Guide avancé d'écriture des scripts Bash

Une exploration en profondeur de l'art de la programmation shell

Mendel Cooper

Guide avancé d'écriture des scripts Bash: Une exploration en profondeur de l'art de la programmation shell

Mendel Cooper

5.3

Publié le 11 mai 2008

Résumé

Ce tutoriel ne suppose aucune connaissance de la programmation de scripts, mais permet une progression rapide vers un niveau intermédiaire/avancé d'instructions *tout en se plongeant dans de petites astuces du royaume d'UNIX®*. Il est utile comme livre, comme manuel permettant d'étudier seul, et comme référence et source de connaissance sur les techniques de programmation de scripts. Les exercices et les exemples grandement commentés invitent à une participation active du lecteur avec en tête l'idée que **`la seule façon pour vraiment apprendre la programmation de scripts est d'écrire des scripts`**.

Ce livre est adapté à une utilisation en classe en tant qu'introduction générale aux concepts de la programmation.

La dernière mise à jour de ce document, comme une « archive tar » compressée avec bzip2 incluant à la fois le source SGML et le HTML généré, peut être téléchargée à partir du site personnel de l'auteur. Une *version PDF* est aussi disponible (*site miroir du PDF*). Voir le *journal des modifications* pour un historique des révisions.

Dédicace

Pour Anita, la source de toute magie

Part 1. Introduction 1
1. Pourquoi la programmation Shell ? 2
2. Lancement avec un « #! » 4
2.1. Appeler le script 7
2.2. Exercices préliminaires 8
Part 2. Bases 9
3. Caractères spéciaux 10
4. Introduction aux variables et aux paramètres 28
4.1. Substitution de variable 28
4.2. Affectation de variable 30
4.3. Les variables Bash ne sont pas typées 32
4.4. Types spéciaux de variables 33
5. Guillemets et apostrophes 37
5.1. Placer les variables entre guillemets 37
5.2. Échappement 39
6. Sortie et code de sortie (ou d'état) 44
7. Tests 46
7.1. Constructions de tests 46
7.2. Opérateurs de test de fichiers 52
7.3. Autres opérateurs de comparaison 55
7.4. Tests if/then imbriqués 59
7.5. Tester votre connaissance des tests 60
8. Opérations et sujets en relation 61
8.1. Opérateurs 61
8.2. Constantes numériques 67
Part 3. Après l'approche basique 69
9. Les variables revisitées 70
9.1. Variables internes 70
9.2. Manipuler les chaînes de caractères 87
9.3. Substitution de paramètres 94
9.4. Typer des variables : **declare** ou **typeset** 102
9.5. Références indirectes 105
9.6. $RANDOM : générer un nombre aléatoire 107
9.7. La construction en double parenthèse 117
10. Boucles et branchements 119
10.1. Boucles 119
10.2. Boucles imbriquées 130
10.3. Contrôle de boucles 130
10.4. Tests et branchements 133
11. Substitution de commandes 141
12. Expansion arithmétique 147
13. Récréation 148
Part 4. Commandes 149
14. Commandes internes et intégrées 159
14.1. Commandes de contrôle des jobs 185
15. Filtres externes, programmes et commandes 190
15.1. Commandes de base 190
15.2. Commandes complexes 195
15.3. Commandes de date et d'heure 204
15.4. Commandes d'analyse de texte 208
15.5. Commandes pour les fichiers et l'archivage 229
15.6. Commandes de communications 245
15.7. Commandes de contrôle du terminal 260
15.8. Commandes mathématiques 261
15.9. Commandes diverses 269
16. Commandes système et d'administration 282
16.1. Analyser un script système 310
Part 5. Thèmes avancés 312
17. Expressions rationnelles 313
17.1. Une brève introduction aux expressions rationnelles 313
17.2. Remplacement 316
18. Documents en ligne 318

18.1. Chaînes en ligne 327
19. Redirection d'E/S (entrées/sorties) 330
19.1. Utiliser *exec* 333
19.2. Rediriger les blocs de code 336
19.3. Applications 340
20. Sous-shells 342
21. Shells restreints 347
22. Substitution de processus 349
23. Fonctions 352
23.1. Fonctions complexes et complexité des fonctions 354
23.2. Variables locales 364
23.3. Récursion sans variables locales 367
24. Alias 370
25. Constructeurs de listes 372
26. Tableaux 375
27. `/dev` et `/proc` 399
27.1. `/dev` 399
27.2. `/proc` 400
28. Des Zéros et des Nulls 406
29. Débogage 410
30. Options 419
31. Trucs et astuces 421
32. Écrire des scripts avec style 428
32.1. Feuille de style non officielle d'écriture de scripts 428
33. Divers 431
33.1. Shells et scripts interactifs et non interactifs 431
33.2. Précédence des opérateurs 432
33.3. Scripts d'appel 434
33.4. Tests et comparaisons : alternatives 438
33.5. Un script s'appelant lui-même (récursion) 438
33.6. « Coloriser » des scripts 440
33.7. Optimisations 452
33.8. Astuces assorties 453
33.9. Problèmes de sécurité 463
33.10. Problèmes de portabilité 464
33.11. Scripts sous Windows 464
34. Bash, version 2 et 3 465
34.1. Bash, version 2 465
34.2. Bash, version 3 469
35. Notes finales 472
35.1. Note de l'auteur 472
35.2. À propos de l'auteur 472
35.3. Où trouver de l'aide 472
35.4. Outils utilisés pour produire ce livre 472
35.5. Remerciements 473
35.6. Avis de non-responsabilité 475
Bibliographie 476
A. Contribution de scripts 477
B. Cartes de référence 626
C. Petit guide sur Sed et Awk 631
C.1. Sed 631
C.2. Awk 634
D. Codes de sortie ayant une signification particulière 637
E. Une introduction détaillée sur les redirections d'entrées/sorties 638
F. Options en ligne de commande 640
F.1. Options standards en ligne de commande 640
F.2. Options en ligne de commande de Bash 641
G. Fichiers importants 642
H. Répertoires système importants 643
I. Localisation 645
J. Commandes d'historique 648
K. Un exemple de fichier `.bashrc` 649
L. Convertir des fichiers batch DOS en scripts shell 659

M. Exercices 663
M.1. Analyse de scripts 663
M.2. Écriture de scripts 664
N. Historique des révisions 672
O. Sites miroirs 675
P. Liste de choses à faire 676
Q. Droits d'utilisation 678
R. Copyright 680
S. Table ASCII 682
T. Index 683

Partie Part 1. Introduction

Le shell est un interpréteur de commandes. Plus qu'une simple couche isolante entre le noyau du système d'exploitation et l'utilisateur, il est aussi un langage de programmation puissant. Un programme shell, appelé un *script*, est un outil facile à utiliser pour construire des applications en « regroupant » des appels système, outils, utilitaires et binaires compilés. Virtuellement, le répertoire entier des commandes UNIX, des utilitaires et des outils est disponible à partir d'un script shell. Si ce n'était pas suffisant, les commandes shell internes, telles que les constructions de tests et de boucles, donnent une puissance et une flexibilité supplémentaires aux scripts. Les scripts shell conviennent particulièrement bien pour les tâches d'administration du système et pour d'autres routines répétitives ne réclamant pas les particularités d'un langage de programmation structuré complet.

Chapitre 1. Pourquoi la programmation Shell ?

Aucun langage de programmation n'est parfait. Il n'existe même pas un langage meilleur que d'autre ; 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 de système, même pour ceux qui ne pensent pas avoir à écrire un script un jour. Pensez qu'au démarrage de la machine Linux, des scripts shell du répertoire `/etc/rc.d` sont exécutés pour restaurer la configuration du système et permettre la mise en fonctionnement des services. Une compréhension détaillée de ces scripts de démarrage est importante pour analyser le comportement d'un système, et éventuellement le modifier.

Écrire des scripts shell n'est pas difficile à apprendre car, d'une part, les scripts peuvent être construits par petites sections et, d'autre part, il n'y a qu'un assez petit nombre d'opérateurs et d'options [1] spécifiques au shell à connaître. La syntaxe est simple et directe, similaire à une suite d'appels de différents utilitaires en ligne de commande et il n'existe que peu de « règles » à apprendre. La plupart des petits scripts fonctionnent du premier coup et le débogage, même des plus longs, est assez simple.

Un script shell est une méthode « rapide et sale » pour prototyper une application complexe. Avoir même un sous-ensemble limité de fonctionnalités dans un script shell est souvent une première étape utile lors d'un projet de développement. De cette façon, la structure de l'application peut être testée et les problèmes majeurs trouvés avant d'effectuer le codage final en *C*, *C++*, *Java*, Perl437 ou *Python*.

La programmation shell ramène à la philosophie classique des UNIX, c'est à dire, casser des projets complexes en sous-tâches plus simples et assembler des composants et des utilitaires. Beaucoup considèrent que cette approche de la résolution de problème est meilleure ou, du moins, plus abordable que l'utilisation de langages de nouvelle génération puissamment intégré comme *Perl*, qui essaient de tout faire pour tout le monde mais au prix de vous forcer à changer votre processus de réflexion pour vous adapter à l'outil.

D'après Herbert Mayer, « un langage utile doit comprendre des tableaux, des pointeurs et un mécanisme générique pour construire des structures de données. » Suivant ces critères, les langages des scripts shell ne sont pas « utiles ». Peut-être que si...

Quand ne pas utiliser les scripts shell

- pour des tâches demandant beaucoup de ressources et particulièrement lorsque la rapidité est un facteur (tri, hachage, récursion [2] ...) ;
- pour des procédures impliquant des opérations mathématiques nombreuses et complexes, spécialement pour de l'arithmétique à virgule flottante, des calculs à précision arbitraire ou des nombres complexes (optez plutôt pour le *C++* ou le *FORTRAN* dans ce cas) ;
- pour une portabilité inter-plateformes (utilisez le *C* ou *Java* à la place) ;
- pour des applications complexes où une programmation structurée est nécessaire (typage de variables, prototypage de fonctions, etc.) ;
- pour des applications critiques sur lesquelles vous misez l'avenir de votre société ;
- pour des situations où la *sécurité* est importante, où vous avez besoin de garantir l'intégrité de votre système et de vous protéger contre les intrusions et le vandalisme ;
- pour des projets consistant en de nombreux composants avec des dépendances inter-verrouillées ;
- lorsque des opérations importantes sur des fichiers sont requises (*Bash* est limité à un accès fichier en série, ligne par ligne, ce qui est particulièrement maladroit et inefficace) ;
- si le support natif des tableaux multidimensionnels est nécessaire ;
- si vous avez besoin de structures de données, telles que des listes chaînées ou des arbres ;
- si vous avez besoin de générer ou de manipuler des graphiques ou une interface utilisateur (GUI) ;
- lorsqu'un accès direct au matériel est nécessaire ;
- lorsque vous avez besoin d'accéder à un port, à un socket399 d'entrée/sortie ;

[1] Ils sont connus sous le nom de commandes intégrées159, c'est-à-dire des fonctionnalités internes au shell.
[2] Bien que la récursion *est* possible dans un script shell365, elle tend à être lente et son implémentation est souvent le résultat d'un code sale367.

- si vous avez besoin d'utiliser des bibliothèques ou une interface propriétaire ;
- pour des applications propriétaires, à sources fermées (les sources des shells sont forcément visibles par tout le monde).

Dans l'un des cas ci-dessus, considérez l'utilisation d'un langage de scripts plus puissant, peut-être *Perl*, *Tcl*, *Python*, *Ruby*, voire un langage compilé tel que *C*, *C++* ou *Java*. Même dans ce cas, prototyper l'application avec un script shell peut toujours être une étape utile au développement.

Nous utiliserons Bash, un acronyme pour « Bourne-Again shell » et un calembour sur le désormais classique *Bourne* shell de Stephen Bourne. Bash est devenu un standard *de facto* pour la programmation de scripts sur tous les types d'UNIX. La plupart des principes discutés dans ce livre s'appliquent également à l'écriture de scripts avec d'autres shells tels que le *Korn Shell*, duquel dérivent certaines des fonctionnalités de Bash, [3], le *shell C* et ses variantes (notez que la programmation en *shell C* n'est pas recommandée à cause de certains problèmes inhérents, comme indiqué en octobre 1993 sur un *message Usenet* par Tom Christiansen).

Ce qui suit est un tutoriel sur l'écriture de scripts shell. Il est en grande partie composé d'exemples illustrant différentes fonctionnalités du shell. Les scripts en exemple ont été testés, autant que possible, et certains d'entre eux peuvent même être utiles dans la vraie vie. Le lecteur peut jouer avec le code des exemples dans l'archive des sources (`nom_script.sh` ou `nom_script.bash`), [4] leur donner le droit d'*exécution* (**`chmod u+rx nom_du_script`**) et les exécuter pour voir ce qu'il se passe. Si les sources de l'archive ne sont pas disponibles, alors copier/coller à partir de la version *HTML* ou *pdf*. Sachez que certains scripts présentés ici introduisent des fonctionnalités avant qu'elle ne soient expliquées et que ceci pourrait réclamer du lecteur de lire temporairement plus avant pour des éclaircissements.

Sauf mention contraire, l'*auteur* de ce livre a écrit les scripts d'exemples qui suivent.

[3] Beaucoup de fonctionnalités de *ksh88*, et même quelques unes de la version mise à jour *ksh93*, ont été intégrées à Bash.
[4] Par convention, les scripts shell écrits par l'utilisateur, compatibles avec le shell Bourne, sont nommés avec l'extension `.sh`. Les scripts système, tels que ceux trouvés dans `/etc/rc.d`, ne suivent pas cette nomenclature.

Chapitre 2. Lancement avec un « #! »

La programmation shell est un juke box des années 50...

-- Larry Wall

Dans le cas le plus simple, un script n'est rien de plus qu'une liste de commandes système enregistrées dans un fichier. À tout le moins, cela évite l'effort de retaper cette séquence particulière de commandes à chaque fois qu'elle doit être appelée.

Exemple 2.1. *cleanup* : Un script pour nettoyer les journaux de trace dans /var/log

```
# cleanup
# À exécuter en tant que root, bien sûr.

cd /var/log
cat /dev/null > messages
cat /dev/null > wtmp
echo "Journaux nettoyés."
```

Il n'y a rien d'inhabituel ici, seulement un ensemble de commandes qui pourraient tout aussi bien être appelées l'une après l'autre à partir de la ligne de commande sur la console ou dans une émulation *xterm*. Les avantages de les placer dans un script vont bien au-delà de ne pas avoir à les retaper. Le script devient un *outil* et peut facilement être modifié ou personnalisé pour une application particulière.

Exemple 2.2. *cleanup* : Un script de nettoyage amélioré

```
#!/bin/bash
# En-tête propre d'un script Bash.

# Nettoyage, version 2

# À exécuter en tant que root, bien sûr
# Insérez du code ici pour afficher les messages d'erreur et sortir si
# l'utilisateur n'est pas root.

REP_TRACES=/var/log
# Les variables sont préférées aux valeurs codées en dur.
cd $REP_TRACES

cat /dev/null > messages
cat /dev/null > wtmp


echo "Journaux nettoyés."

exit # La bonne méthode pour "sortir" d'un script.
```

Maintenant, cela commence à ressembler à un vrai script. Mais nous pouvons aller encore plus loin...

Exemple 2.3. *cleanup* : Une version améliorée et généralisée des scripts précédents

```
#!/bin/bash
# Nettoyage, version 3.

#  Attention :
#  -----------
#  Ce script utilise un nombre de fonctionnalités qui seront expliquées bien
#+ après.
#  Après avoir terminé la première moitié de ce livre, il ne devrait plus comporter
#+ de mystère.

REP_TRACES=/var/log
UID_ROOT=0     # Seuls les utilisateurs avec un $UID valant 0 ont les droits de root.
```

```
LIGNES=50      # Nombre de lignes sauvegardées par défaut.
E_XCD=66       # On ne peut pas changer de répertoire?
E_NONROOT=67   # Code de sortie si non root.


# À exécuter en tant que root, bien sûr.
if [ "$UID" -ne "$UID_ROOT" ]
then
  echo "Vous devez être root pour exécuter ce script."
  exit $E_NONROOT
fi

if [ -n "$1" ]
# Teste si un argument est présent en ligne de commande (non vide).
then
  lignes=$1
else
  lignes=$LIGNES # Par défaut, s'il n'est pas spécifié sur la ligne de commande.
fi


#  Stephane Chazelas suggère ce qui suit,
#+ une meilleure façon de vérifier les arguments en ligne de commande,
#+ mais c'est un peu trop avancé à ce stade du tutoriel.
#
#    E_MAUVAISARGS=65  # Argument non numérique (mauvais format de l'argument)
#
#    case "$1" in
#    ""      ) lignes=50;;
#    *[!0-9]*) echo "Usage: `basename $0` Nbre_de_Ligne_a_Garder"; exit
$E_MAUVAISARGS;;
#    *       ) lignes=$1;;
#    esac
#
#* Passer au chapitre "Boucle" pour comprendre tout ceci.


cd $REP_TRACES

if [ `pwd` != "$REP_TRACES" ]  # ou   if [ "$PWD" != "$REP_TRACES" ]
                               # Pas dans /var/log ?
then
  echo "Impossible d'aller dans $REP_TRACES."
  exit $E_XCD
fi  # Double vérification du bon répertoire, pour ne pas poser problème avec le
    # journal de traces.

# bien plus efficace:
#
# cd /var/log || {
#   echo "Impossible d'aller dans le répertoire." >&2
#   exit $E_XCD;
# }




tail -n $lignes messages > mesg.temp # Sauvegarde la dernière section du journal
                                     # de traces.
mv mesg.temp messages                # Devient le nouveau journal de traces.


# cat /dev/null > messages
#* Plus nécessaire, car la méthode ci-dessus est plus sûre.

cat /dev/null > wtmp  #  ': > wtmp' et '> wtmp'  ont le même effet.
echo "Journaux nettoyés."

exit 0
```

```
#  Un code de retour zéro du script indique un succès au shell.
```

Comme vous pouvez ne pas vouloir supprimer toutes les traces système, cette variante du script conserve la dernière section des traces intacte. Vous découvrirez en permanence de nouvelles façons pour affiner des scripts précédemment écrits et améliorer ainsi leur efficacité.

Le *sha-bang* (#!) [1] en en-tête de ce fichier indique à votre système que ce fichier est un ensemble de commandes pour l'interpréteur indiqué. Les caractères #! sont codés sur deux octets [2] et correspondent en fait à un *nombre magique*, un marqueur spécial qui désigne un type de fichier, ou dans ce cas, un script shell exécutable (lancez **man magic** pour plus de détails sur ce thème fascinant). Tout de suite après le *sha-bang* se trouve un *chemin*. C'est le chemin vers le programme qui interprète les commandes de ce script, qu'il soit un shell, un langage de programmation ou un utilitaire. Ensuite, cet interpréteur de commande exécute les commandes du script, en commençant au début (ligne après le #!), en ignorant les commentaires. [3]

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

Chacune des lignes d'en-tête du script ci-dessus appelle un interpréteur de commande différent, qu'il soit `/bin/sh`, le shell par défaut (**bash** dans un système Linux) ou autre chose.

[1] Aussi connu sous le nom de *she-bang* et de *sh-bang*. Ceci est dérivé de la concaténation des caractères # (en anglais, *sharp*) et ! (en anglais, *bang*).

[2] Certains systèmes UNIX (ceux basés sur 4.2BSD) prétendent coder ce nombre magique sur quatre octets, réclamant une espace après le !, **#! /bin/sh**. *D'après Sven Mascheck*, c'est probablement un mythe.

[3] La ligne #! d'un script shell est la première chose que l'interpréteur de commande (**sh** ou **bash**) voit. Comme cette ligne commence avec un #, il sera correctement interprété en tant que commentaire lorsque l'interpréteur de commandes exécutera finalement le script. La ligne a déjà été utilisé pour appeler l'interpréteur de commandes.

En fait, si le script inclut une ligne #! *supplémentaire*, alors **bash** l'interprètera comme un commentaire.

```
#!/bin/bash

echo "Partie 1 du script."
a=1

#!/bin/bash
# Ceci ne lance *pas* un nouveau script.

echo "Partie 2 du script."
echo $a  # Valeur de $a est toujours 1.
```

[4] Ceci permet des tours de passe-passe.

```
#!/bin/rm
# Script se supprimant lui-même.

#  Rien de plus ne semble se produire lorsque vous lancez ceci... si on enlève
#+ le fait que le fichier disparait.

QUOIQUECESOIT=65

echo "Cette ligne ne s'affichera jamais."

exit $QUOIQUECESOIT  # Importe peu. Le script ne se terminera pas ici.
                     # Tester un echo $? après la fin du script.
                     # Vous obtiendrez 0, au lieu de 65.
```

De la même manière, essayer de lancer un fichier `README` avec un **#!/bin/more** après l'avoir rendu exécutable. Le résultat est un fichier de documentation s'affichant lui-même. (Un document en ligne318 utilisant cat est certainement une meilleure alternative -- voir Exemple 18.3, « Message multi-lignes en utilisant *cat* »).

Utiliser **#!/bin/sh**, par défaut Bourne Shell dans la plupart des variantes commerciales d'UNIX, rend le script portable aux machines non-Linux, malheureusement en faisant le sacrifice des fonctionnalités spécifiques à Bash. Le script se conformera néanmoins au standard **s**h de POSIX [5]

Notez que le chemin donné à « sha-bang » doit être correct, sinon un message d'erreur -- habituellement « Command not found » -- sera le seul résultat du lancement du script. [6]

#! peut être omis si le script consiste seulement en un ensemble de commandes système génériques, sans utiliser de directives shell interne. Le second exemple, ci-dessus, requiert le #! initial car la ligne d'affectation des variables, **lignes=50**, utilise une construction spécifique au shell. [7] Notez encore que **#!/bin/sh** appelle l'interpréteur shell par défaut, qui est `/bin/bash` sur une machine Linux.

Astuce

Ce tutoriel encourage une approche modulaire de la construction d'un script. Prenez note et collectionnez des astuces sous forme de « blocs simples » de code pouvant être utiles pour de futurs scripts. À la longue, vous pouvez obtenir une bibliothèque assez étendue de routines bien conçues. Comme exemple, le début du script suivant teste si le script a été appelé avec le bon nombre de paramètres.

```
E_MAUVAIS_ARGS=65
parametres_scripts="-a -h -m -z"
#                   -a = all, -h = help, etc.

if [ $# -ne $Nombre_arguments_attendus ]
then
echo "Usage: `basename $0` $parametres_scripts"
  # `basename $0` est le nom du fichier contenant le script.
  exit $E_MAUVAIS_ARGS
fi
```

De nombreuses fois, vous écrirez un script réalisant une tâche particulière. Le premier script de ce chapitre en est un exemple. Plus tard, il pourrait vous arriver de généraliser le script pour faire d'autres tâches similaires. Remplacer les constantes littérales (« codées en dur ») par des variables est une étape dans cette direction, comme le fait de remplacer les blocs de code répétitifs par des fonctions352.

2.1. Appeler le script

Après avoir écrit le script, vous pouvez l'appeler avec **sh nom_script** [8], ou avec **bash nom_script** (il n'est pas recommandé d'utiliser **sh nom_script** car cela désactive la lecture de `stdin`638 à l'intérieur du script). Il est bien plus aisé de rendre le script directement exécutable avec un chmod.

Soit

chmod 555 nom_script (donne les droits de lecture/exécution à tout le monde) [9]

soit

chmod +rx nom_script (donne les droits de lecture et d'exécution à tout le monde)

chmod u+rx nom_script (donne les droits de lecture et d'exécution seulement à son propriétaire)

Maintenant que vous avez rendu le script exécutable, vous pouvez le tester avec **./nom_script** [10]. S'il commence par une ligne « sha-bang », appeler le script appelle le bon interpréteur de commande.

Enfin, après les tests et le débogage final, vous voudrez certainement le déplacer dans `/usr/local/bin` (en tant que *root*, bien sûr), pour le rendre utilisable par vous et par tous les autres utilisateurs du système. Le script pourra alors être appelé en tapant

[5] **P**ortable **O**perating **S**ystem **I**nterface, an attempt to standardize UNI**X**-like OSes (NdT : interface de systèmes d'exploitation portables, un essai pour standardiser les UNIX). Les spécifications POSIX sont disponibles sur le *site Open Group*.

[6] Pour éviter cette possibilité, un script peut commencer avec une ligne *sha-bang*, #!/bin/env bash309. Ceci pourrait être utile sur les machines UNIX o*bash* n'est pas dans `/bin`

[7] Si *Bash* est votre shell par défaut, alors #! n'est pas nécessaire. Par contre, si vous lancez un script à partir d'un shell différent, comme *tcsh*, alors vous *aurez* besoin de #!.

[8] Attention : appeler un script *Bash* avec **sh nom_script** désactive les extensions spécifiques à Bash, et donc le script peut ne pas fonctionner.

[9] Pour pouvoir être lancé, un script a besoin du droit de *lecture* (*read*) en plus de celui d'exécution, car le shell a besoin de le lire.

[10] Pourquoi ne pas simplement appeler le script avec **nom_script** ? Si le répertoire où vous vous trouvez ($PWD) est déjà celui où se trouve `nom_script`, pourquoi cela ne fonctionne-t'il pas ? Cela échoue parce que, pour des raisons de sécurité, le répertoire courant (`./`) n'est pas inclus par défaut dans le $PATH de l'utilisateur. Il est donc nécessaire d'appeler le script de façon explicite dans le répertoire courant avec **./nom_script**.

simplement **nom_script [ENTER]** sur la ligne de commande.

2.2. Exercices préliminaires

1. Les administrateurs système écrivent souvent des scripts pour automatiser certaines tâches. Donnez quelques exemples où de tels scripts sont utiles.
2. Écrivez un script qui, lors de son exécution, donne la date et l'heure, la liste de tous les utilisateurs connectés et le temps passé depuis le lancement du système (uptime) du système. Enfin, le script doit sauvegarder cette information dans un journal.

Partie Part 2. Bases

Chapitre 3. Caractères spéciaux

Qu'est-ce qui rend un caractère *spécial* ? S'il a une signification en dehors de la *signification littérale*, une meta signification, alors nous lui donnons le nom de *caractère spécial*.

Caractères spéciaux se trouvant dans les scripts et ailleurs

#

Commentaires. Les lignes commençant avec un # (à l'exception de #!) sont des commentaires et ne seront pas *exécutées*.

```
# Cette ligne est un commentaire.
```

Les commentaires peuvent apparaître après la fin d'une commande.

```
echo "Un commentaire va suivre." # Un commentaire ici.
#                                 ^ Notez l'espace blanc devant #
```

Les commentaires peuvent aussi suivre un blanc au début d'une ligne.

```
        # Une tabulation précède ce commentaire.
```

Attention

Un commentaire ne peut pas être suivi d'une commande sur la même ligne. Il n'existe pas de façon de terminer le commentaire pour que le « vrai code » commence sur la même ligne. Utilisez une nouvelle ligne pour la commande suivante.

Note

Bien sûr, un guillemet37 ou un # échappé39 dans une instruction echo ne commence *pas* un commentaire. De la même manière, un # apparaît dans certaines constructions de substitution de paramètres et dans les expressions numériques constantes67.

```
echo "Le # ici ne commence pas un commentaire."
echo 'Le # ici ne commence pas un commentaire.'
echo Le \# ici ne commence pas un commentaire.
echo Le # ici commence un commentaire.

echo ${PATH#*:}       # Substitution de paramètres, pas un commentaire.
echo $(( 2#101011 ))  # Conversion de base, pas un commentaire.

# Merci, S.C.
```

Les caractères standards de guillemet et d'échappement37 (" ' \) échappent le #.

Certaines opérations de filtrage de motif font aussi appel au #.

;

Séparateur de commande [point-virgule]. Permet de placer deux commandes ou plus sur la même ligne.

```
echo bonjour; echo ici


if [ -x "$nomfichier" ]; then  # Notez que "if" et "then" doivent être séparés
                               # par un espace blanc. Pourquoi ?
  echo "Le fichier $nomfichier existe."; cp $nomfichier $nomfichier.sauve
else
  echo "Le fichier $nomfichier est introuvable."; touch $nomfichier
fi; echo "Test du fichier terminé."
```

Notez que le « ; » a parfois besoin d'être *échappé*195.

;;

Fin de ligne dans une sélection par cas case [double point-virgule].

```
case "$variable" in
  abc)  echo "\$variable = abc" ;;
  xyz)  echo "\$variable = xyz" ;;
esac
```

.

Commande « point » [point]. Équivalent au source (voir l'Exemple 14.22, « « Inclure » un fichier de données »). C'est une commande intégrée159 de Bash.

.

« point », comme composant d'un nom de fichier. Lors de l'utilisation de noms de fichiers, un point au début est le préfixe d'un fichier « caché », un fichier que ls ne montre habituellement pas.

```
bash$ touch .fichier_caché
bash$ ls -l
total 10
-rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 donnée1.carnet_d_adresses
-rw-r--r--    1 bozo  bozo      4602 May 25 13:58 donnée1.carnet_d_adresses.bak
-rw-r--r--    1 bozo  bozo       877 Dec 17  2000 boulot.carnet_d_adresse


bash$ ls -al
total 14
drwxrwxr-x    2 bozo  bozo      1024 Aug 29 20:54 ./
drwx------   52 bozo  bozo      3072 Aug 29 20:51 ../
-rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 donnée1.carnet_d_adresses
-rw-r--r--    1 bozo  bozo      4602 May 25 13:58 donnée1.carnet_d_adresses.bak
-rw-r--r--    1 bozo  bozo       877 Dec 17  2000 boulot.carnet_d_adresse
-rw-rw-r--    1 bozo  bozo         0 Aug 29 20:54 .fichier_caché
```

En ce qui concerne les noms des répertoires, *un seul point* représente le répertoire courant et *deux points de suite* indiquent le répertoire parent.

```
bash$ pwd
/home/bozo/projets

bash$ cd .
bash$ pwd
/home/bozo/projets

bash$ cd ..
bash$ pwd
/home/bozo/
```

Le *point* apparaît souvent comme répertoire de destination d'une commande de mouvement de fichiers. Dans ce contexte, cela signifie le *répertoire courant.*.

```
bash$ cp /home/bozo/travail_en_cours/débarras/* .
```

Copiez tous les fichiers du répertoire « débarras » dans $PWD.

.

Filtrage d'un caractère par le « point ». Pour le filtrage de caractères313 au sein d'une expression rationnelle313, un « point » correspond à un seul caractère313.

"

Citation partielle28 [guillemet double]. *"CHAÎNE"* empêche l'interprétation de la plupart des caractères spéciaux présents dans la *CHAÎNE*. Voir le Chapitre 5, Guillemets et apostrophes.

'

Citation totale [guillemet simple]. *'CHAÎNE'* empêche l'interprétation de tous les caractères spéciaux présents dans la *CHAÎNE*. Ces guillemets sont plus puissants que *"CHAÎNE"*. Voir aussi le Chapitre 5, Guillemets et apostrophes.

,

Opérateur virgule. L'*opérateur virgule* [1] relie une suite d'opérations arithmétiques. Toutes sont évaluées, mais seul le résultat de la dernière est renvoyé.

```
let "t2 = ((a = 9, 15 / 3))"  # Initialise "a = 9" et "t2 = 15 / 3".
```

\

Échappement39 [antislash]. Un mécanisme d'échappement pour les caractères seuls.

\X *échappe* le caractère *X*. Cela a pour effet de « mettre *X* entre guillemets », et est équivalent à *'X'*. Le \ peut être utilisé pour mettre " et ' entre guillemets, ce qui permet de les écrire sous forme littérale.

Voir le Chapitre 5, Guillemets et apostrophes pour une explication plus détaillée des caractères échappés.

/

Séparateur dans le chemin d'un fichier [barre oblique]. Sépare les composants d'un nom de fichier (comme dans `/home/bozo/projets/Makefile`).

C'est aussi l'opérateur arithmétique de division.

`

Substitution de commandes141 [guillemet inversé]. La construction **`commande`** rend la sortie de **commande** disponible pour l'affecter à une variable. Connu sous le nom de guillemets inversés141.

:

Commande nul [deux-points]. Il s'agit de l'équivalent shell d'un « NOP » (*no op*, c'est-à-dire « pas d'opération »). Elle peut être considérée comme un synomyme pour la commande intégrée true. La commande « : » est elle-même une commande intégrée159 *Bash* et son état de sortie44 est*true* (0).

```
:
echo $?   # 0
```

Boucle sans fin :

```
while :
do
   operation-1
   operation-2
   ...
   operation-n
done

# Identique à :
#    while true
#    do
#      ...
#    done
```

Sert de bouche-trou dans un test if/then :

```
if condition
then :   # Ne rien faire et continuer
else
   faire_quelque_chose
fi
```

Sert de bouche-trou quand on attend une opération binaire, voir l'Exemple 8.2, « Utiliser des opérations arithmétiques » et les paramètres par défaut.

[1] Un *opérateur* est un agent qui exécute une *opération*. L'exemple habituel est l'opérateur arithmétique, + - * /. Avec Bash, il y a croisement entre les concepts d'*opérateur* et de mots clés159.

```
: ${nom_utilisateur=`whoami`}
# ${nom_utilisateur=`whoami`}   donne une erreur sans les deux-points en tout début
#                               sauf si "nom_utilisateur" est une commande, intégrée
ou non...
```

Sert de bouche-trou quand on attend une commande dans un document en ligne318. Voir l'Exemple 18.10, « Document en ligne « anonyme » ».

Évalue une suite de variables en utilisant la substitution de paramètres (comme dans l'Exemple 9.16, « Utiliser la substitution et les messages d'erreur »).

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
#  Affiche un message d'erreur
#+ si une variable d'environnement (ou plusieurs) n'est pas initialisée.
```

Expansion de variable / remplacement d'une sous-chaîne.

En combinaison avec l'opérateur de redirection >, tronque un fichier à la taille zéro sans modifier ses droits. Crée le fichier s'il n'existait pas auparavant.

```
: > données.xxx   # Fichier "données.xxx" maintenant vide

# Même effet que cat /dev/null >données.xxx
# Néanmoins, cela ne crée pas un nouveau processus, car ":" est une commande
intégrée.
```

Voir aussi l'Exemple 15.15, « Utiliser *tail* pour surveiller le journal des traces système ».

En combinaison avec l'opérateur de redirection >>, elle n'a pas d'effet sur un fichier cible déjà existant (**`: >> nouveau_fichier`**). Crée le fichier s'il n'existait pas.

Note

Cela s'applique aux fichiers réguliers, mais pas aux tubes, aux liens symboliques et à certains fichiers spéciaux.

Peut servir à commencer une ligne de commentaire bien que ce ne soit pas recommandé. Utiliser # pour un commentaire désactive la vérification d'erreur pour le reste de la ligne, donc vous pouvez y mettre pratiquement n'importe quoi. En revanche, ce n'est pas le cas avec :.

```
: Ceci est un commentaire qui génère une erreur, ( if [ $x -eq 3] ).
```

Le « : » sert aussi de séparateur de champ, dans `/etc/passwd` et dans la variable $PATH.

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

!

Inverse le sens d'un test ou d'un état de sortie. L'opérateur ! inverse l'état de sortie44 de la commande à laquelle il est appliqué (voir l'Exemple 6.2, « Inverser une condition en utilisant ! »). Il inverse aussi la signification d'un opérateur de test. Par exemple, cela peut changer le sens d'un *égal* (=) en un *différent* (!=). L'opérateur ! est un mot-clé159 Bash.

Dans un autre contexte, le ! apparaît aussi dans les références indirectes de variable.

Dans un contexte encore différent, à partir de la *ligne de commande*, le ! appelle le *mécanisme d'historique* de Bash (voir l'Annexe J, Commandes d'historique). Notez que ce mécanisme est désactivé dans les scripts.

*

Joker [astérisque]. Le caractère * sert de « joker » pour l'expansion des noms de fichiers dans le remplacement. Utilisé seul, il correspond à tous les noms de fichiers d'un répertoire donné.

```
bash$ echo *
abs-book.sgml add-drive.sh agram.sh alias.sh
```

L'astérisque * représente tout caractère répété plusieurs fois (ou zéro)313 dans une expression rationnelle313.

*

Opérateur arithmétique. Dans le contexte des opérations arithmétiques, * indique la multiplication.

Le double astérisque ** indique l'opérateur exponentiel.

?

Opérateur de test. À l'intérieur de certaines expressions, le ? indique un test pour une condition.

Dans une construction entre parenthèses doubles, ? peut servir en tant qu'élément d'opérateur *à trois arguments* dans le style du C, `?:`.

```
(( var0 = var1<98?9:21 ))
#                ^ ^

# if [ "$var1" -lt 98 ]
# then
#   var0=9
# else
#   var0=21
# fi
```

Dans une expression de substitution de paramètres, le ? teste si une variable a été initialisée.

?

Joker. Le caractère ? sert de joker pour un seul caractère dans l'expansion d'un nom de fichier dans un remplacement, et représente également un caractère314 dans une expression rationnelle étendue.

$

Substitution de variable (contenu d'une variable).

```
var1=5
var2=23skidoo

echo $var1     # 5
echo $var2     # 23skidoo
```

Un $ préfixant un nom de variable donne la *valeur* que contient cette variable.

$

Fin de ligne. Dans une expression rationnelle313, un $ signifie la fin d'une ligne313 de texte.

${}

Substitution de paramètres.

$*, $@

Paramètres de position.

$?

Variable contenant l'état de sortie. La variable $?44 contient l'état de sortie44 d'une commande, d'une fonction352 ou d'un script.

$$

Variable contenant l'identifiant du processus. La variable $$ contient le *PID* [2] du script dans lequel elle apparaît.

()

Groupe de commandes.

```
(a=bonjour; echo $a)
```

Important

Une liste de commandes entre `parenthèses` lance un sous-shell342.

Les variables comprises dans ces parenthèses, à l'intérieur du sous-shell, ne sont pas visibles par le reste du

[2] Un *PID*, ou *identifiant de processus*, est un numéro affecté à un processus en cours d'exécution. Les *PID* des processus sont visibles avec la commande ps.

Définition : un *processus* est un programme en cours d'exécution, quelque fois même appelé un *job*.

script. Le processus parent, le script, ne peut pas lire les variables créées dans le processus fils342, le sous-shell.

```
a=123
( a=321; )

echo "a = $a"   # a = 123
# "a" à l'intérieur des parenthèses agit comme une variable locale.
```

Initialisation de tableaux.

```
Tableau=(element1 element2 element3)
```

{xxx, yyy, zzz, ...}

Expansion d'accolades.

```
cat {fichier1,fichier2,fichier3} > fichier_combiné
# Concatène les fichiers fichier1, fichier2 et fichier3 dans fichier_combiné.


cp fichier22.{txt,sauve}
# Copie "fichier22.txt" dans "fichier22.sauve"
```

Une commande peut agir sur une liste de fichiers séparés par des virgules entre des *accolades* [3]. L'expansion de noms de fichiers (remplacement) s'applique aux fichiers contenus dans les accolades.

Attention

Aucune espace n'est autorisée à l'intérieur des accolades *sauf si* les espaces sont comprises dans des guillemets ou échappés.

```
echo {fichier1,fichier2}\ :{\ A," B",' C'}

fichier1 : A fichier1 : B fichier1 : C fichier2 : A fichier2 : B fichier2
: C
```

{a..z}

Expansion étendue d'accolades.

```
echo {a..z} # a b c d e f g h i j k l m n o p q r s t u v w x y z
# Affiche les caractères entre a et z.

echo {0..3} # 0 1 2 3
# Affiche les caractères entre 0 et 3.
```

La construction *{a..z}* d'expansion étendue d'accolades469 est une fonctionnalité introduite dans la version 3469 de *Bash*.

{}

Bloc de code [accolade]. Aussi connu sous le nom de *groupe en ligne*, cette construction crée une *fonction anonyme* (une fonction sans nom). Néanmoins, contrairement à une fonction352 standard, les variables d'un bloc de code restent visibles par le reste du script.

```
bash$ { local a;
a=123; }
bash: local: can only be used in a function
```

```
a=123
{ a=321; }
echo "a = $a"   # a = 321   (valeur à l'intérieur du bloc de code)

# Merci, S.C.
```

Le bloc de code entouré par des accolades peut utiliser la redirection d'entrées/sorties.

[3] Le shell fait l'*expansion des accolades*. La commande elle-même agit sur le *résultat* de cette expansion.

Exemple 3.1. Blocs de code et redirection d'entrées/sorties

```
#!/bin/bash
# Lit les lignes de /etc/fstab.

Fichier=/etc/fstab

{
read ligne1
read ligne2
} < $Fichier

echo "La première ligne dans $Fichier est :"
echo "$ligne1"
echo
echo "La deuxième ligne dans $Fichier est :"
echo "$ligne2"

exit 0

# Maintenant, comment analysez-vous les champs séparés de chaque ligne ?
# Astuce : utilisez awk, ou...
# ... Hans-Joerg Diers suggère d'utiliser la commande set de Bash.
```

Exemple 3.2. Sauver la sortie d'un bloc de code dans un fichier

```
#!/bin/bash
# rpm-check.sh

#  Recherche une description à partir d'un fichier rpm, et s'il peut être
#+ installé.
#  Sauvegarde la sortie dans un fichier.
#
#  Ce script illustre l'utilisation d'un bloc de code.

SUCCES=0
E_SANSARGS=65

if [ -z "$1" ]
then
        echo "Usage: `basename $0` fichier-rpm"
  exit $E_SANSARGS
fi

{ # Début du bloc de code
  echo
  echo "Description de l'archive :"
  rpm -qpi $1       # Requête pour la description.
  echo
  echo "Contenu de l'archive :"
  rpm -qpl $1       # Requête pour la liste.
  echo
  rpm -i --test $1  # Requête pour savoir si le fichier rpm est installable.
  if [ "$?" -eq $SUCCES ]
  then
    echo "$1 est installable."
  else
    echo "$1 n'est pas installable."
  fi
  echo # Fin du bloc de code
} > "$1.test"       # Redirige la sortie de tout le bloc vers un fichier.

echo "Les résultats du test rpm sont dans le fichier $1.test"

# Voir la page de manuel de rpm pour des explications sur les options.
```

```
exit 0
```

Note

Contrairement à un groupe de commandes entre parenthèses, comme ci-dessus, un bloc de code entouré par des accolades ne sera *pas* lancé dans un sous-shell342[4].

{}

Emplacement pour du texte. Utilisé après xargs `-i`199 (option de *remplacement de chaîne*). Les doubles accolades {} sont un emplacement pour du texte en sortie.

```
ls . | xargs -i -t cp ./{} $1
#            ^^         ^^

# Provient de l'exemple "ex42.sh" (copydir.sh).
```

{} \;

Chemin. Principalement utilisé dans les constructions find. Ce n'est *pas* une commande intégrée159 du shell.

Note

Le « ; » termine l'option `-exec` d'une séquence de commandes **find**. Il a besoin d'être échappé pour que le shell ne l'interprète pas.

[ ]

Test.

Teste46 l'expression entre **[ ]**. Notez que **[** fait partie de la commande *intégrée* test49 (et en est un synonyme), ce n'est *pas* un lien vers la commande externe `/usr/bin/test`.

[[ ]]

Test.

Teste l'expression entre [[ ]]. C'est un mot-clé159 du shell.

Voir les explications sur la structure [[...]]50.

[ ]

Élément d'un tableau.

Accolés au nom d'un tableau375, les crochets indiquent l'indice d'un élément.

```
Tableau[1]=slot_1
echo ${Tableau[1]}
```

[ ]

Ensemble de caractères.

Dans une expression rationnelle313, les crochets désignent un ensemble de caractères313 devant servir de motif (N.d.T : cet ensemble peut être un intervalle).

(( ))

Expansion d'entiers.

Développe et évalue une expression entière entre (( )).

Voir les explications sur la structure ((...)).

[4] Exception : un bloc de code entre accolades dans un tube *peut* être lancé comme sous-shell342.

```
> &> >& >> < <>
ls | { read ligne1; read ligne2; }
#  Erreur. Le bloc de code entre accolades tourne comme un sous-shell,
#+ donc la sortie de "ls" ne peut être passée aux variables de ce bloc.
echo "La première ligne est $ligne1; la seconde ligne est $ligne2"  # Ne fonctionnera pas.
# Merci, S.C.
```

Redirection.

`nom_script >nom_fichier` redirige la sortie de `nom_script` vers le fichier `nom_fichier` et écrase

`nom_fichier` s'il existe déjà.

`commande &>nom_fichier` redirige à la fois `stdout`638 et `stderr` de `commande` vers `nom_fichier`.

`commande >&2` redirige `stdout` de `commande` vers `stderr`.

`nom_script >>nom_fichier` ajoute la sortie de `nom_script` à la fin du fichier `nom_fichier`. Si le fichier n'existe pas déjà, il est créé.

`[i]<>nom_fichier` ouvre le fichier `nom_fichier` en lecture et écriture, et lui affecte le descripteur de fichier330 i. Si `nom_fichier` n'existe pas, il est créé.

Substitution de processus349.

`(commande)>`

`<(commande)`

Dans un autre contexte, les caractères < et > agissent comme des opérateurs de comparaison de chaînes de caractères.

Dans un contexte encore différent, les caractères < et > agissent comme des opérateurs de comparaison d'entiers. Voir aussi l'Exemple 15.9, « Utiliser *expr* ».

<<

Redirection utilisée dans un document en ligne318.

<<<

Redirection utilisée dans une chaîne en ligne327.

<, >

Comparaison ASCII.

```
leg1=carottes
leg2=tomates

if [[ "$leg1" < "$leg2" ]]
then
  echo -n "Le fait que $leg1 précède $leg2 dans le dictionnaire "
  echo "n'a rien à voir avec mes préférences culinaires."
else
  echo "Mais quel type de dictionnaire utilisez-vous?"
fi
```

\<, \>

Délimitation d'un mot314 dans une expression rationnelle313.

```
bash$ grep '\<mot\>' fichier_texte
```

|

Tube. Passe la sortie (`stdout`) de la commande précédente à l'entrée (`stdin`) de la suivante ou au shell. Cette méthode permet de chaîner les commandes ensemble.

```
echo ls -l | sh
#  Passe la sortie de "echo ls -l" au shell
#+ avec le même résultat qu'un simple "ls -l".


cat *.lst | sort | uniq
#  Assemble et trie tous les fichiers ".lst", puis supprime les lignes
#+ dupliquées.
```

Un tube, méthode classique de communication inter-processus, envoie le canal `stdout` d'un processus14 au canal `stdin` d'un autre processus. Dans un cas typique, une commande, comme cat ou echo, envoie un flux de données à un *filtre* (une autre commande) qui opérera des transformations sur ces données. [5]

`cat $nom_fichier1 $nom_fichier2 | grep $mot_recherché`

[5] Même si dans les temps anciens, un *philtre* dénotait une potion dotée de pouvoirs de transformation magiques, un *filtre* UNIX transforme sa cible d'une façon identique. (Le codeur qui arrivera à réaliser un « philtre d'amour » qui fonctionne sur une machine Linux gagnera certainements accolades et honneur.)

La *partie 3 de la FAQ UNIX* contient une note intéressante sur la complexité de l'utilisation de tubes UNIX.

La sortie d'une ou plusieurs commandes peut être envoyée à un script via un tube.

```
#!/bin/bash
# uppercase.sh : Change l'entrée en majuscules.

tr 'a-z' 'A-Z'
#  La plage de lettres doit être entre guillemets pour empêcher que la
#+ génération des noms de fichiers ne se fasse que sur les fichiers à un
#+ caractère.

exit 0
```

Maintenant, envoyons par le tube la sortie de **ls -l** à ce script.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R--    1 BOZO  BOZO       109 APR  7 19:49 1.TXT
-RW-RW-R--    1 BOZO  BOZO       109 APR 14 16:48 2.TXT
-RW-R--R--    1 BOZO  BOZO       725 APR 20 20:56 FICHIER-DONNEES
```

Note

Le canal `stdout` de chaque processus dans un tube doit être lu comme canal `stdin` par le suivant. Si ce n'est pas le cas, le flux de données va se *bloquer* et le tube ne se comportera pas comme il devrait.

```
cat fichier1 fichier2 | ls -l | sort
# La sortie à partir de "cat fichier1 fichier2" disparaît.
```

Un tube tourne en tant que processus fils et ne peut donc modifier les variables du script.

```
variable="valeur_initiale"
echo "nouvelle_valeur" | read variable
echo "variable = $variable"     # variable = valeur_initiale
```

Si une des commandes du tube échoue, l'exécution du tube se termine prématurément. Dans ces conditions, on a un *tube cassé* et on envoie un signal *SIGPIPE*.

>|

Force une redirection (même si l' option noclobber est activée). Ceci va forcer l'écrasement d'un fichier déjà existant.

||

Opérateur logique OU. Dans une structure de test , l'opérateur || a comme valeur de retour 0 (succès) si *l'une des conditions* est vraie.

&

Exécuter la tâche en arrière-plan. Une commande suivie par un & fonctionnera en tâche de fond.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                    sleep 10
```

À l'intérieur d'un script, les commandes et même les boucles peuvent tourner en tâche de fond.

Exemple 3.3. Exécuter une boucle en tâche de fond

```
#!/bin/bash
# background-loop.sh

for i in 1 2 3 4 5 6 7 8 9 10            # Première boucle.
do
```

```
  echo -n "$i "
done & # Exécute cette boucle en tâche de fond.
       # S'exécutera quelques fois après la deuxième boucle.

echo   # Ce 'echo' ne s'affichera pas toujours.

for i in 11 12 13 14 15 16 17 18 19 20   # Deuxième boucle.
do
  echo -n "$i "
done

echo   # Ce 'echo' ne s'affichera pas toujours.

# ======================================================

# La sortie attendue de ce script :
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20

# Mais, quelque fois, vous obtenez :
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (Le deuxième 'echo' ne s'exécute pas. Pourquoi ?)

# Occasionnellement aussi :
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (Le premier 'echo' ne s'exécute pas. Pourquoi ?)

# Et très rarement :
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# La boucle en avant plan s'exécute avant celle en tâche de fond.

exit 0

#  Nasimuddin Ansari suggère d'ajouter    sleep 1
#+ après le   echo -n "$i"   aux lignes 6 et 14,
#+ pour un peu d'amusement.
```

Attention

Une commande exécutée en tâche de fond à l'intérieur d'un script peut faire se suspendre l'exécution, attendant l'appui sur une touche. Heureusement, il est possible d'y remédier.

&&

Opérateur logique ET. Dans une structure de test, l'opérateur && renvoie 0 (succès) si et seulement si les *deux* conditions sont vraies.

-

Option, préfixe. Introduit les options pour les commandes ou les filtres. Sert aussi de préfixe pour un opérateur. Préfixe pour un paramètre par défaut dans la substitution de paramètres.

`COMMANDE -[Option1][Option2][...]`

`ls -al`

`sort -dfu $nom_fichier`

```
if [ $fichier1 -ot $fichier2 ]
then #         ^
  echo "Le fichier $fichier1 est plus ancien que le $fichier2."
fi

if [ "$a" -eq "$b" ]
then #    ^
  echo "$a est égal à $b."
fi
```

```
if [ "$c" -eq 24 -a "$d" -eq 47 ]
then #    ^             ^
  echo "$c vaut 24 et $d vaut 47."
fi


param2=${param1:-$DEFAULTVAL}
#              ^
```

--

Le *tiret double* `--` est le préfixe des options *longues* pour les commandes.

`sort --ignore-leading-blanks`

Utilisé avec une commande interne Bash159, il signifie la *fin des options* de cette commande spécifique.

Astuce

Ceci donne un moyen simple pour supprimer les fichiers dont *les noms commencent avec un tiret*.

```
bash$ ls -l
-rw-r--r-- 1 bozo bozo 0 Nov 25 12:29 -mauvaisnom


bash$ rm -- -mauvaisnom

bash$ ls -l
total 0
```

Le *double-tiret* est aussi utilisé avec set.

`set -- $variable` (comme dans Exemple 14.18, « Réaffecter les paramètres de position »)

-

Redirection à partir de ou vers `stdin` ou `stdout` [tiret].

```
bash$ cat -
abc
abc

...

Ctl-D
```

Comme attendu, **`cat -`** lit sur `stdin`, dans ce cas la saisie de l'utilisateur au clavier, et envoie vers `stdout`. Mais, est-ce que la redirection des entrées/sorties utilisant - ont une application réelle ?

```
(cd /source/répertoire && tar cf - . ) | (cd /dest/répertoire && tar xpvf -)
# Déplace l'ensemble des fichiers d'un répertoire vers un autre
# [courtoisie d'Alan Cox <a.cox@swansea.ac.uk>, avec une modification mineure]

# 1) cd /source/répertoire
#    Répertoire source, où se trouvent les fichiers à déplacer.
# 2) &&
#    "liste ET": si l'opération 'cd' a fonctionné,
#    alors il exécute la commande suivante.
# 3) tar cf - .
#    L'option 'c' de la commande d'archivage 'tar' crée une nouvelle archive,
#    l'option 'f' (fichier), suivie par '-' désigne stdout comme fichier cible.
#    et place l'archive dans le répertoire courant ('.').
# 4) |
#    Tube...
# 5) ( ... )
#    Un sous-shell.
# 6) cd /dest/répertoire
#    Se déplace dans le répertoire de destination.
```

```
# 7) &&
#    "liste ET", comme ci-dessus.
# 8) tar xpvf -
#    Déballe l'archive ('x'), préserve l'appartenance
#    et les droits des fichiers ('p'),
#    puis envoie de nombreux messages vers stdout ('v'),
#    en lisant les données provenant de stdin
#    ('f' suivi par un '-').
#
#    Notez que 'x' est une commande, et 'p', 'v', 'f'
#    sont des options.
# Ouf !



# Plus élégant, mais équivalent à :
#   cd /source/répertoire
#   tar cf - . | (cd ../dest/répertoire; tar xpvf -)
#
#     A aussi le même effet :
# cp -a /source/répertoire/* /dest/répertoire
#     Ou :
# cp -a /source/répertoire/* /source/répertoire/.[^.]* /dest/répertoire
#     S'il y a des fichiers cachés dans /source/répertoire.
```

```
bunzip2 -c linux-2.6.16.tar.bz2 | tar xvf -
#  --décompresse l'archive--  | --puis la passe à "tar"--
#  Si "tar" n'a pas intégré le correctif de support de "bunzip2",
#+ il faut procéder en deux étapes distinctes avec un tube.
#  Le but de cet exercice est de désarchiver les sources du noyau compressées
#+ avec bzip2.
```

Notez que dans ce contexte le signe « - » n'est pas en lui-même un opérateur Bash, mais plutôt une option reconnue par certains utilitaires UNIX qui écrivent dans `stdout` ou lisent dans `stdin`, tels que **tar**, **cat**, etc.

```
bash$ echo "quoiquecesoit" | cat -
quoiquecesoit
```

Lorsqu'un nom de fichier est attendu, un `-` redirige la sortie vers `stdout` (vous pouvez le rencontrer avec **`tar cf`**), ou accepte une entrée de `stdin`, plutôt que d'un fichier. C'est une méthode pour utiliser un outil principalement destiné à manipuler des fichiers comme filtre dans un tube.

```
bash$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

Tout seul sur la ligne de commande, file échoue avec un message d'erreur.

Ajoutez un « - » pour pouvoir vous en servir. Le shell attend alors une entrée de l'utilisateur.

```
bash$ file -
abc
standard input:              ASCII text



bash$ file -
#!/bin/bash
standard input:              Bourne-Again shell script text executable
```

Maintenant, la commande accepte une entrée de `stdin` et l'analyse.

Le « - » peut être utilisé pour envoyer `stdout` à d'autres commandes via un tube, ce qui permet quelques astuces comme l'ajout de lignes au début d'un fichier454.

Par exemple, vous pouvez utiliser diff pour comparer un fichier avec une *partie* d'un autre fichier :

`grep Linux fichier1 | diff fichier2 -`

Finalement, un exemple réel utilisant `-` avec tar.

Exemple 3.4. Sauvegarde de tous les fichiers modifiés dans les dernières 24 heures

```
#!/bin/bash

#  Sauvegarde dans une archive tar compressée tous les fichiers
#+ du répertoire courant modifiés dans les dernières 24 heures.

FICHIERSAUVE=backup-$(date +%m-%d-%Y)
#                 Intégration de la date dans le nom du fichier de sauvegarde.
#                 Merci pour cette idée, Joshua Tschida.
archive=${1:-$FICHIERSAUVE}
#  Si aucun nom de fichier n'est spécifié sur la ligne de commande,
#+ nous utiliserons par défaut "backup-MM-JJ-AAAA.tar.gz."

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Répertoire $PWD sauvegardé dans un fichier archive \"$archive.tar.gz\"."


#  Stephane Chazelas indique que le code ci-dessus échouera si il existe trop
#+ de fichiers ou si un nom de fichier contient des espaces blancs.

# Il suggère les alternatives suivantes:
# -------------------------------------------------------------------
#   find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#         avec la version GNU de "find".


#   find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
#         portable aux autres UNIX, mais plus lent.
# -------------------------------------------------------------------


exit 0
```

Attention

Les noms de fichiers commençant avec un « - » peuvent poser problème lorsqu'ils sont couplés avec l'opérateur de redirection « - ». Votre script doit détecter de tels fichiers et leur ajouter un préfixe approprié, par exemple `./-NOMFICHIER`, `$PWD/-NOMFICHIER`, ou `$NOMCHEMIN/-NOMFICHIER`.

Il y aura probablement un problème si la valeur x d'une variable commence avec un `-`.

```
var="-n"
echo $var
# A le même effet qu'un "echo -n" et donc n'affiche rien.
```

-

Répertoire précédent. cd - revient au répertoire précédent en utilisant la variable d'environnement $OLDPWD.

Attention

Ne confondez pas « - » utilisé dans ce sens avec l'opérateur de redirection « - » vu précédemment. L'interprétation du « - » dépend du contexte dans lequel il apparaît.

-

Moins. Le signe moins indique l'opération arithmétique.

=

Égal. Opérateur d'affectation.

```
a=28
echo $a   # 28
```

Dans un autre contexte, le signe = est un opérateur de comparaison de chaînes de caractères.

+

Plus. Opérateur arithmétique d'addition.

Dans un autre contexte315, le + est un opérateur d'expression rationnelle.

+

Option. Option pour une commande ou un filtre.

Certaines commandes, intégrées159 ou non, utilisent le + pour activer certaines options et le - pour les désactiver. Dans la substitution de paramètres, le **+** préfixe une autre valeur qu'une variable étend.

%

Modulo. Opérateur arithmétique modulo (reste d'une division entière).

```
let "z = 5 % 3"
echo $z  # 2
```

Dans un autre contexte, le % est un opérateur de reconnaissance de motifs.

~

Répertoire de l'utilisateur [tilde]. Le ~ correspond à la variable interne $HOME. *~bozo* est le répertoire de l'utilisateur bozo et **ls ~bozo** liste son contenu. ~/ est le répertoire de l'utilisateur courant et **ls ~/** liste son contenu.

```
bash$ echo ~bozo
/home/bozo

bash$ echo ~
/home/bozo

bash$ echo ~/
/home/bozo/

bash$ echo ~:
/home/bozo:

bash$ echo ~utilisateur-inexistant
~utilisateur-inexistant
```

~+

Répertoire courant. Correspond à la variable interne $PWD.

~-

Répertoire courant précédent. Correspond à la variable interne $OLDPWD.

=~

correspondance d'une expression rationnelle470. Cet opérateur a été introduit avec la version 3469 de Bash.

^

Début de ligne. Dans une expression rationnelle313, un « ^ » correspond au début d'une ligne313 de texte.

Caractères de contrôle

Modifient le comportement d'un terminal ou de l'affichage d'un texte. Un caractère de contrôle est une combinaison **CONTROL** + **touche** (appuyés simultanément). Un caractère de contrôle peut aussi être écrit en notation *octal* ou *hexadécimal*, après un *échappement*.

Les caractères de contrôle ne sont normalement pas utiles à l'intérieur d'un script.

- **Ctl-A**

  Déplace le curseur au début de la ligne de texte (sur la ligne de commande).

- **Ctrl-B**

  Retour en arrière (*backspace*) non destructif.

- **Ctrl-C**

  Termine un job en avant-plan.

- **Ctrl-D**

  Se déconnecte du shell (similaire à un exit44).

  C'est le caractère **EOF** (End Of File, fin de fichier), qui termine aussi l'entrée de `stdin`.

  Lors de la saisie de texte sur la console ou dans une fenêtre *xterm*, **Ctl-D** efface le caractère sous le curseur. Quand aucun caractère n'est présent, **Ctl-D** vous déconnecte de la session. Dans une fenêtre xterm, ceci a pour effet de fermer la fenêtre.

- **Ctl-E**

  Moves cursor to end of line of text (on the command-line).

- **Ctl-F**

  Moves cursor forward one character position (on the command-line).

- **Ctl-G**

  CLOCHE (bip). Sur quelques anciens terminaux comme les télétypes, ceci fera vraiment sonner une cloche. Dans un *xterm*, cela pourrait sonner.

- **Ctrl-H**

  Supprime le caractère précédent (*Backspace*). Efface les caractères sur lequel le curseur passe en arrière.

```
#!/bin/bash
# Intègre Ctrl-H dans une chaîne de caractères.

a="^H^H"                   # Deux Ctrl-H (backspaces)
                           # Ctrl-V Ctrl-H si vous utilisez vi/vim
echo "abcdefg"             # abcdefg
echo
echo -n "abcdefg$a "       # abcd fg
# Espace à la fin ^             ^ Deux fois backspaces.
echo
echo -n "abcdef$a"         # abcdef
#  Pas d'espace à la fin         ^ Ne fait pas de backspace (pourquoi?).
                           # Les résultats pourraient ne pas être ceux attendus.
echo; echo

# Constantin Hagemeier suggère d'essayer :
# a=$'\010\010'
# a=$'\b\b'
# a=$'\x08\x08'
# mais cela ne modifie pas les résultats.
```

- **Ctrl-I**

  Tabulation horizontale.

- **Ctrl-J**

  Saut à la ligne (*line feed*). Dans un script, cela pourrait aussi s'exprimer en notation octale -- '\012' ou en notation hexadécimal -- '\x0a'.

- **Ctrl-K**

  Tabulation verticale.

  Lors de la saisie de texte sur la console ou dans une fenêtre *xterm*, **Ctl-K** efface les caractères en commençant à partir du curseur jusqu'à la fin de la ligne. Within a script, **Ctl-K** may behave differently, as in Lee Lee Maschmeyer's example, below.

- **Ctrl-L**

  Formfeed (efface l'écran du terminal). Dans un terminal, ceci a le même effet que la commande clear. Une fois envoyé à une imprimante, un **Ctl-L** éjecte la page de papier.

- **Ctrl-M**

  Retour chariot.

```
#!/bin/bash
# Merci, Lee Maschmeyer, pour cet exemple.

read -n 1 -s -p \
  $'Control-M place le curseur au début de cette ligne. Tapez sur Enter. \x0d'
                                  #  Bien sûr, '0d' est l'équivalent en
                                  #+ hexadécimal de Control-M.
echo >&2   #  Le '-s' rend la frappe invisible, donc il est nécessaire d'aller
           #+ explicitement sur la nouvelle ligne.

read -n 1 -s -p $'Control-J place le curseur sur la ligne suivante. \x0a'
           #  '0a' est l'équivalent hexadécimal de Control-J, le retour chariot.
echo >&2

###

read -n 1 -s -p $'Et Control-K\x0bva en bas.'
echo >&2   #  Control-K est la tabulation verticale.

# Un meilleur exemple de l'effet d'une tabulation verticale est :

var=$'\x0aCeci est la ligne du bas\x0bCeci est la ligne du haut\x0a'
echo "$var"
#  Ceci fonctionne de la même façon que l'exemple ci-dessus. Néanmoins :
echo "$var" | col
#  Ceci fait que la fin de ligne droite est plus haute que la gauche.
#  Ceci explique pourquoi nous avons commencé et terminé avec un retour chariot,
#+ pour éviter un écran déséquilibré.

# Comme l'explique Lee Maschmeyer :
# --------------------------
#  Dans le [premier exemple de tabulation verticale]... la tabulation verticale
#+ fait que l'affichage va simplement en-dessous sans retour chariot.
#  Ceci est vrai seulement sur les périphériques, comme la console Linux, qui ne
#+ peuvent pas aller "en arrière".
#  Le vrai but de VT est d'aller directement en haut, et non pas en bas.
#  Cela peut être utilisé sur une imprimante.
#  L'utilitaire col peut être utilisé pour émuler le vrai comportement de VT.

exit 0
```

- **Ctl-N**

  Supprime une ligne de texte rappelée à partir du *tampon de l'historique* [6] (sur la ligne de commande).

- **Ctl-O**

  Lance un *retour à la ligne* (sur la ligne de commande).

- **Ctl-P**

  Rappelle la dernière commande à partir du *tampon historique* (sur la ligne de commande).

- **Ctrl-Q**

  Sort du mode pause du terminal (**XON**).

  Ceci réactive le `stdin` du terminal après qu'il ait été mis en pause.

- **Ctl-R**

  Recherche arrière pour le texte dans le *tampon historique* (sur la ligne de commande).

- **Ctrl-S**

  Pause du terminal (**XOFF**).

  Ceci gèle le `stdin` du terminal (utilisez Ctrl-Q pour en sortir).

[6] Bash stocke une liste de commandes auparavant lancées à partir de la ligne de commande dans un *tampon*, ou dans un espace en mémoire, pour retrouver l'historique à partir des commandes internes159 de l'*historique*.

- **Ctl-T**

  Inverse la position du caractère sous le curseur avec celle du caractère précédent (sur la ligne de commande).

- **Ctrl-U**

  Efface une ligne de l'entrée depuis le début de la ligne jusqu'à la position du curseur. Avec certains paramétrages, **Ctl-U** efface la ligne d'entrée entière, *quelque soit la position du curseur*.

- **Ctl-V**

  Lors d'une saisie de texte, **Ctl-V** permet l'insertion de caractères de contrôle. Par exemple, les deux lignes suivantes sont équivalentes :

  ```
  echo -e '\x0a'
  echo <Ctl-V><Ctl-J>
  ```

  Ctl-V est utile principalement dans un éditeur de texte.

- **Ctl-W**

  Lors de la saisie d'un texte dans une console ou une fenêtre xterm, **Ctl-W** efface les caractères en commençant à partir du curseur et en reculant jusqu'au premier espace blanc. Avec certains paramétrages, **Ctl-W** efface vers l'arrière jusqu'au premier caractère non alphanumérique.

- **Ctl-X**

  Dans certains traitements de texte, *coupe* le texte surligné et le place dans le *presse-papier*.

- **Ctrl-Y**

  Colle le texte à l'endroit où il a été supprimé (avec **Ctrl-K** ou **Ctrl-U**).

- **Ctrl-Z**

  Met en *pause* un job en avant-plan.

  Opération de *substitution* dans certains traitements de texte.

  Caractère **EOF** (end-of-file) dans le système de fichiers MSDOS.

Espace blanc
: **Fonctionne comme un séparateur, séparant les commandes ou les variables.** Les espaces blancs peuvent être des*espaces*, des *tabulations*, des *lignes blanches* ou d'une combinaison de ceux-ci. [7] Dans certains contextes, tels que les affectations de variable421, les espaces blancs ne sont pas permis et sont considérés comme une erreur de syntaxe.

  Les lignes blanches n'ont aucun effet sur l'action d'un script et sont donc utiles pour séparer visuellement les différentes parties.

  La variable $IFS est une variable spéciale définissant pour certaines commandes le séparateur des champs en entrée. Elle a pour valeur par défaut une espace blanche.

  Pour conserver les espaces blancs dans une chaîne ou dans une variable, utilisez des guillemets37.

[7] Un saut de ligne (*newline*) est aussi une espace blanche. Ceci explique pourquoi une *ligne blanche*, consistant seulement d'un saut de ligne, est considérée comme une espace blanche.

Chapitre 4. Introduction aux variables et aux paramètres

Les *variables* sont la façon dont tout langage de programmation ou de script représente les données. Une variable n'est rien de plus qu'un *label*, un nom affecté à un emplacement ou à un ensemble d'emplacements dans la mémoire de l'ordinateur contenant un élément de données.

Les variables apparaissent dans les opérations arithmétiques et dans les manipulations de quantité et dans l'analyse des chaînes.

4.1. Substitution de variable

Le *nom* d'une variable est un point de repère pour sa *valeur*, la donnée qu'elle contient. L'action de référencer sa valeur est appelée *substitution de variable*.

$

Commençons par distinguer soigneusement le *nom* d'une variable de sa *valeur*. Si **`variable1`** est le nom d'une variable, alors **`$variable1`** est une référence à sa *valeur*, la donnée qu'elle contient. [1]

```
bash$ variable=23


bash$ echo variable
variable

bash$ echo $variable
23
```

Les seules fois où une variable apparaît « nue », sans le symbole $ en préfixe, est lorsqu'elle est déclarée ou assignée, lorsqu'elle est *détruite*, lorsqu'elle est exportée, ou dans le cas particulier d'une variable désignant un signal (voir l'Exemple 29.5, « Récupérer la sortie »). Les affectations s'effectuent avec un = (comme dans *`var1=27`*), ou dans une déclaration read ou en début de boucle (*`for var2 in 1 2 3`*).

Entourer une valeur référencée de *guillemets* (" ") n'interfère pas avec la substitution de variable. On appelle cette action les *guillemets partiels* et quelque fois la *protection faible*. Utiliser une apostrophe (' ') provoque une utilisation littérale du nom de la variable et aucune substitution n'est effectuée. C'est ce qu'on appelle les *guillemets complets* ou la « protection forte ». Voir le Chapitre 5, Guillemets et apostrophes pour une discussion détaillée.

Notez que **`$variable`** est en fait une forme simplifiée de **`${variable}`**. Dans les cas où la syntaxe **`$variable`** provoque une erreur, la forme complète peut fonctionner (voir la Section 9.3, « Substitution de paramètres », plus bas).

Exemple 4.1. Affectation de variable et substitution

```
#!/bin/bash
# ex9.sh

# Variables : affectation et substitution

a=375
bonjour=$a

#-----------------------------------------------------------------------
#  Aucune espace de chaque côté du signe =  n'est permis lorsque qu'on initialise
#+ des variables.
#  Que se passe-t'il s'il y a une espace ?

#  "VARIABLE =valeur",
#           ^
#% Le script tente d'exécuter la commande "VARIABLE" avec en argument "=valeur".
```

[1] Techniquement, le *nom* d'une variable est appelé une *lvalue* (*left value*), signifiant qu'il apparaît sur le côté *gauche* d'une instruction d'affectation, comme dans **`VARIABLE=23`**. La *valeur* d'une variable est une *rvalue* (*right value*), signifiant qu'elle apparaît à la *droite* d'une instruction d'affectation, comme dans **`VAR2=$VARIABLE`**.

Un *nom* de variable est, en fait, une *référence*, un *pointeur* vers des emplacements mémoire où les données réelles associées à cette variable sont conservées.

```
#  "VARIABLE= valeur",
#           ^
#% Le script tente d'exécuter la commande "valeur" avec
#+ la variable d'environnement "VARIABLE" initialisée à  "".
#-----------------------------------------------------------------------


echo bonjour    # bonjour
# Pas une référence à une variable, juste la chaine "bonjour"

echo $hello   # 375
#    ^          Ceci *est* une référence de variable.
echo ${hello} # 375
# Aussi une référence de variable, comme ci-dessus.

# Entre guillemets...
echo "$hello"    # 375
echo "${hello}"  # 375

echo

bonjour="A B  C   D"
echo $bonjour   # A B C D
echo "$bonjour" # A B  C   D
# Comme on peut le voir echo $bonjour et echo "$bonjour" donnent des résultats
différents.
# Pourquoi ?
# ============================================================
# Mettre une variable entre guillemets préserve les espaces.
# ============================================================

echo

echo '$bonjour'  # $bonjour
#    ^        ^
#  Désactive (échappe) le référencement de variables à l'aide d'apostrophes simples,
#+ ce qui provoque l'interprétation littérale du "$".

# Notez l'effet des différents types de protection.


bonjour=    # L'affecte d'une valeur nulle
echo "\$bonjour (null value) = $bonjour"
#  Noter qu'affecter une valeur nulle à une variable n'est pas la même chose
#+ que de la "détruire" bien que le résultat final soit le même (voir plus bas).

# ------------------------------------------------------------

#  Il est tout à fait possible d'affecter plusieurs variables sur une même ligne,
#+ si elles sont séparées par des espaces.
#  Attention, cela peut rendre le script difficile à lire et peut ne pas être
portable.

var1=21  var2=22  var3=$V3
echo
echo "var1=$var1   var2=$var2  var3=$var3"

# Peut causer des problèmes avec les vieilles versions de "sh"...

# ------------------------------------------------------------

echo; echo

nombres="un deux trois"
#          ^    ^
autres_nombres="1 2 3"
#                ^ ^
# En cas d'espaces à l'intérieur d'une variable, une protection est nécessaire.
#  autres_nombres=1 2 3                  # Donne un message d'erreur.
```

```
echo "nombres = $nombres"
echo "autres_nombres = $autres_nombres"   # autres_nombres = 1 2 3
#  L'échappement d'une espace blanche fonctionne aussi.
paquet_mixe=2\ ---\ Quoiquecesoit
#            ^    ^ Espace après l'échappement (\).

echo "$paquet_mixe"         # 2 --- Quoiquecesoit

echo; echo

echo "variable_non_initialisee = $variable_non_initialisee"
# Une variable non initialisée a une valeur vide  (pas de valeur du tout).
variable_non_initialisee=   #  Déclaration sans initialisation
                            #+ (même chose que de lui assigner une valeur vide
                            #+ comme ci-dessus)
echo "variable_non_initialisee = $variable_non_initialisee"
                            # Elle a encore une valeur nulle.

uninitialized_variable=23       # On lui affecte une valeur.
unset  variable_non_initialisee  # On la désaffecte.
echo "variable_non_initialisee = $variable_non_initialisee"
                                # Elle a encore une valeur nulle

echo

exit 0
```

Attention

Une variable non initialisée a une valeur « nulle » - pas de valeur assignée du tout (*pas* zéro !). Utiliser une variable avant de lui avoir assigné une valeur est généralement source de problèmes.

Il est néanmoins possible de réaliser des opérations arithmétiques sur une variable non initialisée.

```
echo "$non_initialisee"                                 # (ligne vide)
let "non_initialisee += 5"                              # lui ajoute 5.
echo "$non_initialisee"                                 # 5

#  Conclusion:
#  Une variable non initialisée n'a pas de valeur, néanmoins
#+ elle se comporte comme si elle contenait 0 dans une opération
arithmétique.
#  C'est un comportement non documenté (et probablement non portable)
#+ qui ne devrait pas être utilisé dans un script.
```

Voir aussi Exemple 14.23, « Un script (inutile) qui se charge lui-même ».

4.2. Affectation de variable

=

L'opérateur d'affectation (*pas d'espace avant et après*)

Attention

Ne pas confondre ceci avec = et -eq, qui teste46, au lieu d'affecter !

Cependant, notez que = peut être un *opérateur d'affectation* ou de *test*, suivant le contexte.

Exemple 4.2. Affectation basique de variable

```
#!/bin/bash
# Variables nues
```

```
echo

# Quand une variable est-elle «nue», c'est-à-dire qu'il lui manque le signe '$' ?
# Quand on lui affecte une valeur, plutôt que quand elle est référencée.

# Affectation
a=879
echo "La valeur de \"a\" est $a."

# Affectation utilisant 'let'
let a=16+5
echo "La valeur de \"a\" est maintenant $a."

echo

# Dans une boucle 'for' (en fait, un type d'affectation déguisée) :
echo -n "Les valeurs de \"a\" dans la boucle sont : "
for a in 7 8 9 11
do
  echo -n "$a "
done

echo
echo

# Dans une instruction 'read' (un autre type d'affectation) :
echo -n "Entrez \"a\" "
read a
echo "La valeur de \"a\" est maintenant $a."

echo

exit 0
```

Exemple 4.3. Affectation de variable, basique et élaborée

```
#!/bin/bash

a=23              # Cas simple
echo $a
b=$a
echo $b

# Maintenant, allons un peu plus loin (substitution de commande).

a=`echo Hello!`   # Affecte le résultat de la commande 'echo' à 'a'
echo $a
#  Notez qu'inclure un point d'exclamation à l'intérieur d'une substitution de
#+ commandes ne fonctionnera pas à partir de la ligne de commande,
#+ car ceci déclenche le mécanisme d'historique de Bash.
#  Néanmoins, à l'intérieur d'un script, les fonctions d'historique sont
#+ désactivées.

a=`ls -l`         #  Affecte le résultat de la commande 'ls -l' à 'a'
echo $a           #  Néanmoins, sans guillemets, supprime les tabulations et les
                  #+ retours chariots.
echo
echo "$a"         # La variable entre guillemets préserve les espaces blancs
                  # (voir le chapitre sur les "Guillemets").

exit 0
```

Affectation de variable utilisant le mécanisme *$(...)* (une méthode plus récente que l'apostrophe inverse141). En fait, c'est un type de substitution de commandes141.

```
# provenant de  /etc/rc.d/rc.local
R=$(cat /etc/redhat-release)
arch=$(uname -m)
```

4.3. Les variables Bash ne sont pas typées

À l'inverse de nombreux langages de programmation, Bash ne regroupe pas ses variable par « type ». Essentiellement, les variables bash sont des chaînes de caractères mais, suivant le contexte, Bash autorise des opérations entières et des comparaisons sur ces variables, le facteur décisif étant la seule présence de chiffres dans la variable.

Exemple 4.4. Entier ou chaîne?

```
#!/bin/bash
# int-or-string.sh : Entier ou chaîne de caractères ?

a=2334                   # Entier.
let "a += 1"
echo "a = $a "           # a = 2335
echo                     # Entier, toujours.


b=${a/23/BB}             # Substitue "BB" à "23".
                         # Ceci transforme $b en une chaîne de caractères.
echo "b = $b"            # b = BB35
declare -i b             # Le déclarer comme entier n'aide pas.
echo "b = $b"            # b = BB35

let "b += 1"             # BB35 + 1 =
echo "b = $b"            # b = 1
echo

c=BB34
echo "c = $c"            # c = BB34
d=${c/BB/23}             # Substitue "23" à "BB".
                         # Ceci fait de $d un entier.
echo "d = $d"            # d = 2334
let "d += 1"             # 2334 + 1 =
echo "d = $d"            # d = 2335
echo

# Et à propos des variables nulles ?
e=""
echo "e = $e"            # e =
let "e += 1"             # Les opérations arithmétiques sont-elles permises sur
                         # une variable nulle ?
echo "e = $e"            # e = 1
echo                     # Variable nulle transformée en entier.

# Et concernant les variables non déclarées ?
echo "f = $f"            # f =
let "f += 1"             # Opérations arithmétiques permises ?
echo "f = $f"            # f = 1
echo                     # Variable non déclarée transformée en entier.



# Les variables dans Bash sont essentiellement non typées.

exit 0
```

Les variables non typées ont des bons et des mauvais côtés. Elles permettent plus de flexibilité dans l'écriture des scripts (assez de corde pour se pendre soi même !) et rendent plus aisé le ciselage des lignes de code. Néanmoins, elles permettent à des erreurs subtiles de s'infiltrer dans les programmes et encouragent des habitudes de code bâclé.

Pour alléger la dure tâche du développeur qui doit garder trace des types de variables dans un script, Bash *permet* la déclaration de

variables.

4.4. Types spéciaux de variables

variables locales

variables visibles seulement à l'intérieur d'un bloc de code ou d'une fonction (voir aussi variables locales dans fonctions352)

variables d'environnement

variables qui affectent le comportement du shell et de l'interface utilisateur

Note

Dans un contexte plus général, chaque processus14 a un « environnement », c'est-à-dire un groupe de variables contenant des informations auxquelles pourrait faire référence le processus. En ce sens, le shell se comporte comme n'importe quel processus.

Chaque fois qu'un shell démarre, il crée les variables shell correspondantes à ses propres variables d'environnement. Mettre à jour ou ajouter de nouvelles variables d'environnement force le shell à mettre à jour son environnement, et tous les processus fils (les commandes qu'il exécute) héritent de cet environnement.

Attention

L'espace alloué à l'environnement est limité. Créer trop de variables d'environnement ou une variable d'environnement qui utilise un espace excessif peut causer des problèmes.

```
bash$ eval "`seq 10000 | sed -e 's/.*/export var&=ZZZZZZZZZZZZZZ/'`"

bash$ du
bash: /usr/bin/du: Argument list too long
```

(Merci à Stéphane Chazelas pour la clarification et pour avoir fourni l'exemple ci-dessus.)

Si un script déclare des variables d'environnement, il faut qu'elles soient « exportées », c'est-à-dire, rapportées à l'environnement local du script. C'est la fonction de la commande export .

Note

Un script peut **export**er des variables seulement aux processus14 fils, c'est-à-dire seulement aux commandes ou processus que ce script particulier initie. Un script invoqué depuis la ligne de commande *ne peut pas* ré-exporter des variables à destination de l'environnement de la ligne de commande dont il est issu. *Des processus fils ne peuvent pas réexporter de variables aux processus parents qui les ont fait naître.*

Définition : un *processus enfant* est un sous-processus lancé par un autre processus appelé *parent*.

---

paramètres positionnels

Ce sont les arguments passés aux scripts depuis la ligne de commande - `$0`, `$1`, `$2`, `$3`...

`$0` est le nom du script lui-même, `$1` est le premier argument, `$2` le second, `$3` le troisième, et ainsi de suite. [2] Après `$9`, les arguments doivent être entourés d'accolades, par exemple `${10}`, `${11}`, `${12}`.

Les variables spéciales $* et $@ représentent *tous* les paramètres positionnels.

Exemple 4.5. Paramètres positionnels

```
#!/bin/bash

# Appelez ce script avec au moins 10 paramètres, par exemple
```

[2] Le processus appelant le script affecte le paramètre `$0`. Par convention, ce paramètre est le nom du script. Voir la page man d'**execv**.

```
# ./nom_script 1 2 3 4 5 6 7 8 9 10
MINPARAMS=10

echo

echo "Le nom de ce script est \"$0\"."
# Ajoutez ./ pour le répertoire courant.
echo "Le nom de ce script est \"`basename $0`\"."
# Supprime le chemin du script (voir 'basename')

echo

if [ -n "$1" ]              # La variable testée est entre guillemets.
then
 echo "Le paramètre #1 est $1"  # Nous avons besoin des guillemets pour échapper #
fi

if [ -n "$2" ]
then
 echo "Le paramètre #2 est $2"
fi

if [ -n "$3" ]
then
 echo "Le paramètre #3 est $3"
fi

# ...


if [ -n "${10}" ]  #  Les paramètres supérieures à $9 doivent être compris entre
                   #+ accolades.
then
 echo "Le paramètre #10 est ${10}"
fi

echo "-----------------------------------"
echo "Tous les paramètres de la ligne de commande sont: "$*""

if [ $# -lt "$MINPARAMS" ]
then
  echo
  echo "Ce script a besoin d'au moins $MINPARAMS arguments en ligne de commande!"
fi

echo

exit 0
```

La *notation avec accolades* pour les paramètres positionnels permet de référencer plutôt simplement le *dernier* argument passé à un script sur la ligne de commande. Ceci requiert également le référencement indirect.

```
args=$#           # Nombre d'arguments passés.
dernarg=${!args}
# Ou :      dernarg=${!#}
#           (Merci à Chris Monson)
# Notez que dernarg=${!$#} ne fonctionne pas.
```

Certains scripts peuvent effectuer différentes opérations suivant le nom sous lequel ils sont invoqués. Pour que cela fonctionne, le script a besoin de tester `$0`, le nom sous lequel il a été invoqué. Il doit aussi y avoir des liens symboliques vers tous les différents noms du script. Voir l'Exemple 15.2, « Hello or Good-bye ».

Astuce

Si un script attend un paramètre en ligne de commande mais qu'il est invoqué sans, cela peut causer une *affectation à valeur nulle*, généralement un résultat non désiré. Une façon d'empêcher cela est d'ajouter un caractère

supplémentaire des deux côtés de l'instruction d'affectation utilisant le paramètre positionnel attendu.

```
variable1_=$1_ # Plutôt que variable1_=$1
# Cela préviendra l'erreur, même si le paramètre positionnel est absent.

argument_critique01=$variable1_

# Le caractère supplémentaire peut être retiré plus tard comme ceci.
variable1=${variable1_/_/}
# Il n'y aura d'effets de bord que si $variable1_ commence par un tiret bas.
# Ceci utilise un des patrons de substitution de paramètres discutés plus tard
# (laisser vide le motif de remplacement aboutit à une destruction).

# Une façon plus directe de résoudre ce problème est de simplement tester
#+ si le paramètre postionnel attendu a bien été passé.
if [ -z $1 ]
then
  exit $E_PARAM_POS_MANQUANT
fi


#  Néanmoins, comme l'indique Fabian Kreutz,
#+ la méthode ci-dessus pourrait avoir des effets de bord.
#  Une meilleure méthode est la substitution de paramètres :
#         ${1:-$DefaultVal}
#  Voir la section « Substitution de paramètres »
#+ dans le chapitre « Les variables revisitées ».
```

Exemple 4.6. *wh*, recherche d'un nom de domaine avec *whois*

```
#!/bin/bash
# ex18.sh

# Fait une recherche 'whois nom-domaine' sur l'un des trois serveurs:
#                    ripe.net, cw.net, radb.net

# Placez ce script -- renommé 'wh' -- dans /usr/local/bin

# Requiert les liens symboliques :
# ln -s /usr/local/bin/wh /usr/local/bin/wh-ripe
# ln -s /usr/local/bin/wh /usr/local/bin/wh-cw
# ln -s /usr/local/bin/wh /usr/local/bin/wh-radb

E_SANSARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` [nom-domaine]"
  exit $E_SANSARGS
fi

# Vérifie le nom du script et appelle le bon serveur
case `basename $0` in # Ou :    case ${0##*/} in
    "wh"     ) whois $1@whois.ripe.net;;
    "wh-ripe") whois $1@whois.ripe.net;;
    "wh-radb") whois $1@whois.radb.net;;
    "wh-cw"  ) whois $1@whois.cw.net;;
    *        ) echo "Usage: `basename $0` [nom-domaine]";;
esac

exit $?
```

La commande **shift** réassigne les paramètres positionnels, ce qui a le même effet que de les déplacer vers la gauche d'un rang.

$1 <--- $2, $2 <--- $3, $3 <--- $4, etc.

Le vieux $1 disparaît mais *$0 (le nom du script) ne change pas*. Si vous faites usage d'un grand nombre de paramètres positionnels dans un script, **shift** vous permet d'accèder à ceux au-delà de 10, bien que la notation {accolades} le permette également.

Exemple 4.7. Utiliser *shift*

```
#!/bin/bash
# shft.sh : Utilisation de 'shift' pour voir tous les paramètres de position.

#  Nommez ce script quelque chose comme shft.sh,
#+ et exécutez-le avec quelques paramètres.
#+ Par exemple :
#          sh shft.sh a b c def 23 skidoo

until [ -z "$1" ]  # Jusqu'à ne plus avoir de paramètres...
do
  echo -n "$1 "
  shift
done

echo               # Retour chariot supplémentaire.

exit 0

#  Voir aussi le script echo-params.sh
#+ pour une méthode n'utilisant pas shift pour passer de paramètre en paramètre.
```

La commande **shift** peut prendre un paramètre numérique indiquant le nombre de décalages.

```
#!/bin/bash
# shift-past.sh

shift 3    # Décale de 3 positions.
#  n=3; shift $n
#  a le même effet.

echo "$1"

exit 0


$ sh shift-past.sh 1 2 3 4 5
4

#  Néanmoins, comme Eleni Fragkiadaki l'indique,
#+ tenter un 'shift' après le nombre de paramètres de position
#+ ($#) renvoie un code de sortie de 1, et les paramètres de
#+ position ne changent pas.
#  Cela signifie qu'il est possible d'aboutir à une boucle sans fin...
#  Par exemple :
#      until [ -z "$1" ]
#      do
#         echo -n "$1 "
#         shift 20    #  Si moins de 20 paramètres de position,
#      done           #+ la boucle ne termine jamais !
#
# En cas de doute, ajoutez une vérification...
#           shift 20 || break
#                    ^^^^^^^^
```

Note

La commande **shift** fonctionne d'une façon similaire à la façon de passer des paramètres à une fonction352. Voir l'Exemple 33.15, « Astuce de valeur de retour ».

Chapitre 5. Guillemets et apostrophes

Encadrer une chaîne de caractères avec des apostrophes a pour effet de protéger les caractères spéciaux et d'empêcher leur réinterprétation ou expansion par le shell ou par un script shell. Un caractère est « spécial » si son interprétation a une autre signification que la chaîne elle-même, comme par exemple le caractère joker *.

```
bash$ ls -l [Vv]*
-rw-rw-r--    1 bozo  bozo       324 Apr  2 15:05 VIEWDATA.BAT
 -rw-rw-r--    1 bozo  bozo       507 May  4 14:25 vartrace.sh
 -rw-rw-r--    1 bozo  bozo       539 Apr 14 17:11 viewdata.sh


bash$ ls -l '[Vv]*'
ls: [Vv]*: No such file or directory
```

> Dans le langage de tous les jours, lorsque nous mettons une phrase « entre guillemets », nous la plaçons à part et nous lui donnons une signification spéciale. Dans un script Bash, quand nous écrivons une phrase *entre guillemets*, nous la plaçons à part et nous protégeons sa signification *littérale*.

Certains programmes et utilitaires réinterprètent ou étendent les caractères spéciaux placés dans une chaîne de caractères entre apostrophes. Une fonction importante des apostrophes est donc de protéger du shell les paramètres dans une ligne de commande, tout en laissant au programme appelé la possibilité de les étendre et réinterpréter.

```
bash$ grep '[Pp]remière' *.txt
fichier1.txt:C'est la première ligne de fichier1.txt.
fichier2.txt:C'est la Première ligne de fichier2.txt.
```

Notez que la version sans apostrophes **`grep [Pp]remière *.txt`** fonctionne avec le shell Bash. [1]

Les guillemets peuvent supprimer l'appétit d'echo pour les nouvelles lignes.

```
bash$ echo $(ls -l)
total 8 -rw-rw-r-- 1 bo bo 13 Aug 21 12:57 t.sh -rw-rw-r-- 1 bo bo 78 Aug 21 12:57 u.sh


bash$ echo "$(ls -l)"
total 8
 -rw-rw-r--  1 bo bo  13 Aug 21 12:57 t.sh
 -rw-rw-r--  1 bo bo  78 Aug 21 12:57 u.sh
```

5.1. Placer les variables entre guillemets

Lors du référencement d'une variable, il est généralement conseillé de placer son nom entre guillemets. Cela empêche la réinterprétation de tous les caractères spéciaux à l'intérieur de la chaîne -- le nom de la variable [2] -- sauf $, ` (apostrophe inversée) et \ (échappement). [3] Comme $ reste interprété comme un caractère spécial, cela permet de référencer une variable entre guillemets (*`"$variable"`*), c'est-à-dire de remplacer la variable par sa valeur (voir plus haut l'Exemple 4.1, « Affectation de variable et substitution »).

Vous pouvez utiliser les guillemets pour éviter la séparation de mots. [4] Un argument compris entre guillemets se présente comme un seul mot, même s'il contient des espaces blancs.

```
Liste="un deux trois"

for a in $Liste    #  Divise la variable en plusieurs parties
                   #+ en utilisant les espaces blancs comme séparateur.
do
```

[1] Sauf s'il existe un fichier nommé `first` dans le répertoire courant. Encore une autre raison pour placer des *guillemets* (merci pour cette indication, Harald Koenig).
[2] Cela a aussi des effets de bord sur la *valeur* de la variable (voir ci-dessous)
[4] Ici, la « séparation de mots » signifie que la chaîne de caractères est divisée en un certain nombre d'arguments séparés : un par mot.

```
  echo "$a"
done
# un
# deux
# trois

echo "---"

for a in "$Liste"  # Préserve les espaces blancs dans une seule variable.
do #     ^     ^
  echo "$a"
done
# un deux trois
```

Un exemple plus élaboré :

```
variable1="une variable contenant cinq mots"
COMMANDE Ceci est $variable1    # Exécute COMMANDE avec sept arguments :
# "Ceci" "est" "une" "variable" "contenant" "cinq" "mots"

COMMANDE "Ceci est $variable1"  # Exécute COMMANDE avec un argument :
# "Ceci est une variable contenant cinq mots"


variable2=""    # Vide.

COMMANDE $variable2 $variable2 $variable2
    # Exécute COMMANDE sans arguments.
COMMANDE "$variable2" "$variable2" "$variable2"
    # Exécute COMMANDE avec trois arguments vides.
COMMANDE "$variable2 $variable2 $variable2"
    # Exécute COMMANDE avec un argument (deux espaces).

# Merci, Stéphane Chazelas.
```

Astuce

Mettre les arguments d'une instruction **echo** entre guillemets est nécessaire seulement lorsque la séparation de mots ou la préservation d'espaces blancs pose problème.

Exemple 5.1. Afficher des variables bizarres

```
#!/bin/bash
# weirdvars.sh : Affiche des variables bizarres.

var="'(]\\{}\$\""
echo $var        # '(]\{}$"
echo "$var"      # '(]\{}$"     Ne fait pas de différence.

echo

IFS='\'
echo $var        # '(] {}$"     \ converti en espace. Pourquoi ?
echo "$var"      # '(]\{}$"

# Exemples ci-dessus donnés par Stéphane Chazelas.

exit 0
```

Les apostrophes (' ') opèrent de manière similaire aux guillemets mais ne permettent pas de référencer des variables car la signification spéciale de $ est désactivée. À l'intérieur des apostrophes, *tous* les caractères spéciaux autres que ' sont interprétés littéralement. Vous pouvez considérer que les apostrophes (« citation totale ») sont une façon plus stricte de protéger que les guillemets (« citation partielle »).

Note

Même le caractère d'échappement (\) aboutit à une interprétation littérale avec les apostrophes.

```
echo "Pourquoi ne puis-je pas écrire le caractère ' avec des apostrophes"

echo

# la méthode de contournement:
echo 'Pourquoi ne puis-je pas écrire le caractère '\' "'" ' avec une
apostrophe'
#    |-----------------------------------------|
|--------------------------|
# Deux chaînes chacune avec apostrophes, et, intercalés, des apostrophes.

# Exemple de Stéphane Chazelas.
```

5.2. Échappement

Échapper est une méthode pour mettre entre guillemets un caractère seul. L'échappement (\) précédant un caractère dit au shell d'interpréter le caractère littéralement.

Attention

Avec certaines commandes et utilitaires, tels que echo et sed631, échapper un caractère peut avoir l'effet inverse - cela peut activer un comportement particulier pour ce caractère.

Significations spéciales des caractères échappés

utilisé avec **echo** et **sed**

\n

passe à la ligne

\r

renvoie le curseur en début de ligne

\t

tabulation

\v

tabulation verticale

\b

retour en arrière

\a

« alerte » (sonore ou visuelle)

\0xx

Transcode en octal le caractère dont le code ASCII est *0xx*

Exemple 5.2. Caractères d'échappement

```
#!/bin/bash
# escaped.sh: caractères d'échappement

echo; echo

# Échapper un retour à la ligne.
# ----------------------------
```

```
echo ""

echo "Ceci s'affichera
sur deux lignes."
# Ceci s'affichera
# sur deux lignes.

echo "Ceci s'affichera \
sur une lignes."
# Ceci s'affichera sur une ligne.

echo; echo

echo "============="


echo "\v\v\v\v\v"      # Affiche \v\v\v\v litérallement.
# Utilisez l'option -e avec 'echo' pour afficher les caractères d'échappement.
echo "======================"
echo "TABULATIONS VERTICALES"
echo -e "\v\v\v\v\v"   # Affiche 4 tabulations verticales.
echo "======================"

echo "GUILLEMET DOUBLE"
echo -e "\042"       # Affiche " (guillemet, caractère octal ASCII 42).
echo "======================"

# La construction $'\X' rend l'option -e inutile.
echo; echo "RETOUR CHARIOT ET SON"
echo $'\n'           # Retour chariot.
echo $'\a'           # Alerte (son).

echo "======================"
echo "GUILLEMETS"
#  Les version 2 et ultérieures de Bash permettent l'utilisation de la
#+ construction $'\nnn'.
# Notez que, dans ce cas, '\nnn' est une valeur octale.
echo $'\t \042 \t'   # Guillemet (") entouré par des tabulations.

#  Cela fonctionne aussi avec des valeurs hexadécimales, dans une construction
#+ du type $'\xhhh'.
echo $'\t \x22 \t'  # Guillemet (") entouré par des tabulations.
# Merci, Greg Keraunen, pour nous l'avoir indiqué.
# Les versions précédentes de Bash permettent '\x022'.
echo "======================"
echo


# Affecter des caractères ASCII à une variable.
# ---------------------------------------------
guillemet=$'\042'        # " affecté à une variable.
echo "$guillemet Ceci est un chaîne de caractères mise entre guillemets, $guillemet
" \
  "et ceci reste en dehors des guillemets."

echo

# Concaténer des caractères ASCII dans une variable.
trois_soulignes=$'\137\137\137'  # 137 est le code octal ASCII pour '_'.
echo "$trois_soulignes SOULIGNE $trois_soulignes"

echo

ABC=$'\101\102\103\010'  # 101, 102, 103 sont les codes octals de A, B, C.
echo $ABC

echo; echo
```

```
echappe=$'\033'                     # 033 est le code octal pour l'échappement.
echo "\"echappe\" s'affiche comme $echappe"
#                                     pas de sortie visible.

echo; echo

exit 0
```

Voir l'Exemple 34.1, « Expansion de chaîne de caractères » pour un autre exemple d'extension de chaînes avec la structure **$' '**.

\"

donne au guillemet sa signification littérale

```
echo "Bonjour"                     # Bonjour
echo "\"Bonjour\", a-t-il dit."  # "Bonjour", a-t-il dit.
```

\$

donne au dollar sa signification littérale (un nom de variable suivant un \$ ne sera pas référencé)

```
echo "\$variable01"  # donne $variable01
```

\\

donne à l'antislash sa signification littérale

```
echo "\\"  # donne \

# Alors que...

echo "\"   # Appelle une deuxième invite de la ligne de commande.
           # Dans un script, donne un message d'erreur.
```

Note

Le comportement de \ est dicté par son « auto-échappement », sa mise entre guillemets ou son apparition dans une substitution de commandes141 ou dans un document en ligne318.

```
                        #  Simple échappement et mise entre guillemets
echo \z                 #  z
echo \\z                # \z
echo '\z'               # \z
echo '\\z'              # \\z
echo "\z"               # \z
echo "\\z"              # \z

                        #  Substitution de commandes
echo `echo \z`          #  z
echo `echo \\z`         #  z
echo `echo \\\z`        # \z
echo `echo \\\\z`       # \z
echo `echo \\\\\\z`     # \z
echo `echo \\\\\\\z`    # \\z
echo `echo "\z"`        # \z
echo `echo "\\z"`       # \z

                        # Document en ligne
cat <<EOF
\z
EOF                     # \z

cat <<EOF
\\z
EOF                     # \z
```

```
# Ces exemples ont été donnés par Stéphane Chazelas.
```

Les éléments d'une chaîne de caractères affectée à une variable peuvent être échappés, mais le caractère d'échappement seul ne devrait pas être affecté à une variable.

```
variable=\
echo "$variable"
# Ne fonctionne pas et donne le message d'erreur :
# test.sh: : command not found
# Un échappement seul ne peut être affecté correctement à une variable.
#
#  Ce qui arrive ici est que "\" échappe le saut de ligne et
#+ l'effet est          variable=echo "$variable"
#+                      affectation invalide de variable

variable=\
23skidoo
echo "$variable"        #  23skidoo
                        #  Ça fonctionne car la deuxième ligne est une
affectation
                        #+ valide de variable.

variable=\
#        \^    échappement suivi d'une espace
echo "$variable"        # espace

variable=\\
echo "$variable"        # \

variable=\\\
echo "$variable"
# Ne fonctionnera pas et donne le message d'erreur :
# test.sh: \: command not found
#
#  La première séquence d'échappement échappe la deuxième, mais la troisième
est laissée
#+ seule avec le même résultat que dans le premier exemple ci-dessus.

variable=\\\\
echo "$variable"        # \\
                        # Deuxième et quatrième séquences d'échappement
                        # Ça marche.
```

Échapper une espace peut empêcher la séparation de mots dans une liste d'arguments pour une commande.

```
liste_fichiers="/bin/cat /bin/gzip /bin/more /usr/bin/less /usr/bin/emacs-20.7"
# Liste de fichiers comme argument(s) d'une commande.

# On demande de tout lister, avec deux fichiers en plus.
ls -l /usr/X11R6/bin/xsetroot /sbin/dump $file_list

echo "------------------------------------------------------------------------"

# Qu'arrive-t'il si nous échappons un ensemble d'espaces ?
ls -l /usr/X11R6/bin/xsetroot\ /sbin/dump\ $file_list
# Erreur: les trois premiers fichiers sont concaténés en un seul argument pour 'ls -l'
#        parce que les deux espaces échappés empêchent la séparation des arguments
(mots).
```

L'échappement permet également d'écrire une commande sur plusieurs lignes. Normalement, chaque ligne séparée constitue une commande différente mais un échappement à la fin d'une ligne *échappe le caractère de saut de ligne*, et la séquence de la commande continue sur la ligne suivante.

```
(cd /source/repertoire && tar cf - . ) | \
(cd /dest/repertoire && tar xpvf -)
# Répétant la commande de copie de répertoires d'Alan Cox, mais séparée en deux lignes
```

```
# pour accroître la lisibilité.

# Comme alternative :
tar cf - -C /source/directory . |
tar xpvf - -C /dest/directory
# Voir note ci-dessous.
# (Merci, Stéphane Chazelas.)
```

Note

Si la ligne d'un script termine avec un |, le caractère tube, alors il n'est pas strictement nécessaire de mettre un échappement \. Il est néanmoins considéré comme une bonne pratique de programmation de toujours échapper une ligne de code qui continue sur la ligne suivante.

```
echo "foo
bar"
#foo
#bar

echo

echo 'foo
bar'    # Pas encore de différence.
#foo
#bar

echo

echo foo\
bar     # Saut de ligne échappé.
#foobar

echo

echo "foo\
bar"     #  Pareil ici, car \ toujours interpreté comme un échappement à l'intérieur de
         #  guillemets faibles.
#foobar

echo

echo 'foo\
bar'     # Le caractère d'échappement \ est pris littéralement à cause des guillemets
forts.
#foo\
#bar

# Exemples suggérés par Stéphane Chazelas.
```

Chapitre 6. Sortie et code de sortie (ou d'état)

... il existe des coins sombres dans le shell Bourne et les gens les utilisent tous.

-- Chet Ramey

La commande **exit** est utilisée pour terminer un script, comme dans un programme *C*. Elle peut également renvoyer une valeur, qui sera disponible pour le processus parent du script.

Chaque commande renvoie un *code de sortie* (quelque fois nommé *état de retour*).

Une commande ayant réussi renvoie un 0, alors qu'une ayant échoué renvoie une valeur différente de zéro qui est habituellement interprétable comme un code d'erreur. Les commandes, programmes et utilitaires UNIX bien réalisés, renvoient un code de sortie 0 lors de leur exécution réussie, bien qu'il y ait quelques exceptions.

De même, les fonctions dans un script et le script lui-même renvoient un code de sortie. La dernière commande exécutée dans la fonction ou le script détermine le code de sortie. À l'intérieur d'un script, une commande **`exit` *nnn*** peut être employée pour retourner un code de sortie *nnn* au shell (*nnn* doit être un nombre décimal compris entre 0 et 255).

Note

Lorsqu'un script se termine avec un **exit** sans paramètre, le code de sortie du script est le code de sortie de la dernière commande exécutée dans le script (avant **exit**).

```
#!/bin/bash

COMMANDE_1

. . .

# Sortira avec le code de la dernière commande.
DERNIERE_COMMANDE

exit
```

L'équivalent d'un simple **exit** est **exit $?**, voire même en omettant le **exit**.

```
#!/bin/bash

COMMANDE_1

. . .

# Sortira avec le code de la dernière commande.
DERNIERE_COMMANDE

exit $?
```

```
#!/bin/bash

COMMANDE1

. . .

# Sortira avec le code de la dernière commande.
DERNIERE_COMMANDE
```

`$?` lit le code de sortie de la dernière commande exécutée. Après la fin d'une fonction, `$?` donne le code de sortie de la dernière commande exécutée dans la fonction. C'est la manière de Bash de donner aux fonctions une « valeur de retour ». Après la fin d'un script, un `$?` sur la ligne de commande indique le code de sortie du script, c'est-à-dire celui de la dernière commande exécutée dans le script qui est, par convention, **`0`** en cas de succès ou un entier compris entre 1 et 255 en cas d'erreur.

Exemple 6.1. exit / code de sortie

```
#!/bin/bash

echo bonjour
echo $?    # Code de sortie 0 renvoyé car la commande s'est correctement
           # exécutée.

lskdf      # Commande non reconnue.
echo $?    # Code de sortie différent de zéro car la commande a échoué.

echo

exit 113   # Retournera 113 au shell.
           # Pour vérifier ceci, tapez "echo $?" une fois le script terminé.

#  Par convention, un 'exit 0' indique un succès,
#+ alors qu'un code de sortie différent de zéro indique une erreur ou une
#+ condition anormale.
```

$? est particulièrement utile pour tester le résultat d'une commande dans un script (voir l'Exemple 15.35, « Utiliser *cmp* pour comparer deux fichiers à l'intérieur d'un script. » et l'Exemple 15.20, « Chercher les mots dans une liste pour tester leur validité »).

Note

Le !, qualificateur du *non logique*, inverse le résultat d'un test ou d'une commande et ceci affecte son code de sortie44.

Exemple 6.2. Inverser une condition en utilisant !

```
true    # La commande intégrée "true"
echo "code de sortie de \"true\" = $?"     # 0

! true
echo "code de sortie de \"! true\" = $?"   # 1
# Notez que "!" nécessite une espace entre lui et la commande.
#    !true   renvoie une erreur "command not found"
#
#  L'opérateur '!' préfixant une commande appelle le mécanisme d'historique
de
#+ Bash.

true
!true
# Aucune erreur cette fois, mais pas de négation non plus.
# Il répète simplement la précédente commande (true).

# Merci, Stéphane Chazelas et Kristopher Newsome.
```

Attention

Certains codes de sortie ont une signification spéciale et ne devraient pas être employés par l'utilisateur dans un script.

Chapitre 7. Tests

Tout langage de programmation complet peut tester des conditions et agir suivant le résultat du test. Bash dispose de la commande test49, de différents opérateurs à base de crochets et de parenthèses, ainsi que de la construction**if/then**.

7.1. Constructions de tests

- Une construction **if/then** teste si l'état de la sortie44 d'une liste de commandes vaut 0 (car 0 indique le « succès » suivant les conventions UNIX) et, dans ce cas, exécute une ou plusieurs commandes.
- Il existe une commande dédiée appelée **[** (caractère spécial crochet gauche). C'est un synonyme de **test**, qui est intégré159 pour des raisons d'optimisation. Cette commande considère ses arguments comme des expressions de comparaisons ou comme des tests de fichiers et renvoie un état de sortie correspondant au résultat de la comparaison (0 pour vrai et 1 pour faux).
- Avec la version 2.02, Bash a introduit la *commande de test étendue* [[...]]50, réalisant des comparaisons d'une façon familière aux programmeurs venant d'autres langages. Notez que **[[** est un mot clé159, pas une commande.

  Bash considère **`[[ $a -lt $b ]]`** comme un seul élément, renvoyant un état de sortie.

  Les constructions ((...)) et let ... renvoient aussi un état de sortie de 0 si les expressions arithmétiques qu'elles évaluent se résolvent en une valeur non nulle. Ces constructions d'expansion arithmétique147 peuvent donc être utilisées pour réaliser des comparaisons arithmétiques.

```
let "1<2" renvoie 0 (car "1<2" se transforme
            en "1")
(( 0 && 1 )) renvoie 1 (car "0 && 1" donne "0")
```

- Un **if** peut tester n'importe quelle commande, pas seulement des conditions entourées par des crochets.

```
if cmp a b &> /dev/null  # Supprime la sortie.
then echo "Les fichiers a et b sont identiques."
else echo "Les fichiers a et b sont différents."
fi

# La construction "if-grep" très utile:
# --------------------------------------
if grep -q Bash fichier
then echo "fichier contient au moins une occurrence du mot Bash."
fi

mot=Linux
sequence_lettres=inu
if echo "$mot" | grep -q "$sequence_lettres"
# L'option "-q" de grep supprime l'affichage du résultat.
then
  echo "$sequence_lettres trouvé dans $mot"
else
  echo "$sequence_lettres non trouvé dans $mot"
fi

if COMMANDE_DONT_LA_SORTIE_EST_0_EN_CAS_DE_SUCCES
then echo "Commande réussie."
else echo "Commande échouée."
fi
```

- Une construction **if/then** peut contenir des comparaisons et des tests imbriqués.

```
if echo "Le *if* suivant fait partie de la
comparaison du premier *if*."

  if [[ $comparaison = "integer" ]]
    then (( a < b ))
  else
    [[ $a < $b ]]
  fi

then
  echo '$a est plus petit que $b'
```

```
fi
```

L'explication détaillée du « if-test » provient de Stéphane Chazelas.

Exemple 7.1. Où est le vrai?

```
#!/bin/bash

#  Astuce :
#  Si vous n'êtes pas sûr de la façon dont une certaine condition sera évaluée,
#+ testez-la avec un if.

echo

echo "Test de \"0\""
if [ 0 ]      # zéro
then
  echo "0 est vrai."
else
  echo "0 est faux."
fi            # 0 est vrai.

echo

echo "Test de \"1\""
if [ 1 ]      # un
then
  echo "1 est vrai."
else
  echo "1 est faux."
fi            # 1 est vrai.

echo

echo "Test de \"-1\""
if [ -1 ]     # moins un
then
  echo "-1 est vrai."
else
  echo "-1 est faux."
fi            # -1 est vrai.

echo

echo "Test de \"NULL\""
if [ ]        # NULL (condition vide)
then
  echo "NULL est vrai."
else
  echo "NULL est faux."
fi            # NULL est faux.

echo

echo "Test de \"xyz\""
if [ xyz ]    # chaîne de caractères
then
  echo "Chaîne de caractères au hasard est vrai."
else
  echo "Chaîne de caractères au hasard est faux."
fi            # Chaîne de caractères au hasard est vrai.

echo

echo "Test de \"\$xyz\""
if [ $xyz ]   # Teste si $xyz est nul, mais...
              # c'est seulement une variable non initialisée.
```

```
then
  echo "Une variable non initialisée est vrai."
else
  echo "Une variable non initialisée est faux."
fi            # Une variable non initialisée est faux.

echo

echo "Test de \"-n \$xyz\""
if [ -n "$xyz" ]            # Plus correct.
then
  echo "Une variable non initialisée est vrai."
else
  echo "Une variable non initialisée est faux."
fi            # Une variable non initialisée est faux.

echo


xyz=          # Initialisé, mais à une valeur nulle.

echo "Test de \"-n \$xyz\""
if [ -n "$xyz" ]
then
  echo "Une variable nulle est vrai."
else
  echo "Une variable nulle est faux."
fi            # Une variable nulle est faux.


echo


# Quand "faux" est-il vrai?

echo "Test de \"false\""
if [ "false" ]              #  Il semble que "false" ne soit qu'une chaîne de
                            #+ caractères.
then
  echo "\"false\" est vrai." #+ et il est testé vrai.
else
  echo "\"false\" est faux."
fi                          # "false" est vrai.

echo

echo "Test de \"\$false\""  # De nouveau, une chaîne non initialisée.
if [ "$false" ]
then
  echo "\"\$false\" est vrai."
else
  echo "\"\$false\" est faux."
fi            # "$false" est faux.
              # Maintenant, nous obtenons le résultat attendu.


#  Qu'arriverait-t'il si nous testions la variable non initialisée "$true" ?

echo

exit 0
```

Exercice. Expliquez le comportement de l'Exemple 7.1, « Où est le vrai? », ci-dessus.

```
if [ condition-vraie ]
then
   commande 1
   commande 2
```

```
   ...
else
   # Optionnel (peut être oublié si inutile).
   # Ajoute un code par défaut à exécuter si la condition originale se révèle
   # fausse.
   commande 3
   commande 4
   ...
fi
```

Note

Quand *if* et *then* sont sur la même ligne lors d'un test, un point-virgule doit finir l'expression *if*. *if* et *then* sont des mots clés159. Les mots clés (et les commandes) commençant une expression doivent être terminés avant qu'une nouvelle expression sur la même ligne puisse commencer.

```
if [ -x "$nom_fichier" ]; then
```

Else if et elif

elif

`elif` est une contraction pour else if. Le but est de faire tenir une construction if/then dans une autre construction déjà commencée.

```
if [ condition1 ]
then
   commande1
   commande2
   commande3
elif [ condition2 ]
# Idem que else if
then
   commande4
   commande5
else
   commande_par_defaut
fi
```

La construction **`if test condition-vraie`** est l'exact équivalent de **`if [ condition-vraie ]`**. De cette façon, le crochet gauche, **[**, est un raccourci appelant la commande **test**. Le crochet droit fermant, **]**, ne devrait donc pas être nécessaire dans un test if , néanmoins, les dernières versions de Bash le requièrent.

Note

La commande **test** est une commande interne159 de Bash, permettant de tester les types de fichiers et de comparer des chaînes de caractères. Donc, dans un script Bash, **test** n'appelle *pas* le binaire externe `/usr/bin/test`, qui fait partie du paquet *sh-utils*. De même, **[** n'appelle pas `/usr/bin/[`, qui est un lien vers `/usr/bin/test`.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

Si, pour une raison ou une autre, vous souhaitez utiliser `/usr/bin/test` dans un script Bash, alors indiquez le chemin complet.

Exemple 7.2. Équivalences de *test*, `/usr/bin/test`, [ ], et `/usr/bin/[`

```
#!/bin/bash

echo

if test -z "$1"
then
  echo "Pas d'arguments sur la ligne de commande."
else
  echo "Le premier argument de la ligne de commande est $1."
fi

echo

if /usr/bin/test -z "$1"      # Même résultat que la commande intégrée "test".
#  ^^^^^^^^^^^^              # Spécification du chemin complet.
then
  echo "Pas d'arguments sur la ligne de commande."
else
  echo "Le premier argument de la ligne de commande est $1."
fi

echo

if [ -z "$1" ]                # Identique fonctionnellement au bloc de code.
#   if [ -z "$1"                devrait fonctionner, mais...
#+  Bash répond qu'un crochet fermant manque.
then
  echo "Pas d'arguments sur la ligne de commande."
else
  echo "Le premier argument de la ligne de commande est $1."
fi

echo


if /usr/bin/[ -z "$1" ]       # Encore une fois, fonctionnalité identique à ci-dessus.
# if /usr/bin/[ -z "$1"       # Fonctionne, mais donne un message d'erreur.
#                             # Note :
#                               Ceci a été corrigé dans Bash, version 3.x.
then
  echo "Pas d'arguments sur la ligne de commande."
else
  echo "Le premier argument de la ligne de commande est $1."
fi

echo

exit 0
```

La construction [[ ]] est la version plus souple de [ ] dans Bash. C'est la *commande étendue de test*, venant de *ksh88*.

* * *

Il n'est pas possible de faire de la complétion de noms de fichiers ou de la séparation de mots lorsqu'on se trouve entre [[et]], mais la complétion de paramètres et la substitution de commandes sont disponibles.

```
fichier=/etc/passwd

if [[ -e $fichier ]]
then
  echo "Le fichier de mots de passe existe."
fi
```

Utiliser la construction **[[...]]**, au lieu de **[...]** peut vous permettre d'éviter des erreurs de logique dans vos scripts. Par exemple, les opérateurs &&, ||, < et > fonctionnent à l'intérieur d'un test [[ ]] bien qu'ils génèrent une erreur à l'intérieur d'une construction [ ].

Arithmetic evaluation of octal / hexadecimal constants takes place automatically within a [[...]] construct.

```
# [[ Octal and hexadecimal evaluation ]]
# Thank you, Moritz Gronbach, for pointing this out.


decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [ "$decimal" -eq "$octal" ]
then
  echo "$decimal equals $octal"
else
  echo "$decimal is not equal to $octal"       # 15 is not equal to 017
fi      # Doesn't evaluate within [ single brackets ]!


if [[ "$decimal" -eq "$octal" ]]
then
  echo "$decimal equals $octal"                # 15 equals 017
else
  echo "$decimal is not equal to $octal"
fi      # Evaluates within [[ double brackets ]]!

if [[ "$decimal" -eq "$hex" ]]
then
  echo "$decimal equals $hex"                  # 15 equals 0x0f
else
  echo "$decimal is not equal to $hex"
fi      # [[ $hexadecimal ]] also evaluates!
```

Note

Après un **if**, ni la commande **test** ni les crochets de test ([ ] ou [[ ]]) ne sont nécessaires.

```
repertoire=/home/bozo

if cd "$repertoire" 2>/dev/null; then   # "2>/dev/null" cache les messages
d'erreur
  echo "Je suis maintenant dans $repertoire."
else
  echo "Je ne peux pas aller dans $repertoire."
fi
```

La construction « if COMMANDE » renvoie l'état de sortie de la COMMANDE.

De manière identique, une condition à l'intérieur de crochets de test peut fonctionner sans **if** si elle est utilisée avec une construction en liste372.

```
var1=20
var2=22
[ "$var1" -ne "$var2" ] && echo "$var1 n'est pas égal à $var2"

home=/home/bozo
[ -d "$home" ] || echo "Le répertoire $home n'existe pas."
```

La construction (( )) évalue une expression arithmétique. Si l'expression vaut 0, elle renvoie un code de sortie44 de 1, ou « false ». Une expression différente de 0 renvoie 0, ou « true ». Ceci est en totale contradiction avec l'utilisation des constructions **test** et [ ] évoquées précédemment.

Exemple 7.3. Tests arithmétiques en utilisant (( ))

```
#!/bin/bash
# Tests arithmétiques.

# La construction (( ... )) évalue et teste les expressions numériques.
# Code de sortie opposé à la construction [ ... ] !

(( 0 ))
echo "Le code de sortie de \"(( 0 ))\" est $?."         # 1

(( 1 ))
echo "Le code de sortie de \"(( 1 ))\" est $?."         # 0

(( 5 > 4 ))                                             # vrai
echo "Le code de sortie de \"(( 5 > 4 ))\" est $?."     # 0

(( 5 > 9 ))                                             # faux
echo "Le code de sortie de \"(( 5 > 9 ))\" est $?."     # 1

(( 5 - 5 ))                                             # 0
echo "Le code de sortie de \"(( 5 - 5 ))\" est $?."     # 1

(( 5 / 4 ))                                             # Division OK.
echo "Le code de sortie de \"(( 5 / 4 ))\" est $?."     # 0

(( 1 / 2 ))                                             # Résultat de la division < 1.
echo "Le code de sortie de \"(( 1 / 2 ))\" est $?."     # Arrondie à 0.
                                                         # 1

(( 1 / 0 )) 2>/dev/null                                 # Division par 0... illégale.
#           ^^^^^^^^^^^
echo "Le code de sortie de \"(( 1 / 0 ))\" est $?."     # 1

# Quel effet a "2>/dev/null"?
# Qu'arriverait-t'il s'il était supprimé?
# Essayez de le supprimer, et ré-exécutez le script.

exit 0
```

7.2. Opérateurs de test de fichiers

Renvoie vrai si...

