du "*set*".
Bien évidemment les variables "*$#*", "*$**" et "*$@*" sont aussi modifiées pour correspondre à la nouvelle réalité. Comme toujours, la variable "*$0*" n'est pas modifiée.
Dans le cas où il serait nécessaire d'affecter la valeur "-x" (ou toute autre valeur avec "tiret"), il est alors nécessaire de faire suivre le "set" d'un double tiret "--" pour éviter que le Shell interprète ce "-x" comme une option d'activation.

Remarque : Rien n'oblige les valeurs placées après "*set*" d'être des chaînes figées. Il est donc possible d'y inclure des variables ; ou des sous-exécutions de commandes. En revanche, l'instruction "***set***" est la seule permettant de modifier les variables "*$1*", "*$2*", etc. Autrement dit, on ne peut pas modifier ces variables par une instruction du style "*1=valeur*" ou "*${1:=valeur}*".

4) Le séparateur de champs internes

Syntaxe :

```
IFS=chaîne
```

Lorsque l'instruction "*set valeur1 [valeur2 …]*" est exécutée, le Shell arrive à isoler et dé concaténer les différentes valeurs dans les différentes variables "*$1*", "*$2*", etc., grâce à la variable d'environnement "***IFS***" (*Internal Field Separator*) en majuscules, qui contient le ou les caractères devant être utilisés pour séparer les différentes valeurs ("espace" par défaut).
Une modification du contenu de cette variable permet d'utiliser un (ou plusieurs) autres caractères pour séparer des valeurs avec la commande "*set*".

Remarque : La variable "***IFS***" étant très importante pour l'analyseur syntaxique du Shell, il est conseillé de la sauvegarder avant de la modifier pour pouvoir la restaurer dans son état original après l'avoir utilisée.

Exemple :

```
#!/bin/sh
chaine="toto:titi/tata"          # Préparation de la chaîne à affecter
old=$IFS                         # Sauvegarde de la variable IFS
IFS=:/                           # L'IFS devient ":/" => le shell utilisera ":" ou "/" comme séparateur
set $chaine                      # Déconcaténation de la chaîne suivant l'IFS
IFS=$old                         # Récupération de l'ancien IFS
echo $3                          # Affichage du troisième argument ("tata")
```

VI) NEUTRALISATION DES META CARACTERES

Certains caractères alphanumériques du Shell ont une signification particulière.

Exemple : Essayer de faire afficher tel quel à l'écran le texte encadré ci-dessous

```
* (une étoile)
Il a crié "assez"; mais cela ne suffisait pas
L'habit à $100 ne fait pas le moine
A > 0 <=> A + B > B
```

Les problèmes rencontrés pour afficher ces quelques lignes proviennent des caractères ***étoile***, ***parenthèse ouvrante***, ***parenthèse fermante***, ***guillemet***, ***point-virgule***, ***dollar***, ***apostrophe***, ***backslashe***, ***inférieur*** et ***supérieur*** qui ont une signification spéciale en Shell. On les appelle "*méta caractères*" (*méta* en grec signifie "transformer") parce que le Shell les transforme avant de les traiter**.** Afin de les utiliser tels quels par le langage, on est obligé d'y appliquer un "mécanisme de neutralisation de transformation". C'est à dire qu'il faut demander au Shell de ne pas les transformer en autre chose que ce qu'ils sont.

1) Rappel sur les méta caractères

Les méta caractères suivants sont utilisés lorsque l'on désire référencer un fichier sans connaître exactement son nom

Méta caractère	Signification
*****	Toute chaîne de caractère, même une chaîne vide
?	Un caractère quelconque, mais présent
[xyz]	Tout caractère correspondant à l'un de ceux contenus dans les crochets
[x-y]	Tout caractère compris entre "*x*" et "*y*"
[!x-y]	Tout caractère qui n'est pas compris entre "*x*" et "*y*"

Les méta caractères suivants sont utilisés dans le Shell

Méta caractère	Signification
$	Contenu d'une variable
" "	Neutralisation de certains méta caractères placés à l'intérieur
' '	Neutralisation de tous les méta caractères placés à l'intérieur
\	Neutralisation de tout méta caractère placé après
( )	Groupement de commandes
;	Séparateur de commande (permet d'en placer plusieurs sur une ligne)
<	Redirection en entrée à partir d'un fichier
<<	Redirection en entrée à partir des lignes suivantes
>	Redirection en sortie vers un fichier
>>	Redirection en sortie vers un fichier en mode "ajout"
|	Redirection vers une commande (pipe mémoire)

2) Le "backslashe"

Le méta caractère "***backslashe***" a pour effet de neutraliser tout méta caractère qui le suit, quel qu'il soit. Mais il n'en neutralise qu'un seul.

Exemple : Solution avec le méta caractère "*backslashe*"

```
Prompt> echo \* \(une étoile\)
Prompt> echo Il a crié \"assez\"\; mais cela ne suffisait pas
Prompt> echo L\'habit à \$100 ne fait pas le moine
Prompt> echo A et B \> 0 \<=\> A + B \> B
```

3) L'apostrophe ou "quotte simple"

Le méta caractère "***apostrophe***" appelé aussi "***quotte simple***" a pour effet de neutraliser tous les méta caractères situés après, sauf lui-même. Cette exception permet ainsi d'avoir un début et une fin de zone de neutralisation.
Mais on ne peut pas, avec cette option, neutraliser le méta caractère "***apostrophe***" puisqu'il marque la fin de la zone. Et le méta caractère "***backslashe***" qui pourrait résoudre cet inconvénient ne fonctionnera pas puisqu'il sera lui-même neutralisé.

Exemple : Solution avec le méta caractère "*quotte simple*"

```
Prompt> echo '* (une étoile)'
Prompt> echo 'Il a crié "assez"; mais cela ne suffisait pas'
Prompt> # Impossible d'afficher la troisième ligne avec la quotte simple
Prompt> echo 'A et B > 0 <=> A + B > B'
```

4) Le double guillemet

Le méta caractère "***double guillemet***" a pour effet de neutraliser tous les méta caractères situés après, sauf le méta caractère "***dollar***" (qui permet d'accéder au contenu d'une variable), le méta caractère "***accent grave***" (qui permet la sous-exécution), le méta caractère "***double guillemet***" (qui permet de marquer la fin de la zone de neutralisation), et le méta caractère "***backslashe***" s'il est suivi d'un méta caractère de la liste ci-dessus (ce qui permet donc de neutraliser ledit méta caractère). On peut donc, avec cette option, neutraliser le méta caractère " ***double guillemet***" puisqu'il fait partie de la liste ; donc qu'il est neutralisable par le méta caractère "***backslashe***".

Exemple : Solution avec le méta caractère "*double guillemet*"

```
Prompt> echo "* (une étoile)"
Prompt> echo "Il a crié \"assez\"; mais cela ne suffisait pas"
Prompt> echo "L'habit à \$100 ne fait pas le moine"
Prompt> echo "A et B > 0 <=> A + B > B"
```

VII) LES CONTROLES BOOLEENS

1) Introduction

Le Shell étant un langage, il permet l'utilisation de contrôles "*vrai/faux*" appelés aussi "*booléens*".
Le principe est que chaque commande, chaque programme exécuté par Unix, donc par le Shell, lui retransmet en fin d'exécution un ***code de retour*** appelé aussi ***code de statut***.
La convention qui a été établie veut que si la commande s'est bien exécutée, le code de statut ait pour valeur ***zéro***. En revanche, si la commande a eu un problème dans son exécution (pas de droit d'accès, pas de fichier à éditer, etc.), son code de statut aura une valeur ***différente de zéro***. En effet, il existe toujours plusieurs raisons qui peuvent faire qu'un programme ne s'exécute pas ou mal ; en revanche il n'y a qu'une seule raison qui fait qu'il s'exécute correctement.

Cette convention ayant été établie, le Shell considèrera alors un programme de statut "*0*" comme étant "***vrai***" ; et un programme de statut "*différent de 0*" comme étant "***faux***". Grâce à cette convention, l'utilisateur peut programmer de manière booléenne en vérifiant le statut du programme qui s'est exécuté.
Remarque : Une convention n'est jamais une obligation. Rien n'empêche donc un programmeur de faire renvoyer un statut quelconque par ses programmes. En retour, les autres utilisateurs ou les autres scripts ne pourront jamais se fier au statut du programme en question.

Le statut de la dernière commande exécutée se trouve dans la variable "***$?***" du processus ayant lancé la commande. On peut donc visualiser facilement un résultat "vrai" ou "faux" de la dernière commande avec la syntaxe "***echo $?***". Bien évidemment, visualiser un statut de cette manière perd ledit statut puisque la variable "***$?***" prend la valeur de la dernière commande exécutée (ici "*echo*").

Attention : Il ne faut pas confondre "affichage" (ce que la commande affiche à l'écran) et "statut" (état de son exécution). Une commande peut ne rien afficher mais renvoie toujours un statut.

Exemple :

```
Prompt> cd /tmp                  # La commande n'affiche rien mais elle réussi
Prompt> echo $?
0

Prompt> rm /tmp                  # Si on efface un répertoire sans en avoir le droit…
rm: 0653-603 Cannot remove directory /tmp
Prompt> echo $?
2

Prompt> rm /tmp 2>/dev/null      # Même si on redirige l'affichage des erreurs…
Prompt> echo $?
2

Prompt> echo $?                  # L'affichage précédant s'étant bien exécuté…
0
```

2) La commande "*test*"

La commande "***test***" est une… commande. A ce titre, "***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, etc.
La syntaxe générale d'une commande test est "***test expression***" ; mais elle peut aussi être "***[expression]***" (à condition de ne pas oublier les espaces séparant l'expression des crochets). Ici, les crochets ne signifient pas "élément optionnel" mais bien "crochets".

a) Test simple sur les fichiers

Syntaxe :

```
test option "fichier"
```

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")

b) D'autres test simple sur les fichiers ("Korn Shell" et "Bourne Again Shell" et shells descendants)

Syntaxe :

```
test option "fichier"
```

Option	Signification
-S	fichier "socket" ("Korn Shell" et "Bourne Again Shell" et shells descendants)
-e	fichier existe quel que soit son type ("Bourne Again Shell" et shells descendants)
-O	fichier m'appartient ("Korn Shell" et "Bourne Again Shell" et shells descendants)
-G	fichier appartient à mon groupe ("Korn Shell" et "Bourne Again Shell" et shells descendants)
-N	fichier modifié depuis sa dernière lecture ("Korn Shell" et "Bourne Again Shell" et shells descendants)

c) Test complexe sur les fichiers ("Korn Shell" et en "Bourne Again Shell" et shells descendants)

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)

d) Test sur les longueurs de chaînes de caractères

Syntaxe :

```
test option "chaîne"
```

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

e) Test sur les chaînes de caractères

Syntaxe :

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

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

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

f) Test sur les 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*)

Attention : Utiliser "=" ou "*!*=" à la place de "*-eq*" ou "*-ne*" peut produire des résultats erronés.

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

g) 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)

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

```
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.

3) Les commandes "*true*" et "*false*"

Syntaxe :

```
true

false
```

Les commandes "***true***" et "***false***" n'ont d'autre but que de renvoyer un état respectivement à "*vrai*" ou "*faux*". Ces commandes jouent le rôle de commandes neutres (lorsque le Shell attend une instruction alors que le programmeur n'a rien besoin de faire; il peut toujours mettre l'instruction "***true***") ou bien permettent la création de boucles infinies (*cf. Chapitre sur les structures de contrôles*)

Exemple :

```
Prompt> true                    # La commande renvoi la valeur "vrai"
Prompt> echo $?
0

Prompt> false                   # La commande renvoi la valeur "faux"
Prompt> echo $?
1
```

4) La commande "*read*"

Syntaxe :

```
read [var1 var2 …]
```

Cette commande a déjà été vue lors de la saisie de variables. Mais elle est reprise ici pour y apporter un complément. En effet, en plus 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 est donc possible de programmer une alternative ou une boucle (*cf. Chapitre sur les structures de contrôles*) sur une saisie réussie ou pas.

VIII) LES STRUCTURES DE CONTROLES

1) Introduction

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…*")

2) L'alternative simple

Syntaxe :

```
cde1 && cde2

cde1 || cde2
```

La première syntaxe correspond à un "cde1 **ET** cde2" et 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".
La seconde syntaxe correspond à un "cde1 **OU** cde2" et 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".

Exemple : Ecrire un script affichant si on lui a passé zéro, un ou plusieurs paramètres. Ensuite il affiche 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 $*"
```

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.