-e
: le fichier existe

-a
: le fichier existe

  Ceci a le même effet que -e mais est « obsolète ». [1] Du coup, son utilisation est déconseillée.

-f
: le fichier est un fichier *ordinaire* (ni un répertoire ni un fichier périphérique)

-s
: le fichier a une taille supérieure à zéro

-d
: le fichier est un répertoire

[1] D'après l'édition 1913 du *Dictionnaire Webster* :

```
Deprecate


To pray against, as an evil;
to seek to avert by prayer;
to desire the removal of;
to seek deliverance from;
to express deep regret for;
to disapprove of strongly.
```

-b
le fichier est un périphérique de type bloc (lecteur de disquettes, lecteur de cdroms, etc.)

-c
le fichier est un périphérique de type caractère (clavier, modem, carte son, etc...)

-p
le fichier est un tube nommé

-h
le fichier est un lien symbolique194

-L
le fichier est un lien symbolique

-S
le fichier est un socket399

-t
le fichier (descripteur330) est associé avec un terminal

Cette option permet de tester dans un script si `stdin` (**`[ -t 0 ]`**) ou `stdout` (**`[ -t 1 ]`**) est un terminal.

-r
le fichier dispose du droit de lecture (*pour l'utilisateur ayant exécuté la commande*)

-w
le fichier dispose du droit d'écriture (*pour l'utilisateur ayant exécuté la commande*)

-x
le fichier dispose du droit d'exécution (*pour l'utilisateur ayant exécuté la commande*)

-g
le fichier dispose du droit set-group-id (sgid) sur ce fichier ou répertoire

Si un répertoire dispose du droit *`sgid`*, alors un fichier créé dans ce répertoire appartient au groupe du répertoire, et pas nécessairement au groupe de l'utilisateur qui a créé ce fichier. Ceci est utile pour un répertoire partagé par un groupe de travail.

-u
le fichier dispose du droit set-user-id (suid)

Un binaire appartenant à *root* et disposant du droit *`set-user-id`* sera lancé avec les privilèges de *root*, même si un utilisateur ordinaire l'utilise. [2] C'est intéressant pour les exécutables (tels que **pppd** et **cdrecord**) qui ont besoin d'accéder au matériel du système. Sans cette option, ces binaires ne pourraient pas être utilisés par un utilisateur ordinaire.

```
-rwsr-xr-t    1 root       178236 Oct  2  2000 /usr/sbin/pppd
```

Un fichier disposant du droit *`suid`* affiche un *s* dans ses droits.

-k
`sticky bit` mis

Habituellement connu sous le nom de *sticky bit*, le droit *save-text-mode* est un droit très particulier pour les fichiers. Si un fichier en dispose, celui-ci sera conservé en mémoire cache, pour un accès plus rapide. [3] Placé sur un répertoire, il restreint les droits d'écriture. Cela ajoute un *t* aux droits du fichier ou du répertoire.

```
drwxrwxrwt    7 root         1024 May 19 21:26 tmp/
```

Si un utilisateur ne possède pas un répertoire qui dispose du droit sticky bit, mais qu'il a le droit d'écriture sur ce répertoire, il peut seulement supprimer les fichiers dont il est le propriétaire. Ceci empêche les utilisateurs de supprimer par inadvertance les fichiers des autres utilisateurs. Un répertoire disposant de ce droit est par exemple `/tmp` (le *propriétaire* du répertoire et *root* peuvent, bien sûr, supprimer ou renommer les fichiers).

-O

[2] Faites attention que les binaires *suid* peuvent apporter des failles de sécurité et que l'option *suid* n'a pas d'effet sur les script shell.
[3] Sur les systèmes UNIX modernes, ce droit n'est plus utilisé sur les fichiers, mais seulement sur les répertoires.

vous êtes le propriétaire du fichier

-G
vous faites partie du groupe propriétaire du fichier

-N
le fichier a été modifié depuis sa dernière lecture

f1 -nt f2
le fichier *f1* est plus récent que le fichier *f2*

f1 -ot f2
le fichier *f1* est plus ancien que le fichier *f2*

f1 -ef f2
le fichier *f1* et le fichier *f2* sont des liens physiques vers le même fichier

!
« not » -- inverse le sens des tests précédents (renvoie vrai si la condition est fausse).

Exemple 7.4. Test de liens cassés

```
#!/bin/bash
# broken-link.sh
# Écrit par Lee bigelow &lt;ligelowbee@yahoo.com&gt;
# Utilisé dans le guide ABS avec sa permission.

#  Un pur script shell pour trouver des liens symboliques morts et les afficher
#+ entre guillemets pour qu'ils puissent être envoyés à xargs et être ainsi mieux
#+ gérés :)
#  eg. sh broken-link.sh /repertoire /autrerepertoire|xargs rm
#
#  Néanmoins, ceci est une meilleure méthode :
#
#  find "repertoire" -type l -print0|\
#  xargs -r0 fichier|\
#  grep "lien symbolique mort"|
#  sed -e 's/^\|: *lienmort.*$/"/g'
#
#+ mais cela ne serait pas du bash pur.
#  Attention au système de fichiers /proc et aux liens circulaires !
##############################################################


#  Si aucun argument n'est passé au script, initialise repertoires au répertoire
#+ courant. Sinon, initialise repertoires aux arguments passés.
##############################################################

[ $# -eq 0 ] && repertoires=`pwd` || repertoires=$@

#  Configure la fonction verifliens pour vérifier si le répertoire en argument
#+ ne contient pas de liens morts et pour les afficher.
#  Si un des éléments du répertoire est un sous-répertoire, alors envoie ce
#+ sous-répertoire à la fonction verifliens.
############################################

verifliens () {
    for element in $1/*; do
    [ -h "$element" -a ! -e "$element" ] && echo \"$element\"
    [ -d "$element" ] && verifliens $element
    # Bien sûr, '-h' teste les liens symboliques, '-d' les répertoires.
    done
}

#  Envoie chaque argument qui a été passé au script à la fonction verifliens
#+ s'il s'agit d'un répertoire validé. Sinon, affiche un message d'erreur et
#+ le message d'usage.
######################
for repertoire in $repertoires; do
```

```
    if [ -d $repertoire ]
        then verifliens $repertoire
        else
            echo "$repertoire n'est pas un répertoire"
            echo "Usage: $0 repertoire1 repertoire2 ..."
    fi
done

exit $?
```

L'Exemple 28.1, « Cacher le cookie jar », l'Exemple 10.7, « Un remplaçant de *grep* pour les fichiers binaires », l'Exemple 10.3, « *Fileinfo* : opérer sur une liste de fichiers contenue dans une variable », l'Exemple 28.3, « Créer un disque ram » et l'Exemple A.1, « *mailformat* : Formater un courrier électronique » illustrent aussi l'utilisation des opérateurs de test de fichiers.

7.3. Autres opérateurs de comparaison

Un opérateur de comparaison *binaire* compare deux variables ou quantités. *Notez que la comparaison entre entiers et chaînes utilisent un différent ensemble d'opérateurs.*

comparaison d'entiers

-eq
: est égal à

```
if [ "$a" -eq "$b" ]
```

-ne
: n'est pas égal à

```
if [ "$a" -ne "$b" ]
```

-gt
: est plus grand que

```
if ["$a" -gt "$b" ]
```

-ge
: est plus grand ou égal à

```
if [ "$a" -ge "$b" ]
```

-lt
: est plus petit que

```
if [ "$a" -lt "$b" ]
```

-le
: est plus petit ou égal à

```
if [ "$a" -le "$b" ]
```

<
: est plus petit que (à l'intérieur de parenthèses doubles)

```
(("$a" < "$b"))
```

<=
: est plus petit ou égal à (à l'intérieur de parenthèses doubles117)

```
(("$a" <= "$b"))
```

>
: est plus grand que (à l'intérieur de parenthèses doubles)

```
(("$a" > "$b"))
```

>=
: est plus grand ou égal à (à l'intérieur de parenthèses doubles)

```
(("$a" >= "$b"))
```

comparaison de chaînes de caractères

=
: est égal à

```
if [ "$a" = "$b" ]
```

==
: est égal à

```
if [ "$a" == "$b" ]
```

: Ceci est un synonyme de =.

Note

L'opérateur de comparaison == se comporte différemment à l'intérieur d'un test à double crochets50 qu'à l'intérieur de crochets simples.

```
[[ $a == z* ]]    # Vrai si $a commence avec un "z" (correspondance de
modèle).

[[ $a == "z*" ]]  # Vrai si $a est égal à z* (correspondance littérale).

[ $a == z* ]      # Correspondance de fichiers et découpage de mots.
[ "$a" == "z*" ]  # Vrai si $a est égal à z* (correspondance littérale).

# Merci, Stéphane Chazelas
```

!=
: n'est pas égal à

```
if [ "$a" != "$b" ]
```

: Cet opérateur utilise la reconnaissance de motifs à l'intérieur de constructions [[...]]50.

<
: est plus petit que, d'après l'ordre alphabétique ASCII

```
if [[ "$a" &lt; "$b" ]]
if [ "$a" \< "$b" ]
```

: Notez que « < » a besoin d'être dans une séquence d'échappement s'il se trouve à l'intérieur de **[ ]**.

>
: est plus grand que, d'après l'ordre alphabétique ASCII

```
if [[ "$a" > "$b" ]]
if [ "$a" \> "$b" ]
```

: Notez que « > » a besoin d'être dans une séquence d'échappement s'il se trouve à l'intérieur de **[ ]**.

: Voir l'Exemple 26.10, « Le tri bulle : *Bubble Sort* » pour une application de cet opérateur de comparaison.

-n
: la chaîne de caractères n'est pas « vide ».

Attention

Attention : Le test **-n** nécessite absolument que la chaîne de caractères soit entre guillemets à l'intérieur des crochets de test. Utiliser une chaîne sans guillemets avec **! -z**, voire simplement la chaîne sans guillemets à l'intérieur des crochets (voir l'Exemple 7.6, « Vérification si une chaîne est *nulle* ») fonctionne habituellement, néanmoins, c'est une pratique peu sûre. Placez *toujours* vos chaînes de caractères à tester entre guillemets. [4]

<varlistentry>-z
la chaîne de caractères est « vide », c'est-à-dire qu'elle a une taille nulle
</varlistentry>

[4] Comme S.C. l'a indiqué, dans un test composé, mettre la variable chaîne de caractères entre quotes pourrait ne pas suffire. **[-n "$chaine" -o "$a" = "$b"]** peut causer une erreur avec certaines versions de Bash si `$chaine` est vide. La façon la plus sûre est d'ajouter un caractère supplémentaire aux variables potentiellement vides, **["x$chaine" != x -o "x$a" = "x$b"]** (les « x » sont annulés).

Exemple 7.5. Comparaisons de nombres et de chaînes de caractères

```
#!/bin/bash

a=4
b=5

#  Ici, "a" et "b" peuvent être traités soit comme des entiers soit comme des
#+ chaînes de caractères.
#  Il y a un peu de flou entre les comparaisons arithmétiques et de chaînes de
#+ caractères car les variables Bash ne sont pas typées fortement.

#  Bash permet des opérations et des comparaisons d'entiers sur des variables
#+ contenant des caractères uniquements numériques.
#  Néanmoins, faites attention.

echo

if [ "$a" -ne "$b" ]
then
  echo "$a n'est pas égal à $b"
  echo "(comparaison arithmétique)"
fi

echo

if [ "$a" != "$b" ]
then
  echo "$a n'est pas égal à $b."
  echo "(comparaison de chaînes de caractères)"
  #     "4"  != "5"
  # ASCII 52 != ASCII 53
fi

# Pour cette instance particulière, "-ne" et "!=" fonctionnent.

echo

exit 0
```

Exemple 7.6. Vérification si une chaîne est *nulle*

```
#!/bin/bash
# str-test.sh: Tester des chaînes nulles et sans guillemets,
# "but not strings and sealing wax, not to mention cabbages and kings..."

# En utilisant   if [ ... ]


# Si une chaîne n'a pas été initialisée, elle n'a pas de valeur définie.
# Cet état est appelé "null" (ce qui n'est pas identique à zéro).

if [ -n $chaine1 ]    # $chaine1 n'est ni déclaré ni initialisé.
then
  echo "La chaîne \"chaine1\" n'est pas nulle."
else
  echo "La chaîne \"chaine1\" est nulle."
fi
# Mauvais résultat.
# Affiche $chaine1 comme non nulle bien qu'elle n'ait pas été initialisée.


echo


# Essayons de nouveau.
```

```
if [ -n "$chaine1" ]  # Cette fois, $chaine1 est entre guillemet.
then
  echo "La chaîne \"chaine1\" n'est pas nulle."
else
  echo "La chaîne \"chaine1\" est nulle."
fi      # Entourer les chaînes avec des crochets de test.


echo


if [ $chaine1 ]       # Cette fois, $chaine1 est seule.
then
  echo "La chaîne \"chaine1\" n'est pas nulle."
else
  echo "La chaîne \"chaine1\" est nulle."
fi
# Ceci fonctionne.
# L'opérateur de test [ ] tout seul détecte si la chaîne est nulle.
# Néanmoins, une bonne pratique serait d'y mettre des guillemets ("$chaine1").
#
# Comme Stéphane Chazelas le dit,
#    if [ $chaine1 ]    a un argument, "]"
#    if [ "$chaine1" ]  a deux arguments, la chaîne "$chaine1" vide et "]"



echo



chaine1=initialisée

if [ $chaine1 ]       # Une fois encore, $chaine1 est seule.
then
  echo "La chaîne \"chaine1\" n'est pas nulle."
else
  echo "La chaîne \"chaine1\" est nulle."
fi
# De nouveau, cela donne le résultat correct.
# Il est toujours préférable de la mettre entre guillemets ("$chaine1"), parce
# que...


chaine1="a = b"

if [ $chaine1 ]       # $chaine1 est de nouveau seule.
then
  echo "La chaîne \"chaine1\" n'est pas nulle."
else
  echo "La chaîne \"chaine1\" est nulle."
fi
# Ne pas mettre "$chaine1" entre guillemets donne un mauvais résultat !

exit 0
# Merci aussi à Florian Wisser pour le "heads up".
```

Exemple 7.7. ***zmore***

```
#!/bin/bash

#Visualiser des fichiers gzip avec 'more'

SANSARGS=65
PASTROUVE=66
NONGZIP=67
```

```
if [ $# -eq 0 ] # même effet que:  if [ -z "$1" ]
# $1 peut exister mais doit être vide:  zmore "" arg2 arg3
then
  echo "Usage: `basename $0` nomfichier" >&2
  # Message d'erreur vers stderr.
  exit $SANSARGS
  # Renvoie 65 comme code de sortie du script (code d'erreur).
fi

nomfichier=$1

if [ ! -f "$nomfichier" ]  #  Mettre $nomfichier entre guillemets permet d'avoir
                           #+ des espaces dans les noms de fichiers.
then
  echo "Fichier $nomfichier introuvable !" >&2
  # Message d'erreur vers stderr.
  exit $PASTROUVE
fi

if [ ${nomfichier##*.} != "gz" ]
# Utilisation de crochets pour la substitution de variables.
then
  echo "Le fichier $1 n'est pas compressé avec gzip !"
  exit $NONGZIP
fi

zcat $1 | more

# Utilise le filtre 'more'.
# Peut se substituer à 'less', si vous le souhaitez.


exit $?   # Le script renvoie le code d'erreur du tube.
#  En fait, "exit $?" n'est pas nécessaire, car le script retournera, pour
#+ chaque cas, le code de sortie de la dernière commande exécutée.
```

comparaison composée

-a
: et logique

  exp1 -a exp2 renvoie vrai si *à la fois* exp1 et exp2 sont vrais.

-o
: ou logique

  exp1 -o exp2 renvoie vrai si soit exp1 *soit* exp2 sont vrais.

Elles sont similaires aux opérateurs de comparaison Bash **&&** et **||**, utilisés à l'intérieur de double crochets50.

```
[[ condition1 && condition2 ]]
```

Les opérateurs **-o** et **-a** fonctionnent avec la commande **test** ou à l'intérieur de simples crochets de test.

```
if [ "$exp1" -a "$exp2" ]
```

Référez-vous à l'Exemple 8.3, « Tests de conditions composées en utilisant && et || », à l'Exemple 26.16, « Simuler un tableau à deux dimensions, puis son test » et à l'Exemple A.31, « Chasse aux spammeurs » pour voir des opérateurs de comparaison composée en action.

7.4. Tests if/then imbriqués

Les tests utilisant les constructions *if/then* peuvent être imbriqués. Le résultat est équivalent à l'utilisation de l'opérateur de comparaison composée **&&** ci-dessus.

```
if [ condition1 ]
```

```
then
  if [ condition2 ]
  then
    faire-quelquechose  # Mais seulement si "condition1" et "condition2" sont valides.
  fi
fi
```

Voir l'Exemple 34.4, « Utiliser des tableaux et autres astuces pour gérer quatre mains aléatoires dans un jeu de cartes » pour des tests de condition *if/then* imbriqués.

7.5. Tester votre connaissance des tests

Le fichier global `xinitrc` est utilisé pour lancer le serveur X. Ce fichier contient un certain nombre de tests *if/then*. Ce qui suit est un extrait d'une « ancienne » version de `xinitrc` (*Red Hat 7.1*, ou très proche).

```
if [ -f $HOME/.Xclients ]; then
  exec $HOME/.Xclients
elif [ -f /etc/X11/xinit/Xclients ]; then
  exec /etc/X11/xinit/Xclients
else
     # En cas de soucis. Bien que nous ne devrions jamais arriver ici (nous
     # apportons un code de secours pour les clients X), cela ne gêne pas.
     xclock -geometry 100x100-5+5 &
     xterm -geometry 80x50-50+150 &
     if [ -f /usr/bin/netscape -a -f /usr/share/doc/HTML/index.html ]; then
            netscape /usr/share/doc/HTML/index.html &
     fi
fi
```

Expliquez les constructions de *test* dans l'extrait ci-dessus, puis examinez une version à jour du fichier, `/etc/X11/xinit/xinitrc`, et analysez les constructions de test *if/then*. Vous pouvez avoir besoin de vous référer aux discussions sur grep, sed631 et les expressions rationnelles313.

Chapitre 8. Opérations et sujets en relation

8.1. Opérateurs

affectation

affectation de variable
: Initialiser ou changer la valeur d'une variable

=
: Opérateur d'affectation à buts multiples, qui fonctionne à la fois pour les affectations arithmétiques et de chaînes de caractères.

```
var=27
categorie=mineraux  # Pas d'espaces permis après le "=".
```

Attention

Ne confondez pas l'« opérateur d'affectation = » avec l'opérateur de test =.

```
#    = comme opérateur de test

if [ "$chaine1" = "$chaine2" ]
then
   commande
fi

# if [ "X$chaine1" = "X$chaine2" ] est plus sûr,
# pour empêcher un message d'erreur si une des variables devait être vide
# (les caractères "X" postfixés se neutralisent).
```

opérateurs arithmétiques

+
: plus

-
: moins

*
: multiplication

/
: division

**
: exponentiel

```
# Bash, version 2.02, introduit l'opérateur exponentiel "**".

let "z=5**3"
echo "z = $z"   # z = 125
```

%
: modulo, ou mod (renvoie le *reste* de la division d'un entier)

```
bash$ expr 5 % 3
2
```

5/3 = 1 avec un reste de 2

Cet opérateur trouve son utilité, entre autres choses, dans la génération de nombres compris dans un intervalle donné (voir l'Exemple 9.26, « Générer des nombres aléatoires » et l'Exemple 9.30, « Lancement d'un seul dé avec RANDOM ») et pour le formatage de la sortie d'un programme (voir l'Exemple 26.15, « Application complexe des tableaux *Exploration d'une étrange série mathématique* » et l'Exemple A.6, « *collatz* : Séries de Collatz »). Il peut même être utilisé pour générer des nombres premiers (voir Exemple A.16, « *primes*: Générer des nombres premiers en utilisant l'opérateur modulo »). De manière surprenante, l'opérateur Modulo revient assez fréquemment dans de nombreuses astuces numériques.

Exemple 8.1. Plus grand diviseur commun

```
#!/bin/bash
# gcd.sh: plus grand diviseur commun
#         Utilise l'algorithme d'Euclide

#  Le "plus grand diviseur commun" (pgcd) de deux entiers est l'entier le plus
#+ important qui divisera les deux sans reste.

#  L'algorithme d'Euclide utilise des divisions successives.
#  À chaque passe,
#+ dividende <---  diviseur
#+ diviseur  <---  reste
#+ jusqu'à ce que reste 0.
#+ pgcd = dividende, à la dernière passe.
#
#  Pour une excellente discussion de l'algorithme d'Euclide, voir le site
#+ de Jim Loy, http://www.jimloy.com/number/euclids.htm.


# ---------------------------------------------------
# Vérification des arguments
ARGS=2
E_MAUVAISARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` premier_nombre deuxieme_nombre"
  exit $E_MAUVAISARGS
fi
# ---------------------------------------------------


pgcd ()
{

                                #  Affectation arbitraire.
  dividende=$1                  #  Il importe peu de savoir lequel est le
  diviseur=$2                   #+ plus grand.
                                #  Pourquoi pas ?

  reste=1                       #  Si une variable non initialisée est utilisée
                                #+ dans la boucle,
                                #+ cela finit en un message d'erreur lors de
                                #+ la première passe dans la boucle.

  until [ "$reste" -eq 0 ]
  do
    let "reste = $dividende % $diviseur"
    dividende=$diviseur         #  Maintenant, répétez avec les deux plus
                                #+ petits nombres.
    diviseur=$reste
  done                          # Algorithme d'Euclide

}                               # Le dernier $dividende est le pgcd.


pgcd $1 $2
```

```
echo; echo "PGCD de $1 et $2 = $dividende"; echo


# Exercice :
# --------
#  Vérifier les arguments en ligne de commande pour s'assurer que ce soit des
#+ entiers et quitter le script avec une erreur appropriée dans le cas contraire.

exit 0
```

+=

plus-égal (incrémente une variable par une constante)

let "var += 5" renvoie dans *var* sa propre valeur incrémentée de 5.

-=

moins-égal (décrémente une variable par une constante)

*=

multiplication-égal (multiplie une variable par une constante)

let "var *= 4" renvoie dans *var* sa propre valeur multipliée par 4.

/=

division-égal (divise une variable par une constante)

%=

modulo-égal (*reste* de la division de la variable par une constante)

Les opérateurs arithmétiques sont trouvés souvent dans une expression expr ou let.

Exemple 8.2. Utiliser des opérations arithmétiques

```
#!/bin/bash
# Compter jusqu'à 11 de 10 façons différentes.

n=1; echo -n "$n "

let "n = $n + 1"   # let "n = n + 1"   fonctionne aussi.
echo -n "$n "

: $((n = $n + 1))
#  ":" nécessaire parce que sinon Bash essaie d'interpréter
#+ "$((n = $n + 1))" comme une commande.
echo -n "$n "

(( n = n + 1 ))
#  Une alternative plus simple par rapport à la méthode ci-dessus.
#  Merci, David Lombard, pour nous l'avoir indiqué.
echo -n "$n "

n=$(($n + 1))
echo -n "$n "

: $[ n = $n + 1 ]
#  ":" nécessaire parce que sinon Bash essaie d'interpréter
#+ "$[ n = $n + 1 ]" comme une commande.
# Fonctionne même si "n" a été initialisé comme une chaîne de caractères.
echo -n "$n "

n=$[ $n + 1 ]
#  Fonctionne même si "n" a été initialisé comme une chaîne de caractères.
#* Eviter ce type de construction, car elle est obsolète et non portable.
# Merci, Stephane Chazelas.
echo -n "$n "
```

```
# Maintenant des opérateurs d'incrément style C.
# Merci de l'indication, Frank Wang.

let "n++"          # let "++n"  fonctionne aussi.
echo -n "$n "

(( n++ ))          # (( ++n )  fonctionne aussi.
echo -n "$n "

: $(( n++ ))       # : $(( ++n )) fonctionne aussi.
echo -n "$n "

: $[ n++ ]         # : $[ ++n ]] fonctionne aussi.
echo -n "$n "

echo

exit 0
```

Note

Les variables de type entier dans Bash sont réellement de type entier *long* signé (32-bit), dans la plage -2147483648 à 2147483647. Une opération qui prend une variable en dehors de ces limites donnera un résultat erroné.

```
a=2147483646
echo "a = $a"      # a = 2147483646
let "a+=1"         # Incrémente "a".
echo "a = $a"      # a = 2147483647
let "a+=1"         # Incrémente encore "a", en dehors de la limite.
echo "a = $a"      # a = -2147483648
                   #      ERREUR (hors limites)
```

À partir de la version 2.05b, Bash dispose des entiers à 64 bits.

Attention

Bash ne comprend pas l'arithmétique à virgule flottante. Il traite les nombres contenant un point décimal comme des chaînes de caractères.

```
a=1.5

let "b = $a + 1.3"  # Erreur.
# t2.sh: let: b = 1.5 + 1.3: erreur de syntaxe dans l'expression
#                            (error token is ".5 + 1.3")

echo "b = $b"       # b=1
```

Utiliser bc dans des scripts qui ont besoin de calculs à virgule flottante ou de fonctions de la bibliothèque math.

opérateurs de bits. Les opérateurs de bits font rarement une apparition dans les scripts shell. Leur utilisation principale semble être la manipulation et le test de valeurs lues à partir de ports ou de sockets399. Le « renversement de bit » est plus intéressant pour les langages compilés, comme le C et le C++, qui fournissent un accès direct au matériel.

opérateurs binaires

<<

décalage gauche d'un bit (revient à multiplier par 2 pour chaque décalage)

<<=

décalage gauche-égal

`let "var <<= 2"` renvoie dans *`var`* sa propre valeur décalée à gauche de 2 bits (donc multipliée par 4)

>>

décalage droit d'un bit (revient à diviser par 2 pour chaque position du décalage)

>>=

décalage droit-égal (inverse de >>=)

&

ET binaire

&=

ET binaire-égal

|

OU binaire

|=

OU binaire-égal

~

NON binaire

^

XOR binaire

^=

XOR binaire-égal

opérateurs (booléens) logiques

!

NOT

```
if [ ! -f $NOMFICHIER ]
then
  ...
```

&&

ET

```
if [ $condition1 ] && [ $condition2 ]
# Identique à :  if [ $condition1 -a $condition2 ]
# Renvoie vrai si condition1 et condition2 sont vraies...

if [[ $condition1 && $condition2 ]]    # Fonctionne aussi.
# Notez que l'opérateur && n'est pas autorisé dans une construction [ ... ].
```

Note

Suivant le contexte, && peut aussi être utilisé dans une liste ET372 pour concaténer des commandes.

||

OU

```
if [ $condition1 ] || [ $condition2 ]
# Identique à:  if [ $condition1 -o $condition2 ]
# Renvoie vrai si condition1 ou condition2 est vraie...

if [[ $condition1 || $condition2 ]]    # Fonctionne aussi.
```

```
# Notez que l'opérateur || n'est pas autorisé dans des constructions [ ... ].
```

Note

Bash teste l'état de sortie 44 de chaque instruction liée avec un opérateur logique.

Exemple 8.3. Tests de conditions composées en utilisant && et ||

```
#!/bin/bash

a=24
b=47

if [ "$a" -eq 24 ] && [ "$b" -eq 47 ]
then
  echo "Le test #1 a réussi."
else
  echo "Le test #1 a échoué."
fi

# ERREUR:  if [ "$a" -eq 24 && "$b" -eq 47 ]
#          essaie d'exécuter ' [ "$a" -eq 24 '
#          et échoue à trouver le ']' correspondant.
#
# Note :   if [[ $a -eq 24 && $b -eq 24 ]]   fonctionne
#    Le test if avec double crochets est plus flexible que la version avec
#    simple crochet.
#    (Le "&&" a une signification différente en ligne 17 qu'en ligne 6).
#    Merci, Stephane Chazelas.


if [ "$a" -eq 98 ] || [ "$b" -eq 47 ]
then
  echo "Le test #2 a réussi."
else
  echo "Le test #2 a échoué."
fi


#  Les options -a et -o apportent une alternative au test de la condition composée.
#  Merci à Patrick Callahan pour avoir remarqué ceci.


if [ "$a" -eq 24 -a "$b" -eq 47 ]
then
  echo "Le test #3 a réussi."
else
  echo "Le test #3 a échoué."
fi


if [ "$a" -eq 98 -o "$b" -eq 47 ]
then
  echo "Le test #4 a réussi."
else
  echo "Le test #4 a échoué."
fi


a=rhino
b=crocodile
if [ "$a" = rhino ] && [ "$b" = crocodile ]
then
  echo "Le test #5 a réussi."
else
  echo "Le test #5 a échoué."
```

```
fi

exit 0
```

Les opérateurs && et || trouvent aussi leur utilité dans un contexte arithmétique.

```
bash$ echo $(( 1 && 2 )) $((3 && 0)) $((4 || 0)) $((0 || 0))
1 0 1 0
```

opérateurs divers

,

Opérateur virgule

L'**opérateur virgule** chaîne ensemble deux ou plusieurs opérations arithmétiques. Toutes les opérations sont évaluées (avec des possibles *effets indésirables*) [1]), mais seule la dernière opération est renvoyée.

```
let "t1 = ((5 + 3, 7 - 1, 15 - 4))"
echo "t1 = $t1"               # t1 = 11

let "t2 = ((a = 9, 15 / 3))"  # Initialise "a" et calcule "t2".
echo "t2 = $t2    a = $a"     # t2 = 5    a = 9
```

L'opérateur virgule trouve son utilité principalement dans les boucles for. Voir l'Exemple 10.12, « Une boucle *for* à la C ».

8.2. Constantes numériques

Un script shell interprète un nombre comme décimal (base 10), sauf si ce nombre a un certain préfixe ou notation. Un nombre précédé par un *0* est *octal* (base 8). Un nombre précédé par *0x* est *hexadécimal* (base 16). Un nombre comprenant un *#* est évalué comme *BASE#NOMBRE* (avec les restrictions d'intervalle et de codification).

Exemple 8.4. Représentation des constantes numériques

```
#!/bin/bash
# numbers.sh: Représentation des nombres en différentes bases.

# Décimal: par défaut
let "dec = 32"
echo "nombre décimal = $dec"             # 32
# Rien qui ne sort de l'ordinaire ici.


# Octal: nombres précédés par '0' (zero)
let "oct = 032"
echo "nombre octal = $oct"               # 26
# Exprime le résultat en décimal.
# ------- -- -------- -- -------

# Hexadecimal: nombres précédés par '0x' ou '0X'
let "hex = 0x32"
echo "nombre hexadécimal = $hex"         # 50

echo $((0x9abc))                         # 39612
#     ^^      ^^   expansion/évaluation arithmétique entre parenthèses
# Exprime le résultat en décimal.
```

[1] Les *effets indésirables* ont, bien sûr, des conséquences inattendues et indésirables.

```
# Autres bases: BASE#NOMBRE
# BASE entre 2 et 64.
# NUMBER doit utiliser les symboles compris dans l'intervalle BASE, voir ci-dessous.

let "bin = 2#111100111001101"
echo "nombre binaire = $bin"              # 31181

let "b32 = 32#77"
echo "nombre en base-32 = $b32"             # 231

let "b64 = 64#@_"
echo "nombre en base-64 = $b64"             # 4031
# Cette notation fonctionne seulement pour un intervalle limité (2 - 64) des caractères
ASCII
# 10 chiffres + 26 caractères minuscules + 26 caractères majuscules + @ + _


echo

echo $((36#zz)) $((2#10101010)) $((16#AF16)) $((53#1aA))
                                          # 1295 170 44822 3375


#  Note importante:
#  ----------------
#  Utiliser un chiffre en dehors de l'échelle de la notation spécifiée
#+ donne un message d'erreur.

let "bad_oct = 081"
# Message d'erreur (partiel) en sortie:
# bad_oct = 081 : valeur trop élevée pour la base (l'erreur est "081")
#             Les nombres octal utilisent seulement des chiffres dans l'intervalle 0 -
7.

exit 0   # Merci, Rich Bartell et Stephane Chazelas, pour cette clarification.
```

Partie Part 3. Après l'approche basique

Chapitre 9. Les variables revisitées

Utilisées proprement, les variables peuvent ajouter puissance et flexibilité à vos scripts. Ceci nécessite l'apprentissage de leurs subtilités et de leurs nuances.

9.1. Variables internes

Variables intégrées159
Variables affectant le comportement des scripts bash.

$BASH

Le chemin vers le binaire *Bash*.

```
bash$ echo $BASH
/bin/bash
```

$BASH_ENV

Une variable d'environnement pointant vers un script Bash de démarrage lu lorsqu'un script est invoqué.

$BASH_SUBSHELL

une variable indiquant le niveau du sous-shell342. C'est un nouvel ajout de Bash, version 3469.

Voir l'Exemple 20.1, « Étendue des variables dans un sous-shell » pour son utilisation.

$BASH_VERSINFO[n]

Un tableau375 à six éléments contenant des informations sur la version installée de Bash. Ceci est similaire à $BASH_VERSION, ci-dessous, mais en un peu plus détaillé.

```
# Infos sur la version de Bash :

for n in 0 1 2 3 4 5
do
  echo "BASH_VERSINFO[$n] = ${BASH_VERSINFO[$n]}"
done

# BASH_VERSINFO[0] = 3                      # No majeur de version.
# BASH_VERSINFO[1] = 00                     # No mineur de version.
# BASH_VERSINFO[2] = 14                     # Niveau de correctifs.
# BASH_VERSINFO[3] = 1                      # Version construite.
# BASH_VERSINFO[4] = release                # État de la version.
# BASH_VERSINFO[5] = i386-redhat-linux-gnu  # Architecture.
                                            # (identique à $MACHTYPE).
```

$BASH_VERSION

La version de Bash installée sur le système.

```
bash$ echo $BASH_VERSION
3.00.14(1)-release
```

```
tcsh% echo $BASH_VERSION
BASH_VERSION: Undefined variable.
```

Vérifier $BASH_VERSION est une bonne méthode pour déterminer le shell qui est en cours d'exécution. $SHELL ne donne pas nécessairement la bonne réponse.

$DIRSTACK

La valeur du dessus de la pile de répertoires (affectée par pushd169 et popd169)

Cette variable intégrée correspond à la commande dirs. Néanmoins, **dirs** affiche le contenu entier de la pile de répertoires.

`$EDITOR`

L'éditeur invoqué par défaut par un script, habituellement **vi** ou **emacs**.

`$EUID`

Numéro d'identifiant « effectif » de l'utilisateur.

Numéro d'identification, quelle que soit l'identité que l'utilisateur actuel assume, peut-être suite à un su.

Attention

`$EUID` n'est pas nécessairement le même que $UID.

`$FUNCNAME`

Nom de la fonction en cours.

```
xyz23 ()
{
  echo "$FUNCNAME en cours d'exécution."  # xyz23 en cours d'exécution.
}

xyz23

echo "FUNCNAME = $FUNCNAME"        # FUNCNAME =
                                   # vide en dehors d'une fonction
```

`$GLOBIGNORE`

Une liste de modèles de noms de fichiers à exclure de la correspondance lors d'un remplacement.

`$GROUPS`

Groupes auxquels appartient l'utilisateur.

C'est une liste (de type tableau) des numéros d'identifiant de groupes pour l'utilisateur actuel, identique à celle enregistrée dans `/etc/passwd` et `/etc/group`.

```
root# echo $GROUPS
0


root# echo ${GROUPS[1]}
1


root# echo ${GROUPS[5]}
6
```

`$HOME`

Répertoire personnel de l'utilisateur, habituellement `/home/utilisateur` (voir l'Exemple 9.16, « Utiliser la substitution et les messages d'erreur »)

`$HOSTNAME`

La commande hostname définit le nom de l'hôte au démarrage en utilisant un script de démarrage. Néanmoins, la fonction `gethostname()` initialise la variable interne Bash `$HOSTNAME`. Voir aussi l'Exemple 9.16, « Utiliser la substitution et les messages d'erreur ».

`$HOSTTYPE`

Type de l'hôte.

Comme $MACHTYPE, identifie le matériel du système.

```
bash$ echo $HOSTTYPE
i686
```

`$IFS`

Séparateur interne du champ de saisie.

Cette variable détermine la façon dont Bash reconnaît les champs ou les limites de mots lorsqu'il interprète des chaînes de caractères.

La valeur par défaut est un espace blanc (espace, tabulation et retour chariot) mais peut être changé, par exemple, pour analyser un fichier de données séparées par des virgules. Notez que $* utilise le premier caractère contenu dans `$IFS`. Voir l'Exemple 5.1, « Afficher des variables bizarres ».

```
bash$ echo "$IFS"


(Avec la valeur par défaut de $IFS, une ligne blanche apparaît.)


bash$ echo "$IFS" | cat -vte
 ^I$
 $
(Affiche un espace blanc -- un espace, ^I [tabulation horizontale],
  ou un retour chariot -- et affiche  un dollar ($) en fin de ligne.)


bash$ bash -c 'set w x y z; IFS=":-;"; echo "$*"'
w:x:y:z
(Lit les commandes à partir de la chaîne et affecte tout
argument suivant les paramètres de position)
```

Attention

`$IFS` ne gère pas les espaces blancs de la même façon que les autres caractères.

Exemple 9.1. $IFS et espaces blancs

```
#!/bin/bash
# $IFS traite les espaces blancs différemment des autres caractères.

affiche_un_argument_par_ligne()
{
  for arg
  do echo "[$arg]"
  done
}

echo; echo "IFS=\" \""
echo "-------"

IFS=" "
var=" a  b c   "
affiche_un_argument_par_ligne $var  # affiche_un_argument_par_ligne `echo
" a  b c   "`
#
# [a]
# [b]
# [c]


echo; echo "IFS=:"
echo "-----"

IFS=:
```

```
var=":a::b:c:::"        # Identique à ci-dessus, mais substitue ":" à " ".
affiche_un_argument_par_ligne $var
#
# []
# [a]
# []
# [b]
# [c]
# []
# []
# []

# La même chose arrive avec le séparateur de champs "FS" dans awk.

# Merci, Stephane Chazelas.

echo

exit 0
```

(Merci beaucoup, Stéphane Chazelas, pour cette clarification et ces exemples.)

Voir aussi l'Exemple 15.41, « Analyser le domaine d'un courrier indésirable » , Exemple 10.7, « Un remplaçant de *grep* pour les fichiers binaires » et Exemple 18.14, « Analyser une boîte mail » pour des exemples instructifs sur l'utilisation de `$IFS`.

`$IGNOREEOF`

Ignore EOF : nombre de fins de fichier (control-D) que le shell va ignorer avant de déconnecter.

`$LC_COLLATE`

Souvent intégré dans les fichiers `.bashrc` ou `/etc/profile`, cette variable contrôle l'ordre d'examen dans l'expansion des noms de fichiers et les correspondances de modèles. Si elle est mal gérée, `LC_COLLATE` peut apporter des résultats inattendus dans le remplacement de noms de fichiers.

Note

À partir de la version 2.05 de Bash, le remplacement de noms de fichiers ne tient plus compte des lettres en minuscules et en majuscules dans une suite de caractères entre crochets. Par exemple, **ls [A-M]*** correspondrait à la fois à `Fichier1.txt` et à `fichier1.txt`. Pour annuler le comportement personnalisé de la correspondance par crochets, initialisez `LC_COLLATE` à `C` par un **`export LC_COLLATE=C`** dans `/etc/profile` et/ou `~/.bashrc`.

`$LC_CTYPE`

Cette variable interne contrôle l'interprétation des caractères pour le remplacement et la correspondance de modèles.

`$LINENO`

Cette variable correspond au numéro de ligne du script shell dans lequel cette variable apparaît. Elle n'a une signification que dans le script où elle apparait et est surtout utilisée dans les phases de débogage.

```
# *** DEBUT BLOC DEBUG ***
dernier_argument_command=$_  # Le sauver.

echo "À la ligne numéro $LINENO, la variable \"v1\" = $v1"
echo "Dernier argument de la ligne exécutée = $dernier_argument_command"
# *** FIN BLOC DEBUG ***
```

`$MACHTYPE`

Type de machine.

Identifie le matériel du système.

```
bash$ echo $MACHTYPE
i686
```

`$OLDPWD`

Ancien répertoire courant (« OLD-print-working-directory », ancien répertoire où vous étiez).

`$OSTYPE`

Type de système d'exploitation.

```
bash$ echo $OSTYPE
linux
```

`$PATH`

Chemin vers les binaires, habituellement `/usr/bin/`, `/usr/X11R6/bin/`, `/usr/local/bin`, etc.

Lorsqu'une commande est donnée, le shell recherche automatiquement l'exécutable dans les répertoires listés dans le *chemin*. Le chemin est stocké dans la variable d'environnement, `$PATH`, une liste des répertoires, séparés par le symbole ":". Normalement, le système enregistre la définition de `$PATH` dans `/etc/profile` et/ou `~/.bashrc` (voir l'Annexe G, Fichiers importants).

```
bash$ echo $PATH
/bin:/usr/bin:/usr/local/bin:/usr/X11R6/bin:/sbin:/usr/sbin
```

`PATH=${PATH}:/opt/bin` ajoute le répertoire `/opt/bin` au chemin actuel. Dans un script, il peut être avantageux d'ajouter temporairement un répertoire au chemin de cette façon. Lorsque le script se termine, le `$PATH` original est restauré (un processus fils, tel qu'un script, ne peut pas changer l'environnement du processus père, le shell).

Note

Le « répertoire » courant, `./`, est habituellement omis de `$PATH` pour des raisons de sécurité.

`$PIPESTATUS`

Variable de type tableau375 contenant les codes de sortie de la dernière *commande* exécutée via un tube. De façon étonnante, ceci ne donne pas obligatoirement le même résultat que le code de sortie44 de la dernière commande exécutée.

```
bash$ echo $PIPESTATUS
0

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $PIPESTATUS
141

bash$ ls -al | bogus_command
bash: bogus_command: command not found
bash$ echo $?
127
```

Les membres du tableau `$PIPESTATUS` contiennent le code de sortie de chaque commande respective exécutée via un tube. `$PIPESTATUS[0]` contient le code de sortie de la première commande du tube, `$PIPESTATUS[1]` le code de sortie de la deuxième commande et ainsi de suite.

Attention

La variable `$PIPESTATUS` peut contenir une valeur 0 erronée dans un shell de connexion (dans les versions précédant la 3.0 de Bash).

```
tcsh% bash

bash$ who | grep nobody | sort
bash$ echo ${PIPESTATUS[*]}
0
```

Les lignes ci-dessus contenues dans un script produiraient le résultat attendu, `0 1 0`.

Merci, Wayne Pollock pour avoir partagé ceci en apportant l'exemple ci-dessus.

Note

La variable `$PIPESTATUS` donne des résultats inattendus dans certains contextes.

```
bash$ echo $BASH_VERSION
3.00.14(1)-release

bash$ $ ls | commande_boguee | wc
bash: commande_boguee: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
141 127 0
```

Chet Ramey attribue l'affichage ci-dessus au comportement de ls. Si *ls* écrit dans un *tube* dont la sortie n'est pas lue, alors SIGPIPE le tue et son code de sortie44 est 141. Sinon, son code de sortie est 0, comme attendu. C'est certainement le cas pour tr.

Note

`$PIPESTATUS` est une variable « volatile ». Elle doit être immédiatement capturée après le tube, c'est-à-dire avant que d'autres commandes n'interviennent.

```
bash$ $ ls | commande_boguee | wc
bash: commande_boguee: command not found
 0       0       0

bash$ echo ${PIPESTATUS[@]}
0 127 0

bash$ echo ${PIPESTATUS[@]}
0
```

Note

L'option pipefail470 pourrait être utile dans les cas où `$PIPESTATUS` ne donne pas l'information désirée.

`$PPID`

Le `$PPID` d'un processus est l'identifiant du processus (`PID`) père. [1]

Comparez ceci avec la commande pidof.

`$PROMPT_COMMAND`

Une variable contenant une commande à exécuter juste avant l'affichage de l'invite principale, `$PS1`.

`$PS1`

Ceci est l'invite principale, vue sur la ligne de commande.

`$PS2`

La deuxième invite, vue lorsqu'une saisie supplémentaire est attendue. Elle s'affiche comme « > ».

[1] Le PID du script en cours est $$, bien sûr.

`$PS3`

La troisième invite, affichée lors d'une boucle select (voir l'Exemple 10.29, « Créer des menus en utilisant *select* »)

`$PS4`

La quatrième invite, affichée au début de chaque ligne d'affichage lorsqu'un script a été appelé avec l'option419 -x. Elle affiche un « + ».

`$PWD`

Répertoire courant (répertoire où vous êtes actuellement)

Ceci est analogue à la commande intégrée pwd.

```
#!/bin/bash

E_MAUVAIS_REPERTOIRE=73

clear # Efface l'écran.

RepertoireCible=/home/bozo/projects/GreatAmericanNovel

cd $RepertoireCible
echo "Suppression des anciens fichiers de $RepertoireCible."

if [ "$PWD" != "$RepertoireCible" ]
then    # Empêche la suppression d'un mauvais répertoire par accident.
  echo "Mauvais répertoire!"
  echo "Dans $PWD, plutôt que $RepertoireCible!"
  echo "Je quitte!"
  exit $E_MAUVAIS_REPERTOIRE
fi

rm -rf *
rm .[A-Za-z0-9]*    # Supprime les fichiers commençant par un point.
# rm -f .[^.]* ..?*   pour supprimer les fichiers commençant par plusieurs points.
# (shopt -s dotglob; rm -f *)   fonctionnera aussi.
# Merci, S.C., pour nous l'avoir indiqué.

# Les noms de fichier peuvent contenir tous les caractères de 0 à 255,
# à l'exception de "/".
# La suppression des fichiers commençant par des caractères bizarres est laissé
# en exercice.

# Autres opérations ici, si nécessaire.

echo
echo "Fait."
echo "Anciens fichiers supprimés de $RepertoireCible."
echo


exit 0
```

`$REPLY`

La variable par défaut lorsqu'aucune n'est adjointe au read. Aussi applicable au menu select, mais renvoie seulement le numéro de l'élément de la variable choisie et non pas la valeur de la variable elle-même.

```
#!/bin/bash
# reply.sh

# REPLY est la variable par défaut d'une commande 'read'

echo
echo -n "Quel est votre légume favori? "
read

echo "Votre légume favori est $REPLY."
```

```
#  REPLY contient la valeur du dernier "read" si et seulement si aucune variable
#+ n'est spécifiée.

echo
echo -n "Quel est votre fruit favori? "
read fruit
echo "Votre fruit favori est $fruit."
echo "mais..."
echo "La valeur de \$REPLY est toujours $REPLY."
#  $REPLY est toujours initialisé à sa précédente valeur car la variable $fruit
#+ a absorbé la nouvelle valeur obtenue par "read".

echo

exit 0
```

`$SECONDS`

Le nombre de secondes pris par l'exécution du script.

```
#!/bin/bash

LIMITE_TEMPS=10
INTERVALLE=1

echo
echo "Appuyez sur Control-C pour sortir avant $LIMITE_TEMPS secondes."
echo

while [ "$SECONDS" -le "$LIMITE_TEMPS" ]
do
  if [ "$SECONDS" -eq 1 ]
  then
    unites=seconde
  else
    unites=secondes
  fi

  echo "Ce script tourne depuis $SECONDS $unites."
  #  Sur une machine lente, le script peut laisser échapper quelquefois
  #+ un élément du comptage dans la boucle while.
  sleep $INTERVALLE
done

echo -e "\a"  # Beep!

exit 0
```

`$SHELLOPTS`

La liste des options419 activées du shell, une variable en lecture seule.

```
bash$ echo $SHELLOPTS
braceexpand:hashall:histexpand:monitor:history:interactive-comments:emacs
```

`$SHLVL`

Niveau du shell, à quel point Bash est imbriqué. [2] Si, à la ligne de commande, $SHLVL vaut 1, alors dans un script, il sera incrémenté et prendra la valeur 2.

Note

Cette variable n'est *pas* affectée par les sous-shells. Utilisez $BASH_SUBSHELL quand vous avez besoin d'une indication d'une imbrication de sous-shell.

[2] Un peu analogue à la récursion365, dans ce contexte,*il*h*brication* réfère à un modèle embarqué à l'intérieur d'un modèle plus large. Une des définitions de *nest*, d'après l'édition 1913 du *dictionnaire Webster*, illustre très bien ceci : « *une collection de boîtes, cases ou d'objets de ce type, d'une taille graduée, les unes dans les autres.* »

`$TMOUT`

Si la variable d'environnement *`$TMOUT`* est initialisée à une valeur différente de zéro appelée `time`, alors l'invite shell dépassera son délai au bout de `time` secondes. Ceci causera une déconnexion.

À partir de la version 2.05b de Bash, il est possible d'utiliser *`$TMOUT`* dans un script avec un read.

```
#  Fonctionne avec des scripts pour Bash, versions
#+ 2.05b et ultérieures.

TMOUT=3    # L'invite s'arrête dans trois secondes.

echo "Quelle est votre chanson favorite?"
echo "Faites vite car vous n'avez que $TMOUT secondes pour répondre !"
read chanson

if [ -z "$chanson" ]
then
  chanson="(sans réponse)"
  # Réponse par défaut.
fi

echo "Votre chanson favorite est $chanson."
```

Il existe d'autres façons, certaines plus complexes, pour implémenter une entrée avec temporisation. Une alternative consiste à configurer une boucle rythmée pour signaler au script la fin de l'attente. Ceci requiert aussi une routine de gestion du signal pour récupérer (voir l'Exemple 29.5, « Récupérer la sortie ») l'interruption créée par la boucle.

Exemple 9.2. Saisie avec délai

```
#!/bin/bash
# timed-input.sh

# TMOUT=3            Fonctionne aussi, depuis les dernières versions de Bash.

INTERRUPTION_TIMER=14
LIMITETEMPS=3  # Trois secondes dans cette instance
               # Peut être configuré avec une valeur différente.

AfficheReponse()
{
  if [ "$reponse" = TIMEOUT ]
  then
    echo $reponse
  else       # ne pas mixer les deux interfaces.
    echo "Votre légume favori est le $reponse"
    kill $!  #  Kill n'est plus nécessaire pour la fonction TimerOn lancée en
             #+ tâche de fond.
             # $! est le PID du dernier job lancé en tâche de fond.
  fi

}


TimerOn()
{
  sleep $LIMITETEMPS && kill -s 14 $$ &
  # Attend trois secondes, puis envoie sigalarm au script.
}


VecteurInt14()
{
  reponse="TIMEOUT"
  AfficheReponse
  exit $INTERRUPTION_TIMER
}
```

```
trap VecteurInt14 $INTERRUPTION_TIMER   #  Interruption de temps (14)
                                        #+ détournée pour notre but.

echo "Quel est votre légume favori?"
TimerOn
read reponse
AfficheReponse


#  C'est une implémentation détournée de l'entrée de temps.
#  Néanmoins l'option "-t" de "read" simplifie cette tâche.
#  Voir le script "t-out.sh", ci-dessous.

#  Si vous avez besoin de quelque chose de réellement élégant...
#+ pensez à écrire l'application en C ou C++,
#+ en utilisant les fonctions de la bibliothèque appropriée, telles que
#+ 'alarm' et 'setitimer'.
#  Néanmoins, pourquoi ne pas chronométrer un script complet,
#+ à la place d'une simple saisie d'un utilisateur ?

exit 0
```

Une autre méthode est d'utiliser stty.

Exemple 9.3. Encore une fois, saisie avec délai

```
#!/bin/bash
# timeout.sh

#  Écrit par Stephane Chazelas,
#+ et modifié par l'auteur de ce document.

INTERVALLE=5                # délai

lecture_delai() {
  delai=$1
  nomvariable=$2
  ancienne_configuration_tty=`stty -g`
  stty -icanon min 0 time ${delai}0
  eval read $nomvariable      # ou simplement    read $nomvariable
  stty "$ancienne_configuration_tty"
  # Voir la page man de "stty".
}

echo; echo -n "Quel est votre nom ? Vite !"
lecture_delai $INTERVALLE votre_nom

#  Ceci pourrait ne pas fonctionner sur tous les types de terminaux.
#+ Le temps imparti dépend du terminal (il est souvent de 25,5 secondes).

echo

if [ ! -z "$votre_nom" ]  #  Si le nom est entré avant que le temps ne se soit
                          #+ écoulé...
then
  echo "Votre nom est $votre_nom."
else
  echo "Temps écoulé."
fi

echo

# Le comportement de ce script diffère un peu de "timed-input.sh".
# À chaque appui sur une touche, le compteur est réinitialisé.

exit 0
```

Peut-être que la méthode la plus simple est d'utiliser l'option `-t` de read.

Exemple 9.4. *read* avec délai

```
#!/bin/bash
# t-out.sh
# Inspiré d'une suggestion de "syngin seven" (merci).

LIMITETEMPS=4        # Quatre secondes

read -t $LIMITETEMPS variable <&1
#                             ^^^
#  Dans ce cas, "<&1" est nécessaire pour Bash 1.x et 2.x,
#  mais inutile pour Bash 3.x.

echo

if [ -z "$variable" ]  # Est nul ?
then
  echo "Temps écoulé, la variable n'est toujours pas initialisée."
else
  echo "variable = $variable"
fi

exit 0
```

`$UID`

Numéro de l'identifiant utilisateur.

Numéro d'identification de l'utilisateur actuel, comme enregistré dans `/etc/passwd`.

C'est l'identifiant réel de l'utilisateur actuel, même s'il a temporairement endossé une autre identité avec su. `$UID` est une variable en lecture seule, non sujette au changement à partir de la ligne de commande ou à l'intérieur d'un script, et est la contrepartie de l'intégré id.

Exemple 9.5. Suis-je root ?

```
#!/bin/bash
# am-i-root.sh:   Suis-je root ou non ?

ROOT_UID=0   # Root a l'identifiant $UID 0.

if [ "$UID" -eq "$ROOT_UID" ]  # Le vrai "root" peut-il se lever, s'il-vous-plaît ?
then
  echo "Vous êtes root."
else
  echo "Vous êtes simplement un utilisateur ordinaire (mais maman vous aime tout
autant)."
fi

exit 0


# ============================================================================ #
# Le code ci-dessous ne s'exécutera pas, parce que le script s'est déjà arrêté.

# Une autre méthode d'arriver à la même fin :

NOM_UTILISATEURROOT=root

nomutilisateur=`id -nu`              # Ou...   nomutilisateur=`whoami`
if [ "$nomutilisateur" = "$NOM_UTILISATEURROOT" ]
then
  echo "Vous êtes root."
else
  echo "Vous êtes juste une personne ordinaire."
```

```
fi
```

Voir aussi l'Exemple 2.3, « *cleanup* : Une version améliorée et généralisée des scripts précédents ».

Note

Les variables `$ENV`, `$LOGNAME`, `$MAIL`, `$TERM`, `$USER` et `$USERNAME` ne sont *pas* des variables intégrées[159] à Bash. Néanmoins, elles sont souvent initialisées comme variables d'environnement dans un des fichiers de démarrage de Bash. `$SHELL`, le nom du shell de connexion de l'utilisateur, peut être configuré à partir de `/etc/passwd` ou dans un script d'« initialisation », et ce n'est pas une variable intégrée à Bash.

```
tcsh% echo $LOGNAME
bozo
tcsh% echo $SHELL
/bin/tcsh
tcsh% echo $TERM
rxvt

bash$ echo $LOGNAME
bozo
bash$ echo $SHELL
/bin/tcsh
bash$ echo $TERM
rxvt
```

Paramètres de position

`$0`, `$1`, `$2`, etc.

Paramètres de positions, passés à partir de la ligne de commande à un script, passés à une fonction, ou initialisés (set) à une variable (voir l'Exemple 4.5, « Paramètres positionnels » et l'Exemple 14.16, « Utiliser *set* avec les paramètres de position »)

`$#`

Nombre d'arguments sur la ligne de commande [3] ou de paramètres de position (voir l'Exemple 33.2, « Un *script d'appel* légèrement plus complexe »)

`$*`

Tous les paramètres de position, vus comme un seul mot.

Note

"`$*`" doit être entre guillemets.

`$@`

Identique à $*, mais chaque paramètre est une chaîne entre guillemets, c'est-à-dire que les paramètres sont passés de manière intacte, sans interprétation ou expansion. Ceci signifie, entre autres choses, que chaque paramètre dans la liste d'arguments est vu comme un mot séparé.

Note

Bien sûr, "$@" doit être entre guillemets.

Exemple 9.6. *arglist* : Affichage des arguments avec $* et $@

```
#!/bin/bash
```

[3] Les mots « argument » et « paramètre » sont souvent utilisés sans distinction. Dans le contexte de ce document, ils ont exactement la même signification, celle d'*une variable passée à un script ou à une fonction*.

```
# arglist.sh
# Appelez ce script avec plusieurs arguments, tels que "un deux trois".

E_MAUVAISARGS=65

if [ ! -n "$1" ]
then
  echo "Usage: `basename $0` argument1 argument2 etc."
  exit $E_MAUVAISARGS
fi

echo

index=1    # Initialise le compteur.

echo "Liste des arguments avec \"\$*\" :"
for arg in "$*"  # Ne fonctionne pas correctement si "$*" n'est pas entre
guillemets.
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $* voit tous les arguments comme un mot entier.
echo "Liste entière des arguments vue comme un seul mot."

echo

index=1    # Ré-initialisation du compteur.
           # Qu'arrive-t'il si vous oubliez de le faire ?

echo "Liste des arguments avec \"\$@\" :"
for arg in "$@"
do
  echo "Arg #$index = $arg"
  let "index+=1"
done             # $@ voit les arguments comme des mots séparés.
echo "Liste des arguments vue comme des mots séparés."

echo

index=1    # Ré-initialisation du compteur.

echo "Liste des arguments avec \$* (sans guillemets) :"
for arg in $*
do
  echo "Argument #$index = $arg"
  let "index+=1"
done             # $* sans guillemets voit les arguments comme des mots séparés.
echo "Liste des arguments vue comme des mots séparés."


exit 0
```

Suite à un **shift**, $@ contient le reste des paramètres de la ligne de commande, sans le `$1` précédent qui a été perdu.

```
#!/bin/bash
# Appelé avec ./script 1 2 3 4 5

echo "$@"    # 1 2 3 4 5
shift
echo "$@"    # 2 3 4 5
shift
echo "$@"    # 3 4 5

# Chaque "shift" perd le paramètre $1.
# "$@" contient alors le reste des paramètres.
```

Le paramètre spécial $@ trouve son utilité comme outil pour filtrer l'entrée des scripts shell. La construction **cat "$@"** accepte l'entrée dans un script soit à partir de `stdin`, soit à partir de fichiers donnés en paramètre du script. Voir

Exemple 15.24, « *rot13* : rot13, cryptage ultra-faible. » et l'Exemple 15.25, « Générer des énigmes « Crypto-Citations » ».

Attention

Les paramètres $* et $@ affichent quelque fois un comportement incohérent et bizarre, suivant la configuration de $IFS.

Exemple 9.7. Comportement de $* et $@ incohérent

```
#!/bin/bash

#  Comportement non prévisible des variables internes Bash "$*" et "$@",
#+ suivant qu'elles soient ou non entre guillemets.
#  Gestion incohérente de la séparation de mots et des retours chariot.


set -- "Premier un" "second" "troisième:un" "" "Cinquième: :un"
# Initialise les arguments du script, $1, $2, etc.

echo

echo 'IFS inchangée, utilisant "$*"'
c=0
for i in "$*"               # entre guillemets
do echo "$((c+=1)): [$i]"   # Cette ligne reste identique à chaque instance.
                            # Arguments de echo.
done
echo ---

echo 'IFS inchangée, utilisant $*'
c=0
for i in $*                 # sans guillemets
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS inchangée, utilisant "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS inchangée, utilisant $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

IFS=:
echo 'IFS=":", utilisant "$*"'
c=0
for i in "$*"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilisant $*'
c=0
for i in $*
do echo "$((c+=1)): [$i]"
done
echo ---

var=$*
echo 'IFS=":", utilisant "$var" (var=$*)'
c=0
```

```
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilisant $var (var=$*)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

var="$*"
echo 'IFS=":", utilisant $var (var="$*")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilisant "$var" (var="$*")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilisant "$@"'
c=0
for i in "$@"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilisant $@'
c=0
for i in $@
do echo "$((c+=1)): [$i]"
done
echo ---

var=$@
echo 'IFS=":", utilisant $var (var=$@)'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilisant "$var" (var=$@)'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

var="$@"
echo 'IFS=":", utilisant "$var" (var="$@")'
c=0
for i in "$var"
do echo "$((c+=1)): [$i]"
done
echo ---

echo 'IFS=":", utilisant $var (var="$@")'
c=0
for i in $var
do echo "$((c+=1)): [$i]"
done
```

```
echo

# Essayez ce script avec ksh ou zsh -y.

exit 0

# Ce script exemple par Stephane Chazelas,
# et légèrement modifié par l'auteur de ce document.
```

Note

Les paramètres **$@** et **$*** diffèrent seulement lorsqu'ils sont entre guillemets.

Exemple 9.8. $* et $@ lorsque $IFS est vide

```
#!/bin/bash

#+ Si $IFS est initialisé mais vide,
#+ alors "$*" et "$@" n'affichent pas les paramètres de position
#+ comme on pourrait s'y attendre.

mecho ()       # Affiche les paramètres de position.
{
echo "$1,$2,$3";
}


IFS=""         # Initialisé, mais vide.
set a b c      # Paramètres de position.

mecho "$*"     # abc,,
mecho $*       # a,b,c

mecho $@       # a,b,c
mecho "$@"     # a,b,c

#  Le comportement de $* et $@ quand $IFS est vide dépend de la version de
#+ Bash ou sh.
#  Personne ne peux donc conseiller d'utiliser cette «fonctionnalité» dans un
#+ script.


# Merci, Stephane Chazelas.

exit 0
```

Autres paramètres spéciaux

`$-`

Les options passées au script (en utilisant set). Voir l'Exemple 14.16, « Utiliser *set* avec les paramètres de position ».

Attention

Ceci était originellement une construction de *ksh* adoptée dans Bash et, malheureusement, elle ne semble pas fonctionner de façon fiable dans les scripts Bash. Une utilité possible pour ceci est d'avoir un script testant lui-même s'il est interactif.

`$!`

Identifiant du processus (PID) du dernier job ayant fonctionné en tâche de fond.

```
TRACE=$0.log

COMMANDE1="sleep 100"

echo "Trace des PID des commandes en tâche de fond pour le script : $0" >> "$TRACE"
# Pour qu'ils soient enregistrés et tués si nécessaire.
echo >> "$TRACE"

# Commandes de trace.

echo -n "PID de \"$COMMANDE1\" :  " >> "$TRACE"
${COMMANDE1} &
echo $! >> "$TRACE"
# PID de "sleep 100" :  1506

# Merci, Jacques Lederer, pour cette suggestion.
```

Utiliser `$!` pour contrôler un job :

```
job_qui_peut_se_bloquer & { sleep ${TIMEOUT}; eval 'kill -9 $!' &> /dev/null; }
# Force la fin d'un programme qui se comporte mal.
# Utile, par exemple, dans les scripts d'initialisation.

# Merci, Sylvain Fourmanoit, pour cette utilisation ingénieuse de la variable "!".
```

Ou autrement :

```
# Exemple de Matthew Sage.
# Utilisé avec sa permission.

DELAI=30   # Délai d'attente en secondes
nombre=0

job_qui_peut_se_bloquer & {
        while ((nombre < DELAI )); do
                eval '[ ! -d "/proc/$!" ] && ((count = TIMEOUT))'
                #  /proc est l'endroit où sont disponibles des informations
                #+ sur les processus en cours d'exécution.
                # "-d" teste si le répertoire existe.
                # Donc, nous attendons que le job en question se manifeste.
                ((nombre++))
                sleep 1
        done
        eval '[ -d "/proc/$!" ] && kill -15 $!'
        # Si le job est en cours d'exécution, tuons-le.
}
```

`$_`

Variable spéciale initialisée avec le dernier argument de la dernière commande exécutée.

Exemple 9.9. Variable tiret bas

```
#!/bin/bash

echo $_              # /bin/bash
                     # Simple appel de /bin/bash pour lancer ce script.

du >/dev/null        # Donc pas de sortie des commandes
echo $_              # du

ls -al >/dev/null    # Donc pas de sortie des commandes
echo $_              # -al  (dernier argument)

:
echo $_              # :
```

$?

Code de sortie44 d'une commande, d'une fonction352 ou du script lui-même (voir l'Exemple 23.7, « Maximum de deux nombres »)

$$

Identifiant du processus du script lui-même. La variable $$ trouve fréquemment son utilité dans les scripts pour construire des noms de fichiers temporaires « uniques » (voir l'Exemple A.13, « *ftpget*: Télécharger des fichiers via ftp », l'Exemple 29.6, « Nettoyage après un **Control-C** », l'Exemple 15.31, « Déballer une archive *rpm* » et l'Exemple 14.27, « Un script qui se tue lui-même »). Ceci est généralement plus simple que d'appeler mktemp.

9.2. Manipuler les chaînes de caractères

Bash supporte un nombre surprenant d'opérations de manipulation de chaînes de caractères. Malheureusement, ces outils manquent d'unité. Certains sont un sous-ensemble de la substitution de paramètre et les autres font partie des fonctionnalités de la commande UNIX expr. Ceci produit une syntaxe de commande non unifiée et des fonctionnalités qui se recoupent, sans parler de la confusion engendrée.

Longueur de chaînes de caractères

${#chaine}

expr length $chaine

C'est l'équivalent de la fonction *strlen()* en *C*.

expr "$chaine" : '.*'

```
chaineZ=abcABC123ABCabc

echo ${#chaineZ}                 # 15
echo `expr length $chaineZ`      # 15
echo `expr "$chaineZ" : '.*'`    # 15
```

Exemple 9.10. Insérer une ligne blanche entre les paragraphes d'un fichier texte

```
#!/bin/bash
# paragraph-space.sh

# Insère une ligne blanche entre les paragraphes d'un fichier texte.
# Usage: $0 <NOMFICHIER

LONGUEUR_MINI=45        # Il peut être nécessaire de changer cette valeur.
#  Suppose que les lignes plus petites que $LONGUEUR_MINI caractères
#+ terminent un paragraphe.

while read ligne  # Pour toutes les lignes du fichier...
do
  echo "$ligne"   # Afficher la ligne.

  longueur=${#ligne}
  if [ "$longueur" -lt "$LONGUEUR_MINI" ]
    then echo    # Ajoute une ligne blanche après chaque petite ligne.
  fi
done

exit 0
```

Longueur de sous-chaînes correspondant à un motif au début d'une chaîne

expr match "$chaine" '$souschaine'

`$souschaine` est une expression rationnelle313.

expr "$chaine" : '$souschaine'

$souschaine est une expression rationnelle.

```
chaineZ=abcABC123ABCabc
#       |------|

echo `expr match "$chaineZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$chaineZ" : 'abc[A-Z]*.2'`       # 8
```

Index

expr index $chaine $souschaine

Position numérique dans $chaine du premier caractère dans $souschaine qui correspond.

```
chaineZ=abcABC123ABCabc
echo `expr index "$chaineZ" C12`             # 6
                                             # C position.

echo `expr index "$chaineZ" 1c`              # 3
# 'c' (à la position #3) correspond avant '1'.
```

Ceci est l'équivalent le plus proche de *strchr()* en **C**.

Extraction d'une sous-chaîne

${chaine:position}

Extrait une sous-chaîne de *$chaine* à partir de la position *$position*.

Si le paramètre `$chaine` est « * » ou « @ », alors cela extrait les paramètres de position, [4] commençant à `$position`.

${chaine:position:longueur}

Extrait *$longueur* caractères d'une sous-chaîne de *$chaine* à la position *$position*.

```
chaineZ=abcABC123ABCabc
#       0123456789.....
#       indexage base 0.

echo ${chaineZ:0}                            # abcABC123ABCabc
echo ${chaineZ:1}                            # bcABC123ABCabc
echo ${chaineZ:7}                            # 23ABCabc

echo ${chaineZ:7:3}                          # 23A
                                             # Trois caractères de la sous-chaîne.



# Est-il possible d'indexer à partir de la fin de la chaîne ?

echo ${chaineZ:-4}                           # abcABC123ABCabc
# Par défaut la chaîne complète, comme dans ${parametre:-default}.
# Néanmoins...

echo ${chaineZ:(-4)}                         # Cabc
echo ${chaineZ: -4}                          # Cabc
# Maintenant, cela fonctionne.
#  Des parenthèses ou des espaces ajoutés permettent un échappement du paramètre
#+ de position.

# Merci, Dan Jacobson, pour cette indication.
```

Les arguments *position* et *longueur* peuvent devenir des « paramètres », c'est-à-dire représentés par une variable, plutôt que

[4] Ceci s'applique soit aux arguments en ligne de commande soit aux paramètres passés à une fonction352.

par une constante numérique.

Exemple 9.11. Générer « aléatoirement » une chaîne de huit caractères

```
#!/bin/bash
# rand-string.sh
# Générer aléatoirement une chaîne de huit caractères.

if [ "-n $1" ]  #  Si présence d'un argument en ligne de commande,
then            #+ alors l'utiliser comme chaîne de départ.
  chaine0="$1"
else            #  Sinon, utiliser le PID du script.
  chaine0="$$"
fi

POS=2  # On commence en position 2.
LONG=8  # Extraction de huit caractères.

chaine1=$( echo "$chaine0" | md5sum | md5sum )
# Double mixage :            ^^^^^^   ^^^^^^

chainealeatoire="${chaine1:$POS:$LONG}"
# Peut se paramétrer       ^^^^ ^^^^^

echo "$chainealeatoire"

exit $?

# bozo$ ./rand-string.sh mon-motdepasse
# 1bdd88c4

#  Non, ceci n'est pas recommandé
#+ comme méthode sûre de génération de mots de passe.
```

Si le paramètre `$chaine` est « * » ou « @ », alors ceci extrait un maximum de `$longueur` du paramètre de position, en commençant à `$position`.

```
echo ${*:2}          # Affiche le deuxième paramètre de position et les suivants.
echo ${@:2}          # Identique à ci-dessus.

echo ${*:2:3}        # Affiche trois paramètres de position, en commençant par le
deuxième.
```

expr substr $chaine $position $longueur

Extrait *`$longueur`* caractères à partir de *`$chaine`* en commençant à *`$position`*.

```
chaineZ=abcABC123ABCabc
#       123456789......
#       indexage base 1.

echo `expr substr $chaineZ 1 2`              # ab
echo `expr substr $chaineZ 4 3`              # ABC
```

expr match "$chaine" '\($souschaine\)'

Extrait *`$souschaine`* à partir du début de *`$chaine`*, et où *`$souschaine`* est une expression rationnelle313.

expr "$chaine" : '\($souschaine\)'

Extrait *`$souschaine`* à partir du début de *`$chaine`*, et où *`$souschaine`* est une expression rationnelle.

```
chaineZ=abcABC123ABCabc
#       =======

echo `expr match "$chaineZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
```

```
echo `expr "$chaineZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
echo `expr "$chaineZ" : '\(.......\)'`                   # abcABC1
# Toutes les formes ci-dessus donnent un résultat identique.
```

expr match "$chaine" '.*\($souschaine\)'

Extrait *$souschaine* à la *fin* de *$chaine*, et où *$souschaine* est une expression rationnelle.

expr "$chaine" : '.*\($souschaine\)'

Extrait *$souschaine* à la *fin* de *$chaine*, et où *$souschaine* est une expression rationnelle.

```
chaineZ=abcABC123ABCabc
#               ======

echo `expr match "$chaineZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    # ABCabc
echo `expr "$chaineZ" : '.*\(......\)'`                       # ABCabc
```

Suppression de sous-chaînes

${chaine#souschaine}

Supprime la correspondance la plus petite de *$souschaine* à partir du *début* de *$chaine*.

${chaine##souschaine}

Supprime la correspondance la plus grande de *$souschaine* à partir du *début* de *$chaine*.

```
chaineZ=abcABC123ABCabc
#       |----|
#       |----------|

echo ${chaineZ#a*C}      # 123ABCabc
# Supprime la plus petite correspondance entre 'a' et 'C'.

echo ${chaineZ##a*C}     # abc
# Supprime la plus grande correspondance entre 'a' et 'C'.
```

${chaine%souschaine}

Supprime la plus petite correspondance de *$souschaine* à partir de la *fin* de *$chaine*.

Par exemple :

```
# Renomme tous les fichiers de $PWD
#+ en remplaçant le suffixe "TXT" par "txt".
# Par exemple, "fichier1.TXT" devient "fichier1.txt" . . .

SUFF=TXT
suff=txt

for i in $(ls *.$SUFF)
do
  mv -f $i ${i%.$SUFF}.$suff
  #  Ne modifie rien *en dehors* de la correspondance la plus courte
  #+ commençant du côté droit de $i . . .
done ### Ceci pourrait être condenser en une ligne si nécessaire.

# Thank you, Rory Winston.
```

${chaine%%souschaine}

Supprime la plus grande correspondance de *$souschaine* à partir de la *fin* de *$chaine*.

```
chaineZ=abcABC123ABCabc
#                  ||
#        |------------|

echo ${chaineZ%b*c}      # abcABC123ABCa
```

```
#  Supprime la plus petite correspondance entre 'b' et 'c', à partir de la fin
#+ de $chaineZ.

echo ${chaineZ%%b*c}     # a
#  Supprime la plus petite correspondance entre 'b' et 'c', à partir de la fin
#+ de $chaineZ.
```

Cet opérateur est utilisé pour générer des noms de fichier.

Exemple 9.12. Convertir des formats de fichiers graphiques avec une modification du nom du fichier

```
#!/bin/bash
#  cvt.sh:
#  Convertit les fichiers image MacPaint contenus dans un répertoire dans le
#+ format "pbm".

#  Utilise le binaire "macptopbm" provenant du paquetage "netpbm",
#+ qui est maintenu par Brian Henderson (bryanh@giraffe-data.com).
#  Netpbm est un standard sur la plupart des distributions Linux.

OPERATION=macptopbm
SUFFIXE=pbm         # Suffixe pour les nouveaux noms de fichiers.

if [ -n "$1" ]
then
  repertoire=$1      # Si le nom du répertoire donné en argument au script...
else
  repertoire=$PWD    # Sinon, utilise le répertoire courant.
fi

#  Suppose que tous les fichiers du répertoire cible sont des fichiers image
# + MacPaint avec un suffixe de nom de fichier ".mac".

for fichier in $repertoire/*  # Filename globbing.
do
  nomfichier=${fichier%.*c} #  Supprime le suffixe ".mac" du nom du fichier
                            #+ ('.*c' correspond à tout ce qui se trouve
                            #+ entre '.' et 'c', inclus).
  $OPERATION $fichier > $nomfichier.$SUFFIXE
    # Redirige la conversion vers le nouveau nom du fichier.
    rm -f $fichier          # Supprime le fichier original après sa convertion.
  echo "$nomfichier.$SUFFIXE"  # Trace ce qui se passe sur stdout.
done

exit 0

# Exercice
# --------
#  À ce stade, ce script convertit *tous* les fichiers du répertoire courant.
#  Modifiez le pour qu'il renomme *seulement* les fichiers dont l'extension est
#+ ".mac".
```

Exemple 9.13. Convertir des fichiers audio en *ogg*

```
#!/bin/bash
# ra2ogg.sh : Convertit des fichiers audio de streaming (*.ra) en ogg.

# Utilise le programme "mplayer" :
#      http://www.mplayerhq.hu/homepage
#      Vous aurez peut-être besoin d'installer les codecs appropriés
#+     pour que ce script fonctionne.
# Utilise la bibliothèque "ogg" et "oggenc" :
#      http://www.xiph.org/


PREFIXE_FICHIER_RESULTAT=${1%%ra}      # Supprime le suffixe "ra".
```

```
SUFFIXE_FICHIER_RESULTAT=wav           # Suffixe pour le fichier wav.
FICHIER_RESULTAT="$PREFIXE_FICHIER_RESULTAT""$SUFFIXE_FICHIER_RESULTAT"
E_SANSARGS=65

if [ -z "$1" ]          # Un nom de fichier à convertir doit être spécifié.
then
  echo "Usage: `basename $0` [nom_fichier]"
  exit $E_SANSARGS
fi


###############################################################################
mplayer "$1" -ao pcm:file=$FICHIER_RESULTAT
oggenc "$FICHIER_RESULTAT"  # Corrige l'extension du fichier ajoutée automatiquement
pas oggenc.
###############################################################################

rm "$FICHIER_RESULTAT"      # Supprime le fichier temporaire *.wav.
                  # Si vous voulez le conserver, commentez la ligne ci-dessus.

exit $?

#  Note :
#  -----
#  Sur un site web, cliquer seulement sur un fichier audio *.ram
#+ récupère l'URL du fichier audio, le fichier *.ra.
#  Vous pouvez ensuite utiliser "wget" ou un autre outil similaire
#+ pour télécharger vous-même le fichier *.ra.


#  Exercices :
#  ----------
#  Actuellement, ce script convertit seulement les noms de fichier *.ra.
#  Ajoutez de la flexibilité en autorisant l'utilisation de *.ram et d'autres noms
de fichier.
#
#  Si vous êtes réellement ambitieux, étendez le script pour réaliser
automatiquement
#+ les téléchargements et les convertions des fichiers audio.
#  À partir d'une URL, récupérez les fichiers audio (en utilisant "wget")
#+ et convertissez-les.
```

Une simple émulation de getopt en utilisant des constructions d'extraction de sous-chaînes.

Exemple 9.14. Émuler *getopt*

```
#!/bin/bash
# getopt-simple.sh
# Auteur : Chris Morgan
# Utilisé dans le guide ABS avec sa permission.


getopt_simple()
{
    echo "getopt_simple()"
    echo "Les paramètres sont '$*'"
    until [ -z "$1" ]
    do
      echo "Traitement du paramètre : '$1'"
      if [ ${1:0:1} = '/' ]
      then
          tmp=${1:1}              # Supprime le '/' devant...
          parametre=${tmp%%=*}     # Extrait le nom.
          valeur=${tmp##*=}        # Extrait la valeur.
          echo "Paramètre : '$parametre', valeur: '$valeur'"
          eval $parametre=$valeur
      fi
```

```
      shift
    done
}

# Passe toutes les options à getopt_simple().
getopt_simple $*

echo "test vaut '$test'"
echo "test2 vaut '$test2'"

exit 0

---

sh getopt_example.sh /test=valeur1 /test2=valeur2

Les paramètres sont '/test=valeur1 /test2=valeur2'
Traitement du paramètre : '/test=valeur1'
Paramètre : 'test', valeur: 'valeur1'
Traitement du paramètre : '/test2=valeur2'
Paramètre : 'test2', valeur : 'valeur2'
test vaut 'valeur1'
test2 vaut 'valeur2'
```

Remplacement de sous-chaîne

${chaine/souschaine/remplacement}

Remplace la première correspondance de *`$souschaine`* par *`$remplacement`*.

${chaine//souschaine/remplacement}

Remplace toutes les correspondances de *`$souschaine`* avec *`$remplacement`*.

```
chaineZ=abcABC123ABCabc

echo ${chaineZ/abc/xyz}           # xyzABC123ABCabc
                                  #  Remplace la première correspondance de
                                  #+ 'abc' avec 'xyz'.

echo ${chaineZ//abc/xyz}          # xyzABC123ABCxyz
                                  #  Remplace toutes les correspondances de
                                  #+ 'abc' avec 'xyz'.
```

${chaine/#souschaine/remplacement}

Si *`$souschaine`* correspond au *début* de *`$chaine`*, substitue *`$remplacement`* à *`$souschaine`*.

${chaine/%souchaine/remplacement}

Si *`$souschaine`* correspond à la *fin* de *`$chaine`*, substitue *`$remplacement`* à *`$souschaine`*.

```
chaineZ=abcABC123ABCabc

echo ${chaineZ/#abc/XYZ}          # XYZABC123ABCabc
                                  #  Remplace la correspondance de fin de
                                  #+ 'abc' avec 'XYZ'.

echo ${chaineZ/%abc/XYZ}          # abcABC123ABCXYZ
                                  #  Remplace la correspondance de fin de
                                  #+ 'abc' avec 'XYZ'.
```

9.2.1. Manipuler des chaînes de caractères avec awk

Un script Bash peut utiliser des fonctionnalités de manipulation de chaînes de caractères de awk634 comme alternative à ses propres fonctions intégrées.

Exemple 9.15. Autres moyens d'extraire des sous-chaînes

```
#!/bin/bash
# substring-extraction.sh

Chaine=23skidoo1
#      012345678    Bash
#      123456789    awk
# Notez les différents systèmes d'indexation de chaînes :
# Bash compte le premier caractère d'une chaîne avec '0'.
# Awk  compte le premier caractère d'une chaîne avec '1'.

echo ${Chaine:2:4} # position 3 (0-1-2), longueur de quatre caractères
                                        # skid

# L'équivalent awk de ${string:position:longueur} est substr(string,position,longueur).
echo | awk '
{ print substr("'"${Chaine}"'",3,4)      # skid
}
'
#  Envoyé un "echo" vide à awk donne une entrée inutile, et donc permet d'éviter
#+ d'apporter un nom de fichier.

exit 0
```

9.2.2. Discussion plus avancée

Pour plus d'informations sur la manipulation des chaînes de caractères dans les scripts, référez-vous à la Section 9.3, « Substitution de paramètres » et à la section consacrée à la commande expr. Pour des exemples de scripts, jetez un oeil sur les exemples suivants :

1. Exemple 15.9, « Utiliser *expr* »
2. Exemple 9.18, « Longueur d'une variable »
3. Exemple 9.19, « Correspondance de modèle dans la substitution de paramètres »
4. Exemple 9.20, « Renommer des extensions de fichiers : »
5. Exemple 9.22, « Modèles correspondant au préfixe ou au suffixe d'une chaîne de caractères »
6. Exemple A.38, « Tri d'insertion »
7. Exemple A.41, « Quacky : un jeu de mots de type Perquackey »

9.3. Substitution de paramètres

Manipuler et/ou étendre les variables

`${parametre}`
Identique à *`$parametre`*, c'est-à-dire la valeur de la variable *`parametre`*. Dans certains contextes, seule la forme la moins ambiguë, *`${parametre}`*, fonctionne.

Peut être utilisé pour concaténer des variables avec des suites de caractères (strings).

```
votre_id=${USER}-sur-${HOSTNAME}
echo "$votre_id"
#
echo "Ancien \$PATH = $PATH"
PATH=${PATH}:/opt/bin  #Ajoute /opt/bin à $PATH pour toute la durée du script.
echo "Nouveau \$PATH = $PATH"
```

`${parametre-defaut}`, **`${parametre:-defaut}`**
Si parametre n'est pas initialisé, utilise defaut.

```
echo ${nom_utilisateur-`whoami`}
# Affichez le résultat de `whoami`, si la variable $nom_utilisateur n'est
toujours pas initialisée.
```

Note

`${parametre-defaut}` et *`${parametre:-defaut}`* sont pratiquement équivalents. Le caractère : supplémentaire fait une différence seulement lorsque *`parametre`* a été déclaré mais est nul.

```
#!/bin/bash
# param-sub.sh

#  Qu'une variable ait été déclarée ou non
#+ a un effet sur le déclenchement de l'option par défaut,
#+ y compris si la variable est nulle.

nomutilisateur0=
echo "nomutilisateur0 a été déclaré mais laissé sans valeur."
echo "nomutilisateur0 = ${nomutilisateur0-`whoami`}"
# Rien ne s'affiche.

echo

echo "nomutilisateur1 n'a pas été déclaré."
echo "nomutilisateur1 = ${nomutilisateur1-`whoami`}"
# S'affiche.

nomutilisateur2=
echo "nomutilisateur2 a été déclaré mais laissé sans valeur."
echo "nomutilisateur2 = ${nomutilisateur2:-`whoami`}"
#                                    ^
# S'affiche à cause du :- au lieu du simple - dans le test conditionnel.
# Comparez à la première instance ci-dessus.

#

# Une fois encore :

variable=
# variable a été déclaré mais est initialisé à null.

echo "${variable-0}"    # (pas de sortie)
echo "${variable:-1}"   # 1
#               ^

unset variable

echo "${variable-2}"    # 2
echo "${variable:-3}"   # 3

exit 0
```

La construction du *paramètre par défaut* a pour principale utilité de fournir les arguments « manquants » de la ligne de commande des scripts.

```
NOM_FICHIER_PAR_DEFAUT=donnees.generiques
nom_fichier=${1:-$NOM_FICHIER_PAR_DEFAUT}
#  S'il n'est pas spécifié, l'ensemble de commandes suivantes opère sur le
#  fichier "donnees.generiques".
#
#  Les commandes suivent.
```

Voir aussi l'Exemple 3.4, « Sauvegarde de tous les fichiers modifiés dans les dernières 24 heures », l'Exemple 28.2, « Créer un fichier de swap en utilisant `/dev/zero` » et l'Exemple A.6, « *collatz* : Séries de Collatz ».

Comparez cette méthode avec l'utilisation d'une *liste ET* pour fournir un argument par défaut à la ligne de commande373.

${parametre=defaut}**, **${parametre:=defaut}
Si le paramètre n'est pas initialisé, alors initialisation à la valeur par *défaut*.

Les deux formes sont pratiquement équivalentes. Le caractère : fait une différence seulement lorsque `$parametre` a été déclaré et est nul, [5] comme ci-dessus.

```
echo ${nom_utilisateur=`whoami`}
# La variable "nom_utilisateur" est maintenant initialisée à `whoami`.
```

${parametre+valeur_alt}**, **${parametre:+valeur_alt}
Si le paramètre est déclaré, utilisez **valeur_alt**, sinon utilisez la chaîne de caractères vide.

Les deux formes sont pratiquement équivalentes. Le caractère : fait la différence seulement lorsque *parametre* a été déclaré nul, voir plus bas.

```
echo "###### \${parametre+valeur_alt} ########"
echo

a=${param1+xyz}
echo "a = $a"      # a =

param2=
a=${param2+xyz}
echo "a = $a"      # a = xyz

param3=123
a=${param3+xyz}
echo "a = $a"      # a = xyz

echo
echo "###### \${parametre:+valeur_alt} ########"
echo

a=${param4:+xyz}
echo "a = $a"      # a =

param5=
a=${param5:+xyz}
echo "a = $a"      # a =
# Résultats différents pour a=${param5+xyz}

param6=123
a=${param6:+xyz}
echo "a = $a"      # a = xyz
```

${parametre?msg_err}**, **${parametre:?msg_err}
Si le paramètre est initialisé, l'utilise, sinon affiche msg_err.

Les deux formes sont pratiquement équivalentes. Le caractère : fait la différence seulement lorsque *parametre* a été déclaré nul, comme ci-dessus.

Exemple 9.16. Utiliser la substitution et les messages d'erreur

```
#!/bin/bash

# Vérifier certaines des variables d'environnements du système.
# C'est une mesure adéquate de maintenance préventive.
#  Si, par exemple, $USER, le nom de la personne sur la console, n'est pas
#+ initialisé, la machine ne vous reconnaîtra pas.

: ${HOSTNAME?} ${USER?} ${HOME?} ${MAIL?}
  echo
```

[5] Si $parametre est nul dans un script non interactif, il se terminera avec un code de retour 127 (le code d'erreur de Bash pour « commande introuvable »).

```
  echo "Le nom de la machine est $HOSTNAME."
  echo "Vous êtes $USER."
  echo "Votre répertoire personnel est $HOME."
  echo "Votre courrier est situé dans $MAIL."
  echo
  echo "Si vous lisez ce message, les variables d'environnement "
  echo "critiques ont été initialisées."
  echo
  echo

# ------------------------------------------------------

#  La construction ${variablename?} peut aussi vérifier les
#+ variables configurées dans un script.

CetteVariable=Valeur-de-CetteVariable
#  Notez que, du coup, les variables chaînes de caractères pourraient être
#+ configurées avec les caractères contenus dans leurs noms.
: ${CetteVariable?}
echo "La valeur de CetteVariable est $CetteVariable".
echo
echo


: ${ZZXy23AB?"ZZXy23AB n'a pas été initialisée."}
#  Si ZZXy23AB n'a pas été initialisée, alors le script se termine avec un
#+ message d'erreur.

# Vous pouvez spécifier le message d'erreur.
# : ${nomvariable?"MESSAGE D'ERREUR."}


# Même résultat avec :  variable_stupide=${ZZXy23AB?}
#                       variable_stupide=${ZZXy23AB?"ZXy23AB n'a pas été initialisée."}
#
#                       echo ${ZZXy23AB?} >/dev/null

#  Comparez ces méthodes de vérification sur l'initialisation d'une variable
#+ avec "set -u" ...



echo "Vous ne verrez pas ce message parce que le script s'est déjà terminé"

ICI=0
exit $ICI   # Ne sortira *pas* ici.

# En fait, ce script quittera avec un code de sortie 1 (echo $?).
```

Exemple 9.17. Substitution de paramètres et messages d'« usage »

```
#!/bin/bash
# usage-message.sh

: ${1?"Usage: $0 ARGUMENT"}
#  Le script sort ici si le paramètre en ligne de commande est absent,
#+ avec le message d'erreur suivant.
#    usage-message.sh: 1: Usage: usage-message.sh ARGUMENT

echo "Ces deux lignes ne s'affichent que si le paramètre en ligne de commande est
donné."
echo "paramètre en ligne de commande = \"$1\""

exit 0 # Sortira ici seulement si le paramètre en ligne de commande est présent.


#  Vérifiez le code de sortie, à la fois avec et sans le paramètre en ligne de
```

```
#+ commande.
# Si le paramètre en ligne de commande est présent, alors "$?" vaut 0.
# Sinon, "$?" vaut 1.
```

Substitution de paramètres et/ou expansion. Les expressions suivantes sont le complément des opérations sur les suites de caractères comme **match** *dans* **expr** (voir l'Exemple 15.9, « Utiliser *expr* »). Ces derniers sont utilisés principalement pour analyser les chemins de fichiers.

Longueur de variables / Suppression d'un sous-ensemble d'une suite de caractères

`${#var}`
Longueur de la suite de caractères (ou nombre de caractères dans `$var`). Pour un tableau375 **${#tableau}** est la longueur du premier élément dans le tableau.

Note

Exceptions :

- **${#*}** et **${#@}** donnent le *nombre de paramètres de position*.
- Pour un tableau, **${#tableau[*]}** et **${#tableau[@]}** donnent le nombre d'éléments dans le tableau.

Exemple 9.18. Longueur d'une variable

```
#!/bin/bash
# length.sh

E_SANS_ARGS=65

if [ $# -eq 0 ]  # Doit avoir des arguments en ligne de commande.
then
  echo "Merci d'appeler ce script avec un ou plusieurs argument(s) en ligne de
commande."
  exit $E_SANS_ARGS
fi

var01=abcdEFGH28ij
echo "var01 = ${var01}"
echo "Longueur de var01 = ${#var01}"
# Maintenant, essayons d'intégrer une espace.
var02="abcd EFGH28ij"
echo "var02 = ${var02}"
echo "Longueur de var02 = ${#var02}"

echo "Nombre d'arguments en ligne de commande passés au script = ${#@}"
echo "Nombre d'arguments en ligne de commande passés au script = ${#*}"

exit 0
```

`${var#Modele}`, **`${var##Modele}`**
Supprime à partir de `$var` la plus *courte* partie de `$Modele` qui correspond au *`début`* de `$var`.

${var##Modele} Supprime de `$var` la *plus longue* partie de `$Modele` qui correspond au *`début`* de `$var`.

Un exemple d'usage à partir de l'Exemple A.7, « *days-between* : Calculer le nombre de jours entre deux dates » :

```
# Fonction provenant de l'exemple "days-between.sh"
# Supprimer les zéros du début à partir de l'argument donné.

supprimer_les_zeros_du_debut () # Supprime les zéros éventuels au début
{                       # à partir des arguments donnés.
  return=${1#0}         # Le "1 correspond à "$1", argument donné.
                        # Le "0" correspond à ce qui doit être supprimé de "$1".
}
```

Une version plus élaborée par Manfred Schwarb :

```
supprimer_les_zeros_du_debut_2 () # Supprimer les zéros du début, car sinon
{                       # Bash interprétera de tels numéros en valeurs octales.
  shopt -s extglob      # Active le globbing local.
  local val=${1##+(0)} # Utilise une variable locale, série correspondante la
                        # plus longue avec des 0.
  shopt -u extglob      # Désactive le globbing local.
  _strip_leading_zero2=${val:-0}
                        # Si l'entrée était 0, renvoie 0 au lieu de "".
}
```

Un autre exemple d'usage :

```
echo `basename $PWD`        # Base du nom du répertoire courant.
echo "${PWD##*/}"           # Base du nom du répertoire actuel.
echo
echo `basename $0`          # Nom du script.
echo $0                     # Nom du script.
echo "${0##*/}"             # Nom du script.
echo
filename=test.data
echo "${filename##*.}"      # données
                            # Extension du fichier.
```

${var%Modele}**, **${var%%Modele}

{$var%Modele} Supprime de `$var` la *plus petite* partie de `$Modele` correspondant à la *fin* de `$var`.

{$var%%Modele} Supprime de `$var` la *plus grande* partie de `$Modele` qui corresponde à la *fin* de `$var`.

La version 2465 de Bash a ajouté des options supplémentaires.

Exemple 9.19. Correspondance de modèle dans la substitution de paramètres

```
#!/bin/bash
# patt-matching.sh

# Reconnaissance de modèles en utilisant les opérateurs de substitution # ## % %%

var1=abcd12345abc6789
modele1=a*c  # * (joker) recherche tout ce qui se trouve entre a - c.

echo
echo "var1 = $var1"           # abcd12345abc6789
echo "var1 = ${var1}"         # abcd12345abc6789   (autre forme)
echo "Nombre de caractères dans ${var1} = ${#var1}"
echo

echo "modele1 = $modele1"     # a*c  (tout entre 'a' et 'c')
echo "------------------"
echo '${var1#$modele1}  =' "${var1#$modele1}"    #
d12345abc6789
#  Correspondance la plus petite, supprime les trois premiers caractères de
abcd12345abc6789
#                                                  ^^^^^^^^              |-|
echo '${var1##$modele1} =' "${var1##$modele1}"   #
6789
#  Correspondance la plus grande possible, supprime les 12 premiers caractères de
abcd12345abc6789
#                                                        ^^^^^^^^
|----------|

echo; echo; echo

modele2=b*9             # tout entre 'b' et '9'
echo "var1 = $var1"     # Toujours  abcd12345abc6789
echo
```

```
echo "modele2 = $modele2"
echo "------------------"
echo '${var1%modele2}  =' "${var1%$modele2}"     #                        abcd12345a
#  Correspondance la plus petite, supprime les six derniers caractères de
abcd12345abc6789
#                                                  ^^^^^^^^                    |----|
echo '${var1%%modele2} =' "${var1%%$modele2}"    #                       a
#  Correspondance la plus grande, supprime les douze derniers caractères de
abcd12345abc6789
#                                                    ^^^^^^^^
|-------------|

#  Souvenez-vous, # et ## fonctionnent à partir de la gauche (début) de la chaîne
#                 % et %% fonctionnent à partir de la droite.

echo

exit 0
```

Exemple 9.20. Renommer des extensions de fichiers :

```
#!/bin/bash

# rfe.sh : Renommer les extensions de fichier (Renaming File Extensions).
#
#         rfe ancienne_extension nouvelle_extension
#
# Exemple :
# Pour renommer tous les fichiers *.gif d'un répertoire en *.jpg,
#          rfe gif jpg

E_MAUVAISARGS=65

case $# in
  0|1)            # La barre verticale signifie "ou" dans ce contexte.
  echo "Usage: `basename $0` ancien_suffixe nouveau_suffixe"
  exit $E_MAUVAISARGS  # Si 0 ou 1 argument, alors quitter.
  ;;
esac


for fichier in *.$1
# Traverse la liste des fichiers dont le nom termine avec le premier argument.
do
  mv $fichier ${fichier%$1}$2
  #  Supprime la partie du fichier contenant le premier argument
  #  puis ajoute le deuxième argument.
done

exit 0
```

Expansion de variables / Remplacement de sous-chaînes

Ces constructions proviennent de *ksh*.

`${var:pos}`
La variable *`var`* étendue, commençant à la position *`pos`*.

`${var:pos:len}`
Augmentation d'un maximum de *`len`* caractères de la variable *`var`*, à partir de la position *`pos`*. Voir l'Exemple A.14, « *password*: Générer des mots de passe aléatoires de 8 caractères » pour un exemple d'utilisation particulièrement intéressante de cet opérateur.

`${var/Modele/Remplacement}`
Première occurrence de *`Modele`*, à l'intérieur de *`var`* remplacé par *`Remplacement`*.

Si *Remplacement* est omis, alors la première occurrence de *Modele* est remplacé par *rien*, c'est-à-dire qu'il est supprimé.

${var//Modele/Remplacement}

Remplacement global. Toutes les occurrences de *Modele*, à l'intérieur de *var* sont remplacées par *Remplacement*.

Comme ci-dessus, si *Remplacement* est omis, alors toutes les occurrences de *Modele* sont remplacées par *rien*, c'est-à-dire supprimées.

Exemple 9.21. Utiliser la concordance de modèles pour analyser des chaînes de caractères diverses

```
#!/bin/bash

var1=abcd-1234-defg
echo "var1 = $var1"

t=${var1#*-*}
echo "var1 (avec tout, jusqu'au et incluant le premier - supprimé) = $t"
#  t=${var1#*-}  fonctionne de la même façon,
#+ car # correspond à la plus petite chaîne de caractères,
#+ et * correspond à tout ce qui précède, incluant la chaîne vide.
# (Merci, Stéphane Chazelas, pour l'avoir indiqué.)

t=${var1##*-*}
echo "Si var1 contient un \"-\", renvoie une chaîne vide...   var1 = $t"


t=${var1%*-*}
echo "var1 (avec tout à partir de la fin - supprimé) = $t"

echo

# ------------------------------------------
nom_chemin=/home/bozo/idees/pensees.pour.aujourdhui
# ------------------------------------------
echo "nom_chemin = $nom_chemin"
t=${nom_chemin##/*/}
echo "nom_chemin, sans les préfixes = $t"
# Même effet que   t=`basename $nom_chemin` dans ce cas particulier.
#  t=${nom_chemin%/}; t=${t##*/}   est une solution plus générale,
#+ mais elle échoue quelques fois.
#  Si $nom_chemin finit avec un retour chariot, alors `basename $nom_chemin`
#+ ne fonctionnera pas mais l'expression ci-dessus le fera.
# (Merci, S.C.)

t=${nom_chemin%/*.*}
# Même effet que t=`dirname $nom_chemin`
echo "nom_chemin, sans les suffixes = $t"
# Ceci va échouer dans certains cas, comme "../", "/foo////", # "foo/", "/".
#  Supprimer les suffixes, spécialement quand le nom de base n'en a pas, mais
#+ que le nom du répertoire en a un, complique aussi le problème.
# (Merci, S.C.)

echo

t=${nom_chemin:11}
echo "$nom_chemin, avec les 11 premiers caractères supprimés = $t"
t=${nom_chemin:11:5}
echo "$nom_chemin, avec les 11 premiers caractères supprimés, longueur 5 = $t"

echo

t=${nom_chemin/bozo/clown}
echo "$nom_chemin avec \"bozo\" remplacé par \"clown\" = $t"
t=${nom_chemin/today/}
echo "$nom_chemin avec \"today\" supprimé = $t"
t=${nom_chemin//o/O}
echo "$nom_chemin avec tous les o en majuscule = $t"
t=${nom_chemin//o/}
echo "$nom_chemin avec tous les o supprimés = $t"
```

```
exit 0
```

${var/#Modele/Remplacement}
Si le *préfixe* de *var* correspond à *Modele*, alors *Remplacement* remplace *Modele*.

${var/%Modele/Remplacement}
Si le *suffixe* de *var* correspond à *Modele*, alors *Remplacement* remplace *Modele*.

Exemple 9.22. Modèles correspondant au préfixe ou au suffixe d'une chaîne de caractères

```
#!/bin/bash
# varmatch.sh
#  Démonstration de remplacement de modèle sur le préfixe / suffixe d'une chaîne de
#+ caractères.

v0=abc1234zip1234abc    # Variable originale.
echo "v0 = $v0"         # abc1234zip1234abc
echo

# Correspond au préfixe (début) d'une chaîne de caractères.
v1=${v0/#abc/ABCDEF}    # abc1234zip1234abc
                        # |-|
echo "v1 = $v1"         # ABCDEF1234zip1234abc
                        # |----|

# Correspond au suffixe (fin) d'une chaîne de caractères.
v2=${v0/%abc/ABCDEF}    # abc1234zip1234abc
                        #                |-|
echo "v2 = $v2"         # abc1234zip1234ABCDEF
                        #                |----|

echo

#  --------------------------------------------------
#  Doit correspondre au début / fin d'une chaîne de caractères.
#  sinon aucun remplacement ne se fera.
#  --------------------------------------------------
v3=${v0/#123/000}       # Correspond, mais pas au début.
echo "v3 = $v3"         # abc1234zip1234abc
                        # PAS DE REMPLACEMENT.
v4=${v0/%123/000}       # Correspond, mais pas à la fin.
echo "v4 = $v4"         # abc1234zip1234abc
                        # PAS DE REMPLACEMENT.

exit 0
```

${!varprefixe*}**, **${!varprefixe@}
Établit une correspondance des *noms* de toutes les variables déjà déclarées commençant par *varprefixe*.

```
xyz23=quoiquecesoit
xyz24=

a=${!xyz*}      # Se développe en les *noms* des variables précédemment déclarées
                # commençant par "xyz".
echo "a = $a"   # a = xyz23 xyz24
a=${!xyz@}      # Même chose que ci-dessus.
echo "a = $a"   # a = xyz23 xyz24

# Bash, version 2.04, ajoute cette fonctionnalité.
```

9.4. Typer des variables : declare ou typeset

Les commandes internes[158] *declare* et *typeset*, qui sont des synonymes exactes, permettent de modifier les propriétés de variables.

Ceci est une forme très faible de la *saisie* [6] disponible dans certains langages de programmation. La commande *declare* est spécifique aux versions 2 et ultérieures de Bash. La commande *typeset* fonctionne aussi dans les scripts ksh.

Options pour declare/typeset

-r *lecture seule*

```
declare -r var1
```

(**`declare -r var1`** fonctionne de la même façon que **`readonly var1`**)

Ceci est l'équivalent du qualificateur C **const**. Une tentative de modification de la valeur d'une variable en lecture seule échoue avec un message d'erreur.

-i *entier*

```
declare -i nombre
# Ce script va traiter les occurrences suivantes de "nombre" comme un entier.

nombre=3
echo "Nombre = $nombre"     # Nombre = 3

nombre=trois
echo "Nombre = $nombre"     # Nombre = 0
# Essaie d'évaluer la chaîne "trois" comme un entier.
```

Certaines opérations arithmétiques sont permises pour des variables déclarées entières sans avoir besoin de expr ou de let.

```
n=6/3
echo "n = $n"       # n = 6/3

declare -i n
n=6/3
echo "n = $n"       # n = 2
```

-a *tableau (array)*

```
declare -a index
```

La variable *index* sera traitée comme un tableau375.

-f *fonction(s)*

```
declare -f
```

Une ligne **`declare -f`** sans argument dans un script donnera une liste de toutes les fonctions352 définies auparavant dans ce script.

```
declare -f nom_fonction
```

Un **`declare -f nom_fonction`** dans un script liste simplement la fonction nommée.

-x export

```
declare -x var3
```

Ceci déclare la disponibilité d'une variable pour une exportation en dehors de l'environnement du script lui-même.

-x var=$value

```
declare -x var3=373
```

La commande **declare** permet d'assigner une valeur à une variable lors de sa déclaration.

Exemple 9.23. Utiliser *declare* pour typer des variables

[6] Dans ce contexte, *saisir* une variable signifie la classifier et restreindre ses propriétés. Par exemple, une variable *declared* ou *typed* en tant qu'entier n'est plus disponible pour les opérations de chaîne.

```
#!/bin/bash

fonc1 ()
{
echo Ceci est une fonction.
}

declare -f        # Liste la fonction ci-dessus.

echo

declare -i var1   # var1 est un entier.
var1=2367
echo "var1 déclaré comme $var1"
var1=var1+1       # La déclaration d'un entier élimine le besoin d'utiliser let.
echo "var1 incrémenté par 1 vaut $var1."
# Essai de modification de la variable déclarée comme entier.
echo "Essai de modification de var1 en une valeur flottante, 2367.1."
var1=2367.1       # Résultat: un message d'erreur, et une variable non modifiée.
echo "var1 vaut toujours $var1"

echo

declare -r var2=13.36          #  'declare' permet de configurer une variable
                               #+ proprement et de lui affecter une valeur.
echo "var2 déclaré comme $var2" #  Essai de modification d'une valeur en lecture
                               #+ seule.
var2=13.37                     # Génère un message d'erreur, et sort du script.

echo "var2 vaut toujours $var2" # Cette ligne ne s'exécutera pas.

exit 0                         # Le script ne terminera pas ici.
```

Attention

Utiliser la commande interne *declare* restreint la portée344 d'une variable.

```
foo ()
{
FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar   # Affiche bar.
```

Néanmoins...

```
foo (){
declare FOO="bar"
}

bar ()
{
foo
echo $FOO
}

bar  # N'affiche rien.


# Merci pour cette indication, Michael Iatrou.
```

9.5. Références indirectes

Supposez que la valeur d'une variable soit le nom d'une seconde variable. Est-il possible de retrouver la valeur de cette deuxième variable à partir de la première ? Par exemple, si *a=lettre_de_l_alphabet* et *lettre_de_l_alphabet=z*, est-ce qu'une référence à *a* pourrait renvoyer *z* ? En fait, c'est possible et cela s'appelle une *référence indirecte*. On utilise la notation inhabituelle *eval var1=\$$var1*.

Exemple 9.24. Références indirectes aux variables

```
#!/bin/bash
# index-ref.sh : Référencement de variable indirecte.
# Accéder au contenu du contenu d'une variable.

a=lettre_de_l_alphabet # La variable a contient le nom d'une autre variable.
lettre_de_l_alphabet=z

echo

# Référence directe.
echo "a = $a"          # a = lettre_de_l_alphabet

# Référence indirecte.
eval a=\$$a
echo "Maintenant, a = $a" # Maintenant, a = z

echo


# Maintenant, essayons de changer la référence du deuxième.

t=tableau_cellule_3
tableau_cellule_3=24
echo "\"tableau_cellule_3\" = $tableau_cellule_3"  # "tableau_cellule_3" = 24
echo -n "\"t\" déréférencé  = "; eval echo \$$t    # "t" déréférencé  = 24
# Dans ce cas simple, ce qui suit fonctionne aussi (pourquoi ?).
#   eval t=\$$t; echo "\"t\" = $t"

echo

t=tableau_cellule_3
NOUVELLE_VALEUR=387
tableau_cellule_3=$NOUVELLE_VALEUR
echo "Modification de la valeur de \"tableau_cellule_3\" en $NOUVELLE_VALEUR."
echo "\"tableau_cellule_3\" vaut maintenant $tableau_cellule_3"
echo -n "\"t\" déréférencé maintenant "; eval echo \$$t
# "eval" prend deux arguments "echo" et "\$$t" (valeur égale à $tableau_cellule_3)

echo

# (Merci, Stéphane Chazelas, pour la clarification sur le comportement ci-dessus.)


#  Une autre méthode est la notation ${!t}, discutée dans la section
#+ "Bash, version 2".
#  Voir aussi ex78.sh.

exit 0
```

Quel est l'utilité du référencement indirect des variables ? Cela donne à Bash une partie de la fonctionnalité des pointeurs28, comme en *C*, par exemple dans la recherche dans des tables. Et, cela a aussi quelques autres applications intéressantes...

Nils Radtke montre comment construire des noms de variables « dynamiques » et comment évaluer leur contenu. Ceci peut être utile lors de l'intégration de fichiers de configuration.

```
#!/bin/bash
```

```
# ------------------------------------------
# Ceci pourrait être "récupéré" d'un fichier séparé.
isdnMonFournisseurDistant=172.16.0.100
isdnTonFournisseurDistant=10.0.0.10
isdnServiceInternet="MonFournisseur"
# ------------------------------------------


netDistant=$(eval "echo \$$(echo isdn${isdnServiceInternet}Distant)")
netDistant=$(eval "echo \$$(echo isdnMonFournisseurDistant)")
netDistant=$(eval "echo \$isdnMonFournisseurDistant")
netDistant=$(eval "echo $isdnMonFournisseurDistant")

echo "$netDistant"    # 172.16.0.100

# ============================================================

#  Et cela devient encore meilleur.

#  Considérez l'astuce suivant étant donnée une variable nommée getSparc,
#+ mais sans variable getIa64 :

chkMirrorArchs () {
  arch="$1";
  if [ "$(eval "echo \${$(echo get$(echo -ne $arch |
        sed 's/^\(.\).*/\1/g' | tr 'a-z' 'A-Z'; echo $arch |
        sed 's/^.\(.*\)/\1/g')):-false}")" = true ]
  then
    return 0;
  else
    return 1;
  fi;
}

getSparc="true"
unset getIa64
chkMirrorArchs sparc
echo $?        # 0
               # True

chkMirrorArchs Ia64
echo $?        # 1
               # False

# Notes :
# ------
# Même la partie du nom de la variable à substituer est construite explicitement.
# Les paramètres des appels de chkMirrorArchs sont tous en minuscule.
# Le nom de la variable est composé de deux parties : "get" et "Sparc" . . .
```

Exemple 9.25. Passer une référence indirecte à *awk*

```
#!/bin/bash

# Une autre version du script "column totaler"
# qui ajoute une colonne spécifiée (de nombres) dans le fichier cible.
# Celui-ci utilise les références indirectes.

ARGS=2
E_MAUVAISARGS=65

if [ $# -ne "$ARGS" ] # Vérifie le bon nombre d'arguments sur la ligne de
                      # commande.
then
   echo "Usage: `basename $0` nomfichier numéro_colonne"
   exit $E_MAUVAISARGS
fi
```

```
nomfichier=$1
numero_colonne=$2

#===== Identique au script original, jusqu'à ce point =====#


# Un script multi-ligne est appelé par awk ' ..... '


# Début du script awk.
# ------------------------------------------------
awk "

{ total += \$${numero_colonne} # référence indirecte
}
END {
     print total
     }

     " "$nomfichier"
# ------------------------------------------------
# Fin du script awk.

# La référence de variable indirecte évite les problèmes de
# référence d'une variable shell à l'intérieur d'un script embarqué.
# Merci, Stephane Chazelas.


exit 0
```

Attention

Cette méthode de référence indirecte est un peu délicate. Si la variable de second ordre change de valeur, alors la variable de premier ordre doit être correctement déréférencée (comme sur l'exemple ci-dessus). Heureusement, la notation *`${!variable}`* introduite avec la version 2465 de Bash (voir l'Exemple 34.2, « Références de variables indirectes - la nouvelle façon » et l'Exemple A.24, « Encore plus sur les fonctions de hachage ») rend les références indirectes plus intuitives.

Bash ne supporte pas l'arithmétique des pointeurs et cela limite de façon sévère l'utilité du référencement indirect. En fait, le référencement indirect dans un langage de scripts est, au mieux, un agglomérat monstrueux.

9.6. $RANDOM : générer un nombre aléatoire

`$RANDOM` est une fonction352 interne Bash (pas une constante) renvoyant un entier *pseudo-aléatoire* [7] dans l'intervalle 0 - 32767. Il ne devrait *`pas`* être utilisé pour générer une clé de chiffrement.

Exemple 9.26. Générer des nombres aléatoires

```
#!/bin/bash

# $RANDOM renvoie un entier différent à chaque appel.
# Échelle : 0 - 32767 (entier signé sur 16 bits).

NBMAX=10
index=1

echo
echo "$NBMAX nombres aléatoires :"
echo "-----------------"
```

[7] Un vrai « hasard », si tant est qu'il puisse exister, peut seulement être trouvé dans certains phénomènes naturels compris partiellement tels que la destruction radioactive. Les ordinateurs simulent le hasard et les séquences générées par ordinateur de nombres « aléatoires » sont du coup appelés *pseudo-aléatoires*.

```
while [ "$index" -le $NBMAX ]      # Génère 10 ($NBMAX) entiers aléatoires.
do
  nombre=$RANDOM
  echo $nombre
  let "index += 1"  # Incrémente l'index.
done
echo "-----------------"

#  Si vous avez besoin d'un entier aléatoire dans une certaine échelle, utilisez
#+ l'opérateur 'modulo'.
#  Il renvoie le reste d'une division.

ECHELLE=500

echo

nombre=$RANDOM
let "nombre %= $ECHELLE"
#           ^^
echo "Nombre aléatoire inférieur à $ECHELLE  ---  $nombre"

echo

#  Si vous avez besoin d'un entier aléatoire supérieur à une borne, alors
#+ faites un test pour annuler tous les nombres en dessous de cette borne.

PLANCHER=200

nombre=0   #initialise
while [ "$nombre" -le $PLANCHER ]
do
  nombre=$RANDOM
done
echo "Nombre aléatoire supérieur à $PLANCHER ---  $nombre"
echo

   # Examinons une alternative simple  à la boucle ci-dessus
   #       let "nombre = $RANDOM + $PLANCHER"
   # Ceci éliminerait la boucle while et s'exécuterait plus rapidement.
   # Mais, il resterait un problème. Lequel ?



# Combine les deux techniques pour récupérer un nombre aléatoire
# compris entre deux limites.
nombre=0   #initialise
while [ "$nombre" -le $PLANCHER ]
do
  nombre=$RANDOM
  let "nombre %= $ECHELLE"  # Ramène $nombre dans $ECHELLE.
done
echo "Nombre aléatoire compris entre $PLANCHER et $ECHELLE ---  $nombre"
echo


# Génère un choix binaire, c'est-à-dire "vrai" ou "faux".
BINAIRE=2
T=1
nombre=$RANDOM

let "nombre %= $BINAIRE"
# Notez que let "nombre >>= 14"    donne une meilleure distribution aléatoire
# (les décalages droits enlèvent tout sauf le dernier nombre binaire).
if [ "$nombre" -eq $T ]
then
  echo "VRAI"
else
  echo "FAUX"
fi
```

```
echo


# Peut générer un lancer de dés
SPOTS=6   # Modulo 6 donne une échelle de 0 à 5.
          # Incrémenter de 1 donne l'échelle désirée, de 1 à 6.
          # Merci, Paulo Marcel Coelho Aragao, pour cette simplification.
die1=0
die2=0
# Serait-il mieux de seulement initialiser SPOTS=7 et de ne pas ajouter 1 ?
# Pourquoi ou pourquoi pas ?

# Jette chaque dé séparément, et donne ainsi une chance correcte.

    let "die1 = $RANDOM % $SPOTS +1" # Le premier.
    let "die2 = $RANDOM % $SPOTS +1" # Et le second.
    # Quelle opération arithmétique ci-dessus a la plus grande précédence
    # le modulo (%) ou l'addition (+) ?

let "throw = $die1 + $die2"
echo "Throw of the dice = $throw"
echo


exit 0
```

Exemple 9.27. Piocher une carte au hasard dans un tas

```
#!/bin/bash
# pick-card.sh

# Ceci est un exemple pour choisir au hasard des éléments d'un tableau.


# Prenez une carte, n'importe quelle carte.

Suites="Carreau
Pique
Coeur
Trefle"

Denominations="2
3
4
5
6
7
8
9
10
Valet
Dame
Roi
As"

# Notez que le contenu de la variable continue sur plusieurs lignes.

suite=($Suites)                # Lire dans une variable de type tableau.
denomination=($Denominations)

num_suites=${#suite[*]}        # Compter le nombre d'éléments.
num_denominations=${#denomination[*]}

echo -n "${denomination[$((RANDOM%num_denominations))]} of "
echo ${suite[$((RANDOM%num_suites))]}
```

```
# $bozo sh pick-cards.sh
# Valet de trèfle


# Merci, "jipe", pour m'avoir indiqué cette utilisation de $RANDOM.
exit 0
```

Exemple 9.28. Simulation « Brownian Motion »

```
#!/bin/bash
# brownian.sh
# Author: Mendel Cooper
# Reldate: 10/26/07
# License: GPL3

#  ---------------------------------------------------------------
#  This script models Brownian motion:
#+ the random wanderings of tiny particles in a fluid,
#+ as they are buffeted by random currents and collisions.
#+ This is colloquially known as the "Drunkard's Walk."

#  It can also be considered as a stripped-down simulation of a
#+ Galton Board, a slanted board with a pattern of pegs,
#+ down which rolls a succession of marbles, one at a time.
#+ At the bottom is a row of slots or catch basins in which
#+ the marbles come to rest at the end of their journey.
#  Think of it as a kind of bare-bones Pachinko game.
#  As you see by running the script,
#+ most of the marbles cluster around the center slot.
#+ This is consistent with the expected binomial distribution.
#  As a Galton Board simulation, the script
#+ disregards such parameters as
#+ board tilt-angle, rolling friction of the marbles,
#+ angles of impact, and elasticity of the pegs.
#  To what extent does this affect the accuracy of the simulation?
#  ---------------------------------------------------------------

PASSES=500            #  Number of particle interactions / marbles.
ROWS=10               #  Number of "collisions" (or horiz. peg rows).
RANGE=3               #  0 - 2 output range from $RANDOM.
POS=0                 #  Left/right position.
RANDOM=$$             #  Seeds the random number generator from PID
                      #+ of script.

declare -a Slots      # Array holding cumulative results of passes.
NUMSLOTS=21           # Number of slots at bottom of board.


Initialize_Slots () { # Zero out all elements of the array.
for i in $( seq $NUMSLOTS )
do
  Slots[$i]=0
done

echo                  # Blank line at beginning of run.
  }


Show_Slots () {
echo -n " "
for i in $( seq $NUMSLOTS )   # Pretty-print array elements.
do
  printf "%3d" ${Slots[$i]}   # Allot three spaces per result.
done
```

```
echo # Row of slots:
echo " |__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|__|"
echo "                                    ^^"
echo #  Note that if the count within any particular slot exceeds 99,
     #+ it messes up the display.
     #  Running only(!) 500 passes usually avoids this.
  }


Move () {              # Move one unit right / left, or stay put.
  Move=$RANDOM         # How random is $RANDOM? Well, let's see ...
  let "Move %= RANGE"  # Normalize into range of 0 - 2.
  case "$Move" in
    0 ) ;;                     # Do nothing, i.e., stay in place.
    1 ) ((POS--));;            # Left.
    2 ) ((POS++));;            # Right.
    * ) echo -n "Error ";;     # Anomaly! (Should never occur.)
  esac
  }


Play () {                      # Single pass (inner loop).
i=0
while [ "$i" -lt "$ROWS" ]       # One event per row.
do
  Move
  ((i++));
done

SHIFT=11                       # Why 11, and not 10?
let "POS += $SHIFT"            # Shift "zero position" to center.
(( Slots[$POS]++ ))            # DEBUG: echo $POS
  }


Run () {                       # Outer loop.
p=0
while [ "$p" -lt "$PASSES" ]
do
  Play
  (( p++ ))
  POS=0                        # Reset to zero. Why?
done
  }


# --------------
# main ()
Initialize_Slots
Run
Show_Slots
# --------------

exit $?

#  Exercises:
#  ---------
#  1) Show the results in a vertical bar graph, or as an alternative,
#+    a scattergram.
#  2) Alter the script to use /dev/urandom instead of $RANDOM.
#     Will this make the results more random?
```

Jipe nous a indiqué un autre ensemble de techniques pour générer des nombres aléatoires à l'intérieur d'un intervalle donné.

```
#  Génére des nombres aléatoires entre 6 et 30.
rnumber=$((RANDOM%25+6))

#  Générer des nombres aléatoires dans le même intervalle de 6 à 30,
```

```
#+ mais le nombre doit être divisible de façon exacte par 3.
rnumber=$(((RANDOM%30/3+1)*3))

# Notez que ceci ne fonctionnera pas tout le temps.
# Il échoue si $RANDOM%30 renvoie 0.

#  Frank Wang suggère l'alternative suivante :
   rnumber=$(( RANDOM%27/3*3+6 ))
```

Bill Gradwohl est parvenu à une formule améliorée fonctionnant avec les numéros positifs.

```
rnumber=$(((RANDOM%(max-min+divisiblePar))/divisiblePar*divisiblePar+min))
```

Ici, Bill présente une fonction versatile renvoyant un numéro au hasard entre deux valeurs spécifiques.

Exemple 9.29. Un nombre au hasard entre deux valeurs

```
#!/bin/bash
# random-between.sh
# Nombre aléatoire entre deux valeurs spécifiées.
# Script par Bill Gradwohl, avec des modifications mineures par l'auteur du document.
# Utilisé avec les droits.


aleatoireEntre() {
   #  Génère un numéro aléatoire positif ou négatif
   #+ entre $min et $max
   #+ et divisible par $divisiblePar.
   #  Donne une distribution "raisonnablement aléatoire" des valeurs renvoyées.
   #
   #  Bill Gradwohl - 1er octobre 2003

   syntax() {
   # Fonction imbriquée dans la fonction.
      echo
      echo    "Syntax: aleatoireEntre [min] [max] [multiple]"
      echo
      echo    "Attend au plus trois paramètres mais tous sont complètement optionnels."
      echo    "min est la valeur minimale"
      echo    "max est la valeur maximale"
      echo    "multiple spécifie que la réponse est un multiple de cette valeur."
      echo    "    c'est-à-dire qu'une réponse doit être divisible de manière entière"
      echo    "    par ce numéro."
      echo
      echo    "Si cette valeur manque, l'aire par défaut supportée est : 0 32767 1"
      echo    "Un résultat avec succès renvoie 0. Sinon, la syntaxe de la fonction"
      echo    "est renvoyée avec un 1."
      echo    "La réponse est renvoyée dans la variable globale aleatoireEntreAnswer"
      echo    "Les valeurs négatives pour tout paramètre passé sont gérées
correctement."
   }

   local min=${1:-0}
   local max=${2:-32767}
   local divisiblePar=${3:-1}
   #  Valeurs par défaut affectées, au cas où les paramètres ne sont pas passés à la
   #+ fonction.

   local x
   local spread

   # Assurez-vous que la valeur divisiblePar est positive.
   [ ${divisiblePar} -lt 0 ] && divisiblePar=$((0-divisiblePar))

   # Vérification.
   if [ $# -gt 3 -o ${divisiblePar} -eq 0 -o  ${min} -eq ${max} ]; then
      syntax
      return 1
```

```
   fi

   # Vérifiez si min et max ne sont pas inversés.
   if [ ${min} -gt ${max} ]; then
      # Les inversez.
      x=${min}
      min=${max}
      max=${x}
   fi

   #  Si min est lui-même non divisible par $divisiblePar,
   #+ alors corrigez le min pour être à l'échelle.
   if [ $((min/divisiblePar*divisiblePar)) -ne ${min} ]; then
      if [ ${min} -lt 0 ]; then
         min=$((min/divisiblePar*divisiblePar))
      else
         min=$((((min/divisiblePar)+1)*divisiblePar))
      fi
   fi

   #  Si max est lui-même non divisible par $divisiblePar,
   #+ alors corrigez le max pour être à l'échelle.
   if [ $((max/divisiblePar*divisiblePar)) -ne ${max} ]; then
      if [ ${max} -lt 0 ]; then
         max=$((((max/divisiblePar)-1)*divisiblePar))
      else
         max=$((max/divisiblePar*divisiblePar))
      fi
   fi

   #  ----------------------------------------------------------------
   #  Maintenant, pour faire le vrai travail.

   #  Notez que pour obtenir une distribution correcte pour les points finaux,
   #+ l'échelle des valeurs aléatoires doit être autorisée pour aller entre 0 et
   #+ abs(max-min)+divisiblePar, et non pas seulement abs(max-min)+1.

   #  La légère augmentation produira une distribution correcte des points finaux.

   #  Changer la formule pour utiliser abs(max-min)+1 produira toujours des réponses
   #+ correctes mais le côté aléatoire des réponses est erroné dans le fait que le
   #+ nombre de fois où les points finaux ($min et $max) sont renvoyés est
   #+ considérablement plus petit que lorsque la formule correcte est utilisée.
   #  ----------------------------------------------------------------

   spread=$((max-min))
   #  Omair Eshkenazi indique que ce test n'est pas nécessaire
   #+ car max et min ont déjà été basculés.
   [ ${spread} -lt 0 ] && spread=$((0-spread))
   let spread+=divisiblePar
   aleatoireEntreAnswer=$(((RANDOM%spread)/divisiblePar*divisiblePar+min))

   return 0

   #  Néanmoins, Paulo Marcel Coelho Aragao indique que
   #+ quand $max et $min ne sont pas divisibles par $divisiblePar,
   #+ la formule échoue.
   #
   #  Il suggère à la place la formule suivante :
   #    rnumber = $(((RANDOM%(max-min+1)+min)/divisiblePar*divisiblePar))

}

# Testons la fonction.
min=-14
max=20
divisiblePar=3
```

```
#  Génère un tableau des réponses attendues et vérifie pour s'assurer que nous obtenons
#+ au moins une réponse si nous bouclons assez longtemps.

declare -a reponse
minimum=${min}
maximum=${max}
   if [ $((minimum/divisiblePar*divisiblePar)) -ne ${minimum} ]; then
      if [ ${minimum} -lt 0 ]; then
         minimum=$((minimum/divisiblePar*divisiblePar))
      else
         minimum=$((((minimum/divisiblePar)+1)*divisiblePar))
      fi
   fi


   #  Si max est lui-même non divisible par $divisiblePar,
   #+ alors corrigez le max pour être à l'échelle.

   if [ $((maximum/divisiblePar*divisiblePar)) -ne ${maximum} ]; then
      if [ ${maximum} -lt 0 ]; then
         maximum=$((((maximum/divisiblePar)-1)*divisiblePar))
      else
         maximum=$((maximum/divisiblePar*divisiblePar))
      fi
   fi


#  Nous avons besoin de générer seulement les sous-scripts de tableaux positifs,
#+ donc nous avons besoin d'un déplacement qui nous garantie des résultats positifs.

deplacement=$((0-minimum))
for ((i=${minimum}; i<=${maximum}; i+=divisiblePar)); do
   reponse[i+deplacement]=0
done


# Maintenant, bouclons avec un gros nombre de fois pour voir ce que nous obtenons.
loopIt=1000   #  L'auteur du script suggère 100000,
              #+ mais cela prend beaucoup de temps.

for ((i=0; i<${loopIt}; ++i)); do

   #  Notez que nous spécifions min et max en ordre inverse ici pour s'assurer que les
   #+ fonctions sont correctes dans ce cas.

   aleatoireEntre ${max} ${min} ${divisiblePar}

   # Rapporte une erreur si une réponse est inattendue.
   [ ${aleatoireEntreAnswer} -lt ${min} -o ${aleatoireEntreAnswer} -gt ${max} ] \
     && echo MIN or MAX error - ${aleatoireEntreAnswer}!
   [ $((aleatoireEntreAnswer%${divisiblePar})) -ne 0 ] \
     && echo DIVISIBLE BY error - ${aleatoireEntreAnswer}!

   # Stocke la réponse statistiquement.
reponse[aleatoireEntreAnswer+deplacement]=$((reponse[aleatoireEntreAnswer+deplacement]+1))
done



# Vérifions les résultats.

for ((i=${minimum}; i<=${maximum}; i+=divisiblePar)); do
   [ ${reponse[i+deplacement]} -eq 0 ] \
    && echo "We never got an reponse of $i." \
    || echo "${i} occurred ${reponse[i+deplacement]} times."
done


exit 0
```

À quel point $RANDOM est-il aléatoire ? la meilleure façon de le tester est d'écrire un script qui enregistre la suite des nombres « aléatoires » générés par $RANDOM. Faisons tourner $RANDOM plusieurs fois...

Exemple 9.30. Lancement d'un seul dé avec RANDOM

```
#!/bin/bash
# À quel point RANDOM est aléatoire?

RANDOM=$$      #  Réinitialise le générateur de nombres aléatoires en utilisant
               #+ le PID du script.

PIPS=6         # Un dé a 6 faces.
COMPTEURMAX=600# Augmentez ceci si vous n'avez rien de mieux à faire.
compteur=0     # Compteur.

un=0           # Doit initialiser les comptes à zéro
deux=0         # car une variable non initialisée est nulle, et ne vaut pas zéro.
trois=0
quatre=0
cinq=0
six=0

Affiche_resultat ()
{
echo
echo "un =   $un"
echo "deux =   $deux"
echo "trois = $trois"
echo "quatre =  $quatre"
echo "cinq =  $cinq"
echo "six =  $six"
echo
}

mise_a_jour_compteur()
{
case "$1" in
  0) let "un += 1";;     # Comme le dé n'a pas de "zéro", ceci correspond à 1.
  1) let "deux += 1";;   # Et ceci à 2, etc.
  2) let "trois += 1";;
  3) let "quatre += 1";;
  4) let "cinq += 1";;
  5) let "six += 1";;
esac
}

echo


while [ "$compteur" -lt "$COMPTEURMAX" ]
do
  let "die1 = RANDOM % $PIPS"
  mise_a_jour_compteur $die1
  let "compteur += 1"
done

Affiche_resultat

exit 0

#  Les scores devraient être distribués de façon égale en supposant que RANDOM
#+ soit correctement aléatoire.
#  Avec $COMPTEURMAX à 600, tout devrait tourner autour de 100, plus ou moins
#+ 20.
#
# Gardez en tête que RANDOM est un générateur pseudo-aléatoire,
```

```
# et pas un particulièrement bon.

#  Le hasard est un sujet profond et complexe.
#  Des séquences "au hasard" suffisamment longues pourraient exhiber un
#+ comportement cahotique et un autre comportement non aléatoire.

# Exercice (facile):
# ------------------
# Réécrire ce script pour lancer une pièce 1000 fois.
# Les choix sont "PILE" ou "FACE".
```

Comme nous avons vu sur le dernier exemple, il est préférable de *réinitialiser* le générateur *RANDOM* à chaque fois qu'il est invoqué. Utiliser le même germe pour *RANDOM* ne fera que répéter la même série de nombres [8] (ceci reflète le comportement de la fonction *C random()*).

Exemple 9.31. Réinitialiser RANDOM

```
#!/bin/bash
# seeding-random.sh: Utiliser la variable RANDOM.

NBMAX=25       # Combien de nombres à générer.

nombres_aleatoires ()
{
compteur=0
while [ "$compteur" -lt "$NBMAX" ]
do
  nombre=$RANDOM
  echo -n "$nombre "
  let "compteur += 1"
done
}

echo; echo

RANDOM=1          #  Initialiser RANDOM met en place le générateur de nombres
                  #+ aléatoires.
nombres_aleatoires

echo; echo

RANDOM=1          # Même élément pour RANDOM...
nombres_aleatoires    # ...reproduit la même série de nombres.
                  #
                  #  Quand est-il utile de dupliquer une série de nombres
                  #+ "aléatoires" ?

echo; echo

RANDOM=2          # Nouvel essai, mais avec un 'germe' différent...
nombres_aleatoires    # donne une autre série...

echo; echo

# RANDOM=$$  initialise RANDOM à partir du PID du script.
#  Il est aussi possible d'initialiser RANDOM à partir des commandes 'time' et
#+ 'date'.

# Un peu plus d'amusement...
SEED=$(head -1 /dev/urandom | od -N 1 | awk '{ print $2 }')
#  Sortie pseudo-aléatoire récupérée de /dev/urandom (fichier périphérique
#+ pseudo-aléatoire),
#+ puis convertit la ligne en nombres (octal) affichables avec "od".
```

[8] La *graine* d'une série de nombres pseudo-aléatoires générés par un ordinateur peut être considérée comme un label d'identification. Par exemple, pensez à la série pseudo-aléatoire avec une graine de *23* comme la *série #23*.

Une propriété d'une série de nombres pseudo-aléatoires est la longueur du cycle avant qu'il ne commence à se répéter. Un bon générateur pseudo-aléatoire produira des séries avec de très longs cycles.

```
#+ Finalement "awk" récupère un seul nombre pour SEED.
RANDOM=$SEED
nombres_aleatoires

echo; echo

exit 0
```

Note

Le pseudo fichier périphérique `/dev/urandom` apporte une méthode pour générer des nombres pseudo-aléatoires bien plus « aléatoires » que la variable `$RANDOM`. **`dd if=/dev/urandom of=fichier_cible bs=1 count=XX`** crée un fichier de nombres pseudo-aléatoires bien distribués. Néanmoins, assigner ces nombres à une variable dans un script nécessite un petit travail supplémentaire, tel qu'un filtrage par l'intermédiaire de od (comme dans l'exemple ci-dessus, dans l'Exemple 15.14, « Générer des nombres aléatoires de dix chiffres » et dans Exemple A.38, « Tri d'insertion ») ou tel que l'utilisation de dd (voir l'Exemple 15.59, « Effacer les fichiers de façon sûre ») ou même d'envoyer via un tube dans md5sum (voir l'Exemple 33.14, « Un jeu de « courses de chevaux » »).

Il existe aussi d'autres moyens pour générer des nombres pseudo aléatoires dans un script. **Awk** propose une façon agréable de le faire.

Exemple 9.32. Nombres pseudo-aléatoires, en utilisant awk634

```
#!/bin/bash
# random2.sh: Renvoie un nombre pseudo-aléatoire compris entre 0 et 1.
# Utilise la fonction rand() d'awk.

SCRIPTAWK=' { srand(); print rand() } '
# Commande(s) / paramètres passés à awk
# Notez que srand() réinitialise le générateur de nombre aléatoire de awk.

echo -n "Nombre aléatoire entre 0 et 1 = "

echo | awk "$SCRIPTAWK"
# Que se passe-t-il si vous oubliez le 'echo' ?

exit 0


# Exercices :
# ----------

# 1) En utilisant une construction boucle, affichez 10 nombres aléatoires
#    différents.
#      (Astuce : vous devez réinitialiser la fonction "srand()" avec une
donnée
#      différente à chaque tour de la boucle. Qu'arrive-t'il si vous échouez
à le
#      faire ?)

# 2) En utilisant un multiplicateur entier comme facteur d'échelle, générez
des
#    nombres aléatoires compris entre 10 et 100.

# 3) De même que l'exercice #2, ci-dessus, mais en générant des nombres
#    aléatoires entiers cette fois.
```

La commande date tend elle-même à générer des séquences d'entiers pseudo-aléatoires206.

9.7. La construction en double parenthèse

De façon similaire à la commande let, la construction **((...))** permet une évaluation arithmétique. Dans sa forme la plus simple,

a=$((5 + 3)), exécutera le calcul **5 + 3**, soit **8**, et attribuera sa valeur à la variable **a**. Néanmoins, cette construction en double parenthèse est aussi un mécanisme permettant la manipulation de variables à la manière du C dans Bash, par exemple ((var++)).

Exemple 9.33. Manipulation, à la façon du C, de variables

```
#!/bin/bash
# Manipuler une variable, style C, en utilisant la construction ((...)).


echo

(( a = 23 ))  # Initialiser une valeur, style C, avec des espaces des deux
              # côtés du signe "=".
echo "a (valeur initiale) = $a"

(( a++ ))     # Post-incrémente 'a', style C.
echo "a (après a++) = $a"

(( a-- ))     # Post-décrémente 'a', style C.
echo "a (après a--) = $a"


(( ++a ))     # Pre-incrémente 'a', style C.
echo "a (après ++a) = $a"

(( --a ))     # Pre-décrémente 'a', style C.
echo "a (après --a) = $a"

echo

########################################################################
#  Notez que, comme en C, les opérateurs pré- et post-décrémentation
#+ ont des effets de bord légèrement différents.

n=1; let --n && echo "True" || echo "False"  # Faux
n=1; let n-- && echo "True" || echo "False"  # Vrai

#  Merci, Jeroen Domburg.
########################################################################

echo

(( t = a<45?7:11 ))   # opérateur à trois opérandes, style C.
#       ^  ^ ^
echo "If a < 45, then t = 7, else t = 11."
echo "t = $t "        # Oui!

echo


# ------------------
# Alerte Easter Egg!
# ------------------
#  Chet Ramey semble avoir laissé un ensemble de constructions C non
#+ documentées dans Bash (déjà adapté de ksh).
#  Dans les documents Bash, Ramey appelle ((...)) un shell arithmétique,
#+ mais cela va bien au-delà.
#  Désolé, Chet, le secret est maintenant découvert.

# Voir aussi les boucles "for" et "while" utilisant la construction ((...)).

# Elles fonctionnent seulement avec Bash, version 2.04 ou ultérieure.

exit 0
```

Voir aussi l'Exemple 10.12, « Une boucle *for* à la C » et Exemple 8.4, « Représentation des constantes numériques ».

Chapitre 10. Boucles et branchements

What needs this iteration, woman?

--Shakespeare, **Othello**

Les opérations sur des blocs de code sont la clé pour des scripts shell structurés, organisés. Les constructions de boucles et de branchement fournissent les outils pour accomplir ceci.

10.1. Boucles

Une *boucle* est un bloc de code qui répète [1] une liste de commandes aussi longtemps que la *condition de contrôle de la boucle* est vraie.

boucles for

for *arg* in *[liste]*

C'est la construction de boucle de base. Elle diffère de façon significative de sa contre-partie en *C*.

`for` *arg* in [*liste*]
do
commande(s)...
done

Note

À chaque passage dans la boucle, *arg* prend successivement la valeur de toutes les variables de la *liste*.

```
for arg in "$var1" "$var2" "$var3" ... "$varN"
# Lors du tour 1 de la boucle, arg = $var1
# Lors du tour 2 de la boucle, arg = $var2
# Lors du tour 3 de la boucle, arg = $var3
# ...
# Lors du tour N de la boucle, arg = $varN

#  Les arguments dans [liste] sont mis entre guillemets pour empêcher une
#+ possible séparation des mots.
```

L'argument *liste* peut contenir des caractères joker.

Si *do* est sur la même ligne que *for*, il est impératif d'avoir un point virgule après la liste.

`for` *arg* in [*liste*] ; do

Exemple 10.1. Des boucles *for* simples

```
#!/bin/bash
# Liste les planètes.

for planete in Mercure Vénus Terre Mars Jupiter Saturne Uranus Neptune Pluton
do
  echo $planete         # Chaque plannète sur une ligne séparée.
done

echo

for planete in "Mercure Vénus Terre Mars Jupiter Saturne Uranus Neptune Pluton"
   # Toutes les planètes sur la même ligne.
   # La 'liste' entière entourée par des guillemets crée une variable simple.
   # Pourquoi ? Espaces blancs dans la variable.
do
```

[1] *Itération* : exécution répétée d'une commande ou d'un groupe de commande -- habituellement mais pas toujours -- tant qu'une certaine condition reste vraie ou jusqu'à ce qu'une certaine condition soit rencontrée.

```
  echo $planete
done

exit 0
```

Chaque élément de la **[liste]** peut contenir de multiples paramètres. C'est utile pour travailler sur des paramètres en groupe. Dans de tels cas, utilisez la commande set (voir l'Exemple 14.16, « Utiliser *set* avec les paramètres de position ») pour forcer l'analyse de chaque élément de la **[liste]** et l'affectation de chaque composant aux paramètres positionnels.

Exemple 10.2. Boucle *for* avec deux paramètres dans chaque élément de la [liste]

```
#!/bin/bash
# Planètes revisitées.

# Associe le nom de chaque planète à sa distance du soleil.

for planete in "Mercure 36" "Vénus 67" "Terre 93"  "Mars 142" "Jupiter 483"
do
  set -- $planete  #  Analyse la variable "planete"
                   #+ et initialise les paramètres de position.
  #  Le "--" empêche de mauvaises surprises si $planete est nul
  #+ ou commence avec un tiret.

  #  Il peut être utile de sauvegarder les paramètres de position originaux
  #+ car ils seront écrasés.
  # Une façon de le faire est d'utiliser un tableau,
  #        parametres_originaux=("$@")

  echo "$1         $2.000.000 miles du soleil"
  #-------deux tabulations---concatènent les zéros dans le paramètre $2
done

# (Merci, S.C., pour les clarifications supplémentaires.)

exit 0
```

Une variable peut fournir la **[liste]** dans une boucle *for*.

Exemple 10.3. *Fileinfo* : opérer sur une liste de fichiers contenue dans une variable

```
#!/bin/bash
# fileinfo.sh

FICHIERS="/usr/sbin/accept
/usr/sbin/pwck
/usr/sbin/chroot
/usr/bin/fakefile
/sbin/badblocks
/sbin/ypbind"   # Liste de fichiers qui vous intéressent.
                # Envoyez-les dans un fichier quelconque, /usr/bin/fauxfichier.

echo

for fichier in $FICHIERS
do

  if [ ! -e "$fichier" ]       # Vérifie si le fichier existe.
  then
    echo "$fichier n'existe pas."; echo
    continue               # Au suivant.
  fi

  ls -l $fichier | awk '{ print $9 "         taille: " $5 }' # Affiche 2 champs.
  whatis `basename $fichier`   # Informations sur le fichier.
  #  Notez que la base de données whatis doit avoir été configurée
```

```
  #+ pour que ceci fonctionne.
  #  Pour cela, en tant que root, lancez /usr/bin/makewhatis.
  echo
done

exit 0
```

Si la **[liste]** dans une *boucle for* contient des caractères joker (* et ?) utilisés dans le remplacement des noms de fichier, alors l'expansion des noms de fichiers a lieu.

Exemple 10.4. Agir sur des fichiers à l'aide d'une boucle *for*

```
#!/bin/bash
#  list-glob.sh: Générer une [liste] dans une boucle for
#+ en utilisant le remplacement.

echo

for fichier in *
#              ^  Bash réalise une expansion de noms de fichiers
#+                sur les expressions que le "globbing" reconnaît.
do
  ls -l "$fichier"  # Liste tous les fichiers de $PWD (répertoire courant).
  #  Rappelez-vous que le caractère joker "*" correspond à chaque nom de fichier,
  #+ néanmoins, lors du remplacement, il ne récupère pas les fichier commençant
  #+ par un point.

  #  Si le modèle ne correspond à aucun fichier, il s'étend à lui-même.
  #  Pour empêcher ceci, utilisez l'option nullglob
  #+ (shopt -s nullglob).
  #  Merci, S.C.
done

echo; echo

for fichier in [jx]*
do
  rm -f $fichier    # Supprime seulement les fichiers commençant par un "j" ou
                    # un "x" dans $PWD.
  echo "Suppression du fichier \"$fichier\"".
done

echo

exit 0
```

Omettre la partie **in [liste]** d'une *boucle for* fait en sorte que la boucle opère sur $@, les paramètres de position. Une illustration particulièrement intelligente de ceci est l'Exemple A.16, « *primes*: Générer des nombres premiers en utilisant l'opérateur modulo ». Voir aussi Exemple 14.17, « Inverser les paramètres de position ».

Exemple 10.5. `in [liste]` manquant dans une boucle *for*

```
#!/bin/bash

# Appeler ce script à la fois avec et sans arguments, et voir ce que cela donne.

for a
do
 echo -n "$a "
done

#  La 'liste' est manquante, donc la boucle opère sur '$@'
#+ (la liste d'arguments sur la ligne de commande, incluant les espaces blancs).

echo
```

```
exit 0
```

Il est possible d'utiliser la substitution de commandes141 pour générer la **[liste]** d'une *boucle for*. Voir aussi l'Exemple 15.53, « Utiliser *seq* pour générer l'incrément d'une boucle », l'Exemple 10.10, « Afficher les *liens symboliques* dans un répertoire » et l'Exemple 15.47, « Conversion de base ».

Exemple 10.6. Générer la `[liste]` dans une boucle *for* avec la substitution de commandes

```
#!/bin/bash
# for-loopcmd.sh : Une boucle for avec une [liste]
# générée par une substitution de commande.

NOMBRES="9 7 3 8 37.53"

for nombre in `echo $NOMBRES`  # for nombre in 9 7 3 8 37.53
do
  echo -n "$nombre "
done

echo
exit 0
```

Voici un exemple un peu plus complexe de l'utilisation de la substitution de commandes pour créer la **[liste]**.

Exemple 10.7. Un remplaçant de *grep* pour les fichiers binaires

```
#!/bin/bash
#  bin-grep.sh: Trouve les chaînes de caractères correspondantes dans un fichier
#+ binaire.

# Un remplacement de "grep" pour les fichiers binaires.
# Similaire par son effet à "grep -a"

E_MAUVAISARGS=65
E_SANSFICHIER=66

if [ $# -ne 2 ]
then
  echo "Usage: `basename $0` chaine_recherché nomfichier"
  exit $E_MAUVAISARGS
fi

if [ ! -f "$2" ]
then
        echo "Le fichier \"$2\" n'existe pas."
  exit $E_SANSFICHIER
fi

IFS=$'\012'       # Suivant la suggestion de Anton Filippov.
                  # était auparavant :  IFS="\n"
for word in $( strings "$2" | grep "$1" )
#  La commande "strings" liste les chaînes de caractères dans les fichiers
#+ binaires.
# Sortie envoyée via un tube dans "grep", qui cherche la chaîne désirée.
do
  echo $word
done

#  Comme S.C. l'a indiqué, les lignes 23 à 31 ci-dessus pourraient être
#+ remplacées avec la chaîne
#    strings "$2" | grep "$1" | tr -s "$IFS" '[\n*]'


#  Essayez quelque chose comme "./bin-grep.sh mem /bin/ls"
```

```
#+ pour comprendre ce script.

exit 0
```

Un peu la même chose.

Exemple 10.8. Afficher tous les utilisateurs du système

```
#!/bin/bash
# userlist.sh

FICHIER_MOTS_DE_PASSE=/etc/passwd
n=1           # Nombre d'utilisateurs

for nom in $(awk 'BEGIN{FS=":"}{print $1}' < "$FICHIER_MOTS_DE_PASSE" )
# Champ séparateur = :  ^^^^^^
# Affiche le premier champ      ^^^^^^^^
# Obtient l'entrée à partir du fichier        ^^^^^^^^^^^^^^^^^^^^^
do
  echo "UTILISATEUR #$n = $nom"
  let "n += 1"
done


# UTILISATEUR #1 = root
# UTILISATEUR #2 = bin
# UTILISATEUR #3 = daemon
# ...
# UTILISATEUR #30 = bozo

exit 0

#  Exercice :
#  ---------
#  Comment se fait-il qu'un utilisateur (ou un script lancé par cet utilisateur)
#+ puisse lire /etc/passwd ?
#  N'est-ce pas un trou de sécurité ? Pourquoi ou pourquoi pas ?
```

Un dernier exemple d'une **`[liste]`** résultant d'une substitution de commande.

Exemple 10.9. Rechercher les auteurs de tous les binaires d'un répertoire

```
#!/bin/bash
# findstring.sh :
#  Cherche une chaîne de caractères particulière dans des binaires d'un
#+ répertoire particulier.

repertoire=/usr/bin/
chainef="Free Software Foundation"  # Voir quels fichiers viennent de la FSF.

for fichier in $( find $repertoire -type f -name '*' | sort )
do
  strings -f $fichier | grep "$chainef" | sed -e "s%$repertoire%%"
  #  Dans l'expression "sed", il est nécessaire de substituer le délimiteur
  #+ standard "/" parce que "/" se trouve être un caractère filtré. Ne pas le
  #+ faire provoque un message d'erreur (essayez).
done

exit 0

#  Exercice (facile):
#  ------------------
#  Convertir ce script pour prendre en paramètres de ligne de commande les
#+ variables $repertoire et $chainef.
```

La sortie d'une *boucle for* peut être envoyée via un tube à une ou plusieurs commandes.

Exemple 10.10. Afficher les *liens symboliques* dans un répertoire

```
#!/bin/bash
# symlinks.sh : Liste les liens symboliques d'un répertoire.


repertoire=${1-`pwd`}
#  Par défaut, le répertoire courant, si le répertoire n'est pas spécifié.
#  Équivalent au bloc de code ci-dessous.
# ---------------------------------------------------------------
# ARGS=1                # Attend un argument en ligne de commande.
#
# if [ $# -ne "$ARGS" ]  # Si sans argument...
# then
#   repertoire=`pwd`     # répertoire courant
# else
#   repertoire=$1
# fi
# ---------------------------------------------------------------

echo "Liens symboliques du répertoire \"$repertoire\""

for fichier in "$( find $repertoire -type l )"   # -type l = liens symboliques
do
  echo "$fichier"
done | sort        # Sinon la liste de fichiers n'est pas triée.
#  Une boucle n'est pas réellement nécessaire ici,
#+ car la sortie de la commande "find" est étendue en un seul mot.
#  Néanmoins, il est facile de comprendre et d'illustrer ceci.

#  Comme Dominik 'Aeneas' Schnitzer l'indique, ne pas mettre entre guillemets
#+ $( find $repertoire -type l )
#+ fera échouer le script sur les noms de fichier comprenant des espaces.
#  Même ceci ne prendra que le premier champ de chaque argument.

exit 0


# ------------------------------------------------------
# Jean Helou propose l'alternative suivante :

echo "Liens symboliques du répertoire \"$repertoire\""
# Sauvegarde du IFS actuel. On n'est jamais trop prudent.
OLDIFS=$IFS
IFS=:

for fichier in $(find $repertoire -type l -printf "%p$IFS")
do     #                                  ^^^^^^^^^^^^^^^^
        echo "$fichier"
done|sort

# Et James "Mike" Conley suggère la modification du code de Helou :

OLDIFS=$IFS
IFS='' # Un IFS vide singifie aucun séparateur de mots
for file in $( find $directory -type l )
do
  echo $file
  done | sort

#  Ceci fonctionne dans le cas "pathologique" d'un nom de répertoire contenant
#+ une virgule.
#  "Ceci corrige aussi le cas "pathologique" d'un nom de répertoire contenant
#+ une virgule (ou une espace dans un exemple précédent)."
```

Le `stdout` d'une boucle peut être redirigé vers un fichier, comme cette légère modification du précédent exemple le montre.

Exemple 10.11. Liens symboliques dans un répertoire, sauvés dans un fichier

```
#!/bin/bash
# symlinks.sh : Liste les liens symboliques dans un répertoire.

FICHIER_DE_SORTIE=liste.liens_symboliques  # fichier de sauvegarde

repertoire=${1-`pwd`}
#  Par défaut, le répertoire courant si aucun autre n'a été spécifié.


echo "liens symboliques dans le répertoire \"$repertoire\"" > "$FICHIER_DE_SORTIE"
echo "----------------------------------------------------" >> "$FICHIER_DE_SORTIE"

for fichier in "$( find $repertoire -type l )"    # -type l = liens symboliques
do
  echo "$fichier"
done | sort >> "$FICHIER_DE_SORTIE"    # stdout de la boucle
#              ^^^^^^^^^^^^^^^^^^      redirigé vers le fichier de sauvegarde.

exit 0
```

Il existe une autre syntaxe pour une *boucle for* ressemblant fortement à celle du C. Elle nécessite des parenthèses doubles117.

Exemple 10.12. Une boucle *for* à la C

```
#!/bin/bash
# Deux façons de compter jusqu'à 10.

echo

# Syntaxe standard.
for a in 1 2 3 4 5 6 7 8 9 10
do
  echo -n "$a "
done

echo; echo

# +=========================================+

# Maintenant, faisons de même en utilisant une syntaxe C.

LIMITE=10

for ((a=1; a <= LIMITE ; a++)) # Double parenthèses, et "LIMITE" sans "$".
do
  echo -n "$a "
done                           # Une construction empruntée à 'ksh93'.

echo; echo

# +=================================================================+

#  Utilisons l'opérateur "virgule" C pour incrémenter deux variables en même
#+ temps.

for ((a=1, b=1; a <= LIMITE ; a++, b++))  # La virgule chaîne les opérations.
do
  echo -n "$a-$b "
done

echo; echo
```

```
exit 0
```

Voir aussi l'Exemple 26.15, « Application complexe des tableaux *Exploration d'une étrange série mathématique* », l'Exemple 26.16, « Simuler un tableau à deux dimensions, puis son test » et l'Exemple A.6, « *collatz* : Séries de Collatz ».

Maintenant, une *boucle for* utilisée dans un contexte de la « vie quotidienne ».

Exemple 10.13. Utiliser *efax* en mode batch

```
#!/bin/bash
# Fax (doit avoir installé 'efax').

ARGUMENTS_ATTENDUS=2
E_MAUVAISARGS=65
MODEM_PORT="/dev/ttyS2" # Cela peut être différent sur votre machine.
                        # Port par défaut de la carte modem PCMCIA.

if [ $# -ne $ARGUMENTS_ATTENDUS ]
# Vérifie le bon nombre d'arguments en ligne de commande.
then
        echo "Usage: `basename $0` téléphone# fichier-texte"
   exit $E_MAUVAISARGS
fi


if [ ! -f "$2" ]
then
  echo "Le fichier $2 n'est pas un fichier texte"
  # Ce fichier n'est pas un fichier standard ou il n'existe pas.
  exit $E_MAUVAISARGS
fi


fax make $2             #  Crée des fichiers formatés pour le fax à partir de
                        #+ fichiers texte.

for fichier in $(ls $2.0*)  # Concatène les fichiers convertis.
                       # Utilise le caractère joker dans la liste des variables.
do
  fic="$fic $fichier"
done

efax -d "$MODEM_PORT" -t "T$1" $fic   # Fait le boulot.
# Essayez d'ajouter -o1 si la ligne ci-dessus échoue.


# Comme S.C. l'a indiqué, la boucle for peut être supprimée avec
#    efax -d /dev/ttyS2 -o1 -t "T$1" $2.0*
# mais ce n'est pas aussi instructif.

exit $?  # De plus, efax envoie des messages de diagnostique sur stdout.
```

while

Cette construction teste une condition au début de la boucle et continue à boucler tant que la condition est vraie (renvoie un 0 comme code de sortie44). Par opposition à une boucle for, une *boucle while* trouve son utilité dans des situations où le nombre de répétitions n'est pas connu dès le départ.

```
while[ condition ]
do
commande(s)...
done
```

La construction utilisant des crochets dans une *boucle while* n'est rien de plus que notre ancien ami, le test entre crochets utili-

sé dans un test *if/then*. En fait, une *boucle while* peut être légalement utilisé avec la construction à double chrochets50 (while [[condition]]) car elle est plus versatile.

Comme c'est le cas avec les *boucles for*119, placer le *do* sur la même ligne que le test de la condition nécessite un point virgule.

`while [ condition ]; do`

Note that the *test brackets* are *not* mandatory in a *while* loop. See, for example, the getopts construct.

Exemple 10.14. Simple boucle *while*

```
#!/bin/bash

var0=0
LIMITE=10

while [ "$var0" -lt "$LIMITE" ]
#      ^                    ^
# Espaces ajoutés car ce sont des tests entre crochets.
do
  echo -n "$var0 "        # -n supprime le retour chariot.
  #             ^          espace, pour séparer les numéros affichés.

  var0=`expr $var0 + 1`   # var0=$(($var0+1)) fonctionne aussi.
                          # var0=$((var0 + 1)) fonctionne aussi.
                          # let "var0 += 1"    fonctionne aussi.
done                      # D'autres méthodes fonctionnent aussi.

echo

exit 0
```

Exemple 10.15. Une autre boucle *while*

```
#!/bin/bash

echo
                              # Équivalent à
while [ "$var1" != "fin" ]    # while test "$var1" != "fin"
do
  echo "Variable d'entrée #1 (quitte avec fin) "
  read var1                   # pas de 'read $var1' (pourquoi?).
  echo "variable #1 = $var1"  # A besoin des guillemets à cause du "#"...
  # Si l'entrée est 'fin', l'affiche ici.
  # Ne teste pas la condition de fin avant de revenir en haut de la boucle.
  echo
done

exit 0
```

Une *boucle while* peut avoir de multiples conditions. Seule la condition finale détermine quand la boucle se termine. Malgré tout, ceci nécessite une syntaxe de boucle légèrement différente.

Exemple 10.16. Boucle *while* avec de multiples conditions

```
#!/bin/bash

var1=unset
precedent=$var1

while echo "Variable précédente = $precedent"
      echo
      precedent=$var1
```

```
      [ "$var1" != fin ] # Garde trace de ce que $var1 valait précédemment.
      #  Quatre conditions sur "while", mais seule la dernière contrôle la
      #+ boucle.
      # Le *dernier* code de sortie est celui qui compte.
do
  echo "Variable d'entrée #1 (quitte avec fin) "
  read var1
  echo "variable #1 = $var1"
done

# Essayez de comprendre comment cela fonctionne.
# Il y a un peu d'astuce.

exit 0
```

Comme pour une *boucle for*, une *boucle while* peut employer une syntaxe identique à C en utilisant la construction avec des parenthèses doubles (voir aussi l'Exemple 9.33, « Manipulation, à la façon du C, de variables »).

Exemple 10.17. Syntaxe à la C pour une boucle *while*

```
#!/bin/bash
# wh-loopc.sh : Compter jusqu'à 10 dans une boucle "while".

LIMITE=10
a=1

while [ "$a" -le $LIMITE ]
do
  echo -n "$a "
  let "a+=1"
done           # Pas de surprise jusqu'ici.

echo; echo

# +==========================================================+

# Maintenant, de nouveau mais avec une syntaxe C.

((a = 1))      # a=1
#  Les doubles parenthèses permettent l'utilisation des espaces pour initialiser
#+ une variable, comme en C.

while (( a <= LIMITE )) # Doubles parenthèses, et pas de "$" devant la variable.
do
  echo -n "$a "
  ((a += 1))   # let "a+=1"
  #  Oui, en effet.
  #  Les doubles parenthèses permettent d'incrémenter une variable avec une
  #+ syntaxe style C.
done

echo

# Les programmeurs C se sentent chez eux avec Bash.

exit 0
```

À l'intérieur des crochets de test, une *boucle while* peut appeler une fonction352.

```
t=0

condition ()
{
  ((t++))

  if [ $t -lt 5 ]
```

```
  then
    return 0  # true
  else
    return 1  # false
  fi
}

while condition
#     ^^^^^^^^^
#     Appel de fonction -- quatre itérations de boucle.
do
  echo "Toujours en cours : t = $t"
done

# Toujours en cours : t = 1
# Toujours en cours : t = 2
# Toujours en cours : t = 3
# Toujours en cours : t = 4
```

Identique à une construction if-test, une boucle *while* peut omettre les crochets de test.

```
while condition
do
   ...
done
```

En couplant la puissance de la commande read avec une *boucle while*, nous obtenons la construction while read, utile pour lire et analyser des fichiers.

```
cat $nomfichier |   # Fournit des informations à partir d'un fichier.
while read ligne   # Tant qu'il y a une nouvelle ligne à lire...
do
  ...
done
```

Note

Une *boucle while* peut avoir son `stdin` redirigé vers un fichier336 par un < à la fin.

Une *boucle while* peut avoir son entrée standard (`stdin`) fourni via un tube168.

until

Cette construction teste une condition au début de la boucle et continue à boucler tant que la condition est fausse (l'opposé de la *boucle while*).

`until[` *`condition-est-vraie`* `]`
do
`commande(s)`...
done

Notez qu'une *boucle until* teste la condition de fin au début de la boucle, contrairement aux constructions similaires dans certains langages de programmation.

Comme c'est le cas avec les *boucles for*, placez *do* sur la même ligne que le test de la condition nécessite un point virgule.

`until[` *`condition-est-vraie`* `]`; do

Exemple 10.18. Boucle *until*

```
#!/bin/bash

CONDITION_FINALE=fin
```

```
until [ "$var1" = "$CONDITION_FINALE" ]
# Condition du test ici, en haut de la boucle.
do
  echo "Variable d'entrée #1 "
  echo "($CONDITION_FINALE pour sortir)"
  read var1
  echo "variable #1 = $var1"
done

exit 0
```

Comment choisir entre une boucle *for*, une boucle *while* et une boucle *until* ? En **C**, vous utiliserez typiquement une boucle *for* quand le nombre d'itérations est déjà connu. Néanmoins, avec *Bash*, la situation est plus compliquée. La boucle *for* en Bash est moins structurée et plus flexible que son équivalent dans d'autres langages. Du coup, n'hésitez pas à utiliser le type de boucle qui vous permet de faire ce que vous souhaitez de la façon la plus simple.

10.2. Boucles imbriquées

Une *boucle imbriquée* est une boucle dans une boucle, une boucle à l'intérieur du corps d'une autre boucle. Ce qui se passe est que le premier tour de la boucle externe déclenche la boucle interne, qui s'exécute jusqu'au bout. Puis le deuxième tour de la boucle externe déclenche la boucle interne une nouvelle fois. Ceci se répète jusqu'à ce que la boucle externe termine. Bien sûr, un *break* à l'intérieur de la boucle interne ou externe peut interrompre ce processus.

Exemple 10.19. Boucles imbriquées

```
#!/bin/bash
# nested-loop.sh : Boucles "for" imbriquées.

externe=1          # Initialisation du compteur de la boucle externe.

# Début de la boucle externe.
for a in 1 2 3 4 5
do
  echo "Tour $externe dans la boucle externe."
  echo "------------------------------"
  interne=1        # Initialisation du compteur de la boucle interne.

  # ===============================================================
  # Début de la boucle interne.
  for b in 1 2 3 4 5
  do
    echo "Tour $interne dans la boucle interne."
    let "interne+=1"  # Incrémentation du compteur de la boucle interne.
  done
  # Fin de la boucle interne.
  # ===============================================================

  let "externe+=1"    # Incrémentation du compteur de la boucle externe.
  echo                # Espace entre chaque bloc en sortie de la boucle externe.
done
# Fin de la boucle externe.

exit 0
```

Voir l'Exemple 26.10, « Le tri bulle : *Bubble Sort* » pour une illustration de boucles while imbriquées, et l'Exemple 26.12, « Crible d'Ératosthene » pour voir une boucle while imbriquée dans une boucle until.

10.3. Contrôle de boucles

Commandes affectant le comportement des boucles

break, **continue**

Les commandes de contrôle de boucle **break** et **continue** [2] correspondent exactement à leur contre-partie dans d'autres langages de programmation. La commande **break** termine la boucle (en sort), alors que **continue** fait un saut à la prochaine *itération* (répétition) de la boucle, oubliant les commandes restantes dans ce cycle particulier de la boucle.

Exemple 10.20. Effets de *break* et continue dans une boucle

```
#!/bin/bash

LIMITE=19  # Limite haute.

echo
echo "Affiche les nombres de 1 à 20 (mais pas 3 et 11)."

a=0

while [ $a -le "$LIMITE" ]
do
 a=$(($a+1))

 if [ "$a" -eq 3 ] || [ "$a" -eq 11 ]  # Exclut 3 et 11.
 then
   continue  # Continue avec une nouvelle itération de la boucle.
 fi

 echo -n "$a " # Ceci ne s'exécutera pas pour 3 et 11.
done

# Exercice :
# Pourquoi la boucle affiche-t'elle jusqu'au 20 ?

echo; echo

echo "Affiche les nombres de 1 à 20, mais quelque chose se passe après 2."

##########################################################################

# Même boucle, mais en substituant 'continue' avec 'boucle'.

a=0

while [ "$a" -le "$LIMITE" ]
do
 a=$(($a+1))

 if [ "$a" -gt 2 ]
 then
   break  # Ne continue pas le reste de la boucle.
 fi

 echo -n "$a "
done

echo; echo; echo

exit 0
```

La commande **break** peut de façon optionnelle prendre un paramètre. Un simple **break** termine seulement la boucle interne où elle est incluse mais un **break N** sortira de *N* niveaux de boucle.

Exemple 10.21. Sortir de plusieurs niveaux de boucle

```
#!/bin/bash
# break-levels.sh: Sortir des boucles.
```

[2] Ce sont des commandes intégrées159 du shell, alors que les autres commandes de boucle, telles que while et case, sont des mots clés159.

```
# "break N" sort de N niveaux de boucles.

for boucleexterne in 1 2 3 4 5
do
  echo -n "Groupe $boucleexterne:   "

  # ----------------------------------------------------------
  for boucleinterne in 1 2 3 4 5
  do
    echo -n "$boucleinterne "

    if [ "$boucleinterne" -eq 3 ]
    then
      break  # Essayez   break 2   pour voir ce qui se passe.
             # (Sort des boucles internes et externes.)
    fi
  done
  # ----------------------------------------------------------

  echo
done

echo

exit 0
```

La commande **continue**, similaire à **break**, prend un paramètre de façon optionnelle. Un simple **continue** court-circuite l'itération courante et commence la prochaine itération de la boucle dans laquelle elle se trouve. Un **continue N** termine toutes les itérations à partir de son niveau de boucle et continue avec l'itération de la boucle N niveaux au-dessus.

Exemple 10.22. Continuer à un plus haut niveau de boucle

```
#!/bin/bash
# La commande "continue N" continue jusqu'au niveau de boucle N.

for exterieur in I II III IV V           # Boucle extérieure
do
  echo; echo -n "Groupe $exterieur : "

  # ----------------------------------------------------------
  for interieur in 1 2 3 4 5 6 7 8 9 10  # Boucle intérieure
  do

    if [ "$interieur" -eq 7 ]
    then
      continue 2  #  Continue la boucle au deuxième niveau, c'est-à-dire la
                  #+ boucle extérieure.
                  #  Remplacez la ligne ci-dessus avec un simple "continue"
                  #  pour voir le comportement normal de la boucle.
    fi

    echo -n "$interieur "  # 7 8 9 10 ne s'afficheront jamais.
  done
  # ----------------------------------------------------------

done

echo; echo

# Exercice :
# Parvenir à un emploi utile pour "continue N" dans un script.

exit 0
```

Exemple 10.23. Utiliser *continue N* dans une tâche courante

```
# Albert Reiner donne un exemple pour l'utilisation de "continue N" :
# ------------------------------------------------------------------

#  Supposez que j'ai un grand nombre de jobs Ã  exÃ©cuter, avec des donnÃ©es Ã
#+ traiter dans des fichiers dont le nom correspond Ã  un certain modÃ¨le
#+ et qui font tous partie d'un mÃªme rÃ©pertoire.
#+ Plusieurs machines accÃ¨dent Ã  ce rÃ©pertoire et je veux distribuer le
#+ travail entre ces diffÃ©rentes machines. Alors, j'exÃ©cute ce qui suit
#+ avec nohup sur toutes les machines :

while true
do
  for n in .iso.*
  do
    [ "$n" = ".iso.opts" ] && continue
    beta=${n#.iso.}
    [ -r .Iso.$beta ] && continue
    [ -r .lock.$beta ] && sleep 10 && continue
    lockfile -r0 .lock.$beta || continue
    echo -n "$beta: " `date`
    run-isotherm $beta
    date
    ls -alF .Iso.$beta
    [ -r .Iso.$beta ] && rm -f .lock.$beta
    continue 2
  done
  break
done

#  Les dÃ©tails, en particulier le sleep N, sont spÃ©cifiques Ã  mon
#+ application mais le modÃ¨le gÃ©nÃ©ral est :

while true
do
  for job in {modÃ¨le}
  do
    {job dÃ©jÃ  terminÃ© ou en cours d'exÃ©cution} && continue
    {indiquez que ce job est en cours d'exÃ©cution, exÃ©cutez le job, indiquez-le
comme terminÃ©}
    continue 2
  done
  break        # Ou quelque chose comme `sleep 600' pour Ã©viter la fin.
done

#  De cette faÃ§on, le script s'arrÃªtera seulement quand il n'y aura plus de jobs
#+ Ã  faire (en incluant les jobs qui ont Ã©tÃ© ajoutÃ©s Ã  l'exÃ©cution). Ã#
travers
#  l'utilisation de fichiers verrous appropriÃ©s, il peut Ãªtre exÃ©cutÃ© sur
#+ plusieurs machines en mÃªme temps sans duplication des calculs [qui ont
#+ demandÃ© quelques heures dans mon cas, donc je veux vraiment Ã©viter ceci].
#+ De plus, comme la recherche recommence toujours au dÃ©but, vous pouvez
coder des prioritÃ©s dans les noms des fichiers. Bien sÃ»r, vous pouvez le
#+ faire sans `continue 2' mais alors vous devrez vÃ©rifier rÃ©ellement si
#+ un job s'est terminÃ© (pour rechercher immÃ©diatement le prochain
#+ job) ou non (auquel cas nous arrÃªtons le programme ou l'endormissons
#+ pour un long moment le temps que vous cherchions un autre job)..
```

Attention

La construction **continue N** est difficile à comprendre et complexe à utiliser dans tous les contextes. Il est probablement raisonnable de l'éviter.

10.4. Tests et branchements

Les constructions **case** et **select** ne sont pas techniquement des boucles puisqu' elles n'exécutent pas un bloc de code de façon ité-

rative. Néanmoins, comme les boucles, elles orientent le flot d'exécution du programme suivant certaines conditions au début ou à la fin du bloc.

Contrôler le flot du programme dans un bloc de code

case (in) / esac

La construction **case** est l'équivalent shell de **switch** en C/C++. Elle permet le branchement vers un bloc parmi un certain nombre de blocs de code, suivant des tests de condition. Elle agit comme une espèce de raccourcis pour de multiples instructions if/then/else et est un outil approprié pour la création de menus.

```
case "$variable" in
"$condition1" )
commande…
;;
"$condition2" )
commande…
;;
esac
```

Note

- Protéger les variables n'est pas obligatoire car la séparation de mots n'est pas effective.
- Chaque ligne de test se termine avec une parenthèse droite).
- Chaque bloc de conditions termine avec un *double* points virgule ;;.
- Le bloc **case** entier se termine avec un **esac** (*case* épelé à l'envers).

Exemple 10.24. Utiliser *case*

```
#!/bin/bash
# Tester des suites de caractères.

echo; echo "Appuyez sur une touche, puis faites ENTER."
read Touche

case "$Touche" in
  [[:lower:]]   ) echo "Lettre minuscule";;
  [[:upper:]]   ) echo "Lettre majuscule";;
  [0-9]   ) echo "Nombre";;
  *       ) echo "Ponctuation, espace blanc ou autre";;
esac  #  Permet un ensemble de caractères dans des [crochets].
      #+ ou des ensembles POSIX dans des [[crochets doubles]].

#  Dans la première version de cet exemple,
#+ les tests des caractères minuscules/majuscules étaient
#+ [a-z] et [A-Z].
#  Ceci ne fonctionne plus avec certaines locales et/ou distributions Linux.
#  POSIX est plus portable.
#  Merci à Frank Wang de me l'avoir fait remarquer.

# Exercice :
# ---------
# Ce script accepte un simple appui sur une touche, puis se termine.
# Modifiez le script pour qu'il accepte une saisie répétée,
# rapportez chaque appui sur une touche, et terminez lors de l'appui sur "X".
# Astuce : mettre tout dans une boucle "while".

exit 0
```

Exemple 10.25. Créer des menus en utilisant *case*

```
#!/bin/bash

# Base de données d'adresse.

clear # Efface l'écran.

echo "          Liste de Contacts"
echo "          -----------------"
echo "Choisissez une des personnes suivantes:"
echo
echo "[E]vans, Roland"
echo "[J]ones, Mildred"
echo "[S]mith, Julie"
echo "[Z]ane, Morris"
echo

read personne

case "$person" in
# Notez que la variable est entre guillemets.

  "E" | "e" )
  # Accepte les entrées en majuscule ou minuscule.
  echo
  echo "Roland Evans"
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;
  # Notez le double point-virgule pour terminer chaque option.

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Ex-girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Ajoutez de l'info pour Smith & Zane plus tard.

         * )
   # Option par défaut.
   # Entrée vide (en appuyant uniquement sur la touche RETURN) vient ici aussi.
   echo
   echo "Pas encore dans la base de données."
  ;;

esac

echo

#  Exercice:
#  --------
#  Modifier le script pour qu'il accepte plusieurs saisies,
#+ au lieu de s'arrêter après avoir affiché une seule adresse.

exit 0
```

Une utilisation exceptionnellement intelligente de **case** concerne le test des paramètres de ligne de commande.

```
#! /bin/bash

case "$1" in
  "") echo "Usage: ${0##*/} <nomfichier>"; exit $E_PARAM;;
                      #  Pas de paramètres en lignes de commande
                      #+ ou premier paramètre vide.
# Notez que ${0##*/} est la substitution de paramètres ${var##modèle}.
                      # Le résultat net est $0.

  -*) NOMFICHIER=./$1;; #  Si le nom de fichier passé en premier argument ($1)
                      #+ commence avec un tiret,
                      #+ le remplacez par ./$1
                      #+ pour que les commandes suivants ne l'interprètent pas
                      #+ comme une option.

  * ) NOMFICHIER=$1;;   # Sinon, $1.
esac
```

Voici un exemple plus direct de gestion de paramètres en ligne de commande :

```
#! /bin/bash


while [ $# -gt 0 ]; do    # Jusqu'à la fin des paramètres...
  case "$1" in
    -d|--debug)
              # paramètre "-d" ou "--debug" ?
              DEBUG=1
              ;;
    -c|--conf)
              CONFFILE="$2"
              shift
              if [ ! -f $CONFFILE ]; then
                echo "Erreur : le fichier indiqué n'existe pas !"
                exit $E_FICHIERCONF     # Erreur pour un fichier inexistant.
              fi
              ;;
  esac
  shift       # Vérifiez le prochain ensemble de paramètres.
done

#  À partir du script "Log2Rot" de Stefano Falsetto,
#+ faisant partie de son paquetage "rottlog".
#  Utilisé avec sa permission.
```

Exemple 10.26. Utiliser la *substitution de commandes* pour générer la variable *case*

```
#!/bin/bash
#  case-cmd.sh
#+ Utilisation de la substitution de commandes pour générer une variable "case".

case $( arch ) in   # "arch" renvoie l'architecture de la machine.
                    # Équivalent à 'uname -m'...
  i386 ) echo "Machine 80386";;
  i486 ) echo "Machine 80486";;
  i586 ) echo "Machine Pentium";;
  i686 ) echo "Machine Pentium2+";;
  *    ) echo "Autre type de machine";;
esac

exit 0
```

Une construction **case** peut filtrer les chaînes sur des paramètres de remplacement.

Exemple 10.27. Simple correspondance de chaîne

```
#!/bin/bash
# match-string.sh: simple correspondance de chaînes de caractères

chaines_correspondent ()
{
  CORRESPOND=0
  CORRESPOND_PAS=90
  PARAMS=2     # La fonction requiert deux arguments.
  MAUVAIS_PARAMS=91

  [ $# -eq $PARAMS ] || return $MAUVAIS_PARAMS

  case "$1" in
  "$2") return $CORRESPOND;;
  *   ) return $CORRESPOND_PAS;;
  esac

}


a=un
b=deux
c=trois
d=deux


chaines_correspondent $a     # mauvais nombre de paramètres
echo $?             # 91

chaines_correspondent $a $b  # pas de correspondance
echo $?             # 90

chaines_correspondent $b $d  # correspondance
echo $?             # 0


exit 0
```

Exemple 10.28. Vérification d'une entrée alphabétique

```
#!/bin/bash
#  isalpha.sh: Utiliser une structure "case" pour filtrer une chaîne de
#+ caractères.

SUCCES=0
ECHEC=-1

est_alpha ()  # Teste si le *premier caractère* de la chaîne est alphabétique.
{
if [ -z "$1" ]               # Pas d'argument passé?
then
  return $ECHEC
fi

case "$1" in
  [a-zA-Z]*) return $SUCCES;;  # Commence avec une lettre?
  *        ) return $ECHEC;;
esac
}            # Comparer ceci avec la fonction "isalpha ()" en C.


est_alpha2 ()   # Teste si la *chaîne entière* est alphabétique.
{
  [ $# -eq 1 ] || return $ECHEC

  case $1 in
  *[!a-zA-Z]*|"") return $ECHEC;;
```

```
               *) return $SUCCES;;
  esac
}

est_numerique ()    # Teste si la *chaîne entière* est numérique.
{                   # En d'autres mots, teste si la variable est de type entier.
  [ $# -eq 1 ] || return $ECHEC

  case $1 in
  *[!0-9]*|"") return $ECHEC;;
            *) return $SUCCES;;
  esac
}



verif_var ()  # Interface à est_alpha ().
{
if est_alpha "$@"
then
  echo "\"$*\" commence avec un caractère alpha."
  if est_alpha2 "$@"
  then        # Aucune raison de tester si le premier caractère est non alpha.
    echo "\"$*\" contient seulement des caractères alpha."
  else
    echo "\"$*\" contient au moins un caractère non alpha."
  fi
else
  echo "\"$*\" commence avec un caractère non alpha."
              # Aussi "non alpha" si aucun argument n'est passé.
fi

echo

}

verif_numerique ()  # Interface à est_numerique ().
{
if est_numerique "$@"
then
  echo "\"$*\" contient seulement des chiffres [0 - 9]."
else
  echo "\"$*\" a au moins un caractère qui n'est pas un chiffre."
fi

echo

}

a=23skidoo
b=H3llo
c=-What?
d=What?
e=`echo $b`   # Substitution de commandes.
f=AbcDef
g=27234
h=27a34
i=27.34

verif_var $a
verif_var $b
verif_var $c
verif_var $d
verif_var $e
verif_var $f
verif_var     # Pas d'argument passé, donc qu'arrive-t'il?
#
verif_numerique $g
verif_numerique $h
```

```
verif_numerique $i


exit 0        # Script amélioré par S.C.

# Exercice:
# --------
#  Ecrire une fonction 'est_flottant ()' qui teste les nombres en virgules
#+ flottantes.
#  Astuce: La fonction duplique 'est_numerique ()',
#+ mais ajoute un test pour le point décimal nécessaire.
```

select

La construction **select**, adoptée du Korn Shell, est encore un autre outil pour construire les menus.

`select` *`variable`* [in *`liste`*]
do
`commande`...
break
done

Ceci demande à l'utilisateur d'entrer un des choix présentés dans la variable liste. Notez que **select** utilise l'invite `PS3` (`#?`) par défaut mais que ceci peut être changé.

Exemple 10.29. Créer des menus en utilisant *select*

```
#!/bin/bash

PS3='Choisissez votre légume favori : ' # Affiche l'invite.

echo

select legume in "haricot" "carotte" "patate" "ognion" "rutabaga"
do
  echo
  echo "Votre légume favori est $legume."
  echo
  break  #  Qu'arriverait-il s'il n'y avait pas de 'break' ici ?
         #+ fin.
done

exit 0
```

Si une **<replaceable>liste</replaceable> in** est omise, alors **select** utilise la liste des arguments en ligne de commandes (`$@`) passée au script ou à la fonction dans lequel la construction **select** est intégrée.

Comparez ceci avec le comportement de la construction

`for` *`variable`* [in *`liste`*]
avec **in <replaceable>liste</replaceable>** omis.

Exemple 10.30. Créer des menus en utilisant select dans une fonction

```
#!/bin/bash

PS3='Choisissez votre légume favori: '

echo

choix_entre()
{
select legume
# [in list] omise, donc 'select' utilise les arguments passés à la fonction.
do
```

```
  echo
  echo "Votre légume favori est $vegetable."
  echo
  break
done
}

choix_entre haricot riz carotte radis tomate épinard
#           $1      $2  $3      $4    $5     $6
#         passé à la fonction choix_entre()

exit 0
```

Voir aussi l'Exemple 34.3, « Simple application de base de données, utilisant les références de variables indirectes ».

Chapitre 11. Substitution de commandes

Une substitution de commande réassigne la sortie d'une commande [1] ou même de multiples commandes ; elle branche littéralement la sortie d'une commande sur un autre contexte. [2]

La forme classique de la substitution de commande utilise l'*apostrophe inverse* (`...`). Les commandes placées à l'intérieur de ces apostrophes inverses génèrent du texte en ligne de commande.

```
nom_du_script=`basename $0`
echo "Le nom de ce script est $nom_du_script."
```

La sortie des commandes peut être utilisée comme argument d'une autre commande, pour affecter une variable, voire pour génerer la liste des arguments dans une boucle for.

```
rm `cat nomfichier`   # <quote>nomfichier</quote> contient une liste de fichiers à
effacer.
#
#  S. C. fait remarquer qu'une erreur "arg list too long" (liste d'arguments
#+ trop longue) pourrait en résulter.
#  Mieux encore              xargs rm -- < nomfichier
#  ( -- couvre les cas dans lesquels  <quote>nomfichier</quote> commence par un
#+ <quote>-</quote> )

listing_fichierstexte=`ls *.txt`
#  Cette variable contient les noms de tous les fichiers *.txt
#+ du répertoire de travail actuel.
echo $listing_fichierstexte

listing_fichierstexte2=$(ls *.txt)   #  La forme alternative d'une substitution
                                     #+ de commande.
echo $listing_fichierstexte2
# Même résultat.

#  Un problème qui peut survenir lorsqu'on place une liste de fichiers dans
#+ une chaîne simple est qu'une nouvelle ligne peut s'y glisser.
#  Une méthode plus sûre pour assigner une liste de fichiers à un paramètre est
#+ d'utiliser un tableau.
#      shopt -s nullglob    #  S'il n'y a pas de correspondance, les noms de
#+                          #+ fichier sont transformés en chaîne vide.
#      listing_fichierstextes=( *.txt )
#
# Merci, S.C.
```

Note

La substitution de commandes appelle un sous-shell342.

Attention

Les substitutions de commandes peuvent provoquer des coupures de mots.

```
COMMANDE `echo a b`     # 2 arguments: a et b

COMMANDE "`echo a b`"   # 1 argument : "a b"

COMMANDE `echo`         # pas d'argument

COMMANDE "`echo`"       # un argument vide


# Merci, S.C.
```

[1] Dans le cadre des *substitutions de commande*, une **commande** peut être une commande système externe, une commande intégrée du shell159 voire même une fonction d'un script457.
[2] Sur le plan technique, la *substitution de commandes* extrait la sortie (`stdout`) d'une commande et l'affecte à une variable en utilisant l'opérateur =.

Même s'il n'y a pas coupure de mots, une substitution de commandes peut ôter les retours à la ligne finaux.

```
# cd "`pwd`"  # Ceci devrait toujours fonctionner.
# Néanmoins...

mkdir 'répertoire avec un retour à la ligne final
'

cd 'répertoire avec un retour à la ligne final
'

cd "`pwd`"  # Message d'erreur:
#  bash: cd: /tmp/fichier avec un retour à la ligne final : Pas de fichier
#+ ou répertoire

cd "$PWD"   # Fonctionne parfaitement.




ancien_parametrage_tty=$(stty -g)   # Sauve les anciens paramètres du
terminal.
echo "Appuyez sur une touche "
stty -icanon -echo           # Désactive le mode "canonique" du terminal.
                             # Désactive également l'écho *local* .
touche=$(dd bs=1 count=1 2> /dev/null)   #  Utilisation de dd pour obtenir
                                         #+ l'appui d'une touche.
stty "$ancien_parametrage_tty"      # Restaure les anciens paramètres.
echo "Vous avez appuyé sur ${#touche} touche."  # ${#variable} =  $variable
#
#  Appuyez sur toute autre touche que  RETURN, et la sortie devient  "Vous
avez
#+ appuyé sur 1 touche"
# Appuyez sur  RETURN, et c'est  "Vous avez appuyé sur 0 touche."
# Le retour à la ligne a été avalé par la substitution de commande.

Merci, S.C.
```

Attention

L'utilisation d'**echo** pour afficher la valeur d'une variable *non protégée* affectée à l'aide d'une substitution de commande retire les caractères de nouvelle ligne finaux de la sortie des commandes ainsi redirigées, ce qui peut créer des surprises désagréables.

```
listing_rep=`ls -l`
echo $listing_rep     # non protégée

# Dans l'attente de la liste bien ordonnée du contenu d'un répertoire.

# En fait, voici ce que l'on obtient:
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1 bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13 wi.sh

# Les retours à la ligne ont disparu.


echo "$listing_rep"   # protégée
# -rw-rw-r--    1 bozo       30 May 13 17:15 1.txt
# -rw-rw-r--    1 bozo       51 May 15 20:57 t2.sh
# -rwxr-xr-x    1 bozo      217 Mar  5 21:13 wi.sh
```

La substitution de commande permet même d'affecter à une variable le contenu d'un fichier, en utilisant soit une redirection soit la commande cat

```
variable1=`<fichier1`     # Affecte à "variable1" le contenu de "fichier1".
```

```
variable2=`cat fichier2`  # Affecte à "variable2" le contenu de "fichier2".
                          #  Néanmoins, ceci lance un nouveau processus,
                          #+ donc la ligne de code s'exécute plus lentement que
                          #+ la version ci-dessus.

# Note :
#  Les variables peuvent contenir des espaces,
#+ voire même (horreur), des caractères de contrôle.
```

```
#  Extraits des fichiers système, /etc/rc.d/rc.sysinit
#+ (sur une installation Red Hat Linux)


if [ -f /fsckoptions ]; then
        fsckoptions=`cat /fsckoptions`
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
             hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
#
if [ ! -n "`uname -r | grep -- "-"`" ]; then
       ktag="`cat /proc/version`"
...
fi
#
#
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=02"`
    kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E "^I.*Cls=03.*Prot=01"`
...
fi
```

Attention

Ne pas affecter le contenu d'un *gros* fichier texte à une variable à moins que vous n'ayez une bonne raison de le faire. Ne pas affecter le contenu d'un fichier *binaire* à une variable, même pour blaguer.

Exemple 11.1. Trucs de script stupides

```
#!/bin/bash
# stupid-script-tricks.sh : Ne tentez pas ça chez vous, les gars !
# D'après "Trucs de Scripts Stupides", Volume I.


variable_dangereuse=`cat /boot/vmlinuz`   # Le noyau Linux compressé en
personne.

echo "longueur de la chaîne  \$variable_dangereuse = ${#variable_dangereuse}"
# longueur de la chaîne  $variable_dangereuse = 794151
# (ne donne pas le même résultat que  'wc -c /boot/vmlinuz')

# echo "$variable_dangereuse"
# N'essayez pas de faire ça ! Cela figerait le script.


#  L'auteur de ce document n'a pas connaissance d'une utilité quelconque pour
#+ l'affectation à une variable du contenu d'un fichier binaire.

exit 0
```

> Notez qu'on ne provoque pas de *surcharge de tampon*. C'est un exemple où un langage interprété, tel que Bash, fournit plus de protection vis à vis des erreurs de programmation qu'un langage compilé.

Une substitution de commande permet d'affecter à une variable la sortie d'une boucle. L'idée pour y parvenir est de se servir de la sortie d'une commande echo placée à l'intérieur de la boucle.

Exemple 11.2. Générer le contenu d'une variable à partir d'une boucle

```
#!/bin/bash
# csubloop.sh: Initialiser une variable à la sortie d'une boucle.

variable1=`for i in 1 2 3 4 5
do
  echo -n "$i"                 #  La commande 'echo' est essentielle
done`                          #+ à la substitution de commande.

echo "variable1 = $variable1"  # variable1 = 12345


i=0
variable2=`while [ "$i" -lt 10 ]
do
  echo -n "$i"                 # A nouveau le nécessaire 'echo'.
  let "i += 1"                 # Incrémentation.
done`

echo "variable2 = $variable2"  # variable2 = 0123456789

#  Démontre qu'il est possible d'intégrer une boucle à l'intérieur de la
#+ déclaration d'une variable.

exit 0
```

La substitution de commande permet d'augmenter l'ensemble des outils disponibles en Bash. Il suffit simplement d'écrire un programme ou un script dont la sortie est `stdout` (comme il se doit pour tout bon outil UNIX) et d'affecter cette sortie à une variable.

```
#include <stdio.h>

/* Le  programme C   "Hello, world."*/

int main()
{
  printf( "Hello, world." );
  return (0);
}

bash$ gcc -o hello hello.c


#!/bin/bash
# hello.sh

salutation=`./hello`
echo $salutation

bash$ sh hello.sh
Hello, world.
```

Note

La syntaxe **$(...)** a remplacé les apostrophes inverses pour la substitution de commande.

```
sortie=$(sed -n /"$1"/p $fichier) # Tiré de l'exemple "grp.sh".

# Initialiser une variable avec le contenu d'un fichier texte.
Contenu_fichier1=$(cat $fichier1)
Contenu_fichier2=$(<$fichier2)        # Bash le permet aussi.
```

La forme **$(...)** de la substitution de commande traite les doubles antislash d'une façon différente que **`...`**.

```
bash$ echo `echo \\`


bash$ echo $(echo \\)
\
```

La forme **$(...)** de la substitution de commandes autorise l'imbrication. [3]

```
word_count=$( wc -w $(ls -l | awk '{print $9}') )
```

ou quelque chose d'un peu plus élaboré...

Exemple 11.3. Découvrir des anagrammes

```
#!/bin/bash
# agram2.sh
# Exemple de substitution de commandes imbriquées.

#  Utilise l'outil "anagram"
#+ qui fait partie du paquetage de liste de mots "yawl" de l'auteur.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz

E_SANSARGS=66
E_MAUVAISARG=67
LONGUEUR_MIN=7

if [ -z "$1" ]
then
  echo "Usage $0 LETTRES"
  exit $E_SANSARGS         # Le script a besoin d'un argument en ligne de
commande.
elif [ ${#1} -lt $LONGUEUR_MIN ]
then
  echo "L'argument doit avoir au moins $LONGUEUR_MIN lettres."
  exit $E_MAUVAISARG
fi



FILTRE='.......'         # Doit avoir au moins sept lettres.
#       1234567
Anagrammes=( $(echo $(anagram $1 | grep $FILTRE) ) )
#            $(     $( sous-commande imbriquée ) )
#          (         affectation de tableaux       )
# (1) substitution de commandes imbriquées

echo
echo "${#Anagrammes[*]}  anagrammes trouvés de sept lettres ou plus"
echo
```

[3] En fait, l'imbrication est aussi possible avec des guillemets inversés mais seulement en 'échappant' les guillemets inversés interne comme l'indique John Default.

```
nb_mots=` wc -w \`ls -l | awk '{print $9}'\` `
```

```
echo ${Anagrammes[0]}      # Premier anagramme.
echo ${Anagrammes[1]}      # Deuxième anagramme.
                          # Etc.

# echo "${Anagrammes[*]}"  # Pour lister tous les anagrammes sur une seule
ligne...

#  Regardez dans le chapitre "Tableaux"
#+ pour des informations sur ce qu'il se passe ici.

# Voir aussi le script agram.sh pour un exemple de recherche d'anagramme.

exit $?
```

Exemples de substitution de commandes dans des scripts shell :

1. Exemple 10.7, « Un remplaçant de *grep* pour les fichiers binaires »
2. Exemple 10.26, « Utiliser la *substitution de commandes* pour générer la variable *case* »
3. Exemple 9.31, « Réinitialiser RANDOM »
4. Exemple 15.3, « *Badname* élimine dans le répertoire courant les fichiers dont le nom contient des caractères incorrects et des espaces blancs. »
5. Exemple 15.22, « *lowercase* : Change tous les noms de fichier du répertoire courant en minuscule. »
6. Exemple 15.17, « Émuler *grep* dans un script »
7. Exemple 15.53, « Utiliser *seq* pour générer l'incrément d'une boucle »
8. Exemple 10.13, « Utiliser *efax* en mode batch »
9. Exemple 10.10, « Afficher les *liens symboliques* dans un répertoire »
10. Exemple 15.32, « Supprimer les commentaires des programmes C »
11. Exemple 19.8, « Boucle *for* redirigée »
12. Exemple A.17, « *tree*: Afficher l'arborescence d'un répertoire »
13. Exemple 27.2, « Trouver le processus associé à un PID »
14. Exemple 15.46, « Paiement mensuel sur une hypothèque »
15. Exemple 15.47, « Conversion de base »
16. Exemple 15.48, « Appeler *bc* en utilisant un *document en ligne* »

.

Chapitre 12. Expansion arithmétique

L'expansion arithmétique fournit un outil puissant pour réaliser des opérations arithmétiques (entières) dans des scripts. Traduire une chaîne en une expression numérique est relativement immédiat en utilisant des *apostrophes inverses*, des *double parenthèses* ou *let*.

Variations

Expansion arithmétique avec apostrophes inverses141 (souvent utilisée en conjonction avec expr)

```
z=`expr $z + 3`          # La commande 'expr' réalise l'expansion.
```

Expansion arithmétique avec double parenthèses, et utilisant let

L'utilisation des *apostrophes inverses* dans le cadre de l'expansion arithmétique s'est trouvée dépréciée en faveur des *parenthèses doubles* **((...))**, **$((...))** ou de la très commode construction **let**.

```
z=$(($z+3))
z=$((z+3))                                #  Correct aussi.
                                          #  À l'intérieur de parenthèses
                                          #  doubles, le déréférencement de
                                          #  paramètres est optionnel.

# $((EXPRESSION)) est une expansion arithmétique.
# À ne pas confondre avec une substitution de commande.


# Vous pouvez aussi utiliser des opérations à l'intérieur de parenthèses doubles
# sans affectation.

  n=0
  echo "n = $n"                           # n = 0

  (( n += 1 ))                            # Incrément.
# (( $n += 1 )) est incorrect!
  echo "n = $n"                           # n = 1


let z=z+3
let "z += 3"  #  En présence d'apostrophes doubles, les espaces sont permis
              #+ dans l'affectation des variables.
#  'let' réalise une évaluation arithmétique, plutôt qu'une expansion à
#+ proprement parler.
```

Exemples d'expansions arithmétiques dans des scripts:

1. Exemple 15.9, « Utiliser *expr* »
2. Exemple 10.14, « Simple boucle *while* »
3. Exemple 26.1, « Utilisation d'un tableau simple »
4. Exemple 26.10, « Le tri bulle : *Bubble Sort* »
5. Exemple A.17, « *tree*: Afficher l'arborescence d'un répertoire »

Chapitre 13. Récréation

Cet étrange petit divertissement donne au lecteur une chance de se détendre et peut-être de rire un peu.

Félicitations ami Linuxien ! Vous êtes en train de lire
quelque chose qui va vous apporter chance et fortune.
Il vous suffit juste d'envoyer une copie de ce document à
dix de vos amis. Avant de faire les dix copies, envoyez un
script Bash de 100 lignes à la première personne se trouvant
dans la liste à la fin de cette lettre. Ensuite, effacez leur
nom et ajoutez-le vôtre à la fin de cette lettre.

Ne brisez pas la chaîne! Faites les copies dans les 48 heures.
Wilfred P. de Brooklyn a omis d'envoyer ces dix copies et s'est
réveillé le lendemain matin pour découvrir que sa description
de fonction avait été changée en "programmeur COBOL". Howard L.
a envoyé ces dix copies et, dans le mois, il a récupéré assez de
matériel pour monter un cluster Beowulf de 100 noeuds dédiés
à jouer à *TuxRacer*. Amelia V. de Chicago s'est
moquée de cette lettre et a brisé la chaîne. Quelques temps
après son terminal a brûlé et elle passe maintenant ses journée à
écrire de la documentation pour MS Windows.

Ne brisez pas la chaîne! Envoyez vos dix copies aujourd'hui !

Courtesy 'NIX "fortune cookies", avec quelques modifications et beaucoup d'excuses.

Partie Part 4. Commandes

Maîtriser les commandes de votre machine Linux est un prérequis indispensable à l'écriture de scripts shell efficaces.
Cette section couvre les commandes suivantes :

- . (Voir aussi le source)
- ac
- adduser
- agetty
- agrep220
- ar
- arch
- at
- autoload
- awk634 (Voir aussi Utiliser **awk** pour les opérations mathématiques)
- badblocks
- banner
- basename
- batch
- bc
- bg
- bind
- bison228
- builtin
- bzgrep220
- bzip2
- cal
- caller
- cat
- cd

- chattr
- chfn
- chgrp
- chkconfig
- chmod
- chown
- chroot
- cksum
- clear
- clock
- cmp
- col
- colrm
- column
- comm
- command
- compress
- cp
- cpio
- cron
- crypt
- csplit
- cu
- cut
- date
- dc
- dd
- debugfs
- declare
- depmod
- df
- dialog462
- diff
- diff3
- diffstat238
- dig
- dirname
- dirs
- disown
- dmesg

- doexec
- dos2unix
- du
- dump
- dumpe2fs
- e2fsck
- echo
- egrep218
- enable
- enscript
- env
- eqn
- eval
- exec
- exit (Thème lié : code de sortie44)
- expand
- export
- expr
- factor
- false
- fdformat
- fdisk
- fg
- fgrep219
- file
- find
- finger
- flex228
- flock
- fmt
- fold
- free
- fsck
- ftp
- fuser
- getopt
- getopts
- gettext
- getty
- gnome-mount

- grep
- groff
- groupmod
- groups (Thème lié : la variable $GROUPS)
- gs
- gzip
- halt
- hash
- hdparm
- head
- help
- hexdump
- host
- hostid
- hostname (Thème lié : la variable $HOSTNAME)
- hwclock
- iconv
- id (Thème lié : la variable $UID)
- ifconfig
- info
- infocmp
- init
- insmod
- install
- ip
- ipcalc
- iwconfig
- jobs
- join
- jot
- kill
- killall
- last
- lastcomm
- lastlog
- ldd
- less
- let
- lex
- ln

- locate
- lockfile
- logger
- logname
- logout
- logrotate
- look
- losetup
- lp
- ls
- lsdev
- lsmod
- lsof
- lspci
- lsusb
- ltrace
- lynx
- lzcat
- lzma
- m4
- mail
- mailto
- make
- MAKEDEV
- man
- mcookie
- md5sum
- mesg
- mimencode
- mkbootdisk
- mkdir
- mke2fs
- mkfifo
- mkisofs
- mknod
- mkswap
- mktemp
- mmencode
- modinfo
- modprobe

- more
- mount
- msgfmt
- mv
- nc
- netconfig
- netstat
- newgrp
- nice
- nl
- nm
- nmap
- nohup
- nslookup
- objdump
- od
- passwd
- paste
- patch237 (Thème lié : diff)
- pathchk
- pgrep
- pidof
- ping
- pkill
- popd
- pr
- printenv
- printf
- procinfo
- ps
- pstree
- ptx
- pushd
- pwd (Thème lié : la variable $PWD)
- quota
- rcp
- rdev
- rdist
- read
- readelf

- readlink
- readonly
- reboot
- recode
- renice297
- reset
- restore
- rev
- rlogin
- rm
- rmdir
- rmmod
- route
- rpm
- rpm2cpio
- rsh
- rsync
- runlevel
- run-parts
- rx
- rz
- sar
- scp
- script
- sdiff
- sed631
- seq
- service
- set
- setquota
- setserial
- setterm
- sha1sum
- shar
- shopt
- shred
- shutdown
- size
- skill297
- sleep

- slocate
- snice297
- sort
- source
- sox
- split
- sq
- ssh
- stat
- strace
- strings
- strip
- stty
- su
- sudo
- sum
- suspend
- swapoff
- swapon
- sx
- sync
- sz
- tac
- tail
- tar
- tbl
- tcpdump
- tee
- telinit
- telnet
- Tex
- texexec
- time
- times
- tmpwatch
- top
- touch
- tput
- tr
- traceroute

- true
- tset
- tsort
- tty
- tune2fs
- type
- typeset
- ulimit
- umask
- umount
- uname
- unarc
- unarj
- uncompress
- unexpand
- uniq
- units
- unlzma
- unrar
- unset
- unsq
- unzip
- uptime
- usbmodules
- useradd
- userdel
- usermod
- users
- usleep
- uucp
- uudecode
- uuencode
- uux
- vacation
- vdir
- vmstat
- vrfy
- w
- wait
- wall

- watch
- wc
- wget
- whatis
- whereis
- which
- who
- whoami283
- whois
- write
- xargs
- yacc
- yes
- zcat232
- zdiff237
- zdump
- zegrep220
- zfgrep220
- zgrep220
- zip

Chapitre 14. Commandes internes et intégrées

Une commande *intégrée* est une **commande** contenue dans la boîte à outils de Bash, elle est donc littéralement *intégrée*. C'est soit pour des raisons de performance -- les commandes intégrées s'exécutent plus rapidement que les commandes externes, qui nécessitent habituellement de dupliquer le processus -- soit parce qu'une commande intégrée spécifique a besoin d'un accès direct aux variables internes du shell.

Quand une commande ou le shell lui-même crée un sous-processus pour réaliser une tâche, cela s'appelle un *fork*. Ce nouveau processus est le *fils*, et le processus qui l'a *exécuté* est le *père*. Pendant que le *processus fils* fait son travail, le *processus père* est toujours en cours d'exécution.

Notez que bien qu'un *processus père* obtient l'*identifiant de processus* du *processus fils* et peut, du coup, lui passer des arguments, *le contraire n'est pas vrai*. Ceci peut créer des problèmes subtils et difficiles à trouver.424

Exemple 14.1. Un script exécutant plusieurs instances de lui-même

```
#!/bin/bash
# spawn.sh


PIDS=$(pidof sh $0)  # Identifiants des différentes instances du processus de ce
script.
P_array=( $PIDS )    # Les place dans un tableau (pourquoi ?).
echo $PIDS           # Affiche les identifiants des processus parents et enfants.
let "instances = ${#P_array[*]} - 1"  # Compte les éléments, moins 1.
                                      # Pourquoi soustraire 1 ?
echo "$instances instance(s) de ce script en cours d'exécution."
echo "[Ctl-C pour quitter.]"; echo


sleep 1              # Attente.
sh $0                # Play it again, Sam.

exit 0               # Pas nécessaire ; le script n'arrivera jamais ici.
                     # Pourquoi pas ?

#  Après avoir quitté avec un Ctl-C,
#+ est-ce que toutes les instances du script meurent ?
#  Si oui, pourquoi ?

# Note :
# -----
# Faites attention à ne pas laisser ce script s'exécuter trop longtemps.
# Il finirait par consommer trop de ressources système.

#  Est-ce qu'un script exécutant plusieurs instances de lui-même est une bonne
technique de script ?
#  Pourquoi ou pourquoi pas ?
```

Généralement, une commande *intégrée* Bash ne lance pas de sous-processus lorsqu'elle s'exécute à partir d'un script. Une commande système externe ou un filtre dans un script *va* généralement exécuter un sous-processus.

Une commande intégrée peut être le synonyme d'une commande système du même nom mais Bash la réimplémente en interne. Par exemple, la commande Bash **echo** n'est pas la même que `/bin/echo` bien que leurs comportements soient pratiquement identiques.

```
#!/bin/bash

echo "Cette ligne utilise la commande intégrée \"echo\"."
/bin/echo "Cette ligne utilise la commande système /bin/echo."
```

Un *mot clé* est un mot, une expression ou un opérateur *réservé*. Les mots clés ont une signification particulière pour le shell et sont

en fait les blocs permettant la construction de la syntaxe du shell. Comme exemples, « for », « while », « do » et « ! » sont des mots clés. Identiques à une commande intégrée159, un mot clé est codé en dur dans Bash mais, contrairement à une *commande intégrée*, un mot clé n'est pas en lui-même une commande mais fait partie d'un ensemble plus large de commandes. [1]

I/O

echo

envoie (vers `stdout`) une expression ou une variable (voir l'Exemple 4.1, « Affectation de variable et substitution »).

```
echo Bonjour
echo $a
```

Un **echo** nécessite l'option `-e` pour afficher des séquences d'échappement. Voir l'Exemple 5.2, « Caractères d'échappement ».

Habituellement, chaque commande **echo** envoie un retour à la ligne, mais l'option `-n` désactive ce comportement.

Note

Un **echo** peut être utilisé pour envoyer des informations à un ensemble de commandes via un tube.

```
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
then
  echo "$VAR contient la sous-chaîne \"txt\""
fi
```

Note

Un **echo**, en combinaison avec une substitution de commande141 peut définir une variable.

a=`echo "HELLO" | tr A-Z a-z`

Voir aussi l'Exemple 15.22, « *lowercase* : Change tous les noms de fichier du répertoire courant en minuscule. », l'Exemple 15.3, « *Badname* élimine dans le répertoire courant les fichiers dont le nom contient des caractères incorrects et des espaces blancs. », l'Exemple 15.46, « Paiement mensuel sur une hypothèque » et l'Exemple 15.47, « Conversion de base ».

Sachez que **echo `commande`** supprime tous les retours chariot que la sortie de *commande* génère.

La variable $IFS (séparateur interne de champ) contient habituellement \n (retour chariot) comme un des éléments de ses espaces blancs. Du coup, Bash divise la sortie de *commande* suivant les retours chariot et les prend comme argument pour **echo**. Ensuite, **echo** affiche ces arguments séparés par des espaces.

```
bash$ ls -l /usr/share/apps/kjezz/sounds
-rw-r--r--    1 root     root         1407 Nov  7  2000 reflect.au
 -rw-r--r--    1 root     root          362 Nov  7  2000 seconds.au



bash$ echo `ls -l /usr/share/apps/kjezz/sounds`
total 40 -rw-r--r-- 1 root root 716 Nov 7 2000 reflect.au -rw-r--r-- 1 root root ...
```

Donc, comment pouvons-nous intégrer un retour chariot dans la chaîne de caractère d'un echo ?

```
# Intégrer un retour chariot ?
echo "Pourquoi cette chaîne \n ne s'affiche pas sur deux lignes ?"
# Pas de deuxième ligne.

# Essayons autre chose.
```

[1] Une exception à ceci est la commande time, listée dans la documentation Bash officielle en tant que mot clé.

```
echo

echo $"Une ligne de texte contenant
un retour chariot."
# S'affiche comme deux lignes distinctes (retour chariot intégré).
# Mais, le préfixe "$" des variables est-il réellement nécessaire ?

echo

echo "Cette chaîne se divise
en deux lignes."
# Non, le "$" n'est pas nécessaire.

echo
echo "---------------"
echo

echo -n $"Autre ligne de texte contenant
un retour chariot."
# S'affiche comme deux lignes distinctes (retour chariot intégré).
# Même l'option -n échoue à la suppression du retour chariot ici.

echo
echo
echo "---------------"
echo
echo

# Néanmoins, ce qui suit ne fonctionne pas comme attendu.
# Pourquoi pas ? Astuce : affectation d'une variable.
chaine1=$"Encore une autre ligne de texte contenant
un retour chariot (peut-être)."

echo $chaine1
# Encore une autre ligne de texte contenant un retour chariot (peut-être).
#                                           ^
# Le retour chariot est devenu une espace.

# Merci pour cette indication, Steve Parker.
```

Note

Cette commande est une commande intégrée au shell, et n'est pas identique à `/bin/echo`, bien que son comportement soit similaire.

```
bash$ type -a echo
echo is a shell builtin
 echo is /bin/echo
```

printf

La commande **printf**, un print formaté, est un **echo** amélioré. C'est une variante limitée de la fonction `printf()` en langage C, et sa syntaxe est quelque peu différente.

`printf` *`format-string`*... *`parametre`*...

Il s'agit de la version intégrée à Bash de la commande `/bin/printf` ou `/usr/bin/printf`. Voir la page de manuel pour **printf** (la commande système) pour un éclairage détaillé.

Attention

Les anciennes versions de Bash peuvent ne pas supporter **printf**.

Exemple 14.2. *printf* en action

```
#!/bin/bash
# printf demo

PI=3.14159265358979
ConstanteDecimale=31373
Message1="Greetings,"
Message2="Earthling."

echo

printf "Pi avec deux décimales = %1.2f" $PI
echo
printf "Pi avec neuf décimales = %1.9f" $PI  # Il arrondit même correctement.

printf "\n"                                  # Affiche un retour chariot.
                                             # Équivalent à 'echo'.

printf "Constante = \t%d\n" $ConstanteDecimale  # Insère une tabulation (\t).

printf "%s %s \n" $Message1 $Message2

echo

# ===========================================#
# Simulation de la fonction C, sprintf().
# Changer une variable avec une chaîne de caractères formatée.

echo

Pi12=$(printf "%1.12f" $PI)
echo "Pi avec 12 décimales = $Pi12"      # Erreur d'arrondi !

Msg=`printf "%s %s \n" $Message1 $Message2`
echo $Msg; echo $Msg

#  La fonction 'sprintf' est maintenant accessible en tant que module chargeable
#+ de Bash mais ce n'est pas portable.

exit 0
```

Formater les messages d'erreur est une application utile de **printf**

```
E_MAUVAISREP=65

var=repertoire_inexistant

error()
{
  printf "$@" >&2
  # Formate les paramètres de position passés et les envoie vers stderr.
  echo
  exit $E_MAUVAISREP
}

cd $var || error $"Ne peut aller dans %s." "$var"

# Merci, S.C.
```

read

« Lit » la valeur d'une variable à partir de `stdin`, c'est-à-dire récupère interactivement les entrées à partir du clavier. L'option `-a` permet à **read** de lire des variables tableau (voir l'Exemple 26.5, « Quelques propriétés spéciales des tableaux »).

Exemple 14.3. Affectation d'une variable, en utilisant *read*

```
#!/bin/bash
# "Lire" des variables.
```

```
echo -n "Entrez la valeur de la variable 'var1' : "
# L'option -n d'echo supprime le retour chariot.

read var1
#  Notez qu'il n'y a pas de '$' devant var1 car elle est en train d'être
#+ initialisée.

echo "var1 = $var1"


echo

# Une simple instruction 'read' peut initialiser plusieurs variables.
echo -n "Entrez les valeurs des variables 'var2' et 'var3' " \
   "(séparées par des espaces ou des tabulations): "
read var2 var3
echo "var2 = $var2      var3 = $var3"
#  Si vous entrez seulement une valeur,
#+ les autres variables resteront non initialisées (null).

exit 0
```

Un **read** sans variable associée assigne son entrée à la variable dédiée $REPLY.

Exemple 14.4. Qu'arrive-t'il quand *read* n'a pas de variable

```
#!/bin/bash
# read-novar.sh

echo

# ------------------------------ #
echo -n "Saisissez une valeur : "
read var
echo "\"var\" = "$var""
# Tout se passe comme convenu.
# ------------------------------ #

echo

# ---------------------------------------------------------------------- #
echo -n "Saisissez une nouvelle valeur : "
read           #  Aucune variable n'est donnée à 'read', donc...
               #+ La saisie par 'read' est affectée à la variable par défaut,
$REPLY.
var="$REPLY"
echo "\"var\" = "$var""
# Ceci est équivalent au premier bloc de code.
# ---------------------------------------------------------------------- #

echo
echo "========================="
echo


#  Cet exemple est similaire au script "reply.sh".
#  Néanmoins, celui-ci montre que $REPLY est disponible
#+ même après un 'read' dans une variable de la façon classique.

# ====================================================================== #

#  Dans certaines instances, vous pourriez souhaiter ignorer la première valeur lue.
#  Dans de tels cas, ignorez tout simplement la variable $REPLY.

{ # Bloc de code.
read            # Ligne 1, à ignorer
```

```
read ligne2     # Ligne 2, sauvegarder dans la variable.
  } <$0
echo "La ligne 2 de ce script est :"
echo "$ligne2"  #   # read-novar.sh
echo            #   #!/bin/bash  ligne ignorée.

# Voir aussi le script soundcard-on.sh.

exit 0
```

Habituellement, saisir un **** supprime le retour chariot lors de la saisie suite à un **read**. Avec l'option `-r`, un caractère **** saisi sera interprété littéralement.

Exemple 14.5. Lecture de plusieurs lignes par *read*

```
#!/bin/bash

echo

echo "Saisissez une chaîne de caractères terminée par un \\, puis appuyez sur
ENTER."
echo "Ensuite, saisissez une deuxième chaîne de caractères (sans \\ cette fois), " \
  "puis appuyez de nouveau sur ENTER."
read var1     # Le "\" supprime le retour chariot lors de la lecture de $var1.
              #     première ligne \
              #     deuxième ligne

echo "var1 = $var1"
#     var1 = première ligne deuxième ligne

#  Pour chaque ligne terminée par un "\",
#+ vous obtenez une invite sur la ligne suivante pour continuer votre entrée
#+ dans var1.

echo; echo

echo "Saisissez une autre chaîne de caractères terminée par un \\ , puis appuyez sur
ENTER."
read -r var2  # L'option -r fait que le "\" est lu littéralement.
              #     première ligne \

echo "var2 = $var2"
#     var2 = première ligne \

# La saisie de données se termine avec le premier ENTER.

echo

exit 0
```

La commande **read** a quelques options intéressantes permettant d'afficher une invite et même de lire des frappes clavier sans appuyer sur **ENTER**.

```
# Lit une touche sans avoir besoin d'ENTER.

read -s -n1 -p "Appuyez sur une touche " touche
echo; echo "La touche était "\"$touche\""."

# L'option -s permet de supprimer l'écho.
# L'option -n N signifie que seuls N caractères sont acceptés en entrée.
# L'option -p permet l'affichage d'une invite avant de lire l'entrée.

#  Utiliser ces options est assez complexe car elles nécessitent d'être saisies dans
le
#+ bon ordre.
```

L'option `-n` pour **read** permet aussi la détection des **flèches de direction** et certaines des autres touches inhabituelles.

Exemple 14.6. Détecter les flèches de direction

```
#!/bin/bash
# arrow-detect.sh : Détecte les flèches du clavier et quelques autres touches.
# Merci, Sandro Magi, pour m'avoir montré comment faire.

# --------------------------------------------
# Codes générés par l'appui sur les touches.
flechehaut='\[A'
flechebas='\[B'
flechedroite='\[C'
flechegauche='\[D'
insert='\[2'
delete='\[3'
# --------------------------------------------

SUCCES=0
AUTRE=65

echo -n "Appuyer sur une touche...  "
#  Il est possible qu'il faille appuyer aussi sur ENTER si une touche non gérée
#+ ici est utilisée.
read -n3 touche                      # Lit 3 caractères.

echo -n "$touche" | grep "$flechehaut"  # Vérifie si un code est détecté.
if [ "$?" -eq $SUCCES ]
then
  echo "Appui sur la touche flèche haut."
  exit $SUCCES
fi

echo -n "$touche" | grep "$flechebas"
if [ "$?" -eq $SUCCES ]
then
  echo "Appui sur la touche flèche bas."
  exit $SUCCES
fi

echo -n "$touche" | grep "$flechedroite"
if [ "$?" -eq $SUCCES ]
then
  echo "Appui sur la touche flèche droite."
  exit $SUCCES
fi

echo -n "$touche" | grep "$flechegauche"
if [ "$?" -eq $SUCCES ]
then
  echo "Appui sur la touche flèche gauche."
  exit $SUCCES
fi

echo -n "$touche" | grep "$insert"
if [ "$?" -eq $SUCCES ]
then
  echo "Appui sur la touche \"Insert\"."
  exit $SUCCES
fi

echo -n "$touche" | grep "$delete"
if [ "$?" -eq $SUCCES ]
then
  echo "Appui sur la touche \"Delete\"."
  exit $SUCCES
fi
```

```
echo "Autre touche."

exit $AUTRE

# ======================================== #

#  Mark Alexander vient avec une version
#+ simplifiée du script ci-dessus (Merci !).
#  Cela supprime le besoin du grep.

#!/bin/bash

  flechehaut=$'\x1b[A'
  flechebas=$'\x1b[B'
  flechegauche=$'\x1b[D'
  flechedroite=$'\x1b[C'

  read -s -n3 -p "Appuyez sur une flèche : " x

  case "$x" in
  $flechehaut)
     echo "Vous avez appuyé sur la flèche haute"
     ;;
  $flechebas)
     echo "Vous avez appuyé sur la flèche basse"
     ;;
  $flechegauche)
     echo "Vous avez appuyé sur la flèche gauche"
     ;;
  $flechedroite)
     echo "Vous avez appuyé sur la flèche droite"
     ;;
  esac

exit $?

# ======================================== #

# Antonio Macchi propose une alternative plus simple.

#!/bin/bash

while true
do
  read -sn1 a
  test "$a" == `echo -en "\e"` || continue
  read -sn1 a
  test "$a" == "[" || continue
  read -sn1 a
  case "$a" in
    A)  echo "haut";;
    B)  echo "bas";;
    C)  echo "droite";;
    D)  echo "gauche";;
  esac
done

# ======================================== #

#  Exercice :
#  ---------
#  1) Ajouter la détection des touches "Home", "End", "PgUp" et "PgDn".
```

L'option `-n` de **read** ne détectera pas la touche *Entrée* (saut de ligne).

L'option `-t` de **read** permet de limiter le temps de réponse (voir l'Exemple 9.4, « *read* avec délai »).

La commande **read** peut aussi « lire » l'entrée à partir d'un fichier redirigé vers `stdin`. Si le fichier contient plus d'une ligne,

seule la première ligne est affectée à la variable. Si **read** a plus d'un paramètre, alors chacune des variables se voit assignée une suite de mots séparés par des espaces blancs. Attention !

Exemple 14.7. Utiliser *read* avec la redirection de fichier

```
#!/bin/bash

read var1 < fichier-donnees
echo "var1 = $var1"
# var1 initialisée avec la première ligne du fichier d'entrées "fichier-donnees"

read var2 var3 < fichier-donnees
echo "var2 = $var2   var3 = $var3"
# Notez le comportement non intuitif de "read" ici.
# 1) Revient au début du fichier d'entrée.
# 2) Chaque variable est maintenant initialisée avec une chaîne correspondante,
#    séparée par des espaces blancs, plutôt qu'avec une ligne complète de texte.
# 3) La variable finale obtient le reste de la ligne.
# 4) S'il existe plus de variables à initialiser que de chaînes terminées par
#    une espace blanche sur la première ligne du fichier, alors les variables
#    supplémentaires restent vides.

echo "---------------------------------------------------"

# Comment résoudre le problème ci-dessus avec une boucle :
while read ligne
do
  echo "$ligne"
done <fichier-donnees
# Merci à Heiner Steven de nous l'avoir proposé.

echo "---------------------------------------------------"

#  Utilisez $IFS (variable comprenant le séparateur interne de fichier, soit
#+ Internal File Separator) pour diviser une ligne d'entrée pour "read", si vous
#+ ne voulez pas des espaces blancs par défaut.

echo "Liste de tous les utilisateurs:"
OIFS=$IFS; IFS=:       # /etc/passwd utilise ":" comme séparateur de champ.
while read nom motpasse uid gid nomcomplet ignore
do
  echo "$nom ($nomcomplet)"
done </etc/passwd   # Redirection d'entrées/sorties.
IFS=$OIFS              # Restaure l'$IFS original.
# Cette astuce vient aussi de Heiner Steven.



#  Initialiser la variable $IFS à l'intérieur même de la boucle élimine le
#+ besoin d'enregistrer l'$IFS originale dans une variable temporaire.
#  Merci à Dim Segebart de nous l'avoir indiqué.
echo "---------------------------------------------------"
echo "Liste de tous les utilisateurs:"

while IFS=: read nom motpasse uid gid nomcomplet ignore
do
  echo "$nom ($nomcomplet)"
done </etc/passwd   # Redirection d'entrées/sorties.

echo
echo "\$IFS vaut toujours $IFS"

exit 0
```

Note

Envoyer la sortie d'un tube vers une commande *read* en utilisant echo pour définir des variables échouera425.

Cependant, envoyer la sortie d'un cat à travers un tube *semble* fonctionner.

```
cat fichier1 fichier2 |
while read ligne
do
echo $ligne
done
```

Néanmoins, comme Björn Eriksson le montre :

Exemple 14.8. Problèmes lors de la lecture d'un tube

```
#!/bin/sh
# readpipe.sh
# Cet exemple est une contribution de Bjon Eriksson.

dernier="(null)"
cat $0 |
while read ligne
do
    echo "{$ligne}"
    dernier=$ligne
done
printf "\nTout est fait, dernier :$dernier\n"

exit 0  # Fin du code.
        # La sortie (partielle) du script suit.
        # Le 'echo' apporte les crochets supplémentaires.

################################################

./readpipe.sh

{#!/bin/sh}
{dernier="(null)"}
{cat $0 |}
{while read ligne}
{do}
{echo "{$ligne}"}
{dernier=$ligne}
{done}
{printf "\nTout est fait, dernier :$dernier\n"}


Tout est fait, dernier :(null)

La variable (dernier) est initialisée à l'intérieur du sous-shell
mais est non initialisée à l'extérieur.
```

Le script *gendiff*, habituellement trouvé dans `/usr/bin` sur un grand nombre de distributions Linux, envoie la sortie de find via un tube vers la construction *while read*.

```
find $1 \( -name "*$2" -o -name ".*$2" \) -print |
while read f; do
. . .
```

Astuce

Il est possible de *coller* le texte dans le champ en entrée d'un *read*. Voir Exemple A.39, « Un générateur de fichiers *pad* pour les auteurs de shareware ».

Système de fichiers

cd

La commande familière de changement de répertoire, **cd**, trouve son intérêt dans les scripts où l'exécution d'une commande requiert d'être dans un répertoire spécifique.

```
(cd /source/repertoire && tar cf - . ) | (cd /dest/repertoire && tar xpvf -)
```

[à partir de l'exemple précédemment cité22 d'Alan Cox]

L'option `-P` (physique) pour **cd** fait qu'il ignore les liens symboliques.

cd - affecte la variable $OLDPWD.

Attention

La commande **cd** ne fonctionne pas de la façon attendue si deux slashs se suivent.

```
bash$ cd //
bash$ pwd
//
```

La sortie devrait être `/`. Ceci est un problème à la fois à partir de la ligne de commande et dans un script.

pwd

Print Working Directory (NdT : Affiche le répertoire courant). Cela donne le répertoire courant de l'utilisateur (ou du script) (voir l'Exemple 14.9, « Modifier le répertoire courant »). L'effet est identique à la lecture de la variable intégrée $PWD.

pushd, **popd**, **dirs**

Cet ensemble de commandes est un mécanisme pour enregistrer les répertoires de travail, un moyen pour revenir en arrière ou aller en avant suivant les répertoires d'une manière ordonnée. Une pile LIFO est utilisée pour conserver la trace des noms de répertoires. Des options permettent diverses manipulations sur la pile de répertoires.

`pushd nom-rep` enregistre le chemin de *`nom-rep`* dans la pile de répertoires et change le répertoire courant par *`nom-rep`*

popd supprime (enlève du haut) le chemin du dernier répertoire et, en même temps, change de répertoire courant par celui qui vient d'être récupéré dans la pile.

dirs liste le contenu de la pile de répertoires (comparez ceci avec la variable $DIRSTACK). Une commande **pushd** ou **popd** satisfaite va automatiquement appeler **dirs**.

Les scripts requérant différents changements du répertoire courant sans coder en dur les changements de nom de répertoire peuvent faire un bon usage de ces commandes. Notez que la variable tableau implicite `$DIRSTACK`, accessible depuis un script, tient le contenu de la pile des répertoires.

Exemple 14.9. Modifier le répertoire courant

```
#!/bin/bash

rep1=/usr/local
rep2=/var/spool

pushd $rep1
# Fera un 'dirs' automatiquement (liste la pile des répertoires sur stdout).
echo "Maintenant dans le répertoire `pwd`." # Utilise les guillemets inverses
                                            # pour 'pwd'.

# Maintenant, faisons certaines choses dans le répertoire 'rep1'.
pushd $rep2
echo "Maintenant dans le répertoire `pwd`."

# Maintenant, faisons certaines choses dans le répertoire 'rep2'.
echo "L'entrée supérieure du tableau DIRSTACK est $DIRSTACK."
popd
echo "Maintenant revenu dans le répertoire `pwd`."
```

```
# Maintenant, faisons certaines choses de plus dans le répertoire 'rep1'.
popd
echo "Maintenant revenu dans le répertoire original `pwd`."

exit 0

# Que se passe-t'il si vous n'exécutez pas 'popd' puis quittez le script ?
# Dans quel répertoire vous trouverez-vous ? Pourquoi ?
```

Variables

let

La commande **let** réalise des opérations arithmétiques sur des variables. Dans la majorité des cas, il fonctionne comme une version simplifiée de expr.

Exemple 14.10. Laisser *let* faire un peu d'arithmétique.

```
#!/bin/bash

echo

let a=11            # Identique à 'a=11'
let a=a+5           # Équivalent à  let "a = a + 5"
                    # (double guillemets et espaces pour le rendre plus lisible)
echo "11 + 5 = $a"  # 16

let "a <<= 3"       # Équivalent à  let "a = a << 3"
echo "\"\$a\" (=16) décalé de 3 places = $a"
                    # 128

let "a /= 4"        # Équivalent à  let "a = a / 4"
echo "128 / 4 = $a" # 32

let "a -= 5"        # Équivalent à  let "a = a - 5"
echo "32 - 5 = $a"  # 27

let "a = a * 10"    # Équivalent à  let "a = a * 10"
echo "27 * 10 = $a" # 270

let "a %= 8"        # Équivalent à  let "a = a % 8"
echo "270 modulo 8 = $a  (270 / 8 = 33, reste $a)"
                    # 6

echo

exit 0
```

eval

`eval arg1 [arg2] ... [argN]`

Combine les arguments dans une expression ou liste d'expressions et les *évalue*. Toute variable contenue dans l'expression sera étendue. Le résultat se traduit en une commande. C'est utile pour de la génération de code à partir de la ligne de commande ou à l'intérieur d'un script.

```
bash$ processus=xterm
bash$ affiche_processus="eval ps ax | grep $processus"
bash$ $affiche_processus
1867 tty1     S      0:02 xterm
 2779 tty1     S      0:00 xterm
 2886 pts/1    S      0:00 grep xterm
```

Chaque appel à *eval* force à la ré-*évaluation* de ses arguments.

```
a='$b'
b='$c'
c=d

echo $a             # $b
                    # Premier niveau.
eval echo $a        # $c
                    # Second niveau.
eval eval echo $a   # d
                    # Troisième niveau.

# Merci, E. Choroba.
```

Exemple 14.11. Montrer l'effet d'*eval*

```
#!/bin/bash
# Exercising "eval" ...

y=`eval ls -l`  # Similaire à y=`ls -l`
echo $y         # mais les retours chariot sont supprimés parce que la variable
                # n'est pas entre guillemets.
echo
echo "$y"       # Les retours chariot sont préservés lorsque la variable se
                # trouve entre guillemets.

echo; echo

y=`eval df`     # Similaire à y=`df`
echo $y         # mais les retours chariot ont été supprimés.

#  Quand LF n'est pas préservé, il peut être plus simple d'analyser la sortie,
#+ en utilisant des outils comme "awk".

echo
echo "============================================================="
echo

# Maintenant, nous vous montrons quoi faire d'utile avec "eval"...
# (Merci E. Choroba!)

version=3.4     #  Pouvons-nous répartir la version en deux parties, la version
                #+ majeure et la version mineure en une seule commande ?
echo "version = $version"
eval majeur=${version/./;mineur=}     #  Remplace '.' dans la version par ';mineur='
                                      #  La substitution ramène '3; minor=4'
                                      #+ donc eval donne mineur=4, majeur=3
echo majeur: $major, mineur: $mineur  #  majeur: 3, mineur: 4
```

Exemple 14.12. *Afficher* les *paramètres en ligne de commande*

```
#!/bin/bash
# echo-params.sh

# Appeler ce script avec quelques paramètres en ligne de commande.
# Par exemple :
#     sh echo-params.sh premier deuxieme troisieme quatrieme cinquieme

params=$#              # Nombre de paramètres en ligne de commande.
param=1                # Commencer par le premier paramètre.

while [ "$param" -le "$params" ]
do
  echo -n "Paramètre "
```

```
  echo -n \$$param     #  Donne seulement le *nom* de la variable.
#         ^^^          #  $1, $2, $3, etc.
                       #  Pourquoi ?
                       #  \$ échappe le premier "$"
                       #+ donc il l'affiche littéralement,
                       #+ et $param déréférence "$param" . . .
                       #+ ... comme on s'y attendait.
  echo -n " = "
  eval echo \$$param   #  Donne la *valeur* de la variable.
# ^^^^      ^^^        #  Cet "eval" force l'*évaluation*
                       #+ de \$$
                       #+ comme une référence indirecte de variable.

(( param ++ ))         # Au suivant.
done

exit $?

# =================================================

$ sh echo-params.sh premier deuxieme troisieme quatrieme cinquieme
Paramètre $1 = premier
Paramètre $2 = deuxieme
Paramètre $3 = troisieme
Paramètre $4 = quatrieme
Paramètre $5 = cinquieme
```

Exemple 14.13. Forcer une déconnexion

```
#!/bin/bash
# Tuer ppp pour forcer une déconnexion

# Le script doit être exécuté en tant qu'utilisateur root.

killppp="eval kill -9 `ps ax | awk '/ppp/ { print $1 }'`"
#                     -------- ID du processus ppp -----

$killppp                 # Cette variable est maintenant une commande.


# Les opérations suivantes doivent être faites en tant qu'utilisateur root.

chmod 666 /dev/ttyS3     #  Restaure les droits de lecture/écriture, sinon que
                         #+ se passe-t'il ?
#  Comme nous lançons un signal SIGKILL à ppp après avoir changé les droits sur
#+ le port série, nous restaurons les droits à l'état initial.

rm /var/lock/LCK..ttyS3  #  Supprime le fichier de verrouillage du port série.
                         #+ Pourquoi ?

#  Note :
#  Suivant le matériel et même la version du noyau,
#+ le port du modem de votre machine pourrait être différent --
#+ /dev/ttyS1 ou /dev/ttyS2.

exit 0

# Exercices:
# ---------
# 1) Que le script vérifie si l'utilisateur root l'appelle.
# 2) Faire une vérification concernant le processus à tuer (qu'il existe bien).
# 3) Écrivez une autre version de ce script basé sur 'fuser' :
#+   if [ fuser -s /dev/modem ]; then ...
```

Exemple 14.14. Une version de *rot13*

```
#!/bin/bash
# Une version de "rot13" utilisant 'eval'.
# Comparez à l'exemple "rot13.sh".

setvar_rot_13()             # "rot13" scrambling
{
  local nomvar=$1 valeurvar=$2
  eval $nomvar='$(echo "$valeurvar" | tr a-z n-za-m)'
}


setvar_rot_13 var "foobar"   # Lancez "foobar" avec rot13.
echo $var                    # sbbone

setvar_rot_13 var "$var"     # Lance "sbbone" à travers rot13.
                             # Revenu à la variable originale.
echo $var                    # foobar

# Exemple de Stephane Chazelas.
# Modifié par l'auteur du document.

exit 0
```

Rory Winston a apporté sa contribution en donnant un autre exemple de l'utilité de la commande *eval*.

Exemple 14.15. Utiliser *eval* pour forcer une substitution de variable dans un script *Perl*

```
Dans le script Perl "test.pl" :
        ...
        my $WEBROOT = &lt;WEBROOT_PATH&gt;;
        ...

Pour forcer une substitution de variables, essayez :
        $export WEBROOT_PATH=/usr/local/webroot
        $sed 's/&lt;WEBROOT_PATH&gt;/$WEBROOT_PATH/' &lt; test.pl &gt; out

Mais ceci donne simplement :
        my $WEBROOT = $WEBROOT_PATH;

Néanmoins :
        $export WEBROOT_PATH=/usr/local/webroot
        $eval sed 's%\&lt;WEBROOT_PATH\&gt;%$WEBROOT_PATH%' &lt; test.pl &gt; out
#        ====

Ceci fonctionne bien et donne la substitution attendue :
        my $WEBROOT = /usr/local/webroot;


### Correction appliquée à l'exemple original de Paulo Marcel Coelho Aragao.
```

Attention

La commande **eval** est risquée et devrait normalement être évitée quand il existe une alternative raisonnable. Un **`eval $COMMANDES`** exécute le contenu de *`COMMANDES`*, qui pourrait contenir des surprises désagréables comme **rm -rf ***. Lancer **eval** sur un code inconnu écrit par des personnes inconnues vous fait prendre des risques importants.

set

La commande **set** modifie la valeur de variables/options internes au script. Une utilisation est de modifier les options419 qui déterminent le comportement du script. Une autre application est d'affecter aux paramètres de position du script le résultat d'une commande (**`set `commande``**). Le script peut alors séparer les différents champs de la sortie de la commande.

Exemple 14.16. Utiliser *set* avec les paramètres de position

```
#!/bin/bash

# script "set-test"

# Appeler ce script avec trois paramètres en ligne de commande,
# par exemple, "./set-test one two three".

echo
echo "Paramètres de position avant set \`uname -a\` :"
echo "Argument #1 = $1"
echo "Argument #2 = $2"
echo "Argument #3 = $3"


set `uname -a` # Configure les paramètres de position par rapport à la sortie
               # de la commande `uname -a`

echo $_        # inconnu
# Drapeaux initialisés dans le script.

echo "Paramètres de position après set \`uname -a\` :"
# $1, $2, $3, etc. reinitialisés suivant le résultat de `uname -a`
echo "Champ #1 de 'uname -a' = $1"
echo "Champ #2 de 'uname -a' = $2"
echo "Champ #3 de 'uname -a' = $3"
echo ---
echo $_        # ---
echo

exit 0
```

Plus de jeu avec les paramètres de position.

Exemple 14.17. Inverser les paramètres de position

```
#!/bin/bash
# revposparams.sh : Inverse les paramètres de position.
# Script de Dan Jacobson, avec quelques corrections de style par l'auteur du
document.


set a\ b c d\ e;
#     ^     ^     Espaces échappés
#       ^ ^       Espaces non échappés
OIFS=$IFS; IFS=:;
#             ^   Sauvegarde de l'ancien IFS et initialisation du nouveau.

echo

until [ $# -eq 0 ]
do          #     Passage des différents paramètres de position.
  echo "### k0 = "$k""     # Avant
  k=$1:$k;  #     Ajoute chaque paramètre de position à la variable de la boucle.
#     ^
  echo "### k = "$k""      # Après
  echo
  shift;
done

set $k  #  Initialise les nouveaux paramètres de position.
echo -
echo $# #  Nombre de paramètres de position.
echo -
echo
```

```
for i   #  Oublier la "liste in" initialise la variable -- i --
        #+ avec les paramètres de position.
do
  echo $i  # Affiche les nouveaux paramètres de position.
done

IFS=$OIFS  # Restaure IFS.

#  Question :
#  Est-il nécessaire d'initialiser un nouvel IFS pour que ce script fonctionne
#+ correctement ?
#  Que se passe-t'il dans le cas contraire ? Essayez.
#  Et pourquoi utiliser le nouvel IFS -- une virgule -- en ligne 17,
#+ pour l'ajout à la variable de la boucle ?
#  Quel est le but de tout ceci ?

exit 0

$ ./revposparams.sh

### k0 =
### k = a b

### k0 = a b
### k = c a b

### k0 = c a b
### k = d e c a b

-
3
-

d e
c
a b
```

Invoquer **set** sans aucune option ou argument liste simplement toutes les variables d'environnement ainsi que d'autres variables qui ont été initialisées.

```
bash$ set
AUTHORCOPY=/home/bozo/posts
 BASH=/bin/bash
 BASH_VERSION=$'2.05.8(1)-release'
 ...
 XAUTHORITY=/home/bozo/.Xauthority
 _=/etc/bashrc
 variable22=abc
 variable23=xzy
```

Utiliser **set** avec l'option `--` affecte explicitement le contenu d'une variable aux paramètres de position. Si aucune variable ne suit `--`, cela *déconfigure* les paramètres de positions.

Exemple 14.18. Réaffecter les paramètres de position

```
#!/bin/bash

variable="un deux trois quatre cinq"

set -- $variable
# Initialise les paramètres de position suivant le contenu de "$variable".

premier_param=$1
deuxieme_param=$2
```

```
shift; shift       # Shift fait passer les deux premiers paramètres de position.
# shift 2 fonctionne aussi
params_restant="$*"

echo
echo "premier paramètre = $premier_param"            # un
echo "deuxième paramètre = $deuxieme_param"          # deux
echo "paramètres restants = $params_restant"         # trois quatre cinq

echo; echo

# De nouveau.
set -- $variable
premier_param=$1
deuxieme_param=$2
echo "premier paramètre = $premier_param"            # un
echo "deuxième paramètre = $deuxieme_param"          # deux

# =======================================================

set --
# Désinitialise les paramètres de position si aucun variable n'est spécifiée.

premier_param=$1
deuxieme_param=$2
echo "premier paramètre = $premier_param"            # (valeur null)
echo "deuxième paramètre = $deuxieme_param"          # (valeur null)

exit 0
```

Voir aussi l'Exemple 10.2, « Boucle *for* avec deux paramètres dans chaque élément de la [liste] » et l'Exemple 15.55, « Utiliser *getopt* pour analyser les paramètres de la ligne de commande ».

unset

La commande **unset** supprime une variable shell en y affectant réellement la valeur *null*. Notez que cette commande n'affecte pas les paramètres de position.

```
bash$ unset PATH

bash$ echo $PATH




bash$
```

Exemple 14.19. « Déconfigurer » une variable

```
#!/bin/bash
# unset.sh: Dés-initialiser une variable.

variable=hello                      # Initialisée.
echo "variable = $variable"

unset variable                      # Dés-initialisée.
                                    # Même effet que : variable=
echo "(unset) variable = $variable"  # $variable est null.

if [ -z "$variable" ]               # Tente un test de longueur de chaîne.
then
  echo "\$variable a une taille nulle."
fi

exit 0
```

export

La commande **export** [2] rend disponibles des variables aux processus fils du script ou shell en cours d'exécution. Une utilisation importante de la commande **export** se trouve dans les fichiers de démarrage pour initialiser et rendre accessible les variables d'environnement aux processus utilisateur suivants.

Attention

Malheureusement, il n'existe pas de moyens pour exporter des variables dans le processus parent424, vers le processus appelant ou qui a exécuté le script ou le shell.

Exemple 14.20. Utiliser *export* pour passer une variable à un script *awk* embarqué

```
#!/bin/bash

# Encore une autre version du script "column totaler" (col-totaler.sh)
# qui ajoute une colonne spécifiée (de nombres) dans le fichier cible.
#  Il utilise l'environnement pour passer une variable de script à 'awk'...
#+ et place le script awk dans une variable.

ARGS=2
E_MAUVAISARGS=65

if [ $# -ne "$ARGS" ] # Vérifie le bon nombre d'arguments de la ligne de
                      # commande.
then
   echo "Usage: `basename $0` nomfichier numéro_colonne"
   exit $E_MAUVAISARGS
fi

nomfichier=$1
numero_colonne=$2

#===== Identique au script original, jusqu'à ce point =====#

export numero_colonne
#  Exporte le numéro de colonne dans l'environnement de façon à ce qu'il soit
#+ disponible plus tard.


# -----------------------------------------------
awkscript='{ total += $ENVIRON["numero_colonne"] }
END { print total }' $nomfichier
# Oui, une variable peut contenir un script awk.
# -----------------------------------------------

# Maintenant, exécute le script awk
awk "$awkscript" "$nomfichier"


# Merci, Stephane Chazelas.

exit 0
```

Astuce

Il est possible d'initialiser et d'exporter des variables lors de la même opération, en faisant **export var1=xxx**.

Néanmoins, comme l'a indiqué Greg Keraunen, dans certaines situations, ceci peut avoir un effet différent que d'initialiser une variable, puis de l'exporter.