3) L'alternative complexe

Syntaxe :

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

La structure "***if…then…[else]…fi***" évalue toutes les commandes placées après le "***if***" mais ne vérifie que le code de retour de la dernière commande de la liste. Dans le cas où le programmeur voudrait placer plusieurs commandes dans la "liste de commandes", il doit les séparer par le caractère "*point-virgule*" qui est un séparateur de commandes Shell. Dans la pratique, cette possibilité est très rarement utilisée, un script étant plus lisible si les commandes non-vérifiées par le "if" sont placées avant celui-ci.
Si l'état de la dernière commande est "*vrai*", le Shell ira exécuter l'ensemble des instructions placées après dans le bloc "***then***" ; sinon, il ira exécuter l'ensemble des instructions placées dans le bloc "***else***" si celui-ci existe.
Dans tous les cas, le Shell ira exécuter les instructions éventuellement placées derrière le mot-clef "***fi***" car celui-ci termine le "***if***".

Il est possible d'imbriquer plusieurs blocs "*if…fi*" à condition de placer un mot-clef "***fi***" pour chaque mot-clef "***if***".

On peut se permettre de raccourcir une sous-condition "*else if…*" par le mot-clef "***elif***". Dans ce cas, il n'y a plus qu'un seul "***fi***" correspondant au "***if***" initial.

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
```

4) Le branchement à choix multiple

Syntaxe :

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

La structure "***case…esac***" évalue la chaîne en fonction des différents choix proposés. A 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. De plus, 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 "autre cas")

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

# Evaluation 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
```

5) La boucle sur condition

Syntaxe :

```
while liste de commandes
do
        commande1
        [ commande2 …]
done

until liste de commandes
do
        commande1
        [ commande2 …]
done
```

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).

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
        # Evaluation 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
```

6) La boucle sur liste de valeurs

Syntaxe :

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

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

Dans le cas où une valeur contient un méta caractère de génération de nom de fichier ("*étoile*", "*point d'interrogation*", etc.), le Shell examinera alors les fichiers présents dans le répertoire demandé au moment de l'exécution du script et remplacera le méta caractère par le ou les fichiers dont le nom correspond au méta caractère.

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"

```
#!/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
        # Evaluation 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
```

Remarque : Ce script présente un léger "bug" 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 bug 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".

7) Interruption d'une ou plusieurs boucles

Syntaxe :

```
break [n]

continue [n]
```

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.
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.
La numérotation des boucles s'effectue à partir de la boucle la plus proche de l'instruction "***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***".

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 : Script qui fasse 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".

```
#!/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
```

8) Interruption d'un programme

Syntaxe :

```
exit [n]
```

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).

9) Le générateur de menus en boucle (Korn Shell et Bourne Again Shell et shells descendants)

Syntaxe :

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

La structure "***select… do… done***" proposera à l'utilisateur un menu pré-numéroté commençant à "*1*". A chaque numéro sera associé une chaîne prise séquentiellement dans les chaînes de la liste. Il lui sera aussi proposé de saisir un des numéros du menu (le prompt de saisie provenant de la variable "*$PS3*").
Après la saisie, la chaîne correspondant au numéro choisi sera stockée dans la variable "*$var*" pendant que la valeur du numéro choisi sera stocké dans la variable interne "*$REPLY*". Il appartient alors au programmeur d'évaluer correctement l'une de ces deux variables ("*if…fi*" ou "*case…esac*") pour la suite de son programme. Dans le cas où l'utilisateur choisit un numéro qui n'est pas dans la liste, la variable "*$var*" recevra alors une chaîne vide mais le numéro choisi sera quand-même stocké dans la variable "*$REPLY*". Cependant, la variable de statut "*$?*" n'est pas modifiée par ce choix erroné.

Comme pour la boucle "*for*", les valeurs de la liste peuvent être obtenues de différentes façons (variables, sous-exécutions, etc.). La syntaxe "*in chaîne1 …*" est optionnelle. Dans le cas où elle est omise, les valeurs sont prises dans la variable "*$**" contenant les arguments passés au programme ou à la fonction. Dans le cas où une valeur contient un méta caractère de génération de nom de fichier ("*étoile*", "*point d'interrogation*", etc.), le *Shell* examinera alors les fichiers présents dans le répertoire de travail au moment de l'exécution du script et remplacera le méta caractère par le ou les fichiers dont le nom correspondants au méta caractère.

Remarque : La phase "menu + choix" se déroule en boucle infinie. Il est donc nécessaire de programmer l'interruption de la boucle sur une valeur particulière de la variable "*$var*" ou de la variable "*$REPLY*" en utilisant une des instructions "***break***", "***return***" ou "***exit***".

IX) LES FONCTIONS

1) Introduction

Syntaxe :

```
nom_de_fonction()
{
        commande1
        [ commande2 …]
}
…
nom_de_fonction
```

Une fonction permet de regrouper des instructions fréquemment employées dans un ensemble identifié par un nom.
Ce nom, utilisé ensuite dans le script comme toute autre commande Unix, exécutera l'ensemble des instructions contenues dans la fonction. Une fois le corps de la fonction créé, il n'y a aucune différence entre "appeler une fonction" et "appeler une commande Unix".

Leur nom est soumis aux mêmes impératifs que les noms de variables : 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.

Exemple : Une fonction qui affiche la date puis un "ls"

```
#!/bin/sh
# Fonction qui affiche la date puis fait un "ls"
newls()
{
        date                            # Affichage de la date
        ls -l                           # Affichage du ls
}

# Utilisation de la fonction
newls                                   # Appel de la fonction
var1=`newls`                            # Récupération de ce que la fonction affiche

# Utilisation d'une commande Unix
ls -l                                   # Appel de la commande classique "ls -l"
var2=`ls -l`                            # Récupération de ce que la commande "ls -l" affiche

# Il n'y a aucune différence syntaxique entre l'utilisation d'une fonction ou d'une commande…
```

2) Passage de valeurs

Syntaxe :

```
nom_de_fonction()
{
        echo $0 $1 $2 $3 $4 $5 $6 $7 $8 $9 $* $#
}
…
nom_de_fonction paramètre1 [paramètre2 …]
```

Comme pour un script Shell, une fonction peut avoir besoin de valeurs non-connues à l'avance. De la même manière, ces valeurs lui seront passées comme "*argument*" ou "*paramètre*" lors de l'appel de la fonction, qui les récupèrera dans les variables bien connues "*$1*" (premier paramètre), "*$2*" (second paramètre), etc.
Il faut bien comprendre que, même si leur rôle est analogue, il y a différenciation complète entre le contenu des variables "*$1*", "*$2*", etc. du corps de la fonction et le contenu des variables "*$1*", "*$2*", etc. du programme principal. Dans le programme principal ces variables font références aux valeurs passées depuis l'extérieur du script vers le script ; dans le corps de la fonction ces variables font références aux valeurs passées depuis l'extérieur de la fonction vers la fonction.
Seule exception: "*$0*" reste invariante en conservant toujours le nom du script.

Exemple : Une fonction qui affiche si la valeur qu'elle reçoit est paire ou impaire

```
#!/bin/sh
# Fonction qui affiche la parité d'une valeur
pair_impair()
{
        test `expr $1 % 2` -eq 0 && echo "$1 est pair" || echo "$1 est impair"
}

# Pour chaque nombre passé au programme
for nb in $*
do
        # Vérification de la parité de ce nombre
        pair_impair $nb
done
```

3) Retour de fonction

Syntaxe :

```
nom_de_fonction()
{
        return [n]
}
…
nom_de_fonction
echo $?
```

L'instruction "***return [n]***" met immédiatement fin à l'exécution de la fonction.
Le paramètre "*n*" facultatif vaut "*0*" par défaut mais ne peut pas dépasser "*255*". Il correspond au "*statut*" de la fonction et est, de ce fait, retransmis à la variable "*$?*" du programme ayant appelé cette fonction. Cette instruction peut donc rendre une fonction "*vrai*" ou "*faux*" selon les conventions du Shell.
On peut faire un parallèle entre l'instruction "***return***", qui sert à interrompre l'exécution d'une fonction en faisant remonter une valeur de l'intérieur vers l'extérieur de la fonction, et l'instruction "***exit***" qui sert à interrompre l'exécution d'un script en faisant remonter une valeur de l'intérieur vers l'extérieur d'un script.

Exemple : Une fonction qui renvoie "vrai" ou "faux" si la valeur qu'elle reçoit est paire ou impaire

```
#!/bin/sh
# Fonction qui teste la parité d'une valeur
pair_impair()
{
        test `expr $1 % 2` -eq 0 && return 0 || return 1
}

# Pour chaque nombre passé au programme
for nb in $*
do
        # Vérification de la parité de ce nombre
        if pair_impair $nb
        then
                echo "$nb est pair"
        else
                echo "$nb est impair"
        fi
done
```

4) Renvoi d'une valeur par une fonction

Syntaxe :

```
nom_de_fonction()
{
        echo [valeur]
}
…
var=`nom_de_fonction`
echo $var
```

Il ne faut pas confondre la notion de "*retour de fonction*" en Shell et la notion de "*valeur renvoyée par une fonction*" telle qu'on l'entend dans d'autres langages, comme le C ou le PASCAL. En effet, en Shell, cette notion de "*valeur renvoyée*" ne peut être que simulée par l'utilisation d'un "*affichage unique et final*" dans la fonction ; ce qui permet au programmeur de récupérer dans une variable ce que la fonction affiche en utilisant les "*accents graves*" bien connu du mécanisme de la sous-exécution.

Exemple : Une fonction qui renvoie le carré du nombre qu'elle reçoit

```
#!/bin/sh
# Fonction qui renvoie le carré du nombre qu'elle reçoit
carre()
{
        # Affichage du carré du nombre reçu
        expr $1 \* $1
}

# Pour chaque nombre passé au programme
for nb in $*
do
        # Récupération du carré de ce nombre
        result=`carre $nb`

        # Affichage (ou autre traitement quel qu'il soit) de ce résultat
        echo "Le carré de $nb vaut $result"
done
```

5) Portée des variables

Il n'y a pas recopie de variable pour une fonction. Autrement dit, modifier une variable dans une fonction répercute la modification dans tout le script. En effet, comme il n'y a pas de notion de "pointeur" en Shell, c'est le seul moyen de pouvoir faire modifier une variable par une fonction. Malheureusement cela peut produire des effets de bord difficilement décelables si le programmeur ne fait pas attention.
Cependant, l'instruction "***local***" employé lors de la création de la variable a pour effet d'isoler les modifications apportées à la variable à la fonction dans laquelle la variable est modifiée. Cet effet est définitif pour la variable dans toute la fonction.

Exemple : Une fonction qui modifie deux variables : une "globale" et une "locale".

```
#!/bin/sh
# Fonction qui modifie deux variables
modif()
{
        i=5                             # Modification de la variable globale "i"
        local j=8                       # La variable "j" est rendue "locale" - La modification est locale
        j=12                            # La variable "j" reste locale - La modification reste locale
        echo "Fonction: i=$i; j=$j"     # Affichage de "i" et "j" dans la fonction
}

# Programme principal
i=0; j=0                                # Initialisation de "i" et "j"
echo "i=$i; j=$j"                       # Affichage de "i" et "j" avant l'appel
modif                                   # Appel de la fonction
echo "i=$i; j=$j"                       # Affichage de "i" et de "j" après l'appel => "j" n'a pas changé
```

Dans le cas où toutes les variables doivent être locales, et que rajouter l'instruction "local" semble fastidieux (sans compter qu'un oubli est toujours possible), une astuce simple pour transformer d'un coup toutes les variables en "local" consiste à isoler le corps de la fonction avec des parenthèses; ce qui a pour effet de faire créer un sous-shell dans lequel la fonction pourra travailler mais dans lequel les variables modifiées ne seront pas répercutées au niveau du shell parent.

Exemple : Une fonction qui modifie deux variables… mais la seconde est modifiée dans un sous-shell.