[2] *Export*er des informations revient à les rendre disponibles dans un contexte plus général. Voir aussi la portée344.

```
bash$ export var=(a b); echo ${var[0]}
(a b)


bash$ var=(a b); export var; echo ${var[0]}
a
```

declare, **typeset**

Les commandes declare et typeset spécifient et/ou restreignent les propriétés des variables.

readonly

Identique à declare -r, configure une variable en lecture-seule ou, du coup, la transforme en constante. Essayer de modifier la variable échoue avec un message d'erreur. C'est l'équivalent shell du type **const** pour le langage *C*.

getopts

Ce puissant outil analyse les arguments en ligne de commande passés au script. C'est l'équivalent Bash de la commande externe getopt et de la fonction *getopt* familière aux programmeurs *C*. Il permet de passer et de concaténer de nombreuses options [3] et les arguments associés à un script (par exemple **`nomscript -abc -e /usr/local`**).

La construction **getopts** utilise deux variables implicites. `$OPTIND` est le pointeur de l'argument (*OPTion INDex*) et `$OPTARG` (*OPTion ARGument*) l'argument (optionnel) attaché à une option. Deux points suivant le nom de l'option lors de la déclaration marque cette option comme ayant un argument associé.

Une construction **getopts** vient habituellement dans une boucle while, qui analyse les options et les arguments un à un, puis incrémente la variable implicite `$OPTIND` pour passer à la suivante.

Note

1. Les arguments passés à la ligne de commande vers le script doivent être précédés par un tiret (`-`). Le préfixe `-` permet à **getopts** de reconnaitre les arguments en ligne de commande comme des *options*. En fait, **getopts** ne traitera pas les arguments sans les préfixes `-` et terminera l'analyse des options au premier argument rencontré qui ne les aura pas.
2. Le modèle **getopts** diffère légèrement de la boucle **while** standard dans le fait qu'il manque les crochets de condition.
3. La construction **getopts** remplace la commande getopt qui est obsolète.

```
while getopts ":abcde:fg" Option
# Déclaration initiale.
# a, b, c, d, e, f et g sont les options (indicateurs) attendues.
# Le : après l'option 'e' montre qu'il y aura un argument associé.
do
  case $Option in
    a ) # Fait quelque chose avec la variable 'a'.
    b ) # Fait quelque chose avec la variable 'b'.
    ...
    e)  # Fait quelque chose avec la variable 'e', et aussi avec $OPTARG,
        # qui est l'argument associé passé avec l'option 'e'.
    ...
    g ) # Fait quelque chose avec la variable 'g'.
  esac
done
shift $(($OPTIND - 1))
# Déplace le pointeur d'argument au suivant.

# Tout ceci n'est pas aussi compliqué qu'il n'y paraît <grin>.
```

[3] Une option est un argument agissant comme un indicateur, changeant les comportements du script de façon binaire. L'argument associé avec une option particulière indique le comportement que l'option active ou désactive.

Exemple 14.21. Utiliser *getopts* pour lire les options/arguments passés à un script

```
#!/bin/bash
#+ S'exercer avec getopts et OPTIND
#+ Script modifié le 10/09/03 suivant la suggestion de Bill Gradwohl.

#  Nous observons ici comment 'getopts' analyse les arguments en ligne de
#+ commande du script.
#  Les arguments sont analysés comme des "options" (flags) et leurs arguments
#+ associés.

# Essayez d'appeller ce script avec
# 'nomscript -mn'
# 'nomscript -oq qOption' (qOption peut être une chaîne de caractère arbitraire.)
# 'nomscript -qXXX -r'
#
# 'nomscript -qr'    - Résultat inattendu, prend "r" comme argument à l'option
#                      "q"
# 'nomscript -q -r'  - Résultat inattendu, identique à ci-dessus
# 'scriptname -mnop -mnop'  - Résultat inattendu
# (OPTIND est incapable d'indiquer d'où provient une option)

#  Si une option attend un argument ("flag:"), alors il récupèrera tout ce qui
#+ se trouve ensuite sur la ligne de commandes.

SANS_ARGS=0
E_ERREUROPTION=65

if [ $# -eq "$SANS_ARGS" ]  # Script appelé sans argument?
then
  echo "Usage: `basename $0` options (-mnopqrs)"
  exit $E_ERREUROPTION        # Sort et explique l'usage, si aucun argument(s)
                              # n'est donné.
fi
# Usage: nomscript -options
# Note: tiret (-) nécessaire


while getopts ":mnopq:rs" Option
do
  case $Option in
    m     ) echo "Scénario #1: option -m-   [OPTIND=${OPTIND}]";;
    n | o ) echo "Scénario #2: option -$Option-   [OPTIND=${OPTIND}]";;
    p     ) echo "Scénario #3: option -p-   [OPTIND=${OPTIND}]";;
    q     ) echo "Scénario #4: option -q- \
avec l'argument \"$OPTARG\"   [OPTIND=${OPTIND}]";;
    # Notez que l'option 'q' doit avoir un argument associé,
    # sinon il aura la valeur par défaut.
    r | s ) echo "Scénario #5: option -$Option-"'';;
    *     ) echo "Option non implémentée.";;   # DEFAULT
  esac
done

shift $(($OPTIND - 1))
# Décrémente le pointeur d'argument de façon à ce qu'il pointe vers le prochain.
#  $1 référence maintenant le premier élément n'étant pas une option sur la
#+ ligne de commande si un tel élément existe.

 exit 0

#   Comme Bill Gradwohl le dit,
#  "Le mécanisme getopts vous permet de spécifier :  nomscript -mnop -mnop
#+  mais il n'y a pas de moyen de différencier d'où cela vient en utilisant
#+  OPTIND."
```

Comportement des scripts

source, . (commande point)

Cette commande, lorsqu'elle est appelée à partir de la ligne de commande, exécute un script. À l'intérieur d'un script, un **`source nom-fichier`** charge le fichier `nom-fichier`. Exécuter le source d'un fichier (point de commandes) *importe* le code dans le script, s'ajoutant au script (même effet que la directive **`#include`** dans un programme *C*). Le résultat est le même que si les lignes « sourcées » de code étaient présentes physiquement dans le corps du script. Ceci est utile dans les situations où de multiples scripts utilisent un fichier de données communes ou une bibliothèque de fonctions.

Exemple 14.22. « Inclure » un fichier de données

```
#!/bin/bash

. fichier-donnees    # charge un fichier de données.
# Même effet que "source fichier-donnees", mais plus portable.

#  Le fichier "fichier-donnees" doit être présent dans le répertoire courant,
#+ car il est référencé par rappor à son 'basename'.

# Maintenant, référençons quelques données à partir de ce fichier.

echo "variable1 (de fichier-donnees) = $variable1"
echo "variable3 (de fichier-donnees) = $variable3"

let "sum = $variable2 + $variable4"
echo "Somme de variable2 + variable4 (de fichier-donnees) = $sum"
echo "message1 (de fichier-donnees) est \"$message1\""
# Note:                                 guillemets échappés

print_message Ceci est la fonction message-print de fichier-donnees.


exit 0
```

Le fichier `fichier-données` pour l'Exemple 14.22, « « Inclure » un fichier de données », ci-dessus, doit être présent dans le même répertoire.

```
# This is a data file loaded by a script.
# Files of this type may contain variables, functions, etc.
# It may be loaded with a 'source' or '.' command by a shell script.

# Let's initialize some variables.

variable1=22
variable2=474
variable3=5
variable4=97

message1="Hello, how are you?"
message2="Enough for now. Goodbye."

print_message ()
{
# Echoes any message passed to it.

  if [ -z "$1" ]
  then
    return 1
    # Error, if argument missing.
  fi

  echo

  until [ -z "$1" ]
  do
    # Step through arguments passed to function.
```

```
    echo -n "$1"
    # Echo args one at a time, suppressing line feeds.
    echo -n " "
    # Insert spaces between words.
    shift
    # Next one.
  done

  echo

  return 0
}
```

Si le fichier *inclus* est lui-même un script exécutable, alors il sera exécuté, puis renverra le contrôle au script qui l'a appelé. Un script exécutable *inclus* pourrait utiliser un return358 dans ce but.

Des arguments pourraient être passés (en option) au fichier *inclus* en tant que paramètres de position.

```
source $fichier $arg1 arg2
```

Il est même possible pour un script de s'intégrer (se *sourcer*) lui-même, bien qu'il ne semble pas que cela ait la moindre application pratique.

Exemple 14.23. Un script (inutile) qui se charge lui-même

```
#!/bin/bash
# self-source.sh : un script qui s'exécute lui-même "récursivement."
# De "Stupid Script Tricks", Volume II.

NBTOURSMAX=100    # Nombre maximal de tours d'exécution.

echo -n  "$nb_tour  "
#  Lors du premier tour, ceci va juste afficher deux espaces car $nb_tour n'est
#+ toujours pas initialisé.

let "nb_tour += 1"
#  Suppose que la variable non initialisée $nb_tour peut être incrémentée la
#+ première fois.
#  Ceci fonctionne avec Bash et pdksh mais cela repose sur un comportement
#+ non portable (et certainement dangereux).
#  Il serait mieux d'initialiser $nb_tour à 0 avant de l'incrémenter.

while [ "$nb_tour" -le $NBTOURSMAX ]
do
  . $0   # Le script "s'inclut" lui-même, plutôt que de s'appeler.
         # ./$0 (qui serait une vraie récursion) ne fonctionne pas ici.
         # Pourquoi ?
done

#  Ce qui arrive ici n'est pas réellement une récursion, car le script
#+ s'étend lui-même effectivement, c'est-à-dire que cela génère une nouvelle
#+ section de code, à chaque tour de la boucle 'while' lors du 'source' en ligne
#+ 20.
#
#  Bien sûr, le script interprète chaque nouvelle ligne incluse "#!" comme un
#+ commentaire, et non pas comme le début d'un nouveau script.

echo

exit 0   # L'effet très net est le comptage de 1 à 100.
         # Très impressionnant.

# Exercice :
# ---------
#  Écrire un script qui utilise cette astuce pour faire quelque chose de
#+ réellement utile.
```

exit

Termine un script sans condition. [4] La commande **exit** peut prendre de façon optionnelle un argument de type entier, qui est renvoyé au script en tant qu'état de sortie44 du script. C'est une bonne pratique de terminer tous les scripts, même les plus simples, avec un **`exit 0`**, indiquant un succès.

Note

Si un script se termine avec un **exit** sans argument, l'état de sortie est le statut de exit lors de son dernier lancement dans le script, sans compter le **exit**. C'est équivalent à un **exit $?**.

Note

Une commande **exit** peut aussi être utilisé pour terminer un sous-shell342.

exec

Cette commande shell intégrée remplace le processus courant avec une commande spécifiée. Normalement, lorsque le shell rencontre une commande, il lance un processus fils pour exécuter la commande. En utilisant la commande intégrée **exec**, le shell n'exécute aucun processus fils et la commande bénéficiant du **exec** remplace purement et simplement le shell. Lorsqu'elle est utilisée dans un script, cela force la sortie (exit) du script lorsque la commande bénéficiant du **exec** se termine. [5]

Exemple 14.24. Effets d'*exec*

```
#!/bin/bash

exec echo "Je sors \"$0\"."   # Sortie du script ici.

# ----------------------------------
# Les lignes suivantes ne s'exécutent jamais.

echo "Cet echo ne sera jamais exécuté."

exit 99                       #  Ce script ne sortira jamais par ici.
                              #  Vérifier le code de sortie après l'exécution du
                              #+ du script avec un 'echo $?'.
                              #  Cela ne sera *pas* 99.
```

Exemple 14.25. Un script lançant *exec* sur lui-même

```
#!/bin/bash
# self-exec.sh

echo

echo "Cette ligne apparaît UNE FOIS dans le script, cependant elle continue à
s'afficher."
echo "Le PID de cette instance du script est toujours $$."
#     Démontre qu'un sous-shell n'est pas un processus fils.

echo "=================== Tapez Ctl-C pour sortir ==================="

sleep 1

exec $0   #  Lance une autre instance du même script remplaçant le précédent.

echo "Cette ligne ne s'affichera jamais!"  # Pourquoi pas ?

exit 99   # Ne quittera pas ici.
          # Le code de retour ne sera pas 99.
```

[4] Techniquement, une commande **exit** termine seulement le processus ou le shell dans lequel il s'exécute, *pas* le *processus parent*.
[5] Sauf si **exec** est utilisé pour affecter de nouveau les descripteurs de fichiers333.

Un **exec** sert aussi à réaffecter les descripteurs de fichiers330. Par exemple, **`exec <fichier-zzz`** remplace `stdin` par le fichier `fichier-zzz`.

Note

L'option `-exec` de find n'est *`pas du tout`* la même chose que la commande shell intégrée **exec**.

shopt

Cette commande permet de changer les *options du shell* au vol (voir l'Exemple 24.1, « Alias à l'intérieur d'un script » et l'Exemple 24.2, « *unalias* : Configurer et supprimer un alias »). Elle apparaît souvent dans les fichiers de démarrage de Bash mais a aussi son utilité dans des scripts. Nécessite la version 2465, ou ultérieure, de Bash.

```
shopt -s cdspell
# Permet des petites erreurs dans le nom des répertoires avec 'cd'

cd /hpme  # Oups! J'ai mal tapé '/home'.
pwd       # /home
          # Le shell a corrigé la faute de frappe.
```

caller

Placer une commande **caller** dans une fonction352 affiche des informations sur `stdout` à propos de celui qui a *appelé* cette fonction.

```
#!/bin/bash

fonction1 ()
{
  # À l'intérieur de fonction1 ().
  caller 0   # Parle-moi de lui.
}

fonction1    # Ligne 9 du script.

# 9 main test.sh
# ^                 Numéro de ligne où a eu lieu l'appel de la fonction.
#   ^^^^            Appelé depuis la partie "main" du script.
#        ^^^^^^^    Nom du script appelant.

caller 0     # N'a aucun effet parce qu'il n'est pas à l'intérieur d'une fonction.
```

Une commande **caller** peut aussi renvoyer des informations de l'*appelant* sur un script inclus à l'intérieur d'un autre script. De façon analogue à une fonction, ceci est un « appel de sous-routine ».

Cette commande est utile pour le débogage.

Commandes

true

Une commande qui renvoie un succès (zéro) comme état de sortie44, mais ne fait rien d'autre.

```
bash$ true
bash$ echo $?
0
```

```
# Boucle sans fin
while true   # alias pour ":"
do
   operation-1
   operation-2
   ...
   operation-n
```

```
   # A besoin d'un moyen pour sortir de la boucle ou le script ne s'arrêtera pas.
done
```

false

Une commande qui renvoie un état de sortie44 correspondant à un échec, mais ne fait rien d'autre.

```
bash$ false
bash$ echo $?
1
```

```
# Tester "false"
if false
then
  echo "false évalué à \"true\""
else
  echo "false évalué à \"false\""
fi
# false s'évalue "false"


# Boucle while "false" (boucle nulle)
while false
do
   # Le code suivant ne sera pas exécuté.
   operation-1
   operation-2
   ...
   operation-n
   # Rien ne se passe!
done
```

type [cmd]

Identique à la commande externe which, **type cmd** identifie « cmd ». Contrairement à **which**, **type** est une commande intégrée à Bash. L'option `-a` est très utile pour que **type** identifie des *`mots clés`* et des *`commandes internes`*, et localise aussi les commandes système de nom identique.

```
bash$ type '['
[ is a shell builtin
bash$ type -a '['
[ is a shell builtin
 [ is /usr/bin/[


bash$ type type
type is a shell builtin
```

hash [cmds]

Enregistre le chemin des commandes spécifiées -- dans une table de hachage du shell [6] -- donc le shell ou le script n'aura pas besoin de chercher le `$PATH` sur les appels futurs à ces commandes. Quand **hash** est appelé sans arguments, il liste simplement les commandes qui ont été stockées. L'option `-r` réinitialise la table de hachage.

bind

La commande intégrée **bind** affiche ou modifie les correspondances de touche de *readline* [7] .

[6] Le *hachage* (ou découpage) est une méthode pour créer des clés de recherche pour des données stockées dans une table. Les *éléments de données eux-mêmes* sont « découpés » pour créer des clés en utilisant un des nombreux *algorithmes* (méthodes ou recettes) simples de mathématiques.

Un avantage du *hachage* est qu'il est rapide. Un inconvénient est que les « collisions » -- où une seule clé correspond à plus d'un élément de données -- sont possibles.

Pour des exemples de hachage, voir Exemple A.22, « Bibliothèque de fonctions de hachage » et Exemple A.23, « Coloriser du texte en utilisant les fonctions de hachage ».

[7] La bibliothèque *readline* est utilisée par Bash pour lire les entrées utilisateur dans un shell interactif.

help

Récupère un petit résumé sur l'utilisation d'une commande intégrée au shell. C'est l'équivalent de whatis pour les commandes intégrées.

```
bash$ help exit
exit: exit [N]
    Exit the shell with a status of N.  If N is omitted, the exit status
    is that of the last command executed.
```

14.1. Commandes de contrôle des jobs

Certaines des commandes de contrôle de jobs prennent en argument un *identifiant de job (job identifier)*. Voir la table à la fin de ce chapitre.

jobs

Liste les jobs exécutés en tâche de fond en indiquant le *numéro de job*. Pas aussi utile que ps.

Note

Il est trop facile de confondre les *jobs* et les *processus*. Certaines commandes intégrées159, telles que **kill**, **disown** et **wait** acceptent soit un numéro de job soit un numéro de processus comme argument. Les commandes fg, bg et **jobs** acceptent seulement un numéro de job.

```
bash$ sleep 100 &
[1] 1384

bash $ jobs
[1]+  Running                 sleep 100 &
```

« 1 » est le numéro de job (les jobs sont maintenus par le shell courant). « 1384 » est le PID ou *numéro de processus* (les processus sont maintenus par le système). Pour tuer ce job/processus, faites soit un **kill %1** soit un **kill 1384**.

Merci, S.C.

disown

Supprime le(s) job(s) de la table du shell des jobs actifs.

fg, **bg**

La commande **fg** fait basculer un job, qui tournait en tâche de fond, en avant-plan. La commande **bg** relance un job suspendu en tâche de fond. Si aucun numéro de job n'est spécifié, alors la commande **fg** ou **bg** agit sur le job en cours d'exécution.

wait

Suspend l'exécution du script jusqu'à ce que tous les jobs en tâche de fond aient terminé, ou jusqu'à ce que le numéro de job ou l'identifiant de processus spécifié en option se termine. Retourne l'état de sortie44 de la commande attendue.

Vous pouvez utiliser la commande **wait** pour empêcher un script de se terminer avant qu'un job en arrière-plan ne finisse son exécution (ceci créerait un processus orphelin).

Exemple 14.26. Attendre la fin d'un processus avant de continuer

```
#!/bin/bash

ROOT_UID=0   # Seulement les utilisateurs ayant $UID 0 ont les privilèges de
             # root.
E_NONROOT=65
E_SANSPARAM=66

if [ "$UID" -ne "$ROOT_UID" ]
then
```

```
  echo "Vous devez être root pour exécuter ce script."
  # "Passe ton chemin gamin, il est temps d'aller au lit."
  exit $E_NONROOT
fi

if [ -z "$1" ]
then
  echo "Usage: `basename $0` chaine-find"
  exit $E_SANSPARAM
fi


echo "Mise à jour de la base 'locate'..."
echo "Ceci peut prendre du temps."
updatedb /usr &     # Doit être lancé en tant que root.

wait
# Ne pas lancez le reste du script jusqu'à ce que 'updatedb' finisse.
# La base de données doit être mise à jour avant de chercher quelque chose.

locate $1

#  Sans la commande 'wait', avec le pire scénario, le script sortirait
#+ alors que 'updatedb' serait toujours en cours d'exécution,
#+ le laissant orphelin.

exit 0
```

Optionnellement, **wait** peut prendre un *identifiant de job* en tant qu'argument, par exemple, *wait%1* ou *wait $PPID*. Voir la table des identifiants de job.

Astuce

À l'intérieur d'un script, lancer une commande en arrière-plan avec un "et commercial" (&) peut faire que le script se bloque jusqu'à un appui sur la touche **ENTER**. Ceci semble arriver avec les commandes qui écrivent sur `stdout`. Cela peut être un gros problème.

```
#!/bin/bash
# test.sh

ls -l &
echo "Terminé."
```

```
bash$ ./test.sh
Terminé.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
 _
```

Comme l'explique Walter Brameld IV :

Ces scripts ne se bloquent pas. Il semble qu'ils le fassent car la commande en tâche de fond écrit le texte sur la console après l'invite. L'utilisateur a l'impression que l'invite n'a jamais été affichée. Voici la séquence des événements :

1. Le script lance la commande en tâche de fond.
2. Le script quitte.
3. Le shell affiche l'invite.
4. La commande en tâche de fond continue son exécution et l'écriture du texte sur la console.
5. La commande en tâche de fond se termine.

6. L'utilisateur ne voit pas une invite en bas de l'affichage, pensant du coup que le script est bloqué.

Placer un **wait** après la commande de tâche de fond semble remédier à ceci.

```
#!/bin/bash
# test.sh

ls -l &
echo "Terminé."
wait
```

```
bash$ ./test.sh
Terminé.
 [bozo@localhost test-scripts]$ total 1
 -rwxr-xr-x    1 bozo     bozo           34 Oct 11 15:09 test.sh
```

Rediriger la sortie de la commande dans un fichier ou même sur `/dev/null` permet aussi d'éviter ce problème.

suspend

Ceci a un effet similaire à **Controle**+**Z**, mais cela suspend le shell (le processus père du shell devrait le relancer à un moment approprié).

logout

Sort d'un login shell, quelque fois en spécifiant un état de sortie44.

times

Donne des statistiques sur le temps système passé lors de l'exécution des commandes de la façon suivante :

```
0m0.020s 0m0.020s
```

Cette fonctionnalité est d'une valeur relativement limitée car il est peu commun d'évaluer la rapidité des scripts shells.

kill

Force la fin d'un processus en lui envoyant le signal de *terminaison* approprié (voir l'Exemple 16.6, « *pidof* aide à la suppression d'un processus »).

Exemple 14.27. Un script qui se tue lui-même

```
#!/bin/bash
# self-destruct.sh

kill $$  # Le script tue son propre processus ici.
         # Rappelez-vous que "$$" est le PID du script.

echo "Cette ligne ne s'affichera pas."
# À la place, le shell envoie un message "Terminated" sur stdout.

exit 0   # Fin normale ? Non !

#  Après que le script se soit terminé prématurément,
#+ quel code de sortie retourne-t'il?
#
# sh self-destruct.sh
# echo $?
# 143
#
# 143 = 128 + 15
#             signal TERM
```

Note

`kill -l` liste tous les signaux (comme le fait le fichier `/usr/include/asm/signal.h`). Un **`kill -9`** est une *mort certaine*, qui terminera un processus qui refuse obstinément de mourir avec un simple **kill**. Quelque fois, un **`kill -15`** fonctionne. Un processus *zombie*, c'est-à-dire un processus qui a terminé mais dont le processus père n'a pas encore été tué, ne peut pas être tué par un utilisateur connecté -- vous ne pouvez pas tuer quelque chose qui est déjà mort -- mais **init** nettoiera habituellement cela plus ou moins tôt.

killall

La commande **killall** tue un processus en cours d'exécution suivant son *nom*, plutôt que son identifiant de processus. S'il existe plusieurs instances d'une même commande, *killall* les tuera *toutes*.

Note

Ceci fait référence à la commande **killall** de `/usr/bin`, *pas* au script killall310 dans `/etc/rc.d/init.d`.

command

La directive **command** désactive les alias et les fonctions pour la commande « COMMANDE » qui la suit immédiatement.

```
bash$ command ls
```

Note

C'est une des trois directives qui modifient le traitement de commandes de script. Les autres sont des commandes intégrées et activées.

builtin

Appeler **builtin COMMANDE_INTEGREE** lance la commande *`COMMANDE_INTEGREE`* en tant que commande intégrée159 du shell, désactivant temporairement à la fois les fonctions et les commandes externes du système disposant du même nom.

enable

Ceci active ou désactive une commande intégrée du shell. Comme exemple, *`enable -n kill`* désactive la commande intégrée kill, de façon à ce que, quand Bash rencontre *kill*, il appelle la commande externe `/bin/kill`.

L'option `-a` d'*enable* liste toutes les commandes intégrées du shell, indiquant si elles sont ou non activées. L'option `-f nom-fichier` permet à *enable* de charger une commande intégrée159 en tant que module de bibliothèque partagée (DLL) à partir d'un fichier objet correctment compilé. [8].

autoload

Ceci est une transposition à Bash du chargeur automatique de *ksh*. Avec *autoload* activé, une fonction avec une déclaration « autoload » se chargera depuis un fichier externe à sa première invocation. [9] Ceci sauve des ressources système.

Notez qu'*autoload* ne fait pas partie de l'installation de base de Bash. Il a besoin d'être chargé avec *`enable -f`* (voir ci-dessus).

Tableau 14.1. Identifiants de jobs

Notation	Signification
%N	Numéro de job [N]
%S	Appel (ligne de commande) de jobs commençant par la chaîne de caractères *S*
%?S	Appel (ligne de commande) de jobs contenant la chaîne de ca-

[8] Le source C pour un certain nombre de commandes intégrées chargeables est disponible typiquement dans le répertoire `/usr/share/doc/bash-?.??/functions`.

Notez que l'option `-f` d'**enable** n'est pas reconnue sur tous les systèmes.

[9] Le même effet qu'**autoload** peut être réalisé avec typeset -fu.

Notation	**Signification**
	ractères *S*
%%	Job « courant » (dernier job arrêté en avant-plan ou lancé en tâche de fond)
%+	Job « courant » (dernier job arrêté en avant-plan ou lancé en tâche de fond)
%-	Dernier job
$!	Dernier processus en tâche de fond

Chapitre 15. Filtres externes, programmes et commandes

Les commandes UNIX standards rendent les scripts shell plus polyvalents. La puissance des scripts provient du mélange de commandes systèmes et de directives shell avec des structures de programmation simples.

15.1. Commandes de base

Commandes incontournables pour le débutant

ls

La commande élémentaire de « listage » du contenu d'un répertoire. Il est très facile d'en sous-estimer la puissance. Par exemple, en utilisant `-R`, l'option de récursivité, **ls** affiche une structure de répertoire sous la forme d'une arborescence. D'autres options utiles sont `-S`, qui trie selon la taille du fichier, `-t`, qui trie selon la date de modification des fichiers, `-b`, qui affiche les caractères d'échappement, et `-i`, qui affiche les inodes des fichiers (voir l'Exemple 15.4, « Effacer un fichier par son numéro d'*inode* »).

Astuce

La commande *ls* renvoie un code de sortie44 non nul lors d'une tentation d'obtention d'une fichier inexistant.

```
bash$ ls abc
ls: abc: No such file or directory


bash$ echo $?
2
```

Exemple 15.1. Utilisation de *ls* pour créer une liste de fichiers à graver sur un CDR

```
#!/bin/bash
# ex40.sh (burn-cd.sh)
# Script d'automatisation de gravure de CD.


VITESSE=2          # Peut être plus élevée si votre graveur en est capable.
FICHIER_IMAGE=cdimage.iso
FICHIER_CONTENU=contenu
PERIPHERIQUE=cdrom
#PERIPHERIQUE="0,0" pour les anciennes versions de cdrecord
REPERTOIRE_PAR_DEFAUT=/opt  # C'est le répertoire contenant les fichiers à graver.
                            # Assurez-vous qu'il existe bien.
                            # Exercice : ajoutez cette vérification.

# Utilise le package "cdrecord" de Joerg Schilling.
# http://www.fokus.fhg.de/usr/schilling/cdrecord.html

#  Si ce script est exécuté par un utilisateur normal, alors il pourrait être
#+ nécessaire de lancer suid sur cdrecord
#+ (chmod u+s /usr/bin/cdrecord, en tant que root).
#+ Bien sûr, ceci crée une petite faille de sécurité.

if [ -z "$1" ]
then
  REPERTOIRE_IMAGE=$REPERTOIRE_PAR_DEFAUT
  # Le répertoire par défaut, si non défini sur la ligne de commande.
else
  REPERTOIRE_IMAGE=$1
fi

# Créer un fichier "sommaire".
ls -lRF $REPERTOIRE_IMAGE > $REPERTOIRE_IMAGE/$FICHIER_CONTENU
```

```
# L'option "l" donne une "longue" liste de fichier.
# L'option "R" rend la liste récursive.
# L'option "F" marque le type des fichiers (les répertoires se voient ajouter un /
#+ final).
echo "Sommaire en cours de création."

# Créer un fichier image avant de le graver sur CD.
mkisofs -r -o $FICHIER_IMAGE $REPERTOIRE_IMAGE
echo "Image ISO9660 ($FICHIER_IMAGE) en cours de création."

# Grave le CD.
echo "Gravure du CD."
echo "Veuillez patientez."
cdrecord -v -isosize speed=$VITESSE dev=$PERIPHERIQUE $FICHIER_IMAGE

exit $?
```

cat, **tac**

cat, un acronyme de *concatenate* (NdT : concaténer en français), affiche le contenu d'un fichier sur `stdout`. Lorsqu'il est combiné avec une redirection (> ou >>), il est couramment utilisé pour concaténer des fichiers.

```
# Utilisation de cat
cat nom_fichier # Liste le fichier.
cat fichier.1 fichier.2 fichier.3 &gt; fichier.123 # Combine les trois fichiers en
un seul.
```

L'option `-n` de **cat** insère, avant chaque début de ligne, un numéro de ligne dans le(s) fichier(s) cible(s). L'option `-b` sert à numéroter uniquement les lignes qui ne sont pas blanches. L'option `-v` montre les caractères non imprimables en utilisant la notation ^. L'option `-s` n'affiche qu'une seule ligne blanche lors de multiples lignes blanches consécutives.

Voir aussi l'Exemple 15.28, « *nl* : un script d'autonumérotation. » et l'Exemple 15.24, « *rot13* : rot13, cryptage ultra-faible. ».

Note

Dans un tube, il pourrait être plus efficace de rediriger l'entrée standard (`stdin`) dans un fichier plutôt que d'utiliser la commande **cat** avec un fichier.

```
cat nom_fichier | tr a-z A-Z

tr a-z A-Z < nom_fichier #  Même effet mais lance un processus de moins
                         #+ et dispense aussi du tube.
```

tac, le contraire de *cat*, affiche le contenu d'un fichier en commençant par sa fin.

rev

Inverse chaque ligne d'un fichier et affiche le résultat vers `stdout`. Le résultat est différent d'une utilisation de **tac**, dans le sens où **rev** conserve l'ordre des lignes mais traite chacune d'elle de sa fin vers son début (image du miroir).

```
bash$ cat fichier1.txt
Coucou, je suis la ligne 1.
Coucou, je suis la ligne 2.


bash$ tac fichier1.txt
Coucou, je suis la ligne 2.
Coucou, je suis la ligne 1.


bash$ rev fichier1.txt
.1 engil al sius ej ,uocuoC
.2 engil al sius ej ,uocuoC
```

cp

Il s'agit de la commande de copie de fichier. **`cp fichier1 fichier2`** copie `fichier1` dans `fichier2`. Il écrase `fichier2` s'il existait auparavant (voir l'Exemple 15.6, « Copier les fichiers du répertoire courant vers un autre répertoire en utilisant **xargs** »).

Astuce

Les options `-a`, pour l'archive (copier une arborescence entière de répertoire), -u pour la mise à jour (qui empêche l'écrasement de fichiers de même nom, `-r` et `-R` pour la récursivité sont particulièrement utiles.

```
cp -u rep_source/* rep_dest
#  "Synchronise" rep_dest_dir à partir de rep_source
#+  en copiant tous les nouveaux fichiers auparavant inexistants.
```

mv

C'est la commande de déplacement (*move*) de fichier. Elle est équivalente à une combinaison des commandes **cp** et **rm**. Elle peut être utilisée pour déplacer plusieurs fichiers vers un répertoire ou même pour renommer un répertoire. Pour des exemples d'utilisation dans un script, voir l'Exemple 9.20, « Renommer des extensions de fichiers : » et l'Exemple A.2, « *rn* : Un utilitaire simple pour renommer des fichiers ».

Note

Lors de l'utilisation de **mv** dans un script non-interactif, on doit ajouter l'option `-f` (*forcer*) pour empêcher l'interaction avec l'utilisateur.

Quand un répertoire est déplacé vers un répertoire déjà existant, il devient un sous-répertoire du répertoire existant.

```
bash$ mv rep_source rep_cible

bash$ ls -lF rep_cible
total 1
 drwxrwxr-x    2 bozo  bozo      1024 nov 21 23:30 rep_source/
```

rm

Efface, supprime (« remove » en anglais) un ou plusieurs fichiers. L'option `-f` force même la suppression de fichiers en lecture seule et est utile pour ignorer toute interaction de l'utilisateur durant son exécution dans un script.

Note

La commande *rm* échouera, d'elle-même, dans la suppression des fichiers commençant par un tiret. Pourquoi ? parce que *rm* voit un nom de fichier préfixé par un tiret d'*option*.

```
bash$ rm -mauvaisnom
rm: invalid option -- b
 Try `rm --help' for more information.
```

Un contournement intelligent permet de précéder le nom du fichier avec un « -- » (l'option *end-of-options*).

```
bash$ rm -- -mauvaisnom
```

Une autre méthode revient à ajouter au début du nom du fichier un point et un slash.

```
bash$ rm ./-mauvaisnom
```

Avertissement

Lorsqu'elle est exécutée avec l'option de récursivité (NdT : en anglais, « recursive flag ») `-r`, cette commande efface les fichiers de tous les sous-répertoires de l'arborescence à partir du répertoire actuel. Lancer **rm -rf *** sans faire trop attention peut supprimer une grosse partie de la structure d'un répertoire.

rmdir

Efface un répertoire (« remove directory » en anglais). Il est nécessaire que le répertoire soit vide de tout fichier -- ce qui inclut les fichiers invisibles (NdT : en anglais, les *dotfiles*), [1] -- pour que cette commande s'exécute correctemment.

mkdir

Crée un répertoire (NdT : « make directory » en anglais). Par exemple, **`mkdir -p projet/programmes/Decembre`** crée le répertoire indiqué. L'option *`-p`* s'occupe, au besoin, de la création des répertoires parents automatiquement.

chmod

Change les attributs d'un fichier ou d'un répertoire existant (voir l'Exemple 14.13, « Forcer une déconnexion »).

```
chmod +x nom_fichier
# Rend "nom_fichier" exécutable pour tous les utilisateurs.

chmod u+s nom_fichier
# Active le bit de droit "suid" de "nom_fichier".
#  Un utilisateur ordinaire peut exécuter "nom_fichier" avec les mêmes
#+ droits que son propriétaire.
# (Ceci ne s'applique pas aux scripts shell.)
```

```
chmod 644 nom_fichier
#  Active les droits de lecture/écriture de "nom_fichier" pour son
#+ propriétaire et lecture seulement pour
# les autres (mode octal).

chmod 444 nom_fichier
#  Rend "nom_fichier" en lecture seule pour tous.
#  Modifier le fichier (par exemple, avec un éditeur texte)
#+ non autorisé pour un utilisateur qui ne possède par le fichier (sauf root),
#+ et même le propriétaire du fichier doit forcer une sauvegarde du fichier
#+ en cas de modification.
#  Les même restrictions s'appliquent à la suppression du
fichier.</programlisting></para>
```

```
chmod 1777 nom_rep
#  Donne à tout le monde les droits de lecture, d'écriture et d'exécution
#+ dans le répertoire mais active aussi le "sticky bit".
#  Cela signifie que seul le propriétaire du répertoire, le propriétaire du
#+ fichier et, bien sûr, root peuvent effacer un fichier de ce
#+ répertoire.

chmod 111 nom_rep
#  Donne seulement le droit d'exécution à tout le monde dans un répertoire.
#  Ceci signifie que vous pouvez exécuter et lire les fichiers de ce
#+ répertoire (les droits d'exécution incluent nécessairement les droits de
#+ lecture car vous ne pouvez pas exécuter un fichier sans le lire).
#  Mais vous ne pouvez pas lister les fichiers ou les rechercher avec la
#+ commande "find".
#  Ces restrictions ne s'appliquent pas à root.

chmod 000 nom_rep
#  Aucun droit pour ce répertoire.
#  Impossible de lire, écrire ou exécuter les fichiers qui y sont contenus.
#  Impossible même d'y lister les fichiers ou de s'y déplacer.
#  Mais vous pouvez renommer (mv) le répertoire
#+ ou le supprimer (rmdir) s'il est vide.
#  Vous pouvez même créer un lien symbolique vers les fichiers de ce répertoire
#+ mais vous ne pouvez pas lire, écrire ou exécuter les liens symboliques.
#  Ces restrictions ne s'appliquent pas à root.
```

chattr

Change les **attr**ibuts de fichier (NdT : « change file attributes » en anglais). Ceci est analogue à **chmod** ci-dessus mais avec

[1] Il s'agit de fichiers dont le nom commence par un point, par exemple `~/.Xdefaults`. De tels noms de fichiers ne sont pas affichés lors d'un **ls**, et ne risquent donc pas d'être effacés accidententellement par une commande **rm -rf ***. Ces fichiers sont utilisés habituellement en tant que fichiers de configuration situés dans le répertoire principal d'un utilisateur.

des options différentes et une syntaxe d'appel différente. Cela fonctionne seulement sur un système de fichiers *ext2*.

Une option particulièrement intéressante de **chattr** est `i`. **chattr +i `filename`** marque le fichier comme non modifiable. Le fichier ne peut pas être modifié ou supprimé, un lien ne peut pas être établi vers lui, *y compris par root*. Cet attribut de fichier ne peut être initialisé ou supprimé que par *root*. D'une façon similaire, l'option `a` marque le fichier de façon à ce que les utilisateurs ne puissent qu'ajouter des informations.

```
root# chattr +i fichier1.txt
root# rm fichier1.txt
rm: remove write-protected regular file `file1.txt'? y
rm: cannot remove `file1.txt': Operation not permitted
```

Si le fichier a l'attribut `s` (sécurité), alors, quand il est supprimé, les blocs sont écrasés avec des zéros sur le disque.

Si le fichier a l'attribut `u` (non supprimable), alors, à sa suppression, son contenu pourra toujours être récupéré.

Si un fichier a l'attribut `c` (compression), alors il sera automatiquement compressé lors de son écriture sur le disque et décompressé lors de sa lecture.

Note

Les attributs du fichier configurés avec **chattr** ne s'affichent pas dans la liste des fichiers (**ls -l**).

ln

Crée des liens vers des fichiers déjà existants. Un « lien » est une référence vers un fichier. La commande **ln** permet de référencer le fichier lié par plus d'un nom et est une alternative supérieure au système d'alias (voir l'Exemple 4.6, « *wh*, recherche d'un nom de domaine avec *whois* »).

ln crée simplement une référence, un pointeur vers le fichier pour une taille de seulement quelques octets.

La commande **ln** est le plus souvent utilisée avec l'option `-s`, option de lien symbolique ou ou lien « soft ». Les avantages de l'utilisation de l'option `-s` est que cela permet de faire des liens entre systèmes de fichiers ou des répertoires.

La syntaxe de la commande est un peu spéciale. **`ln -s ancien_fichier nouveau_fichier`** lie le fichier `ancien_fichier` au lien nouvellement créé, `nouveau_fichier`.

Attention

Si un fichier nommé `nouveau_fichier` existe, un message d'erreur apparaîtra.

Quel type de lien utiliser ?

Comme John Macdonald l'explique :

Les deux types de liens permettent un référencement multiple -- si vous éditez le contenu du fichier quelque soit le nom utilisé, vos changements affecteront à la fois l'original et le nouveau lien (physique ou symbolique). La différence se situe à un niveau plus élevé de l'arborescence. L'avantage d'un lien physique est que le nouveau nom est totalement indépendant de l'ancien -- si vous supprimez ou renommez l'ancien nom, ceci n'affecte pas le lien physique, qui continue à pointer vers la donnée alors qu'un lien symbolique pointerait toujours vers l'ancien nom qui n'existerait plus. L'avantage d'un lien symbolique est qu'il peut se référer à un autre système de fichier (car il est seulement une référence à un nom de fichier et non pas aux données réelles). Et, contrairement à un lien physique, un lien symbolique peut faire référence à un répertoire.

Les liens permettent d'appeller un script (ou tout autre type d'exécutable) avec des noms multiples et de faire en sorte que ce script se comporte suivant la façon dont il a été appelé.

Exemple 15.2. Hello or Good-bye

```
#!/bin/bash
# hello.sh: Dire "bonjour" ou "bonsoir"
#+          suivant la façon dont le script a été appelé.

# Faire un lien dans le répertoire courant ($PWD) vers ce script :
#    ln -s bonjour.sh bonsoir
```

```
# Maintenant, essayez d'appeler le script de deux façons :
# ./bonjour.sh
# ./bonsoir


APPEL_BONJOUR=65
APPEL_BONSOIR=66

if [ $0 = "./bonsoir" ]
then
  echo "Bonsoir !"
  # Autres commandes du type au-revoir, de façon approprié.
  exit $APPEL_BONSOIR
fi

echo "Bonjour !"
# Autres commandes du type bonjour, de façon approprié.
exit $APPEL_BONJOUR
```

man, **info**

Ces commandes accèdent aux pages de manuel et d'information relatives aux commandes systèmes et autres utilitaires installés sur la machine. Les pages *info*, si disponibles, contiennent habituellement des descriptions bien plus détaillées que celles des pages *man*.

15.2. Commandes complexes

Commandes pour utilisateurs plus expérimentés

find

-exec *COMMANDE* \;

Exécute *COMMANDE* sur chaque fichier trouvé par **find**. La séquence de commandes se termine par un ; (le « ; » est échappé39 pour être certain que le shell le passe de façon littérale à **find**, sans l'interpréter comme un caractère spécial).

```
bash$ find ~/ -name '*.txt'
/home/bozo/.kde/share/apps/karm/karmdata.txt
/home/bozo/misc/irmeyc.txt
/home/bozo/test-scripts/1.txt
```

Si *COMMAND* contient {}, alors **find** substitue le chemin complet du fichier sélectionné à « {} ».

```
find ~/ -name 'core*' -exec rm {} \;
# Supprime tous les fichiers core à partir du répertoire de l'utilisateur.
```

```
find /home/bozo/projects -mtime 1
#  Liste tous les fichiers situés dans le répertoire /home/bozo/projects
#+ qui ont été modifiés il y a, au plus tard, 24 heures.
#
#  mtime = date de dernière modification du fichier cible
#  ctime = date de dernier changement d'état (via 'chmod' ou autre)
#  atime = date du dernier accès

REP=/home/bozo/fichiers_bidons
find "$REP" -type f -atime +5 -exec rm {} \;
#                                    ^^
#  Les accolades sont un indicateur pour le chemin trouvé par "find."
#
#  Efface tous les fichiers contenus dans "/home/bozo/fichiers_bidons"
#+ qui n'ont pas été accédés depuis au moins 5 jours.
#
#  "-type typefichier", où
#  f = fichier classique
```

```
#  d = répertoire
#  l = lien symbolique, etc.
#  (La page de manuel et la page info de 'find' en ont une liste complète.)
```

```
find /etc -exec grep '[0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*[.][0-9][0-9]*' {} \;

#  Trouve toutes les adresses IP (xxx.xxx.xxx.xxx) contenues dans les fichiers
#+ situés dans le répertoire /etc .
#  Quelques correspondances n'auront rien à voir - Peuvent-elles être
#+ éliminées ?

# Peut-être en faisant:

find /etc -type f -exec cat '{}' \; | tr -c '.[:digit:]' '\n' \
 | grep '^[^.][^.]*\.[^.][^.]*\.[^.][^.]*\.[^.][^.]*$'
#
#  [:digit:] est un ensemble de caractères POSIX 1003.2
#+ introduit avec le standard POSIX 1003.2.

# Merci, Stéphane Chazelas.
```

Note

L'option `-exec` de **find** ne doit pas être confondue avec la commande intégrée du shell exec.

Exemple 15.3. *Badname* élimine dans le répertoire courant les fichiers dont le nom contient des caractères incorrects et des espaces blancs.

```
#!/bin/bash
# badname.sh

# Efface les fichiers du répertoire courant contenant des mauvais caractères.

for nomfichier in *
do
  mauvaisnom=`echo "$nomfichier" | sed -n /[\+\{\;\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
# mauvaisnom=`echo "$nomfichier" | sed -n '/[+{;"\=?~()&lt;&gt;&*|$]/p'`  fonctionne
aussi.
#  Supprime les fichiers contenant les "mauvais" caractères :
#+   + { ; " \ = ? ~ ( ) < > & * | $
#
  rm $mauvaisnom 2>/dev/null
#                ^^^^^^^^^^^ Supression des messages d'erreur.
done

# Maintenant, faire attention aux noms de fichiers contenant des espaces blancs.
find . -name "* *" -exec rm -f {} \;
# Le chemin du fichier trouvé par _find_ remplace "{}".
#  Le '\' nous assure que le ';' est interprété littéralement, c'est-à-dire comme
une fin
#+ de commande.

exit 0

#-----------------------------------------------------------------------
# Les commandes ci-dessous ne seront pas exécutées à cause de la commande
# _exit_ au dessus.

# Voici une alternative au script ci-dessus:
find . -name '*[+{;"\\=?~()&lt;&gt;&*|$ ]*' -maxdepth 0 \
-exec rm -f '{}' \;
#  L'option "-maxdepth 0" nous assure que _find_ ne cherchera pas dans les
#+ sous-répertoires de $PWD.

# (Merci, S.C.)
```

Exemple 15.4. Effacer un fichier par son numéro d'*inode*

```
#!/bin/bash
# idelete.sh : Effacer un fichier grâce à son inode.

#  C'est très utile quand un nom de fichier commence avec un caractère illégal,
#+ comme un ? ou -.

NBARGS=1          # L'argument du nom de fichier doit être passé au script.
E_MAUVAISARGS=70
E_FICHIER_INEXISTANT=71
E_CHANGE_D_ESPRIT=72

if [ $# -ne "$NBARGS" ]
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

if [ ! -e "$1" ]
then
  echo "Le fichier \""$1"\" n'existe pas."
  exit $E_FICHIER_INEXISTANT
fi

inum=`ls -i | grep "$1" | awk '{print $1}'`
# inum = inode (NdT : index node) numéro de fichier
# -----------------------------------------------------------------------
#  Chaque fichier possède un inode contenant ses informations d'adresses
#+ physiques.
# -----------------------------------------------------------------------

echo; echo -n "Effacer vraiment \"$1\" (o/n)? "
# L'option '-v' de 'rm' pose la même question.
read reponse
case "$reponse" in
[nN]) echo "Vous avez changé d'avis."
      exit $E_CHANGE_D_ESPRIT
      ;;
*)    echo "Effacement en cours du fichier \"$1\".";;
esac

find . -inum $inum -exec rm {} \;
#                          ^^
#        Les accolades sont des emplacements réservés
#+       pour la sortie de texte par "find".
echo "Fichier "\"$1"\" effacé !"

exit 0
```

La commande **find** fonctionne aussi sans l'option `-exec`.

```
#!/bin/bash
#  Trouve les fichiers suid root.
#  Un fichier suid étrange pourrait indiquer une faille de sécurité
#+ voire même une intrusion dans le système.

repertoire="/usr/sbin"
# Essayer aussi /sbin, /bin, /usr/bin, /usr/local/bin, etc.
droits="+4000"  # suid root (dangerous!)


for fichier in $( find "$repertoire" -perm "$droits" )
do
  ls -ltF --author "$fichier"
done
```

Voir l'Exemple 15.30, « Utiliser *cpio* pour déplacer un répertoire complet », l'Exemple 3.4, « Sauvegarde de tous les fichiers modifiés dans les dernières 24 heures » et l'Exemple 10.9, « Rechercher les auteurs de tous les binaires d'un répertoire » pour des exemples de scripts utilisant **find**. La page de manuel de cette commande, complexe et puissante, apporte des détails supplémentaires.

xargs

Un filtre qui sert à passer des paramètres à une commande, et aussi un outil pour réunir les commandes elles-mêmes. Il découpe un flux de données en des morceaux suffisamment petits pour laisser les filtres et les commandes opérer. Considérez-le comme une puissante alternative aux guillemets inversés141. Dans les situations où la substitution de commandes141 échoue avec une erreur too many arguments (trop d'arguments), utiliser **xargs** règle souvent les problèmes. [2] Habituellement, **xargs** lit depuis `stdin` ou depuis un tube mais il accepte aussi de lire dans la sortie d'un fichier.

La commande par défaut d'**xargs** est echo. Cela signifie que tout flux entrant transmis via un tube vers **xargs** peut voir ses sauts de ligne et caractères d'espacements supprimés.

```
bash$ ls -l
total 0
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 fichier1
 -rw-rw-r--    1 bozo  bozo         0 Jan 29 23:58 fichier2


bash$ ls -l | xargs
total 0 -rw-rw-r-- 1 bozo bozo 0 Jan 29 23:58 fichier1 -rw-rw-r-- 1 bozo bozo 0
Jan...



bash$ find ~/mail -type f | xargs grep "Linux"
./misc:User-Agent: slrn/0.9.8.1 (Linux)
 ./sent-mail-jul-2005: hosted by the Linux Documentation Project.
 ./sent-mail-jul-2005: (Linux Documentation Project Site, rtf version)
 ./sent-mail-jul-2005: Subject: Criticism of Bozo's Windows/Linux article
 ./sent-mail-jul-2005: while mentioning that the Linux ext2/ext3 filesystem
 . . .
```

`ls | xargs -p -l gzip` : Compresse avec gzip tous les fichiers du répertoire courant, un à un, et demande confirmation avant chaque opération.

Note

Notez que *xargs* traite les arguments qui lui sont passés séquentiellement, *un à la fois*.

```
bash$ find /usr/bin | xargs file
/usr/bin:          directory
 /usr/bin/foomatic-ppd-options:          perl script text executable
 . . .
```

Astuce

Une option intéressante d'*xargs* est `-n` *`NN`*, qui limite à *`NN`* le nombre d'arguments passés.

`ls | xargs -n 8 echo` : Affiche le contenu du répertoire courant sur 8 colonnes.

Astuce

Une autre option utile est `-0`, combinée avec **`find -print0`** ou **`grep -lZ`**. Ceci permet de manipuler les arguments contenant des espaces ou des quotes.

[2] Et même quand *xargs* n'est pas strictement nécessaire, il peut accélérer l'exécution d'une commande impliquant le traitement en flot de plusieurs fichiers.

find / -type f -print0 | xargs -0 grep -liwZ GUI | xargs -0 rm -f

grep -rliwZ GUI / | xargs -0 rm -f

N'importe laquelle des commande ci-dessus effacera tout fichier contenant « GUI ». *(Merci, S.C.)*

Or:

```
cat /proc/"$pid"/"$OPTION" | xargs -0 echo
#  Formate la sortie :     ^^^^^^^^^^^^^^^
#  À partir de la correction de Han Holl sur le script
"get-commandline.sh"
#+ du chapitre "/dev et /proc".
```

Exemple 15.5. Fichier de traces utilisant *xargs* pour surveiller les journaux système

```
#!/bin/bash

# Génère un journal de traces dans le répertoire courant à partir de la fin de
# /var/log/messages.

# Note : /var/log/messages doit être lisible par tout le monde si le script
# est appelé par un utilisateur simple.
#         #root chmod 644 /var/log/messages

LIGNES=5

( date; uname -a ) >>fichiertraces
# Date, heure et nom de l'ordinateur
echo ------------------------------------------------------------------------
>>fichiertraces
tail -n LIGNES /var/log/messages | xargs |  fmt -s >>fichiertraces
echo >>fichiertraces
echo >>fichiertraces

exit 0

#  Note:
#  ----
#  Frank Wang précise que les guillemets qui ne se suivent pas (soit des
#+ simples soit des doubles) dans le fichier source pourraient donner
#+ une indigestion à xargs.
#
#  Il suggère d'utiliser la substitution suivante pour la ligne 15 :
#     tail -n $LIGNES /var/log/messages | tr -d "\"'" | xargs | fmt -s >>logfile



#  Exercice:
#  --------
#  Modifier ce script pour tracer les modifications dans /var/log/messages
#+ à des intervalles de 20 minutes.
#  Astuce : utilisez la commande "watch".
```

Comme avec **find**195, une paire d'accolades sert à indiquer un texte à remplacer.

Exemple 15.6. Copier les fichiers du répertoire courant vers un autre répertoire en utilisant xargs

```
#!/bin/bash
# copydir.sh

#  Copie verbeusement tous les fichiers du répertoire courant ($PWD)
#+ dans le répertoire spécifié sur la ligne de commande

E_NOARGS=65
```

```
if [ -z "$1" ]   # Quitte si aucun paramètre n'est fourni.
then
  echo "Usage: `basename $0` rep-destination"
  exit $E_NOARGS
fi

ls . | xargs -i -t cp ./{} $1
#            ^^ ^^      ^^
#  -t est l'option "verbeuse" (affiche la ligne de commande sur stderr).
#  -i est l'option de "remplacement des chaînes".
#  {} est un emplacement réservé pour le texte en sortie.
#  C'est similaire en utilisation à une paire d'accolades pour "find."
#
#  Liste les fichiers du répertoire courant (ls .),
#+ utilise la sortie de "ls" comme argument pour "xargs" (options -i -t),
#+ puis copie (cp) ces arguments ({}) vers le nouveau répertoire ($1).
#
#  Le résultat net est l'équivalent exact de
#+   cp * $1
#+ sauf si un des noms de fichiers contient des espaces blancs.

exit 0
```

Exemple 15.7. Tuer des processus par leur nom

```
#!/bin/bash
# kill-byname.sh: Tuer des processus suivant leur nom.
# Comparez ce script avec kill-process.sh.

#  Par exemple,
#+ essayez "./kill-byname.sh xterm" --
#+ et regardez toutes les xterm disparaître de votre bureau.

#  Attention :
#  -----------
#  C'est un script assez dangereux.
#  Lancez-le avec précaution (spécialement en tant que root)
#+ car il peut causer des pertes de données et d'autres effets indésirables.

E_MAUVAISARGUMENTS=66

if test -z "$1"  # Aucun argument n'a été fourni en ligne de commande ?
then
  echo "Usage: `basename $0` Processus_à_tuer"
  exit $E_MAUVAISARGUMENTS
fi


NOM_PROCESSUS="$1"
ps ax | grep "$NOM_PROCESSUS" | awk '{print $1}' | xargs -i kill {} 2&>/dev/null
#                                                      ^^      ^^

# ------------------------------------------------------------------------
# Notes:
# -i est l'option des chaînes de remplacement d'xargs.
# Les accolades sont l'emplacement du remplacement.
# 2&>/dev/null supprime les messages d'erreurs non souhaités.
#
# grep "$NOM_PROCESSUS" peut-il être remplacé par pidof "$NOM_PROCESSUS" ?
# ------------------------------------------------------------------------

exit $?

#  La commande "killall" a le même effet que ce script,
#+ mais l'utiliser n'a pas le même effect éducatif.
```

Exemple 15.8. Analyse de la fréquence des mots en utilisant *xargs*

```
#!/bin/bash
# wf2.sh : Analyse crue de la fréquence des mots sur un fichier texte.

# Utilise 'xargs' pour décomposer les lignes de texte en des mots simples.
# Comparez cet exemple avec le script "wf.sh" qui suit.


# Vérification du fichier en entrée sur la ligne de commande.
ARGS=1
E_MAUVAISARG=65
E_FICHIERINEXISTANT=66

if [ $# -ne "$ARGS" ]
# Est-ce que le bon nombre d'arguments a été passé au script ?
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARG
fi

if [ ! -f "$1" ]       # Vérifie si le fichier existe.
then
  echo "Le fichier \"$1\" n'existe pas."
  exit $E_FICHIERINEXISTANT
fi



########################################################
cat "$1" | xargs -n1 | \
#  Liste le fichier, un mot par ligne.
tr A-Z a-z | \
# Transforme les caractères en minuscule.
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' | \
#  Filtre les points et les virgules
#+ et remplace l'espace entre les mots par des retours chariot,
  sort | uniq -c | sort -nr
# Finalement, ajoute en préfixe le nombre d'occurence et le trie.
########################################################

#  Ceci fait le même travail que l'exemple "wf.sh" qui va suivre,
#+ mais il est un peu plus lourd et fonctionne moins rapidement (pourquoi ?).

exit 0
```

expr

Évaluateur d'expression : Concatène et évalue les arguments suivant l'opération souhaitée (les arguments doivent être séparés par des espaces). Les opérations peuvent être arithmétiques, comparatives, chaînes de caractères ou logiques.

expr 3 + 5
renvoie 8

expr 5 % 3
renvoie 2

expr 1 / 0
renvoie le message d'erreur, expr: division by zero

Opérations arithmétiques illégales non autorisées.

expr 5 * 3
renvoie 15

L'opérateur de multiplication doit être échappé lorsqu'il est utilisé dans une expression arithmétique avec **expr**.

y=`expr $y + 1`

Incrémente une variable, de la même manière que **`let y=y+1`** et **`y=$(($y+1))`**. Ceci est un exemple d'expansion arithmétique147.

`z=`expr substr $chaine $position $longueur``
Extrait une sous-chaîne de caractères de $longueur caractères, en partant de $position.

Exemple 15.9. Utiliser *expr*

```
#!/bin/bash

# Démonstration des possibilités de 'expr'
# ========================================

echo

# Opérations arithmétiques
# ------------------------

echo "Opérations arithmétique"
echo
a=`expr 5 + 3`
echo "5 + 3 = $a"

a=`expr $a + 1`
echo
echo "a + 1 = $a"
echo "(incrémentation d'une variable)"

a=`expr 5 % 3`
# modulo
echo
echo "5 mod 3 = $a"

echo
echo

# Opérations logiques
# -------------------

#  Retourne 1 si vrai, 0 si faux,
#+ à l'opposé des conventions de Bash.

echo "Opérations logiques"
echo

x=24
y=25
b=`expr $x = $y`         # Test d'égalité.
echo "b = $b"            # 0  ( $x -ne $y )
echo

a=3
b=`expr $a \> 10`
echo 'b=`expr $a \> 10`, donc...'
echo "If a > 10, b = 0 (faux)"
echo "b = $b"            # 0  ( 3 ! -gt 10 )
echo

b=`expr $a \< 10`
echo "If a < 10, b = 1 (vrai)"
echo "b = $b"            # 1  ( 3 -lt 10 )
echo
# Notez l'échappement des opérations.

b=`expr $a \<= 3`
echo "If a <= 3, b = 1 (vrai)"
echo "b = $b"            # 1  ( 3 -le 3 )
# Il y a aussi l'opérande "\>=" (plus grand que ou égal à).
```

```
echo
echo



# Opérateur de chaine de caractères
# ---------------------------------

echo "Opérateur de chaînes de caractères"
echo

a=1234zipper43231
echo "Voici \"$a\"."

# length : longueur de la chaine
b=`expr length $a`
echo "La taille de \"$a\" est $b."

# index : position du premier caractère dans une sous-chaine
#         qui correspond à un caractère dans la chaine.
b=`expr index $a 23`
echo "La position du premier \"2\" dans \"$a\" est \"$b\"."

#  substr : extrait une sous-chaîne, en spécifiant la position de départ et la
#+          taille
b=`expr substr $a 2 6`
echo "La sous-chaîne de \"$a\", commençant à la position 2,\
et long de 6 caractère est \"$b\"."


#  L'attitude par défaut de l'opérateur 'match' est de
#+ chercher une occurence à partir ***du début*** de la chaîne.
#
#        utilisation des expressions régulières
b=`expr match "$a" '[0-9]*'`               #  Comptage numérique.
echo Le nombre de chiffres au début de \"$a\" est $b.
b=`expr match "$a" '\([0-9]*\)'`           #  Notez que les parenthèses échappées
#                   ==      ==              + déclenchent une reconnaissance de
#                                           + sous-chaîne.
echo "Les chiffres au début de \"$a\" sont \"$b\"."

echo

exit 0
```

Important

L'opérateur : est équivalent à **match**. Par exemple, **`b=`expr $a : [0-9]*`** est l'équivalent exact de **`b=`expr match $a [0-9]*`** du listing précédent.

```
#!/bin/bash

echo
echo "Opérations avec des chaînes utilisant la construction \"expr
\$string : \"
echo
"==========================================================================="
echo

a=1234zipper5FLIPPER43231

echo "La chaîne en cours est     \"`expr "$a" : '\(.*\)'`\"."
# Parenthèses échappées groupant l'opérateur.    ==  ==

#       *********************************
#+          Les parenthèses échappées
```

```
#+       correspondent à une sous-chaîne.
#       ********************************


#  Si les parenthèses ne sont pas échappées...
#+ alors 'expr' convertit la chaîne en un entier.

echo "La taille de \"$a\" est `expr "$a" : '.*'`."   # Taille de la chaîne

echo "Le nombre de chiffre au début de \"$a\" est `expr "$a" : '[0-9]*'`."

#
------------------------------------------------------------------------
#

echo

echo "Les chiffres au début de \"$a\" sont `expr "$a" : '\([0-9]*\)'`."
#                                                 ==      ==
echo "Les sept premières lettres de \"$a\" sont `expr "$a" :
'\(.......\)'`."
#           =========                                   ==       ==
# Encore, les parenthèses échappées forcent une correspondance de
sous-chaîne
#
echo "Les sept dernières lettres de \"$a\" sont `expr "$a" :
'.*\(.......\)'`."
#           =========          fin de l'opérateur chaîne  ^^
#  (en fait, ça signifie qu'il saute un ou plus jusqu'à une sous-chaîne
#+  spécifiée)

echo

exit 0
```

Le script ci-dessus illustre comment **expr** utilise les opérateurs groupant appelés *parenthèses echappées* -- *\(... \)* -- en tandem avec une analyse basée sur les expressions rationnelles313 pour faire coïncider une sous-chaîne de caractères. Voici un autre exemple, cette fois provenant de la « vie réelle. »

```
# Supprimer les espaces en début et en fin.
LRFDATE=`expr "$LRFDATE" : '[[:space:]]*\(.*\)[[:space:]]*$'`

#  Provient du script "booklistgen.sh" de Peter Knowles,
#+ script convertissant des fichiers au format Librie/PRS-50X de Sony.
#  (http://booklistgensh.peterknowles.com)
```

Perl437, sed631 et awk634 ont des capacités d'analyse de chaînes de caractères très largement supérieures. Une petite sous-routine **sed** ou **awk** dans un script (voir la Section 33.3, « Scripts d'appel ») est aussi une bonne alternative à l'utilisation d'**expr**.

Voir la Section 9.2, « Manipuler les chaînes de caractères » pour en savoir plus sur l'utilisation d'**expr** dans les manipulations des chaînes de caractères.

15.3. Commandes de date et d'heure

L'heure/date et le chronométrage

date

Exécutée telle quelle, **date** affiche la date et l'heure sur `stdout`. Cette commande devient intéressante grâce à ses options de présentation et d'analyse.

Exemple 15.10. Utiliser *date*

```
#!/bin/bash
```

```
# S'exercer avec la commande 'date'

echo "Le nombre de jours écoulés depuis le début de l'année `date +%j`."
# On a besoin d'un '+' au début pour demander un formatage correct.
# %j donne le jour de l'année.

echo "Le nombre de secondes écoulées depuis 01/01/1970 est `date +%s`."
#  %s affiche le nombre de secondes écoulées depuis le début de l'époque UNIX,
#+ mais quelle est son utilité ?

prefixe=temp
suffixe=$(date +%s)  # L'option "+%s" de 'date' est spécifique à GNU.
nomfichier=$prefixe.$suffixe
echo $nomfichier
#  C'est intéressant pour créer des noms de fichiers temporaires "uniques",
#+ voire mieux que d'utiliser $$.

#  Voir la page de manuel de 'date' pour plus d'informations sur ses
#+ possibilités de présentation.

exit 0
```

L'option `-u` renvoie la date au format UTC (Temps Universel Coordonné).

```
bash$ date
Fri Mar 29 21:07:39 MST 2002



bash$ date -u
Sat Mar 30 04:07:42 UTC 2002
```

Cette option facilite le calcul du temps entre des dates différentes.

Exemple 15.11. Calcul de *Date*

```
#!/bin/bash
# date-calc.sh
# Auteur : Nathan Coulter
# Utilisé avec sa permission dans le guide ABS (merci !).

MPHR=60    # Minutes par heure.
HPD=24     # Heures par jour.

diff () {
        printf '%s' $(( $(date -u -d"$CIBLE" +%s) -
                        $(date -u -d"$ACTUEL" +%s)))
#                       %d = day of month.
}


ACTUEL=$(date -u -d '2007-09-01 17:30:24' '+%F %T.%N %Z')
CIBLE=$(date -u -d'2007-12-25 12:30:00' '+%F %T.%N %Z')
# %F = date complète, %T = %H:%M:%S, %N = nanosecondes, %Z = fuseau horaire.

printf '\nEn 2007, %s ' \
       "$(date -d"$ACTUEL +
        $(( $(diff) /$MPHR /$MPHR /$HPD / 2 )) jours" '+%d %B')"
#       %B = nom du mois                  ^ à moitié
printf 'était entre %s ' "$(date -d"$ACTUEL" '+%d %B')"
printf 'et %s\n' "$(date -d"$CIBLE" '+%d %B')"

printf '\nSur %s à %s, ils étaient \n' \
        $(date -u -d"$ACTUEL" +%F) $(date -u -d"$ACTUEL" +%T)
DAYS=$(( $(diff) / $MPHR / $MPHR / $HPD ))
```

```
CURRENT=$(date -d"$ACTUEL +$DAYS days" '+%F %T.%N %Z')
HOURS=$(( $(diff) / $MPHR / $MPHR ))
CURRENT=$(date -d"$ACTUEL +$HOURS hours" '+%F %T.%N %Z')
MINUTES=$(( $(diff) / $MPHR ))
CURRENT=$(date -d"$ACTUEL +$MINUTES minutes" '+%F %T.%N %Z')
printf '%s jours, %s heures, ' "$DAYS" "$HOURS"
printf '%s minutes et %s secondes ' "$MINUTES" "$(diff)"
printf "jusqu'au repas de Noel!\n\n"

#  Exercice :
#  ---------
#  Ré-écrire la fonction diff() pour accepter des paramètres passés,
#+ plutôt que d'utiliser des variables globales.
```

La commande *date* a un certain nombre d'options d'*affichage*. Par exemple, `%N` donne la partie nanosecondes de l'heure. Une utilisation intéressante pour ceci est de générer des entiers sur six chiffres au hasard.

```
date +%N | sed -e 's/000$//' -e 's/^0//'
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Supprime les zéros en début et en fin s'ils existent.
```

Il existe beaucoup d'autres options (essayez **man date**).

```
date +%j
# Affiche le numéro du jour dans l'année (nombre de jours écoulés depuis le 1er
janvier).

date +%k%M
#  Affiche l'heure et les minutes dans un format sur 24 heures, en une chaîne
#+ composée d'un seul nombre.



# Le paramètre 'TZ' permet la surcharge du fuseau horaire par défaut.
date                 # Mon Mar 28 21:42:16 MST 2005
TZ=EST date          # Mon Mar 28 23:42:16 EST 2005
# Merci à Frank Kannemann et Pete Sjoberg pour l'astuce.


SixJoursAvant=$(date --date='6 days ago')
UnMoisAvant=$(date --date='1 month ago')  # Quatre semaines avant (pas un mois).
UneAnneeAvant=$(date --date='1 year ago')
```

Voir aussi l'Exemple 3.4, « Sauvegarde de tous les fichiers modifiés dans les dernières 24 heures ».

zdump

Affichage de fuseau horaire : Affiche la date dans le fuseau horaire spécifié.

```
bash$ zdump EST
EST  Mar Sep 18 22:09:22 2001 EST
```

time

Renvoie des statistiques très détaillées sur le temps d'exécution d'une commande.

`time ls -l /` va donner quelque chose d'équivalent à ceci :

```
0.00user 0.01system 0:00.05elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
 0inputs+0outputs (149major+27minor)pagefaults 0swaps
```

Voir aussi la commande similaire times de la section précédente.

Note

À partir de la version 2.0465 de Bash, **time** est devenu un mot réservé du shell, avec un comportement très légèrement différent dans un tube.

touch

Utilitaire de mise à jour de la date d'accès/modification d'un fichier à partir de la date système courante ou d'une date spécifiée mais aussi utile pour créer un nouveau fichier. La commande **`touch zzz`** créera un nouveau fichier `zzz` de taille nulle, en admettant bien entendu que `zzz` n'existait pas auparavant. Marquer de cette façon des fichiers vides est utile pour stocker la date, par exemple pour garder trace des modifications de date sur un projet.

Note

La commande **touch** est équivalent à **`: >> nouveaufichier`** ou **`>> nouveaufichier`** (pour des fichiers ordinaires).

Astuce

Avant de faire un cp -u (*copy/update*), utilisez **touch** pour mettre à jour le marquage horaire des fichiers que vous ne voulez pas écraser.

Comme exemple, si le répertoire `/home/bozo/tax_audit` contient les fichiers `spreadsheet-051606.data`, `spreadsheet-051706.data` et `spreadsheet-051806.data`, alors exécuter un **touch spreadsheet*.data** protégera ces fichiers d'un écrasement par des fichiers de même nom durant un **cp -u /home/bozo/financial_info/spreadsheet*data /home/bozo/tax_audit**.

at

La commande de contrôle de job **at** exécute une liste de commandes données à l'heure souhaitée. À première vue, **at** ressemble à cron. Cependant, **at** sert surtout à exécuter d'un coup une liste de commandes.

`at 2pm January 15` demande une liste de commandes à exécuter à cette heure précise. Ces commandes devraient être compatibles avec un script shell car, en pratique , l'utilisateur écrit un script shell exécutable une ligne à la fois. L'entrée se termine avec un Ctrl-D.

En utilisant le paramètre `-f` ou la redirection d'entrée (<), **at** lit une liste de commandes depuis un fichier. Ce fichier est un script shell exécutable, bien qu'il devrait être non-interactif. Il est particulièrement malin d'inclure la commande run-parts dans le fichier pour exécuter un différent jeu de scripts.

```
bash$ at 2:30 am Friday < at-jobs.liste
job 2 at 2000-10-27 02:30
```

batch

La commande de contrôle de job **batch** est similaire à **at**, mais elle exécute une liste de commande quand la charge système tombe en dessous de `.8`. Comme **at**, elle peut lire les commandes depuis un fichier avec le paramètre `-f`.

Le concept de *traitement en flot* date du début de l'informatique. Il signifie l'exécution d'un ensemble de commandes sans intervention de l'utilisateur.

cal

Affiche un calendrier mensuel correctement formaté vers `stdout`. **cal** affichera l'année en cours ou bien un large intervalle d'années passées et futures.

sleep

C'est l'équivalent shell d'une *boucle d'attente*. Elle attend durant le nombre spécifié de secondes, ne faisant rien. Elle peut être utile pour un timing ou dans les processus tournant en tâche de fond, en attente d'un évenement spécifique vérifié par intervalle, tel que dans l'Exemple 29.6, « Nettoyage après un **Control-C** ».

```
sleep 3
# Attend 3 secondes.
```

Note

La commande **sleep** se base par défaut sur les secondes, mais des minutes, des heures ou des jours peuvent aussi être spécifiés.

```
sleep 3 h
# Attend 3 heures!
```

Note

La commande watch pourrait être un meilleur choix que **sleep** pour lancer des commandes à des intervalles réguliers.

usleep

Microsleep (le *u* peut être lu de la même manière que la lettre Grecque *mu* ou *micro*). Elle fonctionne de manière identique à **sleep**, décrit juste au dessus, sauf qu'elle « attend » à partir de délai en micro-secondes. On peut l'utiliser pour des chronométrages très fins ou pour interroger un processus en cours à des intervalles très fréquents.

```
usleep 30
# Attend 30 micro-secondes.
```

Cette commande fait partie du paquetage Red Hat *initscripts / rc-scripts*.

Attention

La commande **usleep** ne permet pas des chronométrages particulièrement précis et n'est donc pas adaptée pour des boucles aux temps critiques.

hwclock, **clock**

La commande **hwclock** accède à ou ajuste l'horloge de la machine. Quelques options requièrent les privilèges du super-utilisateur (*root*). Le fichier de démarrage `/etc/rc.d/rc.sysinit` utilise **hwclock** pour ajuster l'heure système depuis l'horloge machine durant le démarrage.

La commande **clock** est un synonyme de **hwclock**.

15.4. Commandes d'analyse de texte

Commandes affectant le texte et les fichiers textes

sort

Outil de tri de fichier, souvent utilisée dans un tube pour trier. Cette commande trie un *flux de texte* ou un fichier, ascendant ou descendant, ou selon diverses clés ou positions de caractère. Avec l'option `-m`, elle combine des fichiers pré-triés. La *page info* recense ses multiples possibilités et options. Voir l'Exemple 10.9, « Rechercher les auteurs de tous les binaires d'un répertoire », l'Exemple 10.10, « Afficher les *liens symboliques* dans un répertoire » et l'Exemple A.8, « *makedict* : Créer un *dictionnaire* ».

tsort

Tri topologique, lisant chaque paire de mots séparés par un espace et triant en fonction des motifs donnés. Le but original de **tsort** était de trier une liste des dépendances pour une version obsolète de l'éditeur de liens *ld* dans une « ancienne » version d'UNIX.

Le résultat d'un *tsort* diffère habituellement du résultat de la commande **sort**, décrite ci-dessus.

uniq

Ce filtre élimine les lignes dupliquées depuis un fichier trié. On le voit souvent dans un tube combiné avec un sort.

```
cat liste-1 liste-2 liste-3 | sort | uniq > liste.finale
# Concatène les fichiers liste,
# les trie,
# efface les lignes doubles,
# et enfin écrit le résultat dans un fichier de sortie.
```

L'option très utile `-c` préfixe chaque ligne du fichier d'entrée avec son nombre d'occurence.

```
bash$ cat fichiertest
Cette ligne apparaît une seule fois.
Cette ligne apparaît deux fois.
Cette ligne apparaît deux fois.
Cette ligne apparaît trois fois.
Cette ligne apparaît trois fois.
Cette ligne apparaît trois fois.


bash$ uniq -c fichiertest
      1 Cette ligne apparaît une seule fois.
      2 Cette ligne apparaît deux fois.
      3 Cette ligne apparaît trois fois.


bash$ sort fichiertest | uniq -c | sort -nr
      3 Cette ligne apparaît trois fois.
      2 Cette ligne apparaît deux fois.
      1 Cette ligne apparaît trois fois.
```

La commande **`sort FICHIER_ENTREE | uniq -c | sort -nr`** renvoie la liste contenant le *nombre d'occurence* des lignes du fichier `FICHIER_ENTREE` (l'option `-nr` de **sort** produit un tri numérique inversé). Ce type de recherche trouve son utilité dans l'analyse de fichiers de traces et de dictionnaires, ainsi que là où la structure lexicale d'un document doit être examinée.

Exemple 15.12. Analyse de fréquence d'apparition des mots

```
#!/bin/bash
# wf.sh : Compte la fréquence de répétition des mots d'un fichier texte.
# Ceci est une version plus efficace du script "wf2.sh".

# Vérifie si un fichier a été fourni en ligne de commande.
ARGS=1
E_MAUVAISARGS=65
E_FICHIERINEXISTANT=66

if [ $# -ne "$ARGS" ]   #  Le nombre d'arguments passés au script
                        #+ est-il correct ?
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

if [ ! -f "$1" ]       # Est-ce que le fichier existe ?
then
  echo "Le fichier \"$1\" n'existe pas."
  exit $E_FICHIERINEXISTANT
fi



################################################################################
# main ()
sed -e 's/\.//g'  -e 's/\,//g' -e 's/ /\
/g' "$1" | tr 'A-Z' 'a-z' | sort | uniq -c | sort -nr
#                           ========================
#                           Fréquence des occurrences
```

```
#  Enlève les points et les virgules, et
#+ change les espaces entre les mots en retours chariot,
#+ puis met les lettres en minuscule et
#+ enfin préfixe avec le nombre d'apparition et
#+ effectue un tri numérique.

#  Arun Giridhar suggère la modification de ce qui est ci-dessus par :
#  . . . | sort | uniq -c | sort +1 [-f] | sort +0 -nr
#  Ceci ajoute une clé de tri secondaire, donc les instances des mêmes
#+ occurences sont triées alphabétiquement.
#  Comme il l'explique :
#  "Ceci est effectivement un tri radical, le premier étant sur la colonne
#+ la moins significatrice
#+ (mot ou chaîne, et une option pour ne pas être sensible à la casse)
#+ et le dernier étant la colonne la plus significative (fréquence)."
#
#  Ainsi que l'explique Frank Wang, voici l'équivalent de ci-dessus
#+       . . . | sort | uniq -c | sort +0 -nr
#+ et le reste fonctionne aussi :
#+       . . . | sort | uniq -c | sort -k1nr -k
###############################################################################

exit 0

# Exercices:
# ---------
#  1) Ajouter une commande 'sed' pour supprimer les autres ponctuations, comme
#+  les deux-points.
#  2) Modifier le script pour filtrer aussi les espaces multiples et autres espaces
blancs.
```

```
bash$ cat fichiertest
Cette ligne apparaît une fois.
Cette ligne apparaît deux fois.
Cette ligne apparaît deux fois.
Cette ligne apparaît trois fois.
Cette ligne apparaît trois fois.
Cette ligne apparaît trois fois.


bash$ ./wf.sh fichiertest
      6 Cette
      6 apparaît
      6 ligne
      3 fois
      3 trois
      2 deux
      1 une
```

expand, **unexpand**

Souvent utilisé dans un tube, **expand** transforme les tabulations en espaces.

unexpand transforme les espaces en tabulations. Elle inverse les modifications d'**expand**.

cut

Un outil d'extraction de champs d'un fichier. Il est similaire à la commande **`print $N`** de awk634 mais en plus limité. Il peut être plus simple d'utiliser *cut* dans un script plutôt que *awk*. À noter les options `-d` (délimitation) et `-f` (spécification du champ).

Utiliser **cut** pour obtenir une liste des systèmes de fichiers montés :

```
cut -d ' ' -f1,2 /etc/mtab
```

Utiliser **cut** pour avoir l'OS et la version du noyau :

```
uname -a | cut -d" " -f1,3,11,12
```

Utiliser **cut** pour extraire les en-têtes des messages depuis un dossier de courriers électroniques :

```
bash$ grep '^Subject:' messages-lus | cut -c10-80
Re: Linux suitable for mission-critical apps?
MAKE MILLIONS WORKING AT HOME!!!
Spam complaint
Re: Spam complaint
```

Utiliser **cut** pour analyser un fichier :

```
# Montre tous les utilisateurs compris dans /etc/passwd.

FICHIER=/etc/passwd

for utilisateur in $(cut -d: -f1 $FICHIER)
do
  echo $utilisateur
done

# Merci à Oleg Philon pour cette suggestion.
```

`cut -d ' ' -f2,3 fichier` est équivalent à **`awk -F'[ ]' '{ print $2, $3 }' fichier`**

Note

Il est même possible de spécifier un saut de ligne comme délimiteur. L'astuce revient à embarquer un retour chariot (**RETURN**) dans la séquence de la commande.

```
bash$ cut -d'
 ' -f3,7,19 testfile
Ceci est la ligne 3 du fichier de test.
Ceci est la ligne 7 du fichier de test.
Ceci est la ligne 19 du fichier de test.
```

Merci pour cette précision, Jaka Kranjc.

Voir aussi l'Exemple 15.47, « Conversion de base ».

paste

Outil pour fusionner différents fichiers dans un seul fichier multi-colonne. Combiné avec cut, c'est utile pour créer des fichiers de traces.

join

Considérez-le comme un cousin de **paste** mais à usage spécifique. Ce puissant outil permet de fusionner deux fichiers d'une façon significative, qui crée essentiellement une simple version de base de données relationelle.

join travaille sur deux fichiers mais récupère seulement les lignes qui possèdent un champ commun (un nombre par exemple) et écrit le résultat vers `stdout`. Les fichiers joints doivent être triés de la même façon sur le champ cible pour que la correspondance fonctionne correctement.

```
Fichier: 1.donnees

100 Chaussures
200 Bretelles
300 Cure-dents
```

```
Fichier: 2.donnees

100 $40.00
200 $1.00
300 $2.00
```

```
bash$ join 1.donnees 2.donnees
Fichier: 1.donnees 2.donnees

 100 Chaussures $40.00
 200 Bretelles $1.00
 300 Cure-dents $2.00
```

Note

Les champs de sélection apparaîtront seulement une fois dans le résultat.

head

Affiche le début d'un fichier sur `stdout`. Par défaut `10` lignes, mais c'est modifiable. Elle possède de nombreuses options.

Exemple 15.13. Quels fichiers sont des scripts ?

```
#!/bin/bash
# script-detector.sh : Detecte les scripts qui sont dans un répertoire.

TESTCHARS=2    # Teste les 2 premiers caractères.
SHABANG='#!'   # Les scripts commencent toujours avec un "#!"

for fichier in *  # Parcours tous les fichiers du répertoire courant.
do
  if [[ `head -c$TESTCHARS "$fichier"` = "$SHABANG" ]]
  #      head -c2                     #!
  #  L'option '-c' de "head" n'affiche que le nombre spécifié de
  #+ caractères, plutôt que de lignes (par défaut).
  then
    echo "Le fichier \"$fichier\" est un script."
  else
    echo "Le fichier \"$fichier\" n'est *pas* un script."
  fi
done

exit 0

#  Exercices :
#  ----------
#  1) Modifiez ce script pour prendre comme argument optionnel
#+    le répertoire à parcourir pour les scripts
#+    (plutôt que seulement le répertoire en cours).
#
#  2) Actuellement, ce script donne des "faux positifs" pour les
#+    scripts des langages Perl, awk, etc.
#     Corrigez ceci.
```

Exemple 15.14. Générer des nombres aléatoires de dix chiffres

```
#!/bin/bash
# rnd.sh : Affiche un nombre aléatoire de dix chiffres

# Script de Stephane Chazelas.

head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'


# ============================================================== #

# Analyse
# -------
```

```
# head:
# -c4 prend les quatre premiers octets.

# od:
# -N4 limite la sortie à quatre octets.
# -tu4 sélectionne le format de sortie décimal non-signé.

# sed:
# -n , combiné avec le drapeau "p" de la commande "s",
# n'affiche que les lignes correspondantes.



# L'auteur explique ci-après le fonctionnement de 'sed'.

# head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# ---------------------------------> |

# On dit que ce que l'on va --------> |
# envoyer à "sed" ------------------> |
# est 0000000 1198195154\n ---------> |

# sed commence par lire les caractères: 0000000 1198195154\n.
# Ici, il trouve un caractère de saut de ligne,
# donc il est prêt pour commencer à s'occuper de la première ligne (0000000
1198195154).
# Il regarde ses <intervalle><action>s. La première est

#   intervalle  action
#   1           s/.* //p

# Le numéro de ligne est dans l'échelle, donc il exécute l'action :
# essaie de substituer la chaîne la plus longue finissant avec une espace dans la
ligne
# ("0000000 ") avec rien (//), et s'il réussit, affiche le résultat
# ("p" est une option à la commande "s", c'est différent de la commande "p").

# sed est maintenant prêt à continuer la lecture de son entrée.
# (Notez qu'avant de continuer, si l'option -n n'a pas été fournie,
# sed aurait affiché de nouveau la ligne).

# Maintenant, sed lit le reste des caractères et trouve la fin du fichier.
# Il est maintenant prêt à traiter la deuxième ligne (qui est aussi numérotée
# '$' comme la dernière).
# Il voit qu'elle ne correspond pas à <intervalle>, donc son travail est terminé.

# En quelques mots, cette commande sed signifie :
# "Sur la première uniquement, supprime tout caractère jusqu'à l'espace le plus à
droite,
# puis affiche-le."

# Une meilleure façon d'y parvenir aurait été :
#           sed -e 's/.* //;q'

# Ici, deux <intervalle><action>s pourraient avoir été écrit
#           sed -e 's/.* //' -e q):

#   intervalle                      action
#   rien (correspond à la ligne)    s/.* //
#   rien (correspond à la ligne)    q (quit)

# Ici, sed lit seulement la première ligne en entrée.
# Il réalise les deux actions et affiche la ligne (substituée) avant de quitter
# (à cause de l'action "q") car l'option "-n" n'est pas passée.

# ================================================================== #

# Une alternative plus simple au script d'une ligne ci-dessus serait :
#           head -c4 /dev/urandom| od -An -tu4
```

```
exit 0
```

Voir aussi l'Exemple 15.39, « Décoder des fichier codés avec uudecode ».

tail

Affiche la fin d'un fichier vers `stdout`. Par défaut `10` lignes mais cela peut être changé. Habituellement utilisé pour voir les changements faits à un fichier de traces avec `-f` qui affiche les lignes ajoutées à un fichier au moment où cela arrive.

Exemple 15.15. Utiliser *tail* pour surveiller le journal des traces système

```
#!/bin/bash

fichier=sys.log

cat /dev/null > $fichier; echo "Crée / efface fichier."
#  Crée ce fichier s'il n'existait pas auparavant,
#+ et le réduit à une taille nulle s'il existait.
#  : > fichier   et   > fichier marchent aussi.

tail /var/log/messages > $fichier
#  /var/log/messages doit avoir le droit de lecture pour que ce programme
#+ fonctionne.

echo "$fichier contient la fin du journal système."

exit 0
```

Astuce

Pour lister une ligne spécifique d'un fichier texte, envoyez la sortie d'un **head** via un tube à **tail -n 1**. Par exemple, **`head -n 8 database.txt | tail -n 1`** liste la huitième ligne du fichier `database.txt`.

Pour configurer une variable avec un bloc donné d'un fichier texte :

```
var=$(head -n $m $nomfichier | tail -n $n)

# nomfichier = nom du fichier
# m = nombre de lignes du début du fichier jusqu'à la fin du bloc
# n = nombre de lignes à récupérer (depuis la fin du bloc)
```

Note

Les nouvelles implémentations de **tail** rendent obsolètes l'utilisation de **tail -$LIGNES fichier**. Le **tail -n $LIGNES fichier** standard est correct.

Voir aussi l'Exemple 15.5, « Fichier de traces utilisant *xargs* pour surveiller les journaux système », l'Exemple 15.39, « Décoder des fichier codés avec uudecode » et l'Exemple 29.6, « Nettoyage après un **Control-C** ».

grep

Un outil de recherche qui utilise les expressions rationnelles313. À la base, c'était un filtre du vénérable**d** éditeur de ligne, **`G.Re.P`** : *global - regular expression - print*.

`grep` *`motif`* [*`fichier`*...]
Recherche dans le fichier cible un motif, où *`motif`* peut être un texte littéral ou une expression rationnelle.

```
bash$ grep '[rst]ystem.$' osinfo.txt
The GPL governs the distribution of the Linux operating system.
```

Si aucun fichier n'est spécifié, **grep** travaillera en tant que filtre sur `stdout`, comme dans un tube.

```
bash$ ps ax | grep clock
765 tty1     S      0:00 xclock
901 pts/1    S      0:00 grep clock
```

`-i` active la recherche insensible à la casse.

`-w` recherche seulement les mots entiers.

`-l` liste seulement les fichiers dans lesquels des concordances ont été trouvées, mais pas les lignes correspondantes.

`-r` (récursif) cherche dans le répertoire et les sous-répertoires.

`-n` montre les lignes concordantes avec le numéro de ligne.

```
bash$ grep -n Linux osinfo.txt
2:This is a file containing information about Linux.
6:The GPL governs the distribution of the Linux operating system.
```

`-v` (ou `--invert-match`) *n'affiche pas* les lignes où le motif concorde.

```
grep motif1 *.txt | grep -v motif2

# Recherche dans "*.txt" de "motif1",
# mais ***pas*** "modif2".
```

`-c` (`--count`) affiche le nombre de concordances trouvées, plutôt que de les afficher.

```
grep -c txt *.sgml   # (nombre d'occurences de "txt" dans les fichiers "*.sgml")


#   grep -cz .
#            ^ point
# signifie compter (-c) les objets séparés par des zéros (-z) correspondant à "."
# c'est à dire, ceux qui ne sont pas vides (contenant au moins 1 caractère).
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz .     # 3
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '$'   # 5
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -cz '^'   # 5
#
printf 'a b\nc  d\n\n\n\n\n\000\n\000e\000\000\nf' | grep -c '$'    # 9
# Par défaut, les caractères de fin de ligne (\n) séparent les objets à rechercher.

# Notez que -z est spécifique à GNU "grep"


# Merci, S.C.
```

L'option `--color` (ou `--colour`) marque la chaîne correspondante en couleur (sur la console ou dans une fenêtre *xterm*). Comme *grep* affiche chaque ligne entière contenant le modèle de correspondance, cela vous permettra de voir exactement ce *qui* a déclenché la correspondance. Voir aussi l'option `-o` qui affiche seulement la partie correspondant au modèle dans la ligne.

Exemple 15.16. Afficher les lignes *From* des courriels stockés sous forme de fichiers

```
#!/bin/bash
# from.sh

#  Émule l'outil "from" de Solaris, BSD, etc.
#  Affiche l'en-tête "From" de tous les messages
#+ compris dans votre répertoire de mails.


REPMAIL=~/mail/*              #  Pas de mise entre guillemets de la variable.
Pourquoi ?
OPTS_GREP="-H -A 5 --color"   #  Affiche le fichier, quelques lignes de contexte
                              #+ et affiche "From" en couleur.
```

```
CHAINECIBLE="^From"            # "From" au début de la ligne.

for file in $REPMAIL           #  Pas de mise entre guillemets de la variable.
do
  grep $OPTS_GREP "$CHAINECIBLE" "$file"
  #    ^^^^^^^^^^              #  De nouveau, pas de mise entre guillemets de la
variable.
  echo
done

exit $?

#  La sortie peut être envoyé vers 'more' ou
#+ redirigé dans un fichier...
```

Lorsqu'il est invoqué avec plus d'un fichier cible donné, **grep** spécifie quel fichier contient les concordances.

```
bash$ grep Linux osinfo.txt misc.txt
osinfo.txt:This is a file containing information about Linux.
osinfo.txt:The GPL governs the distribution of the Linux operating system.
misc.txt:The Linux operating system is steadily gaining in popularity.
```

Astuce

Pour forcer **grep** à montrer le nom du fichier pendant la recherche d'un fichier cible, donnez `/dev/null` comme deuxième fichier.

```
bash$ grep Linux osinfo.txt /dev/null
osinfo.txt:This is a file containing information about Linux.
osinfo.txt:The GPL governs the distribution of the Linux operating system.
```

S'il y a une concordance de motif, **grep** renvoie un code de sortie44 0, ce qui le rend utile comme test conditionnel dans un script, en particulier en combinaison avec l'option `-q` pour supprimer la sortie.

```
SUCCES=0                      # si la recherche avec grep est fructueuse
mot=Linux
nomfichier=donnees.fichier

grep -q "$mot" "$nomfichier"    # -q supprime l'affichage vers stdout

if [ $? -eq $SUCCES ]
# if grep -q "$mot" "$nomfichier"   peut remplacer les lignes 5 à 7.
then
  echo "$mot trouvé dans $nomfichier"
else
  echo "$mot introuvable dans $nomfichier"
fi
```

L'Exemple 29.6, « Nettoyage après un **Control-C** » montre comment utiliser **grep** pour chercher un mot dans un journal de traces.

Exemple 15.17. Émuler *grep* dans un script

```
#!/bin/bash
# grp.sh : Une réimplémentation brute de 'grep'.

E_MAUVAISARGS=65

if [ -z "$1" ]    # Vérification standard des arguments en ligne de commande.
then
  echo "Usage: `basename $0` motif"
  exit $E_MAUVAISARGS
```

```
fi

echo

for fichier in *          # Parcourt tous les fichiers dans $PWD.
do
  sortie=$(sed -n /"$1"/p $fichier)  # Substitution de commande.

  if [ ! -z "$sortie" ]   # Que se passe t-il si "$sortie" n'est pas
                          # entre guillemets ?
  then
    echo -n "$fichier: "
    echo $sortie
  fi     #  sed -ne "/$1/s|^|${fichier}: |p"  est l'équivalent de dessus.

  echo
done

echo

exit 0

# Exercices :
# ----------
# 1) Ajoutez des sauts de lignes à la sortie,
#    s'il y a plus d'une correspondance dans n'importe quel fichier donné.
# 2) Ajoutez des nouvelles possibilités.
```

Comment **grep** peut-il chercher deux modèles (ou plus) ? Que faire si vous voulez que **grep** affiche toutes les lignes d'un ou plusieurs fichiers contenant à la fois « modele1 » *et* « modele2 » ?

Une méthode est d'envoyer le résultat du **grep modele1** via un tube dans **grep modèle2**.

Par exemple, étant donné le fichier suivant :

```
# Nom du fichier : fichiertest

Ceci est un fichier d'exemple.
Ceci est un fichier texte ordinaire.
Ce fichier ne contient aucun texte inhabituel.
Ce fichier n'est pas inhabituel.
Voici un peu de texte.
```

Maintenant, cherchons dans ce fichier des lignes contenant *à la fois* « fichier » et « texte »...

```
bash$ grep fichier fichiertest
# Nom du fichier : fichiertest
Ceci est un fichier d'exemple.
Ceci est un fichier texte ordinaire.
Ce fichier ne contient aucun texte inhabituel.
Ce fichier n'est pas inhabituel.

bash$ grep fichier fichiertest | grep texte
Ceci est un fichier texte ordinaire.
Ce fichier ne contient aucun texte inhabituel.
```

Maintenant, pour une utilisation récréative et intéressante de *grep*...

Exemple 15.18. Solutionneur de mots croisés

```
#!/bin/bash
# cw-solver.sh

#  Crossword puzzle and anagramming word game solver.
#  You know *some* of the letters in the word you're looking for,
#+ so you need a list of all valid words
#+ with the known letters in given positions.
```

```
#  For example: w...i....n
#               1???5????10
# w in position 1, 3 unknowns, i in the 5th, 4 unknowns, n at the end.
# (See comments at end of script.)


E_NOPATT=71
DICT=/usr/share/dict/word.lst
#                    ^^^^^^^^   Looks for word list here.
#  ASCII word list, one word per line.
#  If you happen to need an appropriate list,
#+ download the author's "yawl" word list package.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  or
#  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz


if [ -z "$1" ]   #  If no word pattern specified
then             #+ as a command-line argument . . .
  echo           #+ . . . then . . .
  echo "Usage:"  #+ Usage message.
  echo
  echo ""$0" \"pattern,\""
  echo "where \"pattern\" is in the form"
  echo "xxx..x.x..."
  echo
  echo "The x's represent known letters,"
  echo "and the periods are unknown letters (blanks)."
  echo "Letters and periods can be in any position."
  echo "For example, try:   cw-solver.sh w...i....n"
  echo
  exit $E_NOPATT
fi

echo
# ================================================
# This is where all the work gets done.
grep ^"$1"$ "$DICT"   # Yes, only one line!
#    |    |
# ^ is start-of-word regex anchor.
# $ is end-of-word regex anchor.

#  From _Stupid Grep Tricks_, vol. 1,
#+ a book the ABS Guide author may yet get around
#+ to writing one of these days . . .
# ================================================
echo


exit $?  # Script terminates here.
#  If there are too many words generated,
#+ redirect the output to a file.

cw-solver w...i....n

twichildren
wellington
workingman
workingmen
```

egrep -- *grep étendu* -- est comme **grep -E**. Elle utilise un jeu d'expressions rationnelles313 légèrement différent et étendu, ce qui peut rendre une recherche plus flexible. Il accepte aussi l'opérateur booléen | (*or*).

```
bash $ egrep 'correspond|Correspond' fichier.txt
La ligne 1 correspond.
La ligne 3 correspond.
La ligne 4 contient des correspondances mais aussi des Correspondances.
```

fgrep -- *grep rapide* -- comme **grep -F**; recherche une chaîne littérale (pas d'expressions rationnelles313), ce qui accélère en principe le traitement.

Note

Sur certaines distributions Linux, **egrep** et **fgrep** sont des liens symboliques vers, ou des alias de **grep**, mais appelés avec les options `-E` et `-F`, respectivement.

Exemple 15.19. Rechercher des définitions dans *le dictionnaire Webster de 1913*

```
#!/bin/bash
# dict-lookup.sh

#  Ce script recherche des définitions dans le dictionnaire Webster de 1913.
#  Ce dictionnaire du domaine public est disponible au téléchargement à partir de
#+ plusieurs sites, dont celui du projet Gutenberg
(http://www.gutenberg.org/etext/247).
#
#  Convertisez-le du format DOS au format UNIX (seulement LF à la fin d'une ligne)
#+ avant de l'utiliser avec ce script.
#  Stockez le fichier en ASCII pur, non compressé.
#  Configurez la variable DICO_PARDEFAUT ci-dessous avec chemin/nom du fichier.


E_MAUVAISARGS=65
LIGNESCONTEXTEMAX=50                    # Nombre maximum de lignes à afficher.
DICO_PARDEFAUT="/usr/share/dict/webster1913-dict.txt"
               # Fichier dictionnaire par défaut (chemin et nom du fichier).
               # À modifier si nécessaire.
#  Note :
#  -----
#  Cette édition particulière de 1913 de Webster
#+ commence chaque entrée avec une lettre en majuscule
#+ (minuscule pour le reste des caractères).
#  Seule la *toute première ligne* d'une entrée commence de cette façon,
#+ et c'est pourquoi l'algorithme de recherche ci-dessous fonctionne.



if [[ -z $(echo "$1" | sed -n '/^[A-Z]/p') ]]
#  Doit au moins spécifier un mot à rechercher
#+ et celui-ci doit commencer avec une lettre majuscule.
then
  echo "Usage: `basename $0` Mot-à-définir [dictionnaire]"
  echo
  echo "Note : Le mot à rechercher doit commencer avec une majuscule,"
  echo "le reste du mot étant en minuscule."
  echo "-----------------------------------------------"
  echo "Exemples : Abandon, Dictionary, Marking, etc."
  exit $E_MAUVAISARGS
fi


if [ -z "$2" ]                  # Pourrait spécifier un dictionnaire différent
                                # comme argument de ce script.
then
  dico=$DICO_PARDEFAUT
else
  dico="$2"
fi

# -----------------------------------------------------------------
Definition=$(fgrep -A $LIGNESCONTEXTEMAX "$1 \\" "$dico")
#                  Définitions de la forme "Mot \..."
#
#  Et, oui, "fgrep" est assez rapide pour rechercher même dans un très gros fichier.
```

```
# Maintenant, récupérons le bloc de définition.

echo "$Definition" |
sed -n '1,/^[A-Z]/p' |
#  Affiche la première ligne en sortie
#+ jusqu'à la première ligne de la prochaine entrée.
sed '$d' | sed '$d'
#  Supprime les deux dernières lignes en sortie
#+ (une ligne blanche et la première ligne de la prochaine entrée).
# ------------------------------------------------------------------

exit 0

# Exercices :
# ----------
# 1)  Modifiez le script pour accepter tout type de saisie alphabetique
#   + (majuscule, minuscule, mixe), et convertissez-la dans un format acceptable
#   + pour le traitement.
#
# 2)  Convertissez le script en application GUI,
#   + en utilisant quelque chose comme 'gdialog' ou 'zenity' . . .
#     Le script ne prendre plus d'argument(s) en ligne de commande.
# 3)  Modifiez le script pour analyser un des autres dictionnaires disponibles
#   + dans le domaine public, tel que le « U.S. Census Bureau Gazetter »
```

Note

Voir aussi Exemple A.41, « Quacky : un jeu de mots de type Perquackey » pour un exemple de recherche *fgrep* rapide sur un gros fichier texte.

agrep (*grep approximatif*) étend les possibilités de **grep** à une concordance approximative. La chaîne trouvée peut différer d'un nombre spécifié de caractères du motif. Cette commande ne fait pas partie des distributions Linux.

Astuce

Pour chercher dans des fichiers compressés, utilisez **zgrep**, **zegrep** ou **zfgrep**. Ces commandes marchent aussi avec des fichiers non compressés, bien que plus lentement qu'un simple **grep**, **egrep**, **fgrep**. C'est pratique pour chercher dans divers fichiers, compressés ou non.

Pour chercher dans des fichiers compressés avec bzip, utilisez **bzgrep**.

look

La commande **look** fonctionne comme **grep** mais fait une recherche basée sur un « dictionnaire », une liste de mots triés. Par défaut, **look** cherche une correspondance dans `/usr/dict/words` mais un autre dictionnaire peut être utilisé.

Exemple 15.20. Chercher les mots dans une liste pour tester leur validité

```
#!/bin/bash
#  lookup : Effectue une recherche basée sur un dictionnaire sur chaque mot d'un
#+ fichier de données.

fichier=mots.donnees  #  Le fichier de données à partir duquel on lit les mots à
                      #+ tester.

echo

while [ "$mot" != end ]  # Le dernier mot du fichier de données.
do                       # ^^^
  read mot               #  Depuis le fichier de données, à cause de la redirection à la
                         #+ fin de la boucle.
  look $mot > /dev/null  #  Nous ne voulons pas afficher les lignes dans le
                         #+ dictionnaire.
  lookup=$?      # Code de sortie de 'look'.
```

```
  if [ "$lookup" -eq 0 ]
  then
    echo "\"$mot\" est valide."
  else
    echo "\"$mot\" est invalide."
  fi

done <"$fichier"    #  Redirige stdin vers $fichier, donc les "lectures"
                    #+ commencent à partir d'ici.

echo

exit 0

# ----------------------------------------------------------------------------
# Le code ci-dessous ne s'exécutera pas à cause de la commande exit ci-dessus


# Stephane Chazelas propose aussi ce qui suit, une alternative plus concise :

while read mot && [[ $mot != end ]]
do if look "$mot" > /dev/null
   then echo "\"$mot\" est valide."
   else echo "\"$mot\" est invalide."
   fi
done <"$fichier"

exit 0
```

sed, **awk**

Langages de script convenant bien à l'analyse de fichiers texte et des sorties de commandes. Peuvent être utilisés seuls ou conjointement avec des tubes et des scripts shell.

sed631
« Éditeur de flux » non interactif, permettant d'utiliser plusieurs commandes **ex** dans un mode batch. C'est souvent utile dans des scripts shell.

awk634
Extracteur et formateur programmable de fichiers, bon pour la manipulation ou l'extraction de champs (colonnes) dans des fichiers textes structurés. Sa syntaxe est similaire à C.

wc

wc (word count) donne le nombre de mots d'un fichier ou d'un flux :

```
bash $ wc /usr/share/sed-4.1.2/README
13     70     447 /usr/share/sed-4.1.2/README
[13 lignes  70 mots  447 caractères]
```

wc -w donne seulement le nombre de mots.

wc -l donne seulement le nombre de lignes.

wc -c donne le nombre d'octets.

wc -m donne le nombre de caractères.

wc -L donne la taille de la ligne la plus longue.

Utiliser **wc** pour connaître le nombre de fichiers `.txt` dans le répertoire courant :

```
$ ls *.txt | wc -l
#  Cela ne fonctionnera que si aucun fichier "*.txt" ne contient de saut de ligne
dans
#+ son nom.

# D'autres moyens de faire ça :
#      find . -maxdepth 1 -name \*.txt -print0 | grep -cz .
#      (shopt -s nullglob; set -- *.txt; echo $#)
```

```
# Merci, S.C.
```

Utiliser **wc** pour sommer la taille de tous les fichiers dont le nom commence avec une lettre entre d et h

```
bash$ wc [d-h]* | grep total | awk '{print $3}'
71832
```

Utiliser **wc** pour compter le nombre de fois où « Linux » apparaît dans le source de ce document.

```
bash$ grep Linux abs-book.sgml | wc -l
50
```

Voir aussi l'Exemple 15.39, « Décoder des fichier codés avec uudecode » et l'Exemple 19.8, « Boucle *for* redirigée ».

Certaines commandes incluent quelques fonctionnalités de **wc** comme options.

```
... | grep foo | wc -l
# Cette construction fréquemment utilisée peut être plus concise.

... | grep -c foo
# Utiliser l'option "-c" (or "--count") de grep à la place.

# Merci, S.C.
```

tr

Filtre de transposition de caractères.

Attention

Utilisez les guillemets et/ou les parenthèses19, si besoin est. Les guillemets empêchent le shell de réinterpréter les caractères spéciaux dans les séquences de commande de **tr**. Les parenthèses devraient être mises entre guillemets pour empêcher leur expansion par le shell.

`tr "A-Z" "*" < fichier` ou **`tr A-Z \* < fichier`** remplacent toutes les majuscules de `fichier` par des astérisques (le résultat est écrit dans `stdout`). Sur certains systèmes, ça peut ne pas fonctionner. Cependant **`tr A-Z '[**]'`** fonctionnera.

`-d` efface un intervalle de caractères.

```
echo "abcdef"                  # abcdef
echo "abcdef" | tr -d b-d      # aef


tr -d 0-9 < fichierbidon
# Efface tous les chiffres du fichier "fichierbidon".
```

`--squeeze-repeats` (ou `-s`) efface toute occurence sauf la première, d'une chaîne de caractères. Cette option est utile pour supprimer les espaces blancs superflus.

```
bash$ echo "XXXXX" | tr --squeeze-repeats 'X'
X
```

L'option « complément » `-c` *inverse* l'ensemble de caractères à détecter. Avec cette option, **tr** n'agit que sur les caractères ne faisant *pas* partis de l'ensemble spécifiés.

```
bash$ echo "acfdeb123" | tr -c b-d +
+c+d+b++++
```

Notez que **tr** reconnaît les ensembles de caractères POSIX. [3]

```
bash$ echo "abcd2ef1" | tr '[:alpha:]' -
----2--1
```

[3] Ce n'est vrai que pour la version GNU de **tr**, pas pour les versions génériques se trouvant dans les systèmes UNIX commerciaux.

Exemple 15.21. *toupper* : Transforme un fichier en majuscule.

```
#!/bin/bash
# Met en majuscule un fichier

E_MAUVAISARGS=65

if [ -z "$1" ]  # Vérification standard des arguments en ligne de commande.
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

tr a-z A-Z <"$1"

# Même effet que le programme ci-dessus, mais utilisant la notation POSIX :
#       tr '[:lower:]' '[:upper:]' <"$1"
# Merci, S.C.

exit 0

#  Exercice :
#  Réécrire ce script en donnant le choix de modifier un fichier
#+ soit en majuscule soit en minuscule
```

Exemple 15.22. *lowercase* : Change tous les noms de fichier du répertoire courant en minuscule.

```
#! /bin/bash
#
# Change chaque nom de fichier en minuscules dans le répertoire courant.
#
#  Inspiré par un script de John Dubois,
#+ qui fut traduit en Bash par Chet Ramey,
#+ et considérablement simplifié par l'auteur du guide ABS.


for nomfichier in *             # Parcourt tous les fichiers du répertoire.
do
   nomF=`basename $nomfichier`
   n=`echo $nomF | tr A-Z a-z`  # Change le nom en minuscule.
   if [ "$nomF" != "$n" ]       #  Renomme seulement les fichiers qui ne sont
                                #+ pas en minuscules.
   then
     mv $nomF $n
   fi
done

exit $?


# Le code en dessous ne s'exécutera pas à cause de la commande exit ci-dessus
#-------------------------------------------------------------------------#
# Pour le lancer, effacez la ligne de script ci dessus.

#  Le script suivant ne fonctionnera pas sur les fichiers dont le nom a des
#+ espaces ou des caractères de saut de ligne.

# Stephane Chazelas suggère donc l'alternative suivante :


for nomfichier in *  # Pas nécessaire d'utiliser basename,
                     # car "*" ne retourne aucun fichier contenant "/".
do n=`echo "$nomfichier/" | tr '[:upper:]' '[:lower:]'`
#                            Jeu de notation POSIX.
```

```
#                    Le slash est ajouté, comme ça les saut de lignes ne sont
#                    pas supprimés par la substition de commande.
   # Substitution de variable :
   n=${n%/}          #  Supprime le slash de fin, rajouté au dessus, du nom de
                     #+ fichier.
   [[ $nomfichier == $n ]] || mv "$nomfichier" "$n"
                     # Vérifie si le nom de fichier est déjà en minuscules.
done

exit $?
```

Exemple 15.23. *du* : Convertit les fichiers texte DOS vers UNIX.

```
#!/bin/bash
# Du.sh: Convertisseur de fichier texte DOS vers UNIX.

E_MAUVAISARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` fichier-a-convertir"
  exit $E_MAUVAISARGS
fi

NOUVEAUFICHIER=$1.unx

CR='\015'  # Retour chariot.
           # 015 est le code ASCII en octal pour CR.
           # Les lignes d'un fichier texte DOS finissent avec un CR-LF.
           #  Les lignes d'un fichier texte UNIX finissent uniquement avec
           #+ un LF.

tr -d $CR < $1 > $NOUVEAUFICHIER
# Efface les caractères CR et écrit le résultat dans un nouveau fichier.

echo "Le fichier texte DOS original est \"$1\"."
echo "Le fichier texte UNIX converti est \"$NOUVEAUFICHIER\"."

exit 0

# Exercice :
# ---------
# Modifiez le script ci-dessus pour convertir de UNIX vers DOS.
```

Exemple 15.24. *rot13* : rot13, cryptage ultra-faible.

```
#!/bin/bash
# rot13.sh: L'algorithme classique rot13,
#           cryptage qui pourrait berner un gamin de 3 ans.

# Usage: ./rot13.sh nomfichier
# ou     ./rot13.sh <nomfichier
# ou     ./rot13.sh et fournissez une entrée clavier (stdin)

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M'   # "a" devient "n", "b" devient "o", etc.
#  La commande 'cat "$@"'
#+ permet la saisie de données depuis soit stdin soit un fichier.

exit 0
```

Exemple 15.25. Générer des énigmes « Crypto-Citations »

```
#!/bin/bash
# crypto-quote.sh : Crypte les citations

# Cryptage de célèbres citations par une simple substitution de lettres.
#  Le résultat est similaire aux puzzles "Crypto Quote"
#+ vus sur les pages "Op Ed" du journal Sunday.


key=ETAOINSHRDLUBCFGJMQPVWZYXK
# "key" n'est qu'un alphabet mélangé.
# Changer "key" modifie le cryptage.

# L'instruction 'cat "$@"' prend son entrée soit de stdin, soit de fichiers.
# Si stdin est utilisé, il faut terminer la saisie par un Ctrl-D.
# Sinon, spécifier le nom du fichier en tant que paramètre de la ligne de commande.

cat "$@" | tr "a-z" "A-Z" | tr "A-Z" "$key"
#        |  en majuscule  |     crypte
#  Fonctionne avec n'importe quel type de casse : majuscule, minuscule ou les
#+ deux ensemble.
# Les caractères non-alphabétiques restent inchangés.


# Essayer ce script avec :
# "Nothing so needs reforming as other people's habits."
# --Mark Twain
#
# Il en résulte :
# "CFPHRCS QF CIIOQ MINFMBRCS EQ FPHIM GIFGUI'Q HETRPQ."
# --BEML PZERC

# Pour décrypter, utiliser :
# cat "$@" | tr "$key" "A-Z"


#  Ce programme de cryptage bidon peut être cassé par un gosse de 12 ans
#+ en utilisant simplement un papier et un crayon.

exit 0

#  Exercice :
#  ---------
#  Modifier le script de façon à ce qu'il utilise encrypt ou decrypt,
#+ suivant le(s) argument(s) en ligne de commande.
```

Variantes de *tr*

L'utilitaire **tr** a deux variantes historiques. La version BSD n'utilise pas les crochets (**`tr a-z A-Z`**) contrairement à la version SysV (**`tr '[a-z]' '[A-Z]'`**). La version GNU de **tr** ressemble à celle de BSD.

fold

Un filtre qui scinde les lignes entrées à partir d'une taille spécifiée. C'est spécialement utile avec l'option `-s`, qui coupe les lignes à chaque espace (voir l'Exemple 15.26, « Affichage d'un fichier formaté. » et l'Exemple A.1, « *mailformat* : Formater un courrier électronique »).

fmt

Un formateur de fichier tout bête, utilisé en tant que filtre dans un tube pour « scinder » les longues lignes d'un texte.

Exemple 15.26. Affichage d'un fichier formaté.

```
#!/bin/bash

LARGEUR=40                    # Colonnes de 40 caractères.
```

```
b=`ls /usr/local/bin`       # Récupère la liste des fichiers du répertoire

echo $b | fmt -w $LARGEUR

# Aurait pu aussi être fait avec
#  echo $b | fold - -s -w $LARGEUR

exit 0
```

Voir aussi l'Exemple 15.5, « Fichier de traces utilisant *xargs* pour surveiller les journaux système ».

Astuce

Une puissante alternative à **fmt** est **par** de Kamil Toman disponible sur *http://www.cs.berkeley.edu/~amc/Par/*.

col

Cette commande dont le nom est trompeur supprime les sauts de ligne inversés d'un flux en entrée. Elle tente aussi de remplacer les espaces blancs par des tabulations équivalentes. Le rôle principal de **col** est de filtrer la sortie de certains utilitaires de manipulation de textes, tels que **groff** et **tbl**.

column

Formateur de colonnes. Ce filtre transforme le texte écrit façon "liste" en un « joli » tableau par l'insertion de tabulations aux endroits appropriés.

Exemple 15.27. Utiliser *column* pour formater l'affichage des répertoires

```
#!/bin/bash
#  Il s'agit d'une légère modification du fichier d'exemple dans la page de
#+ manuel de "column".


(printf "PERMISSIONS LIENS PROPRIETAIRE GROUPE TAILLE MOIS JOUR HH:MM NOM-PROG\n" \
; ls -l | sed 1d) | column -t

#  "sed 1d" efface la première ligne écrite,
#+ qui devrait être "total       N",
#+ où "N" est le nombre total de fichiers trouvés par "ls -l".

# L'option -t de "column" affiche un tableau bien formaté.

exit 0
```

colrm

Filtre de suppression de colonnes. Ce filtre enlève les colonnes (caractères) d'un fichier et envoie le résultat vers `stdout`. **`colrm 2 4 < fichier`** efface le deuxième par bloc de 4 caractères de chaque ligne du fichier `fichier`.

Attention

Si le fichier contient des tabulations ou des caractères non imprimables, cela peut causer des comportements imprévisibles. Dans de tel cas, pensez à utiliser expand et **unexpand** dans un tube précédant **colrm**.

nl

Filtre de numérotation de lignes. **`nl fichier`** envoie `fichier` sur `stdout` en insérant un nombre au début de chaque ligne non vide. Si `fichier` est omit, alors ce filtre travaillera sur `stdin`.

La sortie de **nl** est très similaire à **`cat -b`** car, par défaut, **nl** ne liste pas les lignes vides.

Exemple 15.28. *nl* : un script d'autonumérotation.

```
#!/bin/bash
# line-number.sh

# Ce script s'affiche deux fois sur stdout en numérotant les lignes.

# 'nl' voit ceci comme la ligne 4 car il ne compte pas les lignes blanches.
# 'cat -n' voit la ligne ci-dessus comme étant la ligne 6.

nl `basename $0`

echo; echo  # Maintenant, essayons avec 'cat -n'

cat -n `basename $0`
# La différence est que 'cat -n' numérote les lignes blanches.
# Notez que 'nl -ba' fera de même.

exit 0
# -----------------------------------------------------------------
```

pr

Filtre d'impression formaté. Ce filtre paginera des fichiers (ou `stdout`) en sections utilisables pour des impressions papier ou pour les voir à l'écran. Diverses options permettent la manipulation des rangées et des colonnes, le regroupement des lignes, la définition des marges, la numérotation des lignes, l'ajout d'en-têtes par page et la fusion de fichiers entre autres choses. La commande **pr** combine beaucoup des fonctionnalités de **nl**, **paste**, **fold**, **column** et **expand**.

`pr -o 5 --width=65 fileZZZ | more` renvoie un joli affichage paginé à l'écran de `fileZZZ` avec des marges définies à 5 et 65.

Une option particulèrement utile est `-d`, forçant le double-espacement (même effet que **sed -G**).

gettext

Le package GNU **gettext** est un ensemble d'utilitaires pour adapter et traduire la sortie de texte des programmes en des langages étrangers. Bien que à l'origine la cible était les programmes C, il supporte maintenant un certain nombre de langages de programmation et de scripts.

Le *programme* **gettext** fonctionne avec les scripts shell. Voir la *`page info`*.

msgfmt

Un programme pour générer des catalogues binaires de messages. Il est utilisé pour la normalisation.

iconv

Un utilitaire pour convertir des fichiers en un codage différent (jeu de caractère). Son rôle principal concerne la normalisation.

```
# Convertit une chaîne d'UTF-8 vers UTF-16 et l'ajoute dans LISTELIVRES
function ecrit_chaine_utf8 {
    CHAINE=$1
    LISTELIVRES=$2
    echo -n "$CHAINE" | iconv -f UTF8 -t UTF16 | \
    cut -b 3- | tr -d \\n >> "$LISTELIVRES"
}

#  Vient du script "booklistgen.sh" de Peter Knowles
#+ permettant de convertir les fichiers au format Librie/PRS-50X de Sony.
#  (http://booklistgensh.peterknowles.com)
```

recode

Considérez-le comme une version puissante d'**iconv**, ci-dessus. Ce très souple utilitaire de conversion d'un fichier dans un jeu de caractère différent. Notez que **recode** ne fait pas partie d'une installation Linux standard.

TeX, **gs**

TeX et **Postscript** sont des langages de balises utilisés pour préparer une impression ou un formatage pour l'affichage vidéo.

TeX est le système "typesetting" élaboré de Donald Knuth. C'est souvent pratique d'écrire un script qui va encapsuler toutes les options et arguments passés à l'un de ces langages.

Ghostscript (**gs**) est un interpréteur GPL de Postscript .

texexec

Outil pour traiter des fichiers *TeX* et *PDF*. Trouvé dans `/usr/bin` dans plusieurs distributions Linux, c'est réellement un emballage shell434 qui appelle Perl437 pour invoquer *TeX*.

```
texexec --pdfarrange --result=Concatené.pdf *pdf

#  Concatène tous les fichiers PDF du répertoire actuel
#+ dans un seul fichier, Concatené.pdf . . .
#  (L'option --pdfarrange repagine un fichier PDF. Voir aussi --pdfcombine.)
#  La commande ci-dessus pourrait être avec « paramétrisée »
#+ et placée dans un script shell.
```

enscript

Outil pour convertir un fichier texte en PostScript

Par exemple, **enscript fichier.txt -p fichier.ps** crée un fichier PostScript `filename.ps`.

groff, **tbl**, **eqn**

Un autre langage de balises est **groff**. C'est la version avancée GNU de la commande UNIX **roff/troff**. Les pages de manuel utilisent **groff**.

tbl, utilitaire de création de tableau est considéré comme faisant partie de **groff**, dans la mesure où sa fonction est de convertir une balise tableau en commandes **groff**.

Le processeur d'équations **eqn** fait aussi parti de **groff** et sa fonction est de convertir une balise d'équation en commandes **groff**.

Exemple 15.29. *manview* : Visualisation de pages man formatées

```
#!/bin/bash
# manview.sh : Formate la source d'une page man pour une visualisation.

#  Ceci est utile lors de l'écriture de la source d'une page man et que vous
#+ voulez voir les résultats intermédiaires lors de votre travail.

E_MAUVAISARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

groff -Tascii -man $1 | less
# De la page man de groff.

# Si la page man inclut des tables et/ou des équations,
# alors le code ci-dessus échouera.
# La ligne suivante peut gérer de tels cas.
#
#   gtbl < "$1" | geqn -Tlatin1 | groff -Tlatin1 -mtty-char -man
#
#   Merci, S.C.

exit 0
```

lex, **yacc**

lex, analyseur lexical, produit des programmes pour la détection de motifs. Ca a été remplacé depuis par **flex**, non propriétaire, sur les systèmes Linux.

L'utilitaire **yacc** crée un analyseur basé sur un ensemble de spécifications. Elle est depuis remplacée par le **bison**, non propriétaire, sur les systèmes Linux.