```
#!/bin/sh
# Fonction qui modifie deux variables, une dans le shell et l'autre dans un sous-shell
modif()
{
        i=5                                     # Modification de "i" dans le shell principalt
        (                                       # Création d'un sous-shell
                j=8                             # Modification de "j" dans le sous-shell
                echo "Fonction: i=$i; j=$j"     # Affichage de "i" et "j" dans le sous-shell
        )                                       # Fin du sous-shell => La modif de "j" est perdue
}

# Programme principal
i=0; j=0                                # Initialisation de "i" et "j"
echo "i=$i; j=$j"                       # Affichage de "i" et "j" avant l'appel
modif                                   # Appel de la fonction
echo "i=$i; j=$j"                       # Affichage de "i" et de "j" après l'appel => "j" n'a pas changé
```

Remarque : Toute fonction ayant donc toujours connaissance de toute variable créée par l'appelant, le choix est laissé au programmeur, soit de transmettre une variable à une fonction comme il lui transmettrait n'importe quelle valeur (qu'elle récupèrera dans "*$1*", "*$2*", etc.), soit de laisser la fonction utiliser naturellement les variables du script par leurs noms. Les deux solutions ont chacune leurs avantages et leurs inconvénients.

6) Imbrication de fonctions

Il est tout à fait possible d'intégrer la création d'une fonction en plein milieu du code principal du programme. Mais pour qu'une fonction soit "exécutable", son identificateur doit d'abord être connu (avoir été "lu") par le Shell. La lecture d'un script se faisant séquentiellement, il s'ensuit qu'une fonction ne sera exécutable que lorsque l'interpréteur du Shell sera passé par le nom de la fonction et seulement s'il passe par la partie du code contenant le nom.

Exemple : Une fonction intégrée au milieu d'un script

```
#!/bin/sh

# Début du script - La fonction n'est pas encore connue
echo "Début du script"

# Ecriture de la fonction - En passant ici, le shell prend connaissance de l'existence de la fonction
fonction()
{
        echo "Fonction"
}

# Suite du script - La fonction est maintenant connue et utilisable
echo "Suite du script"
fonction
```

Exemple : Une fonction qui ne sera connue que si une condition est vérifiée

```
#!/bin/sh

# Si la condition est vérifiée
if test "$1" = "go"
then
        # Ecriture de la fonction - S'il passe ici, le Shell prendra connaissance de son existence
        fonction()
        {
                echo "Fonction"
        }
fi

# Suite du script

# Si la condition n'a pas été vérifiée, la fonction ne sera pas connue et son appel provoquera une erreur
fonction
```

Exemple : Une fonction "interne" qui ne sera connue que si une autre fonction "externe" est appelée

```
#!/bin/sh

# Ecriture de la fonction externe
externe()
{
	echo "Fonction externe"

	# Ecriture de la fonction interne qui ne sera connue que si la fonction "externe" est appelée
	interne()
	{
		echo "Fonction interne"
	}
}

# Suite du script

# Ici, la fonction "interne" n'est pas connue - Son appel provoquera une erreur
# Cependant la fonction "externe" est connue et son appel rendra fonction "interne" connue
externe

# Maintenant, la fonction "interne" est connue et peut être appelée
interne
```

Exemple : Une fonction "interne" qui ne sera connue que dans une autre fonction "externe"

```
#!/bin/sh

# Ecriture de la fonction externe
externe()
{
	echo "Fonction externe"
	(					# Création d'un sous-shell

		# Ecriture de la fonction interne qui ne sera connue que du sous-shell
		interne()
		{
			echo "Fonction interne"
		}

		# Ici, la fonction "interne" est connue et peut être appelée
		interne
	)					# Fin du sous-shell

	# Ici, la fonction "interne" n'est pas connue - Son appel provoquera une erreur
}

# Suite du script

# Ici, la fonction "interne" n'est pas connue - Son appel provoquera une erreur
# Appel de la fonction externe
externe

# Ici, la fonction "interne" n'est toujours pas connue - Son appel provoquera toujours une erreur
```

A partir de là, il est possible de décomposer un problème en une multitude d'opérations élémentaires ; chacune d'elles exécutée par une fonction qui lui est dévolue ce qui est d'ailleurs le principe d'une fonction. Les fonctions pourront s'appeler mutuellement, et même de façon imbriquées (A appelle B qui appelle A), pour peu que chacune d'elles soit connue du Shell au moment de son appel.
Il est cependant recommandé, pour une bonne maintenance et une bonne lisibilité, de commencer un script par l'écriture de toutes les fonctions qu'il sera amené à utiliser sans complication inutile telles que ces exemples ont montré.

7) La trace de l'appelant

Syntaxe :

```
caller [num]
```

L'instruction "***caller***", qui doit obligatoirement être placée dans une fonction, donne des informations sur l'appelant de la fonction (nom du script, fonction appelante, n° de ligne de l'appel).
Le numéro indique l'incrément à apporter au niveau que l'on veut remonter ("*0*" pour remonter un niveau, "*1*" pour remonter deux niveaux, "*2*" pour remonter trois niveaux, etc.).

Exemple : Une fonction de 3° niveau qui indique la hiérarchie de ses appelants

```
#!/bin/sh

# Fonction de niveau 1
fct1()
{
        # Appel fonction de niveau 2
        fct2
}

# Fonction de niveau 2
fct2()
{
        # Appel fonction de niveau 3
        fct3
}

# Fonction de niveau 3
fct3()
{
        # Informations sur les différents appelants
        caller 0        # Appelant immédiat (fct2)
        caller 1        # Appelant 1 niveau au dessus de l'appelant immédiat (fct1)
        caller 2        # Appelant 2 niveaux au dessus de l'appelant immédiat (programme)
        caller 3        # Appelant 3 niveaux au dessus de l'appelant immédiat (il n'y en a pas)
}

# Corps du programme principal

# Appel de la fonction de base
fct1

# Informations sur les différents appelants
caller 0        # Appelant immédiat (il n'y en a pas car on est dans le programme principal)
```

X) LES INSTRUCTIONS AVANCEES

1) L'inclusion de script dans un script

Syntaxe :

```
source fic

. fic
```

L'instruction "***source***" permet d'importer un fichier à l'endroit où la commande est exécuté, le fichier importé venant remplacer la ligne contenant la commande. Les habitués du langage C reconnaîtront l'effet provoqué par la directive "*#include*".
Ceci est utile dans les situations où de multiples scripts utilisent un fichier de données communes ou une bibliothèque de fonctions.
L'instruction "**.**" (point) est un synonyme de l'instruction "***source***".

Exemple : Un script qui se charge lui-même "n" fois

```
#!/bin/sh

# Nb limite de chargements pris dans $1 s'il existe sinon pris par défaut à 100
limite=${1:-100}

# Initialisation compteur s'il n'existe pas
cpt=${cpt:-0}

# Incrément et affichage du compteur
cpt=`expr $cpt + 1`
echo "cpt=$cpt"

# Si le compteur n'a pas atteint la limite on importe le script ici
test $cpt -lt $limite && source $0          # Tout le script est intégralement recopié sous la ligne courante

# Et tout recommence à l'identique (la partie du code qui suit est une illustration du comportement)
#!/bin/sh (traité ici comme un commentaire car il ne se trouve pas en 1ère ligne)

# Nb limite de chargements pris dans $1 s'il existe sinon pris par défaut à 100
limite=${1:-100}                            # La variable est réinitialisée avec $1 qui n'a pas changé

# Initialisation compteur s'il n'existe pas
cpt=${cpt:-0}                               # Le compteur est réinitialisé avec sa valeur précédente

# Incrément et affichage du compteur
cpt=`expr $cpt + 1`
echo "cpt=$cpt"

# Si le compteur n'a pas atteint la limite on importe le script ici
test $cpt -lt $limite && source $0          # Tout le script est intégralement recopié sous la ligne courante

# Et tout recommence à l'identique (encore et encore…)
```

2) La protection contre les signaux

Syntaxe :

```
trap [-l] [-p] [commande | -] [no_signal | nom_signal …]
```

L'instruction "***trap***" permet de protéger un script contre un signal (*cf. Gestion des processus*).
Par défaut, tout script en exécution recevant un signal quelconque (depuis le clavier par "*CTRL-C*" ou depuis l'extèrieur par "*kill*") s'arrête. L'instruction "***trap***" permet de remplacer ce comportement en demandant le lancement d'une commande particulière dès réception d'un signal "*sig1*" (ou éventuellement d'un signal "*sig2*" …). Le signal à intercepter peut être représenté par son n° ou par son nom.
Ce genre d'instruction permet à un script de programmer le nettoyage de ses fichiers de travail temporaires en cas d'arrêt brutal.
Quelques détails :

- ✓ Mettre une chaîne vide "" comme commande inhibe totalement le signal concerné.
- ✓ Mettre un souligné "-" ou ne rien mettre du tout comme commande restaure le comportement par défaut (interruption du script).
- ✓ L'option "*-l*" liste les signaux disponibles avec leurs noms (tout comme la commande "*kill*" avec l'option "*-l*").
- ✓ L'option "*-p*" liste les signaux qui ont été inhibés avec la commande qui leurs sont associés.
- ✓ Le signal "9" (SIGKILL) ne peut pas être détourné ni inhibé.

Exemple : Un script qui est protégé contre l'interruption "*CTRL-C*"

```
#!/bin/sh

# Fonction qui sera appelée lors d'un "CTRL-C"
protect()
{
        echo "Il est interdit d'arrêter ce script avant sa fin naturelle"
}

# Corps du programme principal

# Mise en place de la protection contre le "CTRL-C"
trap protect SIGINT                     # On peut écrire aussi "trap protect 2"

# Mise en place d'un compteur
limite=100
cpt=0

# Tant que le compteur n'a pas atteint sa limite
while test $cpt -lt $limite
do
        # Incrément et affichage du compteur
        cpt=`expr $cpt + 1`
        echo "cpt=$cpt"

        # Petite tempo pour laisser le temps de taper "CTRL-C"
        sleep 1
done
```

3) Transformer une expression en ordre "Shell"

Syntaxe :

```
eval arguments
```

L'instruction "***eval***" va interpréter les arguments et faire exécuter le résultat comme "commande" par le Shell. Le premier mot des arguments doit donc être une commande valide.

L'utilisation de l'instruction "eval" sert surtout à créer une indirection de variable et simuler ainsi un pointeur puisque il y a deux fois interprétation ; une fois par la commande "eval" et une fois par le Shell.

Exemple :

```
Prompt> couleur="bleu"
Prompt> ptr="couleur"
Prompt>eval echo "\$$ptr      # Evaluation de la ligne echo "$couleur" => Affichera "bleu"
```

4) Arithmétique sur les variables ("Korn Shell" et "Bourne Again Shell" et shells descendants)

Syntaxe :

```
let expression
```

L'instruction "***let***" (qui n'est disponible que dans les shells récents) permet de réaliser des opérations arithmétiques sur des variables.
va interpréter les arguments et faire exécuter le résultat comme "commande" par le Shell. Le premier mot des arguments doit donc être une commande valide.

L'utilisation de l'instruction "eval" sert surtout à créer une indirection de variable et simuler ainsi un pointeur puisque il y a deux fois interprétation ; une fois par la commande "eval" et une fois par le Shell.

Exemple :

```
Prompt> let a=5                 # Identique à l'instruction "a=5"
Prompt> let a=a+1               # Fait passer "a" à "6"
Prompt> let a+=1                # Equivalent à l'instruction "let a=a+1"
```

5) Créer de nouveaux canaux d'entrées/sorties

Syntaxe :

```
exec x <fichier
exec y >fichier

…
commande quelconque de lecture 0<&x
commande quelconque d'écriture 1>&y
```

L'instruction "***exec***" va modifier un canal existant ou créer un nouveau canal numéroté associé à un fichier. Ce canal pourra être en entrée ou en sortie selon le signe utilisé. Le numéro est au choix du programmeur mais doit impérativement être supérieur à "2" s'il ne désire pas perdre ses canaux standards.
Ensuite, il sera possible de rediriger le canal d'entrée standard "0" à partir du canal entrant nouvellement créé. Les informations demandées par le programme seront alors directement lues depuis le fichier correspondant.
De-même, il sera possible de rediriger le canal de sortie standard "1" ou de sorties d'erreurs "2" vers le canal sortant nouvellement créé. Les informations affichées par le programme seront alors directement enregistrées dans le fichier correspondant.

Exemple : Ecriture d'un script lisant en parallèle les fichier "/etc/passwd" et "/etc/group".

```
#!/bin/sh
# Script qui lit en parallèle les fichiers "/etc/passwd" et "/etc/group"

# Chargement des fichiers dans les tampons numérotés "3" et "4"
exec 3 </etc/passwd
exec 4 </etc/group

# Boucle infinie
while true
do
        # Lecture des deux fichiers à partir des tampons numérotés
        read passwd 0<&3
        read group 0<&4

        # Si les deux variables sont vides, sortie de boucle
        test –z "$passwd" –a –z "$group" && break

        # Affichage de chaque variable non-vide
        test –n "$passwd" && echo "passwd: $passwd"
        test –n "$group" && echo "group: $group"
done
```

Remarque : Il est tout à fait possible d'utiliser les n° "0", "1" et "2" en paramètre de "***exec***". Dans ce cas, le n° utilisé perd son association initiale (clavier ou écran) pour être remplacé par la nouvelle association (fichier).

6) Gérer les options

Syntaxe :

```
getopts [:]liste de caractères[:] variable
```

L'instruction "***getopts***" permet de programmer un script Shell pouvant recevoir des options.

Exemple : Essayer de créer un script "essai.sh" pouvant être appelé de plusieurs façons

```
Prompt> ./essai.sh toto                 # Pas d'option et un paramètre
Prompt>./essai.sh –a –b toto            # Deux options distinctes et un paramètre
Prompt>./essai.sh –ab toto              # Deux options écrites en raccourci et un paramètre
Prompt> ./essai.sh –g 5 toto            # Une option avec valeur associée et un paramètre
```

On s'aperçoit rapidement que si on veut programmer "essai.sh" de façon à gérer tous les cas possibles; cela va conduire à une programmation lourde et difficile avec des extractions et manipulations de chaînes et de paramètres.
L'instruction "***getopts***" simplifie grandement le travail car elle gère elle-même toutes les façons possibles de demander une option; avec ou sans valeur associée. Il faut simplement se rappeler que les options se demandent toujours avant les paramètres du script.
L'instruction n'a besoin que de connaître les caractères indiquant les options attendues et d'une variable où stocker l'option effectivement trouvée.
Dans le cas où une option aurait besoin d'une valeur associée; on fait suivre le caractère correspondant à ladite option du caractère "***:***" (deux points). Si l'option est effectivement demandée avec la valeur qui lui est associée; cette valeur sera alors stockée dans la variable interne "*OPTARG*" qui est spécifique à "***getopts***".
L'instruction "***getopts***" ne récupérant que la première option demandée par l'utilisateur, il est nécessaire de l'inclure dans une boucle "*while do… done*" pour lui faire traiter toutes les options du script ; "***getopts***" renvoyant un statut différent de "*0*" (faux) quand il n'y a plus d'option à traiter. A chaque option trouvée; il suffit de positionner une ou plusieurs variables personnelles qu'on pourra utiliser par la suite.
De plus, la variable spécifique "*OPTIND*" étant incrémentée à chaque fois que "***getopts***" est invoquée (y compris la dernière où "***getopts***" renvoie "faux"); il est possible de décaler les paramètres du script avec un "*shift*" de "OPTIND – 1" pour éliminer les options et placer le premier paramètre utile en variable "$1". Plus tard dans le script; on peut réutiliser les variables positionnées lors de la boucle d'analyse des options pour traiter le fait qu'une option a été ou pas demandée.
En cas d'option non prévue dans la liste, l'instruction "***getopts***" affiche un message d'erreur à l'écran. Cet affichage peut être inhibé en faisant précéder la liste des options possibles du caractère "***:***" (deux points).
Et enfin l'instruction "***getopts***" arrête son traitement dès qu'elle rencontre deux "-" (*tirets*) accolés ce qui permet à l'utilisateur de faire traiter comme argument et non comme option une valeur avec un tiret (un nombre négatif par exemple).

Exemple : Ecriture d'un script pouvant recevoir les options "-a" et "–b" sans valeur associée; une fois "-c" avec valeur associée et plusieurs fois "-d" avec valeurs associées

```
#!/bin/sh
# Script qui traite plusieurs options et qui affiche ce qu'il a analysé
# Options possibles: -a -b -c val -d val [-d val] …

# Récupération de chaque option du script
while getopts :abc:d: opt
do
        # Analyse de l'option reçue
        case $opt in
                a)      # Mémorisation option "a" trouvée
                        opt_A="true" ;;
                b)      # Mémorisation option "b" trouvée
                        opt_B="true" ;;
                c)      # Mémorisation option "c" trouvée et mémorisation de sa valeur
                        opt_C="true"
                        val_C="$OPTARG" ;;
                d)      # Mémorisation option "d" trouvée et concaténation de sa valeur
                        opt_D="true"
                        val_D="$val_D $OPTARG" ;;
                *)
                        echo "Usage: `basename $0` [-a] [-b] [-c val] [-d val1] [-d val2] [fic1 …]"
                        exit 1
        esac
done

# Décalage des paramètres pour placer le premier argument non-optionnel en "$1"
shift `expr $OPTIND – 1`

# Affichage du résultat de l'analyse
test –n "$opt_A" && echo "L'option A a été demandée"
test –n "$opt_B" && echo "L'option B a été demandée"
test –n "$opt_C && echo "L'option C a été demandée avec la valeur [$val_C]"
test –n "$opt_D && echo "L'option D a été demandée avec les valeurs [$val_D]"

# Affichage du reste des paramètres s'il y en a
test $# -ne 0 && echo "Il reste encore $# paramètres qui sont $*" || echo "Il n'y a plus de paramètre"
```

XI) LES COMPLEMENTS

Le chapitre précédent marquait la fin de l'apprentissage du Shell de base. Il reste néanmoins un certain manque dans le langage, comme "incrémenter une variable" (dans les shells anciens), "récupérer des informations dans un fichier", etc.
Tous ces manques ont été comblés par les nombreux programmeurs du monde Unix par la création de diverses commandes répondants aux besoins les plus courants.
Voici une description de quelques commandes utiles (liste non exhaustive). Attention, il est possible que les plus récentes ne soient pas encore présentes sur tous les Unix.

1) Evaluer des expressions régulières avec la commande "expr"

Elle répond à beaucoup de besoins dans l'analyse des "*expressions régulières*". Avant d'examiner les possibilités de cette commande, on va d'abord parler de son statut d'exécution car il reste indépendant du but de l'emploi de la commande.
Statut de la commande :

- ✓ si l'expression est valide
  - ☞ si la valeur que "*expr*" affiche n'est pas "0", le statut vaut "*0*"
  - ☞ si la valeur que "*expr*" affiche est "0", le statut vaut "*1*"
- ✓ si l'expression est invalide, le statut vaut une autre valeur (en général il vaut "*2*")

Remarque : Cette commande ne peut pas être utilisée comme booléen *vrai/faux* puisqu'elle ne respecte pas les conventions du Shell et qu'il la considèrera comme "*faux*" lorsqu'elle aura renvoyé "*1*" (alors qu'elle s'est parfaitement bien déroulée). Mais rien n'empêche d'analyser en détail la valeur de son statut "*$?*".

a) Arithmétique

Syntaxe :

```
expr opérande opérateur opérande [opérateur opérande]
```

La commande va réaliser l'opération mathématique demandée sur les opérateurs cités et afficher le résultat sur la sortie standard (l'écran). Les opérandes ne peuvent être que des nombres entiers et le résultat de l'opération sera lui aussi sous forme d'un nombre entier.
La priorité mathématique des opérateurs est respectée (multiplication, division et modulo prioritaires sur addition et soustraction). Voici leur symbolique :

- ✓ + (addition)
- ✓ - (soustraction)
- ✓ * (multiplication ; attention l'étoile est un méta caractère Shell donc il doit être neutralisé)
- ✓ / (division euclidienne ou entière)
- ✓ % (modulo, qui correspond au reste d'une division entière)

Exemple : Incrémenter une variable pouvant jouer le rôle d'un compteur

```
Prompt> i=5                    # Affectation initiale
Prompt> i=`expr $i + 1`        # Utilisation du résultat de la commande
Prompt> echo $i                # Affichage final
```

b) Comparaison

Syntaxe :

```
expr opérande comparaison opérande
```

La commande "***expr***" va réaliser la comparaison mathématique demandée sur les opérateurs cités et afficher le résultat sur la sortie standard (l'écran). Les opérateurs ne peuvent être que des nombres entiers et le résultat de la comparaison sera "*0*" si la comparaison est "faux", "*1*" si elle est "vrai".
Les comparaisons possibles sont :

- ✓ = (égal)
- ✓ != (différent)
- ✓ \< (strictement inférieur à; attention le signe "inférieur" doit est un méta caractère et doit être protégé)
- ✓ \<= (inférieur ou égal à; attention le signe "inférieur" est un méta caractère et doit être protégé)
- ✓ \> (strictement supérieur à; attention le signe "supérieur" est un méta caractère et doit être protégé)
- ✓ \>= (supérieur ou égal à; attention le signe "supérieur" est un méta caractère et doit être protégé)

c) Travail sur chaînes de caractères

Syntaxe :

```
expr chaîne : argument
```

La commande "***expr***" va indiquer si la chaîne demandée débute par l'argument indiqué. Si c'est le cas ; la commande affichera soit :

- ✓ le nombre de caractères de la chaîne qui correspondent à la totalité de l'argument
- ✓ la partie de l'argument demandé s'il a été mis entre parenthèses

Exemples :

```
Prompt> expr "abcd" : "f"          # affiche "0" ("abcd" ne commence pas par "f")
Prompt> expr "abcd" : "a"          # affiche "1" ("abcd" commence par "a")
Prompt> expr "abcd" : "ab"         # affiche "2" ("abcd" commence par "ab")
Prompt> expr "abcd" : "a\(bc\)"    # affiche "bc" ("abcd" commence par "abc" et "bc" est
entre parenthèses)
Prompt> expr "abcd" : "abcde"      # affiche "0" ("abcd" ne commence pas par "abcde")
Prompt> expr "abcd" : ".*"         # affiche "4" (méta caractère "*" permettant d'avoir la
longueur de la chaîne)
```

2) Rechercher des chaînes avec la commande "grep"

Syntaxe :

```
grep [option] expression [fichier1 …]
```

La commande "***grep***" (Global Regular Expression Printer) permet d'extraire et d'afficher toutes les lignes contenant l'expression demandée. Les lignes sont prises dans l'entrée standard (clavier) mais peuvent êtres cherchées dans un ou plusieurs fichiers.
L'expression à chercher peut être une simple chaîne de caractères ; ou bien composée de méta caractères spécifiques à la commande "*grep*" qui sont :

✓ accent circonflexe ("^") : placé en début d'expression, il indique à "*grep*" de ne chercher l'expression qu'en début de chaque ligne

✓ dollar ("***$***") : placé en fin d'expression, il indique à "*grep*" de ne chercher l'expression qu'en fin de ligne

✓ point ("**.**") : il permet de représenter un caractère quelconque mais non nul dans l'expression

✓ étoile ("*****") : il indique que le caractère précédent peut se trouver entre zéro et un nombre infini de fois

✓ accolades("***{x,y}***") : elles permettent de spécifier que le caractère précédent doit être présent entre *x* et *y* fois

✓ crochets ("***[]***") : ils permettent de spécifier des ensembles de caractères recherchés

Les options modifient le fonctionnement de la commande (ne pas tenir compte des majuscules, n'afficher que le nom des fichiers contenant l'expression, n'afficher que le nombre de lignes contenant l'expression, etc.)

D'autres commandes similaires existent.

✓ fgrep (fast grep) : plus rapide en exécution mais ne permettant pas l'utilisation de méta caractères dans l'expression à chercher

✓ egrep (ehanced grep) : élargit les possibilités de recherche en donnant accès à d'autres méta caractères pour spécifier l'expression

✓ awk : utilise un script de programmation dans son propre langage (ressemblant beaucoup au langage "C") pour traiter les données entrantes selon l'algorithme programmé par l'utilisateur

Statut de la commande :

✓ si la commande trouve l'expression demandée (elle affiche au moins une ligne), le statut est "*0*"

✓ si la commande ne trouve pas l'expression demandée (rien n'est affiché), le statut est différent de "*0*" (en général, il vaut "*1*")

Exemple :

```
Prompt> cat /etc/passwd |grep "root"        # Extraction à partir de l'entrée standard

Prompt> grep "root" /etc/passwd             # Extraction à partir d'un fichier
```

3) Découper des lignes avec la commande "cut"

Syntaxe :

```
cut -fn [-dc] [-s] [fichier1 …]

cut -cn [fichier1 …]
```

La commande "***cut***" (couper) est un filtre vertical qui sélectionne le n^{ième} champ (option "*-f*" comme field) ou le n^{ième} caractère (option "*-c*") de chaque ligne. Les lignes sont prises dans l'entrée standard (clavier) mais peuvent êtres cherchées dans un ou plusieurs fichiers.
Les champs de l'option "*-f*" sont découpés suivant le caractère **tabulation**. Ce réglage par défaut peut être changé en mettant l'option "*-d*" pour spécifier le caractère de séparation de champs (délimiteur). Il peut alors être demandé d'ignorer les lignes ne contenant pas le délimiteur spécifié avec l'option "*-s*".

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt> cat /etc/passwd |cut –f3-6 –d:        # Extraction des champs 3 à 6 de chaque ligne

Prompt> cut –f1,6 –d: /etc/passwd             # Extraction des champs 1 et 6 de chaque ligne
```

4) Trier les informations avec la commande "sort"

Syntaxe :

```
sort [-n] [-r] [-o output] [-k pos] [-tc] [fichier1 …]
```

La commande "***sort***" va trier les lignes de façon alphabétiquement croissante pour les afficher à l'écran. Les lignes sont prises dans l'entrée standard (clavier) mais peuvent êtres cherchées dans un ou plusieurs fichiers.
Le tri peut être inversé (option "*-r*"), les lignes triées sur plusieurs champs (option "*-k*"), le délimiteur de champs peut être spécifié (option "*-t*") et le contenu des champs peut être considéré comme étant des nombres (option "*-n*").
Enfin, il est possible de spécifier le fichier dans lequel sera écrit le résultat du tri (option "*-o*") ce qui permet de demander à trier un fichier et le réécrire sur lui-même (ce qui n'est pas possible avec un pipe).