15.5. Commandes pour les fichiers et l'archivage

Archivage

tar

L'utilitaire standard d'archivage sous UNIX. [4] À l'origine, il s'agissait d'un programme d'archivage sur cassette (*Tape ARchiving*) mais il est devenu un paquet plus généraliste qui peut gérer toutes les façons d'archiver sur tout type de support, allant des lecteurs de bande aux fichiers standards, voire même sur `stdout` (voir l'Exemple 3.4, « Sauvegarde de tous les fichiers modifiés dans les dernières 24 heures »). La version GNU de *tar* a été améliorée pour accepter différents filtres de compression tels que **tar czvf archive_name.tar.gz ***, qui, récursivement, archive et compresse (gzip) tous les fichiers d'un répertoire sauf ceux commençant par un point dans le répertoire courant ($PWD). [5]

Quelques options utiles de **tar** :

1. `-c` crée (une nouvelle archive)
2. `-x` extrait (les fichiers d'une archive existante)
3. `--delete` supprime (les fichiers d'une archive existante)

Attention

Cette option ne fonctionnera pas sur les périphériques à bandes magnétiques.

4. `-r` ajoute (des fichiers à une archive existante)
5. `-A` ajoute (des fichiers *tar* à une archive existante)
6. `-t` liste (le contenu d'une archive existante)
7. `-u` met à jour une archive
8. `-d` compare une archive avec un système de fichiers spécifié
9. `--after-date` traite seulement les fichiers dont la date est située chronologiquement *après* la date spécifiée
10. `-z` compresse l'archive avec gzip

. (compresse ou décompresse suivant que cette option est combinée avec l'option `-c` ou `-x`)

11. `-j` bzip2 l'archive (NdT : autre format de compression)

Attention

Il pourrait être difficile de récupérer des données d'une archive tar corrompue *compressée avec gzip*. Lors de l'archivage de fichiers importants, faites plusieurs copies.

shar

Utilitaire d'*archivage shell*. Les fichiers dans une archive shell sont concaténés sans compression et l'archive qui en résulte est essentiellement un script shell complet, avec l'en-tête #!/bin/sh, et contenant toutes les commandes nécessaires pour déballer l'archive. Les *archives shar* sont toujours montrées sur les groupes de nouvelles Usenet, mais sinon **shar** a été assez bien remplacé par **tar**/**gzip**. La commande **unshar** déballe les archives *shar*.

ar

Utilitaire de création et de manipulation d'archives, principalement utilisé pour des bibliothèques de fichiers binaires.

rpm

Le *gestionnaire de paquetages Red Hat* (*Red Hat Package Manager*, ou **rpm**) apporte une sur-couche pour les archives source ou binaire. Il inclut des commandes pour installer et vérifier l'intégrité des paquetages, en plus d'autres choses.

Un simple **rpm -i nom_paquetage.rpm** suffit généralement à installer un paquetage, bien qu'il y ait bien plus d'options disponibles.

[4] Une *archive* est tout simplement un ensemble de fichiers liés stockés en un même emplacement.
[5] Un **tar czvf archive_name.tar.gz *** *incluera* les fichiers commençant par un point pour les répertoires *compris* dans le répertoire courant. C'est une « fonctionnalité » non documentée de GNU **tar**.

Astuce

`rpm -qf` identifie le paquetage dont provient un fichier.

```
bash$ rpm -qf /bin/ls
coreutils-5.2.1-31
```

Astuce

`rpm -qa` donne une liste complète de tous les paquetages *rpm* installés sur un système donné. Un **`rpm -qa nom_paquetage`** liste seulement le(s) paquetage(s) correspondant à `nom_paquetage`.

```
bash$ rpm -qa
redhat-logos-1.1.3-1
glibc-2.2.4-13
cracklib-2.7-12
dosfstools-2.7-1
gdbm-1.8.0-10
ksymoops-2.4.1-1
mktemp-1.5-11
perl-5.6.0-17
reiserfs-utils-3.x.0j-2
...


bash$ rpm -qa docbook-utils
docbook-utils-0.6.9-2


bash$ rpm -qa docbook | grep docbook
docbook-dtd31-sgml-1.0-10
docbook-style-dsssl-1.64-3
docbook-dtd30-sgml-1.0-10
docbook-dtd40-sgml-1.0-11
docbook-utils-pdf-0.6.9-2
docbook-dtd41-sgml-1.0-10
docbook-utils-0.6.9-2
```

cpio

Cette commande d'archivage spécifique à la copie (**c**o**p**y **i**nput and **o**utput, c'est-à-dire copie l'entrée et la sortie) est rarement utilisé car elle a été supplanté par **tar**/**gzip**. Elle a toujours son utilité, comme lors du déplacement d'un répertoire complet. Avec une taille de bloc appropriée (pour la copie), elle peut être beaucoup plus rapide que **tar**.

Exemple 15.30. Utiliser *cpio* pour déplacer un répertoire complet

```
#!/bin/bash

# Copier un répertoire complet en utilisant cpio.

# Avantages de l'utilisation de 'cpio' :
#   Rapidité de la copie. Il est plus rapide que 'tar' avec des tubes.
#   Convient bien pour copier des fichiers spéciaux (tubes nommés, etc.)
#+  sur lesquels 'cp' pourrait avoir du mal.

ARGS=2
E_MAUVAISARGS=65

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` source destination"
```

```
  exit $E_MAUVAISARGS
fi

source=$1
destination=$2

###############################################################################
find "$source" -depth | cpio -admvp "$destination"
#
#  Lire les pages man de find et cpio pour "décrypter" ces options.
#  La commande ci-dessus fonctionne seulement en relatif par rapport à $PWD
#+ (le répertoire courant)... des chemins complets sont indiqués.

# Exercice :
# ----------

#  Ajoutez du code pour vérifier le code de sortie ($?) du tube 'find | cpio'
#+ et affichez les messages d'erreur appropriés si quelque chose se passe mal.
###############################################################################

exit $?
```

rpm2cpio

Cette commande crée une archive **cpio** à partir d'un rpm.

Exemple 15.31. Déballer une archive *rpm*

```
#!/bin/bash
# de-rpm.sh : Déballe une archive 'rpm'

: ${1?"Usage: `basename $0` fichier_cible"}
# Doit spécifier le nom de l'archive 'rpm' comme argument.


FICHIERTEMP=$$.cpio                       # Fichier temporaire avec un nom "unique".
                                          # $$ est l'identifiant du processus du
script.

rpm2cpio < $1 > $FICHIERTEMP                 # Convertir l'archive rpm archive
                                             # en archive cpio.
cpio --make-directories -F $FICHIERTEMP -i   # Déballe l'archive cpio.
rm -f $FICHIERTEMP                           # Supprime l'archive cpio.

exit 0

#  Exercice :
#  Ajouter une vérification pour
#    1) s'assurer que le "fichier-cible" existe bien et
#+   2) que c'est une archive rpm.
#  Astuce :                   Analysez la sortie de la commande 'file'.
```

Compression

gzip

L'utilitaire de compression standard GNU/UNIX, remplaçant **compress**, inférieur et propriétaire. La commande de décompression correspondante est **gunzip**, qui est l'équivalent de **gzip -d**.

Note

L'option `-c` envoit la sortie de **gzip** sur `stdout`. C'est utile pour envoyer la sortie via un tube à d'autres commandes.

Le filtre **zcat** décompresse un fichier *gzip* vers `stdout`, comme possible entrée à une redirection ou un tube. En fait, ceci est une commande **cat** fonctionnant sur des fichiers compressés (incluant les fichiers créés par l'ancien utilitaire compress). La commande **zcat** est l'équivalent de **gzip -dc**.

Attention

Sur certains systèmes UNIX commerciaux, **zcat** est un synonyme pour **uncompress -c**, et ne fonctionnera pas avec les fichiers compressés avec *gzip*.

Voir aussi l'Exemple 7.7, « *zmore* ».

bzip2

Un autre utilitaire de compression, habituellement plus efficace (mais plus lent) que **gzip**, spécialement sur de gros fichiers. La commande de décompression correspondante est **bunzip2**.

Note

Les nouvelles versions de tar ont acquis le support de **bzip2**.

compress, **uncompress**

C'est un utilitaire de compression plus ancien, propriétaire disponible dans les distributions UNIX commerciales. **gzip**, plus efficace, l'a largement remplacé. Les distributions Linux incluent généralement un **compress** pour des raisons de compatibilité, bien que **gunzip** peut déballer des fichiers traités avec **compress**.

Astuce

La commande **znew** transforme les fichiers *compressés* en fichiers *gzip*.

sq

Encore un autre utilitaire de compression (**sq**ueeze), un filtre qui fonctionne seulement sur les listes de mots ASCII triées. Il utilise la syntaxe standard d'appel pour un filtre, **sq < fichier-entrée > fichier-sortie**. Rapide, mais pas aussi efficace que gzip. Le filtre de décompression correspondant est **unsq**, appelé comme **sq**.

Astuce

La sortie de **sq** peut être envoyé via un tube à **gzip** pour une meilleure compression.

zip, **unzip**

Utilitaire inter-plateforme d'archivage et de compression de fichiers compatible avec DOS *pkzip.exe*. Les archives « Zip » semblent être un medium plus ordinaire pour l'échange de fichiers sur Internet que les « archives tar ».

unarc, **unarj**, **unrar**

Ces utilitaires Linux permettent de déballer des archives compressées avec les programmes DOS *arc.exe*, *arj.exe* et *rar.exe*.

lzma, **unlzma**, **lzcat**

Compression Lempel-Ziv-Markov très efficace. La syntaxe de *lzma* est similaire à celui de *gzip*. Le *site web 7-zip* a plus d'informations.

Informations sur les fichiers

file

Un utilitaire pour identifier le type des fichiers. La commande **`file nom-fichier`** renverra une spécification du fichier `nom-fichier`, telle que `ascii text` ou `data`. Il utilise les numéros magiques trouvés dans `/usr/share/magic`, `/etc/magic` ou `/usr/lib/magic` suivant la distribution Linux/UNIX.

L'option `-f` fait que **file** tourne en mode batch, pour lire à partir d'un fichier désigné une liste de noms de fichiers à analyser. L'option `-z`, lorsqu'elle est utilisé sur un fichier compressé, essaie d'analyser le type du fichier décompressé.

```
bash$ file test.tar.gz
```

```
test.tar.gz: gzip compressed data, deflated,
last modified: Sun Sep 16 13:34:51 2001, os: UNIX

bash file -z test.tar.gz
test.tar.gz: GNU tar archive (gzip compressed data, deflated,
last modified: Sun Sep 16 13:34:51 2001, os: UNIX)
```

```
#  Trouve les scripts sh et Bash dans un
#+ répertoire donné :

REPERTOIRE=/usr/local/bin
MOTCLE=Bourne
# Scripts shell Bourne et Bourne-Again

file $REPERTOIRE/* | fgrep $MOTCLE

# Sortie :

# /usr/local/bin/burn-cd:          Bourne-Again shell script text executable
# /usr/local/bin/burnit:           Bourne-Again shell script text executable
# /usr/local/bin/cassette.sh:      Bourne shell script text executable
# /usr/local/bin/copy-cd:          Bourne-Again shell script text executable
# . . .
```

Exemple 15.32. Supprimer les commentaires des programmes C

```
#!/bin/bash
# strip-comment.sh : Supprime les commentaires (/* COMMENT */) d'un progamme C.

E_SANSARGS=0
E_ERREURARG=66
E_MAUVAIS_TYPE_FICHIER=67

if [ $# -eq "$E_SANSARGS" ]
then
  echo "Usage: `basename $0` fichier-C" >&2 # Message d'erreur vers stderr.
  exit $E_ERREURARG
fi

# Test du type de fichier.
type=`file $1 | awk '{ print $2, $3, $4, $5 }'`
# "file $1" affiche le type du fichier...
# Puis awk supprime le premier champ correspondant au nom du fichier...
# Enfin, le résultat remplit la variable "type".
type_correct="ASCII C program text"

if [ "$type" != "$type_correct" ]
then
  echo
  echo "Ce script fonctionne uniquement sur les fichiers C."
  echo
  exit $E_MAUVAIS_TYPE_FICHIER
fi


# Script sed assez complexe:
#--------
sed '
/^\/\*/d
/.*\*\//d
' $1
#--------
#  Facile à comprendre si vous prenez quelques heures pour apprendre les
#+ concepts de sed.


#  Il est possible d'ajouter une ligne supplémentaire au script sed pour gérer
```

```
#+ le cas où la ligne de code a un commentaire le suivant, sur la même ligne.
#  Ceci est laissé en exercice (difficile).

#  De même, le code ci-dessus supprime les lignes, sans commentaires, avec un
#+ "*/" ou "/*", ce qui n'est pas un effet désirable.

exit 0


# ----------------------------------------------------------------------
# Le code ci-dessous ne s'exécutera pas à cause du 'exit 0' ci-dessus.

# Stephane Chazelas suggère l'alternative suivante :

usage() {
  echo "Usage: `basename $0` fichier-C" >&2
  exit 1
}

BIZARRE=`echo -n -e '\377'`   # ou BIZARRE=$'\377'
[[ $# -eq 1 ]] || usage
case `file "$1"` in
  *"C program text"*) sed -e "s%/\*%${BIZARRE}%g;s%\*/%${BIZARRE}%g" "$1" \
     | tr '\377\n' '\n\377' \
     | sed -ne 'p;n' \
     | tr -d '\n' | tr '\377' '\n';;
  *) usage;;
esac

#  Ceci ne fonctionne pas avec, par exemple :
#+ printf("/*");
#+ ou
#+ /*  /* commentaire intégré bogué */
#
#  Pour gérer tous les cas spécifiques (commentaires dans des chaînes,
#+ commentaires dans des chaînes où se trouve un \", \\" ...) la seule façon est
#+ d'écrire un analyseur C (lex ou yacc peut-être ?).

exit 0
```

which

which commande donne le chemin complet vers « commande ». C'est utile pour trouver si une commande ou un utilitaire particulier est installé sur le système.

```
$bash which rm
```

```
/usr/bin/rm
```

Pour une utilisation intéressante de cette commande, voir Exemple 33.14, « Un jeu de « courses de chevaux » ».

whereis

Similaire à **which**, ci-dessus, **whereis commande** donne le chemin complet vers « commande », mais aussi sa page man.

```
$bash whereis rm
```

```
rm: /bin/rm /usr/share/man/man1/rm.1.bz2
```

whatis

whatis commande recherche « commande » dans la base de données *whatis*. C'est utile pour identifier les commandes système et les fichiers de configuration importants. Considérez-le en tant que commande **man** simplifiée.

```
$bash whatis whatis
```

```
whatis               (1)  - search the whatis database for complete words
```

Exemple 15.33. Explorer `/usr/X11R6/bin`

```
#!/bin/bash

# Que sont tous ces mystérieux binaires dans /usr/X11R6/bin ?

REPERTOIRE="/usr/X11R6/bin"
# Essayez aussi "/bin", "/usr/bin", "/usr/local/bin", etc.

for fichier in $REPERTOIRE/*
do
  whatis `basename $fichier`   # affiche des informations sur le binaire.
done

exit 0
# Vous pouvez souhaiter rediriger la sortie de ce script, de cette façon :
# ./what.sh >>whatis.db
# ou la visualiser une page à la fois sur stdout,
# ./what.sh | less
```

Voir aussi l'Exemple 10.3, « *Fileinfo* : opérer sur une liste de fichiers contenue dans une variable ».

vdir

Affiche une liste détaillée du contenu du répertoire. L'effet est similaire à ls -lb.

Il fait partie de GNU *fileutils*.

```
bash$ vdir
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo

bash ls -l
total 10
 -rw-r--r--    1 bozo  bozo      4034 Jul 18 22:04 data1.xrolo
 -rw-r--r--    1 bozo  bozo      4602 May 25 13:58 data1.xrolo.bak
 -rw-r--r--    1 bozo  bozo       877 Dec 17  2000 employment.xrolo
```

locate, **slocate**

La commande **locate** cherche les fichiers en utilisant une base de données enregistrée pour ce seul but. La commande **slocate** est la version sécurisée de **locate** (qui pourrait être un alias de **slocate**).

```
$bash locate hickson
```

```
/usr/lib/xephem/catalogs/hickson.edb
```

readlink

Révèle le fichier sur lequel pointe un lien symbolique.

```
bash$ readlink /usr/bin/awk
../../bin/gawk
```

strings

Utiliser la commande **strings** pour trouver les chaînes de caractères affichables dans un fichier binaire ou de données. Elle listera les séquences de caractères affichables trouvées dans le fichier cible. C'est intéressant pour un examen rapide (et sale) d'un core dump ou pour regarder un fichier image inconnu (`strings fichier-image | more` pourrait afficher quelque chose comme *JFIF*, qui identifierait le fichier en tant que graphique *jpeg*). Dans un script, vous devriez probablement analyser la sortie de **strings** avec grep ou sed631. Voir l'Exemple 10.7, « Un remplaçant d*grep* pour les fichiers binaires » et l'Exemple 10.9, « Rechercher les auteurs de tous les binaires d'un répertoire ».

Exemple 15.34. Une commande *strings* « améliorée »

```
#!/bin/bash
# wstrings.sh: "word-strings" (commande "strings" améliorée)
#
#  Ce script filtre la sortie de "strings" en la comparant avec une liste de
#+ mots communs.
#  Ceci élimine efficacement le bruit et n'affiche que les mots reconnus.

# =================================================================
#               Vérification standard des arguments du script
ARGS=1
E_MAUVAISARGS=65
E_AUCUNFICHIER=66

if [ $# -ne $ARGS ]
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

if [ ! -f "$1" ]                       # Vérifie si le fichier existe.
then
    echo "Le fichier \"$1\" n'existe pas."
    exit $E_AUCUNFICHIER
fi
# =================================================================


LONGUEUR_CHAINE_MINIMUM=3                #  Longueur minimum d'une chaîne.
FICHIER_MOTS=/usr/share/dict/linux.words  #  Dictionnaire.
                                         #  Vous pouvez spécifier un autre
                                         #+ fichier de mots, à condition que
                                         #+ son format soit d'un mot par ligne.


listemots=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`

# Traduit la sortie de la commande 'strings' avec de multiples passes de 'tr'.
#  "tr A-Z a-z" réalise une conversion en minuscule.
#  "tr '[:space:]'" change les espaces blancs par des Z.
#  "tr -cs '[:alpha:]' Z" change les caractères non alphabétiques en Z.
#+ et ne conserve qu'un seul Z pour les Z successifs.
#  "tr -s '\173-\377' Z" convertit tous les caractères après 'z' en Z
#+ et ne conserve qu'un seul Z pour les Z successifs
#+ ce qui supprime tous les caractères bizarres que la précédente passe aurait
#+ oublié de gérer.
#  Finalement, "tr Z ' '" convertit tous les Z en espaces blancs,
#+ ce qui sera vu comme des mots séparés dans la boucle ci-dessous.

# ************************************************************************
#  Notez la technique de remplissage de la sortie de 'tr' vers lui-même,
#+ mais avec différents arguments et/ou options à chaque passe.
# ************************************************************************


for mot in $listemots    # Important :
                         # $listemots ne doit pas être entre guillemets ici.
                         # "$listemots" ne fonctionne pas.
                         # Pourquoi pas ?
do

  longueur_chaine=${#mot}                    # Longueur de la chaîne.
  if [ "$longueur_chaine" -lt "$LONGUEUR_CHAINE_MINIMUM" ]
  then                   # Ne pas tenir compte des petites chaînes.
    continue
  fi

  grep -Fw $mot "$FICHIER_MOTS"       # Correspond seulement aux mots complets.
#       ^^^                           #  Options "chaînes corrigées" et
```

```
                                        #+ "mots complets".
done

exit $?
```

Comparaison

diff, patch

diff : utilitaire de comparaison de fichiers flexible. Il compare les fichiers cibles ligne par ligne, séquentiellement. Dans certaines applications, telles que la comparaison de dictionnaires de mots, il peut être utile pour filtrer les fichiers avec sort et **uniq** avant de les envoyer via un tube à **diff**. **`diff fichier-1 fichier-2`** affiche en sortie les lignes qui diffèrent des deux fichiers, avec des symboles indiquant à quel fichier appartient la ligne en question.

L'option `--side-by-side` de **diff** affiche en sortie chaque fichier comparé, ligne par ligne, dans des colonnes séparées, et avec les lignes ne correspondant pas marquées. Les options `-c` et `-u` rendent la sortie de la commande plus facile à interpréter.

Il existe de nombreuses interfaces agréables pour **diff**, comme **sdiff**, **wdiff**, **xdiff** et **mgdiff**.

Astuce

La commande **diff** renvoie un état de sortie 0 si les fichiers comparés sont identiques et 1 s'ils ne le sont pas. Cela permet d'utiliser **diff** dans une construction de test à l'intérieur d'un script shell (voir ci-dessous).

Une utilisation commune de **diff** est de générer des fichiers de différences à utiliser avec **patch**. L'option `-e` permet la génération de tels fichiers, à utiliser avec des scripts **ed** ou **ex**.

patch : utilitaire de gestion de versions. Suivant un fichier de différences généré par **diff**, **patch** peut mettre à jour une version précédente d'un paquetage en une nouvelle version. Il est bien plus convenable de distribuer un fichier « diff » sensiblement plus petit que le corps entier du paquetage revu. Les correctifs (« patchs ») du noyau sont devenus la méthode préférée pour distribuer les mises à jour fréquentes du noyau Linux.

```
patch -p1 <correctif
# Prend toutes les modifications indiquées dans 'correctif'
# et les applique aux fichiers référencés dans le correctif.
# Ceci met à jour le paquetage en une nouvelle version.
```

Appliquer un correctif au noyau :

```
cd /usr/src
gzip -cd patchXX.gz | patch -p0
# Mettre à jour le source du noyau en utilisant 'patch'.
# De la documentation du noyau Linux, "README",
# par un auteur anonyme (Alan Cox ?).
```

Note

La commande **diff** peut aussi comparer récursivement les répertoires (et les fichiers qui s'y trouvent).

```
bash$ diff -r ~/notes1 ~/notes2
Only in /home/bozo/notes1: fichier02
Only in /home/bozo/notes1: fichier03
Only in /home/bozo/notes2: fichier04
```

Astuce

Utiliser **zdiff** pour comparer des fichiers *compressés avec gzip*.

Astuce

Utiliser **diffstat** pour créer un histogramme (graphe en distribution points) en sortie de **diff**.

diff3

Une version étendue de **diff** qui compare trois fichiers en une fois. Cette commande renvoie un état de sortie de si l'exécution est réussie mais, malheureusement, cela ne donne aucune information sur le résultat de la comparaison.

```
bash$ diff3 fichier-1 fichier-2 fichier-3
====
 1:1c
   Ceci est la ligne 1 de "fichier-1"
 2:1c
   Ceci est la ligne 1 de "fichier-2"
 3:1c
   Ceci est la ligne 1 de "fichier-3"
```

sdiff

Compare et/ou édite les deux fichiers pour les assembler dans un fichier de sortie. Dû à sa nature interactive, cette commande trouvera peu d'utilité dans un script.

cmp

La commande **cmp** est une version simplifiée de **diff**, ci-dessus. Alors que **diff** reporte les différences entre deux fichiers, **cmp** montre simplement à quel point ils diffèrent.

Note

Comme **diff**, **cmp** renvoie un état de sortie de 0 si les fichiers comparés sont identiques et de 1 s'ils diffèrent. Ceci permet une utilisation dans une construction de test à l'intérieur d'un script shell.

Exemple 15.35. Utiliser *cmp* pour comparer deux fichiers à l'intérieur d'un script.

```
#!/bin/bash

ARGS=2  # Deux arguments attendus par le script.
E_MAUVAISARGS=65
E_ILLISIBLE=66

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` fichier1 fichier2"
  exit $E_MAUVAISARGS
fi

if [[ ! -r "$1" || ! -r "$2" ]]
then
  echo "Les deux fichiers à comparer doivent exister et être lisibles."
  exit $E_ILLISIBLE
fi

cmp $1 $2 &> /dev/null  # /dev/null enterre la sortie de la commande "cmp".
#   cmp -s $1 $2  a le même résultat ("-s" option de silence pour "cmp")
#   Merci à Anders Gustavsson pour nous l'avoir indiqué.
#
# Fonctionne aussi avec 'diff', c'est-à-dire   diff $1 $2 &> /dev/null

if [ $? -eq 0 ]         # Test du code de sortie de la commande "cmp".
then
  echo "Le fichier \"$1\" est identique au fichier \"$2\"."
else
  echo "Le fichier \"$1\" diffère du fichier \"$2\"."
fi
```

```
exit 0
```

Astuce

Utiliser **zcmp** sur des fichiers *gzip*.

comm

Utilitaire de comparaison de fichiers souple. Les fichiers doivent être triés pour qu'il soit utile.

comm *-options premier-fichier second-fichier*

`comm fichier-1 fichier-2` affiche trois colonnes :

- colonne 1 = lignes uniques à `fichier-1`
- colonne 2 = lignes uniques à `fichier-2`
- colonne 3 = lignes communes aux deux.

Les options permettent la sortie d'une ou plusieurs colonnes.

- `-1` supprime la colonne `1`
- `-2` supprime la colonne `2`
- `-3` supprime la colonne `3`
- `-12` supprime les deux colonnes `1` et `2`, etc.

Cette commande est utile pour comparer des « dictionnaires » ou des *listes de mots* -- fichiers texte triés avec un mot par ligne.

Utilitaires

basename

Supprime le chemin d'un nom de fichier en affichant seulement le nom. La construction **`basename $0`** permet au script de connaître son nom, c'est-à-dire le nom par lequel il a été invoqué. Ceci peut être utilisé pour les messages d'« usage » si, par exemple, un script est appelé sans ses arguments :

```
echo "Usage: `basename $0` arg1 arg2 ... argn"
```

dirname

Supprime le **basename** d'un nom de fichier en n'affichant que le chemin.

Note

basename et **dirname** peuvent s'exécuter sur des chaînes de caractères arbitraires. L'argument n'a pas besoin de faire référence à un fichier existant, voire même un fichier (voir l'Exemple A.7, « *days-between* : Calculer le nombre de jours entre deux dates »).

Exemple 15.36. *basename* et *dirname*

```
#!/bin/bash

a=/home/bozo/daily-journal.txt

echo "Nom de base       de /home/bozo/daily-journal.txt = `basename $a`"
echo "Nom du répertoire de /home/bozo/daily-journal.txt = `dirname $a`"
echo
echo "Mon répertoire personnel est `basename ~/`."
  # `basename ~` fonctionne aussi.
```

```
echo "Le chemin de mon répertoire personnel est `dirname ~/`."
  # `dirname ~` fonctionne aussi.

exit 0
```

split, **csplit**

Utilitaires pour diviser un fichier en plusieurs petites parties. Ils sont habituellement utilisés pour diviser un gros fichier en fichiers tenant sur une disquette ou pour préparer un courrier électronique ou pour les télécharger.

La commande **csplit** divise un fichier suivant le *contexte*, la division se faisant lorsqu'il y a correspondance de modèles.

Exemple 15.37. Un script qui se copie lui-même en sections

```
#!/bin/bash
# splitcopy.sh

#  A script that splits itself into chunks,
#+ then reassembles the chunks into an exact copy
#+ of the original script.

CHUNKSIZE=4    #  Size of first chunk of split files.
OUTPREFIX=xx   #  csplit prefixes, by default,
               #+ files with "xx" ...

csplit "$0" "$CHUNKSIZE"

# Some comment lines for padding . . .
# Line 15
# Line 16
# Line 17
# Line 18
# Line 19
# Line 20

cat "$OUTPREFIX"* > "$0.copy"  # Concatenate the chunks.
rm "$OUTPREFIX"*               # Get rid of the chunks.

exit $?
```

Codage et chiffrement

sum, **cksum**, **md5sum**, **sha1sum**

Ces utilitaires ont pour but de vérifier une somme de contrôle. Une *somme de contrôle* est un nombre calculé à partir du contenu d'un fichier, dans le but de vérifier son intégrité. Un script peut se référer à une liste de sommes de contrôle pour des raisons de sécurité, comme pour s'assurer que des fichiers clés du système n'ont pas été modifié ou corrompu. Pour les applications de sécurité, utilisez la commande **md5sum** (**m**essage **d**igest **5** check**sum**) ou, encore mieux, le nouveau **sha1sum** (Secure Hash Algorithm).

```
bash$ cksum /boot/vmlinuz
1670054224 804083 /boot/vmlinuz

bash$ echo -n "Top Secret" | cksum
3391003827 10



bash$ md5sum /boot/vmlinuz
0f43eccea8f09e0a0b2b5cf1dcf333ba  /boot/vmlinuz

bash$ echo -n "Top Secret" | md5sum
8babc97a6f62a4649716f4df8d61728f  -
```

Notez que **cksum** affiche aussi la taille, en octet, du fichier cible.

Note

La commande **cksum** affiche la taille de sa cible en octets, qu'elle soit un fichier ou `stdout`.

Les commandes **md5sum** et **sha1sum** affiche un tiret lorsqu'ils reçoivent leur entrée à partir de `stdout`.

Exemple 15.38. Vérifier l'intégrité d'un fichier

```
#!/bin/bash
# file-integrity.sh : Vérifie si les fichiers d'un répertoire donné ont été
#                     modifié.

E_REP_INEXISTANT=70
E_MAUVAIS_FICHIER_BD=71

fichierdb=File_record.md5
# Fichier pour stocker les enregistrements (fichier de base de données).


init_base_donnees ()
{
  echo ""$repertoire"" > "$fichierdb"
  # Écrit le nom du répertoire sur la première ligne du fichier.
  md5sum "$repertoire"/* >> "$fichierdb"
  # Ajoute les sommes de contrôle md5 et les noms de fichiers.
}

verifie_base_donnees ()
{
  local n=0
  local nomfichier
  local somme_controle

  # ------------------------------------------------- #
  #  Cette vérification du fichier devrait être
  #+ inutile mais il est préférable de le faire.

  if [ ! -r "$fichierdb" ]
  then
         echo "Incapable de lire les somme de contrôle du fichier de base de
données!"
    exit $E_MAUVAIS_FICHIER_BD
  fi
  # ------------------------------------------------- #

  while read enregistrement[n]
  do

    repertoire_verifie="${enregistrement[0]}"
    if [ "$repertoire_verifie" != "$repertoire" ]
    then
      echo "Les répertoires ne correspondent pas !"
      # Essayez d'utiliser un fichier d'un autre répertoire.
      exit $E_REP_INEXISTANT
    fi

    if [ "$n" -gt 0 ]   # Pas de nom de répertoire.
    then
      nomfichier[n]=$( echo ${enregistrement[$n]} | awk '{ print $2 }' )
      #  md5sum écrit les enregistrements après,
      #+ effectue en premier un contrôle des sommes, puis du fichier.
      somme_controle[n]=$( md5sum "${nomfichier[n]}" )


      if [ "${enregistrement[n]}" = "${somme_controle[n]}" ]
```

```
      then
        echo "${nomfichier[n]} non modifié."

      elif [ "`basename ${nomfichier[n]}`" != "$dbfile" ]
      #  Saute le fichier de base de données des sommes de contrôle.
      #+ car il changera à chaque appel du script.
      #  ---
      #  Ceci signifie malheureusement que lors du lancement de ce script sur
      #+ $PWD, travailler sur le fichier de base de données des sommes de
      #+ contrôle ne sera pas détecté.
      #  Exercice : Corrigez ceci.
        then
        echo "${nomfichier[n]} : ERREUR DE SOMME DE CONTRÔLE !"
        # Le fichier a été changé depuis la dernière vérification.
      fi

    fi


    let "n+=1"
  done <"$fichierdb"       #  Lit les sommes de contrôle à partir du fichier de
                           #+ base de données.

}

# ================================================= #
# main ()

if [ -z  "$1" ]
then
  repertoire="$PWD"     #  Si non spécifié,
else                    #+ utilise le répertoire courant.
  repertoire="$1"
fi

clear                   # Efface l'écran.
echo " Lancement de la vérification de l'intégrité du fichier sur $repertoire"
echo

# ------------------------------------------------------------------- #
if [ ! -r "$fichierdb" ] # Besoin de créer un fichier de base de données?
  then
          echo "Configuration de la base de données, \
            \""$repertoire"/"$fichierdb"\"."; echo
    init_base_donnees
  fi
# ------------------------------------------------------------------- #

verifie_base_donnees          # Fait le vrai travail.

echo

#  Vous pouvez souhaiter rediriger stdout vers un fichier spécialement si le
#+ répertoire vérifié a de nombreux fichiers.

exit 0

#  Pour une explication sur la vérificaton d'intégrité,
#+ considérez le paquetage
#+ http://sourceforge.net/projects/tripwire/.
```

Voir aussi l'Exemple A.20, « Informations sur un répertoire », l'Exemple 33.14, « Un jeu de « courses de chevaux » » et Exemple 9.11, « Générer « aléatoirement » une chaîne de huit caractères » pour des utilisations créatives de la commande **md5sum**.

Note

Des rapports ont indiqué que la commande **md5sum** 128 bits n'est plus sûre, donc **sha1sum** 160-bit, plus sûre, est un nouvel ajout bienvenu dans les outils de calcul de vérification.

```
bash$ md5sum fichiertest
e181e2c8720c60522c4c4c981108e367  fichiertest


bash$ sha1sum fichiertest
5d7425a9c08a66c3177f1e31286fa40986ffc996  fichiertest
```

Les consultants en sécurité ont démontré que même **sha1sum** peut être compromis. Heureusement, les dernières distributions Linux incluent des commandes dont la longueur de la clé est bien plus importante : **sha224sum**, **sha256sum**, **sha384sum** et **sha512sum**.

shred

Efface de façon sécurisé un fichier en l'écrasant (en écrivant dessus) plusieurs fois avec des octets aléatoires avant de le supprimer. Cette commande a le même effet que l'Exemple 15.59, « Effacer les fichiers de façon sûre », mais le fait de façon plus élégante et plus approfondie.

Il fait partie des utilitaires GNU *fileutils*.

Attention

Des technologies avancées peuvent toujours retrouvées le contenu d'un fichier, même après l'utilisation de **shred**.

uuencode

Cet utilitaire code des fichiers binaires (images, fichiers son, fichiers compressés, ...) en caractères ASCII, leur permettant d'être transmis dans le corps de message email ou d'être envoyé dans un groupe de nouvelles. C'est particulièrement utile quand le codage MIME (multimédia) n'est pas disponible.

uudecode

Ceci inverse le codage, décode des fichiers passés par *uuencode* et récupère les binaires originaux.

Exemple 15.39. Décoder des fichier codés avec uudecode

```
#!/bin/bash
#  Utilise uudecode sur tous les fichiers codés avec uuencode
#+ pour le répertoire actuel.

lignes=35        # Permet 35 lignes pour l'entête (très généreux).

for Fichier in *   # Teste tous les fichiers dans $PWD.
do
  recherche1=`head -n $lignes $Fichier | grep begin | wc -w`
  recherche2=`tail -n $lignes $Fichier | grep end | wc -w`
  #  Les fichiers uuencodés ont un "begin" près du début et un "end" près de
  #+ la fin.
  if [ "$recherche1" -gt 0 ]
  then
    if [ "$recherche2" -gt 0 ]
    then
      echo "uudecoding - $Fichier -"
      uudecode $Fichier
    fi
  fi
done

#  Notez que lancer ce script sur lui-même le trompe et croie qu'il est un
#+ fichier uuencodé, parce qu'il contient les mots "begin" et "end".
```

```
# Exercice:
#  Modifier ce script pour vérifier si le fichier contient un en-tête de news
#+ et pour passer au fichier suivant s'il n'en trouve pas.
exit 0
```

Astuce

La commande fold -s est utile (parfois dans un tube) pour décoder de longs messages téléchargés à partir des groupes de nouvelles Usenet.

mimencode, **mmencode**

Les commandes **mimencode** et **mmencode** s'occupent du codage des pièces-jointes des courriers éléctroniques. Bien que les *clients mail* (MUA tels que *pine* ou *kmail*) gèrent normalement ceci automatiquement, ces utilitaires particuliers permettent de manipuler de telles pièces-jointes manuellement à partir de la ligne de commande ou dans un script shell.

crypt

À un moment, il était l'utilitaire de cryptage standard sous UNIX. [6] Des régulations gouvernementales, basées sur la politique, ont interdit l'export de logiciels de cryptage, ce qui a résulté en la disparition de la commande **crypt** de la majeure partie du monde UNIX et il est toujours manquant sur la plupart des distributions Linux. Heureusement, des programmeurs ont réalisé un certain nombre d'alternatives, dont celle de l'auteur *cruft* (voir l'Exemple A.4, « *encryptedpw* : Charger un fichier sur un site ftp, en utilisant un mot de passe crypté en local »).

Divers

mktemp

Crée un *fichier temporaire* [7] avec un nom de fichier « unique ». Appelé à partir de la ligne de commandes sans arguments, il crée un fichier de longueur nulle dans le répertoire `/tmp`.

```
bash$ mktemp
/tmp/tmp.zzsvql3154
```

```
PREFIXE=nom_fichier
fichier_temporaire=`mktemp $PREFIXE.XXXXXX`
#                          ^^^^^^ A besoin d'au moins six emplacements
#+                                dans le modèle de nom de fichier.
#   Si aucun modèle de nom n'est fourni,
#+ "tmp.XXXXXXXXXX" est la valeur par défaut.
echo "nom de fichier_temporaire = $fichier_temporaire"
# nom fichier_temporaire = nom_fichier.QA2ZpY
#                ou quelque chose de similaire...

#  Crée un fichier de ce nom dans le répertoire courant avec les droits 600.
#  Un "umask 177" est, du coup, inutile
#  mais c'est néanmoins une bonne pratique de programmation.
```

make

Utilitaire pour construire et compiler des paquetages binaires. Il peut aussi être utilisé pour tout type d'opérations qui seraient déclenchées par une modification des fichiers source.

La commande **make** vérifie le `Makefile`, une liste de dépendances de fichiers et les opérations à réaliser.

L'outil *make* est, dans l'effet, un langage de scripts puissant, similaire de nombreuses façons à *Bash*, mais avec la capacité de reconnaître des *dépendances*. Pour une explication complète de cet outil puissant, voir le *site de documentation de GNU software*.

install

[6] C'est un système de chiffrement symétrique de bloc, employé pour crypter des fichiers sur un seul système ou sur un réseau local, par opposition à la classe de chiffrement publique, dont **pgp** est un exemple bien connu.
[7] Crée un *répertoire* temporaire en étant appelé avec l'option `-d`.

Commande de copie de fichier à but spécifique, similaire à **cp** mais est capable de modifier les droits et attributs des fichiers copiés. Cette commande semble faite uniquement pour l'installation de paquetages et, en tant que telle, elle fait souvent son apparition dans les `Makefiles` (dans la section *`make install :`*). Elle pourrait aussi trouver une utilité dans les scripts d'installation.

dos2unix

Cet utilitaire, écrit par Benjamin Lin et ses collaborateurs, convertit des fichiers texte au format DOS (lignes terminées par CR-LF) vers le format UNIX (lignes terminées uniquement par LF), et vice-versa.

ptx

La commande **ptx [fichier_cible]** affiche en sortie un index permuté (liste référencée) du fichier cible. Elle peut être encore filtrée et formatée dans un tube, si nécessaire.

more, **less**

Programmes envoyant un fichier texte ou un flux sur `stdout`, un écran à la fois. Ils peuvent être utilisés pour filtrer la sortie de `stdout`... ou d'un script.

Une application intéressante de **more** est de « tester » une séquence de commandes pour limiter toutes conséquences potentiellement déplaisantes.

```
ls /home/bozo | awk '{print "rm -rf " $1}' | more
#                                           ^^^^

# Tester les effets de la (désastreuse) ligne de commande suivante :
#      ls /home/bozo | awk '{print "rm -rf " $1}' | sh
#      Au shell de l'exécuter...                    ^^
```

15.6. Commandes de communications

Certaines des commandes suivantes trouvent leur utilité dans la chasse aux spammers, ainsi que dans les transferts réseaux et les analyses de données.

Informations et statistiques

host

Recherche de l'information à propos d'un hôte suivant son nom ou son adresse IP en utilisant DNS.

```
bash$ host surfacemail.com
surfacemail.com. has address 202.92.42.236
```

ipcalc

Affiche des informations IP sur un hôte. Avec l'option `-h`, **ipcalc** fait une recherche DNS inversée, trouvant le nom de l'hôte (serveur) à partir de l'adresse IP.

```
bash$ ipcalc -h 202.92.42.236
HOSTNAME=surfacemail.com
```

nslookup

Lance une « recherche sur un serveur de noms » par l'adresse IP d'un hôte. Ceci est l'équivalent de **ipcalc -h** ou **dig -x**. La commande peut être lancée interactivement ou pas, donc elle est utilisable dans un script.

La commande **nslookup** est « obsolète » mais elle a toujours son utilité.

```
bash$ nslookup -sil 66.97.104.180
nslookup kuhleersparnis.ch
 Server:         135.116.137.2
 Address:        135.116.137.2#53

 Non-authoritative answer:
 Name:   kuhleersparnis.ch
```

dig

Domain **I**nformation **G**roper. Similaire à **nslookup**, **dig** fait une « recherche Internet par un serveur de noms » sur un hôte. Peut être lancé interactivement ou non, donc il est utilisable à partir d'un script.

Voici quelques options intéressantes de **dig** : `+time=N` pour configurer un délai de *N* secondes pour obtenir la réponse, `+nofail` pour continuer à demander aux serveurs jusqu'à la réception d'une réponse et `-x` pour faire une recherche inverse.

Comparez la sortie de **dig -x** avec **ipcalc -h** et **nslookup**.

```
bash$ dig -x 81.9.6.2
;; Got answer:
 ;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11649
 ;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

 ;; QUESTION SECTION:
 ;2.6.9.81.in-addr.arpa.         IN      PTR

 ;; AUTHORITY SECTION:
 6.9.81.in-addr.arpa.    3600    IN      SOA     ns.eltel.net. noc.eltel.net.
 2002031705 900 600 86400 3600

 ;; Query time: 537 msec
 ;; SERVER: 135.116.137.2#53(135.116.137.2)
 ;; WHEN: Wed Jun 26 08:35:24 2002
 ;; MSG SIZE  rcvd: 91
```

Exemple 15.40. Trouver où dénoncer un spammeur

```
#!/bin/bash
# spam-lookup.sh : Recherche le contact pour rapporter un spammeur.
# Merci, Michael Zick.

# Vérification de l'argument en ligne de commande.
NBARGS=1
E_MAUVAISARGS=65
if [ $# -ne "$NBARGS" ]
then
  echo "Usage: `basename $0` nom_domaine"
  exit $E_MAUVAISARGS
fi


dig +short $1.contacts.abuse.net -c in -t txt
# Essayez aussi :
#     dig +nssearch $1
#     Essaie de trouver les serveurs de noms principaux
#     et affiche les enregistrements SOA.

# Ce qui suit fonctionne aussi :
#     whois -h whois.abuse.net $1
#           ^^ ^^^^^^^^^^^^^^^  Spécifiez l'hôte.
#     Peut même rechercher plusieurs spammeurs comme ceci, c'est-à-dire
#     whois -h whois.abuse.net $domainespam1 $domainespam2 . . .


#  Exercice :
#  ---------
#  Étendre la fonctionnalité de ce script
#+ pour qu'il envoie automatiquement une notification par courrier électronique
#+ au(x) adresse(s) de contact du responsable du FAI.
#  Astuce : utilisez la commande "mail".

exit $?
```

```
# spam-lookup.sh chinatietong.com
#               Un domaine connu pour le spam.

# "crnet_mgr@chinatietong.com"
# "crnet_tec@chinatietong.com"
# "postmaster@chinatietong.com"


#  Pour une version plus élaborée de ce script,
#+ voir la page de SpamViz, http://www.spamviz.net/index.html.
```

Exemple 15.41. Analyser le domaine d'un courrier indésirable

```
#! /bin/bash
# is-spammer.sh: Identifier les domaines des spams

# $Id: is-spammer.sh,v 1.7 2008-05-10 08:36:14 gleu Exp $
# L'information ci-dessus est l'ID RCS.
#
#  C'est une version simplifiée du script "is_spammer.bash"
#+ dans l'annexe des scripts contribués.

# is-spammer <nom.domaine>

# Utilise un programme externe : 'dig'
# Testé avec la version : 9.2.4rc5

# Utilise des fonctions.
# Utilise IFS pour analyser des chaînes par affectation dans des tableaux.
# Et fait même quelque chose d'utile : vérifie les listes noires d'emails.

# Utilise nom.domaine(s) à partir du corps du message :
# http://www.good_stuff.spammer.biz/just_ignore_everything_else
#
# Ou nom.domaine(s) à partir d'une adresse de courrier électronique :
# Really_Good_Offer@spammer.biz
#                   ^^^^^^^^^^^
# comme seul argument de ce script.
#(PS : votre connexion Internet doit être disponible)
#
# Donc, pour appeller ce script script dans les deux instances ci-dessus :
#       is-spammer.sh spammer.biz


# Espace blanc == :espace:tabulation:retour à la ligne:retour chariot:
WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

# Pas d'espace blanc == retour à la ligne:retour chariot
No_WSP=$'\x0A'$'\x0D'

# Séparateur de champ pour les adresses IP décimales
ADR_IFS=${No_WSP}'.'

# Obtient l'enregistrement de la ressource texte du DNS.
# recupere_txt <code_erreur> <requete>
recupere_txt() {

    # Analyse $1 par affectation sur les points.
    local -a dns
    IFS=$ADR_IFS
    dns=( $1 )
    IFS=$WSP_IFS
    if [ "${dns[0]}" == '127' ]
    then
        # Voir s'il existe une raison.
        echo $(dig +short $2 -t txt)
```

```
    fi
}

# Obtient l'enregistrement de la ressource adresse du DNS.
# verifie_adr <rev_dns> <serveur>
verifie_adr() {
    local reponse
    local serveur
    local raison

    serveur=${1}${2}
    reponse=$( dig +short ${serveur} )

    # Si reponse est un message d'erreur...
    if [ ${#reponse} -gt 6 ]
    then
        raison=$(recupere_txt ${reponse} ${serveur} )
        raison=${raison:-${reponse}}
    fi
    echo ${raison:-' ne fait pas partie de la liste noire.'}
}

# Doit obtenir l'adresse IP du nom.
echo 'Obtenir adresse de : '$1
adr_ip=$(dig +short $1)
reponse_dns=${adr_ip:-' aucune réponse '}
echo ' Adresse trouvée : '${reponse_dns}

# Une réponse valide contient au moins quatre nombres et trois points.
if [ ${#adr_ip} -gt 6 ]
then
    echo
    declare requete

    # Analyse par affectation au niveau des points.
    declare -a dns
    IFS=$ADR_IFS
    dns=( ${adr_ip} )
    IFS=$WSP_IFS

    # Réordonne les octets dans l'ordre de la requête DNS.
    rev_dns="${dns[3]}"'.'"${dns[2]}"'.'"${dns[1]}"'.'"${dns[0]}"'.'

# Voir : http://www.spamhaus.org (Conservatif, bien maintenu)
    echo -n 'spamhaus.org indique : '
    echo $(verifie_adr ${rev_dns} 'sbl-xbl.spamhaus.org')

# Voir : http://ordb.org (Relais ouverts)
    echo -n '   ordb.org  indique : '
    echo $(verifie_adr ${rev_dns} 'relays.ordb.org')

# Voir : http://www.spamcop.net/ (Vous pouvez rapporter les spammers ici)
    echo -n ' spamcop.net indique : '
    echo $(verifie_adr ${rev_dns} 'bl.spamcop.net')

# # # autres opérations de mise sur liste noire # # #

# Voir : http://cbl.abuseat.org.
    echo -n ' abuseat.org indique : '
    echo $(verifie_adr ${rev_dns} 'cbl.abuseat.org')

# Voir : http://dsbl.org/usage (Différents relais)
    echo
    echo 'Liste de serveurs de répertoires'
    echo -n '       list.dsbl.org indique : '
    echo $(verifie_adr ${rev_dns} 'list.dsbl.org')

    echo -n '   multihop.dsbl.org indique : '
    echo $(verifie_adr ${rev_dns} 'multihop.dsbl.org')
```

```
    echo -n 'unconfirmed.dsbl.org indique : '
    echo $(verifie_adr ${rev_dns} 'unconfirmed.dsbl.org')

else
    echo
    echo 'Impossible d\'utiliser cette adresse.'
fi

exit 0

# Exercices:
# --------

# 1) Vérifiez les arguments du script,
#    et quittez avec le message d'erreur approprié si nécessaire.

# 2) Vérifiez l'état de la connexion à l'appel du script,
#    et quittez avec le message d'erreur approprié si nécessaire.

# 3) Substituez des variables génériques pour les domaines BHL "codés en dur".

# 4) Initialiser le délai en utilisant l'option "+time=" pour la commande 'dig'.
```

Pour une version bien plus élaborée de ce script, voir l'Exemple A.30, « Identification d'un spammer ».

traceroute

Trace la route prise par les paquets envoyés à un hôte distant. Cette commande fonctionne à l'intérieur d'un LAN, WAN ou sur Internet. L'hôte distant peut être indiqué par son adresse IP. La sortie de cette commande peut être filtrée par grep ou sed631 via un tube.

```
bash$ traceroute 81.9.6.2
traceroute to 81.9.6.2 (81.9.6.2), 30 hops max, 38 byte packets
 1  tc43.xjbnnbrb.com (136.30.178.8)  191.303 ms  179.400 ms  179.767 ms
 2  or0.xjbnnbrb.com (136.30.178.1)  179.536 ms  179.534 ms  169.685 ms
 3  192.168.11.101 (192.168.11.101)  189.471 ms  189.556 ms *
 ...
```

ping

Envoie un paquet « ICMP ECHO_REQUEST » aux autres machines, soit sur un réseau local soit sur un réseau distant. C'est un outil de diagnostic pour tester des connections réseaux, et il devrait être utiliser avec précaution.

```
bash$ ping localhost
PING localhost.localdomain (127.0.0.1) from 127.0.0.1 : 56(84) bytes of data.
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=0 ttl=255 time=709 usec
 64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=255 time=286 usec

 --- localhost.localdomain ping statistics ---
 2 packets transmitted, 2 packets received, 0% packet loss
 round-trip min/avg/max/mdev = 0.286/0.497/0.709/0.212 ms
```

Un **ping** réussi renvoie un code de sortie44 de 0. Ceci peut être testé dans un script.

```
  HNAME=nastyspammer.com
# HNAME=$HOST     # Débogage : test pour localhost.
nombre=2  # Envoie seulement deux ping.

if [[ `ping -c $nombre "$HNAME"` ]]
then
  echo ""$HNAME" toujours présent et envoyant du SPAM chez vous."
else
  echo ""$HNAME" semble arrêté. Dommage."
fi
```

whois

Réalise une recherche DNS (*Domain Name System*, système de nom de domaine). L'option `-h` permet de spécifier sur quel serveur *whois* particulier envoyer la requête. Voir l'Exemple 4.6, « *wh*, recherche d'un nom de domaine avec *whois* » et l'Exemple 15.40, « Trouver où dénoncer un spammeur ».

finger

Retrouve de l'information sur les utilisateurs d'un réseau. Optionnellement, cette commande peut afficher les fichiers `~/.plan`, `~/.project` et `~/.forward` d'un utilisateur si un des fichiers est présent.

```
bash$ finger
Login  Name           Tty      Idle  Login Time   Office     Office Phone
 bozo   Bozo Bozeman   tty1        8  Jun 25 16:59
 bozo   Bozo Bozeman   ttyp0          Jun 25 16:59
 bozo   Bozo Bozeman   ttyp1          Jun 25 17:07



bash$ finger bozo
Login: bozo                             Name: Bozo Bozeman
 Directory: /home/bozo                   Shell: /bin/bash
 Office: 2355 Clown St., 543-1234
 On since Fri Aug 31 20:13 (MST) on tty1    1 hour 38 minutes idle
 On since Fri Aug 31 20:13 (MST) on pts/0   12 seconds idle
 On since Fri Aug 31 20:13 (MST) on pts/1
 On since Fri Aug 31 20:31 (MST) on pts/2   1 hour 16 minutes idle
 No mail.
 No Plan.
```

En plus de raisons de sécurité, un grand nombre de réseaux désactive **finger** et son démon associé. [8]

chfn

Modifie l'information découverte par la commande **finger**.

vrfy

Vérifie une adresse Internet de courrier électronique.

Cette commande semble absente sur les dernières distributions Linux.

Accès à un hôte distant

sx, **rx**

L'ensemble de commandes **sx** et **rx** sert à transférer des fichiers de et vers un hôte distant en utilisant le protocole *xmodem*. Ils font généralement partie d'un paquetage de communications, tel que **minicom**.

sz, **rz**

L'ensemble de commandes **sz** et **rz** sert à transférer des fichiers de et vers un hôte distant en utilisant le protocole *zmodem*. *Zmodem* a certains avantages sur *xmodem*, tels qu'un meilleur taux de transmission et une reprise des transferts interrompus. Comme **sx** et **rx**, ils font généralement partie d'un paquetage de communications.

ftp

Utilitaire et protocole pour envoyer / recevoir des fichiers vers ou à partir d'un hôte distant. Une session ftp peut être automatisée avec un script (voir l'Exemple 18.6, « Télécharger un ensemble de fichiers dans le répertoire de récupération *Sunsite* », l'Exemple A.4, « *encryptedpw* : Charger un fichier sur un site ftp, en utilisant un mot de passe crypté en local » et l'Exemple A.13, « *ftpget*: Télécharger des fichiers via ftp »).

uucp, **uux**, **cu**

uucp : Copie UNIX vers UNIX (*UNIX to UNIX copy*). C'est un paquetage de communication pour transférer des fichiers

[8] Un *démon* est un processus en tâche de fond non attaché à une session terminal. Les démons réalisent des services désignés soit à des moments précis soit en étant enclenchés par certains événements.

Le mot « démon » signifie fantôme en grec, et il y a certainement quelque chose de mystérieux, pratiquement surnaturel, sur la façon dont les démons UNIX travaillent silencieusement derrière la scène, réalisant leurs différentes tâches.

entre des serveurs UNIX. Un script shell est un moyen efficace de gérer une séquence de commandes **uucp**.

Depuis le développement d'Internet et du courrier électronique, **uucp** semble avoir disparu, mais il existe toujours et reste parfaitement utilisable dans des situations où des connexions Internet ne sont pas disponibles ou appropriées. L'avantage d'**uucp** est qu'il est tolérant aux pannes, donc même s'il y a une interruption de service, l'opération de copie continuera là où elle s'est arrêtée quand la connexion sera restaurée.

---

uux : *exécution d'UNIX à UNIX*. Exécute une commande sur un système distant. Cette commande fait partie du paquetage **uucp**.

---

cu : appelle (*C*all *U*p) un système distant et se connecte comme un simple terminal. C'est une version diminuée de telnet. Cette commande fait partie du paquetage **uucp**.

telnet

Utilitaire et protocole pour se connecter à un hôte distant.

Attention

Le protocole *telnet* contient des failles de sécurité et devrait donc être évité. Son utilisation dans un script shell n'est *pas* recommandée.

wget

L'utilitaire **wget** recupère de façon *non-interactive* ou télécharge des fichiers à partir d'un site Web ou d'un site ftp. Il fonctionne bien dans un script.

```
wget -p http://www.xyz23.com/file01.html
#  L'option -p ou --page-requisite fait que wget récupère tous les fichiers
#+ requis pour afficher la page spécifiée.

wget -r ftp://ftp.xyz24.net/~bozo/project_files/ -O $SAVEFILE
#  L'option -r suit récursivement et récupère tous les liens du site
#+ spécifié.

wget -c ftp://ftp.xyz25.net/bozofiles/filename.tar.bz2
#  L'option -c autorise wget à continuer un téléchargement interrompu.
#  Ceci fonctionne avec les serveurs FTP et beaucoup de sites HTTP.
```

Exemple 15.42. Obtenir la cote d'une valeur de bourse

```
#!/bin/bash
# quote-fetch.sh : Téléchargez une cote boursière.


E_SANSPARAMETRES=66

if [ -z "$1" ]  # Doit spécifier une cote boursière (symbole) à récupérer.
  then echo "Usage: `basename $0` symbole_stock"
  exit $E_SANSPARAMETRES
fi

symbole=$1

suffixe=.html
# Récupère un fichier HTML, donc nommez-le de façon approprié.
URL='http://finance.yahoo.com/q?s='
# Site finances Yahoo, avec le suffixe de la requête.

# -----------------------------------------------------------
wget -O ${symbole}${suffixe} "${URL}${symbole}"
# -----------------------------------------------------------


# Pour rechercher quelque chose sur http://search.yahoo.com :
```

```
# ------------------------------------------------------------
# URL="http://search.yahoo.com/search?fr=ush-news&p=${query}"
# wget -O "$fichier_sauvegarde" "${URL}"
# ------------------------------------------------------------
# Sauvegarde une liste d'URL en rapport.

exit $?

# Exercices :
# ----------
#
# 1) Ajoutez un test pour vous assurer que l'utilisateur ayant lancé le script
# est en ligne.
#    (Astuce : analysez la sortie de 'ps -ax' pour "ppp" ou "connect."
#
# 2) Modifiez ce script pour récupérer le rapport sur le temps local,
#+   en prenant le code postal de l'utilisateur comme argument.
```

Voir aussi l'Exemple A.32, « Rendre *wget* plus facile à utiliser » et l'Exemple A.33, « Un script de *podcasting* ».

lynx

Le navigateur web **lynx** peut être utilisé dans un script (avec l'option `-dump`) pour récupérer un fichier d'un site web ou ftp de façon non interactive.

```
lynx -dump http://www.xyz23.com/file01.html >$FICHIER
```

Avec l'option `-traversal`, **lynx** commence avec l'URL HTTP spécifiée comme argument, puis « navigue » jusqu'aux liens situés sur ce serveur particulier. Utilisée avec l'option `-crawl`, affiche le texte des pages dans un fichier de traces.

rlogin

`Connexion distante`, initie une session sur un hôte distant. Cette commande a des failles de sécurité, donc utilisez à la place ssh.

rsh

`Shell distant`, exécute des commande(s) sur un hôte distant. Il a aussi des failles de sécurité, donc utilisez à la place **ssh**.

rcp

`Copie distante`, copie des fichiers entre deux machines différentes.

rsync

`Remote synchronize` (NdT : synchronisation à distance), met à jour (synchronise) des fichiers entre deux machines différentes sur le réseau.

```
bash$ rsync -a ~/sourcedir/*txt /node1/subdirectory/
```

Exemple 15.43. Mettre à jour FC4

```
#!/bin/bash
# fc4upd.sh

# Auteur du script : Frank Wang.
# Légères modifications du style par l'auteur du guide ABS.
# Utilisé dans le guide ABS avec sa permission.


#  Télécharge les mises à jour de Fedora 4 à partir du site miroir en utilisant
rsync.
# Devrait aussi fonctionner avec les dernières Fedora Cores -- 5, 6, ...
#  Télécharge seulement le dernier package si plusieurs versions existent
#+ pour sauvegarder de l'espace.

URL=rsync://distro.ibiblio.org/fedora-linux-core/updates/
```

```
# URL=rsync://ftp.kddilabs.jp/fedora/core/updates/
# URL=rsync://rsync.planetmirror.com/fedora-linux-core/updates/

DEST=${1:-/var/www/html/fedora/updates/}
LOG=/tmp/repo-update-$(/bin/date +%Y-%m-%d).txt
PID_FILE=/var/run/${0##*/}.pid

E_RETURN=65        # Quelque chose d'inattendu est survenu.


# Options générales de rsync
# -r: téléchargement récursif
# -t: conservation des heures
# -v: verbeux

OPTS="-rtv --delete-excluded --delete-after --partial"

# modèle d'inclusion de rsync
# Le premier slash ajoute une correspondance d'un chemin absolu.
INCLUDE=(
    "/4/i386/kde-i18n-Chinese*"
#   ^                      ^
# Les guillemets sont nécessaires pour empÃªcher les remplacements.
)


# modèle d'exclusion de rsync
# Désactive temporairement les packages non voulus en utilisant "#"...
EXCLUDE=(
    /1
    /2
    /3
    /testing
    /4/SRPMS
    /4/ppc
    /4/x86_64
    /4/i386/debug
   "/4/i386/kde-i18n-*"
   "/4/i386/openoffice.org-langpack-*"
   "/4/i386/*i586.rpm"
   "/4/i386/GFS-*"
   "/4/i386/cman-*"
   "/4/i386/dlm-*"
   "/4/i386/gnbd-*"
   "/4/i386/kernel-smp*"
#  "/4/i386/kernel-xen*"
#  "/4/i386/xen-*"
)


init () {
    # La commande pipe renvoit les erreurs possibles de rsync, par exemple un réseau
saturé.
    set -o pipefail                     # Introduit dans Bash, version 3.

    TMP=${TMPDIR:-/tmp}/${0##*/}.$$     # Stocke la liste de téléchargement défini.
    trap "{
        rm -f $TMP 2>/dev/null
    }" EXIT                             # Efface le fichier temporaire en sortie.
}


check_pid () {
# Vérifie si le processus existe.
    if [ -s "$PID_FILE" ]; then
        echo "PID file exists. Checking ..."
        PID=$(/bin/egrep -o "^[[:digit:]]+" $PID_FILE)
        if /bin/ps --pid $PID &>/dev/null; then
            echo "Process $PID found. ${0##*/} seems to be running!"
```

```
           /usr/bin/logger -t ${0##*/} \
                 "Process $PID found. ${0##*/} seems to be running!"
            exit $E_RETURN
        fi
        echo "Process $PID not found. Start new process . . ."
    fi
}


#  Set overall file update range starting from root or $URL,
#+ according to above patterns.
set_range () {
    include=
    exclude=
    for p in "${INCLUDE[@]}"; do
        include="$include --include \"$p\""
    done

    for p in "${EXCLUDE[@]}"; do
        exclude="$exclude --exclude \"$p\""
    done
}


# Récupère et redéfinit la liste de mise à jour pour rsync.
get_list () {
    echo $$ > $PID_FILE || {
        echo "Can't write to pid file $PID_FILE"
        exit $E_RETURN
    }

    echo -n "Retrieving and refining update list . . ."

    # Récupère la liste -- 'eval' est nécessaire pour exécuter rsync en une seule
commande.
    # $3 et $4 sont la date et l'heure de création du fichier.
    # $5 est le nom complet du package.
    previous=
    pre_file=
    pre_date=0
    eval /bin/nice /usr/bin/rsync \
        -r $include $exclude $URL | \
        egrep '^dr.x|^-r' | \
        awk '{print $3, $4, $5}' | \
        sort -k3 | \
        { while read line; do
            # Obtient le nombre de secondes depuis epoch pour filtrer les packages
obsolètes.
            cur_date=$(date -d "$(echo $line | awk '{print $1, $2}')" +%s)
            #  echo $cur_date

            # Récupère le nom du fichier.
            cur_file=$(echo $line | awk '{print $3}')
            #  echo $cur_file

            # Récupère le nom du package RPM à partir du nom du fichier si possible.
            if [[ $cur_file == *rpm ]]; then
                pkg_name=$(echo $cur_file | sed -r -e \
                    's/(^([^_-]+[_-])+)[[:digit:]]+\..*[_-].*$/\1/')
            else
                pkg_name=
            fi
            # echo $pkg_name

            if [ -z "$pkg_name" ]; then   #  Si ce n'est pas un fichier RPM,
                echo $cur_file >> $TMP    #+ alors l'ajouter à la liste de
téléchargements.
            elif [ "$pkg_name" != "$previous" ]; then   # Un nouveau package trouvé.
                echo $pre_file >> $TMP                  # Affichage du package
```

```
précédent.
                previous=$pkg_name                      # Sauvegarde de l'actuel.
                pre_date=$cur_date
                pre_file=$cur_file
            elif [ "$cur_date" -gt "$pre_date" ]; then  #  Si mÃªme pakage en plus
récent,
                pre_date=$cur_date                      #+ alors mise à jour du
dernier pointeur.
                pre_file=$cur_file
            fi
            done
            echo $pre_file >> $TMP                      #  TMP contient TOUTE
                                                        #+ la liste redéfinie.
            # echo "subshell=$BASH_SUBSHELL"

    }       # Bracket required here to let final "echo $pre_file >> $TMP"
            # Remained in the same subshell ( 1 ) with the entire loop.

    RET=$?  # Récupère le code de retour de la commande pipe.

    [ "$RET" -ne 0 ] && {
        echo "List retrieving failed with code $RET"
        exit $E_RETURN
    }

    echo "done"; echo
}

# La vraie partie du téléchargement par rsync.
get_file () {

    echo "Downloading..."
    /bin/nice /usr/bin/rsync \
        $OPTS \
        --filter "merge,+/ $TMP" \
        --exclude '*'  \
        $URL $DEST     \
        | /usr/bin/tee $LOG

    RET=$?

        #  --filter merge,+/ is crucial for the intention.
        #  + modifier means include and / means absolute path.
        #  Then sorted list in $TMP will contain ascending dir name and
        #+ prevent the following --exclude '*' from "shortcutting the circuit."

    echo "Done"

    rm -f $PID_FILE 2>/dev/null

    return $RET
}

# --------------------
# Programme principal
init
check_pid
set_range
get_list
get_file
RET=$?
# --------------------

if [ "$RET" -eq 0 ]; then
    /usr/bin/logger -t ${0##*/} "Fedora update mirrored successfully."
else
    /usr/bin/logger -t ${0##*/} \
    "Fedora update mirrored with failure code: $RET"
fi
```

```
exit $RET
```

Voir aussi #!/bin/bash # nightly-backup.sh # http://www.richardneill.org/source.php#nightly-backup-rsync # Copyright (c) 2005 Richard Neill <backup@richardneill.org>. # Ce logiciel libre est sous licence GNU GPL. # ==> Inclus dans le guide ABS avec l'aimable autorisation de l'auteur du script. # ==> (Merci !) # Ceci réalise une sauvegarde de l'ordinateur hôte vers un disque dur firewire #+ connecté localement en utilisant rsync et ssh. # Il exécute ensuite une rotation des sauvegardes. # Exécutez-la via cron tous les jours à 5h du matin. # Cela ne sauvegarde que le répertoire principal. # Si le propriétaire (autre que l'utilisateur) doit être conservé, #+ alors exécutez le processus rsync en tant que root (et ajoutez le -o). # Nous sauvegardons tous les jours pendant sept jours, #+ puis chaque semaine pendant quatre semaines, #+ puis chaque mois pendant trois mois. # Voir http://www.mikerubel.org/computers/rsync_snapshots/ #+ pour plus d'informations sur la théorie. # À sauvegarder sous : $HOME/bin/nightly-backup_firewire-hdd.sh # Bogues connus : # ---------- # i) Idéalement, nous voulons exclure ~/.tmp et les caches du navigateur. # ii) Si l'utilisateur est devant son ordinateur à 5h du matin #+ et que les fichiers sont modifiés alors que le rsync est en cours, #+ alors la branche SAUVEGARDE_AUCASOU est appelée. # D'une certaine façon, c'est une fonctionnalité #+ mais cela cause aussi une "fuite d'espace disque". ##### DÉBUT DE LA SECTION DE CONFIGURATION ################################## UTILISATEUR_LOCAL=rjn # Utilisateur dont le répertoire principal sera #+ sauvegardé. POINT_MONTAGE=/backup # Point de montage du répertoire de sauvegarde. # Pas de slash à la fin ! # Il doit être unique #+ (par exemple en utilisant un lien symbolique udev) REP_SOURCE=/home/$UTILISATEUR_LOCAL # Pas de slash à la fin - important pour rsync. REP_DEST_SAUVE=$POINT_MONTAGE/backup/`hostname -s`.${UTILISATEUR_LOCAL}.nightly_backup ESSAI_A_BLANC=false # Si vrai, appelle rsync avec -n, réalisant un test. # Commentez ou configurez à faux pour une utilisation #+ normale. VERBEUX=false # Si vrai, rend rsync verbeux. # Commentez ou configurez à faux sinon. COMPRESSIONION=false # Si vrai, compresse. # Bon pour internet, mauvais sur LAN. # Commentez ou configurez à faux sinon. ### Codes d'erreur ### E_VAR_NON_CONF=64 E_LIGNECOMMANDE=65 E_ECHEC_MONTAGE=70 E_PASREPSOURCE=71 E_NONMONTE=72 E_SAUVE=73 ##### FIN DE LA SECTION DE CONFIGURATION ##################################### # Vérifie que toutes les variables importantes sont configurées : if [-z "$UTILISATEUR_LOCAL"] || [-z "$REP_SOURCE"] || [-z "$POINT_MONTAGE"] || [-z "$REP_DEST_SAUVE"] then echo "Une des variables n'est pas configurée ! Modifiez le fichier $0. ÉCHEC DE LA SAUVEGARDE." exit $E_VAR_NON_CONF fi if ["$#" != 0] # Si des paramètres en ligne de commande... then # Document(ation) en ligne. cat <<-FINDUTEXTE Sauvegarde quotienne automatique exécutée par cron. Lisez les sources pour plus de détails : $0 Le répertoire de sauvegarde est $REP_DEST_SAUVE . Il sera créé si nécessaire ; une initialisation est inutile. ATTENTION : le contenu de $REP_DEST_SAUVE est l'objet de rotation. Les répertoires nommés 'backup.\$i' seront éventuellement supprimés. Nous conservons des répertoires pour chaque jour sur sept jours (1-8), puis pour chaque semaine sur quatre semaines (9-12), puis pour chaque mois sur trois mois (13-15). Vous pouvez ajouter ceci à votre crontab en utilisant 'crontab -e' # Fichiers sauvegardés : $REP_SOURCE dans $REP_DEST_SAUVE #+ chaque nuit à 3:15 du matin 15 03 * * * / home/$UTILISATEUR_LOCAL/bin/nightly-backup_firewire-hdd.sh N'oubliez pas de vérifier que les sauvegardes fonctionnent, surtout si vous ne lisez pas le mail de cron !" FINDUTEXTE exit $E_LIGNECOMMANDE fi # Analyse des options. # ================== if ["$ESSAI_A_BLANC" == "true"]; then ESSAI_A_BLANC="-n" echo "ATTENTION" echo "CECI EST UN TEST SIMPLE !" echo "Aucune donnée ne sera réellement transférée !" else ESSAI_A_BLANC="" fi if ["$VERBEUX" == "true"]; then VERBEUX="-v" else VERBEUX="" fi if ["$COMPRESSION" == "true"]; then COMPRESSION="-z" else COMPRESSION="" fi # Chaque semaine (en fait tous les huit jours) et chaque mois, #+ des sauvegardes supplémentaires seront effectuées. JOUR_DU_MOIS=`date +%d` # Jour du mois (01..31). if [$JOUR_DU_MOIS = 01]; then # Premier du mois. DEBUTMOIS=true elif [$JOUR_DU_MOIS = 08 \ -o $JOUR_DU_MOIS = 16 \ -o $JOUR_DU_MOIS = 24]; then # Jour 8,16,24 # (on utilise 8 et non pas 7 pour mieux gérer les mois à 31 jours) DEBUTSEMAINE=true fi # Vérifie que le disque est monté. # En fait, vérifie que *quelque chose* est monté ici ! # Nous pouvons utiliser quelque chose d'unique sur le périphérique #+ plutôt que de simplement deviner l'ID SCSI en utilisant la bonne règle udev #+ dans /etc/udev/rules.d/10-rules.local #+ et en plaçant une entrée adéquate dans /etc/fstab. # Par exemple, cette règle udev : # BUS="scsi", KERNEL="sd*", SYSFS{vendor}="WDC WD16", # SYSFS{model}="00JB-00GVA0 ", NAME="%k", SYMLINK="lacie_1394d%n" if mount | grep $POINT_MONTAGE >/dev/null; then echo "Le point de montage $POINT_MONTAGE est déjà utilisé. OK" else echo -n "Tentative de montage de $POINT_MONTAGE..." # S'il n'est pas monté, essaie de le monter. sudo mount $POINT_MONTAGE 2>/dev/null if mount | grep $POINT_MONTAGE >/dev/null; then DEMONTE_APRES=TRUE echo "OK" # Note : s'assure qu'il sera aussi démonté #+ si nous quittons prématurément avec erreur. else echo "ÉCHEC" echo -e "Rien n'est monté sur $POINT_MONTAGE. ÉCHEC DE LA SAUVEGARDE!" exit $E_ECHEC_MONTAGE fi fi # Vérifie que le répertoire source existe et est lisible. if [! -r $REP_SOURCE] ; then echo "$REP_SOURCE n'existe pas ou ne peut être lu. ÉCHEC DE LA SAUVEGARDE." exit $E_PASREPSOURCE fi # Vérifie que la structure du répertoire de sauvegarde est bonne. # Sinon, il la crée. # Crée les sous-répertoires. # Notez que backup.0 sera créé si nécessaire par rsync. for ((i=1;i<=15;i++)); do if [! -d $REP_DEST_SAUVE/backup.$i]; then if /bin/mkdir -p $REP_DEST_SAUVE/backup.$i ; then # ^^^

$POINT_MONTAGE && echo "Démontage de $POINT_MONTAGE. Abandon." fi exit $E_NONMONTE fi fi done # Configure les droits à 700 pour de la sécurité #+ sur un système multi-utilisateur. if ! /bin/chmod 700 $REP_DEST_SAUVE ; then echo "ERREUR : n'a pas pu configurer les droits du répertoire $REP_DEST_SAUVE à 700." if ["$DEMONTE_APRES" == "TRUE"]; then # Avant de quitter, démonte le point de montage si nécessaire. cd ; sudo umount $POINT_MONTAGE && echo "Démontage de $POINT_MONTAGE. Abandon." fi exit $E_NONMONTE fi # Création du lien symbolique : current -> backup.1 si nécessaire. # Un échec ici n'est pas critique. cd $REP_DEST_SAUVE if [! -h current] ; then if ! /bin/ln -s backup.1 current ; then echo "Attention : n'a pas pu créer le lien symbolique current -> backup.1" fi fi # Maintenant, exécute le rsync. echo "Sauvegarde en cours avec rsync..." echo "Répertoire source : $REP_SOURCE" echo -e "Répertoire destination : $REP_DEST_SAUVE\n" /usr/bin/rsync $ESSAI_A_BLANC $VERBEUX -a -S --delete --modify-window=60 \ --link-dest=../backup.1 $REP_SOURCE $REP_DEST_SAUVE/backup.0/ # Avertit seulement, plutôt que de quitter, si rsync a échoué, #+ car cela pourrait n'être qu'un problème mineur. # Par exemple, si un fichier n'est pas lisible, rsync échouera. # Ceci ne doit pas empêcher la rotation. # Ne pas utiliser, par exemple, `date +%a` car ces répertoires #+ sont plein de liens et ne consomment pas *tant* d'espace. if [$? != 0]; then SAUVEGARDE_AUCASOU=backup.`date +%F_%T`.justincase echo "ATTENTION : le processus rsync n'a pas complètement réussi." echo "Quelque chose s'est mal passé. Sauvegarde d'une copie supplémentaire dans : $SAUVEGARDE_AUCASOU" echo "ATTENTION : si cela arrive fréquemment, BEAUCOUP d'espace sera utilisé," echo "même si ce ne sont que des liens !" fi # Ajoute un fichier readme dans le répertoire principal de la sauvegarde. # En sauvegarde un autre dans le sous-répertoire recent. echo "La sauvegarde de $REP_SOURCE sur `hostname` a été exécuté le \ `date`" > $REP_DEST_SAUVE/README.txt echo "Cette sauvegarde de $REP_SOURCE sur `hostname` a été créé le \ `date`" > $REP_DEST_SAUVE/backup.0/README.txt # Si nous n'avons pas fait un test, exécute une rotation des sauvegardes. [-z "$ESSAI_A_BLANC"] && # Vérifie l'espace occupé du disque de sauvegarde. # Avertissement si 90%. # Si 98% voire plus, nous échouerons probablement, donc abandon. # (Note : df peut afficher plus d'une ligne.) # Nous le testons ici plutôt qu'avant pour donner une chance à rsync. DISK_FULL_PERCENT=`/bin/df $REP_DEST_SAUVE | tr "\n" ' ' | awk '{print $12}' | grep -oE [0-9]+ ` echo "Vérification de l'espace disque sur la partition de sauvegarde \ remplie à $POINT_MONTAGE $DISK_FULL_PERCENT%." if [$DISK_FULL_PERCENT -gt 90]; then echo "Attention : le disque est rempli à plus de 90%." fi if [$DISK_FULL_PERCENT -gt 98]; then echo "Erreur : le disque est rempli complètement ! Abandon." if ["$DEMONTE_APRES" == "TRUE"]; then # Avant de quitter, démonte le point de montage si nécessaire. cd; sudo umount $POINT_MONTAGE && echo "Démontage de $POINT_MONTAGE. Abandon." fi exit $E_NONMONTE fi # Crée une sauvegarde supplémentaire. # Si cette copie échoue, abandonne. if [-n "$SAUVEGARDE_AUCASOU"]; then if ! /bin/cp -al $REP_DEST_SAUVE/backup.0 $REP_DEST_SAUVE/$SAUVEGARDE_AUCASOU then echo "ERREUR : échec lors de la création de la copie de sauvegarde \ $REP_DEST_SAUVE/$SAUVEGARDE_AUCASOU" if ["$DEMONTE_APRES" == "TRUE"]; then # Avant de quitter, démonte le point de montage si nécessaire. cd ;sudo umount $POINT_MONTAGE && echo "Démontage de $POINT_MONTAGE. Abandon." fi exit $E_NONMONTE fi fi # Au début du mois, exécute une rotation des huit plus anciens. if ["$DEBUTMOIS" == "true"]; then echo -e "\nDébut du mois. \ Suppression de l'ancienne sauvegarde : $REP_DEST_SAUVE/backup.15" && /bin/rm -rf $REP_DEST_SAUVE/backup.15 && echo "Rotation mensuelle, sauvegardes hebdomadaires : \ $REP_DEST_SAUVE/backup.[8-14] -> $REP_DEST_SAUVE/backup.[9-15]" && /bin/mv $REP_DEST_SAUVE/backup.14 $REP_DEST_SAUVE/backup.15 && /bin/mv $REP_DEST_SAUVE/backup.13 $REP_DEST_SAUVE/backup.14 && /bin/mv $REP_DEST_SAUVE/backup.12 $REP_DEST_SAUVE/backup.13 && /bin/mv $REP_DEST_SAUVE/backup.11 $REP_DEST_SAUVE/backup.12 && /bin/mv $REP_DEST_SAUVE/backup.10 $REP_DEST_SAUVE/backup.11 && /bin/mv $REP_DEST_SAUVE/backup.9 $REP_DEST_SAUVE/backup.10 && /bin/mv $REP_DEST_SAUVE/backup.8 $REP_DEST_SAUVE/backup.9 # Au début de la semaine, exécute une rotation des quatre seconds plus anciens. elif ["$DEBUTSEMAINE" == "true"]; then echo -e "\nDébut de semaine. \ Suppression de l'ancienne sauvegarde hebdomadaire : $REP_DEST_SAUVE/backup.12" && /bin/rm -rf $REP_DEST_SAUVE/backup.12 && echo "Rotation des sauvegardes hebdomadaires : \ $REP_DEST_SAUVE/backup.[8-11] -> $REP_DEST_SAUVE/backup.[9-12]" && /bin/mv $REP_DEST_SAUVE/backup.11 $REP_DEST_SAUVE/backup.12 && /bin/mv $REP_DEST_SAUVE/backup.10 $REP_DEST_SAUVE/backup.11 && /bin/mv $REP_DEST_SAUVE/backup.9 $REP_DEST_SAUVE/backup.10 && /bin/mv $REP_DEST_SAUVE/backup.8 $REP_DEST_SAUVE/backup.9 else echo -e "\nSuppression de l'ancienne sauvegarde quotidienne : $REP_DEST_SAUVE/backup.8" && /bin/rm -rf $REP_DEST_SAUVE/backup.8 fi && # Chaque jour, rotation de huit plus anciens. echo "Rotation des sauvegardes quotidiennes : \ $REP_DEST_SAUVE/backup.[1-7] -> $REP_DEST_SAUVE/backup.[2-8]" && /bin/mv $REP_DEST_SAUVE/backup.7 $REP_DEST_SAUVE/backup.8 && /bin/mv $REP_DEST_SAUVE/backup.6 $REP_DEST_SAUVE/backup.7 && /bin/mv $REP_DEST_SAUVE/backup.5 $REP_DEST_SAUVE/backup.6 && /bin/mv $REP_DEST_SAUVE/backup.4 $REP_DEST_SAUVE/backup.5 && /bin/mv $REP_DEST_SAUVE/backup.3 $REP_DEST_SAUVE/backup.4 && /bin/mv $REP_DEST_SAUVE/backup.2 $REP_DEST_SAUVE/backup.3 && /bin/mv $REP_DEST_SAUVE/backup.1 $REP_DEST_SAUVE/backup.2 && /bin/mv $REP_DEST_SAUVE/backup.0 $REP_DEST_SAUVE/backup.1 && SUCCES=true if ["$DEMONTE_APRES" == "TRUE"]; then # Démonte le point de montage s'il n'était pas monté au début. cd ; sudo umount $POINT_MONTAGE && echo "$POINT_MONTAGE de nouveau démonté." fi if ["$SUCCES" == "true"]; then echo 'SUCCÈS !' exit 0 fi # Nous devrions avoir déjà quitté si la sauvegarde a fonctionné. echo 'ÉCHEC DE LA SAUVEGARDE ! Est-ce un test ? Le disque est-il plein ?) ' exit $E_SAUVE .

Note

Utiliser rcp, rsync et d'autres outils similaires avec des implications de sécurité pourrait ne pas être judicieux. À la place, considérez l'utilisation de **ssh**, scp ou d'un script **expect**.

ssh

Shell sécurisé, pour se connecter sur un hôte distant et y exécuter des commandes. Cette alternative sécurisée pour **telnet**, **rlogin**, **rcp** et **rsh** utilise authentification et cryptage. Voir sa page man pour plus de détails.

Exemple 15.44. Utilisation de *ssh*

```
#!/bin/bash
# remote.bash: Utiliser ssh.

# Exemple de Michael Zick.
# Utilisé avec sa permission.


#   Présomptions:
#   ------------
#   fd-2 n'est pas capturé ( '2>/dev/null' ).
#   ssh/sshd présume que stderr ('2') sera affiché à l'utilisateur.
#
#   sshd est lancé sur votre machine.
#   Pour tout distribution 'standard', c'est probablement vrai,
#+  et sans qu'un ssh-keygen n'ait été effectué.

# Essayez ssh sur votre machine à partir de la ligne de commande :
#
# $ ssh $NOM_HOTE
# Sans configuration supplémentaire, un mot de passe vous sera demandé.
#   enter password
#   une fois fait,  $ exit
#
# Cela a-t'il fonctionné ? Si c'est la cas, vous êtes prêt pour plus d'action.

# Essayez ssh sur votre machine en tant que 'root' :
#
#   $  ssh -l root $NOM_HOTE
#   Lorsqu'un mot de passe vous est demandé, saisissez celui de root et surtout
#   pas le votre.
#          Last login: Tue Aug 10 20:25:49 2004 from localhost.localdomain
#   Saisissez 'exit' une fois terminé.

#  Les commandes ci-dessus vous donne un shell interactif.
#  Il est possible pour sshd d'être configuré dans le mode 'commande seule',
#+ mais cela dépasse le cadre de notre exemple.
#  La seule chose à noter est que ce qui suit fonctionnera dans le mode
#+ 'commande seule'.


# Une commande simple d'écriture sur stdout (local).

ls -l

# Maintenant la même commande basique sur une machine distante.
# Passez un nom d'utilisateur et d'hôte différents si vous le souhaitez :
USER=${NOM_UTILISATEUR:-$(whoami)}
HOST=${NOM_HOTE:-$(hostname)}

#  Maintenant, exécutez la commande ci-dessus sur l'hôte distant
#+ avec des communications cryptées.

ssh -l ${NOM_UTILISATEUR} ${NOM_HOTE} " ls -l "

#  Le résultat attendu est une liste du contenu du répertoire personnel de
#  l'utilisateur sur la machine distante.
```

```
#  Pour voir les différences, lancez ce script à partir d'un autre endroit que
#+ votre répertoire personnel.

#  En d'autres termes, la commande Bash est passée comme une ligne entre guillemets
#+ au shell distant, qui l'exécute sur la machine distante.
#  Dans ce cas, sshd fait  ' bash -c "ls -l" '   à votre place.

#  Pour des informations sur des thèmes comme ne pas avoir à saisir un mot de
#  passe pour chaque ligne de commande, voir
#+    man ssh
#+    man ssh-keygen
#+    man sshd_config.

exit 0
```

Attention

À l'intérieur d'une boucle, **ssh** pourrait avoir un comportement inattendu. D'après un *message Usenet* de l'archive comp.unix shell, **ssh** hérite de l'entrée standard (`stdin`) de la boucle. Pour remédier à ceci, passez à **ssh** l'option `-n` ou l'option `-f`.

Merci à Jason Bechtel pour cette indication.

scp

`Secure copy`, similaire en fonction à **rcp**, copie des fichiers entre deux machines différentes sur le réseau mais le fait en utilisant une authentification et avec un niveau de sécurité similaire à **ssh**.

Réseaux locaux

write

Utilitaire pour la communication terminal à terminal. Il permet d'envoyer des lignes à partir de votre terminal (console ou *xterm*) à un autre utilisateur. La commande mesg pourrait, bien sûr, être utilisée pour désactiver l'accès en écriture au terminal.

Comme **write** est interactif, il a peu de chances de prouver son utilité dans un script.

netconfig

Un outil en ligne de commande pour configurer un adaptateur réseau (en utilisant *DHCP*). Cette commande est native pour les distributions Linux basées sur la Red Hat.

Mail

mail

Envoie ou lit des courriers électroniques.

Ce client mail en ligne de commande est très simpliste et fonctionne bien comme commande embarquée dans un script.

Exemple 15.45. Un script qui envoie son fichier source

```
#!/bin/sh
# self-mailer.sh: Script vous envoyant un mail.

adr=${1:-`whoami`}     # Par défaut, l'utilisateur courant, si non spécifié.
#  Tapez 'self-mailer.sh wiseguy@superdupergenius.com'
#+ envoie ce script à cette adresse.
#  Tapez juste 'self-mailer.sh' (sans argument) envoie le script à la personne
#+ l'ayant appelé, par exemple bozo@localhost.localdomain.
#
#  Pour plus d'informations sur la construction ${parameter:-default},
#+ voir la section "Substitution de paramètres" du chapitre "Variables
#+ Revisitées."
```

```
# ============================================================================
  cat $0 | mail -s "Le script \"`basename $0`\" s'est envoyé lui-même à vous."
"$adr"
# ============================================================================

# ----------------------------------------
#  Bonjour du script qui s'envoie par mail.
#  Une personne mal intentionnée a lancé ce script, ce qui a fait que ce mail
#+ vous a été envoyé.
#  Apparemment, certaines personnes n'ont rien de mieux à faire de leur temps.
# ----------------------------------------

echo "Le `date`, le script \"`basename $0`\" vous a été envoyé par mail sur "$adr"."

exit 0

#  Notez que la commande "mailx" (en mode "send") pourrait être substituée
#+ par "mail"... mais avec des options un peu différentes.
```

mailto

Similaire à la commande **mail**, **mailto** envoie des mails à partir de la ligne de commande ou dans un script. Néanmoins, **mailto** permet aussi d'envoyer des messages MIME (multimedia).

vacation

Cet utilitaire répond automatiquement aux courriers électroniques que le destinataire est en vacances et temporairement indisponible. Ceci tourne sur le réseau, en conjonction avec **sendmail**, et n'est pas applicable à un compte POP.

15.7. Commandes de contrôle du terminal

Commandes modifiant l'état de la console ou du terminal

tput

Initialise et/ou recherche des informations relatives à un terminal depuis les données `terminfo`. Certaines options permettent différentes manipulations du terminal. **tput clear** est l'équivalent de **clear**, cité plus haut. **tput reset** est l'équivalent de reset, cité plus haut **tput sgr0** réinitialise aussi le terminal mais ne vide pas l'écran.

```
bash$ tput longname
 xterm terminal emulator (XFree86 4.0 Window System)
```

La commande **tput cup X Y** bouge le curseur à la position (X,Y) sur le terminal actuel. **clear** la précède généralement pour effacer l'écran.

Notez que stty offre un jeu de commandes plus conséquent pour le contrôle des terminaux.

infocmp

Cette commande affiche des informations étendues sur le terminal actuel. Il fait référence à la base de données *terminfo*.

```
bash$ infocmp
 #       Reconstructed via infocmp from file:
 /usr/share/terminfo/r/rxvt
 rxvt|rxvt terminal emulator (X Window System),
 am, bce, eo, km, mir, msgr, xenl, xon,
 colors#8, cols#80, it#8, lines#24, pairs#64,
 acsc=``aaffggjjkkllmmnnooppqqrrssttuuvvwwxxyyzz{{||}}~~,
 bel=^G, blink=\E[5m, bold=\E[1m,
 civis=\E[?25l,
 clear=\E[H\E[2J, cnorm=\E[?25h, cr=^M,
 ...
```

reset

Réinitialise les paramètres du terminal et efface son contenu. Comme avec la commande **clear**, le curseur réapparaît dans le coin supérieur gauche de l'écran.

clear

La commande **clear** efface simplement le contenu textuel d'une console ou d'un *xterm*. Le curseur de l'invite réapparaît dans le coin supérieur gauche du terminal. Cette commande peut être utilisée en ligne de commande ou dans un script. Voir l'Exemple 10.25, « Créer des menus en utilisant *case* ».

script

Cet utilitaire sauve dans un fichier toutes les saisies clavier saisies dans le terminal par l'utilisateur. En fait, cela crée un enregistrement de la session.

15.8. Commandes mathématiques

« Compter »

factor

Décompose un entier en nombre premiers.

```
bash$ factor 27417
 27417: 3 13 19 37
```

bc

Bash ne peut traiter les calculs en virgule flottante et n'intègre pas certaines fonctions mathématiques importantes. Heureusement, **bc** est là pour nous sauver.

bc n'est pas simplement une calculatrice souple à précision arbitraire, elle offre aussi beaucoup de facilités disponibles habituellement dans un langage de programmation.

La syntaxe de **bc** ressemble vaguement à celle du C.

bc est devenu un outil UNIX assez puissant pour être utilisé via un tube et est manipulable dans des scripts.

Ceci est un simple exemple utilisant **bc** pour calculer la valeur d'une variable. Il utilise la substitution de commande141.

```
variable=$(echo "OPTIONS; OPERATIONS" | bc)
```

Exemple 15.46. Paiement mensuel sur une hypothèque

```
#!/bin/bash
# monthlypmt.sh : Calcule le paiement mensuel d'une hypothèque.


#  C'est une modification du code du paquetage "mcalc" (mortgage calculator,
#+ c'est-à-dire calcul d'hypothèque), de Jeff Schmidt et Mendel Cooper
#+ (l'auteur du guide ABS).
#   http://www.ibiblio.org/pub/Linux/apps/financial/mcalc-1.6.tar.gz  [15k]

echo
echo "Étant donné le montant principal, le taux d'intérêt et la fin de
l'hypothèque,"
echo "calcule le paiement mensuel."

bas=1.0

echo
echo -n "Entrez le montant principal (sans virgule) "
read principal
```

```
echo -n "Entrez le taux d'intérêt (pourcentage) "
  # Si 12%, entrez "12" et non pas ".12".
read taux_interet
echo -n "Entrez le nombre de mois "
read nb_mois


 taux_interet=$(echo "scale=9; $taux_interet/100.0" | bc) # Convertit en décimal
                #             ^^^^^^^^^^^^^^^^^^^ Diviser par 100.
                # "scale" détermine le nombre de décimales.

 taux_interet_tmp=$(echo "scale=9; $taux_interet/12 + 1.0" | bc)


 top=$(echo "scale=9; $principal*$taux_interet_tmp^$nb_mois" | bc)
                #    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                #    Formule standard pour calculer un intérêt.

 echo; echo "Merci d'être patient. Ceci peut prendre longtemps."

 let "mois = $nb_mois - 1"
# ================================================================
 for ((x=$mois; x > 0; x--))
 do
   bot=$(echo "scale=9; $taux_interet_tmp^$x" | bc)
   bas=$(echo "scale=9; $bas+$bot" | bc)
#  bas = $(($bas + $bot"))
 done
# ================================================================

# ----------------------------------------------------------------
#  Rick Boivie indique une implémentation plus efficace que la boucle
#+ ci-dessus, ce qui réduit le temps de calcul de 2/3.

# for ((x=1; x <= $mois; x++))
# do
#   bas=$(echo "scale=9; $bas * $taux_interet_tmp + 1" | bc)
# done


#  Puis, il est revenu avec une alternative encore plus efficace,
#+ car elle descend le temps d'exécution de 95%!

# bas=`{
#     echo "scale=9; bas=$bas; taux_interet_tmp=$taux_interet_tmp"
#     for ((x=1; x <= $mois; x++))
#     do
#          echo 'bas = bas * taux_interet_tmp + 1'
#     done
#     echo 'bas'
#     } | bc`       # Intègre une 'boucle for' dans la substitution de commande.

# ----------------------------------------------------------------------
#  D'un autre côté, Frank Wang suggère :
#  bottom=$(echo "scale=9; ($interest_rate^$term-1)/($interest_rate-1)" | bc)

#  Car
#  L'algorithme de la boucle est une somme de série géométrique de proportion.
#  La formule de la somme est e0(1-q^n)/(1-q),
#+ où e0 est le premier élément et q=e(n+1)/e(n)
#+ et n est le nombre d'éléments.
# ----------------------------------------------------------------------

 # let "paiement = $top/$bas"
 paiement=$(echo "scale=2; $top/$bas" | bc)
 # Utilise deux décimales pour les dollars et les cents.

 echo
 echo "paiement mensuel = \$$paiement"
```

```
    # Affiche un signe dollar devant le montant.
 echo


 exit 0

 #  Exercices :
 #    1) Filtrez l'entrée pour permettre la saisie de virgule dans le montant.
 #    2) Filtrez l'entrée pour permettre la saisie du taux d'intérêt en
 #+      pourcentage ou en décimale.
 #    3) Si vous êtes vraiment ambitieux, étendez ce script pour afficher
 #+      les tables d'amortissement complètes.
```

Exemple 15.47. Conversion de base

```
#!/bin/bash
################################################################################
# Script shell: base.sh - affiche un nombre en différentes bases (Bourne Shell)
# Auteur      : Heiner Steven (heiner.steven@odn.de)
# Date        : 07-03-95
# Catégorie   : Desktop
# $Id: base.sh,v 1.10 2008-05-10 08:36:11 gleu Exp $
# ==> La ligne ci-dessus est l'information ID de RCS.
################################################################################
# Description
#
# Modifications
# 21-03-95 stv correction d'une erreur arrivant avec 0xb comme entrée (0.2)
################################################################################

# ==> Utilisé dans le guide ABS avec la permission de l'auteur du script.
# ==> Commentaires ajoutés par l'auteur du guide ABS.

NOARGS=65
PN=`basename "$0"`                              # Nom du programme
VER=`echo '$Revision: 1.10 $' | cut -d' ' -f2`  # ==> VER=1.6

Usage () {
    echo "$PN - Affiche un nombre en différentes bases, $VER (stv '95)
usage: $PN [nombre ...]

Si aucun nombre n'est donné, les nombres sont lus depuis l'entrée standard.
Un nombre peut être
    binaire (base 2)         commençant avec 0b (i.e. 0b1100)
    octal (base 8)         commençant avec 0  (i.e. 014)
    hexadécimal (base 16) commençant avec 0x (i.e. 0xc)
    décimal                 autrement (c'est-à-dire 12)" >&2
    exit $NOARGS
}   # ==> Fonction pour afficher le message d'usage.

Msg () {
    for i   # ==> [liste] manquante.
    do echo "$PN: $i" >&2
    done
}

Fatal () { Msg "$@"; exit 66; }

AfficheBases () {
    # Détermine la base du nombre
    for i      # ==> [liste] manquante...
    do         # ==> donc opère avec le(s) argument(s) en ligne de commande.
        case "$i" in
            0b*)         ibase=2;; # binaire
            0x*|[a-f]*|[A-F]*) ibase=16;; # hexadécimal
            0*)                ibase=8;; # octal
```

```
            [1-9]*)         ibase=10;; # décimal
            *)
                Msg "nombre illégal $i - ignoré"
                continue;;
        esac

        #  Suppression du préfixe, conversion des nombres hexadécimaux en
        #+ majuscule (bc a besoin de cela)
        number=`echo "$i" | sed -e 's:^0[bBxX]::' | tr '[a-f]' '[A-F]'`
        # ==> Utilise ":" comme séparateur sed, plutôt que "/".

        # Conversion des nombres en décimal
        dec=`echo "ibase=$ibase; $number" | bc`  # ==> 'bc' est un utilitaire de
                                                 #+ calcul.
        case "$dec" in
            [0-9]*) ;;                   # nombre ok
            *)         continue;;           # erreur: ignore
        esac

        # Affiche toutes les conversions sur une ligne.
        # ==> le 'document en ligne' remplit la liste de commandes de 'bc'.
        echo `bc <<!
            obase=16; "hex="; $dec
            obase=10; "dec="; $dec
            obase=8;  "oct="; $dec
            obase=2;  "bin="; $dec
!
    ` | sed -e 's: : :g'

    done
}

while [ $# -gt 0 ]
# ==>  est une "boucle while" réellement nécessaire
# ==>+ car tous les cas soit sortent de la boucle
# ==>+ soit terminent le script.
# ==> (commentaire de Paulo Marcel Coelho Aragao.)
do
    case "$1" in
        --) shift; break;;
        -h) Usage;;                  # ==> Message d'aide.
        -*) Usage;;
        *) break;;                  # premier nombre
    esac   #  ==> Plus de vérification d'erreur pour des entrées illégales
           #+ serait utile.
    shift
done

if [ $# -gt 0 ]
then
    AfficheBases "$@"
else                                 #  lit à partir de l'entrée standard
                                        #+ stdin
    while read ligne
    do
        PrintBases $ligne
    done
fi

exit 0
```

Une autre façon d'utiliser **bc** est d'utiliser des documents en ligne318 embarqués dans un bloc de substitution de commandes141. Ceci est très intéressant lorsque le script passe un grand nombre d'options et de commandes à **bc**

```
variable=`bc >> CHAINE_LIMITE
 options
 instructions
 operations
```

```
CHAINE_LIMITE
`


...or...


variable=$(bc >> CHAINE_LIMITE
  options
  instructions
  operations
  CHAINE_LIMITE
  )
```

Exemple 15.48. Appeler *bc* en utilisant un *document en ligne*

```
#!/bin/bash
# Appelle 'bc' en utilisant la substitution de commandes
# en combinaison avec un 'document en ligne'.


var1=`bc << EOF
18.33 * 19.78
EOF
`
echo $var1       # 362.56


# La notation $( ... ) fonctionne aussi.
v1=23.53
v2=17.881
v3=83.501
v4=171.63

var2=$(bc << EOF
scale = 4
a = ( $v1 + $v2 )
b = ( $v3 * $v4 )
a * b + 15.35
EOF
)
echo $var2       # 593487.8452


var3=$(bc -l << EOF
scale = 9
s ( 1.7 )
EOF
)
# Renvoie le sinus de 1,7 radians.
# L'option "-l" appelle la bibliothèque mathématique de 'bc'.
echo $var3       # .991664810


# Maintenant, essayez-la dans une fonction...
hypotenuse ()    # Calculez l'hypoténuse d'un triangle à angle droit.
{                # c = sqrt( a^2 + b^2 )
hyp=$(bc -l << EOF
scale = 9
sqrt ( $1 * $1 + $2 * $2 )
EOF
)
# Can't directly return floating point values from a Bash function.
# But, can echo-and-capture:
echo "$hyp"
}

hyp=$(hypotenuse 3.68 7.31)
echo "hypoténuse = $hyp"    # 8.184039344
```

```
exit 0
```

Exemple 15.49. Calculer PI

```
#!/bin/bash
# cannon.sh: Approximation de PI en tirant des balles de canon.

# C'est une très simple instance de la simulation "Monte Carlo" : un modèle
#+ mathématique d'un événement réel, en utilisant des nombres pseudo-aléatoires
#+ pour émuler la chance.

#  Considérez un terrain parfaitement carré, de 10000 unités par côté.
#  Ce terrain comprend un lac parfaitement circulaire en son centre d'un
#+ diamètre de 10000 unités.
#  Ce terrain ne comprend pratiquement que de l'eau mais aussi un peu de
#+ terre dans ses quatre coins.
#  (pensez-y comme un carré comprenant un cercle.)
#
#  Nous tirons des balles de canon à partir d'un vieux canon situé sur un des côtés
#+ du terrain.
#  Tous les tirs créent des impacts quelque part sur le carré, soit dans le
#+ lac soit dans un des coins secs.
#  Comme le lac prend la majorité de l'espace disponible, la
#+ plupart des tirs va tomber dans l'eau.
#  Seuls quelques tirs tomberont sur un sol rigide compris dans les quatre coins
#+ du carré.
#
#  Si nous prenons assez de tirs non visés et au hasard,
#+ alors le ratio des coups dans l'eau par rapport au nombre total sera
#+ approximativement de PI/4.
#
#  La raison de ceci est que le canon ne tire réellement que dans la partie
#+ haute à droite du carré, premier quadrant des coordonnées cartésiennes.
#  (La précédente explication était une simplification.)
#
#  Théoriquement, plus de tirs sont réalisés, plus cela correspondra.
#  Néanmoins, un script shell, contrairement à un langage compilé avec un
#+ support des calculs à virgule flottante, nécessite quelques compromis.
#  Ceci tend à rendre la simulation moins précise.


DIMENSION=10000  # Longueur de chaque côté.
                 # Initialise aussi le nombre d'entiers générés au hasard.

NB_TIRS_MAX=1000 # Tire ce nombre de fois.
                 # 10000 ou plus serait mieux mais prendrait bien plus de temps.
PMULTIPLIEUR=4.0 # Facteur d'échelle pour l'approximation de PI.

M_PI=3.141592654 # Valeur réelle de PI, dans un but de comparaison.

au_hasard ()
{
RECHERCHE=$(head -n 1 /dev/urandom | od -N 1 | awk '{ print $2 }')
HASARD=$RECHERCHE                    # Du script d'exemple "seeding-random.sh"
let "rnum = $HASARD % $DIMENSION"    # Echelle plus petite que 10000.
echo $rnum
}

distance=        # Déclaration de la variable globale.
hypotenuse ()    # Calcule de l'hypoténuse d'un triangle à angle droit.
{                # A partir de l'exemple "alt-bc.sh".
distance=$(bc -l << EOF
scale = 0
sqrt ( $1 * $1 + $2 * $2 )
```

```
EOF
)
#  Initiale "scale" à zéro fait que le résultat sera une valeur entière, un
#+ compris nécessaire dans ce script.
#  Ceci diminue l'exactitude de la simulation.
}


# main() {

# Initialisation des variables.
tirs=0
dans_l_eau=0
sur_terre=0
Pi=0
error=0

while [ "$tirs" -lt  "$NB_TIRS_MAX" ]        # Boucle principale.
do

  xCoord=$(au_hasard)                         # Obtenir les coordonnées X et Y au
                                              # hasard.
  yCoord=$(au_hasard)
  hypotenuse $xCoord $yCoord                  #  Hypoténuse du triangle rectangle =
                                              #+ distance.
  ((tirs++))

  printf "#%4d   " $tirs
  printf "Xc = %4d  " $xCoord
  printf "Yc = %4d  " $yCoord
  printf "Distance = %5d  " $distance          #  Distance à partir du centre
                                               #+ du lac --
                                               #  l'"origine" --
                                               #+ coordonnées (0,0).

  if [ "$distance" -le "$DIMENSION" ]
  then
    echo -n "Dans l'eau !  "
    ((dans_l_eau++))
  else
    echo -n "Sur terre !   "
    ((sur_terre++))
  fi

  Pi=$(echo "scale=9; $PMULTIPLIEUR*$dans_l_eau/$tirs" | bc)
  # Multipliez le ratio par 4.0.
  echo -n "PI ~ $Pi"
  echo

done

echo
echo "Après $tirs tirs, PI ressemble approximativement à $Pi"
# Tend à être supérieur.
# Probablement dû aux erreurs d'arrondi et au hasard perfectible de $RANDOM.
error=$(echo "scale=9; $Pi - $M_PI" | bc)
echo "Deviation de la valeur mathématique de PI =      $error"
echo

# }

exit 0

#  On peut se demander si un script shell est approprié pour une application
#+ aussi complexe et aussi intensive en calcul.
#
#  Il existe au moins deux justifications.
#  1) La preuve du concept: pour montrer que cela est possible.
#  2) Pour réaliser un prototype et tester les algorithmes avant de le réécrire
```

```
#+    dans un langage compilé de haut niveau.
```

dc

L'utilitaire **dc** (**d**esk **c**alculator) utilise l'empilement et la « notation polonaise inversée » (RPN). Comme **bc**, il possède les bases d'un langage de programmation.

La plupart des gens évitent **dc**, parce qu'il nécessite de saisir les entrées en RPN, ce qui n'est pas très intuitif. Toutefois, cette commande garde son utilité.

Exemple 15.50. Convertir une valeur décimale en hexadécimal

```
#!/bin/bash
# hexconvert.sh : Convertit un nombre décimal en hexadécimal.

E_SANSARGS=65 # Arguments manquants sur la ligne de commande.
BASE=16       # Hexadécimal.

if [ -z "$1" ]
then
  echo "Usage: $0 nombre"
  exit $E_SANSARGS
  # A besoin d'un argument en ligne de commande.
fi
# Exercice : ajouter une vérification de la validité de l'argument.


hexcvt ()
{
if [ -z "$1" ]
then
  echo 0
  return    # "Renvoie" 0 si aucun argument n'est passé à la fonction.
fi

echo ""$1" "$BASE" o p" | dc
#                   "o" demande une sortie en base numérique.
#                   "p" Affiche le haut de la pile.
# Voir 'man dc' pour plus d'options.
return
}

hexcvt "$1"

exit 0
```

L'étude de la page *info* de la commande **dc** est un moyen pénible de prendre conscience de sa complexité. Il semble cependant qu'une poignée de *connaisseurs de dc* se délectent de pouvoir exiber leur maîtrise de cet outil puissant mais mystérieux.

```
bash$ echo "16i[q]sa[ln0=aln100%Pln100/snlbx]sbA0D68736142snlbxq" | dc"
Bash
```

Exemple 15.51. Factorisation

```
#!/bin/bash
# factr.sh : Factorise un nombre

MIN=2       # Ne fonctionnera pas pour des nombres plus petits que celui-ci.
E_SANSARGS=65
E_TROPPETIT=66

if [ -z $1 ]
then
```

```
  echo "Usage: $0 nombre"
  exit $E_SANSARGS
fi

if [ "$1" -lt "$MIN" ]
then
  echo "Le nombre à factoriser doit être supérieur ou égal à $MIN."
  exit $E_TROPPETIT
fi

#  Exercice : Ajouter une vérification du type (pour rejeter les arguments non
#+ entiers).

echo "Les facteurs de $1 :"
# -------------------------------------------------------------------------------
echo "$1[p]s2[lip/dli%0=1dvsr]s12sid2%0=13sidvsr[dli%0=1lrli2+dsi!>.]ds.xd1<2"|dc
# -------------------------------------------------------------------------------
# La ligne de code ci-dessus a été écrite par Michel Charpentier
&lt;charpov@cs.unh.edu&gt;.
# Utilisé dans le guide ABS avec sa permission (merci).

 exit 0
```

awk

Une autre façon d'utiliser les nombres à virgule flottante est l'utilisation des fonctions internes de la commande awk634 dans un emballage shell434 .

Exemple 15.52. Calculer l'hypoténuse d'un triangle

```
#!/bin/bash
# hypotenuse.sh : Renvoie l'"hypoténuse" d'un triangle à angle droit,
#                 (racine carrée de la somme des carrés des côtés)

ARGS=2                # Le script a besoin des côtés du triangle.
E_MAUVAISARGS=65          # Mauvais nombre d'arguments.

if [ $# -ne "$ARGS" ] # Teste le nombre d'arguments du script.
then
  echo "Usage: `basename $0` cote_1 cote_2"
  exit $E_MAUVAISARGS
fi


SCRIPTAWK=' { printf( "%3.7f\n", sqrt($1*$1 + $2*$2) ) } '
#            commande(s) / paramètres passés à awk


# Maintenant, envoyez les paramètres à awk via un tube.
    echo -n "Hypoténuse de $1 et $2 = "
    echo $1 $2 | awk "$SCRIPTAWK"
#   ^^^^^^^^^^^^
# echo et pipe sont un moyen facile de passer des paramètres shell.

exit 0

# Exercice : Ré-écrivez ce script en utilisant 'bc' à la place de awk.
#            Quelle méthode est la plus intuitive ?
```

15.9. Commandes diverses

Commandes qui ne peuvent être classées

jot, **seq**

Ces outils génèrent des séquences de nombres entiers avec une incrémentation que l'utilisateur choisit.

Le retour à la ligne qui sépare par défaut les entiers peut être modifié avec l'option `-s`.

```
bash$ seq 5
1
2
3
4
5



bash$ seq -s : 5
1:2:3:4:5
```

jot et **seq** sont fort pratiques pour les boucles.

Exemple 15.53. Utiliser *seq* pour générer l'incrément d'une boucle

```
#!/bin/bash
# Utiliser "seq"

echo

for a in `seq 80`  # ou   for a in $( seq 80 )
# Identique à   for a in 1 2 3 4 5 ... 80   (évite beaucoup de frappe !).
# Pourrait aussi utiliser 'jot' (si présent sur le système).
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80
# Exemple d'utilisation de la sortie d'une commande pour générer la [liste]
# dans une boucle "for".

echo; echo


COMPTEUR=80  # Oui, 'seq' accepte aussi un compteur remplaçable.

for a in `seq $COMPTEUR`  # ou   for a in $( seq $COMPTEUR )
do
  echo -n "$a "
done      # 1 2 3 4 5 ... 80

echo; echo

DEBUT=75
FIN=80

for a in `seq $DEBUT $FIN`
#  Donner à "seq" deux arguments permet de commencer le comptage au premier et
#+ de le terminer au second.
do
  echo -n "$a "
done      # 75 76 77 78 79 80

echo; echo

DEBUT=45
INTERVALLE=5
FIN=80

for a in `seq $DEBUT $INTERVALLE $FIN`
#  Donner à "seq" trois arguments permet de commencer le comptage au premier,
#+ d'utiliser le deuxième comme intervalle et de le terminer au troisième.
do
```

```
  echo -n "$a "
done      # 45 50 55 60 65 70 75 80

echo; echo

exit 0
```

Un exemple plus simple :

```
#  Crée un ensemble de dix fichiers,
#+ nommés fichier.1, fichier.2 . . . fichier.10.
NOMBRE=10
PREFIXE=fichier

for fichier in `seq $NOMBRE`
do
  touch $PREFIXE.$fichier
  #  Ou vous pouvez réaliser d'autres opérations,
  #+ comme rm, grep, etc.
done
```

Exemple 15.54. Compteur de lettres

```
#!/bin/bash
# letter-count.sh : Compte les occurrences de lettres dans un fichier texte.
# Écrit par Stefano Palmeri.
# Utilisé dans le guide ABS avec sa permission.
# Légèrement modifié et reformaté par l'auteur du guide ABS.

MINARGS=2          # Le script requiert au moins deux arguments.
E_MAUVAISARGS=65
FICHIER=$1

let LETTRES=$#-1   # Nombre de lettres spécifiées (comme argument en ligne de
commande).
                   # (Soustrait 1 du nombre d'arguments en ligne de commande.)


affiche_aide(){
           echo
           echo Usage: `basename $0` fichier LETTRES
           echo Note: les arguments de `basename $0` sont sensibles à la casse.
           echo Exemple: `basename $0` foobar.txt G n U L i N U x.
           echo
}

# Vérification du nombre d'arguments.
if [ $# -lt $MINARGS ]; then
   echo
   echo "Pas assez d'arguments."
   echo
   affiche_aide
   exit $E_MAUVAISARGS
fi


# Vérifie si le fichier existe.
if [ ! -f $FICHIER ]; then
    echo "Le fichier \"$FICHIER\" n'existe pas."
    exit $E_MAUVAISARGS
fi



# Compte l'occurence des lettres.
for n in `seq $LETTRES`; do
      shift
```

```
      if [[ `echo -n "$1" | wc -c` -eq 1 ]]; then               #  Vérifie
l'argument.
            echo "$1" -\> `cat $FICHIER | tr -cd  "$1" | wc -c` #  Compte.
      else
            echo "$1 n'est pas un seul caractère."
      fi
done

exit $?

#  Ce script a exactement les mêmes fonctionnalités que letter-count2.sh
#+ mais s'exécute plus rapidement.
#  Pourquoi ?
```

Note

Avec plus de fonctionnalités que *seq*, **jot** est un outil classique UNIX qui n'est pas habituellement inclus dans une distribution Linux. Néanmoins, le source *rpm* est disponible au téléchargement sur le *dépôt du MIT*.

Contrairement à *seq*, **jot** peut générer une séquence de nombres aléatoires en utilisant l'option `-r`.

```
bash$ jot -r 3 999
1069
 1272
 1428
```

getopt

La commande **getopt** analyse les options de la ligne de commande précédées par un tiret. Cette commande externe correspond à la commande intégrée Bash getopts. Utiliser **getopt** permet la gestion des options longues grâce à l'utilisation de l'option `-l` et cela permet aussi la réorganisation des paramètres.

Exemple 15.55. Utiliser *getopt* pour analyser les paramètres de la ligne de commande

```
#!/bin/bash
# Utiliser getopt.

# Essayez ce qui suit lors de l'appel à ce script.
#   sh ex33a.sh -a
#   sh ex33a.sh -abc
#   sh ex33a.sh -a -b -c
#   sh ex33a.sh -d
#   sh ex33a.sh -dXYZ
#   sh ex33a.sh -d XYZ
#   sh ex33a.sh -abcd
#   sh ex33a.sh -abcdZ
#   sh ex33a.sh -z
#   sh ex33a.sh a
# Expliquez les résultats de chacun.

E_OPTERR=65

if [ "$#" -eq 0 ]
then   # Le script a besoin d'au moins un argument en ligne de commande.
  echo "Usage $0 -[options a,b,c]"
  exit $E_OPTERR
fi

set -- `getopt "abcd:" "$@"`
#  Positionne les paramètres de position par rapport aux arguments en ligne de
#+ commandes.
# Qu'arrive-t'il si vous utilisez "$*" au lieu de "$@" ?

while [ ! -z "$1" ]
do
```

```
  case "$1" in
    -a) echo "Option \"a\"";;
    -b) echo "Option \"b\"";;
    -c) echo "Option \"c\"";;
    -d) echo "Option \"d\" $2";;
     *) break;;
  esac

  shift
done

#  Il est généralement mieux d'utiliser la commande intégrée 'getopts'
#+ dans un script.
#  Voir "ex33.sh".

exit 0
```

Voir l'Exemple 9.14, « Émuler *getopt* » pour une émulation simplifiée de **getopt**.

run-parts

La commande **run-parts** [9] exécute tous les scripts d'un répertoire cible triés par ordre ASCII. Évidemment, ces scripts nécessitent les droits d'exécution.

Le démon cron lance **run-parts** pour exécuter les scripts du répertoire `/etc/cron.*`.

yes

Par défaut, la commande **yes** envoie une suite infinie de lettres `y` suivies de retours à la ligne sur `stdout`. Un **ctrl+c** arrête l'éxécution. Une chaîne différente peut être spécifiée en argument (**`yes chaine_differente`** affichera continuellement `chaine_differente` sur `stdout`).

On pourrait se demander l'intérêt de la chose. En pratique, **yes** peut être utilisé comme un **expect** minimaliste en étant redirigé vers un programme en attente d'une saisie **expect**.

`yes | fsck /dev/hda1` confirme toutes les réparations à **fsck** (méfiance !).

`yes | rm -r nom_repertoire` aura le même effet que **`rm -rf nom_repertoire`** (toujours méfiance !).

Avertissement

La plus grande prudence est conseillée lorsque vous redirigez **yes** vers une commande potentiellement dangereuse pour votre système, comme fsck ou fdisk. Cela pourrait avoir des effets secondaires inattendus.

Note

La commande **yes** analyse les variables. Par exemple :

```
bash$ yes $BASH_VERSION
3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 3.1.17(1)-release
 . . .
```

Cette « fonctionnalité » pourrait ne pas être très utile.

banner

Affiche les paramètres sur `stdout` comme une grande bannière verticale en utilisant un symbole ASCII (# par défaut). On peut rediriger cette sortie vers l'imprimante pour obtenir une copie papier.

printenv

[9] Ces scripts sont inspirés de ceux trouvés dans la distribution debian

Montre toutes les variables d'environnement réglées pour un utilisateur donné.

```
bash$ printenv | grep HOME
 HOME=/home/bozo
```

lp

Les commandes **lp** et **lpr** envoient un (des) fichier(s) à la file d'impression. [10] Ces commandes tirent l'origine de leurs noms des imprimantes « ligne par ligne » d'un autre âge.

`bash$ lp fichier1.txt` ou `bash lp <fichier1.txt`

Il est souvent utile d'envoyer le résultat de la commande **pr** à **lp**.

`bash$ pr -options fichier1.txt | lp`

Les outils de mise en forme comme groff et *Ghostscript* peuvent directement envoyer leurs sorties à **lp**.

`bash$ groff -Tascii fichier.tr | lp`

`bash$ gs -options | lp fichier.ps`

Les commandes sont **lpq** pour visualiser la file d'impression et **lprm** pour retirer des documents de la file d'impression.

tee

[UNIX emprunte une idée aux commerces de tuyauterie]

C'est un opérateur de redirection avec une petite différence : comme le « T » du plombier, il permet de « soutirer » *vers un fichier* la sortie d'une commande ou de plusieurs commandes à l'intérieur d'un tube mais sans affecter le résultat. Ceci est utile pour envoyer le résultat du processus en cours vers un fichier ou un papier, par exemple pour des raisons de débogage.

```
                            (redirection)
                           |----> vers le fichier
                           |
 ===========================|====================
 commande ---> commande ---> |tee ---> commande ---> ---> sortie du tube
 ============================================
```

```
cat listefichiers* | sort | tee fichier.verif | uniq > fichier.resultat
```

(le fichier `fichier.verif` contient les contenus concaténés puis triés des fichiers « listefichiers » avant que les doublons ne soient supprimés par uniq).

mkfifo

Cette commande obscure crée un *tube nommé*, un tampon temporaire pour transférer les données entre les programmes sur le principe du *first-in-first-out* (FIFO : premier arrivé, premier sorti). [11] Classiquement, un processus écrit dans le FIFO et un autre y lit. Voir l'Exemple A.15, « *fifo*: Faire des sauvegardes journalières, en utilisant des tubes nommés ».

```
#!/bin/bash
# Petit script d'Omair Eshkenazi.
# Utilisé dans le guide ABS avec sa permission (merci !).

mkfifo pipe1
mkfifo pipe2

(cut -d' ' -f1 | tr "a-z" "A-Z") >pipe2 <pipe1 &
ls -l | tr -s ' ' | cut -d' ' -f3,9- | tee pipe1 |
cut -d' ' -f2 | paste - pipe2

rm -f pipe1
rm -f pipe2

#  Pas besoin de tuer les processus en tâche de fond quand le script se termine.
```

[10] La *file d'impression* est l'ensemble des documents en attente d'impression.
[11] Pour une excellente vue d'ensemble du sujet, lire l'article de Andy Vaught *Introduction to Named Pipes* (Introduction aux tubes nommés) dans le numéro de septembre 1997 du *Linux Journal*.

```
#+ (pourquoi ?).

exit $?

Maintenant, lancez le script et expliquez sa sortie :
sh mkfifo-example.sh

4830.tar.gz          BOZO
pipe1   BOZO
pipe2   BOZO
mkfifo-example.sh    BOZO
Mixed.msg BOZO
```

pathchk

Ce programme vérifie la validité d'un nom de fichier. Il renvoie un message d'erreur si le nom excède la taille maximale autorisée (255 caractères) ou si un des répertoires du chemin est inaccessible, alors un message d'erreur est affiché.