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt> cat /etc/passwd |sort –k3 –n –t:          # Tri numérique sur le 3ième champ
Prompt> sort –r –k4 –t: /etc/passwd –o /etc/passwd  # Tri inversé et réécriture
```

5) Filtrer les informations avec la commande "sed"

Syntaxe :

```
sed [-e script] [-e script] … [-f fichier_script] [fichier1 …]
```

La commande "***sed***" (Stream Editor) est un éditeur de flux. Elle permet de filtrer un flux de données au travers d'un ou plusieurs scripts basés sur l'éditeur "*ed*" (ex : "s/x/y/g" remplace chaque "*x*" par un "*y*" pour chaque ligne) pour avoir en sortie un flux de données modifiées. Le script peut être pris dans la ligne de commande (option "*-e*") ou dans un fichier externe (option "*-f*").
L'éditeur "ed" a été l'éditeur de base qui a donné naissance à l'éditeur "vi".

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt> cat /etc/passwd |sed –e "/root/d"                    # Suppression de ligne
Prompt> sed –e "s/root/toro/g" –e "s/home/hm/g" /etc/passwd  # Double substitution
```

6) Transformer les informations avec la commande "tr"

Syntaxe :

```
tr [-c] [-s] [-d] chaîne1 chaîne2
```

La commande "***tr***" va transposer l'entrée standard où chaque caractère correspondant à un de ceux de la chaîne 1 sera transformé en caractère pris dans la chaîne 2.
Il est possible de demander la suppression de chaque caractère de la chaîne 1 (option "*-d*") ; d'éliminer les caractères répétés (option "*-s*") et de complétion (option "*-c*").

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt> cat /etc/passwd |tr "[a-z]" "[A-Z]"      # Transposition minuscules en majuscules
```

7) Compter les octets avec la commande "wc"

Syntaxe :

```
wc [-c] [-l] [-w] [fichier1 …]
```

La commande "***wc***" (Word Count) va compter le nombre de lignes, de mots et de caractères de l'entrée standard ou du fichier passé en paramètre. Il est possible de ne demander que le nombres de lignes (option "*-l*") ; le nombre de mots (option "*-w*") ou le nombre de caractères (option "*-c*").

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt> echo "$LOGNAME" |wc –c      # Affichera le nombres de caractères de la variable
```

8) Afficher une séquence de nombres avec la commande "seq"

Syntaxe :

```
seq [option] dernier
seq [option] premier dernier
seq [option] premier incrément dernier
```

La commande "***seq***" (séquence) permet d'afficher les nombres entiers situés entre deux intervalles de façon croissante ou décroissante avec possibilité d'avoir un incrément particulier.

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

9) Afficher des données formatées avec la commande "printf"

Syntaxe :

```
printf format [arguments]
```

La commande "***printf***" (reprise à partir de la fonction "*printf()*" du C) permet d'afficher les arguments au format demandé.

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

10) Découper les noms avec les commandes "basename" et "dirname"

Syntaxe :

```
basename argument [.extension]

dirname argument
```

Les commandes "***basename***" et "***dirname***" ont pour but d'afficher respectivement le nom de base de l'argument demandé (le nom qui suit le dernier "/" de l'argument) ou le nom du répertoire (toute l'arborescence précédent le dernier "/") de l'argument.
Dans le cas de "basename"; si l'argument possède une extension (".gif", ".c", etc.); il est possible de demander à la commande d'enlever cette extension en indiquant l'extension à enlever.

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt>rep=`pwd`          # Récupère le répertoire courant dans la variable "rep"
Prompt> basename $rep     # Donnera le nom de base de la variable "rep"
Prompt> dirname $rep      # Donnera le nom du répertoire père de la variable "rep"
```

11) Filtrer les arguments avec la commande "xargs"

Syntaxe :

```
xargs [-dc] [commande [arguments commande] ]
```

La commande "***xargs***" est un filtre qui sert à passer un grand nombre d'arguments à une commande qui ne peut en accepter qu'un petit nombre. Il constitue une bonne alternative aux guillemets inversés dans les situations où cette écriture échoue avec une erreur "too many arguments".
Cette commande va recevoir en entrée les arguments à traiter et les découpera en morceaux suffisemments petits pour que la commande citée puisse les traiter.
Les arguments sont découpés suivant le caractère **espace** ou **tabulation**. Ce réglage par défaut peut être changé en mettant l'option "*-d*" pour spécifier le caractère de séparation des arguments (délimiteur).

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt>grep -l toto `ls`          # Recherche "toto" dans les fichiers donnés par "ls"
sh: /bin/grep: Too many arguments # "grep" ne peut pas traiter autant de fichiers
Prompt>ls |xargs grep -l toto     # "xargs" découpera le "ls" pour le "grep"
```

12) Mathématiques en virgule flottante avec la commande "bc"

Syntaxe :

```
bc [-l]
```

La commande "***bc***" (Basic Calculator) permet d'effectuer des calculs mathématiques (comme la commande "expr") mais la commande connaît quelques fonctions mathématiques (comme "racine carrée", "logarithme", etc."). Les calculs se font depuis l'entrée standard et se font en valeurs entières sauf si on demande le calcul au format "long" (option "*-l*").

Statut de la commande : toujours "*0*" sauf en cas d'erreur de syntaxe

Exemple :

```
Prompt> echo "20 / 3" |bc –l                # Affichera le résultat de l'opération
```

13) Gérer les options avec la commande "getopt"

Syntaxe :

```
getopt [:]liste de caractères[:] arguments …
```

La commande "***getopt***" correspond en externe à l'instruction interne "*getopts*" et permet la gestion et la réorganisation des paramètres.
L'utilisation de la commande "***getopt***" est similaire à l'instruction "*getopts*". Le programmeur doit lui indiquer les caractères indiquant les options attendues et les arguments où se trouvent les options à traiter. Dans le cas où une option aurait besoin d'une valeur associée; il fait suivre le caractère correspondant à ladite option du caractère "***:***" (deux points).
Cependant, contrairement à l'instruction "*getopts*", la commande "***getopt***" se contente de reformater proprement les options trouvées (une option après l'autre) et renvoie cette suite à l'écran. Donc, afin de pouvoir traiter les options dans un script, il est nécessaire de récupérer le résultat de "***getopt***" en utilisant les "*accents graves*" bien connu du mécanisme de la sous-exécution.
Dans le cas où une option a besoin d'être associée à une valeur, cette valeur est placée en argument suivant l'option à laquelle elle est associée.
Dès que l'ensemble des options est traité, c'est-à-dire dès que la commande "***getopt***" trouve une option qui ne correspond pas à une option attendue ou dès qu'elle ne trouve plus d'options, la commande affiche deux tirets ("-") et affiche le reste des arguments non traités.
En cas d'option non prévue dans la liste, la commande "***getopt***" affiche un message d'erreur à l'écran. Cet affichage peut être inhibé en faisant précéder la liste des options possibles du caractère "***:***" (deux points).
Et enfin la commande "***getopt***" arrête son traitement dès qu'elle rencontre deux "-" (*tirets*) accolés ce qui permet à l'utilisateur de faire traiter comme argument et non comme option une valeur avec un tiret (un nombre négatif par exemple).

Statut de la commande :

✓ si la commande ne trouve pas d'option invalide dans les paramètres du script, le statut est "*0*"

✓ si la commande trouve une option invalide dans les paramètres du script, le statut est différent de "*0*" (en général, il vaut "*1*")

Exemple : Ecriture d'un script pouvant recevoir les options "-a" et "–b" sans valeur associée; une fois "-c" avec valeur associée et plusieurs fois "-d" avec valeurs associées

```
#!/bin/sh
# Script qui traite plusieurs options et qui affiche ce qu'il a analysé
# Options possibles: -a -b -c val -d val [-d val] …

# Réorganisation des options du script dans la variable "opt"
opt=`getopt :abc:d: $*`; statut=$?

# Si une option invalide a été trouvée
if test $statut -ne 0
then
        echo "Usage: `basename $0` [-a] [-b] [-c val] [-d val1] [-d val2] [fic1 …]"
        exit $statut
fi

# Traitement de chaque option du script
set -- $opt                    # Remarquez la protection des options via le double tiret "--"
while true
do
        # Analyse de l'option reçue - Chaque option sera effacée ensuite via un "shift"
        case "$1" in
                -a)     # Mémorisation option "a" trouvée
                        opt_A="true"; shift;;

                -b)     # Mémorisation option "b" trouvée
                        opt_B="true"; shift ;;

                -c)     # Mémorisation option "c" trouvée et mémorisation de sa valeur
                        opt_C="true"; shift; val_C="$1"; shift;;

                -d)     # Mémorisation option "d" trouvée et concaténation de sa valeur
                        opt_D="true"; shift; val_D="$val_D $1"; shift;;

                --)     # Fin des options - Sortie de boucle
                        shift; break;;

        esac

done

# Affichage du résultat de l'analyse
test –n "$opt_A" && echo "L'option A a été demandée"
test –n "$opt_B" && echo "L'option B a été demandée"
test –n "$opt_C && echo "L'option C a été demandée avec la valeur [$val_C]"
test –n "$opt_D && echo "L'option D a été demandée avec les valeurs [$val_D]"

# Affichage du reste des paramètres s'il y en a
test $# -ne 0 && echo "Il reste encore $# paramètres qui sont $*" || echo "Il n'y a plus de paramètre"
```

14) Gérer son affichage à l'écran avec les codes "Escape"

Syntaxe :

echo *code particulier*

tput *argument_tput*

L'utilisation de certains codes appelés "codes Escape" permet d'influer sur l'affichage de l'écran. Ces codes portent ce nom car ils sont tous précédés du caractère "Esc" ("*033*" en octal).
Bien souvent, ces codes varient en fonction de l'écran que l'on veut gérer, c'est pourquoi, il vaut mieux utiliser la commande "*tput*" qui envoie elle-même les codes appropriés en fonction de l'écran utilisé.
Il vous est proposé ici une liste non-exhaustive de certains codes avec leur signification.

Codes échappements	**Commande TPUT**	**Signification**
echo "\033[2J"	tput clear	efface l'écran
echo "\033[0m"	tput smso	aucun attribut (blanc sur noir)
echo "\033[1m"		gras
echo "\033[4m"		souligne
echo "\033[5m"	tput blink	clignote
echo "\033[7m"	tput rmso	vidéo inverse
echo "\033[8m"		invisible
echo "\033[30m"		noire (avant plan)
echo "\033[31m"		rouge
echo "\033[32m"		vert
echo "\033[33m"		jaune
echo "\033[34m"		bleu
echo "\033[35m"		magenta
echo "\033[36m"		cyan
echo "\033[37m"		blanc
echo "\033[40m"		noire (arrière plan)
echo "\033[41m"		rouge
echo "\033[42m"		vert
echo "\033[43m"		jaune
echo "\033[44m"		bleu
echo "\033[45m"		magenta
echo "\033[46m"		cyan
echo "\033[47m"		blanc
echo "\033[#A"		déplace le curseur de # ligne en haut
echo "\033[#B"		déplace le curseur de # ligne en bas
echo "\033[#C"		déplace le curseur de # colonne à droite
echo "\033[#D"		déplace le curseur de # colonne à gauche
echo "\033[s"		sauvegarde la position du curseur
echo "\033[u"		restaure la position du curseur
echo "\033[K"		efface la ligne courante
echo "\033[lig;colH\c"		positionne le curseur en lig - col
echo "\033[r"		réinitialise les attributs

echo "\033[12;19’r’" echo "\033[?6h"		définit une fenêtre lig 12 à 19 active la fenêtre
echo "\033[r"		désactive la fenêtre
echo "\033(B"		mode texte
echo "\033(0"		mode graphique

XII) EXEMPLES DIVERS

1) Afficher une phrase sans que le curseur passe à la ligne suivante

La commande "*echo*" possède l'option "-n" qui permet de ne pas aller à la ligne en fin d'affichage. Malheureusement, cette option n'est pas présente sur tous les Unix.
Cet exemple simple et portable permet d'avoir une invite de saisie sans retour à la ligne en fin de phrase

```
Prompt> echo "Phrase quelconque: " |awk '{printf("%s", $0)}'
```

2) Vérifier l'existence d'un fichier quel que soit son type

La commande "*test*" permet de tester l'existence de fichiers comportant certaines caractéristiques (fichier classique, répertoire, vide, non-vide, etc.). Mais l'option "-*e*" qui permet de tester la seule existence d'un fichier quel que soit son type n'existe pas en Bourne Shell.

```
#!/bin/ksh
# Programme qui affiche si le fichier demandé existe ou n'existe pas (en Korn Shell)
# Usage: prog fichier

test -e "$1" && echo "Le fichier $1 existe" || echo "Le fichier $1 n'existe pas"
```

Cependant il y a la commande "*ls*" qui, en plus de lister le fichier, renvoie un statut "vrai/faux" si le fichier demandé existe ou n'existe pas.
Il suffit de rediriger tout son affichage (normal et erreurs) vers le fichier poubelle "*/dev/null*" pour s'en servir comme simple contrôle pour vérifier l'existence du fichier listé.

```
#!/bin/sh
# Programme qui affiche si le fichier demandé existe ou n'existe pas (en Bourne Shell)
# Usage: prog fichier

ls -d "$1" 1>/dev/null 2>/dev/null && echo "Le fichier $1 existe" || echo "Le fichier $1 n'existe pas"
```

3) Vérifier la numéricité d'une variable en terme de "nombre entier"

En demandant à la commande "*expr*" de faire un calcul simple sur une variable on peut vérifier, si le calcul réussit ou pas, si la variable en question est ou n'est pas numérique.

```
#!/bin/sh
# Programme qui vérifie si son argument est ou n'est pas numérique entier
# Usage: prog chaîne

# Essai de calcul sur l'argument 1 et récupération du code de retour
expr "$1" + 0 1>/dev/null 2>/dev/null; statut=$?

# Vérification de la réussite du calcul
test $statut -lt 2 && echo "L'argument $1 est un nombre" || echo "L'argument $1 n'est pas un nombre"
```

4) Vérifier la numéricité d'une variable en terme de "nombre réel"

La commande "expr" ne travaille qu'en nombre entiers et ne peut donc pas vérifier si une variable est ou n'est pas un nombre réel.
Un des moyens proposés ici sera, pour vérifier qu'un argument est bien un nombre réel, de supprimer chacun de ses chiffres, le point éventuel et le premier "-" du signe négatif et de regarder s'il reste quelque chose.

```
#!/bin/sh
# Programme qui vérifie si son argument est ou n'est pas numérique réel
# Usage: prog chaîne

# Suppression de chaque chiffre, du premier point et du signe "moins" s'il est en début de nombre
verif=`echo $1 |sed –e "s/[0-9]//g" –e "s/\.//" –e "s/^-//"`

# Si le résultat est vide, c'est que c'était un nombre correct
test –z "$verif" && echo "L'argument $1 est un nombre" || echo "L'argument $1 n'est pas un nombre"
```

5) Nom de base du répertoire courant

Ce script affiche le nom de base du répertoire dans lequel on est positionné

```
#!/bin/sh
# Programme qui affiche le dernier nom du répertoire courant (nom de base)
# Usage: prog

# Cas particulier : verification si le répertoire courant est "/"
if test `pwd` = "/"
then
        echo "/"
        exit 0
fi

# Mémorisation variable IFS
OLDIFS="$IFS"

# Configuration variable IFS sur le séparateur "/"
IFS=/

# Eclatement du répertoire courant dans les variables $1, $2, ...
set `pwd`

# Remise en place variable IFS et effacement variable OLDIFS inutile
IFS="$OLDIFS"; unset $OLDIFS

# Décalage du nb d'argument – 1 et affichage paramètre 1 (qui est devenu le dernier nom)
shift `expr $# - 1`
echo $1

# Remarque: Tout ce script peut être avantageusement remplacé par "basename `pwd`"
```

6) Vérification de l'autorisation d'accès d'un utilisateur quelconque

Ce script vérifie si les noms qu'on lui passe en paramètres sont autorisés à se connecter sur la machine ou non. Pour ceux qui sont autorisés, il donne les informations diverses sur l'utilisateur (uid, gid, commentaire, home) puis il affiche s'ils sont effectivement connectés

```
#!/bin/sh
# Programme qui affiche si les utilisateurs existent ou pas et connectés ou pas
# Usage: prog user1 [user2 ...]

# Boucle sur chaque argument passé au programme
for user in $*
do
	# Récupération de la ligne concernant l'utilisateur dans "/etc/passwd"
	lig=`grep "^$user:" /etc/passwd`; statut=$?

	# Vérification si le "grep" a trouvé
	if test $statut -eq 0		# On peut aussi faire if test -n "$lig"
	then
		# Récupérations informations sur l'utilisateur
		uid=`echo $lig |cut -f3 -d:`
		gid=`echo $lig |cut -f4 -d:`
		comment=`echo $lig |cut -f5 -d:`
		home=`echo $lig |cut -f6 -d:`

		# Recherche de la ligne concernant le gid dans "/etc/group"
		lig=`grep ":$gid:" /etc/group`; statut=$?

		# Vérification si le "grep" a trouvé
		if test $statut -eq 0
		then
			# Récupérations informations sur le groupe utilisateur
			groupe=`echo $lig |cut -f1 -d:`
		else
			# Le gid n'est pas dans "/etc/group"
			groupe="inconnu"
		fi

		# Recherche si l'utilisateur est connecté
		who |fgrep $user 1>/dev/null && connect="oui" || connect="non"

		# Affichage des informations trouvées
		echo "$user est autorisé à se connecter sur `uname -n`"
		echo "Uid: $uid"
		echo "Gid: $gid ($groupe)"
		echo "Commentaire: $comment"
		echo "Home: $home"
		echo "Connecté en ce moment: $connect"
	else
		echo "$user n'est PAS autorisé sur `uname -n`"
	fi

	# Saut de ligne avant de passer à l'utilisateur suivant
	echo
done
```

7) Membres d'un groupe

Ce script affiche les utilisateurs membres du groupe qu'on lui passe en argument et s'ils sont connectés ou non

```
#!/bin/sh
# Programme qui donne la liste des membres d'un groupe et les connectés
# Usage: prog groupe1 [group2 ...]

# Boucle sur chaque groupe passé au programme
for groupe
do
	echo "Membres du groupe $groupe"

	# Récupération ligne contenant le groupe demandé dans "/etc/group"
	ligne=`grep "^$groupe:" /etc/group`; statut=$?

	# Si recherche réussie (groupe existe)
	if test $statut -eq 0
	then
		# Extraction du gid (3°champ) de la ligne
		gid=`echo $ligne |cut -f3 -d:`

		# Découpage de /etc/passwd sur les champs 1 et 4 (pour enlever les champs parasites)
		# Extraction des lignes contenant le gid trouvé
		# Découpage de cette extraction sur le premier champ
		# Tri de cette extraction
		# Boucle sur chaque nom de ce tri
		cut -f1,4 -d: /etc/passwd |grep ":$gid$" |cut -f1 -d: |sort |while read user
		do
			# Si l'utilisateur est présent dans la commande "who"
			who |fgrep "$user" 1>/dev/null && connect="connecté" || connect="non connecté'

			# Affichage de l'utilisateur et de son état (connecté/non connecté)
			echo "User: $user ($connect)"
		done
	fi
	echo
done
```

8) Serveurs gérés par "inetd"

Ce script affiche les informations sur le serveur qu'on lui passe en argument si celui-ci est géré par le super-serveur "inetd"

```
#!/bin/sh
# Programme qui donne des informations sur les serveurs gérés par "inetd"
# Usage: prog serveur1 [serveur2 ...]

# Remarque: Ce script ne fonctionne pas sous Linux qui utilise "xinetd" très différent de "inetd"

# Programme
# Pour chaque serveur demandé
for serveur in $*
do
	# Vérification serveur présent dans "/etc/inetd.conf"
	if grep "^$serveur" /etc/inetd.conf 1>/dev/null
	then
		# Traitement lignes contenant serveur dans "/etc/services" et non en commentaire
		grep -v "^#" /etc/services |grep "$serveur" |while read lig
		do
			# Déconcaténation de la ligne
			set $lig

			# Récupération et élimination deux premières info.
			nom=$1
			port=`echo $2 |cut -f1 -d/`
			proto=`echo $2 |cut -f2 -d/`
			shift 2

			# Vérification serveur est dans nom ou aliases
			unset trouve
			for rech in $nom $*
			do
				if test "$rech" = "$serveur"
				then
					trouve="non vide"
					break
				fi
			done

			# Si serveur présent
			if test -n "$trouve"
			then
				echo "Serveur: $nom"
				echo "Port: $port"
				echo "Protocole: $proto"
				test -n "$*" && echo "Aliases: $*"
				echo
			fi
		done
	else
		echo "Serveur $serveur non géré par inetd"
		echo
	fi
done
```

9) Recherche d'une commande

Ce script permet de trouver l'endroit exact de toute commande Unix connue de celui qui le lance. Il découpe le PATH de l'utilisateur et vérifie si la commande demandée se trouve dans un des répertoires de recherche.

```
#!/bin/sh
# Programme qui recherche un fichier par rapport au PATH
# Usage: prog file1 [file2 ...]

# Récupération des arguments passés au programme car ils vont être écrasés
arg=$*

# Décomposition du PATH par rapport au séparateur ":"
OLDIFS="$IFS"
IFS=:
set $PATH                               # Ici on écrase les arguments passés au programme
IFS="$OLDIFS"
unset OLDIFS

# Boucle sur chaque argument
for file in $arg
do
        # On indique par défaut que le fichier n'est pas trouvé
        unset trouve

        # Boucle sur chaque répertoire du PATH
        for rep in $*
        do
                # Vérification si rep contient fichier non vide et exécutable
                if test -f "$rep/$file" -a -s "$rep/$file" -a -x "$rep/$file"
                then
                        # On a trouvé
                        ls -l "$rep/$file"
                        trouve="non vide"

                        # Plus la peine de tester les autres répertoires
                        break           # Facultatif
                fi
        done

        # S'il n'a rien trouvé
        test -z "$trouve" && echo "no $file in $*"
done
```

10) Arborescence d'un répertoire

Ce script affiche le contenu des répertoires qu'on lui passe en argument sous forme d'arborescence (correspond à la commande "tree" du DOS).

```
#!/bin/sh
# Programme qui affiche les répertoires sous forme d'arborescence (tree DOS)
# Usage: prog fic1 [fic2 ...]

# Fonction d'affichage d'un texte précédé de "n" tabulations
# Paramètres entrée :
#	nb de tabulations
#	texte à afficher (…)
# Valeur sortie : aucune
affiche_tabul()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Récupération du nb de tabulations - On supprime $1
		tab=$1; shift

		# Affichage du texte précédé des tabulations en utilisant "awk"
		echo $* |awk -vtab=$tab '{for (i=0; i<tab; i++) printf("\t"); printf("%s\n", $0)}'
	)
}

# Fonction arborescence (fonction récursive)
# Paramètres entrée :
#	nom de fichier
#	profondeur (facultatif)
# Valeur sortie : aucune
tree_r()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Récupération profondeur si elle existe
		prof=${2:-0}

		# Affichage de l'argument 1 avec autant de tabulations que profondeur
		affiche_tabul $prof $1

		# Si argument 1 répertoire et pas lien symbolique
		if test -d "$1" -a ! -L "$1"
		then
			# De nouveau création d'un sous-shell pour isoler de nouveau le contexte
			(
				# Déplacement dans ce répertoire (seul ce contexte se déplace)
				cd "$1"

				# Pour chaque fichier de ce répertoire
				ls -1 |while read file
				do
					# Fonction récursive sur ce fichier avec profondeur incrémentée
					tree_r "$file" `expr $prof + 1`
				done
			)
			# Fin du sous-shell - Le contexte actuel n'a pas changé de répertoire
		fi
	)
}

# Programme
```

```
# Parcourt récursivement les fichiers passés
for fic in $*
do
        echo "Arborescence de '$fic'"
        tree_r $fic
        echo
done
```

11) Factorielle d'un nombre

Ce script donne la factorielle des nombres qu'on lui passe en argument. Il montre le calcul par une fonction récursive et le calcul par une fonction itérative.

```
#!/bin/sh
# Programme qui calcule la factorielle d'un nombre
# Usage: prog nb1 [nb2 ...]

# Fonction de vérification d'argument factorisable (entier et positif)
# Paramètres entrée : argument à vérifier
# Valeur sortie :
#	argument correct (0)
#	argument non-correct (not 0)
factorisable()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Essai de calcul sur l'argument
		expr "$1" + 0 1>/dev/null 2>/dev/null; statut=$?

		# Si calcul échoué alors argument non entier
		test $statut -ge 2 && return 1

		# Si nombre négatif alors argument non factorisable
		test $1 -lt 0 && return 2

		# Argument numérique et positif
		return 0
	)
}

# Fonction factorielle itérative
# Paramètres entrée :
#	arg1: nombre à calculer
# Valeur sortie : aucune
fact_i()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Initialisation des variables
		res=1
		val=$1

		# Tant que nombre supèrieur ou égal à 2
		while test $val -ge 2
		do
			# Calcul du résultat intermédiaire
			res=`expr $res \* $val`

			# Décrément de la valeur
			val=`expr $val - 1`
		done

		# Affichage du résultat - Permet de simuler le renvoi d'une valeur
		echo $res
	)
}

# Fonction factorielle récursive
# Paramètres entrée : nombre à calculer
# Valeur sortie : aucune
```

```
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Si nombre à calculer inférieur à 2
		if test $1 -lt 2
		then
			# Affichage factorielle du nombre (forcément 0 ou 1) qui est 1 et sortie de fonction
				echo 1
			return 0
		fi

		# Calcul du nouvel argument à passer à la factorielle
		decrement=`expr $1 - 1`

		# Affichage de l'argument * factorielle (argument - 1)
		expr $1 \* `fact_r $decrement`
	)
}

# Programme principal
# Boucle sur chaque argument
for nb
do
	# Test argument
	factorisable $nb; statut=$?

	# Evaluation argument
	case $statut in
		0) # Argument correct
			# Factorielle du nombre - Utilisation des deux fonctions (récursives et itératives)
			echo "Factorielle $nb=`fact_i $nb` (itératif)=`fact_r $nb` (récursif)"
			;;
		1) # Argument pas numérique
			echo "On ne peut pas mettre en factorielle $nb car ce n'est pas un nombre"
			;;
		2) # Argument pas positif
			echo "On ne peut pas mettre en factorielle $nb car c'est un nombre négatif"
			;;
	esac
done
```