Malheureusement, **pathchk** ne renvoie pas un code d'erreur interprétable, ce qui le rend assez inutile dans un script. Cherchez du côté des opérateurs de tests sur les fichiers si besoin.

dd

C'est une commande légèrement obscure et l'une des plus craintes des commandes de *duplication des données*. À l'origine, c'était un outil d'échange de données entre les bandes magnétiques des mini-ordinateurs unix et les mainframes d'IBM. Cette commande est encore utilisée à cet effet. **dd** copie simplement un fichier (ou `stdin/stdout`) mais en effectuant une conversion. ASCII/EBCDIC est une « conversion » possible [12] minuscule/majuscule, permutation des paires d'octets entre l'entrée et la sortie, saut et troncature des en-têtes et queues du fichier d'entrées.

```
# Convertir un fichier en majuscule :

dd if=$fichier conv=ucase > $fichier.majuscule
#                   lcase   # pour une conversion en minuscule
```

Voici quelques options basiques de **dd** :

- if=INFILE

  INFILE est le fichier *source*.

- of=OUTFILE

  OUTFILE est le fichier *cible*, le fichier où les données seront écrites.

- bs=BLOCKSIZE

  Ceci est la taille de chaque bloc de données en cours de lecture et d'écriture, habituellement une puissance de 2.

- skip=BLOCKS

  Nombre de blocs à sauter dans INFILE avant de commencer la copie. Ceci est utile quand INFILE commence avec des données corrompues ou quand il est préférable de copier seulement une portion de INFILE.

- seek=BLOCKS

  Nombre de blocs à sauter dans INFILE avant de commencer la copie, laissant des données blanches au début de OUTFILE.

- count=BLOCKS

  Copie seulement ce nombre de blocs de données, plutôt que le fichier INFILE entier.

- conv=CONVERSION

  Type de conversion à appliquer aux données d'INFILE avant l'opération de copie.

[12] EBCDIC (prononcer « ebb-sid-ick ») est l'acronyme de Extended Binary Coded Decimal Interchange Code. C'est un vieux format de données d'IBM qui n'a plus cours aujourd'hui. Une utilisation étrange de l'option `conv=ebcdic` est l'encodage simple (mais pas très sécurisé) de fichiers textes.

```
cat $file | dd conv=swab,ebcdic > $file_encrypted
# Encode (baragouin).
# on peut ajouter l'option swab pour obscurcir un peu plus

 cat $file_encrypted | dd conv=swab,ascii > $file_plaintext
# Decode.
```

dd --help liste toutes les options acceptées par cet outil puissant.

Exemple 15.56. Un script qui se copie lui-même

```
#!/bin/bash
# self-copy.sh

# Ce script se copie lui-même.

fichier_souscript=copy

dd if=$0 of=$0.$fichier_souscript 2>/dev/null
# Supprime les messages de dd:    ^^^^^^^^^^^

exit $?

#  Un programme dont la seule sortie est son propre code est appelé un
#+ "quine" par Willard Quine.
#  Est-ce que ce script peut être qualifié de "quine" ?
```

Exemple 15.57. S'exercer à *dd*

```
#!/bin/bash
# exercising-dd.sh

# Script de Stephane Chazelas.
# Quelque peu modifié par l'auteur du guide ABS.

fichier_en_entree=$0   # Ce script.
fichier_en_sortie=traces.txt # Le fichier en sortie
n=3
p=5

dd if=$fichier_en_entree of=$fichier_en_sortie \
  bs=1 skip=$((n-1)) count=$((p-n+1)) 2> /dev/null
# Extrait les caractères de n à p (3 à 5) à partir de ce script.

# ------------------------------------------------------

echo -n "bonjour le monde" | dd cbs=1 conv=unblock 2> /dev/null
# Affiche "bonjour le monde" verticalement.
# Pourquoi ? un retour de ligne envoyé par dd après chaque caractère.

exit 0
```

Pour montrer à quel point **dd** est souple, utilisons-le pour capturer nos saisies.

Exemple 15.58. Capturer une saisie

```
#!/bin/bash
#  dd-keypress.sh
#+ Capture des touches clavier sans avoir besoin d'appuyer sur ENTER.


touches_appuyees=4                    # Nombre de touches à capturer.


ancien_parametrage_du_tty=$(stty -g)  # Sauve l'ancienne configuration du terminal.

echo "Appuyez sur $touches_appuyees touches."
stty -icanon -echo                    # Désactive le mode canonique.
                                      # Désactive l'echo local.
touches=$(dd bs=1 count=$touches_appuyees 2> /dev/null)
# 'dd' utilise stdin si "if" (input file, fichier en entrée) n'est pas spécifié.
```

```
stty "$ancien_parametrage_du_tty"     # Restaure l'ancien paramètrage du terminal.

echo "Vous avez appuyé sur les touches \"$touches\"."

# Merci, Stéphane Chazelas, pour avoir montré la façon.
exit 0
```

dd peut effectuer un accès aléatoire sur un flux de données.

```
echo -n . | dd bs=1 seek=4 of=fichier conv=notrunc
# l'option "conv=notrunc" signifie que la sortie ne sera pas tronquée.
# Merci, S.C.
```

dd peut copier les données brutes d'un périphérique (comme un lecteur de disquette ou de bande magnétique) vers une image et inversement (Exemple A.5, « *copy-cd* : Copier un CD de données »). On l'utilise couramment pour créer des disques de démarrage.

```
dd if=kernel-image of=/dev/fd0H1440
```

De la même manière, **dd** peut copier le contenu entier d'un disque (même formaté avec un autre OS) vers un fichier image.

```
dd if=/dev/fd0 of=/home/bozo/projects/floppy.img
```

Comme autres exemples d'applications de **dd**, on peut citer l'initialisation d'un fichier swap temporaire (Exemple 28.2, « Créer un fichier de swap en utilisant `/dev/zero` ») ou d'un disque en mémoire (Exemple 28.3, « Créer un disque ram »). **dd** peut même effectuer la copie bas-niveau d'une partition complète d'un disque dur même si la pratique n'est pas conseillée.

Les gens (qui n'ont probablement rien à faire de mieux de leur temps) pensent constamment à de nouvelles applications intéressantes de **dd**.

Exemple 15.59. Effacer les fichiers de façon sûre

```
#!/bin/bash
# blot-out.sh : Efface "toutes" les traces d'un fichier.

#  Ce script écrase un fichier cible avec des octets pris au hasard, puis avec
#+ des zéros, avant de le supprimer définitivement.
#  Après cela, même l'examen des secteurs du disque par des méthodes
#+ conventionnelles ne permet pas de retrouver
#+ les données du fichier d'origine.

PASSES=7         # Nombre d'écriture sur le fichier.
                 #  L'augmenter ralentit l'exécution du script,
                 #+ spécialement sur les gros fichiers.
TAILLEBLOC=1     #  Les entrées/sorties avec /dev/urandom requièrent la taille
                 #+ d'un bloc, sinon vous obtiendrez des résultats bizarres.
E_MAUVAISARGS=70 # Divers codes d'erreur
E_NON_TROUVE=71
E_CHANGE_D_AVIS=72

if [ -z "$1" ]   # Aucun nom de fichier spécifié.
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

fichier=$1

if [ ! -e "$fichier" ]
then
  echo "Le fichier \"$fichier\" est introuvable."
  exit $E_NON_TROUVE
fi

echo
echo -n "Êtes-vous absolument sûr de vouloir complètement écraser \"$fichier\" (o/n)
?"
```

```
read reponse
case "$reponse" in
[nN]) echo "Vous avez changé d'idée, hum ?"
      exit $E_CHANGE_D_AVIS
      ;;
*)    echo "Écrasement du fichier \"$fichier\".";;
esac


longueur_fichier=$(ls -l "$fichier" | awk '{print $5}')
      # Le 5e champ correspond à la taille du fichier.

nb_passe=1

chmod u+w "$fichier"   # Autorise l'écrasement ou la suppression du fichier.

echo

while [ "$nb_passe" -le "$PASSES" ]
do
  echo "Passe #$nb_passe"
  sync         # Vider les tampons.
  dd if=/dev/urandom of=$fichier bs=$TAILLEBLOC count=$longueur_fichier
               # Remplir avec des octets pris au hasard.
  sync         # Vider de nouveau les tampons.
  dd if=/dev/zero of=$fichier bs=$TAILLEBLOC count=$longueur_fichier
               # Remplir avec des zéros.
  sync         # Vider encore une fois les tampons.
  let "nb_passe += 1"
  echo
done


rm -f $fichier # Finalement, supprime le fichier brouillé et déchiqueté.
sync           # Vide les tampons une dernière fois.

echo "Le fichier \"$fichier\" a été complètement écrasé et supprimé."; echo


#  C'est une méthode assez sécurisée, mais inefficace et lente pour massacrer
#+ un fichier. La commande "shred", faisant partie du paquetage GNU "fileutils",
#+ fait la même chose mais de façon plus efficace.

#  Le fichier ne peut pas être récupéré par les méthodes habituelles.
#  Néanmoins...
#+ cette simple méthode ne pourra certainement *pas* résister à des méthodes
#+ d'analyse plus sophistiquées.

#  Ce script pourrait ne pas fonctionner correctement avec un système de fichiers
#+ journalisé.
#  Exercice (difficile) : corrigez ce défaut.

#  Le paquetage de suppression de fichier "wipe" de Tom Vier fait un travail
#+ bien plus en profondeur pour massacrer un fichier que ce simple script.
#     http://www.ibiblio.org/pub/Linux/utils/file/wipe-2.0.0.tar.bz2

#  Pour une analyse en détail du thème de la suppression de fichier et de la
#+ sécurité, voir le papier de Peter Gutmann,
#+     "Secure Deletion of Data From Magnetic and Solid-State Memory".
#           http://www.cs.auckland.ac.nz/~pgut001/secure_del.html


exit 0
```

Voir aussi l'entrée dd thread dans la bibliographie.

od

Le filtre **od** (pour *octal dump*) convertit l'entré en octal (base 8) ou dans une autre base. C'est très utile pour voir ou traiter des

fichiers binaires ou d'autres sources de données illisibles comme `/dev/urandom`.

```
head -c4 /dev/urandom | od -N4 -tu4 | sed -ne '1s/.* //p'
# Exemple d'affichage : 1324725719, 3918166450, 2989231420, etc.

# À partir du script exemple rnd.sh, par Stéphane Chazelas
```

Voir aussi Exemple 9.31, « Réinitialiser RANDOM » et Exemple A.38, « Tri d'insertion ».

hexdump

Liste le contenu en hexadécimal, octal, décimal ou ASCII d'un fichier binaire. **hexdump** est un équivalent moins complet d'**od**, traité ci-dessus. Elle pourrait être utilisée pour visualiser le contenu d'un fichier binaire, en combinaison avec dd et less.

```
dd if=/bin/ls | hexdump -C | less
# L'option -C formate joliment la sortie sous forme d'un tableau.
```

objdump

Affiche des informations sur un objet ou un exécutable binaire sous sa forme hexadécimale ou en tant que code désassemblé (avec l'option `-d`).

```
bash$ objdump -d /bin/ls
 /bin/ls:     file format elf32-i386

 Disassembly of section .init:

 080490bc <.init>:
 80490bc:       55                      push   %ebp
 80490bd:       89 e5                   mov    %esp,%ebp
 . . .
```

mcookie

Cette commande génère un fichier témoin (« magic cookie »), un nombre hexadécimal pseudo-aléatoire de 128 bits (32 caractères) qui est habituellement utilisé par les serveurs X comme « signature » pour l'authentification. Elle peut être utilisée dans un script comme une solution sale mais rapide pour générer des nombres aléatoires.

```
random000=$(mcookie)
```

Évidemment, un script peut utiliser md5 pour obtenir le même résultat.

```
# Génère la somme de contrôle md5 du script lui-même.
 random001=`md5sum $0 | awk '{print $1}'`
# Utilise awk pour supprimer le nom du fichier
```

mcookie est aussi une autre facon de générer un nom de fichier « unique ».

Exemple 15.60. Générateur de nom de fichier

```
#!/bin/bash
# tempfile-name.sh :  générateur de fichier temporaire.

BASE_STR=`mcookie`   # Chaîne magique de 32 caractères.
POS=11               # Position arbitraire dans la chaîne magique.
LONG=5               # Pour obtenir $LONG caractères consécutifs.

prefixe=temp         #  C'est après tout un fichier "temp"oraire.
                     #  Pour que le nom soit encore plus "unique", génère le
                     #+ préfixe du nom du fichier en utilisant la même méthode
                     #+ que le suffixe ci-dessous.

suffixe=${BASE_STR:POS:LONG}
                     # Extrait une chaîne de cinq caractères, commençant à la
                     # position 11.
```

```
nomfichiertemporaire=$prefixe.$suffixe
                     # Construction du nom du fichier.

echo "Nom du fichier temporaire = "$nomfichiertemporaire""

# sh tempfile-name.sh
# Nom du fichier temporaire = temp.e19ea

#  Comparez cette méthode de création de noms de fichier uniques
#+ avec la méthode 'date' dans ex51.sh.

exit 0
```

units

Généralement appelé de façon interactive, cet utilitaire peut être utilisé dans un script. Il sert à convertir des *mesures en différentes unités*.

Exemple 15.61. Convertir des mètres en miles

```
#!/bin/bash
# unit-conversion.sh


convertir_unites ()  # Prend comme arguments les unités à convertir.
{
  cf=$(units "$1" "$2" | sed --silent -e '1p' | awk '{print $2}')
  # Supprime tout sauf le facteur conversion.
  echo "$cf"
}

Unite1=miles
Unite2=meters
facteur_conversion=`convertir_unites $Unite1 $Unite2`
quantite=3.73

resultat=$(echo $quantite*$facteur_conversion | bc)

echo "Il existe $resultat $Unite2 dans $quantite $Unit1."

#  Que se passe-t'il si vous donnez des unités incompatibles, telles que
#+ "acres" et "miles" ?

exit 0
```

m4

Trésor caché, **m4** est un puissant filtre de traitement des macros. [13] Langage pratiquement complet, **m4** fut écrit comme préprocesseur pour *RatFor* avant de s'avérer être un outil autonome très utile. En plus de ses possibilités étendues d'interpolation de macros, **m4** intègre les fonctionnalités d'eval, tr et awk634.

Un très bon article sur **m4** et ses utilisations a été écrit pour le numéro d'avril 2002 du *Linux Journal*.

Exemple 15.62. Utiliser *m4*

```
#!/bin/bash
# m4.sh : Utiliser le processeur de macros m4

# Chaîne de caractères
chaine=abcdA01
echo "len($chaine)" | m4                        # 7
echo "substr($chaine,4)" | m4                   # A01
echo "regexp($chaine,[0-1][0-1],\&Z)" | m4      # 01Z
```

[13] Une *macro* est une constante symbolique qui se substitue à une simple chaine de caractères ou à une operation sur une série d'arguements.

```
# Arithmétique
echo "incr(22)" | m4                                # 23
echo "eval(99 / 3)" | m4                            # 33

exit 0
```

xmessage
Cette variante basée sur X de echo affiche un dialogue sur le bureau.

```
xmessage Clic gauche pour continuer -button okay
```

zenity
L'outil *zenity* permet d'afficher un dialogue utilisant les composants *GTK+* et très intéressant pour l'écriture de scripts463.

doexec

doexec permet de transmettre une liste quelconque d'arguments à un *binaire exécutable*. En particulier , le fait de transmettre *`argv[0]`* (qui correspond à $0 dans un script) permet à l'exécutable d'être invoqué avec des noms différents et d'agir en fonction de cette invocation. Ceci n'est qu'une autre façon de passer des options à un exécutable.

Par exemple , le répertoire `/usr/local/bin` peut contenir un binaire appelé « aaa ». **doexec /usr/local/bin/aaa list** affichera la liste de tous les fichiers du répertoire courant qui commencent par un « a ». Appeler le même binaire par **doexec /usr/local/bin/aaa delete** détruira ces fichiers.

Note

Les différentes actions d'un exécutable doivent être définies à l'intérieur du code exécutable lui-même. De façon similaire au script suivant :

```
case `basename $0` in
 "name1" ) faire_qqchose;;
 "name2" ) faire_qqchose_d_autre;;
 "name3" ) encore_autre_chose;;
 *       ) quitter;;
 esac
```

dialog

La famille d'outils dialog462 fournit une méthode pour appeler des fenêtres de « dialog »ues interactives à partir d'un script. Les variations plus élaborées de **dialog** -- **gdialog**, **Xdialog**, et **kdialog** -- appelle en fait les widgets X-Windows.

sox

La commande **sox**, ou « **so**und e**x**change » (échange de sons), joue et réalise des transformations sur des fichiers son. En fait, l'exécutable `/usr/bin/play` (maintenant obsolète) n'est rien de plus qu'un emballage shell pour *sox*.

Par exemple, **sox fichierson.wav fichierson.au** modifie un fichier son WAV en fichier son AU (format audio de Sun).

Les scripts shells correspondent parfaitement à des exécutions nombreuses comme les opérations de **sox** sur des fichiers son. Par exemple, voir le *guide pratique « Linux Radio Timeshift »* et le *projet MP3do*.

Chapitre 16. Commandes système et d'administration

Les scripts de démarrage et d'arrêt du répertoire `/etc/rc.d` illustrent l'utilisation (et l'intérêt) de ces commandes. Elles sont généralement appelées par *root* et utilisées pour la maintenance du système ou pour des réparation en urgence du système de fichiers. Utilisez-les avec précaution car certaines de ces commandes peuvent endommager votre système en cas de mauvaise utilisation.

Utilisateurs et groupes

users

Affiche tous les utilisateurs connectés. Ceci est l'équivalent approximatif de **who -q**.

groups

Affiche l'utilisateur actuel et les groupes auxquels il appartient. Ceci correspond à la variable interne $GROUPS mais donne les noms des groupes plutôt que leur identifiants.

```
bash$ groups
bozita cdrom cdwriter audio xgrp

bash$ echo $GROUPS
501
```

chown, **chgrp**

La commande **chown** modifie le propriétaire d'un ou plusieurs fichiers. Cette commande est utilisée par *root* pour modifier le propriétaire d'un fichier. Un utilisateur ordinaire peut ne pas pouvoir changer le propriétaire des fichiers, même pas pour ses propres fichiers. [1]

```
root# chown bozo *.txt
```

La commande **chgrp** modifie le *groupe* d'un ou plusieurs fichiers. Vous devez être le propriétaire du fichier ainsi qu'un membre du groupe de destination (ou *root*) pour réaliser cette opération.

```
chgrp --recursive dunderheads *.data
#  Ce groupe "dunderheads" sera le propriétaire de tous les fichiers "*.data"
#+ du répertoire $PWD et de ses sous-répertoires (c'est ce que sous-entend le
#+ "recursive").
```

useradd, **userdel**

La commande d'administration **useradd** ajoute un compte utilisateur au système et crée un répertoire personnel pour cet utilisateur particulier si cela est demandé. La commande correspondante **userdel** supprime le compte de l'utilisateur du système [2] et supprime les fichiers associés.

Note

La commande **adduser** est un synonyme de **useradd** et est habituellement un lien symbolique vers ce dernier.

usermod

Modifie un compte utilisateur. Les modifications concernent le mot de passe, le groupe d'appartenance, la date d'expiration et d'autres attributs d'un compte utilisateur donné. Avec cette commande, le mot de passe d'un utilisateur peut être verrouillé, ce qui a pour effect de désactiver le compte.

groupmod

Modifie un groupe donné. Le nom du groupe et/ou son numéro d'identifiant est modifiable avec cette commande.

id

[1] C'est le cas pour les machines Linux ou UNIX disposant d'un système de gestion de quotas disque.
[2] La commande **userdel** échouera si l'utilisateur en cours de suppression est connecté à ce moment.

La commande **id** affiche les identifiants réels de l'utilisateur et du groupe pour l'utilisateur associé au processus actuel. C'est la contre-partie des variables internes Bash $UID, $EUID et $GROUPS.

```
bash$ id
uid=501(bozo) gid=501(bozo) groups=501(bozo),22(cdrom),80(cdwriter),81(audio)

bash$ echo $UID
501
```

Note

La commande **id** affiche les identifiants *actuels* seulement s'ils diffèrent des *vrais*.

Voir aussi l'Exemple 9.5, « Suis-je root ? ».

who

Affiche tous les utilisateurs connectés sur le système.

```
bash$ who
bozo  tty1     Apr 27 17:45
 bozo  pts/0    Apr 27 17:46
 bozo  pts/1    Apr 27 17:47
 bozo  pts/2    Apr 27 17:49
```

L'option `-m` donne des informations détaillées sur l'utilisateur actuel. Passer n'importe quels arguments, à condition qu'il en ait deux, à **who** est l'équivalent de **who -m**, comme dans **who am i** ou **who The Man**.

```
bash$ who -m
localhost.localdomain!bozo  pts/2    Apr 27 17:49
```

whoami est similaire à **who -m** mais affiche seulement le nom de l'utilisateur.

```
bash$ whoami
bozo
```

w

Affiche tous les utilisateurs connectés et les processus leur appartenant. C'est une version étendue de **who**. La sortie de **w** peut être envoyée via un tube vers grep pour trouver un utilisateur et/ou un processus spécifique.

```
bash$ w | grep startx
bozo  tty1     -                4:22pm  6:41   4.47s  0.45s  startx
```

logname

Affiche le nom de connexion de l'utilisateur actuel (disponible dans `/var/run/utmp`). C'est presque l'équivalent de whoami283, ci-dessus.

```
bash$ logname
bozo

bash$ whoami
bozo
```

Néanmoins...

```
bash$ su
Password: ......

bash# whoami
root
bash# logname
```

```
bozo
```

Note

Bien que **logname** affiche le nom de l'utilisateur connecté, **whoami** donne le nom de l'utilisateur attaché au processus actuel. Comme nous l'avons déjà dit, ils ne sont parfois pas identiques.

su

Lance un programme ou un script en substituant l'utilisateur (**s**ubstitue l'**u**tilisateur). **su rjones** lance un shell en tant qu'utilisateur *rjones*. Une commande **su** sans arguments utilise *root* par défaut. Voir l'Exemple A.15, « *fifo*: Faire des sauvegardes journalières, en utilisant des tubes nommés ».

sudo

Lance une commande en tant que *root* (ou un autre utilisateur). Ceci peut être utilisé dans un script, permettant ainsi à un utilisateur standard de lancer un script.

```
#!/bin/bash

# Quelques commandes.
sudo cp /root/secretfile /home/bozo/secret
# Quelques autres commandes.
```

Le fichier `/etc/sudoers` contient le nom des utilisateurs ayant le droit d'appeller **sudo**.

passwd

Initialise ou modifie le mot de passe d'un utilisateur.

passwd peut être utilisé dans un script mais *ne devrait probablement pas* l'être.

Exemple 16.1. Configurer un nouveau mot de passe

```
#!/bin/bash
#  setnew-password.sh : Pour des raisons de démonstration seulement.
#                       Exécuter ce script n'est pas une bonne idée.
#  Ce script doit être exécuté en tant que root.

UID_ROOT=0                   # Root possède l' $UID 0.
E_MAUVAIS_UTILISATEUR=65     # Pas root ?

E_UTILISATEUR_INEXISTANT=70
SUCCES=0


if [ "$UID" -ne "$UID_ROOT" ]
then
  echo; echo "Seul root peut exécuter ce script."; echo
  exit $E_MAUVAIS_UTILISATEUR
else
  echo
  echo "Vous devriez en savoir plus pour exécuter ce script, root."
  echo "Même les utilisateurs root ont le blues... "
  echo
fi


utilisateur=bozo
NOUVEAU_MOTDEPASSE=security_violation

# Vérifie si bozo vit ici.
grep -q "$utilisateur" /etc/passwd
if [ $? -ne $SUCCES ]
then
  echo "L'utilisateur $utilisateur n'existe pas."
  echo "Le mot de passe n'a pas été modifié."
  exit $E_UTILISATEUR_INEXISTANT
```

```
fi

echo "$NOUVEAU_MOTDEPASSE" | passwd --stdin "$utilisateur"
#  L'option '--stdin' de 'passwd' permet
#+ d'obtenir un nouveau mot de passe à partir de stdin (ou d'un tube).

echo; echo "Le mot de passe de l'utilisateur $utilisateur a été changé !"

# Utiliser la commande 'passwd' dans un script est dangereux.

exit 0
```

Les options `-l`, `-u` et `-d` de la commande **passwd** permettent de verrouiller, déverrouiller et supprimer le mot de passe d'un utilisateur. Seul *root* peut utiliser ces options.

ac

Affiche le temps de connexion des utilisateurs actuellement connectés à partir des informations lues dans `/var/log/wtmp`. Il fait partie des utilitaires de mesure GNU.

```
bash$ ac
        total       68.08
```

last

Affiche les derniers (*last* en anglais) utilisateurs connectés suivant les informations disponibles dans `/var/log/wtmp`. Cette commande peut aussi afficher les connexions distantes.

Par exemple, pour afficher les dernières fois où le système a redémarré :

```
bash$ last reboot
reboot   system boot  2.6.9-1.667      Fri Feb  4 18:18          (00:02)
reboot   system boot  2.6.9-1.667      Fri Feb  4 15:20          (01:27)
reboot   system boot  2.6.9-1.667      Fri Feb  4 12:56          (00:49)
reboot   system boot  2.6.9-1.667      Thu Feb  3 21:08          (02:17)
...

wtmp begins Tue Feb  1 12:50:09 2005
```

newgrp

Modifie l'*identifiant du groupe* de l'utilisateur sans se déconnecter. Ceci permet l'accès aux fichiers du nouveau groupe. Comme les utilisateurs peuvent être membres de plusieurs groupes simultanément, cette commande a peu d'utilité.

Note

Kurt Glaesemann indique que la commande pourrait s'avérer utile pour configurer les droits du groupe par défaut pour les fichiers écrit par un utilisateur. Néanmoins, la commande chgrp pourrait être plus efficace pour cela.

Terminaux

tty

Affiche le nom du terminal de l'utilisateur actuel. Notez que chaque fenêtre *xterm* compte comme un terminal séparé.

```
bash$ tty
/dev/pts/1
```

stty

Affiche et/ou modifie les paramétrages du terminal. Cette commande complexe, utilisée dans un script, peut contrôler le comportement du terminal et la façon dont il affiche des caractères. Voir la page info et l'étudier en profondeur.

Exemple 16.2. Configurer un caractère d'*effacement*

```
#!/bin/bash
# erase.sh : Utilisation de "stty" pour initialiser un caractère d'effacement
# lors de la lecture de l'entrée standard.

echo -n "Quel est ton nom? "
read nom                        #  Essayez la touche Backspace
                                #+ pour effacer quelques caractères.
                                #  Problèmes ?
echo "Votre nom est $nom."

stty erase '#'                  # Initialisation de la "dièse" (#) comme
                                # caractère d'effacement.
echo -n "Quel est ton nom ? "
read nom                        # Utilisez # pour effacer le dernier caractère
                                # saisi.
echo "Votre nom est $nom."

# Attention : même après la sortie du script, la nouvelle clé reste initialisée.


exit 0

# Même après la sortie du script, la nouvelle clé reste initialisée.
# Exercice : Comment réinitialiser le caractère d'échappement à sa valeur par défaut
?
```

Exemple 16.3. ***Mot de passe secret*** **: Désactiver l'écho du terminal**

```
#!/bin/bash
# secret-pw.sh : mot de passe secret

echo
echo -n "Entrez le mot de passe "
read mot_de_passe
echo "Le mot de passe est $mot_de_passe"
echo -n "Si quelqu'un a regardé par dessus votre épaule, "
echo "votre mot de passe pourrait avoir été compromis."

echo && echo  # Deux retours chariot dans une "liste ET".


stty -echo    # Supprime l'echo sur l'écran.

echo -n "Entrez de nouveau le mot de passe "
read mot_de_passe
echo
echo "Le mot de passe est $mot_de_passe"
echo

stty echo     # Restaure l'echo de l'écran.

exit 0

#  Faites un 'info stty'
#+ pour plus d'informations sur cette commande utile mais complexe.
```

Une utilisation originale de **stty** concerne la détection de l'appui d'une touche (sans appuyer sur **ENTER**).

Exemple 16.4. Détection de l'appui sur une touche

```
#!/bin/bash
# keypress.sh : Détecte un appui sur une touche ("hot keys").

echo
```

```
ancienne_config_tty=$(stty -g)   # Sauvegarde de l'ancienne configuration (pourquoi
?).
stty -icanon
Appui_touche=$(head -c1)         # ou $(dd bs=1 count=1 2> /dev/null)
                                 # sur les systèmes non-GNU

echo
echo "La touche est \""$Appui_touche"\"."
echo

stty "$ancienne_config_tty"      # Restaure l'ancienne configuration.

# Merci, Stephane Chazelas.

exit 0
```

Voir aussi l'Exemple 9.3, « Encore une fois, saisie avec délai ».

Terminaux et modes

Normalement, un terminal fonctionne en mode *canonique*. Lorsque l'utilisateur appuie sur une touche, le caractère correspondant ne va pas immédiatement au programme en cours sur le terminal, Un tampon local au terminal enregistre les frappes clavier. Lorsqu'un utilisateur appuie sur la touche **ENTER**, il envoie toutes les touches frappées au programme en cours. Il existe même un éditeur ligne basique dans le terminal.

```
bash$ stty -a
speed 9600 baud; rows 36; columns 96; line = 0;
 intr = ^C; quit = ^\; erase = ^H; kill = ^U; eof = ^D; eol = <undef>; eol2 =
<undef>;
 start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W; lnext = ^V; flush =
^O;
 ...
 isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
```

En utilisant le mode canonique, il est possible de redéfinir les touches spéciales pour l'éditeur ligne local du terminal.

```
bash$ cat > filexxx
wha<ctl-W>I<ctl-H>foo bar<ctl-U>hello world<ENTER>]]</userinput>
<userinput><![CDATA[ctl-D>]]</userinput>
<prompt>bash$ </prompt><userinput><![CDATA[cat fichierxxx
hello world
bash$ wc -c < fichierxxx
17
```

Le processus contrôlant le terminal reçoit seulement 12 caractères (11 alphabétiques, plus le retour chariot), bien que l'utilisateur ait appuyé sur 26 touches.

Dans un mode non canonique (« raw »), chaque appui sur une touche (y compris les touches spéciales d'édition telles que **ctl-H**) envoie un caractère immédiatement au processus de contrôle.

L'invite Bash désactive à la fois `icanon` et `echo` car il remplace l'éditeur ligne basique du terminal avec son propre éditeur plus élaboré. Par exemple, lorsque vous appuyez sur **ctl-A** à l'invite Bash, aucun **^A** n'est affiché par le terminal mais Bash obtient un caractère **\1**, l'interprète et déplace le curseur en début de ligne.

Stéphane Chazelas

setterm

Initialise certains attributs du terminal. Cette commande écrit sur la sortie (`stdout`) de son terminal une chaîne modifiant le comportement de ce terminal.

```
bash$ setterm -cursor off
bash$
```

La commande **setterm** peut être utilisé dans un script pour modifier l'apparence du texte écrit sur `stdout` bien qu'il existe certainement de meilleurs outils440 dans ce but.

```
setterm -bold on
echo bold bonjour

setterm -bold off
echo normal bonjour
```

tset

Affiche ou initialise les paramétrages du terminal. C'est une version **stty** comprenant moins de fonctionnalités.

```
bash$ tset -r
Terminal type is xterm-xfree86.
 Kill is control-U (^U).
 Interrupt is control-C (^C).
```

setserial

Initialise ou affiche les paramètres du port série. Cette commande doit être exécutée par l'utilisateur *root* et est habituellement utilisée dans un script de configuration du système.

```
# From /etc/pcmcia/serial script :

IRQ=`setserial /dev/$DEVICE | sed -e 's/.*IRQ: //'`
setserial /dev/$DEVICE irq 0 ; setserial /dev/$DEVICE irq $IRQ
```

getty, **agetty**

Le processus d'initialisation d'un terminal utilise **getty** ou **agetty** pour demander le nom de connexion d'un utilisateur. Ces commandes ne sont pas utilisées dans des scripts shell d'utilisateurs. Leur contre-partie script est **stty**.

mesg

Active ou désactive les droits d'écriture sur le terminal de l'utilisateur actuel. Désactiver l'accès empêcherait tout utilisateur sur le réseau d'écrire (write en anglais) sur le terminal.

Astuce

Il peut être très ennuyant de voir apparaître un message pour une commande de pizza au milieu du fichier texte en cours d'édition. Sur un réseau multi-utilisateur, vous pourriez du coup souhaiter désactiver les droits d'écriture sur votre terminal lorsque vous ne voulez pas être dérangé.

wall

C'est un acronyme pour « write all », c'est-à-dire écrire un message à tous les utilisateurs sur tous les terminaux connectés sur le réseau. C'est essentiellement un outil pour l'administrateur système, utile par exemple pour prévenir tout le monde que le système sera bientôt arrêté à cause d'un problème (voir l'Exemple 18.1, « *broadcast* : envoie des messages à chaque personne connectée »).

```
bash$ wall Système arrêté pour maintenance dans 5 minutes!
Broadcast message from bozo (pts/1) Sun Jul  8 13:53:27 2001...

 Système arrêté pour maintenance dans 5 minutes!
```

Note

Si le droit d'écriture sur un terminal particulier a été désactivé avec **mesg**, alors **wall** ne pourra pas envoyer un message à ce terminal.

Informations et statistiques

uname

Affiche les spécifications du système (OS, version du noyau, etc.) sur `stdout`. Appelé avec l'option `-a`, donne plus d'informations sur le système (voir l'Exemple 15.5, « Fichier de traces utilisant *xargs* pour surveiller les journaux système »). L'option `-s` affiche seulement le type de l'OS.

```
bash$ uname
Linux

bash$ uname -s
Linux


bash$ uname -a
Linux iron.bozo 2.6.15-1.2054_FC5 #1 Tue Mar 14 15:48:33 EST 2006
 i686 i686 i386 GNU/Linux
```

arch

Affiche l'architecture du système. Équivalent à **uname -m**. Voir l'Exemple 10.26, « Utiliser la *substitution de commandes* pour générer la variable *case* ».

```
bash$ arch
i686

bash$ uname -m
i686
```

lastcomm

Donne une information sur les dernières commandes, disponibles dans le fichier `/var/account/pacct`. Le nom de la commande et de l'utilisateur peuvent être spécifiés en options. Elle fait partie des utilitaires de comptage GNU.

lastlog

Affiche la dernière connexion de tous les utilisateurs système. Ceci prend comme référence le fichier `/var/log/lastlog`.

```
bash$ lastlog
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
 bin                                     **Never logged in**
 daemon                                  **Never logged in**
 ...
 bozo          tty1                       Sat Dec  8 21:14:29 -0700 2001



bash$ lastlog | grep root
root          tty1                      Fri Dec  7 18:43:21 -0700 2001
```

Attention

Cette commande échouera si l'utilisateur l'appellant n'a pas des droits de lecture sur le fichier `/var/log/lastlog`.

lsof

Affiche les fichiers ouverts. Cette commande affiche une table détaillée de tous les fichiers ouverts et donne de l'information sur leur propriétaire, taille, processus associés et bien plus encore. Bien sûr, **lsof** pourrait être redirigé avec un tube vers grep et/ou awk634 pour analyser ce résultat.

```
bash$ lsof
COMMAND    PID    USER   FD   TYPE     DEVICE    SIZE     NODE NAME
 init         1    root  mem    REG        3,5   30748    30303 /sbin/init
 init         1    root  mem    REG        3,5   73120     8069 /lib/ld-2.1.3.so
```

```
 init         1    root  mem    REG        3,5  931668     8075 /lib/libc-2.1.3.so
 cardmgr    213    root  mem    REG        3,5   36956    30357 /sbin/cardmgr
 ...
```

La commande **lsof** est un outil très utile, mais aussi très complexe. Si vous n'arrivez pas à démonter un système de fichiers et que vous obtenez un message d'erreur indiquant qu'il est en cours d'utilisation, exécuter *lsof* aide à déterminer les fichiers qui sont toujours ouverts sur ce système de fichiers. L'option `-i` liste les fichiers socket réseau. Cela peut aider à tracer les intrusions et les tentatives de pénétration.

```
bash$ lsof -an -i tcp
COMMAND  PID USER  FD  TYPE DEVICE SIZE NODE NAME
 firefox 2330 bozo  32u IPv4   9956       TCP 66.0.118.137:57596->67.112.7.104:http
...
 firefox 2330 bozo  38u IPv4  10535       TCP 66.0.118.137:57708->216.79.48.24:http
...
```

strace

System **trace** : outil de diagnostic et de débogage traçant les *appels système* et les signaux. Cette commande et **ltrace**, qui suit, permettent de diagnostiquer l'échec d'un programme en exécution... un échec qui peut être dû à des bibliothèques manquantes ou à une autre cause proche.

```
bash$ strace df
execve("/bin/df", ["df"], [/* 45 vars */]) = 0
 uname({sys="Linux", node="bozo.localdomain", ...}) = 0
 brk(0)                                  = 0x804f5e4

 ...
```

C'est l'équivalent Linux de la commande **truss** sur Solaris.

ltrace

Library **trace** : outil de diagnostic et de débogage, traçant les *appels de bibliothèque* réalisé par une commande donnée.

```
bash$ ltrace df
__libc_start_main(0x804a910, 1, 0xbfb589a4, 0x804fb70, 0x804fb68 <unfinished ...&gt:
 setlocale(6, "")                          = "en_US.UTF-8"
bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("coreutils")                    = "coreutils"
__cxa_atexit(0x804b650, 0, 0, 0x8052bf0, 0xbfb58908) = 0
getenv("DF_BLOCK_SIZE")                    = NULL

 ...
```

nmap

Network **map**per et scanner de port réseau. Cette commande parcourt les ports d'un serveur pour localiser les ports ouverts et les services associés à ces ports. Il peut aussi ramener des informations sur les filtres de paquets et les pares-feu. C'est un important outil de sécurité pour verrouiller un réseau contre les tentatives de pirates.

```
#!/bin/bash

SERVEUR=$HOST                           # localhost.localdomain (127.0.0.1).
NUMERO_PORT=25                          # Port SMTP.

nmap $SERVEUR | grep -w "$NUMERO_PORT"  # Ce port particulier est-il ouvert ?
#               grep -w établit une correspondance avec des mots entiers
#+              seulement, donc cela permet d'éviter le port 1025 par exemple.

exit 0

# 25/tcp     open        smtp
```

nc

L'utilitaire **nc** (*netcat*) est un ensemble d'outils pour se connecter ou pour écouter à des ports TCP et UDP. Il est utile comme outil de diagnostic et de tests, ainsi que comme composant de scripts simples basés sur les clients et serveurs HTTP.

```
bash$ nc localhost.localdomain 25
220 localhost.localdomain ESMTP Sendmail 8.13.1/8.13.1;
 Thu, 31 Mar 2005 15:41:35 -0700
```

Exemple 16.5. Vérification d'*identd* sur un serveur distant

```
#! /bin/sh
## Duplique l' « ident-scan » de DaveG en utilisant netcat. Oooh, ça va l'embêter.
## Args: cible port [port port port ...]
## Assemble stdout _et_ stderr.
##
##  Avantages : s'exécute plus lentement qu'ident-scan,
##+ donnant à un inetd distant moins de raison de s'alarmer
##+ et ne prend pour cible que les quelques ports que vous spécifiez.
##  Inconvénients : requiert les arguments du port dans leur version
##+ numérique uniquement, la paresse de l'affichage,
##+ et ne fonctionnera pas pour les r-services lorsqu'ils proviennent
##+ de ports sources supérieurs.
# Auteur du script : Hobbit &lt;hobbit@avian.org&gt;
# Utilisé dans le guide ABS avec sa permission.

# ---------------------------------------------------
E_MAUVAISARGS=65       # A besoin d'au moins deux arguments.
TWO_WINKS=2            # Combien de temps pour dormir.
THREE_WINKS=3
IDPORT=113             # Port d'authentification avec ident.
HASARD1=999
HASARD2=31337
TIMEOUT0=9
TIMEOUT1=8
TIMEOUT2=4
# ---------------------------------------------------

case "${2}" in
  "" ) echo "A besoin d'un hôte et d'au moins un numéro de port." ; exit
$E_MAUVAISARGS ;;
esac

# "Ping"uez-les une fois et vérifiez s'ils utilisent identd.
nc -z -w $TIMEOUT0 "$1" $IDPORT || \
{ echo "Oups, $1 n'utilise pas identd." ; exit 0 ; }
#  -z parcourt les démons en écoute.
#     -w $TIMEOUT = Durée de l'essai de connexion.

# Génère un port de base au hasard.
RP=`expr $$ % $HASARD1 + $HASARD2`

TRG="$1"
shift

while test "$1" ; do
  nc -v -w $TIMEOUT1 -p ${RP} "$TRG" ${1} < /dev/null > /dev/null &
  PROC=$!
  sleep $THREE_WINKS
  echo "${1},${RP}" | nc -w $TIMEOUT2 -r "$TRG" $IDPORT 2>&1
  sleep $TWO_WINKS

# Est-ce que ceci ressemble à un mauvais script... ?
# Commentaires de l'auteur du guide ABS : "Ce n'est pas réellement si mauvais,
#+                             en fait, plutôt intelligent."

  kill -HUP $PROC
  RP=`expr ${RP} + 1`
```

```
  shift
done

exit $?

#  Notes :
#  ------

#  Essayez de commenter la ligne 30 et d'exécuter ce script
#+ avec "localhost.localdomain 25" comme arguments.

#  Pour plus de scripts d'exemples 'nc' d'Hobbit,
#+ regardez dans la documentation :
#+ le répertoire /usr/share/doc/nc-X.XX/scripts.
```

Et, bien sûr, il y a le fameux script en une ligne du Dr. Andrew Tridgell dans l'affaire BitKeeper :

```
echo clone | nc thunk.org 5000 > e2fsprogs.dat
```

free

Affiche l'utilisation de la mémoire et du cache sous forme de tableau. La sortie de cette commande tend à être analysée avec grep, awk634 ou **Perl**. La commande **procinfo** affiche toute l'information dont dispose la commande **free** et bien plus encore.

```
bash$ free
             total       used       free     shared    buffers     cached
   Mem:         30504      28624       1880      15820       1608       16376
   -/+ buffers/cache:      10640      19864
   Swap:        68540       3128      65412
```

Pour afficher la mémoire RAM inutilisée :

```
bash$ free | grep Mem | awk '{ print $4 }'
1880
```

procinfo

Extrait et affiche des informations et des statistiques à partir du pseudo système de fichiers `/proc`399. Cela donne une liste très détaillée.

```
bash$ procinfo | grep Bootup
Bootup: Wed Mar 21 15:15:50 2001    Load average: 0.04 0.21 0.34 3/47 6829
```

lsdev

Affiche les périphériques, c'est-à-dire le matériel installé.

```
bash$ lsdev
Device            DMA   IRQ  I/O Ports
------------------------------------------------
cascade             4     2
dma                          0080-008f
dma1                         0000-001f
dma2                         00c0-00df
fpu                          00f0-00ff
ide0                    14   01f0-01f7 03f6-03f6
...
```

du

Affiche l'utilisation du disque de façon récursive. Par défaut, il prend en compte le répertoire courant.

```
bash$ du -ach
1.0k    ./wi.sh
1.0k    ./tst.sh
1.0k    ./random.file
6.0k    .
6.0k    total
```

df

Affiche l'utilisation des systèmes de fichiers sous forme de tableau.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda5               273262     92607    166547  36% /
/dev/hda8               222525    123951     87085  59% /home
/dev/hda7              1408796   1075744    261488  80% /usr
```

dmesg

Affiche tous les messages de démarrage du système envoyés à `stdout`. Pratique pour déboguer, pour s'assurer des pilotes de périphériques installés et des interruptions système utilisées. Bien sûr, la sortie de **dmesg** pourrait être analysée avec grep, sed631, ou awk634 à l'intérieur d'un script.

```
bash$ dmesg | grep hda
Kernel command line: ro root=/dev/hda2
hda: IBM-DLGA-23080, ATA DISK drive
hda: 6015744 sectors (3080 MB) w/96KiB Cache, CHS=746/128/63
hda: hda1 hda2 hda3 < hda5 hda6 hda7 > hda4
```

stat

Donne des *stat*istiques détaillées, voire verbeuses, sur un fichier donné (voire un répertoire ou un fichier périphérique) ou sur un ensemble de fichiers.

```
bash$ stat test.cru
  File: "test.cru"
   Size: 49970        Allocated Blocks: 100          Filetype: Regular File
   Mode: (0664/-rw-rw-r--)         Uid: (  501/ bozo)  Gid: (  501/ bozo)
 Device:  3,8   Inode: 18185     Links: 1
 Access: Sat Jun  2 16:40:24 2001
 Modify: Sat Jun  2 16:40:24 2001
 Change: Sat Jun  2 16:40:24 2001
```

Si le fichier cible n'existe pas, **stat** renvoie un message d'erreur.

```
bash$ stat fichier-inexistant
nonexistent-file: No such file or directory
```

Dans un script, vous pouvez utiliser **stat** pour extraire des informations sur les fichiers (et les systèmes de fichiers) et pour configurer des variables suivant le résultat.

```
#!/bin/bash
# fileinfo2.sh

# D'après une suggestion de Joël Bourquard et...
# http://www.linuxquestions.org/questions/showthread.php?t=410766


FICHIER=testfile.txt
nom=$(stat -c%n "$FICHIER")   # Identique à "$FICHIER" bien sûr.
proprietaire=$(stat -c%U "$FICHIER")
taille=$(stat -c%s "$FICHIER")
#  Certainement plus simple que d'utiliser "ls -l $FICHIER"
#+ et d'analyser la chaîne en retour avec sed.
inode=$(stat -c%i "$FICHIER")
type=$(stat -c%F "$FICHIER")
droits=$(stat -c%A "$FICHIER")

echo "Nom:          $nom"
echo "Propriétaire: $proprietaire"
echo "Taille:       $taille"
echo "Inode:        $inode"
```

```
echo "Type:         $type"
echo "Droits:       $droits"

exit 0

sh fileinfo2.sh

Nom:          testfile.txt
Propriétaire: bozo
Taille:       418
Inode:        1730378
Type:         regular file
Droits:       -rw-rw-r--
```

vmstat

Affiche les statistiques concernant la mémoire virtuelle.

```
bash$ vmstat
   procs                      memory    swap          io system         cpu
 r  b  w   swpd   free   buff  cache  si  so    bi    bo   in    cs  us  sy id
 0  0  0      0  11040   2636  38952   0   0    33     7  271    88   8   3 89
```

netstat

Affiche des informations et des statistiques sur le réseau, telles que les tables de routage et les connexions actives. Cet utilitaire accède à l'information avec `/proc/net` (Chapitre 27, `/dev` et `/proc`). Voir l'Exemple 27.3, « État de la connexion ».

netstat -r est équivalent à route.

```
bash$ netstat
Active Internet connections (w/o servers)
 Proto Recv-Q Send-Q Local Address           Foreign Address         State
 Active UNIX domain sockets (w/o servers)
 Proto RefCnt Flags       Type       State         I-Node Path
 unix  11     [ ]         DGRAM                    906    /dev/log
 unix  3      [ ]         STREAM     CONNECTED     4514   /tmp/.X11-unix/X0
 unix  3      [ ]         STREAM     CONNECTED     4513
 . . .
```

Note

La commande **netstat -lptu** affiche les sockets399 qui écoutent sur les ports et les processus associés. Ceci est utile pour déterminer si un ordinateur a été compromis.

uptime

Affiche depuis quand le système est lancé ainsi que quelques autres statistiques.

```
bash$ uptime
10:28pm  up  1:57,  3 users,  load average: 0.17, 0.34, 0.27
```

Note

Une *moyenne de charge* de 1 ou moins indique que le système gère les processus immédiatement. Une moyenne de charge supérieure à 1 signifie que les processus sont placés dans une queue. Quand la moyenne de charge est supérieure à trois, alors les performances système sont significativement dégradées.

hostname

Affiche le nom d'hôte du système. Cette commande initialise le nom d'hôte dans un script de démarrage `/etc/rc.d` (`/etc/rc.d/rc.sysinit` ou similaire). C'est équivalent à **uname -n** et une contrepartie de la variable interne $HOSTNAME.

```
bash$ hostname
localhost.localdomain
```

```
bash$ echo $HOSTNAME
localhost.localdomain
```

Similaire à la commande **hostname**, il existe les commandes **domainname**, **dnsdomainname**, **nisdomainname** et **ypdomainname**. Utilisez-les pour afficher ou initialiser le DNS système ou le nom de domaine NIS/YP. Différentes options de **hostname** réalisent aussi ces fonctions.

hostid

Affiche un identifiant numérique (hexadécimal) sur 32 bits pour la machine hôte.

```
bash$ hostid
7f0100
```

Note

Cette commande récupère prétendument un numéro de série « unique » pour un système particulier. Certaines procédures d'enregistrement d'un produit utilisent ce numéro pour indiquer une licence utilisateur particulière. Malheureusement, **hostid** ne fait que renvoyer l'adresse réseau en hexadécimal avec quelques octets transposés.

L'adresse réseau d'une machine Linux typique ne se trouvant pas sur un réseau est disponible dans `/etc/hosts`.

```
bash$ cat /etc/hosts
127.0.0.1               localhost.localdomain localhost
```

Il arrive que la transposition de **127.0.0.1** soit **0.127.1.0**, ce qui donne en hexadécimal **007f0100**, l'équivalent exact de ce que renvoie **hostid**, ci-dessus. Il existe seulement quelques millions d'autres machines Linux avec ce même *hostid*.

sar

Appeller **sar** (System Activity Reporter) donne une indication minutée et très détaillée des statistiques système. L'« ancien » SCO a sorti **sar** en tant que logiciel OpenSource au mois de juin 1999.

Cette commande ne fait pas partie de la distribution UNIX de base mais peut être obtenue en tant que partie du package des *utilitaires sysstat*, écrit par *Sébastien Godard*.

```
bash$ sar
Linux 2.4.9 (brooks.seringas.fr)        09/26/03

10:30:00          CPU     %user     %nice   %system   %iowait     %idle
10:40:00          all      2.21     10.90     65.48      0.00     21.41
10:50:00          all      3.36      0.00     72.36      0.00     24.28
11:00:00          all      1.12      0.00     80.77      0.00     18.11
Average:          all      2.23      3.63     72.87      0.00     21.27

14:32:30          LINUX RESTART

15:00:00          CPU     %user     %nice   %system   %iowait     %idle
15:10:00          all      8.59      2.40     17.47      0.00     71.54
15:20:00          all      4.07      1.00     11.95      0.00     82.98
15:30:00          all      0.79      2.94      7.56      0.00     88.71
Average:          all      6.33      1.70     14.71      0.00     77.26
```

readelf

Affiche des informations et des statistiques sur un binaire *elf* indiqué. Cela fait partie du package *binutils*.

```
bash$ readelf -h /bin/bash
ELF Header:
   Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
   Class:                             ELF32
   Data:                              2's complement, little endian
```

```
   Version:                           1 (current)
   OS/ABI:                            UNIX - System V
   ABI Version:                       0
   Type:                              EXEC (Executable file)
   . . .
```

size

La commande **size [/chemin/vers/binaire]** donne les tailles des segments d'un exécutable binaire ou d'un fichier archive. C'est utile principalement pour les programmeurs.

```
bash$ size /bin/bash
   text    data     bss     dec     hex filename
  495971   22496   17392  535859   82d33 /bin/bash
```

Journal système

logger

Ajoute un message généré par l'utilisateur dans le journal système (`/var/log/messages`). Vous n'avez pas besoin d'être *root* pour appeler **logger**.

```
logger Instabilité en cours sur la connexion réseau à 23:10, le 21/05.
# Maintenant, lancez un 'tail /var/log/messages'.
```

En embarquant une commande **logger** dans un script, il est possible d'écrire des informations de débogage dans `/var/log/messages`.

```
logger -t $0 -i Trace sur la ligne "$LINENO".
# L'option "-t" spécifie la balise pour l'entrée du journal.
# L'option "-i" enregistre l'identifiant du processus.

# tail /var/log/message
# ...
# Jul  7 20:48:58 localhost ./test.sh[1712]: Trace sur la ligne 3.
```

logrotate

Cet utilitaire gère les journaux système, en utilisant une rotation, en les compressant, supprimant, et/ou en les envoyant par courrier électronique. Ceci empêche que `/var/log` soit rempli d'anciens journaux de traces. Habituellement, cron lance quotidiennement **logrotate**.

Ajouter une entrée appropriée dans `/etc/logrotate.conf` rend possible la gestion de journaux personnels ainsi que des journaux système.

Note

Stefano Falsetto a créé *rottlog*, qu'il considère être une version améliorée de **logrotate**.

Contrôle de job

ps

Statistiques sur les processus (*P*rocess *S*tatistics) : affiche les processus en cours d'exécution avec leur propriétaire et identifiant de processus (PID). Celui-ci est habituellement appelé avec les options `ax` et `aux`. Le résultat peut être envoyé via un tube à grep ou sed631 pour repérer un processus spécifique (voir l'Exemple 14.13, « Forcer une déconnexion » et l'Exemple 27.2, « Trouver le processus associé à un PID »).

```
bash$  ps ax | grep sendmail
295 ?    S   0:00 sendmail: accepting connections on port 25
```

Pour afficher les processus système en un format d'« arbre » graphique : **ps afjx** ou **ps ax --forest**.

pgrep, **pkill**

Combine la commande **ps** avec grep ou kill.

```
bash$ ps a | grep mingetty
2212 tty2     Ss+    0:00 /sbin/mingetty tty2
 2213 tty3     Ss+    0:00 /sbin/mingetty tty3
 2214 tty4     Ss+    0:00 /sbin/mingetty tty4
 2215 tty5     Ss+    0:00 /sbin/mingetty tty5
 2216 tty6     Ss+    0:00 /sbin/mingetty tty6
 4849 pts/2    S+     0:00 grep mingetty


bash$ pgrep mingetty
2212 mingetty
 2213 mingetty
 2214 mingetty
 2215 mingetty
 2216 mingetty
```

Comparer l'action de **pkill** avec celle de killall.

pstree

Affiche les processus en cours d'exécution avec le format « tree » (arbre). L'option `-p` affiche les PID ainsi que les noms des processus.

top

Affiche les processus les plus consommateurs de puissances avec un rafraîchissement permanent. L'option `-b` affiche en mode texte de façon à ce que la sortie puisse être analysée ou tout simplement récupérée à partir d'un script.

```
bash$ top -b
  8:30pm  up 3 min,  3 users,  load average: 0.49, 0.32, 0.13
 45 processes: 44 sleeping, 1 running, 0 zombie, 0 stopped
 CPU states: 13.6% user,  7.3% system,  0.0% nice, 78.9% idle
 Mem:    78396K av,   65468K used,   12928K free,       0K shrd,    2352K buff
 Swap:  157208K av,       0K used,  157208K free                   37244K cached

   PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
   848 bozo      17   0   996  996   800 R     5.6  1.2   0:00 top
     1 root       8   0   512  512   444 S     0.0  0.6   0:04 init
     2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
   ...
```

nice

Lance un job en tâche de fond avec une priorité modifiée. Les priorités vont de 19 (le plus bas) à -20 (le plus haut). Seul *root* peut configurer les priorités négatives (les plus hautes). Les commandes en relation sont **renice**, **snice** qui modifie la priorité d'un processus en cours d'exécution, et **skill** qui envoie un signal kill à un ou plusieurs processus.

nohup

Conserve l'exécution d'une commande même si l'utilisateur se déconnecte. La commande s'exécutera en tant que tâche de fond sauf si il est suivi d'un &. Si vous utilisez **nohup** à l'intérieur d'un script, considérez le fait de le placer avec un wait pour éviter la création d'un processus *orphelin* ou zombie.

pidof

Identifie l'*identifiant du processus (PID)* d'un job en cours d'exécution. Comme les commandes de contrôle de job, telles que kill et renice297 qui agissent sur le *PID* d'un processus (et non pas son nom), il est parfois nécessaire d'identifier ce *PID*. La commande **pidof** est la contrepartie approximative de la variable interne $PPID.

```
bash$ pidof xclock
880
```

Exemple 16.6. *pidof* aide à la suppression d'un processus

```
#!/bin/bash
# kill-process.sh

SANSPROCESSUS=2

processus=xxxyyyzzz  # Utilise un processus inexistant.
# Pour les besoins de la démo seulement...
# ... je ne veux pas réellement tuer un processus courant avec ce script.
#
# Si, par exemple, vous voulez utiliser ce script pour vous déconnecter d'Internet,
#     processus=pppd

t=`pidof $processus` # Trouve le pid (process id) de $processus.
#  Le pid est nécessaire pour 'kill' (vous ne pouvez pas lancer 'kill' sur un nom de
#+ programme).

if [ -z "$t" ]       # Si le processus n'est pas présent, 'pidof' renvoie null.
then
  echo "Le processus $processus n'est pas lancé."
  echo "Rien n'a été tué."
  exit $SANSPROCESSUS
fi

kill $t              # Vous pouvez avoir besoin d'un 'kill -9' pour les processus
fils.

# Une vérification sur l'existence du processus est nécessaire ici.
# Peut-être un autre " t=`pidof $processus` " ou...


# Ce script entier pourrait être remplacé par
#    kill $(pidof -x processus)
# ou
#    killall processus
# mais cela ne serait pas aussi instructif.

exit 0
```

fuser

Identifie les processus (par PID) accédant à un fichier donné, à un ensemble de fichiers ou à un répertoire. Pourrait aussi être appelé avec l'option `-k`, qui tue ces processus. Ceci a des implications intéressantes pour la sécurité du système, spécialement avec des scripts empêchant des utilisateurs non autorisés d'accèder à certains services système.

```
bash$ fuser -u /usr/bin/vim
/usr/bin/vim:         3207e(bozo)



bash$ fuser -u /dev/null
/dev/null:            3009(bozo)  3010(bozo)  3197(bozo)  3199(bozo)
```

Une application importante de **fuser** arrive lors de l'insertion ou de la suppression physique d'un média de stockage, tel qu'un CDRom ou qu'une clé USB. Quelque fois, lancer un umount échoue avec un message d'erreur device is busy (NdT : le périphérique est occupé). Ceci signifie que des utilisateurs et/ou processus accèdent au périphérique. Une commande **fuser -um /dev/device_name** fera disparaître le mystère de façon à ce que vous puissiez supprimer les processus en question.

```
bash$ umount
/mnt/cleusb
umount: /mnt/usbdrive: device is busy



bash$ fuser -um /mnt/cleusb
/mnt/cleusb:        1772c(bozo)
```

```
bash$ kill -9 1772
bash$ umount /mnt/usbdrive
```

La commande **fuser**, appelé avec l'option `-n`, identifie les processus accèdant à un *port*. Ceci est particulièrement utile en combinaison avec nmap.

```
root# nmap localhost.localdomain
PORT     STATE SERVICE
 25/tcp   open  smtp



root# fuser -un tcp 25
25/tcp:               2095(root)

root# ps ax | grep 2095 | grep -v grep
2095 ?        Ss     0:00 sendmail: accepting connections
```

cron

Planificateur de programmes d'administration, réalisant des tâches comme le nettoyage et la suppression des journaux système ainsi que la mise à jour de la base de données slocate. C'est la version *superutilisateur* de at (bien que chaque utilisateur peut avoir son propre fichier `crontab` modifiable avec la commande **crontab**). Il s'exécute comme un démon et exécute les entrées planifiées dans `/etc/crontab`.

Note

Quelques versions de Linux utilisent **crond**, la version de Matthew Dillon pour le **cron**.

Contrôle de processus et démarrage

init

La commande **init** est le parent de tous les processus. Appelé à l'étape finale du démarrage, **init** détermine le niveau d'exécution du système à partir de `/etc/inittab`. Appelé par son alias **telinit** et par *root* seulement.

telinit

Lien symbolique vers **init**, c'est un moyen de changer de niveau d'exécution, habituellement utilisé pour la maintenance système ou des réparations en urgence de systèmes de fichiers. Appelé uniquement par *root*. Cette commande peut être dangereuse - soyez certain de bien la comprendre avant de l'utiliser!

runlevel

Affiche le niveau d'exécution actuel et ancien, c'est-à-dire si le système a été arrêté (niveau `0`), était en mode simple-utilisateur (`1`), en mode multi-utilisateur (`2` ou `3`), dans X Windows (`5`) ou en redémarrage (`6`). Cette commande accède au fichier `/var/run/utmp`.

halt, **shutdown**, **reboot**

Ensemble de commandes pour arrêter le système, habituellement juste avant un arrêt.

service

Exécute ou arrête un *service* système. Les scripts de démarrage compris dans `/etc/init.d` et `/etc/rc.d` utilisent cette commande pour exécuter les services au démarrage.

```
root# /sbin/service iptables stop
Flushing firewall rules:                                   [  OK  ]
Setting chains to policy ACCEPT: filter                    [  OK  ]
Unloading iptables modules:                                [  OK  ]
```

Réseau

ifconfig

Configuration fine de l'*interface réseau*.

```
bash$ ifconfig -a
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:10 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:700 (700.0 b)  TX bytes:700 (700.0 b)
```

La commande **ifconfig** est bien plus utilisée au démarrage lors de la configuration des interfaces ou à l'arrêt lors d'un redémarrage.

```
# Astuces de code provenant de /etc/rc.d/init.d/network

# ...

# Vérification du réseau.
[ ${NETWORKING} = "no" ] && exit 0

[ -x /sbin/ifconfig ] || exit 0

# ...

for i in $interfaces ; do
  if ifconfig $i 2>/dev/null | grep -q "UP" >/dev/null 2>&1 ; then
    action "Arrêt de l'interface $i: " ./ifdown $i boot
  fi
#  L'option spécifique GNU "-q" de "grep" signifie "silencieux",
#+ c'est-à-dire sans sortie.
#  Du coup, rediriger la sortie vers /dev/null n'est plus nécessaire.

# ...

echo "Périphériques actuellement actifs :"
echo `/sbin/ifconfig | grep ^[a-z] | awk '{print $1}'`
#                            ^^^^^  dévrait être entre guillemets pour éviter un
remplacement.
#  Ce qui suit fonctionne aussi.
#    echo $(/sbin/ifconfig | awk '/^[a-z]/ { print $1 })'
#    echo $(/sbin/ifconfig | sed -e 's/ .*//')
#  Merci, S.C., pour les commentaires supplémentaires.
```

Voir aussi l'Exemple 29.6, « Nettoyage après un **Control-C** ».

iwconfig

Ceci est un ensemble de commandes pour configurer un réseau sans-fil. C'est l'équivalent sans-fil de **ifconfig**.

ip

Outil généraliste pour configurer, modifier et analyser les réseaux *IP* (Internet Protocol) et les périphériques attachées. Cette commande fait partie du paquet *iproute2*.

```
bash$ ip link show
1: lo: <LOOPBACK,UP> mtu 16436 qdisc noqueue
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
 2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc pfifo_fast qlen 1000
     link/ether 00:d0:59:ce:af:da brd ff:ff:ff:ff:ff:ff
 3: sit0: <NOARP> mtu 1480 qdisc noop
     link/sit 0.0.0.0 brd 0.0.0.0


bash$ ip route list
169.254.0.0/16 dev lo  scope link
```

Ou, dans un script :

```
#!/bin/bash
# Script de Juan Nicolas Ruiz
# Utilisé avec sa permission.

# Configurer (et stopper) un tunnel GRE.


# --- start-tunnel.sh ---

LOCAL_IP="192.168.1.17"
REMOTE_IP="10.0.5.33"
OTHER_IFACE="192.168.0.100"
REMOTE_NET="192.168.3.0/24"

/sbin/ip tunnel add netb mode gre remote $REMOTE_IP \
  local $LOCAL_IP ttl 255
/sbin/ip addr add $OTHER_IFACE dev netb
/sbin/ip link set netb up
/sbin/ip route add $REMOTE_NET dev netb

exit 0  ##############################################

# --- stop-tunnel.sh ---

REMOTE_NET="192.168.3.0/24"

/sbin/ip route del $REMOTE_NET dev netb
/sbin/ip link set netb down
/sbin/ip tunnel del netb

exit 0
```

route

Affiche des informations sur la façon de modifier la table de routage du noyau.

```
bash$ route
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
 pm3-67.bozosisp *               255.255.255.255 UH        40 0          0 ppp0
 127.0.0.0       *               255.0.0.0       U         40 0          0 lo
 default         pm3-67.bozosisp 0.0.0.0         UG        40 0          0 ppp0
```

chkconfig

Vérifie la configuration du réseau et du système. Cette commande affiche et gère les services réseau lancés au démarrage dans le répertoire `/etc/rc?.d`.

Originellement un port d'IRIX vers Red Hat Linux, **chkconfig** pourrait ne pas faire partie de l'installation principale des différentes distributions Linux.

```
bash$ chkconfig --list
atd             0:off   1:off   2:off   3:on    4:on    5:on    6:off
 rwhod           0:off   1:off   2:off   3:off   4:off   5:off   6:off
 ...
```

tcpdump

« Reniffleur » de paquets réseau. C'est un outil pour analyser et corriger le trafic sur un réseau par l'affichage des en-têtes de paquets correspondant à des critères précis.

Affiche le trafic des paquets ip entre l'hôte *bozoville* et *caduceus*:

```
bash$ tcpdump ip host bozoville and caduceus
```

Bien sûr, la sortie de **tcpdump** est analysable en utilisant certains utilitaires texte préalablement discutés.

Systèmes de fichiers

mount

Monte un système de fichier, généralement sur un périphérique externe, tel qu'un lecteur de disquette ou de CDROM. Le fichier `/etc/fstab` comprend tous les systèmes de fichiers, partitions et périphériques disponibles pouvant être montés manuellement ou automatiquement. Le fichier `/etc/mtab` affiche les systèmes de fichiers et partitions actuellement montés (en incluant les systèmes virtuels tels que `/proc`).

mount -a monte tous les systèmes de fichiers et partitions indiqués dans `/etc/fstab`, à l'exception de ceux disposant de l'option `noauto`. Au démarrage, un script de `/etc/rc.d` (`rc.sysinit` ou un similaire) appelle cette commande pour monter tout ce qui doit l'être.

```
mount -t iso9660 /dev/cdrom /mnt/cdrom
# Monte le CDROM. ISO 9660 est le standard des systèmes de fichiers pour les CDROM.
mount /mnt/cdrom
# Raccourci, à condition que /mnt/cdrom soit compris dans /etc/fstab
```

La commande souple *mount* peut même monter un fichier ordinaire sur un périphérique bloc et ce fichier agira comme si il était un système de fichiers. *mount* accomplit cela en associant le fichier à un périphérique loopback399. Une application de ceci est le montage et l'examen d'une image ISO9660 avant qu'elle ne soit gravée sur un CDR. [3]

Exemple 16.7. Vérifier une image

```
# En tant que root...

mkdir /mnt/cdtest  # Préparez un point de montage, s'il n'existe pas déjà.

mount -r -t iso9660 -o loop cd-image.iso /mnt/cdtest   # Montez l'image.
#                   l'option "-o loop" est équivalent à "losetup /dev/loop0"
cd /mnt/cdtest     # Maintenant, vérifiez l'image
ls -alR            # Listez les fichiers dans cette hiérarchie de répertoires.
                   # Et ainsi de suite.
```

umount

Démonte un système de fichiers actuellement montés. Avant de supprimer physiquement une disquette ou un CDROM monté au prélable, le périphérique doit être démonté (**umount**), sinon des corruptions du système de fichiers pourraient survenir.

```
umount /mnt/cdrom
# Vous pouvez maintenant appuyer sur le bouton d'éjection en toute sécurité.
```

Note

L'utilitaire **automount**, s'il est correctement installé, peut monter et démonter des disquettes et des CDROM s'ils sont utilisés ou enlevés. Sur des portables disposant de lecteurs de disquette et CDROM enlevables, ceci peut poser des problèmes.

gnome-mount

Les dernières distributions Linux ont rendu obsolètes les commandes **mount** et **umount**. Le successeur, pour le montage en ligne de commande des périphériques amovibles, est **gnome-mount**. Il accepte l'option `-d` pour monter un périphérique de `/dev`.

Par exemple, pour monter une clé USB :

```
bash$ gnome-mount -d /dev/sda1
gnome-mount 0.4
```

[3] Pour plus de détails sur la gravure de CDR, voir l'article d'Alex Withers, *Créer des CDs*, dans le numéro d'octobre 1999 du *Linux Journal*.

```
bash$ df
. . .
 /dev/sda1               63584     12034     51550  19% /media/disk
```

sync

Force une écriture immédiate de toutes les données mises à jour à partir des tampons vers le disque dur (synchronisation des lecteurs avec les tampons). Bien que cela ne soit pas strictement nécessaire, **sync** assure à l'administrateur système et à l'utilisateur que les données tout juste modifiées survivront à une soudaine coupure de courant. Aux anciens temps, un **sync; sync** (deux fois, pour être absolument certain) était une mesure de précaution utile avant un redémarrage du système.

Quelque fois, vous pouvez forcer un vidage immédiat des tampons, comme lors de la suppression sécurisée d'un fichier (voir l'Exemple 15.59, « Effacer les fichiers de façon sûre ») ou lorsque les lumières commencent à clignotter.

losetup

Initialise et configure les périphériques loopback399.

Exemple 16.8. Création d'un système de fichiers dans un fichier

```
TAILLE=1000000  # 1 Mo

head -c $TAILLE < /dev/zero > fichier  # Initialise un fichier à la taille indiquée.
losetup /dev/loop0 fichier             # Le configure en tant que périphérique
loopback.
mke2fs /dev/loop0                      # Crée un système de fichiers.
mount -o loop /dev/loop0 /mnt          # Le monte.

# Merci, S.C.
```

mkswap

Crée une partition de swap ou un fichier. Du coup, l'aire de swap doit être activé avec **swapon**.

swapon, **swapoff**

Active/désactive la partition de swap ou le fichier. Ces commandes sont généralement utilisées au démarrage et à l'arrêt.

mke2fs

Crée un système de fichiers *ext2* Linux. Cette commande doit être appelée en tant que *root*.

Exemple 16.9. Ajoute un nouveau disque dur

```
#!/bin/bash

# Ajouter un deuxième disque dur au système.
# Configuration logiciel. Suppose que le matériel est déjà monté.
# A partir d'un article de l'auteur de ce document dans le numéro
# #38 de la "Linux Gazette", http://www.linuxgazette.com.

ROOT_UID=0     # Ce script doit être lancé en tant que root.
E_NOTROOT=67   # Erreur pour les utilisateurs non privilégiés.

if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Vous devez être root pour utiliser ce script."
  exit $E_NOTROOT
fi

# A utiliser avec beaucoup de précautions!
# Si quelque chose se passe mal, vous pourriez supprimer votre système de
#+ fichiers complet.


NOUVEAUDISQUE=/dev/hdb    # Suppose que /dev/hdb est disponible. A vérifier!
POINTMONTAGE=/mnt/newdisk # Ou choisissez un autre point de montage.
```

```
fdisk $NOUVEAUDISQUE1
mke2fs -cv $NOUVEAUDISQUE1 # Vérifie les mauvais blocs et rend la sortie verbeuse.
#  Note:    /dev/hdb1, *pas* /dev/hdb!
mkdir $POINTMONTAGE
chmod 777 $POINTMONTAGE   # Rend le nouveau disque accessible à tous les
utilisateurs.


# Maintenant, testez...
# mount -t ext2 /dev/hdb1 /mnt/newdisk
# Essayez de créer un répertoire.
# Si cela fonctionne, démontez-le et faites.

# Etape finale:
# Ajoutez la ligne suivante dans /etc/fstab.
# /dev/hdb1  /mnt/newdisk  ext2  defaults  1 1

exit 0
```

Voir aussi l'Exemple 16.8, « Création d'un système de fichiers dans un fichier » et l'Exemple 28.3, « Créer un disque ram ».

tune2fs

Configure finement le système de fichiers *ext2*. Peut être utilisé pour modifier les paramètres du système de fichiers, tels que le nombre maximum de montage. Il doit être utilisé en tant que *root*.

Avertissement

Cette commande est extrêmement dangereuse. Utilisez-la à vos propres risques, car vous pourriez détruire par inadvertance votre système de fichiers.

dumpe2fs

Affiche sur `stdout` énormément d'informations sur le système de fichiers. Elle doit aussi être appelée en tant que *root*.

```
root# dumpe2fs /dev/hda7 | grep 'ount count'
dumpe2fs 1.19, 13-Jul-2000 for EXT2 FS 0.5b, 95/08/09
Mount count:              6
Maximum mount count:      20
```

hdparm

Liste ou modifie les paramètres des disques durs. Cette commande doit être appelée en tant que *root* et peut être dangereuse si elle est mal utilisée.

fdisk

Crée ou modifie une table des partitions sur un périphérique de stockage, habituellement un disque dur. Cette commande doit être appelée en tant que *root*.

Avertissement

Utilisez cette commande avec d'infinies précautions. Si quelque chose se passe mal, vous pouvez détruire un système de fichiers existant.

fsck, **e2fsck**, **debugfs**

Ensemble de commandes de vérification, réparation et débogage des systèmes de fichiers.

fsck : une interface pour vérifier un système de fichiers UNIX (peut appeler d'autres utilitaires). Le type de système de fichiers est par défaut ext2.

e2fsck : vérificateur du système de fichiers ext2.

debugfs : débogueur du système de fichiers ext2. Une des utilités de cette commande souple, mais dangereuse, est de récupérer (ou plutôt d'essayer de récupérer) des fichiers supprimés. À réserver aux utilisateurs avancés !

Attention

Toutes ces commandes doivent être appelées en tant que *root* et peuvent endommager, voire détruire, un système de fichiers si elles sont mal utilisées.

badblocks

Vérifie les blocs défectueux (défauts physiques du média) sur un périphérique de stockage. Cette commande trouve son utilité lors du formatage d'un nouveau disque dur ou pour tester l'intégrité du média de sauvegarde. [4] Comme exemple, **badblocks /dev/fd0** teste une disquette.

La commande **badblocks** peut être appelé de façon destructive (écrasement de toutes les données) ou dans un mode lecture-seule non destructif. Si l'*utilisateur root* est le propriétaire du périphérique à tester, comme c'est le cas habituellement, alors *root* doit appeler cette commande.

lsusb, **usbmodules**

La commande **lsusb** affiche tous les bus USB (Universal Serial Bus) et les périphériques qui y sont raccordés.

La commande **usbmodules** affiche des informations sur les modules du pilote pour les périphériques USB connectés.

```
bash$ lsusb
Bus 001 Device 001: ID 0000:0000
 Device Descriptor:
   bLength                18
   bDescriptorType         1
   bcdUSB               1.00
   bDeviceClass            9 Hub
   bDeviceSubClass         0
   bDeviceProtocol         0
   bMaxPacketSize0         8
   idVendor           0x0000
   idProduct          0x0000

   . . .
```

lspci

Liste les bus *pci* présents.

```
bash$ lspci
00:00.0 Host bridge: Intel Corporation 82845 845
 (Brookdale) Chipset Host Bridge (rev 04)
 00:01.0 PCI bridge: Intel Corporation 82845 845
 (Brookdale) Chipset AGP Bridge (rev 04)
 00:1d.0 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #1) (rev 02)
 00:1d.1 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #2) (rev 02)
 00:1d.2 USB Controller: Intel Corporation 82801CA/CAM USB (Hub #3) (rev 02)
 00:1e.0 PCI bridge: Intel Corporation 82801 Mobile PCI Bridge (rev 42)

   . . .
```

mkbootdisk

Crée une disquette de démarrage pouvant être utilisée pour lancer le système si, par exemple, le MBR (master boot record) est corrumpu. L'option `--iso` est très intéressante car elle permet de demander à **mkisofs** la création d'une image *ISO9660* démarrable et utilisable pour graver un CD.

La commande **mkbootdisk** est en fait un script Bash, écrit par Erik Troan, et disponible dans le répertoire `/sbin`.

mkisofs

Crée un système de fichiers *ISO9660* convenable pour une image CD.

chroot

CHange ROOT directory (modifie le répertoire racine). Habituellement, les commandes sont récupérées à partir de $PATH

[4] L'option `-c` de mke2fs demande aussi une vérification des blocs défectueux.

depuis la racine `/`, le *répertoire racine* par défaut. Cette commande modifie le répertoire racine par un autre répertoire (et modifie aussi le répertoire de travail). Ceci est utile dans des buts de sécurité, par exemple lorsqu'un administrateur système souhaite restreindre certains utilisateurs notamment ceux utilisant telnet, pour sécuriser une partie du système de fichiers (c'est souvent assimilé à confiner un utilisateur invité dans une prison chroot (« chroot jail »)). Notez qu'après un **chroot**, le chemin d'exécution des binaires du système n'est plus valide.

Un **`chroot /opt`** ferait que toutes les références à `/usr/bin` seraient traduites en `/opt/usr/bin`. De même, **`chroot /aaa/bbb /bin/ls`** redirigerait tous les futurs appels à **ls** en `/aaa/bbb` comme répertoire de base, plutôt que `/` comme c'est habituellement le cas. Un **alias XX 'chroot /aaa/bbb ls'** dans le `~/.bashrc` d'un utilisateur restreint réllement la portion du système de fichiers où elle peut lancer des commandes.

La commande **chroot** est aussi pratique lors du lancement du disquette d'urgence (**chroot** vers `/dev/fd0`), ou comme option de **lilo** lors de la récupération après un crash système. D'autres utilisations incluent l'installation à partir d'un autre système de fichiers (une option rpm) ou le lancement d'un système de fichiers en lecture-seule à partir d'un CDROM. Ne peut s'appeller qu'en tant que *root*, et à utiliser avec précaution.

Attention

Il pourrait être nécessaire de copier certains fichiers système vers un répertoire compris dans le répertoire de base du *chroot*, car le `$PATH` n'est plus fiable.

lockfile

Cet utilitaire fait partie du package **procmail** (*www.procmail.org*). Il crée un *fichier de verrouillage*, un fichier *sémaphore* [5] qui contrôle l'accès à un fichier, périphérique ou ressource. Le fichier de verrouillage indique qu'un fichier, périphérique, ressource est utilisé par un processus particulier (« occupé ») et ne permet aux autres processus qu'un accès restreint (ou pas d'accès).

```
lockfile /home/bozo/verrous/$0.lock
# Crée un fichier de verrouillage protégé en écriture et préfixé avec le nom du
script.

lockfile /home/bozo/lockfiles/${0##*/}.lock
# Une version plus sûre de l'exemple ci-dessus, comme indiqué par E. Choroba.
```

Les fichiers de verrouillage sont utilisés par des applications pour protéger les répertoires de courriers électroniques des utilisateurs de modifications simultanées, pour indiquer qu'un port modem est utilisé ou pour montrer qu'une instance de Netscape utilise son cache. Les scripts peuvent vérifier l'existence d'un fichier de verrouillage créé par un certain processus pour vérifier si le processus existe. Notez que si un script essaie de créer un fichier de verrouillage déjà existant, le script a toutes les chances de se terminer précipitamment.

Habituellement, les applications créent et vérifient les fichiers de verrouillage dans le répertoire `/var/lock`. [6] Un script peut tester la présence d'un fichier de verrouillage de la façon suivante.

```
nomappl=xyzip
# L'application "xyzip" crée le fichier de verrouillage "/var/lock/xyzip.lock".

if [ -e "/var/lock/$nomappl.lock" ]
then   #+ Empêche les autres programmes et scripts
       #  d'accèder aux fichiers/ressources utilisés par xyzip.
  ...
```

flock

flock est bien moins utile que la commande **lockfile**. Elle configure un verrou « advisory » sur un fichier puis exécute une commande tant que le verrou est actif. Ceci permet d'empêcher un processus de configurer un verrou sur ce fichier jusqu'à la fin de l'exécution de la commande spécifiée.

```
flock $0 cat $0 > lockfile__$0
#  Configurer un verrou sur le script où cette ligne apparaît
#+ tout en envoyant le script sur stdout.
```

[5] Définition : Une *sémaphore* est un drapeau ou un signal. (L'usage date des chemins de fer où un drapeau coloré, une lanterne ou une *sémaphore* indiquait si une piste particulière était utilisée et du coup indisponible pour un autre train.) Un processus UNIX peut vérifier la sémaphore appropriée pour déterminer si une ressource particulière est disponible/accessible.
[6] Comme seul *root* a le droit d'écriture dans le répertoire `/var/lock`, un script utilisateur ne peut pas initialiser un fichier de verrouillage ici.

Note

Contrairement à **lockfile**, **flock** ne crée *pas* automatiquement un fichier de verrouillage.

mknod

Crée des fichiers de périphériques blocs ou caractères (peut être nécessaire lors de l'installation d'un nouveau matériel sur le système). L'outil **MAKEDEV** a virtuellement toutes les fonctionnalités de **mknod** et est plus facile à utiliser.

tmpwatch

Supprime automatiquement les fichiers qui n'ont pas été accédés depuis une certaine période. Appelé habituellement par cron pour supprimer les fichiers journaux.

MAKEDEV

Utilitaire pour la création des fichiers périphériques. Il doit être lancé en tant que *root* et dans le répertoire `/dev`. C'est en quelque sorte une version avancée de **mknod**.

tmpwatch

Supprime automatiquement les fichiers qui n'ont pas été utilisés (accédés) depuis une periode de temps spécifiée. Habituellement appelé par cron pour supprimer les vieux journaux de trace.

Backup

dump, **restore**

La commande **dump** est un utilitaire élaboré de sauvegarde du système de fichiers, généralement utilisé sur des grosses installations et du réseau. [7] Il lit les partitions brutes du disque et écrit un fichier de sauvegarde dans un format binaire. Les fichiers à sauvegarder peuvent être enregistrés sur un grand nombre de média de stockage incluant les disques et lecteurs de cassettes. La commande **restore** restaure les sauvegardes faites avec **dump**.

fdformat

Réalise un formatage bas-niveau sur une disquette (`/dev/fd0*`).

Ressources système

ulimit

Initialise une *limite supérieure* sur l'utilisation des ressources système. Habituellement appelé avec l'option `-f` qui initialise une limite sur la taille des fichiers (**ulimit -f 1000** limite les fichiers à un mégaoctet maximum). L'option `-t` limite la taille du coredump (**ulimit -c 0** élimine les coredumps). Normalement, la valeur de **ulimit** est configurée dans `/etc/profile` et/ou `~/.bash_profile` (voir l'Annexe G, Fichiers importants).

Important

Un emploi judicieux de **ulimit** peut protéger un système contre l'utilisation des *bombes fork*.

```
#!/bin/bash
# Ce script est inclus dans un but d'illustration seulement.
# Exécutez-le à vos risques et périls -- il BLOQUERA votre système.

while true  #  Boucle sans fin.
do
  $0 &      #  Ce script s'appelle lui-même . . .
            #+ un nombre infini de fois . . .
            #+ jusqu'à ce que le système se gèle à cause d'un manque de
ressources.
done        #  C'est le scénario notoire de l'<quote>apprentissage du
sorcier</quote>.

exit 0      #  Ne sortira pas ici car ce script ne terminera jamais.
```

[7] Les opérateurs de systèmes Linux simple utilisateur préfèrent généralement quelque chose de plus simple pour leur sauvegarde, comme **tar**.

> Un **ulimit -Hu XX** (où *XX* est la limite du nombre de processus par utilisateur) dans `/etc/profile` annulerait ce script lorsqu'il dépassera cette limite.

quota

Affiche les quotas disque de l'utilisateur ou du groupe.

setquota

Initialise les quotas disque pour un utilisateur ou un groupe à partir de la ligne de commande.

umask

Masque pour des droits de création d'un fichier utilisateur (*mask*). Limite les attributs par défaut d'un fichier pour un utilisateur particulier. Tous les fichiers créés par cet utilisateur prennent les attributs spécifiés avec **umask**. La valeur (octale) passée à **umask** définit les droits du fichiers *non actifs*. Par exemple, **umask 022** nous assure que les nouveaux fichiers auront tout au plus le droit 0755 (777 NAND 022). [8] Bien sûr, l'utilisateur peut ensuite modifier les attributs de fichiers spécifiques avec chmod. La pratique habituelle est d'initialiser la valeur de **umask** dans `/etc/profile` et/ou `~/.bash_profile` (voir l'Annexe G, Fichiers importants).

Exemple 16.10. Utiliser *umask* pour cacher un fichier en sortie

```
#!/bin/bash
# rot13a.sh
# Identique au script "rot13.sh" mais envoie la sortie dans un fichier sécurisé.

# Usage: ./rot13a.sh nomfichier
# ou     ./rot13a.sh <nomfichier
# ou     ./rot13a.sh et faites une saisie sur le clavier (stdin)

umask 177              #  Masque de création de fichier.
                       #  Les fichiers créés par ce script
                       #+ auront les droits 600.

FICHIERSORTIE=decrypted.txt
  #  Les résultats sont envoyés dans le fichier "decrypted.txt"
  #+ pouvant seulement être lus/écrits par l'utilisateur du script (ou root).

cat "$@" | tr 'a-zA-Z' 'n-za-mN-ZA-M' > $FICHIERSORTIE
#    ^^ Entrée provenant de stdin ou d'un fichier.
                                     # ^^^^^^^^^^^^^^ Sortie redirigée dans un
fichier.

exit 0
```

rdev

Obtenir des informations sur ou modifier le périphérique racine, l'espace swap ou le mode vidéo. La fonctionnalité de **rdev** a été principalement repris par **lilo**, mais **rdev** reste utile pour configurer un disque ram. C'est une commande dangereuse si elle est mal utilisée.

Modules

lsmod

Affiche les modules noyau installés.

```
bash$ lsmod
Module                  Size  Used by
 autofs                  9456   2 (autoclean)
 opl3                   11376   0
 serial_cs               5456   0 (unused)
```

[8] NAND est l'opérateur logique *not-and*. Son effet est similaire à la soustraction.

```
sb                     34752   0
uart401                 6384   0 [sb]
sound                  58368   0 [opl3 sb uart401]
soundlow                 464   0 [sound]
soundcore               2800   6 [sb sound]
ds                      6448   2 [serial_cs]
i82365                 22928   2
pcmcia_core            45984   0 [serial_cs ds i82365]
```

Note

Faire un **cat /proc/modules** donne la même information.

insmod

Force l'installation d'un module du noyau (utilise **modprobe** à la place lorsque c'est possible). Doit être appelé en tant que *root*.

rmmod

Force le déchargement d'un module du noyau. Doit être appelé en tant que *root*.

modprobe

Chargeur de modules normalement appelé à partir d'un script de démarrage. Doit être appelé en tant que *root*.

depmod

Crée un fichier de dépendances de module, appelé habituellement à partir d'un script de démarrage.

modinfo

Affiche des informations sur un module chargeable.

```
bash$ modinfo hid
filename:    /lib/modules/2.4.20-6/kernel/drivers/usb/hid.o
 description: "USB HID support drivers"
 author:      "Andreas Gal, Vojtech Pavlik <vojtech@suse.cz>"
 license:     "GPL"
```

Divers

env

Lance un programme ou un script avec certaines variables d'environnement initialisées ou modifiées (sans modifier l'environnement système complet). `[nomvariable=xxx]` permet la modification d'une variable d'environnement `nomvariable` pour la durée du script. Sans options spécifiées, cette commande affiche tous les paramétrages de variables d'environnement. [9]

Note

La première ligne d'un script (la ligne « #-! ») peut utiliser **env** lorsque le chemin vers le shell ou l'interpréteur est inconnu.

```
#! /usr/bin/env perl

print "Ce script Perl tournera,\n";
print "même si je ne sais pas où se trouve Perl.\n";

# Bon pour les scripts portables entre les platformes,
# où les binaires Perl pourraient être à l'endroit attendu.
# Merci, S.C.
```

[9] Dans Bash et d'autres dérivatifs du shell Bourne, il est possible d'initialiser des variables dans un environnement d'une seule commande.

```
var1=valeur1 var2=valeur2 commandeXXX
# $var1 et $var2 sont uniquement dans l'environnement de 'commandeXXX'.
```

Voire même...

```
#!/bin/env bash
# Utilise la variable d'environnement $PATH pour trouver l'emplacement de
bash.
# Du coup...
# Ce script fonctionnera quand Bash n'est pas à son emplacement habituel,
dans /bin.
...
```

ldd

Affiche les dépendances des bibliothèques partagées d'un exécutable.

```
bash$ ldd /bin/ls
libc.so.6 => /lib/libc.so.6 (0x4000c000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x80000000)
```

watch

Lance une commande plusieurs fois, à des intervalles de temps spécifiés.

Par défaut, il s'agit d'intervalles de deux secondes mais ceci est modifiable avec l'option `-n`.

```
watch -n 5 tail /var/log/messages
# Affiche la fin du journal du système, /var/log/messages, toutes les cinq secondes.
```

Note

Malheureusement, envoyer la sortie de **watch command** dans un tube qui renverra le tout à grep ne fonctionne pas.

strip

Supprime les références symboliques de débogage à partir d'un exécutable. Ceci réduit sa taille mais rend le débogage impossible.

Cette commande est fréquente dans un Makefile244 mais est bien plus rare dans un script shell.

nm

Affiche les symboles dans un binaire compilé sur lequel la commande strip n'a pas agi.

rdist

Client distant : synchronise, clone ou sauvegarde un système de fichiers sur un serveur distant.

16.1. Analyser un script système

En utilisant notre connaissance des commandes administratives, examinons un script système. Une des façons les plus courtes et les plus simples de comprendre les scripts est **killall**, [10] utilisée pour suspendre les processus en cours lors de l'arrêt du système.

Exemple 16.11. *killall*, à partir de `/etc/rc.d/init.d`

```
#!/bin/sh

# --> Commentaires ajoutés par l'auteur de ce document identifiés par "# -->".

# --> Ceci fait partie du paquetage de scripts 'rc'
# --> par Miquel van Smoorenburg, &lt;miquels@drinkel.nl.mugnet.org>.

# --> Ce script particulier semble être spécifique à Red Hat / FC
# --> (il pourrait ne pas être présent dans d'autres distributions).
```

[10] Le script système *killall* ne doit pas être confondu avec la commande killall de `/usr/bin`.

```
#  Arrête tous les services inutiles qui sont en cours d'exécution (ils ne
#+ devraient pas, donc il s'agit juste d'un test)

for i in /var/lock/subsys/*; do
        # --> Boucle for/in standard, mais comme "do" se trouve sur la même
        # --> ligne, il est nécessaire d'ajouter ";".
        # Vérifie si le script existe.
        [ ! -f $i ] && continue
        # --> C'est une utilisation intelligente d'une "liste et", équivalente
        # --> à: if [ ! -f "$i" ]; then continue

        # Obtient  le nom du sous-système.
        subsys=${i#/var/lock/subsys/}
        # --> Correspondance de nom de variable qui, dans ce cas, est le nom du
        # --> fichier. C'est l'équivalent exact de subsys=`basename $i`.

        # --> Il l'obtient du nom du fichier de verrouillage (si il existe un
        # --> fichier de verrou, c'est la preuve que le processus est en cours
        # --> d'exécution).
        # --> Voir l'entrée "lockfile", ci-dessus.


        # Arrête le sous-système.
        if [ -f /etc/rc.d/init.d/$subsys.init ]; then
            /etc/rc.d/init.d/$subsys.init stop
        else
            /etc/rc.d/init.d/$subsys stop
        # --> Suspend les jobs et démons en cours.
        # --> Notez que 'stop' est un paramètre de position, pas une commande
        # --> intégrée.
        fi
done
```

Ce n'était pas si mal. En plus d'un léger travail avec la correspondance de variables, il n'y a rien de plus ici.

Exercice 1. Dans `/etc/rc.d/init.d`, analysez le script **halt**. C'est un peu plus long que **killall** mais similaire dans le concept. Faites une copie de ce script quelque part dans votre répertoire personnel et expérimentez-le ainsi (ne le lancez *pas* en tant que *root*). Lancez-le simultanément avec les options `-vn` (**`sh -vn nomscript`**). Ajoutez des commentaires extensifs. Modifiez les commandes « action » en « echos ».

Exercice 2. Regardez quelques-uns des scripts les plus complexes dans `/etc/rc.d/init.d`. Regardez si vous comprenez certaines parties d'entre eux. Suivez la procédure ci-dessus pour les analyser. Pour plus d'indications, vous pouvez aussi examiner le fichier `sysvinitfiles` dans `/usr/share/doc/initscripts-?.??`, faisant partie de la documentation d'« initscripts ».

Partie Part 5. Thèmes avancés

À ce point, nous sommes prêt à nous enfoncer dans certains des aspects difficiles et inhabituelles de l'écriture de scripts. Tout au long du chemin, nous essaierons de « vous pousser » de plusieurs façons et d'examiner les *conditions limites* (qu'arrive-t'il lorsque nous entrons dans ce territoire inconnu ?).

Chapitre 17. Expressions rationnelles

... l'activité intellectuelle associée avec le développement de logiciels est à coup sûr d'un grand enrichissement.

-- Stowe Boyd

Pour utiliser complètement la puissance de la programmation par script shell, vous devez maîtriser les expressions rationnelles. Certaines commandes et utilitaires habituellement utilisés dans les scripts, tels que grep, expr, sed631 et awk634 interprètent et utilisent les ER. À partir de la version 3469, Bash possède son propre opérateur de correspondance d'expression rationnelle~=.

17.1. Une brève introduction aux expressions rationnelles

Une expression est une chaîne de caractères. Ces caractères qui ont une interprétation en dehors de leur signification littérale sont appelés des *méta caractères*. Par exemple, un symbole entre guillemets peut dénoter la parole d'une personne, *ditto*, ou une méta-signification [1] pour les symboles qui suivent. Les expressions rationnelles sont des ensembles de caractères et/ou méta-caractères qui correspondent ou spécifient des modèles.

Une expression rationnelle contient un élément ou plus parmi les suivants :

- Un *ensemble de caractères*. Ces caractères conservent leur signification littérale. Le type le plus simple d'expression rationnelle consiste en *seulement* un ensemble de caractères, sans métacaractères.
- Une *ancre*. Elles désignent la position dans la ligne de texte à laquelle doit correspondre l'ER. Par exemple, ^ et $ sont des ancres.
- *Modificateurs*. Ils étendent ou réduisent l'ensemble de texte auquel l'ER doit correspondre. Les modificateurs incluent l'astérisque, les crochets et l'antislash.

Les principales utilisations des expressions rationnelles (*ER*) sont la recherche de texte ou la manipulation de chaînes. Une ER *correspond* à un seul caractère ou à un ensemble de caractères (une sous-chaîne ou une chaîne complète).

- L'astérisque -- * -- correspond à toute répétition de caractères d'une chaîne ou d'une ER la précédant, incluant *zéro* caractère.

  « 1133* » correspond à *11 + un ou plus de 3* : *113*, *1133*, *1133333* et ainsi de suite.

- Le *point* -- . -- correspond à un seul caractère, sauf le retour à la ligne. [2]

  « 13. » correspond à *13 + au moins un caractère (incluant une espace)*: *1133*, *11333* mais pas *13* (un caractère supplémentaire manquant).

  Voir Exemple 15.18, « Solutionneur de mots croisés » pour une démonstration de la correspondance par *un point (un seul caractère)*.

- La puissance -- ^ -- correspond au début d'une ligne mais, quelque fois, suivant le contexte, inverse la signification d'un ensemble de caractères dans une ER.

- Le signe dollar, $, à la fin d'une ER correspond à la fin d'une ligne.

  « XXX$ » correspond à XXX à la fin d'une ligne.

- « $ » correspond à des lignes blanches.

- Les crochets -- [...] -- englobent un ensemble de caractères pour réaliser une correspondance dans une seule ER.

  « [xyz] » correspond aux caractères *x*, *y* ou *z*.

  « [c-n] » correspond à tout caractère compris entre *c* et *n*.

  « [B-Pk-y] » correspond à tout caractère compris entre *B* et *P* et entre *k* et *y*.

  « [a-z0-9] » correspond à toute lettre en minuscule et à tout chiffre.

  « [^b-d] » correspond à tous les caractères *sauf* ceux compris entre *b* et *d*. Ceci est un exemple de l'inversion de la signification de l' ER suivante grâce à l'opérateur ^ (prenant le même rôle que ! dans un contexte différent).

[1] Une *méta-signification* est un sens additionnel ou différent d'un caractère ou d'un groupe de caractères. Par exemple, la signification *littérale* d'une *expression rationnelle* est une expression ordinaire qui se conforme à l'usage accepté. La *méta-signification* est drastiquement différent. C'est discuté en longueur dans ce chapitre.

[2] Comme sed631 et awk634 travaillent ligne par ligne, il n'y aura habituellement pas de retour à la ligne à chercher. Dans les cas où il existerait un retour à la ligne dans une expression à plusieurs lignes, le point correspondra au retour à la ligne.

```
#!/bin/bash

sed -e 'N;s/.*/[&]/' << EOF   # Document en ligne
ligne1
ligne2
EOF
# SORTIE:
# [ligne1
# ligne2]


echo

awk '{ $0=$1"\n" $2; if (/ligne 1/) {print}}' <<- EOF
ligne 1
ligne 2
EOF
# SORTIE:
# ligne
# 1


# Merci, S.C.

exit 0
```

Les séquences combinées de caractères entre crochets correspondent à des modèles de mots communs. « [Yy][Ee][Ss] » correspond à *yes*, *Yes*, *YES*, *yEs* et ainsi de suite. « [0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9] » correspond à tout numéro de sécurité sociale (NdT : du pays d'origine de l'auteur).

- L'antislash -- \ -- échappe39 un caractère spécial, ce qui signifie que le caractère est interprété littéralement.

  Un « \$ » renvoie la signification littérale de « $ » plutôt que sa signification ER de fin de ligne. De même un « \\ » a la signification littérale de « \ ».

- Les signes « inférieur et supérieur » échappés39 -- \<...\> -- indiquent les limites du mot.

  Ces signes doivent être échappés, sinon ils n'ont que leur signification littérale.

  « \<le\> » correspond au mot « le » mais pas aux mots « les », « leur », « belle », etc.

```
bash$ cat fichiertexte
This is line 1, of which there is only one instance.
This is the only instance of line 2.
This is line 3, another line.
This is line 4.


bash$ grep 'the' fichiertexte
This is line 1, of which there is only one instance.
This is the only instance of line 2.
This is line 3, another line.


bash$ grep '\<the\>' fichiertexte
This is the only instance of line 2.
```

La seule façon d'être certain qu'une ER fonctionne est de la tester.

```
FICHIER DE TEST : fichiertest               # Pas de correspondance.
                                            # Pas de correspondance.
Lancer   grep "1133*"  sur ce fichier.      # Correspondance.
                                            # Pas de correspondance.
                                            # Pas de correspondance.
Cette ligne contient le nombre 113.         # Correspondance.
Cette ligne contient le nombre 13.          # Pas de correspondance.
Cette ligne contient le nombre 133.         # Pas de correspondance.
Cette ligne contient le nombre 1133.        # Correspondance.
Cette ligne contient le nombre 113312.      # Correspondance.
Cette ligne contient le nombre 1112.        # Pas de correspondance.
Cette ligne contient le nombre 113312312.   # Correspondance.
Cette ligne contient aucun nombre.          # Pas de correspondance.

bash$ grep "1133*" fichiertest
Lancez   grep "1133*"  sur ce fichier.       # Correspondance.
Cette ligne contient le nombre 113.          # Correspondance.
Cette ligne contient le nombre 1133.         # Correspondance.
Cette ligne contient le nombre 113312.       # Correspondance.
Cette ligne contient le nombre 113312312.    # Correspondance.
```

- **ER étendues.** Des méta-caractères supplémentaires ajoutés à l'ensemble de caractères. Utilisées dans egrep218, awk634 et Perl?.

- Le point d'interrogation -- ? -- correspond à aucune ou une instance de la précédente ER. Il est généralement utilisé pour correspondre à des caractères uniques.

- Le signe plus -- + -- correspond à un ou plus de la précédente ER. Il joue un rôle similaire à *, mais ne correspond *pas* à zéro occurrence.

```
# Les versions GNU de sed et awk peuvent utiliser "+",
# mais il a besoin d'être échappé.

echo a111b | sed -ne '/a1\+b/p'
echo a111b | grep 'a1\+b'
echo a111b | gawk '/a1+b/'
# Tous sont équivalents.

# Merci, S.C.
```

- Les « accolades » échappées39 -- \{ \} -- indiquent le nombre d'occurrences à filtrer par une précédente ER.

  Il est nécessaire d'échapper les accolades car, sinon, elles ont leur signification littérale. Cette usage ne fait techniquement pas partie de l'ensemble des ER de base.

  « [0-9]\{5\} » correspond exactement à cinq entiers (caractères entre 0 et 9).

Note

Les accolades ne sont pas disponibles comme ER dans la version « classique » (non conforme à POSIX) de awk634. Néanmoins**gawk** dispose de l'option `--re-interval` qui les autorise (sans être échappés).

```
bash$ echo 2222 | gawk --re-interval '/2{3}/'
2222
```

Perl et quelques versions de **egrep** ne nécessitent pas les accolades échappées.

- Les parenthèses -- **( )** -- délimitent des groupes d'ER. Elles sont utiles avec l'opérateur « | » et lors de l'extraction de sous-chaînes en utilisant expr.

- L'opérateur d'ER « ou » -- **|** -- correspond à n'importe lequel d'un ensemble de caractères constituant l'alternative.

```
bash$ egrep 're(a|e)d' misc.txt
People who read seem to be better informed than those who do not.
 The clarinet produces sound by the vibration of its reed.
```

Note

Quelques versions de **sed**, **ed** et **ex** supportent les versions échappées des expressions rationnelles étendues décrites ci-dessus, comme le font les outils GNU.

- **Classes de caractères POSIX. `[:class:]`**

  Ceci est une autre façon de spécifier un intervalle de caractères à filtrer.

- **`[:alnum:]`** correspond aux caractères alphabétiques et numériques. Ceci est équivalent à **`A-Za-z0-9`**.

- **`[:alpha:]`** correspond aux caractères alphabétiques. Ceci est équivalent à **`A-Za-z`**.

- **`[:blank:]`** correspond à une espace ou à une tabulation.

- **`[:cntrl:]`** correspond aux caractères de contrôle.

- **`[:digit:]`** correspond aux chiffres (décimaux). Ceci est équivalent à **`0-9`**.

[:graph:] (caractères graphiques affichables). Correspond aux caractères compris entre ASCII 33 - 126. Ceci est identique à **[:print:]**, ci-dessous, mais exclut le caractère espace.

- **[:lower:]** correspond aux caractères alphabétiques minuscules. Ceci est équivalent à **a-z**.
- **[:print:]** (caractères imprimables). Correspond aux caractères compris entre ASCII 32 - 126. C'est identique à **[:graph:]**, ci-dessus, mais en ajoutant le caractère espace.
- **[:space:]** correspond à toute espace blanche (espace et tabulation horizontale).
- **[:upper:]** correspond à tout caractère alphabétique majuscule. Ceci est équivalent à **A-Z**.
- **[:xdigit:]** correspond aux chiffres hexadécimaux. Ceci est équivalent à **0-9A-Fa-f**.

Important

Les classes de caractères POSIX nécessitent généralement d'être protégées ou entre doubles crochets50 ([[ ]]).

```
bash$ grep [[:digit:]] fichier.test
abc=723
```

Ces classes de caractères pourraient même être utilisées avec le remplacement, jusqu'à un certain point.

```
bash$ ls -l ?[[:digit:]][[:digit:]]?
-rw-rw-r--    1 bozo  bozo         0 Aug 21 14:47 a33b
```

Pour voir les classes de caractères POSIX utilisées dans des script, référez-vous à l'Exemple 15.21, « *toupper* : Transforme un fichier en majuscule. » et l'Exemple 15.22, « *lowercase* : Change tous les noms de fichier du répertoire courant en minuscule. ».

Sed631, awk634 et Perl437, utilisés comme filtres dans des scripts, prennent des ER en arguments lorqu'une transformation, ou une analyse de fichiers ou de flux doit se faire. Voir l'Exemple A.12, « *behead*: Supprimer les en-têtes des courriers électroniques et des nouvelles » et l'Exemple A.17, « *tree*: Afficher l'arborescence d'un répertoire » pour des illustrations sur ceci.

La référence sur ce thème complexe est *Mastering Regular Expressions* de Friedl. *Sed & Awk* par Dougherty et Robbins donne aussi un traitement très lucide des ER. Voir la Bibliographie pour plus d'informations sur ces livres.

17.2. Remplacement

Bash lui-même ne reconnaît pas les expressions rationnelles. Dans les scripts, les commandes et utilitaires, tels que sed631 et awk634, interprètent les ER.

Bash *effectue bien* l'*expansion de noms de fichiers*. [3] Ce processus est aussi connu sous le nom de « globbing » (NdT : remplacement) mais ceci n'utilise *pas* les ER standards. À la place, le remplacement reconnaît et étend les jokers. Le remplacement interprète les caractères jokers standards (* et ?), les listes de caractères entre crochets et certains autres caractères spéciaux (tels que ^ pour inverser le sens d'une correspondance). Néanmoins, il existe d'importantes limitations sur les caractères jokers dans le remplacement. Les chaînes contenant * ne correspondront pas aux noms de fichiers commençant par un point, comme par exemple `.bashrc`. [4] De même, le *?* a un sens différent dans le cadre du remplacement et comme partie d'une ER.

[3] L'*expansion de noms de fichiers* interprète les caractères spéciaux afin d'étendre aux noms de fichiers qui concordent avec le patron donné. Par exemple, `exemple.???` pourrait être étendu à `exemple.001` et/ou `exemple.txt`.

[4] L'expansion de noms de fichiers *peut* faire des correspondances avec les fichiers commençant par un point, mais seulement si le modèle inclut spécifiquement le point comme caractère littéral.

```
~/[.]bashrc    # N'étendra pas en ~/.bashrc
~/?bashrc      # Là non plus.
               # Les caractères jokers et autres métacaractères ne s'étendront
               # PAS en un point lors d'un remplacement.

~/.[b]ashrc    # Sera étendu en ~./bashrc
~/.ba?hrc      # Ici ausi.
~/.bashr*      # De même.

# Activer l'option "dotglob" désactive ceci.

# Merci, S.C.
```

```
bash$ ls -l
total 2
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l t?.sh
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh

bash$ ls -l [ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1

bash$ ls -l [a-c]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 a.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1

bash$ ls -l [^ab]*
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       466 Aug  6 17:48 t2.sh
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt

bash$ ls -l {b*,c*,*est*}
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 b.1
-rw-rw-r--    1 bozo  bozo         0 Aug  6 18:42 c.1
-rw-rw-r--    1 bozo  bozo       758 Jul 30 09:02 test1.txt
```

Bash réalise une expansion du nom de fichier sur des arguments sans guillemets. La commande echo le démontre.

```
bash$ echo *
a.1 b.1 c.1 t2.sh test1.txt

bash$ echo t*
t2.sh test1.txt
```

Note

Il est possible de modifier la façon dont Bash interprète les caractères spéciaux lors du remplacement. Une commande **set -f** désactive le remplacement, et les options `nocaseglob` et `nullglob` de shopt modifient le comportement du remplacement.

Voir aussi l'Exemple 10.4, « Agir sur des fichiers à l'aide d'une boucle *for* ».

Chapitre 18. Documents en ligne

Ici et maintenant, les gars.

-- Aldous Huxley, *Islande*

Un *document en ligne* est un bloc de code à usage spécial. Il utilise une forme de redirection d'E/S pour fournir une liste de commande à un programme ou une commande interactif, tel que ftp, cat ou l'éditeur de texte *ex*.

```
COMMANDE <<DesEntreesIci
...
DesEntreesIci
```

Une *chaîne de caractères de limite* encadre la liste de commandes. Le symbole spécial << désigne la chaîne de caractères de limite. Ceci a pour effet de rediriger la sortie d'un fichier vers le `stdin` d'un programme ou d'une commande. Ceci est similaire à **`programme-interactif < fichier-commandes`**, où `fichier-commandes` contient

```
commande n°1
commande n°2
...
```

L'alternative au *document en ligne* ressemble à ceci :

```
#!/bin/bash
programme-interactif <<ChaineLimite
commande #1
commande #2
...
ChaineLimite
```

Choisissez une *chaîne de caractères de limite* suffisamment inhabituelle pour qu'elle ne soit pas présente où que ce soit dans la liste de commandes afin qu'aucune confusion ne puisse survenir.

Notez que les *documents en ligne* peuvent parfois être utilisés correctement avec des utilitaires et des commandes non interactifs, tels que wall.

Exemple 18.1. *broadcast* : envoie des messages à chaque personne connectée

```
#!/bin/bash

wall <<zzz23EndOfMessagezzz23
Envoyez par courrier électronique vos demandes de pizzas à votre administrateur
système.
     (Ajoutez un euro supplémentaire pour les anchois et les champignons.)
# Un message texte supplémentaire vient ici.
# Note: Les lignes de commentaires sont affichées par 'wall'.
zzz23EndOfMessagezzz23

# Peut se faire plus efficacement avec
#         wall <fichier-message
#  Néanmoins, intégrer un message modèle dans un script
#+ est une solution rapide bien que sale

exit 0
```

Même de si improbables candidats comme *vi* tendent eux-même aux *documents en ligne*.

Exemple 18.2. *fichierstupide* : Crée un fichier stupide de deux lignes

```
#!/bin/bash

# Utilisation non interactive de 'vi' pour éditer un fichier.
# Émule 'sed'.
```

```
E_MAUVAISARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

FICHIERCIBLE=$1

# Insère deux lignes dans le fichier et le sauvegarde.
#--------Début document en ligne----------#
vi $FICHIERCIBLE <<x23LimitStringx23
i
Ceci est la ligne 1 du fichier exemple.
Ceci est la ligne 2 du fichier exemple.
^[
ZZ
x23LimitStringx23
#--------Fin   document en ligne----------#

#  Notez que ^[ ci-dessus est un échappement littéral, saisi avec
#+ Control-V <Esc>.

#  Bram Moolenaar indique que ceci pourrait ne pas fonctionner avec 'vim',
#+ à cause de problèmes possibles avec l'interaction du terminal.

exit 0
```

Le script ci-dessus pourrait avoir été implémenté aussi efficacement avec **ex**, plutôt que **vi**. Les *documents en ligne* contenant une liste de commandes **ex** sont assez courants pour disposer de leur propre catégorie, connue sous le nom de *scripts ex*.

```
#!/bin/bash
#  Remplace toutes les instances de "Smith" avec "Jones"
#+ dans les fichiers avec extension ".txt".

ORIGINAL=Smith
REMPLACEMENT=Jones

for mot in $(fgrep -l $ORIGINAL *.txt)
do
  # ------------------------------------
  ex $mot <<EOF
  :%s/$ORIGINAL/$REMPLACEMENT/g
  :wq
EOF
  # :%s est la commande de substitution d'"ex".
  # :wq est un raccourci pour deux commandes : sauvegarde puis quitte.
  # ------------------------------------
done
```

Les « scripts cat » sont analogues aux *scripts ex*.

Exemple 18.3. Message multi-lignes en utilisant *cat*

```
#!/bin/bash

#  'echo' est bien pour afficher des messages sur une seule ligne
#+ mais il est parfois problématique pour des blocs de message.
#  Un document en ligne style 'cat' permet de surpasser cette limitation.

cat <<Fin-du-message
------------------------------------
Ceci est la ligne 1 du message.
Ceci est la ligne 2 du message.
Ceci est la ligne 3 du message.
Ceci est la ligne 4 du message.
```

```
Ceci est la dernière ligne du message.
----------------------------------
Fin-du-message

#  le remplacement de la ligne 7, ci-dessus, par
#+   cat > $NouveauFichier <<Fin-du-message
#+       ^^^^^^^^^^
#+ écrit la sortie vers le fichier $NouveauFichier, au lieu de stdout.

exit 0


#-----------------------------------------
# Le code ci-dessous est désactivé à cause du "exit 0" ci-dessus.

# S.C. indique que ce qui suit fonctionne aussi.
echo "----------------------------------
Ceci est la ligne 1 du message.
Ceci est la ligne 2 du message.
Ceci est la ligne 3 du message.
Ceci est la ligne 4 du message.
Ceci est la dernière ligne du message.
----------------------------------"
#  Néanmoins, le texte ne pourrait  pas inclure les doubles guillemets sauf
#+ s'ils sont échappés.
```

L'option - marquant la chaîne de caractères de limite d'un document en ligne (**<<-ChaineLimite**) supprime les tabulations du début (mais pas les espaces) lors de la sortie. Ceci est utile pour réaliser un script plus lisible.

Exemple 18.4. Message multi-lignes, aves les tabulations supprimées

```
#!/bin/bash
# Identique à l'exemple précédent, mais...

#  L'option - pour un document en ligne <<-
#  supprime les tabulations du début dans le corps du document,
#+ mais *pas* les espaces.

cat &lt;&lt;-FINDUMESSAGE
        Ceci est la ligne 1 du message.
        Ceci est la ligne 2 du message.
        Ceci est la ligne 3 du message.
        Ceci est la ligne 4 du message.
        Ceci est la dernière ligne du message.
FINDUMESSAGE
# La sortie du script sera poussée vers la gauche.
# Chaque tabulation de chaque ligne ne s'affichera pas.

# Les cinq lignes du "message" sont préfacées par une tabulation, et non des espaces,
# Les espaces ne sont pas affectés par <<-  .

# Notez que cette option n'a aucun effet sur les tabulations *intégrées*.


exit 0
```

Un *document en ligne* supporte la substitution de paramètres et de commandes. Il est donc possible de passer différents paramètres dans le corps du document en ligne, en changeant la sortie de façon appropriée.

Exemple 18.5. Document en ligne avec une substitution de paramètre

```
#!/bin/bash
# Autre document en ligne 'cat' utilisant la substitution de paramètres.
```

```
# Essayez-le sans arguments,     ./scriptname
# Essayez-le avec un argument,   ./scriptname Mortimer
# Essayez-le avec deux arguments entre guillemets,
#                                ./scriptname "Mortimer Jones"

CMDLINEPARAM=1     # Attendez au moins un paramètre en ligne de commande.

if [ $# -ge $CMDLINEPARAM ]
then
  NOM=$1          #  Si plus d'un paramètre en ligne de commande, prendre
                  #+ seulement le premier.
else
  NOM="John Doe"  # Par défaut, s'il n'y a pas de paramètres.
fi

INTERLOCUTEUR="l'auteur de ce joli script"


cat <<FinDuMessage

Salut, $NOM.
Bienvenue à toi, $NOM, de la part de $INTERLOCUTEUR.

# Ce commentaire s'affiche dans la sortie (pourquoi ?).

FinDuMessage

# Notez que les lignes blanches s'affichent. Ainsi que le commentaire.

exit 0
```

Voici un script utile contenant un document intégré avec une substitution de paramètres.

Exemple 18.6. Télécharger un ensemble de fichiers dans le répertoire de récupération *Sunsite*

```
#!/bin/bash
# upload.sh

#  Téléchargement de fichiers par paires (Fichier.lsm, Fichier.tar.gz)
#+ pour le répertoire entrant de Sunsite (metalab.unc.edu).
#  Fichier.tar.gz est l'archive tar elle-même.
#  Fichier.lsm est le fichier de description.
#  Sunsite requiert le fichier "lsm", sinon cela retournera les contributions.

E_ERREURSARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` fichier_à_télécharger"
  exit $E_ERREURSARGS
fi


NomFichier=`basename $1`             # Supprime le chemin du nom du fichier.

Serveur="ibiblio.org"
Repertoire="/incoming/Linux"
#  Ils n'ont pas besoin d'être codés en dur dans le script,
#+ mais peuvent être changés avec un argument en ligne de commande.

MotDePasse="votre.adresse.courriel"   # A changer suivant vos besoins.

ftp -n $Serveur <<Fin-De-Session
# L'option -n désactive la connexion automatique

user anonymous "$MotDePasse"
binary
```

```
bell                # Sonne après chaque transfert de fichiers.
cd $Repertoire
put "$NomFichier.lsm"
put "$NomFichier.tar.gz"
bye
Fin-De-Session

exit 0
```

Mettre entre guillemets, ou échapper la « chaîne de caractères de limite » au début du document intgr, désactive la substitution de paramètres en son corps.

Exemple 18.7. Substitution de paramètres désactivée

```
#!/bin/bash
#  Un document en ligne 'cat', mais avec la substitution de paramètres
#+ désactivée.

NOM="John Doe"
INTERLOCUTEUR="l'auteur de ce joli script"

cat <<'FinDuMessage'

Salut, $NOM.
Bienvenue à toi, $NOM, de la part de $INTERLOCUTEUR.

FinDuMessage

#  Remplacement de la ligne 7, ci-dessus, avec
#+   cat > $Nouveaufichier <<Fin-du-message
#+       ^^^^^^^^^^
#+ écrit la sortie dans le fichier $Nouveaufichier, plutôt que sur stdout.

#  Pas de substitution de paramètres lorsque la chaîne de fin est entre
#+ guillemets ou échappée.
#  L'une des deux commandes ci-dessous à l'entête du document en ligne aura le
#+ le même effet.
#  cat <<"FinDuMessage"
#  cat <<\FinDuMessage

exit 0
```

Désactiver la substitution de paramètres permet d'afficher le texte littéral. Générer des scripts, ou même du code, en est une des utilités principales.

Exemple 18.8. Un script générant un autre script

```
#!/bin/bash
# generate-script.sh
# Basé sur une idée d'Albert Reiner.

FICHIER_SORTIE=genere.sh         # Nom du fichier à générer.


# ----------------------------------------------------------
# 'Document en ligne contenant le corps du script généré.
(
cat <<'EOF'
#!/bin/bash

echo "Ceci est un script shell généré"
#  Notez que, comme nous sommes dans un sous-shell,
#+ nous ne pouvons pas accéder aux variables du script "externe".
#  Prouvez-le...
```

```
echo "Le fichier généré aura pour nom : $FICHIER_SORTIE"
#  La ligne ci-dessus ne fonctionnera pas comme on pourrait s'y attendre
#+ parce que l'expansion des paramètres a été désactivée.
#  A la place, le résultat est une sortie littérale.

a=7
b=3

let "c = $a * $b"
echo "c = $c"

exit 0
EOF
) > $FICHIER_SORTIE
# ------------------------------------------------------------

#  Mettre entre guillemets la chaîne limite empêche l'expansion de la variable
#+ à l'intérieur du corps du document en ligne ci-dessus.
#  Ceci permet de sortir des chaînes littérales dans le fichier de sortie.

if [ -f "$FICHIER_SORTIE" ]
then
  chmod 755 $FICHIER_SORTIE
  # Rend le fichier généré exécutable.
else
        echo "Problème lors de la création du fichier: \"$FICHIER_SORTIE\""
fi

#  Cette méthode est aussi utilisée pour générer des programmes C, Perl, Python,
#+ Makefiles et d'autres.

exit 0
```

Il est possible d'initialiser une variable à partir de la sortie d'un document en ligne. En fait, il s'agit d'une forme dévié de substitution de commandes141.