12) PGCD de deux nombres

Ce script donne le PGCD de chaque couple de deux nombres qu'on lui passe en paramètre. La fonction "pgcd" est développé en itérative et en récursive. Si on veut en plus le PPCM, il suffit de multiplier les deux nombres et de diviser le résultat par le PGCD.

```
#!/bin/sh
# Programme qui calcule le pgcd de deux nombres
# Usage: prog x y [x y ...]

# Fonction de vérification de tous ses arguments entiers et positifs
# Paramètres entrée : arguments à vérifier (…)
# Valeur sortie :
#	tous arguments corrects (0)
#	au-moins un argument non-correct (not 0)
numeric()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Essai de calcul sur chaque argument
		for nb in $*
		do
			# Si nombre négatif alors argument sans PGCD
			test $nb -lt 0 && return 2

			# Opération permettant de vérifier si l'argument est un nombre
			expr "$nb" + 0 1>/dev/null 2>/dev/null; statut=$?

			# Si calcul échoué alors argument non numérique
			test $statut -gt 1 && return 1
		done

		# Renvoi OK
		return 0
	)
}

# Fonction pgcd itérative
# Paramètres entrée :
#	nombre1
#	nombre2
# Valeur sortie : aucune
pgcd_i()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Tq $2 différent de 0
		while test $2 –ne 0
		do
			# pgcd(x, y)=pgcd(y, x mod y)
			set $2 `expr $1 % $2`
		done

		# Résultat
		echo $1
		return 0
	)
}

# Fonction pgcd récursive
# Paramètres entrée :
```

```
#		nombre2
# Valeur sortie : aucune
pgcd_r()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Si nb2 égal 0 alors pgcd égal nb1
		if test $2 -eq 0
		then
			echo $1
			return 0
		fi

		# pgcd(x, y)=pgcd(x mod y, y)
		pgcd_r $1 `expr $1 % $2`
	)
}

# Programme
# Tant qu'il y a au moins deux paramètres
while test $# -ge 2
do
	# Test argument
	numeric $1 $2; statut=$?

	# Evaluation argument
	case $statut in
		0) # Argument correct
			# Positionnement des deux nombres, le 1er supérieur au 2°
			if test $1 -ge $2
			then
				nb1=$1
				nb2=$2
			else
				nb1=$2
				nb2=$1
			fi
			# PGCD des nombres - Utilisation des deux fonctions (récursives et itératives)
			echo "Pgcd ($1 $2)==`pgcd_r $nb1 $nb2` (récursif)=`pgcd_i $nb1 $nb2` (itératif)"
			;;
		1) # Argument pas numérique
			echo "L'un des deux paramètres '$1 $2' n'est pas un nombre"
			;;
		2) # Argument pas positif
			echo "L'un des deux paramètres '$1 $2' est un nombre négatif"
			;;
	esac

	# Passage aux deux paramètres suivants
	shift 2
done
```

13) Division en nombres réels

Ce script affiche le résultat de la division de deux nombres réels. Le principe de la division est de partir du reste, le multiplier par 10 et continuer à diviser ainsi (comme au CM2). Dans le cas d'une division infinie, il s'arrête par défaut au bout de *n* décimales mais l'utilisateur a la possibilité de lui indiquer la précision qu'il désire.

```
#!/bin/sh
# Programme de division en nombres réels
# Usage: prog [-h] [-p précision] [-v] [--] dividende diviseur
#	Option "-h": Pour avoir l'aide sur l'utilisation de ce programme
#	Option "-p précision": Pour paramétrer la précision décimale du calcul
#	Option "-v": Affichage des détails

# Fonction affichant la façon d'utiliser ce programme
# Paramètres entrée : texte à afficher (facultatif)
# Valeur sortie : aucune
usage()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Affichage des messages particuliers
		test -n "$*" && echo "`basename $0`: $*"

		# Affichage du message d'aide
		echo "Usage: `basename $0` [-h] [-p précision] [-v] [--] dividende diviseur"
	)
}

# Fonction affichant un texte sans retour à la ligne
# Paramètres entrée : texte à afficher (…)
# Valeur sortie : aucune
puts()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Filtre d'affichage
		echo $* |awk '{printf("%s", $0)}'
	)
}

# Fonction de vérification de tous ses arguments numériques
# Paramètres entrée : arguments à vérifier (…)
# Valeur sortie :
#	tous arguments corrects (0)
#	au-moins un argument non-correct (not 0)
numeric()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Boucle sur chaque argument
		for nb in $*
		do
			# Suppression de chaque chiffre et du "." et du signe
			nb=`echo $nb |sed -e "s/[0-9]//g" |sed -e "s/\.//" |sed -e "s/^-//"`

			# Vérification => nb non vide => pas un nombre
			test -n "$nb" && return 1
		done

		# Renvoi OK
```

```
	)
}

# Fonction multipliant un nombre par 10 "n" fois (10 exposant "n")
# Paramètres entrée :
#	nombre à multiplier
#	exposant
# Valeur sortie : aucune
mult10()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Affichage du résultat
		echo "$1 * 10^$2" |bc -l
	)
}

# Fonction enlevant les zéros de droite ou de gauche d'un nombre
# Paramètres entrée :
#	nombre à réduire
#	coté à traiter (g|d)
# Valeur sortie : aucune
enleve0()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Evaluation enlève gauche ou droite
		case $2 in
			d)	# Droite
				string="0\{1,\}$";;
			g)	# Gauche
				string="^0\{1,\}";;
		esac

		# Suppression des zéros
		nb=`echo $1 |sed -e "s/$string//g"`

		# Affichage du résultat
		echo ${nb:-0}
	)
}

# Fonction transformation négatif en positif
# Paramètres entrée : nombre à transformer
# Valeur sortie :
#	le nombre a été modifié (0)
#	le nombre n'a pas été modifié (1)
neg2pos()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Si le nombre ne commence pas par "-"
		if test `echo $1 |cut -c1` != "-"
		then
			# Le nombre est positif
			nb=$1

			# Le nombre n'a pas changé
			flag=1
		else
			# Le nombre est négatif => suppression premier caractère (le "-")
```

```
                        # Le nombre a changé
                        flag=0
                fi

                # Affichage nombre - Renvoi flag
                echo $nb
                return $flag
        )
}

# Fonction décomposant un nombre en partie entière et décimale
# Paramètres entrée : nombre à décomposer
# Valeur sortie : aucune
decomp()
{
        # Création d'un sous-shell pour isoler le contexte
        (
                # Vérification si nombre décimal
                if echo "$1" |fgrep "." 1>/dev/null
                then
                        # Récupération partien entière et fractionnelle
                        ent=`echo $1 |cut -f1 -d.`
                        frc=`echo $1 |cut -f2 -d.`

                        # Elimination zéro non significatif de la partie entière et fractionnelle
                        ent=`enleve0 "$ent" g`
                        frc=`enleve0 "$frc" d`

                        # Récupération longueur partie fraction si celle-ci existe
                        test $frc -ne 0 && lgF=`expr "$frc" : ".*"` || lgF=0
                else
                        ent=`enleve0 "$1" g`
                        frc=0
                        lgF=0
                fi

                # Récupération longueur partie entière
                lgE=`expr "$ent" : ".*"`

                # Affichage des éléments (partie entière, partie fractionnelle et lg)
                echo "$ent $frc $lgE $lgF"
        )
}

# Programme principal

# Gestion des options (on utilise l'instruction "getopts" plus souple que la commande "getopt")
while getopts hp:v opt
do
        case $opt in
                h) # Aide demandée
                        usage
                        exit 0
                        ;;
                p) # Demande de précision
                        opt_prec="$OPTARG"

                        # Précision négative ???
                        if test $opt_prec -lt 0
                        then
```

```
                                    exit 1
                            fi
                            ;;
                    v) # Demande volubilité
                            opt_verb="true"
                            ;;
                    *) # Autre option
                            usage
                            exit 1
            esac
done
shift `expr $OPTIND - 1`

# Vérification assez d'arguments
test $# -lt 2 && usage "Pas assez d'argument" && exit 1

# Vérifications arguments 1 et 2 sont des nombres
numeric $1 $2; statut=$?
test $statut -ne 0 && echo "Une de ces valeurs '$1' ou '$2' n'est pas un nombre" && exit 2

# Initialisation variables par défaut
opt_prec=${opt_prec:-5}                                 # Par défaut si "opt_prec" vide

# Récupération diviseur et dividende avec gestion des négatifs
dividende=`neg2pos $1` && fl_neg=1 || fl_neg=0          # Diviseur + gestion négatif
diviseur=`neg2pos $2` && fl_neg=`expr 1 - $fl_neg`      # Dividende + gestion négatif
test -n "$opt_verb" && echo "$1 / $2 => $dividende / $diviseur (flag négatif=$fl_neg)"

# Décomposition dividende
decomp=`decomp $dividende`
ent1=`echo $decomp |cut -f1 -d" "`
frc1=`echo $decomp |cut -f2 -d" "`
lgF1=`echo $decomp |cut -f4 -d" "`
test -n "$opt_verb" && echo "Décomposition $dividende => $ent1 (ent) + $frc1 (frac)"

# Décomposition diviseur
decomp=`decomp $diviseur`
ent2=`echo $decomp |cut -f1 -d" "`
frc2=`echo $decomp |cut -f2 -d" "`
lgF2=`echo $decomp |cut -f4 -d" "`
test -n "$opt_verb" && echo "Décomposition $diviseur => $ent2 (ent) + $frc2 (frac)"

# Suppression parties fractionnelles des nombres en les multipliant par "10"
dividende=$ent1
diviseur=$ent2
test $lgF1 -gt $lgF2 && lgF=$lgF1 || lgF=$lgF2
dividende=`mult10 $dividende $lgF`
diviseur=`mult10 $diviseur $lgF`

# Si fraction dividende plus petit fraction diviseur
if test $lgF1 -lt $lgF2
then
        lgF=`expr $lgF2 - $lgF1`
        frc1=`mult10 $frc1 $lgF`
fi

# Si fraction diviseur plus petit fraction dividende
if test $lgF2 -lt $lgF1
then
        lgF=`expr $lgF1 - $lgF2`
        frc2=`mult10 $frc2 $lgF`
```

```
dividende=`expr $dividende + $frc1`
diviseur=`expr $diviseur + $frc2`
test -n "$opt_verb" && echo "Calcul réel: $dividende / $diviseur"

# Division par zéro => Interdit sauf si dividende vaut 0
if test $diviseur -eq 0
then
	if test $dividende -eq 0
	then
		# Le résultat vaut "1" par convention
		test -n "$opt_verb" && echo "0 / 0 = 1 par convention"
		echo 1
		exit 0
	fi

	# Division par 0 !!!
	echo "Division par zéro ???"
	exit 3
fi

# Cas particulier
test $dividende -eq 0 && echo 0 && exit 0	# Le "exit" est facultatif

# Gestion du négatif
test $fl_neg -ne 0 && puts "-"

# Boucle de division
while test $dividende -ne 0 -a \( $opt_prec -ne 0 -o -z "$virg" \)
do
	# Calcul quotient et reste
	quot=`expr $dividende / $diviseur`
	rest=`expr $dividende % $diviseur`

	# Affichage quotient calculé
	puts "$quot"

	# Remplacement dividende par reste * 10
	dividende=`expr $rest \* 10`

	# Si reste non nul
	if test $rest -ne 0
	then
		# Si la virgule a été mise
		if test -n "$virg"
		then
			# Décrément précision
			opt_prec=`expr $opt_prec - 1`
		else
			# Affichage virgule si nb décimales demandé non nul
			test $opt_prec -ne 0 && puts "."
			virg="true"
		fi
	fi
done

# Affichage EOL
echo
```

14) Résolution de polynôme du second degré : $Ax^2 + Bx + C = 0$

Ce script résout les équations polynomiale de second degré: $Ax^2 + Bx + C = 0$ (de façon classique).

```
#!/bin/sh
# Programme de résolution de polynome du second degré (Ax2 + Bx + C = 0)
# Usage: prog [-h] [-v] [--] A B C
#	Option "-h": Pour avoir l'aide sur l'utilisation de ce programme
#	Option "-v": Affichage des étapes

# Fonction affichant la façon d'utiliser ce programme
# Paramètres entrée : texte à afficher (facultatif)
# Valeur sortie : aucune
usage()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Affichage des messages particuliers
		test -n "$*" && echo "`basename $0`: $*"

		# Affichage du message d'aide
		echo "Usage: `basename $0` [-h] [-v] [--] A B C"
	)
}

# Fonction de vérification de tous ses arguments numériques
# Paramètres entrée : arguments à vérifier (…)
# Valeur sortie :
#	tous arguments corrects (0)
#	au-moins un argument non-correct (not 0)
numeric()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Boucle sur chaque argument
		for nb in $*
		do
			# Suppression de chaque chiffre et du "." et du signe
			nb=`echo $nb |sed -e "s/[0-9]//g" |sed -e "s/\.//" |sed -e "s/^-//"`

			# Vérification => nb non vide => pas un nombre
			test -n "$nb" && return 1
		done

		# Renvoi OK
		return 0
	)
}

# Fonction réduisant un nombre à son écriture la plus courte
# Paramètres entrée : nombre à réduire (facultatif)
# Valeur sortie : aucune
reduce()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Le nombre est pris dans l'argument s'il existe ou dans stdin
		test -n "$1" && nb="$1" || nb="`cat`"

		# Si le nombre commence par "-"
		if test "`echo $nb |cut -c1`" = "-"
		then
```

```
                        nb=`echo $nb |cut -c2-`
                        fl_neg="true"
                fi

                # Suppression des zéros non significatifs à gauche
                nb=`echo $nb |sed -e "s/^0\{1,\}//g"`

                # Si le nombre est vide ou réduit à "." il vaut 0
                test -z "$nb" -o "$nb" = "." && echo 0 && return

                # Si le nombre est décimal
                if echo $nb |fgrep "." 1>/dev/null
                then
                        # Décomposition du nombre en parties entières et fractionnelles
                        ent=`echo $nb |cut -f1 -d.`
                        frc=`echo $nb |cut -f2 -d.`

                        # Suppression des zéros non significatifs à droite de la fraction
                        frc=`echo $frc |sed -e "s/0\{1,\}$//g"`

                        # Réécriture du nombre
                        nb=${ent:-0}${frc:+".$frc"}
                fi
                echo ${fl_neg:+"-"}$nb
        )
}

# Fonction recherchant la racine carrée d'un nombre
# Paramètres entrée : nombre dont on veut la racine
# Valeur sortie : aucune
racine()
{
        # Création d'un sous-shell pour isoler le contexte
        (
                # Cas particulier (mais ça marche même sans ça)
                if test `expr "$1" = 0` -eq 1 -o `expr "$1" = 1` -eq 1
                then
                        echo $1
                        return 0
                fi

                # Calcul
                echo "sqrt($1)" |bc -l
        )
}

# Programme principal

# Gestion des options (on utilise l'instruction "getopts" plus souple que la commande "getopt")
while getopts hv opt
do
        case $opt in
                h) # Aide demandée
                        usage; exit 0
                        ;;
                v) # Demande volubilité
                        opt_verb="true";;
                *) # Autre option
                        usage; exit 1
        esac
done
```

```
shift `expr $OPTIND - 1`

# Vérification assez d'arguments
test $# -lt 3 && usage "Pas assez d'argument" && exit 1

# Vérifications arguments 1, 2 et 3 sont des nombres
numeric $1 $2 $3
test $? -ne 0 && echo "Une de ces valeurs '$1', '$2' ou '$3' n'est pas un nombre" && exit 2

# Récupération coefficients
A=`reduce $1`
B=`reduce $2`
C=`reduce $3`
test -n "$opt_verb" && echo "Coeff: $A $B $C"

# A nul => Solution = -C/B
if test $A -eq 0
then
        test -n "$opt_verb" && echo "Solution=-$C / $B"
        sol=`echo "-$C / $B" |bc -l |reduce`
        echo "A nul - Une solution classique pour $A $B $C: $sol"
        exit 0
fi

# Calcul du déterminant B2 - 4AC
delta=`echo "$B * $B - 4 * $A * $C" |bc -l |reduce`
test -n "$opt_verb" && echo "Delta = $B * $B - 4 * $A * $C = $delta"

# Delta négatif => Pas de solution réelle
if test `expr $delta \< 0` -eq 1
then
        echo "Delta négatif - Pas de solution réelle pour $A $B $C"
        exit 0
fi

# Delta nul => Une solution -B/2A
if test `expr $delta = 0` -eq 1
then
        test -n "$opt_verb" && echo "Solution=$B / (2 * $A)"
        sol=`echo "$B / ($A * 2)" |bc -l |reduce`
        echo "Une solution réelle pour $A $B $C: $sol"
        exit 0
fi

# Delta positif
racdelta=`racine $delta |reduce`
test -n "$opt_verb" && echo "Racine delta=$racdelta"

# Solution 1
test -n "$opt_verb" && echo "Solution1=(-($B) - $racdelta) / (2 * $A)"
sol=`echo "(-($B) - $racdelta) / ($A * 2)" |bc -l |reduce`
echo "Solution 1: $sol"

# Solution 2
test -n "$opt_verb" && echo "Solution2=(-($B) + $racdelta) / (2 * $A)"
sol=`echo "(-($B) + $racdelta) / ($A * 2)" |bc -l |reduce`
echo "Solution 2: $sol"
```

15) Tours de Hanoï

Ce script résout le puzzle mathématique des tours de Hanoï

```
#!/bin/sh
# Programme de tours de Hanoï
# Usage: prog [-?] nb_pions
#	Option "-?": Pour avoir l'aide sur l'utilisation de ce programme

# Fonction affichant la façon d'utiliser ce programme
# Paramètres entrée : texte à afficher (facultatif)
# Valeur sortie : aucune
usage()
{
	# Création d'un sous-shell pour isoler le contexte
	(
		# Affichage des messages particuliers
		test -n "$*" && echo "`basename $0`: $*"

		# Affichage du message d'aide
		echo "Usage: `basename $0` [-?] nb_pions"
	)
}

# Fonction qui modifie le nb de pions d'une tour +|-
# Paramètres entrée :
#	tour à modifier
#	opération (+|-)
# Valeur sortie : aucune
modif()
{
	# Pas de sous-shell - La fonction doit modifier les variables principales

	# Evaluation de la tour
	case $1 in
		1) # La tour 1 change son nb de pions
			t1=`expr $t1 $2 1`
			;;
		2) # La tour 2 change son nb de pions
			t2=`expr $t2 $2 1`
			;;
		3) # La tour 3 change son nb de pions
			t3=`expr $t3 $2 1`
			;;
	esac
}

# Fonction récursive qui déplace les pions d'une tour x vers une tour y
# Paramètres entrée :
#	nb de pions à déplacer
#	tour de départ
#	tour d'arrivée
#	tour intermédiaire (facultatif)
# Valeur sortie : aucune
deplace()
{
	# Pas de sous-shell - La fonction doit modifier les variables principales

	if test $1 -eq 1
	then
		# Ici on est en fin de récursivité – Il ne reste plus qu'un seul pion à déplacer
		modif $2 -			# La tour de départ perd un pion
```

```
                mvt=`expr $mvt + 1`                     # Le nb de mouvements augmente

                # On affiche le mouvement et on quitte la fonction
                echo "Mvt: $mvt - $2 $3: Etat: $t1 $t2 $t3"
                return
        fi

        # Ici, on est dans la partie intermédiaire du jeu – Il y a encore des pions qui gènent

        # Calcul en local (important sinon bug) du nb de pions restants à bouger
        local nb=`expr $1 - 1`

        # Déplacement récursif des pions restants de la tour de départ vers la tour intermédiaire
        # La tour d'arrivée servira de tour de rangement provisoire
        deplace $nb $2 $4 $3

        # Déplacement du dernier pion de la tour de départ vers la tour d'arrivée
        deplace 1 $2 $3

        # Déplacement récursif des pions restants de la tour intermédiaire vers la tour d'arrivée
        # La tour de départ servira de tour de rangement provisoire
        deplace $nb $4 $3 $2
}

# Programme principal

# Gestion des options (on utilise l'instruction "getopts" plus souple que la commande "getopt")
while getopts ? opt
do
        case $opt in
                \?) # Demande d'aide
                        usage; exit 0
                        ;;
                *) # Option incorrecte
                        usage; exit 1
        esac
done
shift `expr $OPTIND - 1`

# Vérification assez d'arguments
if test $# -lt 1
then
        usage "Pas assez d'argument"; exit 1
fi

# Initialisation tours et compteur de mouvements
t1=$1
t2=0
t3=0
mvt=0

# Affichage position de départ
echo "Départ: $t1 $t2 $t3"

# Lancement mouvement de la tour 1 vers la tour 3 en utilisant la tour 2 comme intermédiaire
deplace $1 1 3 2
```

INDEX

&

&& 34

.

. 49

|

|| 34

A

affichage (echo) 11
affichage (printf) 60
alternative complexe 35
alternative simple 34
apostrophe 28
arguments 24
arithmétique (let) 51
awk 57

B

backslashe 28
basename 61
bc 62
booléens 29
boucle (retour) 39
boucle (sortie) 39
boucle sur condition 37
boucle sur liste 38
branchement multiple 36
break 39

C

calcul (bc) 62
calcul mathématique (expr) 55
caller 48
canaux (exec) 52
case…esac 36
commande (false) 33
commande (read) 33
commande (test) 30
commande (true) 33
commande awk 57
commande basename 61
commande bc 62
commande cut 58
commande dirname 61
commande echo 11, 64
commande egrep 57
commande expr 55, 56
commande fgrep 57
commande getopt 62
commande grep 57
commande printf 60
commande sed 59
commande seq 60
commande set 11
commande sort 58
commande tput 64
commande tr 59
commande wc 60
commande xargs 61
commentaire 11
comparaison mathématique (expr) 56
compteur (wc) 60
continue 39
conventions 10
cut 58

D

débogueur 11
declare 20
décomposition (basename et dirname) 61
découpage (cut) 58
découpage (xargs) 61
dirname 61
double guillemet 28

E

echo 11, 64
éditeur de flux (sed) 59
egrep 57
eval 51
évaluation (eval) 51
exec 52
exit 40
export 18
expr 56
extraction de chaînes (expr) 56

F

false 33
fgrep 57
fonction (déclaration) 41
fonction (imbrication) 46
fonction (interruption) 43
fonction (paramètres) 42
fonction (portée des variables) 45
fonction (renvoi) 44
fonction (retour) 43
fonction (statut) 43
fonction (trace) 48
for… do…done 38

G

gestion d'écran 64
getopt 62
getopts 53
grep 57

I

if…then…else…fi 35
IFS 26
impératif 10
inclusion 49
instruction (&&) 34
instruction (.) 49
instruction (||) 34
instruction (break) 39

instruction (caller) 48
instruction (case…esac) 36
instruction (continue) 39
instruction (declare) 20
instruction (exit) 40
instruction (export) 18
instruction (for… do…done) 38
instruction (if…then…else…fi) 35
instruction (local) 45
instruction (read) 17
instruction (readonly) 17
instruction (return) 43
instruction (select… do…done) 40
instruction (set) 26
instruction (shift) 25
instruction (source) 49
instruction (trap) 50
instruction (typeset) 20
instruction (unset) 18
instruction (until do…done) 37
instruction (while do…done) 37
instruction eval 51
instruction exec 52
instruction getopts 53
instruction let 51
interpréteur 12

L
let 51
local 45

M
menu (générateur) 40
méta caractères (apostrophe) 28
méta caractères (backslashe) 28
méta caractères (double guillemet) 28
méta caractères (neutralisation) 27
méta caractères (rappel) 27

N
neutralisation 27

O
options (getopt) 62
options (getopts) 53

P
paramètres 24
paramètres (décalage) 25
paramètres (ré affectation) 26
parenthèses 45
programme (interruption) 40
programme (retour) 40
programme (statut) 40
protection 50

R
read 17, 33
readonly 17
recherche (awk) 57
recherche (egrep) 57
recherche (fgrep) 57
recherche (grep) 57
return 43
rigidité 10

S
sed 59
select… do…done 40
seq 60
séquence (seq) 60
set 11, 26
shift 25
signaux (protection) 50
sort 58
source 49
sous-exécution 22
statut 29
structures de contrôles 34

T
tableaux (affectation) 14
test 30
test (chaînes) 31
test (connecteurs) 32
test (fichiers) 30, 31
test (numériques) 32
tput 64
tr 59
transposition (tr) 59
trap 50
tri (sort) 58
true 33
typeset 20

U
unset 18
until do…done 37

V
variables (affectation) 13
variables (généralités) 13
variables (prédéfinies) 21
variables (protection) 17
variables (saisie) 17
variables (substitution) 15
variables (suppression) 18
variables (typage) 20
variables (visibilité) 18

W
wc 60
while do…done 37

X
xargs 61