```
variable=$(cat <<SETVAR
Cette variable
est sur plusieurs lignes.
SETVAR)

echo "$variable"
```

Un document en ligne peut donner une entrée à une fonction du même script.

Exemple 18.9. Documents en ligne et fonctions

```
#!/bin/bash
# here-function.sh

ObtientDonneesPersonnelles ()
{
  read prenom
  read nom
  read adresse
  read ville
  read etat
  read codepostal
} # Ceci ressemble vraiment à une fonction interactive, mais...


# Apporter l'entrée à la fonction ci-dessus.
ObtientDonneesPersonnelles <<ENREG001
Bozo
Bozeman
2726 Nondescript Dr.
Baltimore
```

```
MD
21226
RECORD001


echo
echo "$prenom $nom"
echo "$adresse"
echo "$ville, $etat $codepostal"
echo

exit 0
```

Il est possible d'utiliser : comme commande inactive acceptant une sortie d'un document en ligne. Cela crée un document en ligne « anonyme ».

Exemple 18.10. Document en ligne « anonyme »

```
#!/bin/bash

: <<TESTVARIABLES
${HOSTNAME?}${USER?}${MAIL?}  #  Affiche un message d'erreur
                              #+ si une des variables n'est pas configurée.
TESTVARIABLES

exit 0
```

Astuce

Une variante de la technique ci-dessus permet de « supprimer les commentaires » de blocs de code.

Exemple 18.11. Décommenter un bloc de code

```
#!/bin/bash
# commentblock.sh

: &lt;&lt;BLOC_COMMENTAIRE
echo "Cette ligne n'est pas un echo."
C'est une ligne de commentaire sans le préfixe "#".
Ceci est une autre ligne sans le préfixe "#".

&*@!!++=
La ligne ci-dessus ne causera aucun message d'erreur,
Parce que l'interpréteur Bash l'ignorera.
BLOC_COMMENTAIRE

echo "La valeur de sortie du \"BLOC_COMMENTAIRE\" ci-dessus est $?."   # 0
# Pas d'erreur.
echo


#  La technique ici-dessus est aussi utile pour mettre en commentaire un bloc
#+ de code fonctionnel pour des raisons de déboguage.
#  Ceci permet d'éviter de placer un "#" au début de chaque ligne, et d'avoir
#+ ensuite à les supprimer.

echo "Juste avant le bloc de code commenté."
#  Les lignes de code entre les lignes de soulignés doubles ne s'exécuteront pas.
#  ================================================================================
: &lt;&lt;DEBUGXXX
for fichier in *
do
  cat "$fichier"
```

```
done
DEBUGXXX
#  ===================================================================
echo "Juste après le bloc de code commenté."

exit 0



#####################################################################
#  Notez, néanmoins, que si une variable entre crochets est contenu
#+ dans un bloc de code commenté, cela pourrait poser problème.
#  Par exemple :


#/!/bin/bash

  : &lt;&lt;BLOC_COMMENTAIRE
  echo "Cette ligne ne s'affichera pas."
  &*@!!++=
  ${foo_bar_bazz?}
  $(rm -rf /tmp/foobar/)
  $(touch mon_repertoire_de_construction/cups/Makefile)
BLOC_COMMENTAIRE


$ sh commented-bad.sh
commented-bad.sh: line 3: foo_bar_bazz: parameter null or not set

#  Le remède pour ceci est de placer le BLOC_COMMENTAIRE
#+ entre guillemets simples à la ligne 48, ci-dessus.

  : &lt;&lt;'COMMENTBLOCK'

# Merci de nous l'avoir indiqué, Kurt Pfeifle.
```

Astuce

Encore une autre variante de cette sympathique astuce rendant possibles les scripts « auto-documentés ».

Exemple 18.12. Un script auto-documenté

```
#!/bin/bash
# self-document.sh : script auto-documenté
# Modification de "colm.sh".

DEMANDE_DOC=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # Demande de l'aide.
then
  echo; echo "Usage: $0 [nom-repertoire]"; echo
  sed --silent -e '/DOCUMENTATIONXX$/,/^DOCUMENTATIONXX$/p' "$0" |
  sed -e '/DOCUMENTATIONXX$/d'; exit $DEMANDE_DOC; fi


: <<DOCUMENTATIONXX
Liste les statistiques d'un répertoire spécifié dans un format de tabulations.
---------------------------------------------------------------------------
Le paramètre en ligne de commande donne le répertoire à lister.
Si aucun répertoire n'est spécifié ou que le répertoire spécifié ne peut être
lu, alors liste le répertoire courant.

DOCUMENTATIONXX

if [ -z "$1" -o ! -r "$1" ]
then
```

```
  repertoire=.
else
  repertoire="$1"
fi

echo "Liste de "$repertoire":"; echo
(printf "PERMISSIONS LIENS PROP GROUPE TAILLE MOIS  JOUR HH:MM NOM-PROG\n" \
; ls -l "$repertoire" | sed 1d) | column -t

exit 0
```

Utiliser un script cat319 est une autre façon d'accomplir ceci.

```
REQUETE_DOC=70

if [ "$1" = "-h"  -o "$1" = "--help" ]     # Demande d'aide.
then                                       # Utilise un "script cat"...
  cat <<DOCUMENTATIONXX
Liste les statistiques d'un répertoire spécifié au format de tableau.
-----------------------------------------------------------------------
Le paramètre en ligne de commande indique le répertoire à lister.
Si aucun répertoire n'est spécifié ou si le répertoire spécifié ne
peut pas être lu, alors liste le répertoire courant.

DOCUMENTATIONXX
exit $REQUETE_DOC
fi
```

Voir aussi l'Exemple A.30, « Identification d'un spammer » , Exemple A.40, « Pétales autour d'une rose », et Exemple A.41, « Quacky : un jeu de mots de type Perquackey » pour plus d'exemples de script auto-documenté.

Note

Les documents en ligne créent des fichiers temporaires mais ces fichiers sont supprimés après avoir été ouverts et ne sont plus accessibles par aucun autre processus.

```
bash$ bash -c 'lsof -a -p $$ -d0' << EOF
> EOF
lsof    1213 bozo    0r   REG    3,5    0 30386 /tmp/t1213-0-sh (deleted)
```

Attention

Quelques utilitaires ne fonctionneront pas à l'intérieur d'un *document en ligne*.

Avertissement

La *chaîne de limite* fermante, à la ligne finale d'un document en ligne, doit commencer à la position du tout *premier* caractère. Il ne peut pas y avoir d'*espace blanc devant*. Les espaces de fin après la chaîne de limite cause un comportement inattendu. L'espace blanc empêche la chaîne de limite d'être reconnu.

```
#!/bin/bash

echo "-----------------------------------------------------------------------"

cat <<ChaineLimite
echo "Ligne 1 du document en ligne."
echo "Ligne 2 du document en ligne."
echo "Ligne finale du document en ligne."
     ChaineLimite
#^^^^Chaîne de limite indentée. Erreur! Ce script ne va pas se comporter
comme
#+ on s'y attend.
```

```
echo "-------------------------------------------------------------------"

#  Ces commentaires sont en dehors du document en ligne et ne devraient pas
#+ s'afficher.

echo "En dehors du document en ligne."

exit 0

echo "Cette ligne s'affiche encore moins."  # Suit une commande 'exit'.
```

Pour ces tâches trop complexes pour un « document en ligne », considérez l'utilisation du langage de scripts **expect**, qui est conçu spécifiquement pour alimenter l'entrée de programmes interactifs.

18.1. Chaînes en ligne

Une *chaîne en ligne* peut être considéré comme une forme minimale du *document en ligne*. Il consiste simplement en la chaîne **COMMANDE** <<<**$MOT** où `$MOT` est étendu et est initialisé via l'entrée standard (`stdin`) de *`COMMANDE`*.

Comme exemple de base, considérez cette alternative à la construction echo-grep.

```
# Au lieu de :
if echo "$VAR" | grep -q txt   # if [[ $VAR = *txt* ]]
# etc.

# Try:
if grep -q "txt" <<< "$VAR"
then
   echo "$VAR contient la sous-chaîne \"txt\""
fi
# Merci pour la suggestion, Sebastian Kaminski.
```

Ou en combinaison avec read :

```
Chaine="Ceci est une chaîne de mots."

read -r -a Mots <<< "$Chaine"
#  L'option -a pour "lire" affecte les valeurs résultants
#+ aux membres d'un tableau.

echo "Le premier mot de Chaine est   :   ${Mots[0]}"   # Ceci
echo "Le deuxième mot de Chaine est  :   ${Mots[1]}"   # est
echo "Le troisième mot de Chaine est :   ${Mots[2]}"   # une
echo "Le quatrième mot de Chaine est :   ${Mots[3]}"   # chaîne
echo "Le cinquième mot de Chaine est :   ${Mots[4]}"   # de
echo "Le sixième mot de Chaine est   :   ${Mots[5]}"   # mots.
echo "Le septième mot de Chaine est  :   ${Mots[6]}"   # (null)
                                                 # On dépasse la fin de $Chaine.

# Merci à Francisco Lobo pour sa suggestion.
```

Exemple 18.13. Ajouter une ligne au début d'un fichier

```
#!/bin/bash
# prepend.sh: Ajoute du texte au début d'un fichier.
#
#  Exemple contribué par Kenny Stauffer,
#+ et légèrement modifié par l'auteur du document.


E_FICHIERINEXISTANT=65

read -p "Fichier : " fichier   # argument -p pour que 'read' affiche l'invite.
```

```
if [ ! -e "$fichier" ]
then   # Quitte si le fichier n'existe pas.
  echo "Fichier $fichier introuvable."
  exit $E_FICHIERINEXISTANT
fi

read -p "Titre : " titre
cat - $fichier <<<$titre > $fichier.nouveau

echo "Le fichier modifié est $fichier.nouveau"

exit 0

# provenant de 'man bash'
# Chaînes en ligne
#         Une variante des documents en ligne, le format est :
#
#                 <<<mot
#
#         Le mot est étendu et fourni à la commande sur son entrée standard.
```

Exemple 18.14. Analyser une boîte mail

```
#!/bin/bash
#  Script par Francisco Lobo,
#+ et légèrement modifié par l'auteur du guide ABS.
#  Utilisé avec sa permission dans le guide ABS (Merci !).

# Ce script ne fonctionnera pas avec les versions de Bash antérieures à la 3.0.


E_ARGS_MANQUANTS=67
if [ -z "$1" ]
then
  echo "Usage: $0 fichier-mailbox"
  exit $E_ARGS_MANQUANTS
fi

mbox_grep()  # Analyse le fichier mailbox.
{
    declare -i corps=0 correspondance=0
    declare -a date emetteur
    declare mail entete valeur


    while IFS= read -r mail
#         ^^^^                 Réinitialise $IFS.
#  Sinon, "read" supprimera les espaces devant et derrière sa cible.

   do
       if [[ $mail =~ "^From " ]]   # correspondance du champ "From" dans le message.
       then
          (( corps  = 0 ))           # Variables ré-initialisées.
          (( correspondance = 0 ))
          unset date

       elif (( corps ))
       then
            (( correspondance ))
            # echo "$mail"
            #  Décommentez la ligne ci-dessus si vous voulez afficher
            #+ le corps entier du message.

       elif [[ $mail ]]; then
          IFS=: read -r entete valeur <<< "$mail"
          #                          ^^^  "chaîne intégrée"
```

```
          case "$entete" in
          [Ff][Rr][Oo][Mm] ) [[ $valeur =~ "$2" ]] && (( correspondance++ )) ;;
          # correspondance de la ligne "From".
          [Dd][Aa][Tt][Ee] ) read -r -a date <<< "$valeur" ;;
          #                                  ^^^
          # correspondance de la ligne "Date".
          [Rr][Ee][Cc][Ee][Ii][Vv][Ee][Dd] ) read -r -a sender <<< "$valeur" ;;
          #                                                    ^^^
          # correspondance de l'adresse IP (pourrait être f).
          esac

       else
          (( corps++ ))
          (( correspondance  )) &&
          echo "MESSAGE ${date:+of: ${date[*]} }"
       #    Tableau entier $date           ^
          echo "IP address of sender: ${sender[1]}"
       #    Second champ de la ligne "Received"^

       fi


    done < "$1" # Redirige le stdout du fichier dans une boucle.
}


mbox_grep "$1"  # Envoie le fichier mailbox.

exit $?

# Exercices :
# ----------
# 1) Cassez la seule fonction, ci-dessus, dans plusieurs fonctions.
# 2) Ajoutez des analyses supplémentaires dans le script, en vérifiant plusieurs
mots-clés.



$ mailbox_grep.sh scam_mail
  MESSAGE of Thu, 5 Jan 2006 08:00:56 -0500 (EST)
  IP address of sender: 196.3.62.4
```

Exercice : trouver d'autres utilisations des *chaînes en ligne*.

Chapitre 19. Redirection d'E/S (entrées/sorties)

Trois différents « fichiers » sont toujours ouverts par défaut, `stdin` (le clavier), `stdout` (l'écran) et `stderr` (la sortie des messages d'erreur vers l'écran). Ceux-ci, et n'importe quel autre fichier ouvert, peuvent être redirigés. La redirection signifie simplement la capture de la sortie d'un fichier, d'une commande, d'un programme, d'un script, voire même d'un bloc de code dans un script (voir l'Exemple 3.1, « Blocs de code et redirection d'entrées/sorties » et l'Exemple 3.2, « Sauver la sortie d'un bloc de code dans un fichier ») et le renvoi du flux comme entrée d'un autre fichier, commande, programme ou script.

Chaque fichier ouvert se voit affecté un descripteur de fichier. [1] Les descripteurs de fichier pour `stdin`, `stdout` et `stderr` sont, respectivement, 0, 1 et 2. Pour ouvrir d'autres fichiers, il reste les descripteurs 3 à 9. Il est quelque fois utile d'affecter un de ces descripteurs supplémentaires de fichiers pour `stdin`, `stdout` ou `stderr` comme lien dupliqué temporaire. [2] Ceci simplifie le retour à la normale après une redirection complexe et un remaniement (voir l'Exemple 19.1, « Rediriger `stdin` en utilisant *exec* »).

```
SORTIE_COMMANDE >
   # Redirige la sortie vers un fichier.
   # Crée le fichier s'il n'est pas présent, sinon il l'écrase.

   ls -lR > repertoire.liste
   # Crée un fichier contenant la liste des fichiers du répertoire.

: > nom_fichier
   # Le > vide le fichier "nom_fichier".
   # Si le fichier n'est pas présent, crée un fichier vide (même effet que
   # 'touch').
   # Le : sert en tant que contenant, ne produisant aucune sortie.

> nom_fichier
   # Le > vide le fichier "nom_fichier".
   # Si le fichier n'est pas présent, crée un fichier vide (même effet que
   # 'touch').
   # (Même résultat que ": >", ci-dessus, mais ceci ne fonctionne pas avec
   # certains shells.)

SORTIE_COMMANDE >>
   # Redirige stdout vers un fichier.
   # Crée le fichier s'il n'est pas présent, sinon il lui ajoute le flux.


   # Commandes de redirection sur une seule ligne (affecte seulement la ligne
   # sur laquelle ils sont):
   # ----------------------------------------------------------------------

1>nom_fichier
   # Redirige stdout vers le fichier "nom_fichier".
1>>nom_fichier
   # Redirige et ajoute stdout au fichier "nom_fichier".
2>nom_fichier
   # Redirige stderr vers le fichier "nom_fichier".
2>>nom_fichier
   # Redirige et ajoute stderr au fichier "nom_fichier".
&>nom_fichier
   # Redirige à la fois stdout et stderr vers le fichier "nom_fichier".
   #
   #  Notez que   &>>fichier
   #+ -- tente de rediriger et d'*ajouter*
   #+ stdout et stderr au fichier "fichier" --
   #+ échoue avec un message d'erreur,
   #+ syntax error near unexpected token '>'.
```

[1] Un *descripteur de fichier* est simplement un numéro que le système d'exploitation affecte à un fichier ouvert pour garder sa trace. Considérez cela comme une version simplifiée d'un pointeur de fichier. C'est analogue à un *handle vers un fichier* en **C**.

[2] Utiliser le `descripteur de fichier 5` pourrait causer des problèmes. Lorsque Bash crée un processus fils, par exemple avec exec, le fils hérite de fd 5 (voir le courrier électronique archivé de Chet Ramey, *SUBJECT: RE: File descriptor 5 is held open*, NdT: Le descripteur de fichier est laissé ouvert). Il est plus raisonnable de laisser ce descripteur tranquille.

```
   M>N
     # "M" est un descripteur de fichier, par défaut 1 s'il n'est pas précisé.
     # "N" est un nom de fichier.
     # Le descripteur de fichier "M" est redirigé vers le fichier "N".
   M>&N
     # "M" est un descripteur de fichier, par défaut 1 s'il n'est pas précisé.
     # "N" est un autre descripteur de fichier.

      #============================================================================

      # Rediriger stdout, une ligne à la fois.
      FICHIERLOG=script.log

      echo "Cette instruction est envoyée au fichier de traces, \"$FICHIERLOG\"."
1>$FICHIERLOG
      echo "Cette instruction est ajoutée à \"$FICHIERLOG\"." 1>>$FICHIERLOG
      echo "Cette instruction est aussi ajoutée à \"$FICHIERLOG\"." 1>>$FICHIERLOG
      echo "Cette instruction est envoyé sur stdout et n'apparaitra pas dans
\"$FICHIERLOG\"."
      # Ces commandes de redirection sont "réinitialisées" automatiquement après chaque
ligne.



      # Rediriger stderr, une ligne à la fois.
      FICHIERERREURS=script.erreurs

      mauvaise_commande1 2>$FICHIERERREURS     #  Message d'erreur envoyé vers
$FICHIERERREURS.
      mauvaise_commande2 2>>$FICHIERERREURS #  Message d'erreur ajouté à
$FICHIERERREURS.
      mauvaise_commande3                          #  Message d'erreur envoyé sur
stderr,
                                                  #+ et n'apparaissant pas dans
$FICHIERERREURS.
      #  Ces commandes de redirection sont aussi "réinitialisées" automatiquement
      #+ après chaque ligne.
      #============================================================================
```

```
   2>&1 \
      # Redirige stderr vers stdout.
      # Les messages d'erreur sont envoyés à la même place que la sortie standard.

   i>&j
      # Redirige le descripteur de fichier i vers j.
      # Toute sortie vers le fichier pointé par i est envoyée au fichier pointé par j.

   >&j
      # Redirige, par défaut, le descripteur de fichier 1 (stdout) vers j.
      # Toutes les sorties vers stdout sont envoyées vers le fichier pointé par j.
```

```
   0< NOM_FICHIER
    < NOM_FICHIER
      # Accepte l'entrée à partir d'un fichier.
      # Commande compagnon de « > », et souvent utilisée en combinaison avec elle.
      #
      # grep mot_recherché <nom_fichier


   [j]<>nom_fichier
      #  Ouvre le fichier "nom_fichier" pour lire et écrire, et affecter le descripteur
      #+ de fichier "j" à celui-ci.
      # Si "nom_fichier" n'existe pas, le créer.
      # Si le descripteur de fichier "j" n'est pas spécifié, le défaut est fd 0, stdin.
```

```
      #
      # Une application de ceci est d'écrire à une place spécifiée dans un fichier.
      echo 1234567890 > Fichier    # Écrire une chaîne dans "Fichier".
      exec 3<> Fichier             # Ouvrir "Fichier" et lui affecter le fd 3.
      read -n 4 <&3                # Lire seulement quatre caractères.
      echo -n . >&3                # Écrire un point décimal à cet endroit.
      exec 3>&-                   # Fermer fd 3.
      cat Fichier                  # ==> 1234.67890
      # Accès au hasard, par golly.



   |
      # Tube.
      # outil de chaînage de processus et de commande à but général.
      # Similaire à « > », mais plus général dans l'effet.
      # Utile pour chaîner des commandes, scripts, fichiers et programmes.
      cat *.txt | sort | uniq > fichier-resultat
      # Trie la sortie de tous les fichiers .txt et supprime les lignes
      # dupliquées, pour finalement enregistrer les résultats dans
      # « fichier-résultat ».
```

Plusieurs instances de redirection d'entrées et de sorties et/ou de tubes peuvent être combinées en une seule ligne de commande.

```
commande < fichier-entrée > fichier-sortie

commande1 | commande2 | commande3 > fichier-sortie
```

Voir l'Exemple 15.31, « Déballer une archive *rpm* » et l'Exemple A.15, « *fifo*: Faire des sauvegardes journalières, en utilisant des tubes nommés ».

Plusieurs flux de sortie peuvent être redirigés vers un fichier.

```
ls -yz >> commande.log 2>&1
# La capture résulte des options illégales "yz" de "ls" dans le fichier
# "commande.log".
# Parce que stderr est redirigé vers le fichier, aucun message d'erreur ne sera
# visible.

# Néanmoins, notez que ce qui suit ne donne *pas* le même résultat.
ls -yz 2>&1 >> command.log
# Affiche un message d'erreur et n'écrit pas dans le fichier.

#  Si vous redirigez à la fois stdout et stderr, l'ordre des commandes fait une
#+ différence.
```

Fermer les descripteurs de fichiers

n<&-
: Ferme le descripteur de fichier *n*.

0<&-, <&-
: Ferme `stdin`.

n>&-
: Ferme le descripteur de fichiers de sortie *n*.

1>&-, >&-
: Ferme `stdout`.

Les processus fils héritent des descripteurs de fichiers ouverts. C'est pourquoi les tubes fonctionnent. Pour empêcher l'héritage d'un fd, fermez-le.

```
# Rediriger seulement stderr vers un tube.

exec 3>&1                             # Sauvegarde la valeur "actuelle" de stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-    # Ferme fd 3 pour 'grep' (mais pas pour 'ls').
#              ^^^^   ^^^^
```

```
exec 3>&-                                # Maintenant, fermez-le pour le reste du script.

# Merco, S.C.
```

Pour une introduction plus détaillée de la redirection d'E/S, voir l'Annexe E, Une introduction détaillée sur les redirections d'entrées/sorties.

19.1. Utiliser *exec*

Une commande **exec <nomfichier** redirige `stdin` vers un fichier. À partir de là, `stdin` vient de ce fichier plutôt que de sa source habituelle (généralement un clavier). Ceci fournit une méthode pour lire un fichier ligne par ligne et donc d'analyser chaque ligne de l'entrée en utilisant sed631 et/ou awk634.

Exemple 19.1. Rediriger `stdin` en utilisant *exec*

```
#!/bin/bash
# Rediriger stdin en utilisant 'exec'.


exec 6<&0          # Lie le descripteur de fichier #6 avec stdin.
                   # Sauvegarde stdin.

exec < fichier-donnees   # stdin remplacé par le fichier "fichier-donnees"

read a1            # Lit la première ligne du fichier "fichier-donnees".
read a2            # Lit la deuxième ligne du fichier "fichier-donnees".

echo
echo "Les lignes suivantes lisent le fichier."
echo "---------------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  Maintenant, restaure stdin à partir de fd #6, où il a été sauvegardé,
#+ et ferme fd #6 ( 6<&- ) afin qu'il soit libre pour d'autres processus.
#
# <&6 6<&-    fonctionne aussi.

echo -n "Entrez des données "
read b1  #  Maintenant les fonctions lisent ("read") comme d'ordinaire,
         #+ c'est-à-dire à partir de stdin.
echo "Entrée lue à partir de stdin."
echo "-----------------------------"
echo "b1 = $b1"

echo

exit 0
```

De la même façon, une commande **exec >nomfichier** redirige `stdout` vers un fichier désigné. Ceci envoie toutes les sorties des commandes qui devraient normalement aller sur `stdout` vers ce fichier.

Important

exec N > nom_fichier affecte le script entier ou le *shell courant*. La redirection vers le PID du script ou du shell à partir de maintenant à changer. Néanmoins...

N > nom_fichier affecte seulement les processus qui vont être créés, pas le script entier ou le shell.

Merci à Ahmed Darwish de nous l'avoir précisé.

Exemple 19.2. Rediriger `stdout` en utilisant *exec*

```
#!/bin/bash
# reassign-stdout.sh

FICHIERTRACES=fichiertraces.txt

exec 6>&1           # Lie le descripteur de fichier #6 avec stdout.
                    # Sauvegarde stdout.

exec > $FICHIERTRACES     # stdout remplacé par le fichier "fichiertraces.txt".

# ------------------------------------------------------------ #
#  Toute sortie des commandes de ce bloc sera envoyée dans le fichier
#+ $FICHIERTRACES.

echo -n "Fichier traces: "
date
echo "-----------------------------------"
echo

echo "Sortie de la commande \"ls -al\""
echo
ls -al
echo; echo
echo "Sortie de la commande \"df\""
echo
df

# ------------------------------------------------------------ #

exec 1>&6 6>&-      # Restaure stdout et ferme le descripteur de fichier #6.

echo
echo "== stdout restauré à sa valeur par défaut == "
echo
ls -al
echo

exit 0
```

Exemple 19.3. Rediriger à la fois `stdin` et `stdout` dans le même script avec *exec*

```
#!/bin/bash
# upperconv.sh
# Convertit un fichier d'entrée spécifié en majuscule.

E_ACCES_FICHIER=70
E_MAUVAIS_ARGS=71

if [ ! -r "$1" ]     # Est-ce que le fichier spécifié est lisible ?
then
  echo "Ne peut pas lire le fichier d'entrée !"
  echo "Usage: $0 fichier-entrée fichier-sortie"
  exit $E_ACCES_FICHIER
fi  #  Sortira avec la même erreur,
    #+ même si le fichier d'entrée ($1) n'est pas spécifié (pourquoi ?).

if [ -z "$2" ]
then
  echo "A besoin d'un fichier de sortie."
  echo "Usage: $0 fichier-entrée fichier-sortie"
  exit $E_MAUVAIS_ARGS
fi
```

```
exec 4<&0
exec < $1            # Lira le fichier d'entrée.

exec 7>&1
exec > $2            # Écrira sur le fichier de sortie.
                     #  Assume que le fichier de sortie est modifiable
                     #+ (ajoutez une vérification ?).

# -----------------------------------------------
    cat - | tr a-z A-Z   # Conversion en majuscule.
#   ^^^^^               # Lecture de stdin.
#          ^^^^^^^^^^   # Écriture sur stdout.
# Néanmoins, à la fois stdin et stdout ont été redirigés.
# Notez que le 'cat' peut être omis.
# -----------------------------------------------

exec 1>&7 7>&-       # Restaure stdout.
exec 0<&4 4<&-       # Restaure stdin.

#  Après retour à la normal, la ligne suivante affiche sur stdout comme de
#+ normal.
echo "Le fichier \"$1\" a été enregistré dans \"$2\" après une convertion en
majuscule."

exit 0
```

La redirection d'entrée/sortie est un moyen intelligent pour éviter le terrifiant problème des variables inaccessibles à l'intérieur d'un sous-shell342.

Exemple 19.4. Éviter un sous-shell

```
#!/bin/bash
# avoid-subshell.sh
# Suggéré par Matthew Walker.

Lignes=0

echo

cat monfichier.txt | while read ligne;
                 do {
                   echo $ligne
                   (( Lignes++ )); #  Les valeurs incrémentées de cette variable
                                   #+ sont inaccessibles en dehors de la boucle.
                                   #  Problème de sous-shell.
                 }
                 done

echo "Nombre de lignes lues = $Lignes"   # 0
                                         # Mauvais !

echo "------------------------"


exec 3&lt;&gt; monfichier.txt
while read ligne &lt;&3
do {
  echo "$ligne"
  (( Lignes++ ));                # Les valeurs incrémentées de cette variable
                                 #+ sont inaccessibles en dehors de la boucle.
                                 #  Pas de sous-shell, pas de problème.
}
done
exec 3&gt;&-

echo "Nombre de lignes lues = $Lignes"     # 8
```

```
echo

exit 0

# Les lignes ci-dessous ne sont pas vues du script.

$ cat monfichier.txt

Ligne 1.
Ligne 2.
Ligne 3.
Ligne 4.
Ligne 5.
Ligne 6.
Ligne 7.
Ligne 8.
```

19.2. Rediriger les blocs de code

Les blocs de code, comme les boucles while, until et for, voire même les blocs de test if/then46 peuvent aussi incorporer une redirection de `stdin`. Même une fonction peut utiliser cette forme de redirection (voir l'Exemple 23.11, « Vrai nom pour un utilisateur »). L'opérateur < à la fin du bloc de code accomplit ceci.

Exemple 19.5. Boucle *while* redirigée

```
#!/bin/bash
# redir2.sh

if [ -z "$1" ]
then
  Fichier=noms.donnees       # par défaut, si aucun fichier n'est spécifié.
else
  Fichier=$1
fi
#+ Fichier=${1:-noms.donnees}
#  peut remplacer le test ci-dessus (substitution de paramètres).

compteur=0

echo

while [ "$nom" != Smith ]  # Pourquoi la variable $nom est-elle entre guillemets?
do
  read nom                 # Lit à partir de $Fichier, plutôt que de stdin.
  echo $nom
  let "compteur += 1"
done <"$Fichier"           # Redirige stdin vers le fichier $Fichier.
#    ^^^^^^^^^^^^

echo; echo "$compteur noms lus"; echo

exit 0

#  Notez que dans certains vieux langages de scripts shells,
#+ la boucle redirigée pourrait tourner dans un sous-shell.
# Du coup, $compteur renverrait 0, la valeur initialisée en dehors de la boucle.
#  Bash et ksh évitent de lancer un sous-shell *autant que possible*,
#+ de façon à ce que ce script, par exemple, tourne correctement.
# Merci à Heiner Steven pour nous l'avoir indiqué.

# Néanmoins...
#  Bash *peut* quelque fois commencer un sous-shell dans une boucle "while"
#+ alimentée par un *tube*,
#+ à distinguer d'une boucle "while" *redirigée*.

abc=hi
```

```
echo -e "1\n2\n3" | while read l
     do abc="$l"
        echo $abc
     done
echo $abc

#  Merci à Bruno de Oliveira Schneider pour avoir démontré ceci
#+ avec l'astuce de code ci-dessus.
#  Et merci à Brian Onn pour avoir corriger une erreur dans un commentaire.
```

Exemple 19.6. Autre forme de boucle *while* redirigée

```
#!/bin/bash

# Ceci est une forme alternative au script précédent.

#  Suggéré par Heiner Steven
#+ comme astuce dans ces situations où une boucle de redirection est lancée
#+ comme un sous-shell, et donc que les variables à l'intérieur de la boucle
#+ ne conservent pas leurs valeurs une fois la boucle terminée.


if [ -z "$1" ]
then
  Fichier=noms.donnees     # Par défaut, si aucun fichier spécifié.
else
  Fichier=$1
fi


exec 3<&0                  # Sauve stdin sur le descripteur de fichier 3.
exec 0<"$Fichier"          # Redirige l'entrée standard.

compteur=0
echo


while [ "$nom" != Smith ]
do
  read nom               # Lit à partir du stdin redirigé ($Fichier).
  echo $nom
  let "compteur += 1"
done                      #  La boucle lit à partir du fichier $Fichier
                          #+ à cause de la ligne 20.

#  La version originale de ce script terminait la boucle "while" avec
#+      done <"$Filename"
#  Exercice :
#  Pourquoi cela n'est-il pas nécessaire ?


exec 0<&3                 # Restaure l'ancien stdin.
exec 3<&-                 # Ferme le temporaire fd 3.

echo; echo "$compteur noms lus"; echo

exit 0
```

Exemple 19.7. Boucle *until* redirigée

```
#!/bin/bash
# Identique à l'exemple précédent, mais avec une boucle "until".

if [ -z "$1" ]
then
```

```
  Fichier=noms.donnees  # Par défaut, si aucun nom de fichier n'est spécifié.
else
  Fichier=$1
fi

# while [ "$nom" != Smith ]
until [ "$nom" = Smith ]     # Modification de  !=  en =.
do
  read nom                   # Lit à partir de $Fichier, plutôt que de stdin.
  echo $nom
done <"$Fichier"             # Redirige stdin vers le fichier $Fichier.
#    ^^^^^^^^^^^^

# Même résultats qu'avec la boucle "while" du précédent exemple.

exit 0
```

Exemple 19.8. Boucle *for* redirigée

```
#!/bin/bash

if [ -z "$1" ]
then
  Fichier=noms.donnees          # Par défaut, si aucun fichier n'est spécifié.
else
  Fichier=$1
fi

compteur_lignes=`wc $Fichier | awk '{ print $1 }'`
#           Nombre de lignes du fichier cible.
#
#  Très peu naturel, néanmoins cela montre qu'il est possible de rediriger
#+ stdin à l'intérieur d'une boucle "for"...
#+ si vous êtes assez intelligent.
#
# Une autre façon plus concise est    compteur_lignes=$(wc -l < "$Fichier")


for nom in `seq $compteur_lignes`  # Rappelez-vous que "seq" affiche une séquence de
nombres.
# while [ "$nom" != Smith ]   --   plus compliqué qu'une boucle "while" --
do
  read nom                    # Lit à partir de $Fichier, plutôt que de stdin.
  echo $nom
  if [ "$nom" = Smith ]       # A besoin de tout ce bagage supplémentaire ici.
  then
    break
  fi
done <"$Fichier"              # Redirige stdin vers le fichier $Fichier.
#    ^^^^^^^^^^^^

exit 0
```

Nous pouvons modifier le précédent exemple pour rediriger aussi la sortie de la boucle.

Exemple 19.9. Rediriger la boucle *for* (à la fois `stdin` et `stdout`)

```
#!/bin/bash

if [ -z "$1" ]
then
  Fichier=noms.donnees # Par défaut, si aucun fichier n'est spécifié.
else
  Fichier=$1
fi
```

```
FichierSauvegarde=$Fichier.nouveau # Fichier où sauvegarder les résultats.
NomFinal=Jonah                     # Nom par lequel terminer la lecture.

nb_lignes=`wc $Fichier | awk '{ print $1 }'`  # Nombre de lignes du fichier cible.


for nom in `seq $nb_lignes`
do
  read nom
  echo "$nom"
  if [ "$nom" = "$NomFinal" ]
  then
    break
  fi
done < "$Fichier" > "$FichierSauvegarde"  # Redirige stdin dans $Fichier,
#    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^       et sauvegarde dans le fichier.

exit 0
```

Exemple 19.10. Rediriger un test *if/then*

```
#!/bin/bash

if [ -z "$1" ]
then
  Fichier=noms.donnees   #  Valeur par défaut, si aucun nom de fichier n'est
                         #+ spécifié.
else
  Fichier=$1
fi

VRAI=1

if [ "$VRAI" ]          # if true    et   if :   fonctionnent aussi.
then
 read nom
 echo $nom
fi <"$Fichier"
#  ^^^^^^^^^^^^

# Lit seulement la première ligne du fichier.
#  Un test "if/then" n'a aucun moyen de faire une itération sauf si il est
#+ intégré dans une boucle.

exit 0
```

Exemple 19.11. Fichier de données *nom.données* pour les exemples ci-dessus

```
Aristotle
Belisarius
Capablanca
Euler
Goethe
Hamurabi
Jonah
Laplace
Maroczy
Purcell
Schmidt
Semmelweiss
Smith
Turing
Venn
Wilson
```

```
Znosko-Borowski

#  This is a data file for
#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Rediriger `stdout` d'un bloc de code a le même effet que d'en sauver la sortie dans un fichier. Voir l'Exemple 3.2, « Sauver la sortie d'un bloc de code dans un fichier ».

Les documents en ligne318 sont un cas spécial pour la redirection de blocs de code. Dans ce cas, il devrait être possible d'alimenter le `stdin` d'une *boucle while* avec la sortie d'un *document en ligne*.

```
# Exemple de Albert Siersema
# Utilisé avec sa permission (merci !).

function sortie()
 # Pourrait aussi être une commande externe, bien sûr.
 # Ici, nous démontrons que vous pouvez utiliser aussi une fonction.
{
  ls -al *.jpg | awk '{print $5,$9}'
}


nr=0          #  Nous voulons que la boucle while puisse les manipuler et
totalSize=0   #+ nous voulons être capable de voir les changements après la fin de la
boucle.

while read tailleFichier nomFichier ; do
  echo "$nomFichier fait $tailleFichier octets"
  let nr++
  tailleTotale=$((tailleTotale+$tailleFichier))   # Or: "let
tailleTotale+=tailleFichier"
done<<EOF
$(sortie)
EOF

echo "$nr fichiers totalisant $tailleTotale octets"
```

19.3. Applications

Une utilisation intelligente de la redirection d'E/S est l'analyse et le collage de petits bouts de la sortie de commandes (voir l'Exemple 14.7, « Utiliser *read* avec la redirection de fichier »). Ceci permet de générer des rapports et des fichiers de traces.

Exemple 19.12. Enregistrer des événements

```
#!/bin/bash
# logevents.sh
# Auteur : Stephane Chazelas.

# Tracer des événements dans un fichier.
#  Vous devez être root pour exécuter ceci (en fait pour avoir le droit d'écrire dans
#+ /var/log).

ROOT_UID=0     # Seuls les utilisateurs ayant l'identifiant $UID 0 ont les
               #+ privilèges de root.
E_NONROOT=67   # Code de sortie si non root.


if [ "$UID" -ne "$ROOT_UID" ]
then
  echo "Vous devez être root pour exécuter ce script."
  exit $E_NONROOT
fi


FD_DEBUG1=3
FD_DEBUG2=4
```

```
FD_DEBUG3=5

# === Décommentez une des deux lignes ci-dessous pour activer le script. ===
# TRACE_EVENEMENTS=1
# TRACE_VARS=1


log()  # Ecrit la date et l'heure dans le fichier de traces.
{
echo "$(date)  $*" &gt;&7  # Ceci *ajoute* la date dans le fichier.
#     ^^^^^^^ Substitution de commande
                          # Voir ci-dessous.
}



case $NIVEAU_TRACES in
 1) exec 3&gt;&2         4&gt; /dev/null 5&gt; /dev/null;;
 2) exec 3&gt;&2         4&gt;&2         5&gt; /dev/null;;
 3) exec 3&gt;&2         4&gt;&2         5&gt;&2;;
 *) exec 3&gt; /dev/null 4&gt; /dev/null 5&gt; /dev/null;;
esac

FD_TRACEVARS=6
if [[ $TRACE_VARS ]]
then exec 6&gt;&gt; /var/log/vars.log
else exec 6&gt; /dev/null               # Bury output.
fi

FD_TRACEEVENEMENTS=7
if [[ $TRACE_EVENEMENTS ]]
then
  # exec 7 &gt;(exec gawk '{print strftime(), $0}' &gt;&gt; /var/log/event.log)
  # La ligne ci-dessus ne fonctionnera pas avec Bash, version 2.04 et ultérieur.
  # Pourquoi ?
  exec 7&gt;&gt; /var/log/event.log        # Ajoute dans "event.log".
  log                                    # Ecrit la date et l'heure.
else exec 7&gt; /dev/null                 # Supprime le sortie.
fi

echo "DEBUG3: début" &gt;&${FD_DEBUG3}

ls -l &gt;&5 2&gt;&4                      # commande1 &gt;&5 2&gt;&4

echo "Done"                            # commande2

echo "envoi mail" &gt;&${FD_LOGEVENTS}
# Ecrit "envoi mail" sur le descripteur de fichier #7.


exit 0
```

Chapitre 20. Sous-shells

Exécuter un script shell lance un nouveau processus, un *sous-shell*.

> Définition : un *sous-shell* est un processus lancé par un shell (ou *script shell*).

Un sous-shell est une instance séparée du gestionnaire de commande -- le *shell* qui vous donne l'invite sur la console ou dans une fenêtre *xterm*. De la même façon que vos commandes sont interprétées sur l'invite de commandes, un script traite en flot une liste de commandes. Chaque script shelle en cours d'exécution est un sous-processus (processus enfant) du shell parent shell.

Un script shell peut lancer lui-même des sous-processus. Ces *sous-shells* permettent au script de faire de l'exécution en parallèle, donc d'exécuter différentes tâches simultanément.

```
#!/bin/bash
# subshell-test.sh

(
# Parenthèses à l'intérieur, donc un sous-shell...
while [ 1 ]   # Boucle sans fin.
do
  echo "Sous-shell en cours d'exécution..."
done
)

#  Le script s'exécutera sans jamais s'arrêter,
#+ au moins tant que vous ne ferez pas Ctl-C.

exit $?  # Fin du script (mais l'exécution ne parviendra jamais jusqu'ici).



Maintenant, exécutez le script :
sh subshell-test.sh

et tant que le script s'exécute, à partir d'un autre xterm :
ps -ef | grep subshell-test.sh

UID       PID   PPID  C STIME TTY      TIME     CMD
500       2698  2502  0 14:26 pts/4    00:00:00 sh subshell-test.sh
500       2699  2698 21 14:26 pts/4    00:00:24 sh subshell-test.sh

          ^^^^

Analyse :
PID 2698, le script, a lancé le sous-shell de PID 2699.

Note : la ligne "UID ..." peut être filtrée par la commande grep
mais elle est laissée ici dans un but démonstratif.
```

En général, une commande externe dans un script lance un sous-processus [1] alors qu'une commande intégrée159 Bash ne le fait pas. Pour cette raison, les commandes intégrées s'exécutent plus rapidement que leur commande externe équivalente.

Liste de commandes entre parenthèses

(commande1; commande2; commande3; ... commandeN;)
: Une liste de commandes placées entre `parenthèses` est exécutée sous forme de sous-shells

Les variables utilisées dans un sous shell *ne sont pas* visibles en dehors du code du sous-shell. Elles ne sont pas utilisables par le processus parent, le shell qui a lancé le sous-shell. Elles sont en réalité des variables locales.

[1] Une commande externe appelée avec un exec ne lance *pas* (habituellement) un sous-processus / sous-shell.

Exemple 20.1. Étendue des variables dans un sous-shell

```
#!/bin/bash
# subshell.sh

echo

echo "Nous sommes à l'extérieur du sous-shell."
echo "Niveau de sous-shell À L'EXTÉRIEUR DU sous-shell = $BASH_SUBSHELL"
# Bash, version 3, ajoute la nouvelle variable           $BASH_SUBSELL.
echo; echo

variable_externe=externe
variable_globale=
#  Définit une variable globale pour le stockage
#+ de la valeur d'une variable d'un sous-shell.

(
echo "Nous sommes à l'intérieur du sous-shell."
echo "Niveau de sous-shell À L'INTÉRIEUR DU sous-shell = $BASH_SUBSHELL"
variable_interne=interne
echo "À partir du sous-shell interne, \"variable_interne\" = $variable_interne"
echo "À partir du sous-shell interne, \"externe\" = $variable_externe"

variable_globale="$variable_interne" #  Est-ce que ceci permet l'export
                                     #+ d'une variable d'un sous-shell ?
)

echo; echo
echo "Nous sommes à l'extérieur du sous-shell."
echo
echo "Niveau de sous-shell À L'EXTÉRIEUR DU sous-shell = $BASH_SUBSHELL"
echo


if [ -z "$variable_interne" ]
then
  echo "variable_interne non défini dans le corps principal du shell"
else
  echo "variable_interne défini dans le corps principal du shell"
fi

echo "À partir du code principal du shell, \"variable_interne\" = $variable_interne"
#  $variable_interne s'affichera comme non initialisée parce que les variables
#+ définies dans un sous-shell sont des "variables locales".
# Existe-t'il un remède pour ceci ?
echo "variable_globale = "$variable_globale""  # Pourquoi ceci ne fonctionne pas ?


echo

# ==================================================================

# De plus...

echo "-----------------"; echo

var=41                                                # Variable globale.

( let "var+=1"; echo "\$var À L'INTÉRIEUR D'UN sous-shell = $var" )  # 42

echo "\$var EN DEHORS DU sous-hell = $var"                    # 41
#  Les opérations sur des variabledans un sous-shell, même dans une variable
#+ globale, n'affectent pas la valeur de la variable en dehors du sous-shell !

exit 0

#  Question :
```

```
#  ---------
#  Une fois le sous-shell quitté,
#+ existe-il un moyen d'entrer de nouveau dans le même sous-shell
#+ pour modifier ou accéder aux variables du sous-shell ?
```

Voir aussi l'Exemple 31.2, « Problèmes des sous-shell ».

> *Définition :* La *portée* d'une variable est le contexte dans lequel elle a une signification, dans lequel elle a une *valeur* qui peut être référencée. Par exemple, la portée d'une variable locale s'étend seulement à la fonction, au bloc de code ou au sous-shell dans lequel elle a été définie.

Note

Alors que la variable interne the $BASH_SUBSHELL indique le niveau d'imbrication des sous-shells, la variable $SHLVL *ne montre aucune modification* dans un sous-shell.

```
echo " \$BASH_SUBSHELL en dehors du sous-shell  = $BASH_SUBSHELL"           #
0
  ( echo " \$BASH_SUBSHELL dans le sous-shell = $BASH_SUBSHELL" )     # 1
  ( ( echo " \$BASH_SUBSHELL à l'intérieur du sous-shell imbriqué =
$BASH_SUBSHELL" ) ) # 2
# ^ ^                          *** imbriquée ***                        ^ ^

echo

echo " \$SHLVL en dehors du sous-shell = $SHLVL"       # 3
( echo " \$SHLVL à l'intérieur du inside sous-shell  = $SHLVL" )   # 3 (aucun
changement !)
```

Le changement de répertoire effectué dans un sous-shell n'a pas d'incidence sur le shell parent.

Exemple 20.2. Lister les profils utilisateurs

```
#!/bin/bash
# allprofs.sh : affiche tous les profils utilisateur.

# Ce script a été écrit par Heiner Steven et modifié par l'auteur du document.

FICHIER=.bashrc  #  Fichier contenant le profil utilisateur,
                 #+ était ".profile" dans le script original.

for home in `awk -F: '{print $6}' /etc/passwd`
do
  [ -d "$home" ] || continue    #  Si pas de répertoire personnel, passez au
                                #+ suivant.
  [ -r "$home" ] || continue    # Si non lisible, passez au suivant.
  (cd $home; [ -e $FICHIER ] && less $FICHIER)
done

#  Quand le script se termine, il n'y a pas de besoin de retourner dans le
#+ répertoire de départ parce que 'cd $home' prend place dans un sous-shell.

exit 0
```

Un sous-shell peut être utilisé pour mettre en place un « environnement dédié » à un groupe de commandes.

```
COMMANDE1
COMMANDE2
COMMANDE3
(
  IFS=:
```

```
  PATH=/bin
  unset TERMINFO
  set -C
  shift 5
  COMMANDE4
  COMMANDE5
  exit 3 # Sortie du seul sous-shell !
)
#  Le shell parent n'a pas été affecté et son environnement est préservé (ex :
#+ pas de modification de $PATH).
COMMANDE6
COMMANDE7
```

Comme vous le voyez, la commande exit termine seulement le sous-shell dans lequel il s'exécute, mais il ne termine *pas* le shell ou le script parent.

L'intérêt peut être par exemple de tester si une variable est définie ou pas.

```
if (set -u; : $variable) 2> /dev/null
then
  echo "La variable est définie."
fi     #  La variable a été initialisée dans le script en cours,
       #+ ou est une variable interne de Bash,
       #+ ou est présente dans l'environnement (a été exportée).

# Peut également s'écrire [[ ${variable-x} != x || ${variable-y} != y ]]
# ou                      [[ ${variable-x} != x$variable ]]
# ou                      [[ ${variable+x} = x ]]
# ou                      [[ ${variable-x} != x ]]
```

Une autre application est de vérifier si un fichier est marqué comme verrouillé :

```
if (set -C; : > fichier_verrou) 2> /dev/null
then
  :   # fichier_verrou n'existe pas : aucun utilisateur n'exécute ce script
else
  echo "Un autre utilisateur exécute déjà ce script."
  exit 65
fi

#  Code de Stéphane Chazelas,
#+ avec des modifications de Paulo Marcel Coelho Aragao.
```

+

Des processus peuvent être exécutés en parallèle dans différents sous-shells. Cela permet de séparer des tâches complexes en plusieurs sous-composants exécutés simultanément.

Exemple 20.3. Exécuter des processus en parallèle dans les sous-shells

```
(cat liste1 liste2 liste3 | sort | uniq > liste123) &
(cat liste4 liste5 liste6 | sort | uniq > liste456) &
# Concatène et trie les 2 groupes de listes simultanément.
# Lancer en arrière-plan assure une exécution en parallèle.
#
# Peut également être écrit :
#   cat liste1 liste2 liste3 | sort | uniq > liste123 &
#   cat liste4 liste5 liste6 | sort | uniq > liste456 &

wait   # Ne pas exécuter la commande suivante tant que les sous-shells
       # n'ont pas terminé

diff liste123 liste456
```

Redirection des entrées/sorties (I/O) dans un sous-shell en utilisant « | », l'opérateur tube (pipe en anglais), par exemple **`ls -al | (commande)`**.

Note

Un bloc de commandes entre `accolades` ne lance *pas* un sous-shell.

{ commande1; commande2; commande3; ... }

Chapitre 21. Shells restreints

Commandes désactivées en shell restreint

... Exécuter un script ou une partie de script en mode *restreint* désactive certaines commandes qui, sinon, seraient utilisables. C'est une mesure de sécurité ayant pour objectif de limiter les droits de l'utilisateur du script et de minimiser les risques liés à l'exécution de ce script.

... L'usage de *cd* pour changer de répertoire courant.

Le changement de valeur des variables d'environnement suivantes : *$PATH*, *$SHELL*, *$BASH_ENV*, *$ENV*.

La lecture ou le remplacement d'options d'environnement de shell *$SHELLOPTS*.

La redirection de sortie.

L'appel à des commandes contenant un / ou plusieurs.

L'appel à exec pour substituer un processus différent de celui du shell.

Divers autres commandes qui pourraient permettre de détourner le script de son objectif initial.

La sortie du mode restreint à l'intérieur d'un script.

Exemple 21.1. Exécuter un script en mode restreint

```
#!/bin/bash

#  Commencer le script avec "#!/bin/bash -r" lance le script entier en mode
#+ restreint.

echo

echo "Changement de répertoire."
cd /usr/local
echo "Maintenant dans `pwd`"
echo "Je retourne à la maison."
cd
echo "Maintenant dans `pwd`"
echo

# Jusqu'ici, tout est en mode normal, non restreint.

set -r
# set --restricted    a le même effet.
echo "==> Maintenant en mode restreint. <=="

echo
echo

echo "Tentative de changement de répertoire en mode restreint."
cd ..
echo "Toujours dans `pwd`"

echo
echo

echo "\$SHELL = $SHELL"
echo "Tentative de changement de shell en mode restreint."
SHELL="/bin/ash"
echo
echo "\$SHELL= $SHELL"

echo
echo

echo "Tentative de redirection de sortie en mode restreint."
ls -l /usr/bin > bin.fichiers
```

```
ls -l bin.fichiers    # Essayez de lister le fichier que l'on a tenté de créer.
echo
exit 0
```

Chapitre 22. Substitution de processus

Piping the `stdout` of a command into the `stdin` of another is a powerful technique. But, what if you need to pipe the `stdout` of *multiple* commands? This is where *`process substitution`* comes in.

La *substitution de processus* envoit la sortie d'un ou plusieurs processus dans l'entrée standard `stdin` d'un autre processus.

Patron

commande à l'intérieur de parenthèses

>(liste_de_commandes)

<(liste_de_commandes)

La substitution de processus utilise les fichiers `/dev/fd/<n>` pour envoyer le résultat des processus entre parenthèses vers un autre processus. [1]

Attention

Il n'y a *pas* d'espace entre le « < » ou « > » et les parenthèses. Ici, une espace génèrerait un message d'erreur.

```
bash$ echo >(true)
/dev/fd/63

bash$ echo <(true)
/dev/fd/63
```

Bash crée un tube avec deux descripteurs de fichiers330, `--fIn` et `fOut--`. Le `stdin` (entrée standard) de true se connecte à `fOut` (la sortie standard) (dup2(fOut, 0)), puis Bash passe un `/dev/fd/fIn` comme argument à la commande **echo**. Sur les systèmes sans fichier `/dev/fd/<n>`, Bash peut utiliser des fichiers temporaires (merci S.C.).

La substitution de processus peut comparer la sortie de deux commandes différentes, voire même la sortie dûe à différentes options de la même commande.

```
bash$ comm <(ls -l) <(ls -al)
total 12
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
total 20
drwxrwxrwx    2 bozo bozo     4096 Mar 10 18:10 .
drwx------   72 bozo bozo     4096 Mar 10 17:58 ..
-rw-rw-r--    1 bozo bozo       78 Mar 10 12:58 File0
-rw-rw-r--    1 bozo bozo       42 Mar 10 12:58 File2
-rw-rw-r--    1 bozo bozo      103 Mar 10 12:58 t2.sh
```

Utiliser la substitution de processus pour comparer le contenu de deux répertoires (pour connaître les fichiers présents dans l'un mais pas dans l'autre :

```
diff <(ls $premier_repertoire) <(ls $deuxieme_repertoire)
```

Quelques autres utilisations de la substitution de processus :

```
read -a list < <( od -Ad -w24 -t u2 /dev/urandom )
#  Lit une liste de nombres aléatoires à partir de /dev/urandom,
#+ les traite avec « od »
#+ et les envoit dans l'entrée standard de « read »...

#  Provient du script exemple "insertion-sort.bash".
#  Merci à JuanJo Ciarlante.
```

[1] Ceci a le même effet qu'un tube nommé274 (fichier temporaire), et, en fait, les tubes nommés étaient autrefois utilisés dans les substitutions de processus.

```
cat <(ls -l)
# Même chose que   ls -l | cat

sort -k 9 <(ls -l /bin) <(ls -l /usr/bin) <(ls -l /usr/X11R6/bin)
#  Liste tous les fichiers des trois principaux répertoires "bin" et les trie
#+ par nom de fichier.
# Notez les trois commandes distinctes (Comptez les <) vont "nourrir" 'sort'.


diff <(command1) <(command2)    #  Fournit les différences entre les
                                #+ sorties des commandes.

tar cf >(bzip2 -c > file.tar.bz2) $nom_repertoire
# Appelle "tar cf /dev/fd/?? $nom_repertoire" et "bzip2 -c > fichier.tar.bz2"
#
# À cause de la fonctionnalité système /dev/fd/<n>,
# le tube entre les deux commandes n'a pas besoin d'être nommé.
#
# Ceci peut être émulé.
#
bzip2 -c < pipe > fichier.tar.bz2&
tar cf pipe $nom_repertoire
rm pipe
#       ou
exec 3>&1
tar cf /dev/fd/4 $nom_repertoire 4>&1 >&3 3>&- | bzip2 -c > fichier.tar.bz2 3>-
exec 3>&-


# Merci, Stéphane Chazelas
```

Un lecteur a envoyé cet intéressant exemple de substitution de processus.

```
# Fragment de script provenant d'une distribution Suse :

# -------------------------------------------------------------#
while read des what mask iface; do
# Quelques commandes ...
done < <(route -n)
#    ^ ^  Le premier < est la redirection,
#         le second correspond à la substitution du processus.

# Pour le tester, faisons lui faire quelque chose
while read  des what mask iface; do
  echo $des $what $mask $iface
done < <(route -n)

# Sortie:
# Table de routage IP du noyau
# Destination Gateway Genmask Flags Metric Ref Use Iface
# 127.0.0.0 0.0.0.0 255.0.0.0 U 0 0 0 lo
# -------------------------------------------------------------#

# Comme  Stéphane Chazelas le souligne, voici un équivalent plus aisément
compréhensible :
route -n |
  while read des what mask iface; do   #  Les variables sont affectées par la
                                       #+ sortie du tube.
    echo $des $what $mask $iface
  done  #  Ceci engendre la même sortie que ci-dessus.
        #  Néanmoins, comme le précise Ulrich Gayer...
        #+ cet équivalent simplifié utilise un sous-shell pour la boucle while
        #+ et donc les variables disparaissent quand l'envoi via le tube se
        #+ termine.

# -------------------------------------------------------------#

#  Néanmoins, Filip Moritz indique qu'il existe une différence subtile
```

```
#+ entre les deux exemples ci-dessus, comme nous le montre la suite.

(
route -n | while read x; do ((y++)); done
echo $y # $y n'est toujours pas initialisé

while read x; do ((y++)); done < <(route -n)
echo $y # $y a le nombre de lignes en sortie de route -n
)

# Plus généralement
(
: | x=x
# semble lancer un sous-shell comme
: | ( x=x )
# alors que
x=x < <(:)
# ne le fait pas
)

# C'est utile pour analyser csv ou un fichier de ce genre.
# En effet, c'est ce que fait le fragement de code SuSE original.
```

Chapitre 23. Fonctions

Comme les « vrais » langages de programmation, Bash supporte les fonctions bien qu'il s'agisse d'une implémentation quelque peu limitée. Une fonction est une sous-routine, un bloc de code qui implémente un ensemble d'opérations, une « boîte noire » qui réalise une tâche spécifiée. Quand il y a un code répétitif, lorsqu'une tâche se répète avec quelques légères variations dans la procédure, alors utilisez une fonction.

```
function nom_fonction {
commande...
}
```

ou

```
nom_fonction () {
commande...
}
```

Cette deuxième forme plaira aux programmeurs C (et est plus portable).

Comme en C, l'accolade ouvrante de la fonction peut apparaître de manière optionnelle sur la deuxième ligne.

```
nom_fonction ()
{
commande...
}
```

Note

Une fonction peut être « réduite » à une seule ligne.

```
fun () { echo "C'est une fonction"; echo; }
```

Néanmoins, dans ce cas, un *point-virgule* doit suivre la dernière commande de la fonction.

```
fun () { echo "This is a function"; echo } # Error!
```

Les fonctions sont appelées, *lancées*, simplement en invoquant leur noms.

Exemple 23.1. Fonctions simples

```
#!/bin/bash

JUSTE_UNE_SECONDE=1

funky ()
{ # C'est aussi simple que les fonctions get.
  echo "Ceci est la fonction funky."
  echo "Maintenant, sortie de la fonction funky."
} # La déclaration de la fonction doit précéder son appel.


fun ()
{ # Une fonction un peu plus complexe.
  i=0
  REPETITION=30

  echo
  echo "Et maintenant, les choses drôles commencent."
  echo

  sleep $JUSTE_UNE_SECONDE    # Hé, attendez une seconde !
  while [ $i -lt $REPETITION ]
  do
    echo "----------FONCTIONS---------->"
    echo "<---------AMUSANTES----------"
```

```
    echo
    let "i+=1"
  done
}

  # Maintenant, appelons les fonctions.

funky
fun

exit 0
```

La définition de la fonction doit précéder son premier appel. Il n'existe pas de méthode pour « déclarer » la fonction, comme en C par exemple.

```
f1
# Donnera un message d'erreur car la fonction "f1" n'est pas encore définie.

declare -f f1      # Ceci ne nous aidera pas plus.
f1                 # Toujours un message d'erreur.

# Néanmoins...


f1 ()
{
  echo "Appeler la fonction \"f2\" à partir de la fonction \"f1\"."
  f2
}

f2 ()
{
  echo "Fonction \"f2\"."
}

f1  # La fonction "f2" n'est pas appelée jusqu'à ce point bien qu'elle soit
    # référencée avant sa définition.
    # C'est autorisé.

# Merci, S.C.
```

Il est même possible d'intégrer une fonction dans une autre fonction bien que cela ne soit pas très utile.

```
f1 ()
{

  f2 () # intégrée
  {
    echo "La fonction \"f2\", à l'intérieur de \"f1\"."
  }

}

f2  # Donne un message d'erreur.
    # Même un "declare -f f2" un peu avant ne changerait rien.
echo

f1  # Ne donne rien, car appeler "f1" n'appelle pas automatiquement "f2".
f2  # Maintenant, il est tout à fait correct d'appeler "f2",
    # car sa définition est visible en appelant "f1".

# Merci, S.C.
```

Les déclarations des fonctions peuvent apparaître dans des endroits bien étonnants, même là où irait plutôt une commande.

```
ls -l | foo() { echo "foo"; }  # Autorisé, mais sans intérêt.
```

```
if [ "$USER" = bozo ]
then
  bozo_salutations ()   # Définition de fonction intégrée dans une construction
if/then.
  {
    echo "Bonjour, Bozo."
  }
fi

bozo_salutations        #  Fonctionne seulement pour Bozo
                        #+ et les autre utilisateurs ont une erreur.



# Quelque chose comme ceci peut être utile dans certains contextes.
NO_EXIT=1   # Active la définition de fonction ci-dessous.

[[ $NO_EXIT -eq 1 ]] && exit() { true; }     # Définition de fonction dans une "liste
ET".
# Si $NO_EXIT vaut 1, déclare "exit ()".
# Ceci désactive la commande intégrée "exit" en créant un alias vers "true".

exit  # Appelle la fonction "exit ()", et non pas la commande intégrée "exit".

# Ou de façon similaire :
fichier=fichier1

[ -f "$fichier" ] &&
foo () { rm -f "$fichier"; echo "Fichier "$fichier" supprimé."; } ||
foo () { echo "Fichier "$fichier" introuvable."; touch bar; }

foo

# Merci, S.C. et Christopher Head
```

Note

Qu'arrive-t'il quand différentes versions de la même fonction apparaissent dans un script ?

```
#  Comme Yan Chen le précise,
#  quand une fonction est définie plusieurs fois,
#  la version finale est celle qui est appelée.
#  Néanmoins, ce n'est pas particulièrement utile.

fonction ()
{
  echo "Première version de func ()."
}

fonction ()
{
  echo "Deuxième version de func ()."
}

fonction   # Deuxième version de func ().

exit $?

#  Il est même possible d'utiliser des fonctions pour surcharger ou
#+ préempter les commandes systèmes.
#  Bien sûr, ce n'est *pas* conseillé.
```

23.1. Fonctions complexes et complexité des fonctions

Les fonctions peuvent récupérer des arguments qui leur sont passés et renvoyer un code de sortie44 au script pour utilisation ultérieure.

```
nom_fonction $arg1 $arg2
```

La fonction se réfère aux arguments passés par leur position (comme s'ils étaient des paramètres positionnels), c'est-à-dire `$1`, `$2` et ainsi de suite.

Exemple 23.2. Fonction prenant des paramètres

```
#!/bin/bash
# Fonctions et paramètres

DEFAUT=defaut                               # Valeur par défaut.

fonc2 () {
   if [ -z "$1" ]                           # Est-ce que la taille du paramètre
                                            # #1 a une taille zéro ?
   then
     echo "-Le paramètre #1 a une taille nulle.-"  # Ou aucun paramètre n'est passé.
   else
     echo "-Le paramètre #1 est \"$1\".-"
   fi

   variable=${1-$DEFAUT}                    #  Que montre la substitution de
   echo "variable = $variable"              #+ paramètre?
                                            #  --------------------------
                                            #  Elle distingue entre pas de
                                            #+ paramètre et un paramètre nul.

   if [ "$2" ]
   then
     echo "-Le paramètre #2 est \"$2\".-"
   fi

   return 0
}

echo

echo "Aucun argument."
fonc2                           # Appelé sans argument
echo


echo "Argument de taille nulle."
fonc2 ""                        # Appelé avec un paramètre de taille zéro
echo

echo "Paramètre nul."
fonc2 "$parametre_non_initialise" # Appelé avec un paramètre non initialisé
echo

echo "Un paramètre."
fonc2 premier                   # Appelé avec un paramètre
echo

echo "Deux paramètres."
fonc2 premier second            # Appelé avec deux paramètres
echo

echo "\"\" \"second\" comme argument."
fonc2 "" second                 # Appelé avec un premier paramètre de taille nulle
echo                            # et une chaîne ASCII pour deuxième paramètre.

exit 0
```

Important

> La commande shift35 fonctionne sur les arguments passés aux fonctions (voir l'Exemple 33.15, « Astuce de valeur de retour »).

Mais, qu'en est-t'il des arguments en ligne de commande passés au script ? Une fonction les voit-elle ? Il est temps de dissiper toute confusion.

Exemple 23.3. Fonctions et arguments en ligne de commande passés au script

```
#!/bin/bash
# func-cmdlinearg.sh
#  Appelez ce script avec un argument en ligne de commande,
#+ quelque chose comme $0 arg1.


fonction ()

{
echo "$1"
}

echo "premier appel à la fonction : aucun argument passé."
echo "Vérifie si la ligne de commande a été vue."
fonction
# Non ! Argument en ligne de commande non vu.

echo "============================================================="
echo
echo "Second appel à la fonction : argument en ligne de commande passé"
echo "explicitement."
fonction $1
# Maintenant, il est vu !

exit 0
```

Note

Contrairement à d'autres langages de programmation, normalement, les scripts shell passent seulement des paramètres par valeur aux fonctions. Les noms de variable (qui sont réellement des *pointeurs*), s'ils sont passés en tant que paramètres aux fonctions, seront traités comme des chaînes littérales. *Les fonctions interprètent leurs arguments littéralement.*

Les références de variables indirectes (voir l'Exemple 34.2, « Références de variables indirectes - la nouvelle façon ») apportent une espèce de mécanisme peu pratique pour passer des pointeurs aux fonctions.

Exemple 23.4. Passer une référence indirecte à une fonction

```
#!/bin/bash
# ind-func.sh : Passer une référence indirecte à une fonction.

echo_var ()
{
echo "$1"
}

message=Bonjour
Bonjour=Aurevoir

echo_var "$message"        # Bonjour
# Maintenant, passons une référence indirecte à la fonction.
echo_var "${!message}"     # Aurevoir

echo "-------------"
```

```
# Qu'arrive-t'il si nous changeons le contenu de la variable "Bonjour" ?
Bonjour="Bonjour, de nouveau !"
echo_var "$message"        # Bonjour
echo_var "${!message}"     # Bonjour, de nouveau !

exit 0
```

La prochaine question logique est de savoir si les paramètres peuvent être déréférencés *après* avoir été passé à une fonction.

Exemple 23.5. Déréférencer un paramètre passé à une fonction

```
#!/bin/bash
# dereference.sh
# Déréférence un paramètre passé à une fonction.
# Script de Bruce W. Clare.

dereference ()
{
     y=\$"$1"   # Nom de la variable.
     echo $y    # $Crotte

     x=`eval "expr \"$y\" "`
     echo $1=$x
     eval "$1=\"Un texte différent \""  # Affecte une nouvelle valeur.
}

Crotte="Un texte"
echo $Crotte "avant"   # Un texte avant

dereference Crotte
echo $Junk "après"     # Un texte différent après

exit 0
```

Exemple 23.6. De nouveau, déréférencer un paramètre passé à une fonction

```
#!/bin/bash
# ref-params.sh : Déréférencer un paramètre passé à une fonction.
#                 (exemple complexe)

ITERATIONS=3  # Combien de fois obtenir une entrée.
icompteur=1

ma_lecture () {
  # Appelé avec ma_lecture nomvariable,
  # Affiche la précédente valeur entre crochets comme valeur par défaut,
  # et demande une nouvelle valeur.

  local var_locale

  echo -n "Saisissez une valeur "
  eval 'echo -n "[$'$1'] "'  # Valeur précédente.
# eval echo -n "[\$$1] "     #  Plus facile à comprendre,
                             #+ mais perd l'espace de fin à l'invite de l'utilisateur.
  read var_locale
  [ -n "$var_locale" ] && eval $1=\$var_locale

  # "liste-ET" : si "var_locale", alors l'initialiser à "$1".
}

echo

while [ "$icompteur" -le "$ITERATIONS" ]
do
  ma_lecture var
```

```
  echo "Entrée #$icompteur = $var"
  let "icompteur += 1"
  echo
done


# Merci à Stephane Chazelas pour nous avoir apporté cet exemple instructif.

exit 0
```

Sortie et retour

code de sortie

Les fonctions renvoient une valeur, appelée un *code (ou état) de sortie*. Le code de sortie peut être explicitement spécifié par une instruction **return**, sinon, il s'agit du code de sortie de la dernière commande de la fonction (0 en cas de succès et une valeur non nulle sinon). Ce status de sortie44 peut être utilisé dans le script en le référençant à l'aide de la variable $?. Ce mécanisme permet effectivement aux fonctions des scripts d'avoir une « valeur de retour » similaire à celle des fonctions C.

return

Termine une fonction. Une commande **return** [1] prend optionnellement un argument de type *entier*, qui est renvoyé au script appelant comme « code de sortie » de la fonction, et ce code de sortie est affecté à la variable $?.

Exemple 23.7. Maximum de deux nombres

```
#!/bin/bash
# max.sh : Maximum de deux entiers.

E_PARAM_ERR=250    # Si moins de deux paramètres passés à la fonction.
EGAL=251           # Code de retour si les deux paramètres sont égaux.
#  Valeurs de l'erreur en dehors de la plage de tout paramètre
#+ qui pourrait être fourni à la fonction.

max2 ()            # Envoie le plus important des deux entiers.
{                  # Note: les nombres comparés doivent être plus petits que 257.
if [ -z "$2" ]
then
  return $E_PARAM_ERR
fi

if [ "$1" -eq "$2" ]
then
  return $EGAL
else
  if [ "$1" -gt "$2" ]
  then
    return $1
  else
    return $2
  fi
fi
}

max2 33 34
return_val=$?

if [ "$return_val" -eq $E_PARAM_ERR ]
then
        echo "Vous devez donner deux arguments à la fonction."
elif [ "$return_val" -eq $EGAL ]
  then
    echo "Les deux nombres sont identiques."
else
    echo "Le plus grand des deux nombres est $return_val."
fi
```

[1] La commande **return** est une commande intégrée159 Bash.

```
exit 0

#  Exercice (facile) :
#  -------------------
#  Convertir ce script en une version interactive,
#+ c'est-à-dire que le script vous demande les entrées (les deux nombres).
```

Astuce

Pour qu'une fonction renvoie une chaîne de caractères ou un tableau, utilisez une variable dédiée.

```
compte_lignes_dans_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPONSE=$(echo $(wc -l   /etc/passwd))
  # Si /etc/passwd est lisible, met dans REPONSE le nombre de lignes.
  # Renvoie une valeur et un statut.
  # Le 'echo' ne semble pas nécessaire mais...
  # il supprime les espaces blancs excessifs de la sortie.
}

if compte_ligne_dans_etc_passwd
then
  echo "Il y a $REPONSE lignes dans /etc/passwd."
else
  echo "Ne peut pas compter les lignes dans /etc/passwd."
fi

# Merci, S.C.
```

Exemple 23.8. Convertir des nombres en chiffres romains

```
#!/bin/bash

# Conversion d'un nombre arabe en nombre romain
# Échelle : 0 - 200
# C'est brut, mais cela fonctionne.

# Étendre l'échelle et améliorer autrement le script est laissé en exercice.

# Usage: romain nombre-a-convertir

LIMITE=200
E_ERR_ARG=65
E_HORS_ECHELLE=66

if [ -z "$1" ]
then
  echo "Usage: `basename $0` nombre-a-convertir"
  exit $E_ERR_ARG
fi

num=$1
if [ "$num" -gt $LIMITE ]
then
  echo "En dehors de l'échelle !"
  exit $E_HORS_ECHELLE
fi

vers_romain ()   # Doit déclarer la fonction avant son premier appel.
{
nombre=$1
facteur=$2
```

```
rchar=$3
let "reste = nombre - facteur"
while [ "$reste" -ge 0 ]
do
  echo -n $rchar
  let "nombre -= facteur"
  let "reste = nombre - facteur"
done

return $nombre
       # Exercice :
       # ---------
       # Expliquer comment fonctionne cette fonction.
       # Astuce : division par une soustraction successive.
}


vers_romain $nombre 100 C
nombre=$?
vers_romain $nombre 90 XC
nombre=$?
vers_romain $nombre 50 L
nombre=$?
vers_romain $nombre 40 XL
nombre=$?
vers_romain $nombre 10 X
nombre=$?
vers_romain $nombre 9 IX
nombre=$?
vers_romain $nombre 5 V
nombre=$?
vers_romain $nombre 4 IV
nombre=$?
vers_romain $nombre 1 I

echo

exit 0
```

Voir aussi l'Exemple 10.28, « Vérification d'une entrée alphabétique ».

Important

L'entier positif le plus grand qu'une fonction peut renvoyer est 255. La commande **return** est très liée au code de sortie44, qui tient compte de cette limite particulière. Heureusement, il existe quelques astuces457 pour ces situations réclamant une valeur de retour sur un grand entier.

Exemple 23.9. Tester les valeurs de retour importantes dans une fonction

```
#!/bin/bash
# return-test.sh

# La plus grande valeur positive qu'une fonction peut renvoyer est 255.

test_retour ()         # Renvoie ce qui lui est passé.
{
  return $1
}

test_retour 27         # OK.
echo $?                # Renvoie 27.

test_retour 255        # Toujours OK.
echo $?                # Renvoie 255.

test_retour 257        # Erreur!
```

```
echo $?                # Renvoie 1 (code d'erreur divers).

test_retour -151896    #  Néanmoins, les valeurs négatives peuvent être
plus
                       #+ importantes.
echo $?                # Renvoie -151896.
# ==================================================
test_retour -151896    #  Est-ce que les grands nombres négatifs vont
                       #+ fonctionner ?
echo $?                # Est-ce que ceci va renvoyer -151896?
                       # Non! Il renvoie 168.
#  Les versions de Bash antérieures à la 2.05b permettaient les codes de
retour
#+ au format d'un grand entier négatif.
#  Les nouvelles versions ont corrigées cette faille.
#  Ceci peut casser les anciens scripts.
#  Attention !
# ==================================================

exit 0
```

Un contournement pour obtenir des « codes de retour » au format entier long est de tout simplement affecter le « code de retour » à une variable globale.

```
Val_Retour=   #  Variable globale pour recevoir une valeur de retour
              #+ d'une taille trop importante.

alt_return_test ()
{
  fvar=$1
  Val_Retour=$fvar
  return   # Renvoie 0 (succès).
}

alt_return_test 1
echo $?                                 # 0
echo "valeur de retour = $Val_Retour"    # 1

alt_return_test 256
echo "valeur de retour = $Val_Retour"    # 256

alt_return_test 257
echo "valeur de retour = $Val_Retour"    # 257

alt_return_test 25701
echo "valeur de retour = $Val_Retour"    #25701
```

Une méthode plus élégante est de demander à la fonction d'afficher (via **echo**) son « code de retour » sur `stdout` et de le capturer par substitution de commandes. Voir la discussion de ceci457 dans la Section 33.8, « Astuces assorties ».

Exemple 23.10. Comparer deux grands entiers

```
#!/bin/bash
# max2.sh : Maximum de deux GRANDS entiers.

#  Ceci correspond au précédent exemple "max.sh", modifié pour permettre
la
#+ comparaison de grands entiers.

EGAL=0             # Code de retour si les deux paramètres sont égaux.
E_PARAM_ERR=99999  # Pas assez de paramètres fournis à la fonction.
#          ^^^^^^   En dehors de la plage de tout paramètre fourni

max2 ()            # Renvoie le plus gros des deux nombres.
{
```

```
if [ -z "$2" ]
then
  echo $E_PARAM_ERR
  return
fi

if [ "$1" -eq "$2" ]
then
  echo $EGAL
  return
else
  if [ "$1" -gt "$2" ]
  then
    retval=$1
  else
    retval=$2
  fi
fi


echo $retval        # Affiche (sur stdout) plutôt que de retourner la
valeur.
                    # Pourquoi ?
}


valeur_retour=$(max2 33001 33997)
#               ^^^^             nom de la fonction
#                    ^^^^^ ^^^^^ paramètres fournis
#  C'est en fait une forme de substitution de commandes :
#+ traiter une fonction comme s'il s'agissait d'une commande
#+ et affecter la sortie de la fonction à la variable "valeur_retour".


# ========================= SORTIE ======================

if [ "$valeur_retour" -eq "$E_PARAM_ERR" ]
  then
  echo "Erreur : Pas assez de paramètres passés à la fonction de
comparaison."
elif [ "$valeur_retour" -eq "$EGAL" ]
  then
    echo "Les deux nombres sont égaux."
else
    echo "Le plus grand des deux nombres est $valeur_retour."
fi

exit 0

# =======================================================


#  Exercices :
#  ----------
#  1) Trouvez un moyen plus élégant pour tester les paramètres passés à la
#+    fonction.
#  2) Simplifiez la structure du if/then à partir de "SORTIE".
#  3) Réécrire le script pour prendre en entrée des paramètres de la ligne
de
#     commande.
```

Voici un autre exemple de capture de la « valeur de retour » d'une fonction. Le comprendre requiert quelques connaissances d'awk634.

```
longueur_mois () # Prend le numéro du mois en argument.
{                # renvoie le nombre de jours dans ce mois.
moisJ="31 28 31 30 31 30 31 31 30 31 30 31"  # Déclaré en tant que local ?
echo "$moisJ" | awk '{ print $'"${1}"' }'    # Astuce.
```

```
#                               ^^^^^^^^
# Paramètre passé à la fonction   ($1 -- numéro du mois), puis à awk.
# Awk voit ceci comme "print $1 . . . print $12" (suivant le numéro du
mois)
# Modèle pour passer un paramètre à un script awl embarqué :
#                                 $'"${script_parametre}"'

#  Besoin d'une vérification d'erreurs pour les paramètres de l'échelle
(1-12)
#+ et pour l'année bissextile avec février.
}

# ------------------------------------------------
# Exemple d'utilisation :
mois=4        # avril, par exemple (4è mois).
journees=$(longueur_mois $mois)
echo $journees  # 30
# ------------------------------------------------
```

Voir aussi l'Exemple A.7, « *days-between* : Calculer le nombre de jours entre deux dates ».

Exercice: Utiliser ce que nous venons d'apprendre, étendre l'exemple précédent sur les nombres romains pour accepter une entrée arbitrairement grande.

Redirection

Rediriger le stdin d'une fonction

Une fonction est essentiellement un bloc de code, ce qui signifie que `stdin` peut être redirigé (comme dans l'Exemple 3.1, « Blocs de code et redirection d'entrées/sorties »).

Exemple 23.11. Vrai nom pour un utilisateur

```
#!/bin/bash
# realname.sh

# À partir du nom utilisateur, obtenir le "vrai nom" dans /etc/passwd.

NBARGS=1          # Attend un arg.
E_MAUVAISARGS=65

fichier=/etc/passwd
modele=$1

if [ $# -ne "$NBARGS" ]
then
  echo "Usage : `basename $0` NOMUTILISATEUR"
  exit $E_MAUVAISARGS
fi

partie_fichier ()  # Parcours le fichier pour trouver le modèle,
                   #+ la portion pertinente des caractères de la ligne.
{
while read ligne  # "while" n'a pas nécessairement besoin d'une "[ condition]"
do
  echo "$ligne" | grep $1 | awk -F":" '{ print $5 }'
      # awk utilise le délimiteur ":".
done
} <$fichier  # Redirige dans le stdin de la fonction.

partie_fichier $modèle

# Oui, le script entier peut être réduit en
#       grep MODELE /etc/passwd | awk -F":" '{ print $5 }'
# ou
#       awk -F: '/MODELE/ {print $5}'
```

```
# ou
#       awk -F: '($1 == "nomutilisateur") { print $5 }' # vrai nom à partir du nom
utilisateur
# Néanmoins, ce n'est pas aussi instructif.

exit 0
```

Il existe une autre méthode, certainement moins compliquée, de rediriger le `stdin` d'une fonction. Celle-ci fait intervenir la redirection de `stdin` vers un bloc de code entre accolades contenu à l'intérieur d'une fonction.

```
# Au lieu de :
Fonction ()
{
 ...
 } < fichier

# Essayez ceci :
Fonction ()
{
  {
    ...
    } < fichier
}

# De façon similaire,

Fonction ()  # Ceci fonctionne.
{
  {
   echo $*
  } | tr a b
}

Fonction ()  # Ceci ne fonctionne pas.
{
  echo $*
} | tr a b   # Un bloc de code intégré est obligatoire ici.


# Merci, S.C.
```

23.2. Variables locales

Que fait une variable *locale* ?

variables locales

Une variable déclarée *localement* n'est visible qu'à l'intérieur du bloc de code dans laquelle elle apparaît. Elle a une « visibilité » locale. Dans une fonction, une *variable locale* n' a une signification qu'à l'intérieur du bloc de la fonction.

Exemple 23.12. Visibilité de la variable locale

```
#!/bin/bash
# Variables globales et locales à l'intérieur d'une fonction.

fonc ()
{
  local var_local=23       # Déclaré en tant que variable locale.
  echo                     # Utilise la commande intégrée locale.
  echo "\"var_local\" dans la fonction = $var_local"
  var_global=999           # Non déclarée en local.
                           # Retour en global.
  echo "\"var_global\" dans la fonction = $var_global"
}
```

```
fonc

# Maintenant, voyons s'il existe une variable locale en dehors de la fonction.

echo
echo "\"var_loc\" en dehors de la fonction = $var_loc"
                                     # "var_loc" en dehors de la fonction =
                                     # Non, $var_local n'est pas visible
globalement.
echo "\"var_global\" en dehors de la fonction = $var_global"
                                     # "var_global" en dehors de la fontion = 999
                                     # $var_global est visible globalement.
echo

exit 0
#  Au contraire de C, une variable Bash déclarée dans une fonction n'est locale
#+ que si elle est déclarée ainsi.
```

Attention

Avant qu'une fonction ne soit appelée, *toutes* les variables déclarées dans la fonction sont invisibles à l'extérieur du corps de la fonction, et pas seulement celles déclarées explicitement *locales*.

```
#!/bin/bash

func ()
{
var_globale=37   #  Visible seulement à l'intérieur du bloc de la fonction
                 #+ avant que la fonction ne soit appelée.
}                # FIN DE LA FONCTION

echo "var_globale = $var_globale" # var_globale =
                                  #  La fonction "func" n'a pas encore été
appelée,
                                  #+ donc $var_globale n'est pas visible
ici.

func
echo "var_globale = $var_globale" # var_globale = 37
                                  # A été initialisée par l'appel de la
fonction.
```

23.2.1. Variables locales et récursion

La *récursion* est une forme intéressante et quelque fois utile d' *auto-référence*.Herbert Mayer la définit comme « ... exprimant un algorithme en utilisant une version simplifiée du même algorithme... »

Considérez une définition définie par elle-même, [2] une expression implicite dans sa propre expression, [3] *un serpent mangeant sa propre queue*, [4] ou... une fonction qui s'appelle elle-même. [5]

Exemple 23.13. Démonstration d'une fonction récursive simple

```
#!/bin/bash
# recursion-demo.sh
# Démonstration de la récursion.
```

[2] Autrement connue sous le nom de *redondance*.
[3] Autrement connue sous le nom de *tautologie*.
[4] Autrement connue sous le nom de *métaphore*.
[5] Autrement connue sous le nom de *fonction récursive*.

```
RECURSIONS=9   # Nombre de récursions.
r_count=0      # Doit être global. Pourquoi ?

recurse ()
{
  var="$1"

  while [ "$var" -ge 0 ]
  do
    echo "Recursion count = "$r_count"  +-+  \$var = "$var""
    (( var-- )); (( r_count++ ))
    recurse "$var"  #  La fonction s'appelle elle-même (recurse)
  done              #+ jusqu'à ce que la condition soit rencontrée ?
}

recurse $RECURSIONS

exit $?
```

Les variables locales sont un outil intéressant pour écrire du code récursif, mais cette pratique implique généralement un grand moment de réflexion et n'est réellement *pas* recommendée dans un script shell. [6]

Exemple 23.14. Récursion en utilisant une variable locale

```
#!/bin/bash

#               facteurs
#               ---------


# Bash permet-il la récursion ?
# Eh bien, oui, mais...
# C'est si lent que vous devrez vous accrocher pour y arriver.


MAX_ARG=5
E_MAUVAIS_ARGS=65
E_MAUVAISE_ECHELLE=66


if [ -z "$1" ]
then
  echo "Usage : `basename $0` nombre"
```

[6] Trop de niveaux de récursion pourrait arrêter brutalement un script avec une erreur de segmentation.

```
#!/bin/bash

#  Attention: Lancer ce script pourrait empêcher le bon fonctionnement de votre
#+ système !
#  Si vous êtes chanceux, il finira avec une erreur de segmentation avant
#+ d'avoir utiliser toute la mémoire disponible.

fonction_recursive ()
{
echo "$1"     # Fait en sorte que le fonction fait quelque chose et accélère le "segfault".
(( $1 < $2 )) && fonction_recursive $(( $1 + 1 )) $2;
#  Aussi longtemps que le premier paramètres est plus petit que le second,
#+ incrémente le premier et fait une récursion.
}

fonction_recursive 1 50000  # Récursion sur 50.000 niveaux!
#  Grande chance d'obtenir une erreur de segmentation (ceci dépendant de la
#+ taille de la pile, configurée avec ulimit -m).

#  Une récursion d'une telle profondeur peut même arrêter un programme C avec
#+ une erreur de segmentation, suite à l' utilisation de toute la mémoire
#+ allouée à la pile.

echo "Ceci ne s'affichera probablement pas."
exit 0  # Ce script ne finira par normalement.

# Merci, Stéphane Chazelas.
```

```
  exit $E_MAUVAIS_ARGS
fi

if [ "$1" -gt $MAX_ARG ]
then
  echo "En dehors de l'échelle (5 est le maximum)."
  # Maintenant, allons-y.
  #  Si vous souhaitez une échelle plus importante, réécrivez-le dans un vrai
  #+ langage de programmation.
  exit $E_MAUVAISE_ECHELLE
fi

fact ()
{
  local nombre=$1
  # La variable "nombre" doit être déclarée en local.
  # Sinon cela ne fonctionne pas.
  if [ "$nombre" -eq 0 ]
  then
    factoriel=1    # Le factoriel de 0 = 1.
  else
    let "decrnum = nombre - 1"
    fact $decrnum  # Appel à la fonction récursive (la fonction s'appelle elle-même).
    let "factoriel = $nombre * $?"
  fi

  return $factoriel
}

fact $1
echo "Le factoriel de $1 est $?."

exit 0
```

Voir aussi l'Exemple A.16, « *primes*: Générer des nombres premiers en utilisant l'opérateur modulo » pour un exemple de récursion dans un script. Faites attention que la récursion demande beaucoup de ressources et s'exécute lentement. Son utilisation n'est donc pas appropriée dans un script.

23.3. Récursion sans variables locales

Une fonction peut s'appeller récursivement sans même utiliser de variables locales.

Exemple 23.15. ***La séquence de Fibonacci***

```
#!/bin/bash
# fibo.sh : Fibonacci sequence (recursive)
# Author: M. Cooper
# License: GPL3

# ---------------------------------
# Fibo(0) = 0
# Fibo(1) = 1
# else
#   Fibo(j) = Fibo(j-1) + Fibo(j-2)
# ---------------------------------

MAXTERM=15       # Number of terms (+1) to generate.
MINIDX=2         # If idx is less than 2, then Fibo(idx) = idx.

Fibonacci ()
{
  idx=$1   # Doesn't need to be local. Why not?
  if [ "$idx" -lt "$MINIDX" ]
  then
    echo "$idx"  # First two terms are 0 1 ... see above.
  else
```

```
    (( --idx ))  # j-1
    term1=$( Fibonacci $idx )   #  Fibo(j-1)

    (( --idx ))  # j-2
    term2=$( Fibonacci $idx )   #  Fibo(j-2)

    echo $(( term1 + term2 ))
  fi
  #  An ugly, ugly kludge.
  #  The more elegant implementation of recursive fibo in C
  #+ is a straightforward translation of the algorithm in lines 7 - 10.
}

for i in $(seq 0 $MAXTERM)
do  # Calculate $MAXTERM+1 terms.
  FIBO=$(Fibonacci $i)
  echo -n "$FIBO "
done
# 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
# Takes a while, doesn't it? Recursion in a script is slow.

echo

exit 0
```

Exemple 23.16. ***Les tours d'Hanoi***

```
#! /bin/bash
#
# La tour d'Hanoi
# Script bash
# Copyright (C) 2000 Amit Singh. All Rights Reserved.
# http://hanoi.kernelthread.com
#
# Testé avec bash version 2.05b.0(13)-release
# Fonctionne aussi avec Bash version 3.x.
#
#  Utilisé dans le "Guide d'écriture avancé des scripts Bash"
#+ Avec l'autorisation de l'auteur du script.
#  Légèrement modifié et commenté par l'auteur d'ABS.

#==========================================================================#
#  La tour d'Hanoi est un puzzle mathématique attribué à Édouard Lucas,
#+ un mathématicien français du 19è siècle.
#  Il y a un ensemble de trois positions verticales dans une base.
#  Le premier poste dispose d'un ensemble d'anneaux empilés.
#  Les anneaux sont des disques plats avec un trou en leur centre,
#+ de manière à être placés sur les batons.
#  Les anneaux ont des diamètres différents et ils s'assemblent en ordre
#+ descendant suivant leur taille.
#  La plus petite est au-dessus et la plus large à la base.
#
#  Le problème consiste à transférer la pile d'anneaux d'un baton à un autre.
#  Vous pouvez bouger seulement un anneau à la fois.
#  Il vous est permis de replacer les anneaux à leur baton d'origine.
#  Vous pouvez placer un petit anneau sur un plus gros mais pas le contraire.
#  Encore une fois, il est interdit de placer un gros anneau sur un plus petit.
#
#  Pour un petit nombre d'anneaux, seuls quelques mouvements sont nécessaires.
#+ Pour chaque anneau supplémentaire, le nombre de déplacements requis double
#+ approximativement et la "stratégie" devient de plus en plus complexe.
#
#  Pour plus d'informations, voir http://hanoi.kernelthread.com
#+ ou pp. 186-92 de _The Armchair Universe_ par A.K. Dewdney.
#
#
```

```
#          ...                    ...                    ...
#           | |                    | |                    | |
#         _|_|_                   | |                    | |
#        |_____|                  | |                    | |
#       |_______|                 | |                    | |
#      |_________|                | |                    | |
#     |___________|               | |                    | |
#    |             |              | |                    | |
# .-------------------------------------------------------------.
# |*************************************************************|
#          #1                    #2                     #3
#
#=============================================================#


E_SANSPARAM=66    # Aucun paramètre passé au script.
E_MAUVAISPARAM=67 # Nombre illégal de disques.
Deplacements=     # Variable globale contenant le nombre de déplacements.
                  # Modification du script original.

fait_hanoi() {   # Fonction récursive.
    case $1 in
    0)
        ;;
    *)
        fait_hanoi "$(($1-1))" $2 $4 $3
        echo move $2 "-->" $3
        ((Deplacements++))  # Modification du script original.
        fait_hanoi "$(($1-1))" $4 $3 $2
        ;;
    esac
}

case $# in
    1) case $(($1>0)) in     # Il doit y avoir au moins un disque.
       1) # Instruction case imbriquée.
          fait_hanoi $1 1 3 2
          echo "Nombre total de déplacements = $Deplacements"
          exit 0;
          ;;
       *)
          echo "$0: valeur illégale pour le nombre de disques";
          exit $E_MAUVAISPARAM;
          ;;
       esac
       ;;
    *)
       echo "usage: $0 N"
       echo "       où \"N\" est le nombre de disques."
       exit $E_SANSPARAM;
       ;;
esac

# Exercices:
# ---------
# 1) Est-ce que les commandes au delà de ce point seront exécutées ?
#    Pourquoi ? (Facile)
# 2) Expliquer le fonctionnement de la fonction "fait_hanoi".
#    (Difficile -- voir la référence sur Dewdney ci-dessus.)
```

Chapitre 24. Alias

Un *alias* Bash n'est essentiellement rien de plus qu'un raccourci clavier, une abréviation, un moyen d'éviter de taper une longue séquence de commande. Si, par exemple, nous incluons **alias lm="ls -l | more"** dans le fichier `~/.bashrc`, alors chaque **lm** saisi sur la ligne de commande sera automatiquement remplacé par un **ls -l | more**. Ceci peut économiser beaucoup de temps lors de saisies en ligne de commande et éviter d'avoir à se rappeler des combinaisons complexes de commandes et d'options. Disposer de **alias rm="rm -i"** (suppression en mode interactif) peut vous empêcher de faire des bêtises car il prévient la perte par inadvertance de fichiers importants.

Dans un script, les alias ont une utilité très limitée. Il serait assez agréable que les alias assument certaines des fonctionnalités du préprocesseur C, telles que l'expansion de macros, mais malheureusement Bash ne supporte pas l'expansion d'arguments à l'intérieur du corps des alias. [1] Pire encore, un script échoue à étendre un alias lui-même à l'intérieur d'une « construction composée », telle que les instructions if/then46, les boucles et les fonctions. Une limitation supplémentaire est qu'un alias ne peut être étendu récursivement. De façon pratiquement invariable, tout ce que nous voudrions que les alias puissent faire est faisable bien plus efficacement avec une fonction352.

Exemple 24.1. Alias à l'intérieur d'un script

```
#!/bin/bash
#  alias.sh

shopt -s expand_aliases
# Cette option doit être activée, sinon le script n'étendra pas les alias.


# Tout d'abord, un peu d'humour.
alias Jesse_James='echo "\"Alias Jesse James\" était une comédie de 1959 avec Bob
Hope."'
Jesse_James

echo; echo; echo;

alias ll="ls -l"
#  Vous pouvez utiliser soit les simples guillemets (') soit les doubles (") pour
définir
#+ un alias.

echo "Essai de l'alias \"ll\" :"
ll /usr/X11R6/bin/mk*   #* L'alias fonctionne.

echo

repertoire=/usr/X11R6/bin/
prefixe=mk*  # Voir si le caractère joker va causer des problèmes.
echo "Les variables \"repertoire\" + \"prefixe\" = $repertoire$prefixe"
echo

alias lll="ls -l $repertoire$prefixe"

echo "Essai de l'alias \"lll\":"
lll         # Longue liste de tous les fichiers de /usr/X11R6/bin commençant avec mk.
# Les alias peuvent gérer les variables concaténées -- incluant les caractères joker.




VRAI=1

echo

if [ VRAI ]
then
  alias rr="ls -l"
```

[1] Néanmoins, les alias semblent étendre les paramètres de position.

```
  echo "Essai de l'alias \"rr\" à l'intérieur d'une instruction if/then :"
  rr /usr/X11R6/bin/mk*   #* Message d'erreur !
  # Les alias ne sont pas étendus à l'intérieur d'instructions composées.
  echo "Néanmoins, l'alias précédemment étendu est toujours reconnu :"
  ll /usr/X11R6/bin/mk*
fi

echo

nombre=0
while [ $nombre -lt 3 ]
do
  alias rrr="ls -l"
  echo "Essai de l'alias \"rrr\" à l'intérieur de la boucle \"while\":"
  rrr /usr/X11R6/bin/mk*   #* L'alias ne sera pas étendu ici non plus.
                           #  alias.sh: line 57: rrr: command not found
  let nombre+=1
done

echo; echo

alias xyz='cat $0'   # Le script se liste lui-même.
                     # Notez les simples guillemets.
xyz
#  Ceci semble fonctionne,
#+ bien que la documentation Bash suggère que cela ne le devrait pas.
#
#  Néanmoins, comme l'indique Steve Jacobson,
#+ le paramètre "$0" s'étend tout de suite après la déclaration de l'alias.

exit 0
```

La commande **unalias** supprime un *alias* précédemment configuré.

Exemple 24.2. *unalias* : Configurer et supprimer un alias

```
#!/bin/bash
# unalias.sh

shopt -s expand_aliases  # Active l'expansion d'alias.

alias llm='ls -al | more'
llm

echo

unalias llm              # Supprime la configuration de l'alias.
llm
# Résulte en un message d'erreur car 'llm' n'est plus reconnu.

exit 0
```

```
bash$ ./unalias.sh

total 6
drwxrwxr-x    2 bozo     bozo         3072 Feb  6 14:04 .
drwxr-xr-x   40 bozo     bozo         2048 Feb  6 14:04 ..
-rwxr-xr-x    1 bozo     bozo          199 Feb  6 14:04 unalias.sh

./unalias.sh: llm: command not found
```

Chapitre 25. Constructeurs de listes

Les constructions de « liste ET » et de « liste OR » apportent un moyen de réaliser un certain nombre de commandes consécutivement. Elles peuvent remplacer efficacement des **if**/**then** complexes, voire imbriqués ou même des instructions **case**.

Chaîner des commandes

liste ET

```
commande-1 && commande-2 && commande-3 && ...
commande-n
```

Chaque commande s'exécute à son tour à condition que la dernière commande ait renvoyé un code de retour true (zéro). Au premier retour false (différent de zéro), la chaîne de commande s'arrête (la première commande renvoyant false est la dernière à être exécutée).

Exemple 25.1. Utiliser une *liste ET* pour tester des arguments de la ligne de commande

```
#!/bin/bash
# "liste ET"

if [ ! -z "$1" ] && echo "Argument #1 = $1" && [ ! -z "$2" ] \
&& echo "Argument #2 = $2"
then
  echo "Au moins deux arguments passés au script."
  # Toute la commande chaînée doit être vraie.
else
  echo "Moins de deux arguments passés au script."
  # Au moins une des commandes de la chaîne a renvoyé faux.
fi
# Notez que "if [ ! -z $1 ]" fonctionne mais que son supposé équivalent,
#  if [ -n $1 ] ne fonctionne pas.
#    Néanmoins, mettre entre guillemets corrige cela :
#  if [ -n "$1" ] fonctionne.
#    Attention !
# Il est toujours mieux de mettre entre guillemets les variables testées.


# Ceci accomplit la même chose en utilisant une instruction if/then pure.
if [ ! -z "$1" ]
then
  echo "Argument #1 = $1"
fi
if [ ! -z "$2" ]
then
  echo "Argument #2 = $2"
  echo "Au moins deux arguments passés au script."
else
  echo "Moins de deux arguments passés au script."
fi
# C'est plus long et moins élégant que d'utiliser une "liste ET".


exit 0
```

Exemple 25.2. Un autre test des arguments de la ligne de commande en utilisant une *liste and*

```
#!/bin/bash

ARGS=1            # Nombre d'arguments attendus.
E_MAUVAISARGS=65  # Valeur de sortie si un nombre incorrect d'arguments est passé.

test $# -ne $ARGS && \
echo "Usage: `basename $0` $ARGS argument(s)" && exit $E_MAUVAISARGS
#  Si la condition 1 est vraie (mauvais nombre d'arguments passés au script),
```

```
#+ alors le reste de la ligne s'exécute et le script se termine.

# La ligne ci-dessous s'exécute seulement si le test ci-dessus a échoué.
echo "Bon nombre d'arguments passés à ce script."

exit 0

# Pour vérifier la valeur de sortie, faites un "echo $?" après la fin du script.
```

Bien sûr, une *liste ET* peut aussi *initialiser* des variables à une valeur par défaut.

```
arg1=$@ && [ -z "$arg1" ] && arg1=DEFAULT

              # Initialise $arg1 avec les arguments en ligne de commande.
              # Mais... Initialise à DEFAUT si non spécifié sur la ligne de
commande.
```

liste OR

```
commande-1 || commande-2 || commande-3 || ...
commande-n
```

Chaque commande s'exécute à son tour aussi longtemps que la commande précédente renvoie false. Au premier retour true, la chaîne de commandes s'arrête (la première commande renvoyant true est la dernière à être exécutée). C'est évidemment l'inverse de la « liste ET ».

Exemple 25.3. Utiliser des *listes OR* en combinaison avec une *liste ET*

```
#!/bin/bash

#  delete.sh, utilitaire pas-si-stupide de suppression de fichier.
#  Usage : delete nomfichier

E_MAUVAISARGS=65

if [ -z "$1" ]
then
  echo "Usage : `basename $0` nomfichier"
  exit $E_MAUVAISARGS  # Pas d'argument ? On sort.
else
  fichier=$1           # Initialisation du nom du fichier.
fi


[ ! -f "$fichier" ] && echo "Le fichier \"$fichier\" introuvable. \
Je refuse peureusement d'effacer un fichier inexistant."
# LISTE ET, pour donner le message d'erreur si le fichier est absent.
# Notez que le message echo continue sur la seconde ligne avec un échappement.

[ ! -f "$file" ] || (rm -f $file; echo "Fichier \"$file\" supprimé.")
# LISTE OU, pour supprimer le fichier si présent.

# Notez la logique inversée ci-dessus.
# La LISTE-ET s'exécute si vrai, la LISTE-OU si faux.

exit 0
```

Attention

Si la première commande dans une « liste OU » renvoie true, elle *sera* exécutée.

```
# ==> Les astuces suivantes proviennent du
#+==> script /etc/rc.d/init.d/single de Miquel van Smoorenburg
#+==> Illustre l'utilisation des listes "ET" et "OU".
```

```
# ==> Les commentaires "à flèche" ont été ajoutés par l'auteur de ce document.

[ -x /usr/bin/clear ] && /usr/bin/clear
  # ==> Si /usr/bin/clear existe, alors il est exécuté
  # ==> Vérifier l'existence d'une commande avant de l'utiliser
  #+==> évite des messages d'erreur et d'autres conséquences bizarres.

  # ==> . . .

# S'ils veulent lancer quelque chose en mode simple utilisateur, autant le
# lancer...
for i in /etc/rc1.d/S[0-9][0-9]* ; do
        # Vérifier si le script est ici.
        [ -x "$i" ] || continue
  # ==> Si le fichier correspondant n'est *pas* trouvé dans $PWD,
  #+==> alors "continue"z en sautant au début de la boucle.

        # Rejete les fichiers de sauvegarde et les fichiers générés par rpm.
        case "$1" in
                *.rpmsave|*.rpmorig|*.rpmnew|*~|*.orig)
                        continue;;
        esac
        [ "$i" = "/etc/rc1.d/S00single" ] && continue
  # ==> Initialise le nom du script, mais ne l'exécute pas encore.
        $i start
done

  # ==> . . .
```

Important

Le code de sortie44 d'une **`liste ET`** ou d'une **`liste OU`** correspond au code de sortie de la dernière commande exécutée.

Les combinaisons intelligentes de listes « ET » et « OU » sont possibles, mais la logique pourrait rapidement devenir difficile et nécessiter une grande attention aux règles de précédence des opérateurs432, voire même des phases de débogages intensives.

```
false && true || echo false    # false

# Même résultat avec
( false && true ) || echo false     # false
# Mais *pas*
false && ( true || echo false )     # (rien ne s'affiche)

# Notez le groupement de gauche à droite et une évaluation des instructions
# car les opérateurs logiques "&&" et "||" ont la même priorité.

# Il est mieux d'éviter de telles complexités, sauf si vous savez ce que vous
# faites.

# Merci, S.C.
```

Voir l'Exemple A.7, « *days-between* : Calculer le nombre de jours entre deux dates » et l'Exemple 7.4, « Test de liens cassés » pour des illustrations de l'utilisation de **`listes ET / OU`** pour tester des variables.

Chapitre 26. Tableaux

Les versions récentes de Bash supportent les tableaux à une dimension. Les éléments du tableau devraient être initialisés avec la notation **`variable[xx]`**. Autrement, un script peut introduire le tableau entier par une instruction explicite **`declare -a variable`**. Pour déréférencer (trouver le contenu d') un élément du tableau, utilisez la notation à *accolade*, c'est-à-dire **`${variable[xx]}`**.

Exemple 26.1. Utilisation d'un tableau simple

```
#!/bin/bash


aire[11]=23
aire[13]=37
aire[51]=UFOs

# Les membres d'un tableau peuvent ne pas être consécutifs ou contigus.

# Certains membres peuvent rester non initialisés.
# Les trous dans le tableau sont OK.
# En fait, les tableaux avec des données "écartées" sont utiles dans les tableurs.


echo -n "aire[11] = "
echo ${aire[11]}    #  {accolades} nécessaires.

echo -n "aire[13] = "
echo ${aire[13]}

echo "Le contenu de aire[51] est ${aire[51]}."

# Le contenu d'une variable non initialisée d'un tableau n'affiche rien (variable
nulle).
echo -n "aire[43] = "
echo ${aire[43]}
echo "(aire[43] non affecté)"

echo

# Somme de deux variables tableaux affectée à une troisième.
aire[5]=`expr ${aire[11]} + ${aire[13]}`
echo "aire[5] = aire[11] + aire[13]"
echo -n "aire[5] = "
echo ${aire[5]}

aire[6]=`expr ${aire[11]} + ${aire[51]}`
echo "aire[6] = aire[11] + aire[51]"
echo -n "aire[6] = "
echo ${aire[6]}
# Ceci échoue car ajouter un entier à une chaîne de caractères n'est pas permis.

echo; echo; echo

# --------------------------------------------------------------------
# Autre tableau, "aire2".
# Autre façon d'affecter les variables d'un tableau...
# nom_tableau=( XXX YYY ZZZ ... )

aire2=( zero un deux trois quatre )

echo -n "aire2[0] = "
echo ${aire2[0]}
# Aha, indexage commençant par 0 (le premier élément du tableau est [0], et non
# pas [1]).

echo -n "aire2[1] = "
```

```
echo ${aire2[1]}    # [1] est le deuxième élément du tableau.
# ---------------------------------------------------------------

echo; echo; echo

# ------------------------------------------------
# Encore un autre tableau, "aire3".
# Encore une autre façon d'affecter des variables de tableau...
# nom_tableau=([xx]=XXX [yy]=YYY ...)

aire3=([17]=dix-sept [24]=vingt-quatre)

echo -n "aire3[17] = "
echo ${aire3[17]}

echo -n "aire3[24] = "
echo ${aire3[24]}
# ------------------------------------------------

exit 0
```

Comme nous l'avons vu, une façon agréable d'initialiser un tableau complet est la notation `array=( element1 element2 ... elementN)`.

> Bash autorise des opérations de tableaux sur des variables, même si les variables ne sont pas explicitement déclarées en tant que tableau.
>
> ```
> chaine=abcABC123ABCabc
> echo ${chaine[@]}               # abcABC123ABCabc
> echo ${chaine[*]}               # abcABC123ABCabc
> echo ${chaine[0]}               # abcABC123ABCabc
> echo ${chaine[1]}               # Pas de sortie !
>                                 # Pourquoi ?
> echo ${#chaine[@]}              # 1
>                                 # Un élément dans le tableau.
>                                 # La chaîne elle-même.
>
> # Merci, Michael Zick, de nous l'avoir précisé.
> ```
>
> Une fois encore, ceci démontre que les variables Bash ne sont pas typées32.

Exemple 26.2. Formatage d'un poème

```
#!/bin/bash
# poem.sh : affiche joliment un des poèmes préférés de l'auteur du document.

# Lignes d'un poème (simple stanza).
Ligne[1]="I do not know which to prefer,"
Ligne[2]="The beauty of inflections"
Ligne[3]="Or the beauty of innuendoes,"
Ligne[4]="The blackbird whistling"
Ligne[5]="Or just after."

# Attribution.
Attrib[1]=" Wallace Stevens"
Attrib[2]="\"Thirteen Ways of Looking at a Blackbird\""
# Ce poème est dans le domaine public (copyright expiré).

echo

for index in 1 2 3 4 5    # Cinq lignes.
do
  printf "     %s\n" "${Ligne[index]}"
done
```

```
for index in 1 2          # Deux lignes.
do
  printf "          %s\n" "${Attrib[index]}"
done

echo

exit 0

# Exercice :
# ---------
# Modifiez ce script pour afficher joliment un poème à partir d'un fichier de
# données au format texte.
```

Les variables tableau ont une syntaxe propre, et même les commandes standards Bash et les opérateurs ont des options spécifiques adaptées à l'utilisation de tableaux.

Exemple 26.3. Opérations de chaînes sur des tableaux

```
#!/bin/bash
# array-strops.sh : Opérations sur des chaînes comprises dans des tableaux.
# Script de Michael Zick.
# Utilisé avec sa permission.

#  En général, toute opération sur des chaînes avec la notation ${nom ... }
#+ peut être appliquée aux éléments de type chaîne de caractères d'un tableau
#+ en utilisant la notation ${nom[@] ... } ou ${nom[*] ...}.


tableauZ=( un deux trois quatre cinq cinq )

echo

# Extraction de la dernière sous-chaîne
echo ${tableauZ[@]:0}   # un deux trois quatre cinq cinq
                        # Tous les éléments.

echo ${tableauZ[@]:1}   # deux trois quatre cinq cinq
                        # Tous les éléments après element[0].

echo ${tableauZ[@]:1:2} # deux trois
                        # Seulement les deux éléments après element[0].

echo "-----------------------"

#  Suppression d'une sous-chaîne
#  Supprime la plus petite correspondance au début de(s) chaîne(s),
#+ la sous-chaîne étant une expression rationnelle.

echo ${tableauZ[@]#q*e} # un deux trois cinq cinq
                        # Appliqué à tous les éléments du tableau.
                        # Correspond à "quatre" et le supprime.

# Correspondance la plus longue au début d'une chaîne
echo ${tableauZ[@]##t*s}  # un deux quatre cinq cinq
                        # Appliqué à tous les éléments du tableau.
                        # Correspond à "trois" et le supprime.

# Plus petite correspondance à partir de la fin de(s) chaîne(s)
echo ${tableauZ[@]%r*s} # un deux t quatre cinq cinq
                        # Appliqué à tous les éléments du tableau.
                        # Correspond à "rois" et le supprime.

# Plus longue correspondance à partir de la fin des chaînes.
echo ${tableauZ[@]%%t*s}  # un deux quatre cinq cinq
                        # Appliqué à tous les éléments du tableau.
```

```
                        # Correspond à "trois" et le supprime.

echo "----------------------"

# Remplacement de sous-chaînes

# Remplace la première occurence d'une sous-chaîne
echo ${tableauZ[@]/cin/XYZ} # un deux trois quatre XYZq XYZq
                            # Appliqué à tous les éléments de la sous-chaîne.

# Remplace toutes les occurences de la sous-chaîne
echo ${tableauZ[@]//in/YY}  # un deux trois quatre cYYq cYYq
                            # Appliqué à tous les éléments de la sous-chaîne.

# Supprime toutes les occurences de la sous-chaîne
# Ne pas spécifier un remplacement suppose une 'suppression'.
echo ${tableauZ[@]//ci/}    # un deux trois quatre nq nq
                            # Appliqué à tous les éléments de la sous-chaîne.

# Remplace le début des occurences de la sous-chaîne
echo ${tableauZ[@]/#ci/XY}  # un deux trois quatre XYnq XYnq
                            # Appliqué à tous les éléments de la sous-chaîne.

# Remplace la fin des occurences de la sous-chaînes
echo ${tableauZ[@]/%nq/ZZ}  # un deux trois quatre ciZZ ciZZ
                            # Appliqué à tous les éléments de la sous-chaîne.

echo ${tableauZ[@]/%u/XX}   # XX deXX trois qXXXXX cinq cinq
                            # Pourquoi ?

echo "----------------------"


# Avant de regarder awk (ou autre chose)
# Rappel :
#   $( ... ) est une substitution de commande.
#   Les fonctions sont lancées en tant que sous-processus.
#   Les fonctions écrivent leur propre sortie vers stdout.
#   Les affectations lisent le stdout de la fonction.
#   La notation nom[@] spécifie une opération "for-each".

nouvellechaine() {
    echo -n "!!!"
}

echo ${tableauZ[@]/%u/$(nouvellechaine)}
# !!!n de!!!x trois q!!!atre cinq cinq
# Q.E.D: L'action de remplacement est une 'affectation'.

#  Accéder au "For-Each"
echo ${tableauZ[@]//*/$(nouvellechaine arguments_optionnels)}
#  Maintenant, si Bash passait juste la chaîne correspondante comme $0 à la
#+ fonction appelée...

echo

exit 0
```

La substitution de commandes141 peut construire les éléments individuels d'un tableau.

Exemple 26.4. Charger le contenu d'un script dans un tableau

```
#!/bin/bash
# script-array.sh : Charge ce script dans un tableau.
# Inspiré d'un e-mail de Chris Martin (merci !).

contenu_script=( $(cat "$0") )   #  Enregistre le contenu de ce script ($0)
```

```
                                  #+ dans un tableau.

for element in $(seq 0 $((${#contenu_script[@]} - 1)))
  do                #  ${#contenu_script[@]}
                    #+ donne le nombre d'éléments dans le tableau.
                    #
                    #  Question:
                    #  Pourquoi seq 0  est-il nécessaire ?
                    #  Essayez de le changer en seq 1.
  echo -n "${contenu_script[$element]}"
                    # Affiche tous les champs de ce script sur une seule ligne.
  echo -n " -- "    # Utilise " -- " comme séparateur de champs.
done

echo

exit 0

# Exercice :
# ---------
#  Modifiez ce script de façon à ce qu'il s'affiche lui-même dans son format
#+ original, entier avec les espaces blancs, les retours ligne, etc.
```

Dans un contexte de tableau, quelques commandes intégrées159 Bash ont une signification légèrement modifiée. Par exemple, unset supprime des éléments du tableau, voire un tableau entier.

Exemple 26.5. Quelques propriétés spéciales des tableaux

```
#!/bin/bash

declare -a couleurs
#  Toutes les commandes suivantes dans ce script traiteront
#+ la variable "couleurs" comme un tableau.

echo "Entrez vos couleurs favorites (séparées par une espace)."

read -a couleurs  #  Entrez au moins trois couleurs pour démontrer les
                  #+ fonctionnalités ci-dessous.
#  Option spéciale pour la commande 'read'
#+ permettant d'affecter les éléments dans un tableau.

echo

nb_element=${#couleurs[@]}
# Syntaxe spéciale pour extraire le nombre d'éléments d'un tableau.
#     nb_element=${#couleurs[*]} fonctionne aussi.
#
#  La variable "@" permet de diviser les mots compris dans des guillemets
#+ (extrait les variables séparées par des espaces blancs).
#
#  Ceci correspond au comportement de "$@" et "$*"
#+ dans les paramètres de positionnement.

index=0

while [ "$index" -lt "$nb_element" ]
do    # Liste tous les éléments du tableau.
  echo ${couleurs[$index]}
  let "index = $index + 1"
  # ou index+=1 avec Bash 3.1 et suivants
done
# Chaque élément du tableau est listé sur une ligne séparée.
# Si ceci n'est pas souhaité, utilisez echo -n "${couleurs[$index]} "
#
# Pour le faire avec une boucle "for":
#   for i in "${couleurs[@]}"
#   do
```

```
#     echo "$i"
#   done
# (Thanks, S.C.)

echo

#  Encore une fois, liste tous les éléments d'un tableau, mais en utilisant une
#+ méthode plus élégante.
  echo ${couleurs[@]}        # echo ${couleurs[*]} fonctionne aussi.

echo

# La commande "unset" supprime les éléments d'un tableau ou un tableau entier.
unset couleurs[1]            # Supprime le deuxième élément d'un tableau.
                             # Même effet que   couleurs[1]=
echo  ${couleurs[@]}         # Encore un tableau liste, dont le deuxième
                             # élément est manquant.

unset couleurs               # Supprime le tableau entier.
                             #  unset couleurs[*] et
                             #+ unset couleurs[@] fonctionnent aussi.
echo; echo -n "couleurs parties."
echo ${couleurs[@]}          #  Affiche le tableau une nouvelle fois, maintenant
                             #+ vide.

exit 0
```

Comme vu dans l'exemple précédent, soit **${nom_tableau[@]}** soit **${nom_tableau[*]}** fait réfèrence à *tous* les éléments du tableau. De même, pour obtenir le nombre d'éléments dans un tableau, utilisez soit **${#nom_tableau[@]}** soit **${#nom_tableau[*]}**. **${#nom_tableau}** est la longueur (nombre de caractères) de **${nom_tableau[0]}**, le premier élément du tableau.

Exemple 26.6. Des tableaux vides et des éléments vides

```
#!/bin/bash
# empty-array.sh

#  Merci à Stephane Chazelas pour l'exemple original
#+ et à Michael Zick et Omair Eshkenazi pour son extension.


#  Un tableau vide n'est pas la même chose qu'un tableau composé d'éléments
#+ vides.

tableau0=( premier deuxieme troisieme )
tableau1=( '' )   # "tableau1" consiste en un élément vide.
tableau2=( )      # Pas d'éléments . . . "tableau2" est vide.
tableau3=(   )    # Que peut-on dire de ce tableau ?

echo
AfficheTableau()
{
echo
echo "Éléments de tableau0 :  ${tableau0[@]}"
echo "Éléments de tableau1 :  ${tableau1[@]}"
echo "Éléments de tableau2 :  ${tableau2[@]}"
echo "Éléments de tableau3 :  ${tableau3[@]}"
echo
echo "Longueur du premier élément du tableau0 = ${#tableau0}"
echo "Longueur du premier élément du tableau1 = ${#tableau1}"
echo "Longueur du premier élément du tableau2 = ${#tableau2}"
echo "Longueur du premier élément du tableau3 = ${#tableau3}"
echo
echo "Nombre d'éléments du tableau0 = ${#tableau0[*]}"  # 3
echo "Nombre d'éléments du tableau1 = ${#tableau1[*]}"  # 1  (Surprise !)
echo "Nombre d'éléments du tableau2 = ${#tableau2[*]}"  # 0
echo "Nombre d'éléments du tableau3 = ${#tableau3[*]}"  # 0
```

```
}

# ==================================================================

AfficheTableau

# Essayons d'étendre ces tableaux.

# Ajouter un élément à un tableau.
tableau0=( "${tableau0[@]}" "nouveau1" )
tableau1=( "${tableau1[@]}" "nouveau1" )
tableau2=( "${tableau2[@]}" "nouveau1" )
tableau3=( "${tableau3[@]}" "nouveau1" )

AfficheTableau

# ou
tableau0[${#tableau0[*]}]="nouveau2"
tableau1[${#tableau1[*]}]="nouveau2"
tableau2[${#tableau2[*]}]="nouveau2"
tableau3[${#tableau3[*]}]="nouveau2"

AfficheTableau

# Lors d'un ajout comme ci-dessus ; les tableaux sont des piles ('stacks')
# La commande ci-dessus correspond à un 'push'
# La hauteur de la pile est :
hauteur=${#tableau2[@]}
echo
echo "Hauteur de pile pour tableau2 = $hauteur"

# L'opération 'pop' est :
unset tableau2[${#tableau2[@]}-1] #  L'index des tableaux commence à zéro,
hauteur=${#tableau2[@]}           #+ ce qui signifie que le premier élément se
                                  #+ trouve à l'index 0.
echo
echo "POP"
echo "Nouvelle hauteur de pile pour tableau2 = $hauteur"

AfficheTableau

# Affiche seulement les 2è et 3è éléments de tableau0.
de=1         # Numérotation débutant à zéro.
a=2         #
tableau3=( ${tableau0[@]:1:2} )
echo
echo "Éléments de tableau3 :  ${tableau3[@]}"

# Fonctionne comme une chaîne (tableau de caractères).
# Essayez les autres formes de "chaînes".

# Remplacement :
tableau4=( ${tableau0[@]/deuxieme/2è} )
echo
echo "Éléments de tableau4 :  ${tableau4[@]}"

# Remplacez toutes les chaînes correspondantes.
tableau5=( ${tableau0[@]//nouveau?/ancien} )
echo
echo "Éléments de tableau5 :  ${tableau5[@]}"

# Juste quand vous commencez à vous habituer...
tableau6=( ${tableau0[@]#*nouveau} )
echo # Ceci pourrait vous surprendre.
echo "Éléments du tableau6 :  ${tableau6[@]}"

tableau7=( ${tableau0[@]#nouveau1} )
echo # Après tableau6, ceci ne devrait plus être une surprise.
echo "Éléments du tableau7 :  ${tableau7[@]}"
```

```
# Qui ressemble beaucoup à...
tableau8=( ${tableau0[@]/nouveau1/} )
echo
echo "Éléments du tableau8 :  ${tableau8[@]}"

#  Donc, que pouvez-vous conclure de ceci ?

#  Les opérations sur des chaînes sont réalisées sur chaque éléments
#+ de var[@] à la suite.
#  Donc : Bash supporte les opérations vectorielles sur les chaînes.
#  Si le résultat est une chaîne de longueur vide,
#+ l'élément disparaît dans l'affectation résultante.

#  Question, ces chaînes sont-elles entre simples ou doubles guillemets ?

zap='nouveau*'
tableau9=( ${tableau0[@]/$zap/} )
echo
echo "Éléments du tableau9 :  ${tableau9[@]}"

# Juste au moment où vous pensiez être toujours en pays connu...
tableau10=( ${tableau0[@]#$zap} )
echo
echo "Éléments du tableau10 :  ${tableau10[@]}"

# Comparez le tableau7 avec le tableau10.
# Comparez le tableau8 avec le tableau9.

# Réponse : Cela doit être des simples guillemets.

exit 0
```

La relation entre **${nom_tableau[@]}** et **${nom_tableau[*]}** est analogue à celle entre $@ et $*. Cette notation de tableau très puissante a un certain nombre d'intérêts.

```
# Copier un tableau.
tableau2=( "${tableau1[@]}" )
# ou
tableau2="${tableau1[@]}"

#
#  However, this fails with "sparse" arrays,
#+ arrays with holes (missing elements) in them,
#+ as Jochen DeSmet points out.
# ----------------------------------------
  array1[0]=0
# array1[1] not assigned
  array1[2]=2
  array2=( "${array1[@]}" )       # Copy it?

echo ${array2[0]}      # 0
echo ${array2[2]}      # (null), should be 2
# ----------------------------------------



# Ajout d'un élément dans un tableau.
array=( "${array[@]}" "nouvel élément" )
# or
array[${#array[*]}]="nouvel élément"

# Merci, S.C.
```

Astuce

L'opération d'initialisation **tableau=(element1 element2 ... elementN)**, avec l'aide de la substitution de com-

mandes141, rend possible de charger le contenu d'un fichier texte dans un tableau.

```
#!/bin/bash

nomfichier=fichier_exemple

#            cat fichier_exemple
#
#            1 a b c
#            2 d e fg


declare -a tableau1

tableau1=( `cat "$nomfichier" `)  # Charge le contenu
                                  # de $nomfichier dans tableau1.
#           affiche le fichier sur stdout.
# tableau1=( `cat "$nomfichier" | tr '\n' ' '`)
#                                  modifie les retours chariots en espace.
# Non nécessaire car Bash réalise le découpage des mots, modifiant les
# changements de ligne en espaces.

echo ${tableau1[@]}            # Affiche le tableau.
#                              1 a b c 2 d e fg
#
#  Chaque "mot" séparé par une espace dans le fichier a été affecté à un
#+ élément du tableau.

nb_elements=${#tableau1[*]}
echo $nb_elements          # 8
```

Une écriture intelligente de scripts rend possible l'ajout d'opérations sur les tableaux.

Exemple 26.7. Initialiser des tableaux

```
#! /bin/bash
# array-assign.bash

#  Les opérations sur les tableaux sont spécifiques à Bash,
#+ d'où le ".bash" dans le nom du script.

# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
# Version: $ID$
#
# Clarification et commentaires supplémentaires par William Park.

#  Basé sur un exemple de Stephane Chazelas
#+ qui est apparu dans le livre : Advanced Bash Scripting Guide.

# Format en sortie de la commande 'times' :
# CPU Utilisateur &lt;espace&gt; CPU système
# CPU Utilisateur du fils mort &lt;space&gt; CPU système du fils mort

#  Bash a deux versions pour l'affectation de tous les éléments d'un tableau
#+ vers une nouvelle variable tableau.
#  Les deux jetent les éléments à référence nulle
#+ avec Bash version 2.04, 2.05a et 2.05b.
#  Une affectation de tableau supplémentaire qui maintient les relations de
#+ [sousscript]=valeur pour les tableaux pourrait être ajoutée aux nouvelles
#+ versions.

#  Construit un grand tableau en utilisant une commande interne,
#+ mais tout ce qui peut construire un tableau de quelques milliers d'éléments
#+ fera l'affaire.

declare -a grandElement=( /dev/* ) # Tous les fichiers de /dev...
```

```
echo
echo 'Conditions : Sans guillemets, IFS par défaut, Tout élément'
echo "Le nombre d'éléments dans le tableau est ${#grandElement[@]}"

# set -vx



echo
echo '- - test: =( ${array[@]} ) - -'
times
declare -a grandElement2=( ${grandElement[@]} )
# Notez les parenthèses  ^                  ^
times


echo
echo '- - test: =${array[@]} - -'
times
declare -a grandElement3=${grandElement[@]}
# Pas de parenthèses cette fois-ci.
times

#  Comparer les nombres montre que la deuxième forme, indiquée par
#+ Stephane Chazelas, est de trois à quatre fois plus rapide.
#
#  Comme William Park l'explique :
#+ Le tableau grandElement2 est affecté élément par élément à cause des parenthèses
#+ alors que grandElement3 est affecté en une seule chaîne.
#  Donc, en fait, vous avez :
#                  grandElement2=( [0]="..." [1]="..." [2]="..." ... )
#                  grandElement3=( [0]="... ... ..." )
#
#  Vérifiez ceci avec : echo ${grandElement2[0]}
#                       echo ${grandElement3[0]}

#  Je continuerais à utiliser la première forme dans mes descriptions d'exemple
#+ parce que je pense qu'il s'agit d'une meilleure illustration de ce qu'il se
#+ passe.

#  Les portions réutilisables de mes exemples contiendront réellement la
#+ deuxième forme quand elle est appropriée en ce qui concerne sa rapidité.

# MSZ : Désolé à propos de ce survol précédent.

#  Note :
#  -----
#  Les instructions "declare -a" des lignes 31 et 43 ne sont pas strictement
#  nécessaires car c'est implicite dans l'appel de Array=( ... )
#  Néanmoins, éliminer ces déclarations ralentit l'exécution des sections
#  suivantes du script.
#  Essayez et voyez ce qui se passe.

exit 0
```

Note

Ajouter une instruction superflue **declare -a** pour la déclaration d'un tableau pourrait accélérer l'exécution des opérations suivantes sur le tableau.

Exemple 26.8. Copier et concaténer des tableaux

```
#! /bin/bash
# CopieTableau.sh
#
```

```
# Ce script a été écrit par Michael Zick.
# Utilisé ici avec sa permission.

#  Guide pratique "Passage par nom & Retour par nom"
#+ ou "Construire votre propre instruction d'affectation".


CopieTableau_Mac() {

# Constructeur d'instruction d'affectation

    echo -n 'eval '
    echo -n "$2"                   # Nom de la destination
    echo -n '=( ${'
    echo -n "$1"                   # Nom de la source
    echo -n '[@]} )'

# Cela peut être une seule commande.
# Une simple question de style.
}

declare -f CopieTableau                     # Fonction "Pointeur".
CopieTableau=CopieTableau_Mac               # Constructeur d'instruction.

Hype()
{

# Hype le tableau nommé $1.
# (L'ajoute au tableau contenant "Really Rocks".)
# Retour dans le tableau nommé $2.

    local -a TMP
    local -a hype=( Really Rocks )

    $($CopieTableau $1 TMP)
    TMP=( ${TMP[@]} ${hype[@]} )
    $($CopieTableau TMP $2)
}

declare -a avant=( Advanced Bash Scripting )
declare -a apres

echo "Tableau avant = ${avant[@]}"

Hype avant apres !!!!!!!!!!!!!!!!!!!!!!!

echo "Tableau après = ${apres[@]}"

# Trop de 'hype' ?

echo "Qu'est-ce que ${apres[@]:3:2}?"

declare -a modeste=( ${apres[@]:2:1} ${apres[@]:3:2} )
#                  -- extraction de la sous-chaine --

echo "Tableau modeste = ${modeste[@]}"

# Qu'arrive-t'il à 'avant' ?

echo "Tableau avant = ${avant[@]}"

exit 0
```

Exemple 26.9. Plus sur la concaténation de tableaux

```
#! /bin/bash
# array-append.bash
```

```
# Copyright (c) Michael S. Zick, 2003, All rights reserved.
# License: Unrestricted reuse in any form, for any purpose.
# Version: $ID$
#
# Légèrement modifié au niveau formatage par M.C.


# Les opérations sur les tableaux sont spécifiques à Bash.
# Le /bin/sh de l'UNIX standard n'a pas d'équivalent.


#  Envoyer la sortie de ce script à 'more'
#+ de façon à ce que le terminal affiche page par page.


# Sous-script imbriqué.
declare -a tableau1=( zero1 un1 deux1 )
# Sous-script léger ([1] n'est pas défini).
declare -a tableau2=( [0]=zero2 [2]=deux2 [3]=trois3 )

echo
echo '- Confirmez que ce tableau est vraiment un sous-script. -'
echo "Nombre d'éléments : 4"        # Codé en dur pour illustration.
for (( i = 0 ; i < 4 ; i++ ))
do
    echo "Élément [$i] : ${tableau2[$i]}"
done
# Voir aussi l'exemple de code plus général dans basics-reviewed.bash.


declare -a dest

# Combinez (ajoutez) deux tableaux dans un troisième.
echo
echo 'Conditions : Sans guillemets, IFS par défaut, opérateur tous-éléments-de'
echo '- Éléments indéfinis non présents, sous-scripts non maintenus. -'
# # Les éléments indéfinis n'existent pas ; ils ne sont pas réellement supprimés.

dest=( ${tableau1[@]} ${tableau2[@]} )
# dest=${tableau1[@]}${tableau2[@]} # Résultats étranges, probablement un bogue.

# Maintenant, affiche le résultat.
echo
echo "- - Test de l'ajout du tableau - -"
cpt=${#dest[@]}

echo "Nombre d'éléments : $cpt"
for (( i = 0 ; i < cpt ; i++ ))
do
    echo "Élément [$i] : ${dest[$i]}"
done

# Affecte un tableau sur un élément d'un tableau simple (deux fois).
dest[0]=${tableau1[@]}
dest[1]=${tableau2[@]}

# Affiche le résultat.
echo
echo '- - Test du tableau modifié - -'
cpt=${#dest[@]}

echo "Nombre d'éléments : $cpt"
for (( i = 0 ; i < cpt ; i++ ))
do
    echo "Élément [$i] : ${dest[$i]}"
done

# Examine le deuxième élément modifié.
```

```
echo
echo '- - Réaffecte et affiche le deuxième élément - -'

declare -a sousTableau=${dest[1]}
cpt=${#sousTableau[@]}

echo "Nombre d'éléments: $cpt"
for (( i = 0 ; i < cpt ; i++ ))
do
    echo "Element [$i] : ${sousTableau[$i]}"
done

#  L'affectation d'un tableau entier sur un seul élément d'un autre tableau
#+ utilisant l'opérateur d'affectation de tableau '=${ ... }' a converti le
#+ tableau en cours d'affectation en une chaîne de caractères, les éléments
#+ étant séparés par une espace (le premier caractère de IFS).

# Si les éléments originaux ne contenaient pas d'espace blanc ...
# Si la tableau original n'est pas un sous-script ...
# Alors nous pouvons récupérer la structure du tableau original.

# Restaurer à partir du second élément modifié.
echo
echo "- - Affichage de l'élément restauré - -"

declare -a sousTableau=( ${dest[1]} )
cpt=${#sousTableau[@]}

echo "Nombre d'éléments : $cpt"
for (( i = 0 ; i < cpt ; i++ ))
do
    echo "Élément [$i] : ${sousTableau[$i]}"
done
echo '- - Ne dépends pas de ce comportement - -'
echo '- - Ce comportement est sujet à modification - -'
echo '- - dans les versions de Bash ultérieures à la version 2.05b - -'

# MSZ : Désolé pour la confusion précédente.

exit 0
```

--

Les tableaux permettent de déployer de bons vieux algorithmes familiers en scripts shell. Que ceci soit réellement une bonne idée est laissé à l'appréciation du lecteur.

Exemple 26.10. Le tri bulle : *Bubble Sort*

```
#!/bin/bash
# bubble.sh : Tri bulle, en quelque sorte.

# Rappelle l'algorithme de tri bulle. Enfin, une version particulière...

#  À chaque itération successive à travers le tableau à trier, compare deux
#+ éléments adjacents et les échange s'ils ne sont pas ordonnés.
#  À la fin du premier tour, l'élémennt le "plus lourd" est arrivé tout en bas.
#  À la fin du deuxième tour, le "plus lourd" qui suit est lui-aussi à la fin
#+ mais avant le "plus lourd".
#  Et ainsi de suite.
#  Ceci signifie que chaque tour a besoin de se balader sur une partie de plus
#+ en plus petite du tableau.
#  Vous aurez donc noté une accélération à l'affichage lors des derniers tours.


echange()
{
  # Échange deux membres d'un tableau
```

```
  local temp=${Pays[$1]}      #  Stockage temporaire
                              #+ pour les éléments à échanger.
  Pays[$1]=${Pays[$2]}
  Pays[$2]=$temp

  return
}

declare -a Pays  #  Déclaration d'un tableau,
                 #+ optionnel ici car il est initialisé tout de suite après.

#  Est-il permis de diviser une variable tableau sur plusieurs lignes en
#+ utilisant un caractère d'échappement ?
#  Oui.

Pays=(Hollande Ukraine Zaire Turquie Russie Yémen Syrie \
Brésil Argentine Nicaragua Japon Mexique Vénézuela Grèce Angleterre \
Israël Pérou Canada Oman Danemark France Kenya \
Xanadu Qatar Liechtenstein Hongrie)

# "Xanadu" est la place mythique où, selon Coleridge,
#+ Kubla Khan a décrété un summum de plaisir.


clear                # Efface l'écran pour commencer.

echo "0: ${Pays[*]}"  # Liste le tableau entier lors du premier tour.

nombre_d_elements=${#Pays[@]}
let "comparaisons = $nombre_d_elements - 1"

index=1 # Nombre de tours.

while [ "$comparaisons" -gt 0 ]          # Début de la boucle externe.
do

  index=0  #  Réinitialise l'index pour commencer au début du tableau à chaque
           #+ tour.

  while [ "$index" -lt "$comparaisons" ] # Début de la boucle interne.
  do
    if [ ${Pays[$index]} \> ${Pays[`expr $index + 1`]} ]
    #  Si non ordonné...
    #  Rappelez-vous que \> est un opérateur de comparaison ASCII à l'intérieur
    #+ de simples crochets.

    #  if [[ ${Pays[$index]} > ${Pays[`expr $index + 1`]} ]]
    #+ fonctionne aussi.
    then
      echange $index `expr $index + 1`  # Échange.
    fi
    let "index += 1" # Ou index+=1 sur Bash 3.1 et suivants
  done # Fin de la boucle interne.

# ----------------------------------------------------------------------
# Paulo Marcel Coelho Aragao suggère les boucles for comme alternative simple.
#
# for (( dernier = $nombre_d_elements - 1 ; dernier > 0 ; dernier-- ))
##                  Corrigé par C.Y. Hunt             ^  (Merci)
# do
#     for (( i = 0 ; i < dernier ; i++ ))
#     do
#         [[ "${Pays[$i]}" > "${Pays[$((i+1))]}" ]] \
#             && echange $i $((i+1))
#     done
# done
# ----------------------------------------------------------------------

let "comparaisons -= 1" #  Comme l'élément le "plus lourd" est tombé en bas,
```

```
                        #+ nous avons besoin de faire une comparaison de moins
                        #+ à chaque tour.

echo
echo "$index: ${Pays[@]}"  # Affiche le tableau résultat à la fin de chaque tour
echo
let "index += 1"               # Incrémente le compteur de tour.

done                           # Fin de la boucle externe.
                               # Fini.

exit 0
```

--

Est-il possible d'imbriquer des tableaux dans des tableaux ?

```
#!/bin/bash
# Tableaux imbriqués.

#  Michael Zick a fourni cet exemple,
#+ avec quelques corrections et clarifications de William Park.

UnTableau=( $(ls --inode --ignore-backups --almost-all \
        --directory --full-time --color=none --time=status \
        --sort=time -l ${PWD} ) )  # Commandes et options.

#  Les espaces ont une signification... et ne mettez pas entre guillemets quoi
#+ que ce soit ci-dessus.

SousTableau=( ${UnTableau[@]:11:1}  ${UnTableau[@]:6:5} )
#  Ce tableau a six éléments :
#+     SousTableau=( [0]=${UnTableau[11]} [1]=${UnTableau[6]} [2]=${UnTableau[7]}
#      [3]=${UnTableau[8]} [4]=${UnTableau[9]} [5]=${UnTableau[10]} )
#
#  Les tableaux en Bash sont des listes liées (circulaires) de type chaîne de
#+ caractères (char *).
#  Donc, ce n'est pas réellement un tableau imbriqué mais il fonctionne de la
#+ même manière.


echo "Répertoire actuel et date de dernière modification :"
echo "${SousTableau[@]}"

exit 0
```

--

Les tableaux imbriqués combinés avec des références indirectes créent quelques possibilités fascinantes.

Exemple 26.11. Tableaux imbriqués et références indirectes

```
#!/bin/bash
# embedded-arrays.sh
# Tableaux intégrés et références indirectes.

# Script de Dennis Leeuw.
# Utilisé avec sa permission.
# Modifié par l'auteur du document.


TABLEAU1=(
        VAR1_1=valeur11
        VAR1_2=valeur12
        VAR1_3=valeur13
)
```

```
TABLEAU2=(
        VARIABLE="test"
        CHAINE="VAR1=valeur1 VAR2=valeur2 VAR3=valeur3"
        TABLEAU21=${TABLEAU1[*]}
)       # TABLEAU1 intégré dans ce deuxième tableau.

function affiche () {
        OLD_IFS="$IFS"
        IFS=$'\n'       #  Pour afficher chaque élément du tableau
                        #+ sur une ligne séparée.
        TEST1="TABLEAU2[*]"
        local ${!TEST1} # Voir ce que se passe si vous supprimez cette ligne.
        #  Référence indirecte.
        #  Ceci rend accessible les composants de $TEST1 à cette fonction.


        #  Voyons où nous en sommes arrivés.
        echo
        echo "\$TEST1 = $TEST1"       #  Simplement le nom de la variable.
        echo; echo
        echo "{\$TEST1} = ${!TEST1}"  #  Contenu de la variable.
                                      #  C'est ce que fait une référence
                                      #+ indirecte.
        echo
        echo "-----------------------------------------"; echo
        echo


        # Affiche la variable
        echo "Variable VARIABLE : $VARIABLE"

        # Affiche un élément de type chaîne
        IFS="$OLD_IFS"
        TEST2="CHAINE[*]"
        local ${!TEST2}      # Référence indirecte (comme ci-dessus).
        echo "Élément chaîne VAR2 : $VAR2 à partir de CHAINE"

        # Affiche un élément du tableau
        TEST2="TABLEAU21[*]"
        local ${!TEST2}      # Référence indirecte (comme ci-dessus).
        echo "Élément du tableau VAR1_1 : $VAR1_1 à partir de TABLEAU21"
}

affiche
echo

exit 0

#   Comme l'indique l'auteur du script,
#+ "vous pouvez facilement l'étendre pour créer des hashs nommés en bash."
#   Exercice (difficile) pour le lecteur : l'implémenter.
```

--

Les tableaux permettent l'implémentation d'une version script shell du *Crible d'Ératosthene*. Bien sûr, une application intensive en ressources de cette nature devrait être réellement écrite avec un langage compilé tel que le C. Il fonctionne très lentement en tant que script.

Exemple 26.12. Crible d'Ératosthene

```
#!/bin/bash
# sieve.sh (ex68.sh)

# Crible d'Ératosthene
# Ancien algorithme pour trouver les nombres premiers.

# Ceci s'exécute bien moins rapidement que le programme équivalent écrit en C.
```

```
LIMITE_BASSE=1       # Commençant avec 1.
LIMITE_HAUTE=1000    # Jusqu'à 1000.
# (Vous pouvez augmenter cette valeur...  si vous avez du temps devant vous.)

PREMIER=1
NON_PREMIER=0

let DIVISE=LIMITE_HAUTE/2
# Optimisation :
# Nécessaire pour tester les nombres à mi-chemin de la limite supérieure (pourquoi ?).


declare -a Premiers
# Premiers[] est un tableau.


initialise ()
{
# Initialise le tableau.

i=$LIMITE_BASSE
until [ "$i" -gt "$LIMITE_HAUTE" ]
do
  Premiers[i]=$PREMIER
  let "i += 1"
done
# Assume que tous les membres du tableau sont coupables (premiers) avant d'être
# reconnus innocent.
}

affiche_premiers ()
{
# Affiche les membres du tableau Premiers[] indiqués comme premiers.

i=$LIMITE_BASSE

until [ "$i" -gt "$LIMITE_HAUTE" ]
do

  if [ "${Premiers[i]}" -eq "$PREMIER" ]
  then
    printf "%8d" $i
    # 8 espaces par nombre rend l'affichage joli, avec colonne.
  fi

  let "i += 1"

done

}

examine () # Examine minutieusement les non premiers.
{

let i=$LIMITE_BASSE+1
# Nous savons que 1 est premier, donc commençons avec 2.

until [ "$i" -gt "$LIMITE_HAUTE" ]
do

if [ "${Premiers[i]}" -eq "$PREMIER" ]
#  Ne nous embêtons pas à examiner les nombres déjà examinés (indiqués comme
#+ non premiers).
then

  t=$i

  while [ "$t" -le "$LIMITE_HAUTE" ]
```

```
  do
    let "t += $i "
    Premiers[t]=$NON_PREMIER
    # Indiqué comme non premier tous les multiples.
  done

fi

  let "i += 1"
done


}


# ============================================================
# main ()
# Appeler les fonctions séquentiellement.
initialise
examine
affiche_premiers
# C'est ce qu'ils appelent de la programmation structurée.
# ============================================================

echo

exit 0



# ---------------------------------------------------------------------- #
# Le code ci-dessous ne sera pas exécuté à cause du exit ci-dessus.

# Cette version améliorée de Sieve, par Stephane Chazelas,
# s'exécute un peu plus rapidement.

# Doit être appelé avec un argument en ligne de commande (limite des premiers).

LIMITE_HAUTE=$1                   # À partir de la ligne de commande.
let DIVISE=LIMITE_HAUTE/2         # Mi-chemin du nombre max.

Premiers=( '' $(seq $LIMITE_HAUTE) )

i=1
until (( ( i += 1 ) > DIVISE ))  # A besoin de vérifier à mi-chemin.
do
  if [[ -n $Premiers[i] ]]
  then
    t=$i
    until (( ( t += i ) > LIMITE_HAUTE ))
    do
      Premiers[t]=
    done
  fi
done
echo ${Premiers[*]}

exit $?
```

Exemple 26.13. Crible d'Ératosthene, optimisé

```
#!/bin/bash
# Optimized Sieve of Eratosthenes
# Script by Jared Martin, with very minor changes by ABS Guide author.
# Used in ABS Guide with permission (thanks!).

# Based on script in Advanced Bash Scripting Guide.
```

```
# http://tldp.org/LDP/abs/html/arrays.html#PRIMES0 (ex68.sh).

# http://www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf (reference)
# Check results against http://primes.utm.edu/lists/small/1000.txt

# Necessary but not sufficient would be, e.g.,
#     (($(sieve 7919 | wc -w) == 1000)) && echo "7919 is the 1000th prime"

UPPER_LIMIT=${1:?"Need an upper limit of primes to search."}

Primes=( '' $(seq ${UPPER_LIMIT}) )

typeset -i i t
Primes[i=1]='' # 1 is not a prime.
until (( ( i += 1 ) > (${UPPER_LIMIT}/i) ))  # Need check only ith-way.
  do                                         # Why?
    if ((${Primes[t=i*(i-1), i]}))
    # Obscure, but instructive, use of numeric eval in subscript.
    then
      until (( ( t += i ) > ${UPPER_LIMIT} ))
        do Primes[t]=; done
    fi
  done

# echo ${Primes[*]}
echo   # Change to original script for pretty-printing (80-col. display).
printf "%8d" ${Primes[*]}
echo; echo

exit $?
```

Comparez ces générateurs de nombres premiers basés sur les tableaux avec un autre ne les utilisant pas, l'Exemple A.16, « *primes*: Générer des nombres premiers en utilisant l'opérateur modulo ».

--

Les tableaux tendent eux-même à émuler des structures de données pour lesquelles Bash n'a pas de support natif.

Exemple 26.14. Émuler une pile

```
#!/bin/bash
# stack.sh : simulation d'une pile place-down

#  Similaire à la pile du CPU, une pile "place-down" enregistre les éléments
#+ séquentiellement mais les récupère en ordre inverse, le dernier entré étant
#+ le premier sorti.

BP=100            # Pointeur de base du tableau de la pile.
                  # Commence à l'élément 100.

SP=$BP            # Pointeur de la pile.
                  # Initialisé à la base (le bas) de la pile.

Donnees=          # Contenu de l'emplacement de la pile.
                  # Doit être une variable globale à cause de la limitation
                  #+ sur l'échelle de retour de la fonction.

declare -a pile


place()           # Place un élément dans la pile.
{
if [ -z "$1" ]    # Rien à y mettre ?
then
  return
fi
```

```
let "SP -= 1"     # Déplace le pointeur de pile.
pile[$SP]=$1

return
}

recupere()               # Récupère un élément de la pile.
{
Donnees=                 # Vide l'élément.

if [ "$SP" -eq "$BP" ]   # Pile vide ?
then
  return
fi                       #  Ceci empêche aussi SP de dépasser 100,
                         #+ donc de dépasser la capacité du tampon.

Donnees=${pile[$SP]}
let "SP += 1"            # Déplace le pointeur de pile.
return
}

rapport_d_etat()          # Recherche ce qui se passe
{
echo "-----------------------------------"
echo "RAPPORT"
echo "Pointeur de la pile = $SP"
echo "\""$Donnees"\" juste récupéré de la pile."
echo "-----------------------------------"
echo
}


# ========================================================
# Maintenant, amusons-nous...

echo

# Voyons si nous pouvons récupérer quelque chose d'une pile vide.
recupere
rapport_d_etat

echo

place garbage
recupere
rapport_d_etat     # Garbage in, garbage out.

valeur1=23; place $valeur1
valeur2=skidoo; place $valeur2
valeur3=FINAL; place $valeur3

recupere              # FINAL
rapport_d_etat
recupere              # skidoo
rapport_d_etat
recupere              # 23
rapport_d_etat        # dernier entré, premier sorti !

#  Remarquez comment le pointeur de pile décrémente à chaque insertion et
#+ incrémente à chaque récupération.

echo

exit 0

# ========================================================


# Exercices :
```

```
# ----------

# 1)  Modifier la fonction "place()" pour permettre le placement de plusieurs
#   + éléments sur la pile en un seul appel.

# 2)  Modifier la fonction "recupere()" pour récupérer plusieurs éléments de la
#   + pile en un seul appel de la fonction.

# 3)  Ajouter une vérification des erreurs aux fonctions critiques.
#     C'est-à-dire, retournez un code d'erreur
#   + dépendant de la réussite ou de l'échec de l'opération,
#   + et réagissez en effectuant les actions appropriées.

# 4)  En utilisant ce script comme base, écrire une calculatrice 4 fonctions
#   + basée sur une pile.
```

--

Des manipulations amusantes de tableaux pourraient nécessiter des variables intermédiaires. Pour des projets le nécessitant, considérez encore une fois l'utilisation d'un langage de programmation plus puissant comme Perl ou C.

Exemple 26.15. Application complexe des tableaux *Exploration d'une étrange série mathématique*

```
#!/bin/bash

# Les célèbres "Q-series" de Douglas Hofstadter :

# Q(1) = Q(2) = 1
# Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), pour n&gt;2

#  C'est une série chaotique d'entiers avec un comportement étrange et non
#+ prévisible.
# Les 20 premiers termes de la série étaient :
# 1 1 2 3 3 4 5 5 6 6 6 8 8 8 10 9 10 11 11 12

#  Voir le livre d'Hofstadter, _Goedel, Escher, Bach: An Eternal Golden Braid_,
#+ p. 137, ff.


LIMITE=100        # Nombre de termes à calculer.
LONGUEURLIGNE=20  # Nombre de termes à afficher par ligne.

Q[1]=1            # Les deux premiers termes d'une série sont 1.
Q[2]=1

echo
echo "Q-series [$LIMITE termes] :"
echo -n "${Q[1]} "             # Affiche les deux premiers termes.
echo -n "${Q[2]} "

for ((n=3; n <= $LIMITE; n++))  # Expressions de boucle style C.
do   # Q[n] = Q[n - Q[n-1]] + Q[n - Q[n-2]]  for n&gt;2
#  Nécessaire de casser l'expression en des termes intermédiaires
#+ car Bash ne gère pas très bien l'arithmétique des tableaux complexes.

  let "n1 = $n - 1"        # n-1
  let "n2 = $n - 2"        # n-2

  t0=`expr $n - ${Q[n1]}`  # n - Q[n-1]
  t1=`expr $n - ${Q[n2]}`  # n - Q[n-2]

  T0=${Q[t0]}              # Q[n - Q[n-1]]
  T1=${Q[t1]}              # Q[n - Q[n-2]]

Q[n]=`expr $T0 + $T1`      # Q[n - Q[n-1]] + Q[n - Q[n-2]]
echo -n "${Q[n]} "
```

```
if [ `expr $n % $LONGUEURLIGNE` -eq 0 ]    # Formate la sortie.
then   #      ^ Opérateur modulo
  echo # Retour chariot pour des ensembles plus jolis.
fi

done

echo

exit 0

# C'est une implémentation itérative de la Q-series.
# L'implémentation récursive plus intuitive est laissée comme exercice.
#  Attention : calculer cette série récursivement prend BEAUCOUP de temps
#+ via un script. C/C++ aurait mieux convenu.
```

--

Bash supporte uniquement les tableaux à une dimension. Néanmoins, une petite astuce permet de simuler des tableaux à plusieurs dimensions.

Exemple 26.16. Simuler un tableau à deux dimensions, puis son test

```
#!/bin/bash
# twodim.sh : Simuler un tableau à deux dimensions.

# Un tableau à une dimension consiste en une seule ligne.
# Un tableau à deux dimensions stocke les lignes séquentiellement.

Lignes=5
Colonnes=5
# Tableau de 5 sur 5.

declare -a alpha     # char alpha [Lignes] [Colonnes];
                     # Déclaration inutile. Pourquoi ?

charge_alpha ()
{
local rc=0
local index


for i in A B C D E F G H I J K L M N O P Q R S T U V W X Y
do  # Utilisez des symbôles différents si vous le souhaitez.
  local ligne=`expr $rc / $Colonnes`
  local colonne=`expr $rc % $Lignes`
  let "index = $ligne * $Lignes + $colonne"
  alpha[$index]=$i
  # alpha[$ligne][$colonne]
  let "rc += 1"
done

#  Un peu plus simple
#+   declare -a alpha=( A B C D E F G H I J K L M N O P Q R S T U V W X Y )
#+ mais il manque néanmoins le "bon goût" d'un tableau à deux dimensions.
}

affiche_alpha ()
{
local ligne=0
local index

echo

while [ "$ligne" -lt "$Lignes" ]  # Affiche dans l'ordre des lignes -
do                                # les colonnes varient
                                  # tant que ligne (la boucle externe) reste
```

```
                                    # identique
  local colonne=0

  echo -n "       "

  while [ "$colonne" -lt "$Colonnes" ]
  do
    let "index = $ligne * $Lignes + $colonne"
    echo -n "${alpha[index]} "  # alpha[$ligne][$colonne]
    let "colonne += 1"
  done

  let "ligne += 1"
  echo

done

# Un équivalent plus simple serait
#   echo ${alpha[*]} | xargs -n $Colonnes

echo
}

filtrer ()     # Filtrer les index négatifs du tableau.
{

echo -n "  "  # Apporte le tilt.
              # Expliquez comment.

if [[ "$1" -ge 0 &&  "$1" -lt "$Lignes" && "$2" -ge 0 && "$2" -lt "$Colonnes" ]]
then
    let "index = $1 * $Lignes + $2"
    # Maintenant, l'affiche après rotation.
    echo -n " ${alpha[index]}"
    #           alpha[$ligne][$colonne]
fi

}




rotate ()  #  Bascule le tableau de 45 degrés
{          #+ (le "balance" sur le côté gauche en bas).
local ligne
local colonne

for (( ligne = Lignes; ligne > -Lignes; ligne-- ))
do  # Traverse le tableau en sens inverse. Pourquoi ?

  for (( colonne = 0; colonne < Colonnes; colonne++ ))
  do

    if [ "$ligne" -ge 0 ]
    then
      let "t1 = $colonne - $ligne"
      let "t2 = $colonne"
    else
      let "t1 = $colonne"
      let "t2 = $colonne + $ligne"
    fi

    filtrer $t1 $t2   # Filtre les index négatifs du tableau.
                      # Que se passe-t'il si vous ne le faites pas ?
  done

  echo; echo

done
```

```
#  Rotation du tableau inspirée par les exemples (pp. 143-146) de
#+ "Advanced C Programming on the IBM PC", par Herbert Mayer
#+ (voir bibliographie).
#  Ceci ne fait que montrer que ce qui est fait en C peut aussi être fait avec
#+ des scripts shell.

}


#----- Maintenant, que le spectacle commence. --------#
charge_alpha     # Charge le tableau.
affiche_alpha    # L'affiche.
rotate           # Le fait basculer sur 45 degrés dans le sens contraire des
                 # aiguilles d'une montre.
#-----------------------------------------------------#

exit 0

# C'est une simulation assez peu satisfaisante.
#
# Exercices :
# ----------
# 1)  Réécrire le chargement du tableau et les fonctions d'affichage
#     d'une façon plus intuitive et élégante.
#
# 2)  Comprendre comment les fonctions de rotation fonctionnent.
#     Astuce : pensez aux implications de l'indexage arrière du tableau.
#
# 3)  Réécrire ce script pour gérer un tableau non carré, tel qu'un 6 sur 4.
#     Essayez de minimiser la distorsion lorsque le tableau subit une rotation.
```

Un tableau à deux dimensions est essentiellement équivalent à un tableau à une seule dimension mais avec des modes d'adressage supplémentaires pour les références et les manipulations d'éléments individuels par la position de la *ligne* et de la *colonne*.

Pour un exemple encore plus élaboré de simulation d'un tableau à deux dimensions, voir l'Exemple A.10, « « *life* : Jeu de la Vie » ».

Pour des scripts intéressants utilisant les tableaux, voir :

- Exemple 11.3, « Découvrir des anagrammes »
- Exemple A.24, « Encore plus sur les fonctions de hachage »
- Exemple A.42, « Un outil de résolution général »
- Exemple A.41, « Quacky : un jeu de mots de type Perquackey »

Chapitre 27. /dev et /proc

Une machine Linux ou UNIX a typiquement les répertoires spéciaux /dev et /proc.

27.1. /dev

Le répertoire /dev contient des entrées pour les *périphériques* physiques qui pourraient être présents sur votre système. [1] Les partitions du disque dur contenant les systèmes de fichiers montés ont des entrées dans /dev, comme un simple df le montre.

```
bash$ df
Filesystem           1k-blocks      Used Available Use% Mounted on
/dev/hda6               495876    222748    247527  48% /
/dev/hda1                50755      3887     44248   9% /boot
/dev/hda8               367013     13262    334803   4% /home
/dev/hda5              1714416   1123624    503704  70% /usr
```

Entre autre choses, le répertoire /dev contient des périphériques *loopback*, tels que /dev/loop0. Un périphérique loopback est une astuce qui permet à un fichier ordinaire d'être accédé comme s'il était un périphérique bloc. [2] Ceci rend possible le montage d'un système de fichiers entier en un seul gros fichier. Voir l'Exemple 16.8, « Création d'un système de fichiers dans un fichier » et l'Exemple 16.7, « Vérifier une image ».

Quelques pseudo-périphériques dans /dev ont d'autres utilisations spécialisées, telles que /dev/null, /dev/zero, /dev/urandom, /dev/sda1, /dev/udp, et /dev/tcp.

Par exemple :

Pour monter un lecteur flash USB, ajoutez la ligne suivante à /etc/fstab. [3]

```
/dev/sda1    /mnt/flashdrive    auto    noauto,user,noatime    0 0
```

(voir aussi l'Exemple A.25, « Monter des périphériques de stockage USB »).

Vérifier si un disque est dans le graveur de CD (lien vers /dev/hdc):

```
head -1 /dev/hdc


#  head: cannot open '/dev/hdc' for reading: No medium found
#  (Aucun disque dans le lecteur.)

#  head: error reading '/dev/hdc': Input/output error
#  (Il y a un disque dans le lecteur mais il ne peut être lu ;
#+  à priori un CDR vierge.)

#  Flux de caractères et autres cochonneries associées
#  (Il y a un disque pré-enregistré dans le lecteur
#+  et vous avez un affichage direct -- un flux de données ASCII et binaires.)
#  Ici nous voyons  we see l'intérêt de limiter la sortie dans des proportions
#+ gérables, plutôt que d'utiliser 'cat' ou un autre outil similaire.


#  Maintenant, il s'agit seulement de vérifier/analyser la sortie
#+ et d'exécuter une action appropriée.
```

Lors de l'exécution d'une commande sur le fichier pseudo-périphérique /dev/tcp/$host/$port, Bash ouvre une connexion TCP vers la *socket* associée.

> Un *socket* est un noeud de communications associé à un port d'entrée/sortie spécifique (ceci est équivalent à un *socket maté-*

[1] Les entrées dans /dev fournissent des points de montage pour les périphériques physiques et virtuels. Ces entrées utilisent très peu d'espace disque.

Quelques périphériques, tels que /dev/null, /dev/zero, et /dev/urandom sont virtuels. Ce ne sont pas des périphériques physiques et ils existent seulement au niveau logiciel.

[2] Un *périphérique bloc* lit et/ou écrit des données par morceaux, ou blocs en constraste avec un *périphérique caractère*, qui accède aux données caractère par caractère. Des exemples de périphérique bloc sont un disque dur et un lecteur CD ROM. Un exemple de périphérique caractère est un clavier.

[3] Bien sûr, le point de montage /mnt/lecteur_flash doit exister. Dans le cas contraire, en tant qu'utilisateur *root*, **mkdir /mnt/flashdrive**.

Pour monter réellement le lecteur, utilisez la commande suivante : **mount /mnt/lecteur_flash**

Les nouvelles distributions Linux montent automatiquement les clés USB dans le répertoire /media sans intervention de l'utilisateur.

riel ou à un *réceptacle*, pour un câble). Il permet le transfert de données entre les périphériques matériels sur la même machine, entre machines du même réseau, entre machines de différents réseaux et bien sûr entre différents emplacements sur Internet.

Les exemples suivants assument une connexion Internet active.

Obtenir l'heure de `nist.gov` :

```
bash$ cat </dev/tcp/time.nist.gov/13
53082 04-03-18 04:26:54 68 0 0 502.3 UTC(NIST) *
```

[Mark a contribué à l'exemple ci-dessus.]

Téléchargez une URL :

```
bash$ exec 5<>/dev/tcp/www.net.cn/80
bash$ echo -e "GET / HTTP/1.0\n" >&5
bash$ cat <&5
```

[Merci à Mark et Mihai Maties.]

Exemple 27.1. Utiliser `/dev/tcp` pour corriger des problèmes

```
#!/bin/bash
# dev-tcp.sh : redirection /dev/tcp pour vérifier la connexion Internet.

# Script de Troy Engel.
# Utilisé avec sa permission.

HOTE_TCP=www.dns-diy.com # Un FAI connu pour être "spam-friendly"
PORT_TCP=80              # Le port 80 est http.

# Essaie de se connecter. (Un peu équivalent à un 'ping'...)
echo "HEAD / HTTP/1.0" >/dev/tcp/${HOTE_TCP}/${PORT_TCP}
SORTIE=$?

: <<EXPLICATION
Si bash a été compilé avec --enable-net-redirections, il dispose d'un
périphérique caractère spécial pour les redirections TCP et UDP. Ces
redirections sont utilisées de façon identique à STDIN/STDOUT/STDERR. Les entrées
du périphériques sont 30,36 pour /dev/tcp :

  mknod /dev/tcp c 30 36

>De la référence bash :
    /dev/tcp/host/port
Si l'hôte est un nom valide ou une adresse Internet et que le port est un
numéro de port de type entier ou un nom de service, Bash essaie d'ouvrir une
connexion TCP vers le socket correspondant.
EXPLICATION


if [ "X$SORTIE" = "X0" ]; then
  echo "Connexion réussie. Code de sortie : $SORTIE"
else
  echo "Connexion échouée. Code de sortie : $SORTIE"
fi

exit $SORTIE
```

27.2. `/proc`

Le répertoire `/proc` est en fait un pseudo-système de fichiers. Les fichiers dans le répertoire `/proc` sont un miroir du système en

cours d'exécution et des processus14 du noyau, et contiennent des informations et des statistiques sur elles.

```
bash$ cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 14 sound
 29 fb
 36 netlink
128 ptm
136 pts
162 raw
254 pcmcia

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  9 md



bash$ cat /proc/interrupts
           CPU0
  0:      84505          XT-PIC  timer
  1:       3375          XT-PIC  keyboard
  2:          0          XT-PIC  cascade
  5:          1          XT-PIC  soundblaster
  8:          1          XT-PIC  rtc
 12:       4231          XT-PIC  PS/2 Mouse
 14:     109373          XT-PIC  ide0
NMI:          0
ERR:          0



bash$ cat /proc/partitions
major minor  #blocks  name     rio rmerge rsect ruse wio wmerge wsect wuse running use
aveq

    3     0    3007872 hda 4472 22260 114520 94240 3551 18703 50384 549710 0 111550
644030
    3     1      52416 hda1 27 395 844 960 4 2 14 180 0 800 1140
    3     2          1 hda2 0 0 0 0 0 0 0 0 0 0 0
    3     4     165280 hda4 10 0 20 210 0 0 0 0 0 210 210
    ...



bash$ cat /proc/loadavg
0.13 0.42 0.27 2/44 1119

bash$ cat /proc/apm
1.16 1.2 0x03 0x01 0xff 0x80 -1% -1 ?

bash$ cat /proc/acpi/battery/BAT0/info
present:                 yes
 design capacity:         43200 mWh
 last full capacity:      36640 mWh
 battery technology:      rechargeable
 design voltage:          10800 mV
 design capacity warning: 1832 mWh
 design capacity low:     200 mWh
 capacity granularity 1:  1 mWh
 capacity granularity 2:  1 mWh
```

```
 model number:            IBM-02K6897
 serial number:            1133
 battery type:            LION
 OEM info:                Panasonic



bash$ fgrep Mem /proc/meminfo
MemTotal:       515216 kB
 MemFree:        266248 kB
```

Des scripts shells peuvent extraire des données à partir de certains des fichiers de /proc. [4]

```
FS=iso
grep $FS /proc/filesystems # iso9660
```

```
kernel_version=$( awk '{ print $3 }' /proc/version )
```

```
CPU=$( awk '/model name/ {print $5}' < /proc/cpuinfo )

if [ "$CPU" = "Pentium(R)" ]
then
  lance_des_commandes
  ...
else
  lance_autres_commandes
  ...
fi



vitesse_cpu=$( fgrep "cpu MHz" /proc/cpuinfo | awk '{print $4}' )
#  La vitesse d'exécution actuelle (en MHz) du CPU de votre machine.
#  Sur un portable, cela peut varier suivant que vous utilisez la batterie
#+ ou le secteur.
```

```
#!/bin/bash
# get-commandline.sh
# Obtient les paramètres en ligne de commande d'un processus.

OPTION=cmdline

# Identifie le PID.
pid=$( echo $(pidof "$1") | awk '{ print $1 }' )
# Récupère seulement le premier
#                          ^^^^^^^^^^^^^^^^^^ de plusieurs instances.

echo
echo "ID du processus de (la première instance de) "$1" = $pid"
echo -n "Arguments en ligne de commande : "
cat /proc/"$pid"/"$OPTION" | xargs -0 echo
#   Formate la sortie :        ^^^^^^^^^^^^^^
#   (Merci pour la correction, Han Holl !)

echo; echo


# Par exemple :
# sh get-commandline.sh xterm
```

```
fichier_dev="/proc/bus/usb/devices"
texte="Spd"
USB1="Spd=12"
```

[4] Certaines commandes système, telles que procinfo, free, vmstat, lsdev, et uptime le font aussi.

```
USB2="Spd=480"


vitesse_bus=$(fgrep -m 1 "$texte" $fichier_dev | awk '{print $9}')
#                 ^^^^ s'arrête après la première correspondance.

if [ "$vitesse_bus" = "$USB1" ]
then
  echo "Port USB 1.1 découvert."
  # Faire quelque chose d'appropriée avec USB 1.1.
fi
```

Note

Il est même possible de contrôler certains périphériques avec des commandes envoyées au répertoire /proc.

```
root# echo on > /proc/acpi/ibm/light
```

Ceci active *Thinklight* sur certains modèles Thinkpad d'IBM/Lenovo. (Cela pourrait ne pas fonctionner sur toutes les distributions Linux.)

Bien sûr, vous devez faire particulièrement attention quand vous écrivez dans /proc.

Le répertoire /proc contient des sous-répertoires avec des noms numériques inhabituels. Chacun de ces noms correspond à un numéro de processus d'un processus en cours d'exécution. À l'intérieur de ces sous-répertoires, il existe un certain nombre de fichiers contenant des informations sur le processus correspondant. Les fichiers stat et status maintiennent des statistiques sur le processus, le fichier cmdline contient les arguments de la ligne de commande avec lesquels le processus a été appelé et le fichier exe est un lien symbolique vers le chemin complet du processus. Il existe encore quelques autres fichiers, mais ceux-ci sont les plus intéressants du point de vue de l'écriture de scripts.

Exemple 27.2. Trouver le processus associé à un PID

```
#!/bin/bash
# pid-identifier.sh
# Donne le chemin complet du processus associé avec ce pid.

NBARGS=1  # Nombre d'arguments que le script attend.
E_MAUVAISARGS=65
E_MAUVAISPID=66
E_PROCESSUS_INEXISTANT=67
E_SANSDROIT=68
PROCFILE=exe

if [ $# -ne $NBARGS ]
then
  echo "Usage : `basename $0` PID" >&2  # Message d'erreur >stderr.
  exit $E_MAUVAISARGS
fi

nopid=$( ps ax | grep $1 | awk '{ print $1 }' | grep $1 )
# Cherche le pid dans l'affichage de "ps" car il est le champ #1.
# S'assure aussi qu'il s'agit du bon processus, et non pas du processus appelé
# par ce script.
# Le dernier "grep $1" supprime cette possibilité.
#
#    pidno=$( ps ax | awk '{ print $1 }' | grep $1 )
#    fonctionne aussi comme l'indique Teemu Huovila.

if [ -z "$nopid" ] #  Si, après tous ces filtres, le résultat est une chaîne vide,
then               #+ aucun processus en cours ne correspond au pid donné.
  echo "Aucun processus en cours."
  exit $E_PROCESSUS_INEXISTANT
fi

# Autrement :
```

```
#   if ! ps $1 > /dev/null 2>&1
#   then                # Aucun processus ne correspond au pid donné.
#     echo "Ce processus n'existe pas"
#     exit $E_PROCESSUS_INEXISTANT
#    fi

# Pour simplifier tout cet algorithme, utilisez "pidof".


if [ ! -r "/proc/$1/$PROCFILE" ]  # Vérifiez les droits en lecture.
then
  echo "Processus $1 en cours, mais..."
  echo "Ne peut obtenir le droit de lecture sur /proc/$1/$PROCFILE."
  exit $E_SANSDROIT
    # Un utilisateur standard ne peut accéder à certains fichiers de /proc.
fi

# Les deux derniers tests peuvent être remplacés par :
#    if ! kill -0 $1 > /dev/null 2>&1 # '0' n'est pas un signal mais
                                      # ceci testera s'il est possible
                                      # d'envoyer un signal au processus.
#    then echo "PID n'existe pas ou vous n'êtes pas son propriétaire" >&2
#    exit $E_MAUVAISPID
#    fi



fichier_exe=$( ls -l /proc/$1 | grep "exe" | awk '{ print $11 }' )
# Ou       fichier_exe=$( ls -l /proc/$1/exe | awk '{print $11}' )
#
#  /proc/pid-number/exe est un lien symbolique
#+ vers le chemin complet du processus appelé.

if [ -e "$fichier_exe" ]  # Si /proc/pid-number/exe existe,
then                      # alors le processus correspondant existe.
  echo "Processus #$1 appelé par $fichier_exe"
else
  echo "Processus inexistant"
fi


#  Ce script élaboré peut *pratiquement* être remplacé par
#  ps ax | grep $1 | awk '{ print $5 }'
#  Néanmoins, cela ne fonctionnera pas...
#+ parce que le cinquième champ de 'ps' est le argv[0] du processus,
#+ et non pas le chemin vers l'exécutable.
#
#  Néanmoins, une des deux méthodes suivantes devrait fonctionner.
#       find /proc/$1/exe -printf '%l\n'
#       lsof -aFn -p $1 -d txt | sed -ne 's/^n//p'

# Commentaires supplémentaires par Stéphane Chazelas.

exit 0
```

Exemple 27.3. État de la connexion

```
#!/bin/bash

NOMPROC=pppd           # démon ppp.
NOMFICHIERPROC=status  # Où chercher.
NONCONNECTE=65
INTERVALLE=2           # Mise à jour toutes les deux secondes.

nopid=$( ps ax | grep -v "ps ax" | grep -v grep | grep $NOMPROC | awk '{ print $1 }' )
# Trouver le numéro de processus de 'pppd', le 'démon ppp'.
# Doit filtrer les lignes de processus générées par la recherche elle-même.
```

```
#
#  Néanmoins, comme Oleg Philon l'a indiqué,
#+ ceci pourrait être considérablement simplifié en utilisant "pidof".
#  nopid=$( pidof $NOMPROC )
#
#  Morale de l'histoire :
#+ Quand une séquence de commandes devient trop complexe, cherchez un raccourci.


if [ -z "$pidno" ]   # Si pas de pid, alors le processus ne tourne pas.
then
  echo "Non connecté."
  exit $NONCONNECTE
else
  echo "Connecté."; echo
fi

while [ true ]       # Boucle sans fin, le script peut être amélioré ici.
do

  if [ ! -e "/proc/$pidno/$NOMFICHIERPROC" ]
  #  Quand le processus est en cours d'exécution, alors le fichier "status"
  #+ existe.
  then
    echo "Déconnecté."
    exit $NONCONNECTE
  fi

netstat -s | grep "packets received"  #  Obtenir quelques statistiques de
netstat -s | grep "packets delivered" #+ connexion.


  sleep $INTERVALLE
  echo; echo

done

exit 0

# De cette façon, le script ne se termine qu'avec un Control-C.

#    Exercices :
#    ----------
#    Améliorer le script pour qu'il se termine suite à l'appui sur la touche
#    "q".
#    Rendre le script plus facilement utilisable d'autres façons.
```

Avertissement

En général, il est dangereux d'*écrire* dans les fichiers de `/proc` car cela peut corrompre le système de fichiers ou provoquer une erreur fatale.

Chapitre 28. Des Zéros et des Nulls

/dev/zero … /dev/null

Utilisation de /dev/null

Vous pouvez considérer /dev/null comme un *trou noir*. L'équivalent le plus proche serait un fichier en écriture seulement. Tout ce qui y est écrit disparaît à jamais. Toute tentative de lecture n'aboutira à rien. Néanmoins, /dev/null peut être très utile à la fois en ligne de commande et dans certains scripts.

Supprimer stdout.

```
cat $filename >/dev/null
# Le contenu de ce fichier ne s'affichera pas sur la sortie stdout.
```

Supprimer stderr (provenant de l'Exemple 15.3, « *Badname* élimine dans le répertoire courant les fichiers dont le nom contient des caractères incorrects et des espaces blancs. »).

```
rm $mauvaisnom 2>/dev/null
#           Donc les messages d'erreurs [stderr] disparaissent.
```

Supprimer les sorties *à la fois* de stdout et stderr.

```
cat $nom_de_fichier 2>/dev/null >/dev/null
# Si "$nom_de_fichier" n'existe pas, aucun message d'erreur ne s'affichera.
#  Si "$nom_de_fichier" existe bien, le contenu du fichier ne s'affichera pas sur
#+ la sortie stdout.
# Du coup, aucun affichage ne résultera de la ligne précédente.
#
#  Ceci peut être utile dans certaines situations où le code de retour d'une
#+ commande a besoin d'être testée, mais que sa sortie n'est pas souhaitée.
#
# cat $filename &>/dev/null
#     fonctionne aussi d'après l'indication de Baris Cicek.
```

Supprimer le contenu d'un fichier, mais en conservant le fichier lui-même, avec ses droits (provenant de l'Exemple 2.1, « *cleanup* : Un script pour nettoyer les journaux de trace dans /var/log » et l'Exemple 2.3, « *cleanup* : Une version améliorée et généralisée des scripts précédents ») :

```
cat /dev/null > /var/log/messages
#  : > /var/log/messages   a le même résultat mais ne lance pas un nouveau
processus.

cat /dev/null > /var/log/wtmp
```

Vider automatiquement le contenu d'un fichier de traces (spécialement intéressant pour s'occuper de ces fichiers dégoutants que sont les « cookies » envoyés par les sites Web commerciaux) :

Exemple 28.1. Cacher le cookie jar

```
# Navigateur Netscape obsolète.
# Les mêmes principes s'appliquent aux derniers navigateurs.

if [ -f ~/.netscape/cookies ]  # À supprimer s'il existe.
then
  rm -f ~/.netscape/cookies
fi

ln -s /dev/null ~/.netscape/cookies
# Maintenant, tous les cookies se trouvent envoyés dans un trou noir plutôt que
# d'être sauvé sur disque.
```

Utilisation de /dev/zero

Comme /dev/null, /dev/zero est un pseudo fichier périphérique mais il produit réellement un flux de caractères zéros (des zéros *binaires*, pas du type ASCII). Toute écriture dans /dev/zero disparait et il est plutôt difficile de lire les zéros à partir de là bien que ceci puisse se faire avec od ou un éditeur hexadécimal. L'utilisation principale de /dev/zero est de

créer un fichier factice initialisé à une taille prédéterminée pour en faire un fichier de swap temporaire.

Exemple 28.2. Créer un fichier de swap en utilisant `/dev/zero`

```
#!/bin/bash
# Crée un fichier de swap.

#  Un fichier swap fournit un espace de stockage pour le cache,
#+ ce qui aide à accélérer certaines opérations des systèmes de fichiers.

ROOT_UID=0                  # Root a l'$UID 0.
E_MAUVAIS_UTILISATEUR=65    # Pas root ?

FICHIER=/swap
TAILLEBLOC=1024
BLOCSMINS=40
SUCCES=0

# Ce script doit être exécuté en tant que root.
if [ "$UID" -ne "$ROOT_UID" ]
then
  echo; echo "Vous devez être root pour exécuter ce script."; echo
  exit $E_MAUVAIS_UTILISATEUR
fi


blocs=${1:-$BLOCSMINS}          #  Par défaut à 40 blocs, si rien n'est
                                #+ spécifié sur la ligne de commande.
# Ceci est l'équivalent du bloc de commandes ci-dessous.
# -----------------------------------------------------
# if [ -n "$1" ]
# then
#   blocs=$1
# else
#   blocs=$BLOCSMINS
# fi
# -----------------------------------------------------


if [ "$blocs" -lt $BLOCSMINS ]
then
  blocs=$BLOCSMINS             # Doit être au moins long de 40 blocs.
fi


###########################################################################
echo "Création du fichier swap d'une taille de $blocs blocs (Ko)."
dd if=/dev/zero of=$FICHIER bs=$TAILLEBLOC count=$blocs  # Vide le fichier.
mkswap $FICHIER $blocs            # Indique son type : swap.
swapon $FICHIER                   # Active le fichier swap.
#  Notez que si au moins une de ces commandes échouent,
#+ cela posera de sérieux problèmes.
###########################################################################

#  Exercice :
#  Ré-écrivez le bloc de code ci-dessus de façon à ce que,
#+ si une erreur se produit:
#    1) un message d'erreur est envoyé sur stderr,
#    2) tous les fichiers temporaires sont nettoyés,
#    3) le script sort immédiatement avec un code d'erreur approprié.

echo "Fichier swap créé et activé."

exit $SUCCES
```

Une autre application de `/dev/zero` est de « remplir de zéros » un fichier d'une taille spécifiée pour une raison particulière, telle que monter un système de fichiers sur un périphérique loopback399 (voir l'Exemple 16.8, « Création d'un système de fichiers dans un fichier ») ou telle que la suppression « sécurisée » d'un fichier (voir l'Exemple 15.59, « Effacer les fichiers de

façon sûre »).

Exemple 28.3. Créer un disque ram

```
#!/bin/bash
# ramdisk.sh

#  Un disque ram ("ramdisk") est un segment de mémoire RAM système agissant
#+ comme un système de fichiers.
#  Son avantage est son accès très rapide (temps de lecture/écriture).
#  Inconvénients : volatile, perte de données au redémarrage ou à l'arrêt,
#                  moins de RAM disponible pour le système.
#
#  En quoi un disque ram est intéressant ?
#  Conserver un ensemble de données large, comme une table ou un dictionnaire,
#+ sur un disque ram, accélère les recherches de données car l'accès mémoire est
#+ bien plus rapide que l'accès disque.


E_UTILISATEUR_NON_ROOT=70      # Doit être root.
NOM_UTILISATEUR_ROOT=root

POINT_MONTAGE=/mnt/ramdisk
TAILLE=2000                    # 2000 blocs (modifiez comme vous l'entendez)
TAILLE_BLOC=1024               # 1K (1024 octets) en taille de bloc
PERIPH=/dev/ram0               # Premier périphérique ram

nom_utilisateur=`id -nu`
if [ "$nom_utilisateur" != "$NOM_UTILISATEUR_ROOT" ]
then
  echo "Vous devez être root pour exécuter \"`basename $0`\"."
  exit $E_UTILISATEUR_NON_ROOT
fi

if [ ! -d "$POINT_MONTAGE" ]   #  Teste si le point de montage est déjà créé,
then                           #+ pour qu'il n'y ait pas d'erreur après
  mkdir $POINT_MONTAGE         #+ plusieurs exécutions de ce script
fi

###############################################################################
dd if=/dev/zero of=$PERIPH count=$TAILLE bs=$TAILLE_BLOC  # Vide le périphérique
                                                          #+ ram
mke2fs $PERIPH                 # Crée un système de fichiers ext2 dessus.
mount $PERIP $POINT_MONTAGE    # Monte le périphérique.
chmod 777 $POINT_MONTAGE       #  Pour que les utilisateurs standards puissent y
                               #+ accéder.
                               # Néanmoins, seul root pourra le démonter.
###############################################################################
#  Nécessite de tester si les commandes ci-dessus ont réussi. Pourrait poser
#+ problème sinon.
#  Exercice : modifiez ce script pour le rendre plus sûr.

echo "\"$POINT_MONTAGE\" est maintenant disponible"
# Le disque ram est maintenant accessible pour stocker des fichiers, y compris
# par un utilisateur standard.

#  Attention, le disque ram est volatile et son contenu disparaîtra au prochain
#+ redémarrage ou au prochain arrêt.
#  Copiez tout ce que vous voulez sauvegarder dans un répertoire standard.

# Après redémarrage, lancez de nouveau ce script pour initialiser le disque ram.
# Remonter /mnt/ramdisk sans les autres étapes ne fonctionnera pas.

#  Correctement modifié, ce script peut être appelé dans /etc/rc.d/rc.local,
#+ pour initialiser le ramdisk automatiquement au lancement.
#  Cela pourrait être approprié, par exemple, pour un serveur de bases de données.

exit 0
```

En plus de tout ce qui se trouve ci-dessus, `/dev/zero` est nécessaire pour les binaires UNIX/Linux ELF (acronyme de *Executable and Linking Format*).

Chapitre 29. Débogage

Le débogage est deux fois plus difficile que l'écriture de code en premier lieu. Donc, si vous écrivez du code aussi intelligemment que possible, vous êtes, par définition, pas assez intelligent pour le déboguer.

-- Brian Kernighan

Le shell Bash ne contient pas de débogueur intégré mais seulement des commandes et des constructions. Les erreurs de syntaxe ou de frappe dans les scripts génèrent des messages d'erreur incompréhensibles n'apportant souvent aucune aide pour déboguer un script non fonctionnel.

Exemple 29.1. Un script bogué

```
#!/bin/bash
# ex74.sh

# C'est un script bogué.
# Où est donc l'erreur ?

a=37

if [$a -gt 27 ]
then
  echo $a
fi

exit 0
```

Sortie d'un script :

```
./ex74.sh: [37: command not found
```

Que se passe-t'il avec ce script ? (petite aide : après le *if*

Exemple 29.2. Mot clé159 manquant

```
#!/bin/bash
# missing-keyword.sh : Quel message d'erreur sera généré ?

for a in 1 2 3
do
  echo "$a"
# done     # Requiert le mot clé 'done' mis en commentaire ligne 7.

exit 0
```

Sortie d'un script :

```
missing-keyword.sh: line 10: syntax error: unexpected end of file
```

Notez que le message d'erreur ne fait *pas* nécessairement référence à la ligne où l'erreur se trouve mais à la ligne où l'interpréteur Bash s'aperçoit de l'erreur.

Les messages d'erreur peuvent ne pas tenir compte des lignes de commentaires d'un script lors de l'affichage du numéro de ligne de l'instruction ayant provoqué une erreur de syntaxe.

Que faire si le script s'exécute mais ne fonctionne pas comme vous vous y attendiez ? C'est une erreur de logique trop commune.

Exemple 29.3. *test24*, un autre script bogué

```
#!/bin/bash

#  Ce script est supposé supprimer tous les fichiers du répertoire courant
```

```
#+ contenant des espaces dans le nom.
#  Cela ne fonctionne pas.
#  Pourquoi ?


mauvaisnom=`ls | grep ' '`

# Essayez ceci :
# echo "$mauvaisnom"

rm "$mauvaisnom"

exit 0
```

Essayez de trouver ce qui ne va pas avec l'Exemple 29.3, « *test24*, un autre script bogué » en supprimant les caractères de commentaires de la ligne **`echo "$mauvaisnom"`**. Les instructions echo sont utiles pour voir si ce que vous attendiez est bien ce que vous obtenez.

Dans ce cas particulier, **`rm "$mauvaisnom"`** ne donnera pas les résultats attendus parce que `$mauvaisnom` ne devrait pas être entre guillemets. Le placer entre guillemets nous assure que **rm** n'a qu'un seul argument (il correspondra à un seul nom de fichier). Une correction partielle est de supprimer les guillemets de `$mauvaisnom` et de réinitialiser `$IFS` pour contenir seulement un retour à la ligne, **`IFS=$'\n'`**. Néanmoins, il existe des façons plus simples de faire cela.

```
# Bonnes méthodes de suppression des fichiers contenant des espaces dans leur nom.
rm *\ *
rm *" "*
rm *' '*
# Merci, S.C.
```

Résumer les symptômes d'un script bogué,

1. Il quitte brutalement avec un message d'erreur de syntaxe (« syntax error »)
2. Il se lance bien mais ne fonctionne pas de la façon attendue (erreur logique, logic error).
3. Il fonctionne comme vous vous y attendiez mais a des effets indésirables déplaisants (logic bomb).

Il existe des outils pour déboguer des scripts non fonctionnels

1. Insérer des instructions echo aux points critiques du script pour tracer les variables, ou pour donner un état de ce qui se passe.

Astuce

Encore mieux, une instruction **echo** qui affiche seulement lorsque le mode de débogage (*debug*) est activé.

```
### debecho (debug-echo) par Stefano Falsetto ###
### Affichera les paramètres seulement si DEBUG est configuré. ###
debecho () {
  if [ ! -z "$DEBUG" ]; then
     echo "$1" >&2
     #         ^^^ vers stderr
  fi
}

DEBUG=on
Whatever=whatnot
debecho $Whatever   # whatnot

DEBUG=
Whatever=notwhat
debecho $Whatever   # (N'affichera rien.)
```

2. Utiliser le filtre tee pour surveiller les processus ou les données aux points critiques.
3. Initialiser des paramètres optionnelles `-n -v -x`

sh -n nomscript vérifie les erreurs de syntaxe sans réellement exécuter le script. C'est l'équivalent de l'insertion de **set -n** ou **set -o noexec** dans le script. Notez que certains types d'erreurs de syntaxe peuvent passer à côté de cette vérification.

sh -v nomscript affiche chaque commande avant de l'exécuter. C'est l'équivalent de l'insertion de **set -v** ou **set -o verbose** dans le script.

Les options `-n` et `-v` fonctionnent bien ensemble. **sh -nv nomscript** permet une vérification verbeuse de la syntaxe.

sh -x nomscript affiche le résultat de chaque commande mais d'une façon abrégée. C'est l'équivalent de l'insertion de **set -x** ou **set -o xtrace** dans le script.

Insérer **set -u** ou **set -o nounset** dans le script le lance mais donne un message d'erreur « unbound variable » à chaque essai d'utilisation d'une variable non déclarée.

4. Utiliser une fonction « assert » pour tester une variable ou une condition aux points critiques d'un script (cette idée est empruntée du C).

Exemple 29.4. Tester une condition avec un *assert*

```
#!/bin/bash
# assert.sh

assert ()                 #  Si la condition est fausse,
{                         #+ sort du script avec un message d'erreur.
  E_PARAM_ERR=98
  E_ASSERT_FAILED=99


  if [ -z "$2" ]          # Pas assez de paramètres passés
  then                    # à la fonction assert().
    return $E_PARAM_ERR   # Pas de dommages.
  fi

  noligne=$2

  if [ ! $1 ]
  then
    echo "Mauvaise assertion :  \"$1\""
    echo "Fichier \"$0\", ligne $noligne"
                          # Donne le nom du fichier et le numéro de ligne
    exit $E_ASSERT_FAILED
  # else (sinon)
  #   return (retour)
  #   et continue l'exécution du script.
  fi
} # Insèrer une fonction assert() similaire dans un script
  #+  que vous devez déboguer.
########################################################################


a=5
b=4
condition="$a -lt $b"     # Message d'erreur et sortie du script.
                          #  Essayer de configurer la "condition" en autre chose
                          #+ et voir ce qui se passe.

assert "$condition" $LINENO
# Le reste du script s'exécute si assert n'échoue pas.


# Quelques commandes.
# Quelques autres commandes...
echo "Cette instruction s'exécute seulement si \"assert\" n'échoue pas."
# ...
# Quelques commandes de plus.

exit $?
```

5. Utiliser la variable $LINENO et la commande interne caller.
6. Piéger la sortie.

   La commande exit d'un script déclenche un signal 0, terminant le processus, c'est-à-dire le script lui-même. [1] Il est souvent utilisé pour récupérer la main lors de *exit* en forçant un « affichage » des variables par exemple. Le *trap* doit être la première commande du script.

Récupérer les signaux

trap
Spécifie une action à la réception d'un signal ; aussi utile pour le débogage.

> Un *signal* est un message envoyé au processus, soit par le noyau soit par un autre processus lui disant de réaliser une action spécifiée (habituellement pour finir son exécution). Par exemple, appuyer sur Control-C25 envoie une interruption utilisateur, un signal INT, au programme en cours d'exécution.

Une simple instance :

```
trap '' 2
# Ignore l'interruption 2 (Control-C), sans action définie.

trap 'echo "Control-C désactivé."' 2
# Message lorsque Control-C est utilisé.
```

Exemple 29.5. Récupérer la sortie

```
#!/bin/bash
# Chasse aux variables avec un piège.

trap 'echo Liste de Variables --- a = $a  b = $b' EXIT
# EXIT est le nom du signal généré en sortie d'un script.
#
#  La commande spécifiée par le "trap" ne s'exécute pas
#+ tant que le signal approprié n'est pas envoyé.

echo "Ceci s'affiche avant le \"trap\" -- "
echo "même si le script voit le \"trap\" avant"
echo

a=39

b=36

exit 0
# Notez que mettre en commentaire la commande 'exit' ne fait aucune différence
# car le script sort dans tous les cas après avoir exécuté les commandes.
```

Exemple 29.6. Nettoyage après un Control-C

```
#!/bin/bash
# logon.sh: Un script rapide mais sale pour vérifier si vous êtes déjà connecté.

umask 177  #  S'assurer que les fichiers temporaires ne sont pas lisibles
           #+ par tout le monde.

VRAI=1
JOURNAL=/var/log/messages
# Notez que $JOURNAL doit être lisible (en tant que root, chmod 644 /var/log/messages).
```

[1] Par convention, `signal 0` est affecté à exit44.

```
FICHIER_TEMPORAIRE=temp.$$
# Crée un fichier temporaire "unique" en utilisant l'identifiant du processus.
#     Utiliser 'mktemp' est une alternative.
#     Par exemple :
#     FICTMP=`mktemp temp.XXXXXX`
MOTCLE=adresse
# À la connexion, la ligne "remote IP address xxx.xxx.xxx.xxx"
#                     ajoutée à /var/log/messages.
ENLIGNE=22
INTERRUPTION_UTILISATEUR=13
VERIFIE_LIGNES=100
# Nombre de lignes à vérifier dans le journal.

trap 'rm -f $FICHIER_TEMPORAIRE; exit $INTERRUPTION_UTILISATEUR' TERM INT
# Nettoie le fichier temporaire si le script est interrompu avec Control-C.

echo

while [ $VRAI ]  # Boucle sans fin.
do
  tail -n $VERIFIE_LIGNES $JOURNAL> $FICHIER_TEMPORAIRE
  # Sauve les 100 dernières lignes du journal dans un fichier temporaire.
  # Nécessaire car les nouveaux noyaux génèrent beaucoup de messages lors de la
  # connexion.
  search=`grep $MOTCLE $FICHIER_TEMPORAIRE`
  # Vérifie la présence de la phrase "IP address"
  # indiquant une connexion réussie.

  if [ ! -z "$search" ] # Guillemets nécessaires à cause des espaces possibles.
  then
     echo "En ligne"
     rm -f $FICHIER_TEMPORAIRE    # Suppression du fichier temporaire.
     exit $ENLIGNE
  else
     echo -n "."        # l'option -n supprime les retours à la ligne de echo,
                       # de façon à obtenir des lignes de points continus.
  fi

  sleep 1
done


# Note : Si vous modifiez la variable MOTCLE par "Exit",
# ce script peut être utilisé lors de la connexion pour vérifier une déconnexion
# inattendue.

# Exercice : Modifiez le script, suivant la note ci-dessus, et embellissez-le.

exit 0


# Nick Drage suggère une autre méthode.

while true
  do ifconfig ppp0 | grep UP 1> /dev/null && echo "connecté" && exit 0
  echo -n "."   # Affiche des points (.....) jusqu'au moment de la connexion.
  sleep 2
done

# Problème : Appuyer sur Control-C pour terminer ce processus peut être
# insuffisant (des points pourraient toujours être affichés).
# Exercice : Corrigez ceci.



# Stéphane Chazelas a lui-aussi suggéré une autre méthode.

CHECK_INTERVAL=1
```

```
while ! tail -n 1 "$JOURNAL" | grep -q "$MOTCLE"
do echo -n .
   sleep $CHECK_INTERVAL
done
echo "On-line"

# Exercice : Discutez les avantages et inconvénients de chacune des méthodes.
```

Note

L'argument DEBUG pour **trap** exécute une action spécifique après chaque commande dans un script. Cela permet de tracer les variables, par exemple.

Exemple 29.7. Tracer une variable

```
#!/bin/bash

trap 'echo "VARIABLE-TRACE> \$variable = \"$variable\""' DEBUG
# Affiche la valeur de $variable après chaque commande.

variable=29

echo "Initialisation de \"\$variable\" à $variable."

let "variable *= 3"
echo "Multiplication de \"\$variable\" par 3."

exit $?

#  La construction "trap 'commande1 ... commande2 ...' DEBUG" est plus
#+ appropriée dans le contexte d'un script complexe
#+ où placer plusieurs instructions "echo $variable" pourrait être
#+ difficile et consommer du temps.


# Merci, Stéphane Chazelas, pour cette information.

Affichage du script :

VARIABLE-TRACE> $variable = ""
VARIABLE-TRACE> $variable = "29"
Initialisation de "$variable" à 29.
VARIABLE-TRACE> $variable = "29"
VARIABLE-TRACE> $variable = "87"
Multiplication de "$variable" par 3.
VARIABLE-TRACE> $variable = "87"
```

Bien sûr, la commande **trap** a d'autres utilités en dehors du débogage.

Exemple 29.8. Lancer plusieurs processus (sur une machine SMP)

```
#!/bin/bash
# parent.sh
# Exécuter plusieurs processus sur une machine SMP.
# Auteur : Tedman Eng

#  Ceci est le premier de deux scripts,
#+ les deux étant présent dans le même répertoire courant.




LIMITE=$1         # Nombre total de processus à lancer
```

```
NBPROC=4          # Nombre de threads simultanés (processus fils ?)
IDPROC=1          # ID du premier processus
echo "Mon PID est $$"

function lance_thread() {
        if [ $IDPROC -le $LIMITE ] ; then
                ./child.sh $IDPROC&
                let "IDPROC++"
        else
           echo "Limite atteinte."
           wait
           exit
        fi
}

while [ "$NBPROC" -gt 0 ]; do
        lance_thread;
        let "NBPROC--"
done


while true
do

trap "lance_thread" SIGRTMIN

done

exit 0



# ======== Le deuxième script suit ========


#!/bin/bash
# child.sh
# Lancer plusieurs processus sur une machine SMP.
# Ce script est appelé par parent.sh.
# Auteur : Tedman Eng

temp=$RANDOM
index=$1
shift
let "temp %= 5"
let "temp += 4"
echo "Début $index  Temps :$temp" "$@"
sleep ${temp}
echo "Fin   $index"
kill -s SIGRTMIN $PPID

exit 0


# ==================== NOTES DE L'AUTEUR DU SCRIPT ==================== #
#  Ce n'est pas complètement sans bogue.
#  Je l'exécute avec limit = 500 et, après les premières centaines d'itérations,
#+ un des threads simultanés a disparu !
#  Pas sûr que ce soit dû aux collisions des signaux trap.
#  Une fois que le signal est reçu, le gestionnaire de signal est exécuté
#+ un bref moment mais le prochain signal est configuré.
#+ Pendant ce laps de temps, un signal peut être perdu,
#+ donc un processus fils peut manquer.

#  Aucun doute que quelqu'un va découvrir le bogue et nous l'indiquer
#+ ... dans le futur.
```

```
# ============================================================== #



# --------------------------------------------------------------#



#####################################################################
# Ce qui suit est le script original écrit par Vernia Damiano.
# Malheureusement, il ne fonctionne pas correctement.
#####################################################################

# multiple-processes.sh : Lance plusieurs processus sur une machine
# multi-processeurs.

# Script écrit par Vernia Damiano.
# Utilisé avec sa permission.

#  Doit appeler le script avec au moins un paramètre de type entier
#+ (nombre de processus concurrents).
#  Tous les autres paramètres sont passés aux processus lancés.


INDICE=8                 # Nombre total de processus à lancer
TEMPO=5                  # Temps de sommeil maximum par processus
E_MAUVAISARGUMENTS=65    # Pas d'arguments passés au script.

if [ $# -eq 0 ] # Vérifie qu'au moins un argument a été passé au script.
then
  echo "Usage: `basename $0` nombre_de_processus [paramètres passés aux processus]"
  exit $E_MAUVAISARGUMENTS
fi

NBPROCESSUS=$1          # Nombre de processus concurrents
shift
PARAMETRI=( "$@" )      # Paramètres de chaque processus

function avvia() {
        local temp
        local index
        temp=$RANDOM
        index=$1
        shift
        let "temp %= $TEMPO"
        let "temp += 1"
        echo "Lancement de $index Temps:$temp" "$@"
        sleep ${temp}
        echo "Fin $index"
        kill -s SIGRTMIN $$
}

function parti() {
        if [ $INDICE -gt 0 ] ; then
                avvia $INDICE "${PARAMETRI[@]}" &
                let "INDICE--"
        else
                trap : SIGRTMIN
        fi
}

trap parti SIGRTMIN

while [ "$NBPROCESSUS" -gt 0 ]; do
        parti;
        let "NBPROCESSUS--"
done

wait
```

```
trap - SIGRTMIN

exit $?

: &lt;&lt;COMMENTAIRES_AUTEUR_SCRIPT
J'avais besoin de lancer un programme avec des options spécifiées sur un certain
nombre de fichiers différents en utilisant une machine SMP. Donc, j'ai pensé
conserver un certain nombre de processus en cours et en lancer un à chaque fois
qu'un autre avait terminé.

L'instruction "wait" n'aide pas car il attend un processus donné ou "tous" les
processus exécutés en arrière-plan. Donc, j'ai écrit ce script bash, réalisant ce
travail en utilisant l'instruction "trap".
  --Vernia Damiano
COMMENTAIRES_AUTEUR_SCRIPT
```

Note

`trap '' SIGNAL` (deux apostrophes adjacentes) désactive SIGNAL pour le reste du script. **`trap SIGNAL`** restaure la fonctionnalité de SIGNAL. C'est utile pour protéger une portion critique d'un script d'une interruption indésirable.

```
trap '' 2  # Le signal 2 est Control-C, maintenant désactivé.
command
command
command
trap 2     # Réactive Control-C
```

La version 3469 de Bash ajoute les variables spéciales suivantes à utiliser par le débogueur.

1. $BASH_ARGC
2. $BASH_ARGV
3. $BASH_COMMAND
4. $BASH_EXECUTION_STRING
5. $BASH_LINENO
6. $BASH_SOURCE
7. $BASH_SUBSHELL

Chapitre 30. Options

Les options sont des paramétrages modifiant le comportement du shell et/ou du script.

La commande set active les options dans un script. Là où vous voulez que les options soient effectives dans le script, utilisez **set -o nom-option** ou, plus court, **set -abreviation-option**. Ces deux formes sont équivalentes.

```
#!/bin/bash

set -o verbose
# Affiche toutes les commandes avant exécution.
```

```
#!/bin/bash

set -v
# Même effet que ci-dessus.
```

Note

Pour *désactiver* une option dans un script, utilisez **set +o nom-option** ou **set +abreviation-option**.

```
#!/bin/bash

set -o verbose
# Mode echo des commandes activé.
commande
...
commande

set +o verbose
# Mode echo des commandes désactivé.
commande
# Pas d'affichage.


set -v
# Mode echo des commandes activé.
commande
...
commande

set +v
# Mode echo des commandes désactivé.
commande

exit 0
```

Une autre méthode d'activation des options dans un script est de les spécifier tout de suite après l'en-tête *#!* du script.

```
#!/bin/bash -x
#
# Le corps du script suit.
```

Il est aussi possible d'activer les options du script à partir de la ligne de commande. Certaines options qui ne fonctionneront pas avec **set** sont disponibles de cette façon. Parmi celles-ci se trouve *-i*, forçant un script à se lancer de manière interactive.

bash -v nom-script

`bash -o verbose nom-script`

Ce qui suit est une liste de quelques options utiles. Elles sont spécifiées soit dans leur forme abrégée (précédées par un simple tiret), soit par leur nom complet (précédées par un *double* tiret ou par un `-o`).

Tableau 30.1. Options de bash

Abréviation	Nom	Effet
`-C`	noclobber	Empêche l'écrasement de fichiers par une redirection (peut être outrepassé par >|)
`-D`	(aucune)	Affiche les chaînes entre guillemets préfixées par un $ mais n'exécute pas les commandes du script.
`-a`	allexport	Exporte toutes les variables définies
`-b`	notify	Notifie lorsque un travail en tâche de fond se termine (n'est pas d'une grande utilité dans un script)
`-c ...`	(aucune)	Lit les commandes à partir de **...**
`-e`	errexit	Annule un script à la première erreur lorsqu'une commande quitte avec un statut différent de zéro (sauf pour les boucles until ou while, les tests if et les constructions de listes)
`-f`	noglob	Expansion des noms de fichier désactivée
`-i`	interactive	Script lancé dans un mode *interactif*
`-n`	noexec	Lit les commandes du script, mais ne les exécute pas (vérification de syntaxe)
`-o Nom-Option`	(aucune)	Appelle l'option *Nom-Option*
`-o posix`	POSIX	Modifie le comportement de Bash, ou du script appelé, pour se conformer au standard POSIX.
`-o pipefail`	échec d'un tube	Fait qu'un tube renvoie le code de sortie44 de la dernière commande dans le tube qui a renvoyé un code de sortie différent de zéro.
`-p`	privileged	Script lancé avec « suid » (attention !)
`-r`	restricted	Script lancé en mode *restreint* (voir le Chapitre 21, Shells restreints).
`-s`	stdin	Lit les commandes à partir de l'entrée standard (`stdin`)
`-t`	(aucune)	Sort après la première commande
`-u`	nounset	Essayer d'utiliser une variable non définie affiche un message d'erreur et force l'arrêt du script
`-v`	verbose	Affiche chaque commande sur `stdout` avant de les exécuter
`-x`	xtrace	Similaire à `-v`, mais étend les commandes
`-`	(aucune)	Fin des options. Tous les autres arguments sont des paramètres de position.
`--`	(aucune)	Désinitialise les paramètres de position. Si des arguments sont donnés (`-- arg1 arg2`), les paramètres de position sont initialisés avec ces arguments.

Chapitre 31. Trucs et astuces

Turandot: Gli enigmi sono tre, la morte una!

Caleph: No, no! Gli enigmi sono tre, una la vita!
--Puccini

Voici quelques pratiques d'écriture de script (non recommandées !) qui apportent du piment aux vies sans relief.

- Affecter des mots réservés à des noms de variables.

```
case=value0       # Pose problème.
23skidoo=value1   # Et là-aussi.
# Les noms de variables avec un chiffre sont réservés par le shell.
# Essayez _23skidoo=value1. Commencer les variables avec un tiret bas est OK.

# Néanmoins...      n'utiliser que le tiret bas ne fonctionnera pas.
_=25
echo $_           # $_ est une variable spéciale initialisée comme étant le
                  # dernier argument de la dernière commande.

xyz((!*=value2    # Pose de sévères problèmes.
# À partir de la version 3 de Bash, les points ne sont plus autorisés dans les noms
de variables.
```

- Utiliser un tiret ou d'autres caractères réservés dans un nom de variable (ou un nom de fonction).

```
var-1=23
# Utilisez 'var_1' à la place.

fonction-quoiquecesoit () # Erreur
# Utilisez 'fonction_quoiquecesoit ()' à la place.


# À partir de la version 3 de Bash, les points ne sont plus autorisés dans les noms
de variables.
fonction.quoiquecesoit ()   # Erreur
# Utilisez 'fonctionQuoiquecesoit ()' à la place.
```

- Utiliser le même nom pour une variable et une fonction. Ceci rend le script difficile à comprendre.

```
fais_quelquechose ()
{
  echo "Cette fonction fait quelque chose avec \"$1\"."
}

fais_quelquechose=fais_quelquechose

fais_quelquechose fais_quelquechose

# Tout ceci est légal, mais porte à confusion.
```

- Utiliser des espaces blancs inappropriés. En contraste avec d'autres langages de programmation, Bash peut être assez chatouilleux avec les espaces blancs.

```
var1 = 23   # 'var1=23' est correct.
# Sur la ligne ci-dessus, Bash essaie d'exécuter la commande "var1"
# avec les arguments "=" et "23".

let c = $a - $b   # 'let c=$a-$b' et 'let "c = $a - $b"' sont corrects.

if [ $a -le 5]    # if [ $a -le 5 ]   est correct.
# if [ "$a" -le 5 ]   est encore mieux.
# [[ $a -le 5 ]] fonctionne aussi.
```

- Ne pas terminer avec un point-virgule la commande finale d'un bloc de code compris dans des accolades.

```
{ ls -l; df; echo "Done." }
```

```
# bash: syntax error: unexpected end of file

{ ls -l; df; echo "Done."; }
#                       ^     ### La commande finale nécessite un point-virgule.
```

- Supposer que des variables non initialisées (variables avant qu'une valeur ne leur soit affectée) sont « remplies de zéros ». Une variable non initialisée a une valeur *null*, et *non pas zéro*.

```
#!/bin/bash

echo "variable_non_initialisee = $variable_non_initialisee"
# variable_non_initialisee =
```

- Mélanger = et *-eq* dans un test. Rappelez-vous, = permet la comparaison de variables littérales et *-eq* d'entiers.

```
if [ "$a" = 273 ]      # $a est-il un entier ou une chaîne ?
if [ "$a" -eq 273 ]    # Si $a est un entier.

# Quelquefois, vous pouvez mélanger -eq et = sans mauvaises conséquences.
# Néanmoins...


a=273.0   # pas un entier.

if [ "$a" = 273 ]
then
  echo "La comparaison fonctionne."
else
  echo "La comparaison ne fonctionne pas."
fi    # La comparaison ne fonctionne pas.

# Pareil avec   a=" 273"  et a="0273".


# De même, problèmes en essayant d'utiliser "-eq" avec des valeurs non entières.

if [ "$a" -eq 273.0 ]
then
  echo "a = $a"
fi  # Échoue avec un message d'erreur.
# test.sh: [: 273.0: integer expression expected
```

- Mal utiliser les opérateurs de comparaison de chaînes.

Exemple 31.1. Les comparaisons d'entiers et de chaînes ne sont pas équivalentes

```
#!/bin/bash
# bad-op.sh : Essaie d'utiliser une comparaison de chaînes sur des entiers.

echo
nombre=1

#  La boucle "while" suivante contient deux "erreurs" :
#+ une évidente et une plus subtile.

while [ "$nombre" < 5 ]    # Mauvais ! Devrait être :  while [ "$nombre" -lt 5 ]
do
  echo -n "$nombre "
  let "nombre += 1"
done
#  Essayer de lancer ceci s'arrête avec ce message d'erreur :
#+ bad-op.sh: line 10: 5: No such file or directory
#  À l'intérieur de crochets simples, "<" doit être échappé,
#+ et, même là, c'est toujours mauvais pour comparer des entiers.


echo "----------------------"
```

```
while [ "$nombre" \< 5 ]    #  1 2 3 4
do                          #
  echo -n "$nombre "        #  Ceci *semble* fonctionner mais...
  let "nombre += 1"         #+ il fait réellement une comparaison ASCII
done                        #+ et non pas une comparaison numérique.

echo; echo "---------------------"

# Ceci peut causer des problèmes. Par exemple :

pluspetit=5
plusgrand=105

if [ "$plusgrand" \< "$pluspetit" ]
then
  echo "$plusgrand est plus petit que $pluspetit"
fi                          # 105 est plus petit que 5
#  En fait, "105" est réellement plus petit que "5"
#+ lors d'une comparaison de chaîne (ordre ASCII).

echo

exit 0
```

- Quelquefois, des variables à l'intérieur des crochets de « test » ([ ]) ont besoin d'être mises entre guillemets (doubles). Ne pas le faire risque de causer un comportement inattendu. Voir l'Exemple 7.6, « Vérification si une chaîne est *nulle* », l'Exemple 19.5, « Boucle *while* redirigée » et l'Exemple 9.6, « *arglist* : Affichage des arguments avec $* et $@ ».
- Mettre entre guillement une variable contenant des espaces blancs empêche la division37. Quelque fois, c'est la cause de conséquences inattendues38.
- Les commandes lancées à partir d'un script peuvent échouer parce que le propriétaire d'un script ne possède pas les droits d'exécution. Si un utilisateur ne peut exécuter une commande à partir de la ligne de commande, alors la placer dans un script échouera de la même façon. Essayer de changer les droits de la commande en question, peut-être même en initialisant le bit suid (en tant que *root*, bien sûr).
- Tenter d'utiliser - comme opérateur de redirection (qu'il n'est pas) résultera habituellement en une surprise peu plaisante.

```
commande1 2> - | commande2
# Essayer de rediriger la sortie d'erreurs dans un tube...
# ... ne fonctionnera pas

commande1 2>& - | commande2 # Aussi futile.

Merci, S.C.
```

- Utiliser les fonctionnalités de Bash version 2+465 peut poser des soucis avec les messages d'erreur. Les anciennes machines Linux peuvent avoir une version 1.XX de Bash suite à une installation par défaut.

```
#!/bin/bash

minimum_version=2
# Comme Chet Ramey ajoute constamment de nouvelles fonctionnalités à Bash,
# vous pourriez configurer $minimum_version à 2.XX, 3.XX, ou quoi que ce soit
# de plus approprié.
E_MAUVAISE_VERSION=80

if [ "$BASH_VERSION" \< "$minimum_version" ]
then
  echo "Ce script fonctionne seulement avec Bash, version $minimum ou ultérieure."
  echo "Une mise à jour est fortement recommandée."
  exit $E_MAUVAISE_VERSION
fi

...
```

- Utiliser les fonctionnalités spécifiques à Bash dans un script shell Bourne (**#!/bin/sh**) sur une machine non Linux peut causer un comportement inattendu. Un système Linux crée habituellement un alias **sh** vers **bash**, mais ceci n'est pas nécessairement vrai pour une machine UNIX générique.
- Utiliser des fonctionnalités non documentées de Bash se révèle être un pratique dangereuse. Dans les précédentes versions de ce livre, plusieurs scripts dépendaient d'une « fonctionnalité » qui, bien que la valeur maximum d'un exit44 ou d'un return358 soit 255, faisait que cette limite ne s'appliquait pas aux entiers *négatifs*. Malheureusement, à partir de la version 2.05b et des suivantes, cela a disparu. Voir Exemple 23.9, « Tester les valeurs de retour importantes dans une fonction ».
- Un script avec des retours à la ligne DOS (*\r\n*) ne pourra pas s'exécuter car **#!/bin/bash\r\n** n'est pas *reconnu*, *pas* la même chose que l'attendu **#!/bin/bash\n**. La correction est de convertir le script en des retours chariots style UNIX.

```
#!/bin/bash

echo "Ici"

unix2dos $0    # Le script se modifie lui-même au format DOS.
chmod 755 $0   # et modifie son droit d'exécution.
               # La commande 'unix2dos' supprime le doit d'exécution.

./$0           # Le script essaie de se lancer de nouveau.
               # Mais cela ne fonctionnera pas en tant que format DOS.

echo "Là"

exit 0
```

- Un script shell commençant par **#!/bin/sh** ne se lancera pas dans un mode de compatibilité complète avec Bash. Quelques fonctions spécifiques à Bash pourraient être désactivées. Les scripts qui ont besoin d'un accès complet à toutes les extensions spécifiques à Bash devraient se lancer avec **#!/bin/bash**.
- Placer une espace blanche devant la chaîne de limite d'un document en ligne318 pourra causer un comportement inattendu dans un script.
- Placer plus d'une instruction *echo* dans une fonction dont la sortie est capturée457.

```
add2 ()
{
  echo "N'importe quoi... "   # Supprimez cette ligne !
  let "retval = $1 + $2"
    echo $retval
    }

    num1=12
    num2=43
    echo "Somme de $num1 et $num2 = $(add2 $num1 $num2)"

#   Somme de 12 et 43 = N'importe quoi...
#   55

#        Les "echo" se concatènent.
```

Ceci ne fonctionnera pas458.

- Un script peut ne pas faire un **export** de ses variables à son processus parent, le shell ou à l'environnement. Comme nous l'avons appris en biologie, un processus fils peut hériter de son parent, mais le contraire n'est pas vrai.

```
NIMPORTEQUOI=/home/bozo
export NIMPORTEQUOI
exit 0
```

```
bash$ echo $NIMPORTEQUOI



bash$
```

De façon certaine, au retour à l'invite de commande, $NIMPORTEQUOI reste sans valeur.

- Initialiser et manipuler des variables dans un sous-shell342, puis essayer d'utiliser ces mêmes variables en dehors du sous-shell

résultera en une mauvaise surprise.

Exemple 31.2. Problèmes des sous-shell

```
#!/bin/bash
# Problèmes des variables dans un sous-shell.

variable_externe=externe
echo
echo "variable_externe = $variable_externe"
echo

(
# Début du sous-shell

echo "variable_externe à l'intérieur du sous-shell = $variable_externe"
variable_interne=interne  # Configure
echo "variable_interne à l'intérieur du sous-shell = $variable_interne"
variable_externe=interne  # Sa valeur va-t'elle changer globalement?
echo "variable_externe à l'intérieur du sous-shell = $variable_externe"

# Est-ce qu'un export fera une différence ?
#    export variable_interne
#    export variable_externe
# Essayez.

# Fin du sous-shell
)

echo
echo "variable_interne à l'extérieur du sous-shell = $variable_interne"  #
Désinitialise.
echo "variable_externe à l'extérieur du sous-shell = $variable_externe"  # Non
modifié.
echo

exit 0

# Qu'arrive-t'il si vous décommentez les lignes 19 et 20 ?
# Cela fait-il une différence ?
```

- Envoyer dans un tube la sortie de **echo** pour un read peut produire des résultats inattendus. Dans ce scénario, **read** agit comme si elle était lancée dans un sous-shell. À la place, utilisez la commande set (comme dans l'Exemple 14.18, « Réaffecter les paramètres de position »).

Exemple 31.3. Envoyer la sortie de *echo* dans un tube pour un *read*

```
#!/bin/bash
#  badread.sh :
#  Tentative d'utiliser 'echo' et 'read'
#+ pour affecter non interactivement des variables.

a=aaa
b=bbb
c=ccc

echo "un deux trois" | read a b c
# Essaie d'affecter a, b et c.

echo
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
# L'affectation a échoué.

# ------------------------------
```

```
# Essaie l'alternative suivante.

var=`echo "un deux trois"`
set -- $var
a=$1; b=$2; c=$3

echo "-------"
echo "a = $a"  # a = un
echo "b = $b"  # b = deux
echo "c = $c"  # c = trois
# Affectation réussie.

# ------------------------------

#  Notez aussi qu'un echo pour un 'read' fonctionne à l'intérieur d'un
#+ sous-shell.
#  Néanmoins, la valeur de la variable change *seulement* à l'intérieur du
#+ sous-shell.

a=aaa          # On recommence.
b=bbb
c=ccc

echo; echo
echo "un deux trois" | ( read a b c;
echo "À l'intérieur du sous-shell : "; echo "a = $a"; echo "b = $b"; echo "c = $c" )
# a = un
# b = deux
# c = trois
echo "-----------------"
echo "À l'extérieur du sous-shell : "
echo "a = $a"  # a = aaa
echo "b = $b"  # b = bbb
echo "c = $c"  # c = ccc
echo

exit 0
```

En fait, comme l'indique Anthony Richardson, envoyer via un tube à partir de *n'importe quelle* boucle peut amener des problèmes similaires.

```
# Problèmes des tubes dans des boucles.
#  Exemple de Anthony Richardson
#+ avec un ajout de Wilbert Berendsen.


trouve=false
find $HOME -type f -atime +30 -size 100k |
while true
do
   read f
   echo "$f a une taille supérieure à 100 Ko et n'a pas été utilisé depuis au moins
30 jours."
   echo "Prenez en considération le déplacement de ce fichier dans les archives."
   trouve=true
   # ------------------------------------
   echo "Niveau de sous-shell = $BASH_SUBSHELL"
   # Niveau de sous-shell = 1
   # Oui, nous sommes dans un sous-shell.
   # ------------------------------------
done

#  trouve sera toujours faux car il est initialisé dans un sous-shell.
if [ $trouve = false ]
then
   echo "Aucun fichier ne doit être archivé."
```

```
fi

# ================Maintenant, voici une façon correcte de le faire :============

trouve=false
for f in $(find $HOME -type f -atime +30 -size 100k)  # Pas de tube ici.
do
   echo "$f a une taille supérieure à 100 Ko et n'a pas été utilisé depuis au moins
30 jours."
   echo "Prenez en considération le déplacement de ce fichier dans les archives."
   trouve=true
done

if [ $trouve = false ]
then
   echo "Aucun fichier ne doit être archivé."
fi

# =================Et voici une autre alternative=================

#  Place la partie du script lisant les variables à l'intérieur d'un bloc de
#+ code de façon à ce qu'ils partagent le même sous-shell.
#  Merci, W.B.

find $HOME -type f -atime +30 -size 100k | {
     trouve=false
     while read f
     do
       echo "$f a une taille supérieure à 100 Ko et n'a pas été utilisé depuis au
moins 30 jours."
       echo "Prenez en considération le déplacement de ce fichier dans les
archives."
       trouve=true
     done

     if ! $trouve
     then
       echo "Aucun fichier ne doit être archivé."
     fi
}
```

Un problème relatif survient lors de la tentative d'écriture sur `stdout` par un **tail -f** envoyé via un tube sur grep.

```
tail -f /var/log/messages | grep "$MSG_ERREUR" >> erreur.log
# Le fichier "erreur.log" ne sera pas écrit.
```

- Utiliser les commandes « suid » à l'intérieur de scripts est risqué et peut compromettre la sécurité de votre système. [1]
- Utiliser des scripts shell en programmation CGI peut être assez problématique. Les variables des scripts shell ne sont pas « sûres » et ceci peut causer un comportement indésirable en ce qui concerne CGI. De plus, il est difficile de « sécuriser » des scripts shell.
- Bash ne gère pas la chaîne double slash (//) correctement.
- Les scripts Bash écrits pour Linux ou BSD peuvent nécessiter des corrections pour fonctionner sur une machine UNIX commerciale (ou Apple OSX). De tels scripts emploient souvent l'ensemble GNU de commande et de filtres qui ont plus de fonctionnalités que leur contrepartie UNIX. Ceci est particulièrement vrai pour les utilitaires texte comme tr.

Danger is near thee --

Beware, beware, beware, beware.

Many brave hearts are asleep in the deep.

So beware --

Beware.
--A.J. Lamb and H.W. Petrie

[1] Ajouter le droit suid53 sur le script lui-même n'a aucun effet sous Linux et sur les plupart des UNIX.

Chapitre 32. Écrire des scripts avec style

Prenez l'habitude d'écrire vos scripts shell d'une façon structurée et méthodique. Même des scripts écrits « sur le dos d'une enveloppe » et « sans trop réfléchir » peuvent en bénéficier si vous prenez le temps de plannifier et d'organiser vos pensées avant de vous assoir pour l'écrire.

Du coup, il existe quelques lignes de conduites pour le style. Ceci n'a pas pour but d'être la *feuille de style officielle pour l'écriture de scripts*.

32.1. Feuille de style non officielle d'écriture de scripts

- Commentez votre code. Cela le rend plus facile à comprendre (et apprécier) par les autres, et plus facile pour vous à maintenir.

```
PASS="$PASS${MATRIX:$(($RANDOM%${#MATRIX})):1}"
#  Cela avait un sens lorsque vous l'aviez écrit l'année dernière, mais
#+ c'est un mystère complet.
# (À partir du script "pw.sh" de Antek Sawicki.)
```

  Ajoutez des en-têtes descriptives à votre script et à vos fonctions.

```
#!/bin/bash

#********************************************#
#                  xyz.sh                    #
#          écrit par by Bozo Bozeman         #
#                05 Juil. 2001               #
#                                            #
#   Fait le ménage dans les fichiers projets.    #
#********************************************#

E_MAUVAISREPERTOIRE=65          # Répertoire inexistant.
rep_projets=/home/bozo/projects # Répertoire à nettoyer.

# ------------------------------------------------------------ #
# nettoie_pfichiers ()                                         #
# Supprime tous les fichiers dans le répertoire indiqué.       #
# Paramètre : $repertoire_cible                                #
# Renvoie   : 0 en cas de succès, $E_MAUVAISREPERTOIRE sinon. #
# ------------------------------------------------------------ #
nettoie_pfichiers ()
{
  if [ ! -d "$1" ]  # Teste si le répertoire cible existe.
  then
    echo "$1 n'est pas un répertoire."
    return $E_MAUVAISREPERTOIRE
  fi

  rm -f "$1"/*
  return 0   # Succès.
}

nettoie_pfichiers $rep_projets

exit 0
```

  Assurez-vous de mettre le *#!/bin/bash* au début de la première ligne d'un script, précédant tout en-tête de commentaires.

- Éviter d'utiliser des « nombres magiques », [1] c'est-à-dire des constantes littérales « codées en dur ». Utilisez des noms de variables significatifs à la place. Ceci rend le script plus facile à comprendre et permet de faire des changements et des mises à jour sans casser l'application.

```
if [ -f /var/log/messages ]
then
  ...
fi
# Un an après, vous décidez de changer le script pour vérifier /var/log/syslog.
```

[1] Dans ce contexte, les « nombres magiques » ont une signification entièrement différente que les nombres magiques utilisés pour désigner les types de fichier.

```
#  Il est maintenant nécessaire de changer manuellement le script, instance par
#+ instance, et espérer ne rien casser.

# Un meilleur moyen :
FICHIERTRACE=/var/log/messages  # Seule une ligne a besoin d'être modifié.
if [ -f "$FICHIERTRACE" ]
then
  ...
fi
```

- Choisissez des noms descriptifs pour les variables et les fonctions.

```
fl=`ls -al $nomrep`                  # Crypté.
liste_fichiers=`ls -al $nomrep`      # Mieux.


VALMAX=10   # Tout en majuscule pour les constantes du script.
while [ "$index" -le "$VALMAX" ]
...


E_PASTROUVE=75                       # Tout en majuscule pour les codes d'erreur,
                                     # et leur nom commence par "E_".
if [ ! -e "$nomfichier" ]
then
  echo "Fichier $nomfichier introuvable."
  exit $E_PASTROUVE
fi


REPERTOIRE_MAIL=/var/spool/mail/bozo # Tout en majuscule pour une variable
d'environnement.
export REPERTOIRE_MAIL


ObtientReponse ()                    # Majuscule et minuscule pour une fonction.
{
  invite=$1
  echo -n $invite
  read reponse
  return $reponse
}

ObtientReponse "Quel est votre nombre favori ? "
nombre_favori=$?
echo $nombre_favori


_variableutilisateur=23              # OK, mais pas recommandé.
#  Il est mieux pour les variables définies par les utilisateurs de ne pas
#+ commencer avec un tiret bas.
# Laissez cela pour les variables système.
```

- Utiliser des codes de sortie44 d'une façon systématique et significative.

```
E_MAUVAIS_ARGS=65
...
...
exit $E_MAUVAIS_ARGS
```

Voir aussi l'Annexe D, Codes de sortie ayant une signification particulière.

Ender suggère l'utilisation des codes de sortie contenus dans `/usr/include/sysexits.h` dans les scripts shell bien qu'ils aient pour but la programmation en C et C++.

- Utilisez les options de paramètres standardisées pour l'appel de script. *Ender* propose l'ensemble d'options suivant.

```
-a      All: renvoie toutes les informations
        (incluant les informations de fichiers cachés).
```

```
-b      Bref: Version courte, généralement pour les autres scripts.
-c      Copie, concatène, etc.
-d      Daily: Utilise l'information du jour complet et non pas seulement
        l'information pour une instance ou pour un utilisateur spécifique.
-e      Étendu/Élaboré: (n'inclut souvent pas les informations de fichiers
        cachés).
-h      Help: Aide, indication verbeuse sur l'utilisation avec description,
        discussion, aide.
        Voir aussi -V.
-l      Traces du script.
-m      Manuel: page man de la commande de base.
-n      Nombre: Données numériques seulement.
-r      Récursif: Tous les fichiers d'un répertoire (et/ou tous les
        sous-répertoires).
-s      Setup & Maintenance fichier: Fichier de configuration de ce script.
-u      Usage: Liste des options à l'appel du script.
-v      Verbeux: Sortie lisible par un humain, plus ou moins formaté.
-V      Version / Licence / Copy(right|left) / Contributions (par courrier
        électronique aussi).
```

Voir aussi l'Section F.1, « Options standards en ligne de commande ».

- Casser les scripts complexes en modules plus simples. Utiliser des fonctions si c'est approprié. Voir l'Exemple 34.4, « Utiliser des tableaux et autres astuces pour gérer quatre mains aléatoires dans un jeu de cartes ».
- N'utilisez pas une construction complexe lorsqu'une construction plus simple fait l'affaire.

```
COMMANDE
if [ $? -eq 0 ]
...
# Redondant et non intuitif.

if COMMANDE
...
# Plus concis (même si moins compréhensible).
```

... reading the UNIX source code to the Bourne shell (/bin/sh). I was shocked at how much simple algorithms could be made cryptic, and therefore useless, by a poor choice of code style. I asked myself, « Could someone be proud of this code? »

... lisant le code source UNIX du shell Bourne (/bin/sh). J'ai été choqué de voir à quel point de simples algorithmes pouvaient être rendus incompréhensibles, et du coup inutiles, par un mauvais choix dans le style de codage. Je me suis demandé, « Quelqu'un peut-il être fier de ce code ? »
--Landon Noll

Chapitre 33. Divers

Personne ne connait réellement ce qu'est la grammaire du shell Bourne. Même l'examen du code source est de peu d'aide.

-- Tom Duff

33.1. Shells et scripts interactifs et non interactifs

Un shell *interactif* lit les commandes à partir de l'entrée utilisateur sur un terminal `tty`. Entre autres choses, un tel script lit les fichiers de démarrage lors de l'activation, affiche une invite et active un contrôle de job par défaut. L'utilisateur peut *interagir* avec le shell.

Un shell exécutant un script est toujours un shell non interactif. Tout de même, le script peut toujours accéder au `tty`. Il est même possible d'émuler un shell interactif dans un script.

```
#!/bin/bash
MON_INVITE='$ '
while :
do
  echo -n "$MON_INVITE"
  read ligne
  eval "$ligne"
  done

exit 0

# Ce script d'exemple et la plupart des explications ci-dessus sont apportés
# par Stéphane Chazelas (encore merci).
```

Considérons un script *interactif* qui demande une saisie de l'utilisateur, habituellement avec des fonctions read (voir l'Exemple 14.3, « Affectation d'une variable, en utilisant *read* »). Dans la « vraie vie », c'est en fait un peu moins simple que ça. À partir de maintenant, on supposera qu'un script interactif est lié à un terminal tty, script appelé par un utilisateur à partir d'une console ou d'un *xterm*.

Des scripts d'initialisation et de démarrage sont nécessairement non interactifs car ils doivent fonctionner sans intervention humaine. Beaucoup de scripts administratifs et de maintenance système sont aussi non interactifs. Les tâches répétitives invariables nécessitent une automatisation par des scripts non interactifs.

Les scripts non interactifs peuvent fonctionner en arrière-plan alors que les interactifs sont suspendus attendant une saisie qui ne viendra jamais. Gérez cette difficulté en utilisant un script **expect** ou une entrée intégrée de type document en ligne318 vers un script interactif fonctionnant comme une tâche de fond. Dans le cas le plus simple, redirigez un fichier pour apporter l'entrée à la fonction **read** (**read variable <fichier**). Ces détournements particuliers rendent possible l'utilisation de scripts à usage général tournant en mode soit interactif soit non interactif.

Si un script a besoin de tester si, oui ou non, il est exécuté de manière interactive, il suffit simplement de savoir si la variable de l'*invite*, $PS1, est configurée (si le script attend une saisie de l'utilisateur, alors il a besoin d'afficher une invite).

```
if [ -z $PS1 ] # pas d'invite ?
then
  # non interactif
  ...
else
  # interactif
  ...
fi
```

Comme alternative, le script peut tester la présence de l'option « i » dans le drapeau $-.

```
case $- in
*i*)    # shell interactif
;;
*)      # shell non interactif
;;
# (D'après "UNIX F.A.Q.", 1993)
```

Note

Il est possible de forcer un script à fonctionner en mode interactif avec l'option -i ou avec l'en-tête **`#!/bin/bash -i`**. Faites attention au fait que ceci peut entraîner un comportement étrange du script ou afficher des messages d'erreurs même si aucune erreur n'est présente.

33.2. Précédence des opérateurs

Dans un script, les opérations sont exécutées dans l'ordre de *précédence* : les opérations de plus haute précédence s'exécutent *avant* celles de plus basse. [1]

Tableau 33.1. Précédence des opérateurs

Opérateur	Signification	Commentaires
		PLUS HAUTE PRÉCÉDENCE
`var++ var--`	post-incrément, post-décrément	Opérateurs style C
`++var --var`	pre-incrément, pre-décrément	
`! ~`	négation	logique / bit, inverse le sens de l'opérateur qui suit
`**`	exposant	opération arithmétique
`* / %`	multiplication, division, modulo	opération arithmétique
`+ -`	addition, soustraction	opération arithmétique
`<< >>`	décalage à gauche et à droite	bit
`-z -n`	comparaison *unitaire*	chaîne est/n'est pas *null*
`-e -f -t -x, etc.`	comparaison *unitaire*	*fichier test*
`< -lt > -gt <= -le >= -ge`	comparaison *composée*	string and integer
`-nt -ot -ef`	comparaison *composée*	*fichier test*
`== -eq != -ne`	égalité / différence	opérateurs de test, chaîne et entier
`&`	AND	bit
`^`	XOR	OU *exclusif*, bit
`|`	OU	bit
`&& -a`	ET	logique, comparaison *composée*
`|| -o`	OR	logique, comparaison *composée*
`?:`	opérateur à trois arguments14	C-style
`=`	affectation	(ne pas confondre avec les tests d'*égalité*)
`*= /= %= += -= <<= >>= &=`	combinaison d'affectation	égal + multiplication, égal + division, égal + modulo, etc.
`,`	virgule	lie un ensemble d'operations
		PLUS BASSE PRÉCÉDENCE

[1] La *précédence*, dans ce contexte, correspond en quelque sorte à une *priorité*.

En pratique, vous avez seulement besoin de vous rappeler ce qui suit :

- Le mantra « My Dear Aunt Sally » (les initiales forment l'acronyme de *multiplication*, *division*, *addition*, *soustraction*) pour les opérations arithmétiques habituelles.
- Les opérateurs de logiques *composés*, **&&**, ||, **-a**, et **-o** ont une basse précédence.
- L'ordre d'évaluation des opérateurs de précédence identique est habituellement *de gauche à droite*.

Maintenant, utilisons nos connaissances de la précédence des opérateurs pour analyser quelques lignes du fichier `/etc/init.d/functions` trouvé dans la distribution *Fedora Core*.

```
while [ -n "$remaining" -a "$retry" -gt 0 ]; do

# Cela semble complexe à première vue.


# Séparer les conditions :
while [ -n "$remaining" -a "$retry" -gt 0 ]; do
#       --condition 1-- ^^ --condition 2-

#  Si la variable "$remaining" n'est pas vide
#+      ET (-a)
#+ que la valeur de la variable "$retry" est supérieure à zéro
#+ alors
#+ l' [ expression-entre-crochets ] renvoit le code de succès (0)
#+ et la boucle while exécute une itération.
#  ============================================================
#  Évaluer "condition 1" et "condition 2" ***avant***
#+ de les grouper avec AND.
#  Pourquoi ? Parce que AND (-a) a une précédence plus basse
#+ que les opérateurs -n et -gt,
#+ et, du coup, est évalué en *dernier*.

###################################################################

if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then


# De nouveau, séparer les conditions :
if [ -f /etc/sysconfig/i18n -a -z "${NOLOCALE:-}" ] ; then
#    --condition 1--------- ^^ --condition 2-----

#  Si le fichier "/etc/sysconfig/i18n" existe
#+      ET (-a)
#+ que la variable $NOLOCALE est vide
#+ alors
#+ le [ test-expression-dans-les-crochets ] renvoit le code de succès (0)
#+ et les commandes suivantes sont exécutées.
#
#  Comme auparavant, le AND (-a) est évalué en *dernier*
#+ parce qu'il a une précédence plus basse que les tests entre crochets.
#  ============================================================
#  Note :
#  ${NOLOCALE:-} est l'expansion d'un paramètre qui semble redondant.
#  Mais, si $NOLOCALE n'a pas été déclaré, elle est initialisé à *null*,
#+ donc, dans les faits, la déclare.
#  Ceci représente une différence essentielle dans certains contextes.
```

Astuce

Pour éviter la confusion ou des erreurs dans une séquence complexe d'opérateurs de tests, cassez la séquence en sections entre crochets.

```
if [ "$v1" -gt "$v2"  -o  "$v1" -lt "$v2"  -a  -e "v" ]
# Difficile de savoir ce qu'il s'y passe...

if [[ "$v1" -gt "$v2" ]] || [[ "$v1" -lt "$v2" ]] && [[ -e "$nomfichier" ]]
```

```
        # Bien mieux -- les tests sont groupés en sections logiques.
```

33.3. Scripts d'appel

Un « script d'appel » (*wrapper*) est un script shell qui inclut une commande système ou un utilitaire, qui sauvegarde un ensemble de paramètres passés à cette commande. [2] Intégrer un script dans une ligne de commande complexe simplifie son appel. Ceci est vraiment utile avec sed631 et awk634.

Un script **sed** ou **awk** est normalement appelé à partir de la ligne de commande par un **sed -e <replaceable>'commandes'</replaceable>** ou **awk <replaceable>'commandes'</replaceable>**. Intégrer ce type de script dans un script Bash permet de l'appeler plus simplement et le rend « réutilisable ». Ceci autorise aussi la combinaison des fonctionnalités de **sed** et **awk**, par exemple pour renvoyer dans un tuyau, la sortie d'un ensemble de commandes **sed** vers **awk**. Comme un fichier exécutable sauvé, vous pouvez alors l'appeler de manière répétée dans sa forme originale ou modifiée, sans les inconvénients d'avoir à le retaper sur la ligne de commande.

Exemple 33.1. ***Script d'appel***

```
#!/bin/bash

# C'est un simple script supprimant les lignes blanches d'un fichier.
# Pas de vérification des arguments.
#
# Vous pouvez ajouter quelque chose comme ça :
# E_SANSARGS=65
# if [ -z "$1" ]
# then
#  echo "Usage : `basename $0` fichier-cible"
#  exit $E_SANSARGS
# fi


# Identique à
#    sed -e '/^$/d' nomfichier
# appelé à partir de la ligne de commande.

sed -e /^$/d "$1"
#  Le '-e' signifie qu'une commande d'"édition" suit (optionnel ici).
#  '^' est le début de la ligne, '$' en est la fin.
#  Ceci correspond aux lignes n'ayant rien entre le début et la fin de la ligne.
#  'd' est la commande de suppression.

#  Mettre entre guillemets l'argument de la ligne de commande permet de saisir
#+ des espaces blancs et des caractères spéciaux dans le nom du fichier.

#  Notez que ce script ne modifie pas réellement le fichier cible.
#  Si vous avez besoin de le faire, redirigez sa sortie.

exit 0
```

Exemple 33.2. Un ***script d'appel*** **légèrement plus complexe**

```
#!/bin/bash

#  "subst", un script qui substitue un modèle pour un autre dans un fichier,
#+ c'est-à-dire "subst Smith Jones lettre.txt".

ARGS=3             # Le script nécessite trois arguments.
E_MAUVAISARGS=65   # Mauvais nombre d'arguments passé au script.

if [ $# -ne "$ARGS" ]
# Teste le nombre d'arguments du script (toujours une bonne idée).
```

[2] Un assez grand nombre d'outils Linux sont, en fait, des scripts d'appel. Quelques exemples parmi d'autres : `/usr/bin/pdf2ps`, `/usr/bin/batch` et `/usr/X11R6/bin/xmkmf`.

```
then
  echo "Usage : `basename $0` ancien-modele nouveau-modele nom-fichier"
  exit $E_MAUVAISARGS
fi

ancien_modele=$1
nouveau_modele=$2

if [ -f "$3" ]
then
    nom_fichier=$3
else
        echo "Le fichier \"$3\" n'existe pas."
    exit $E_MAUVAISARGS
fi

#  Voici où se trouve le vrai boulot.
sed -e "s/$ancien_modele/$nouveau_modele/g" $nom_fichier
#  Bien sûr, 's' est la commande de substitut dans sed,
#+ et /modele/ appelle la correspondance d'adresse.
#  "g" ou l'option globale est la cause de la substitution pour *toute*
#+ occurence de $ancien_modele sur chaque ligne, pas seulement la première.
#  Lisez les documents sur 'sed' pour une explication en profondeur.

exit 0    # Appel avec succès du script qui renvoie 0.
```

Exemple 33.3. Un *script d'appel* générique qui écrit dans un fichier de traces

```
#!/bin/bash
#  Emballage générique qui réalise une opération et la trace.

# Doit configurer les deux variables suivantes.
OPERATION=
#         Peut-être une chaîne complexe de commandes,
#+        par exemple un script awk ou un tube...
JOURNAL=
#         Arguments en ligne de commande, au cas où, pour l'opération.


OPTIONS="$@"


# La tracer.
echo "`date` + `whoami` + $OPERATION "$@"" >> $JOURNAL
# Maintenant, l'exécuter.
exec $OPERATION "$@"

# Il est nécessaire de tracer avant d'exécuter l'opération.
# Pourquoi ?
```

Exemple 33.4. Un *script d'appel* autour d'un script awk

```
#!/bin/bash
# pr-ascii.sh : affiche une table de caractères ASCII.

DEBUT=33   # Liste de caractères ASCII affichables (décimal).
FIN=125

echo " Décimal   Hex     Caractère"   # En-tête.
echo " -------   ---     ---------"

for ((i=DEBUT; i<=FIN; i++))
do
  echo $i | awk '{printf("  %3d       %2x         %c\n", $1, $1, $1)}'
# Le printf intégré de Bash ne fonctionnera pas dans ce contexte :
```

```
#     printf "%c" "$i"
done

exit 0


#  Décimal   Hex     Caractère
#  -------   ---     ---------
#    33       21         !
#    34       22         "
#    35       23         #
#    36       24         $
#
#    . . .
#
#   122       7a         z
#   123       7b         {
#   124       7c         |
#   125       7d         }


#  Redirigez la sortie de ce script vers un fichier
#+ ou l'envoyez via un tube dans "more" :  sh pr-asc.sh | more
```

Exemple 33.5. Un *script d'appel* autour d'un autre script awk

```
#!/bin/bash

# Ajoute une colonne spécifiée (de nombres) dans le fichier cible.

ARGS=2
E_MAUVAISARGS=65

if [ $# -ne "$ARGS" ] # Vérifie le bon nombre d'arguments sur la ligne de
                      # de commandes.
then
   echo "Usage : `basename $0` nomfichier numéro_colonne"
   exit $E_MAUVAISARGS
fi

nomfichier=$1
numero_colonne=$2

# Passer des variables shell à la partie awk du script demande un peu d'astuces.
#  Une méthode serait de placer des guillemets forts sur la variable du script
#+ Bash à l'intérieur du script awk.
#     $'$BASH_SCRIPT_VAR'
#      ^                ^
# C'est fait dans le script awk embarqué ci-dessous.
# Voir la documentation awk pour plus de détails.

# Un script multi-ligne awk est appelé par awk : ' ..... '


# Début du script awk.
# ----------------------------
awk '

{ total += $'"${numero_colonne}"'
}
END {
     print total
}

' "$nomfichier"
# ----------------------------
# Fin du script awk.
```

```
#   Il pourrait ne pas être sûr de passer des variables shells à un script awk
#+  embarqué, donc Stephane Chazelas propose l'alternative suivante :
#   -------------------------------------
#   awk -v numero_colonne="$numero_colonne" '
#   { total += $numero_colonne
#   }
#   END {
#       print total
#   }' "$nomfichier"
#   -------------------------------------


exit 0
```

Pour ces scripts nécessitant un seul outil qui-fait-tout, il existe une espèce de couteau suisse nommée *Perl*. Perl combine les capacités de **sed** et **awk**, et y ajoute un grand sous-ensemble de fonctionnalités *C*. Il est modulaire et contient le support de pratiquement tout ce qui est connu en commençant par la programmation orientée. Des petits scripts Perl vont eux-mêmes s'intégrer dans d'autres scripts, et il existe quelques raisons de croire que Perl peut totalement remplacer les scripts shells (bien que l'auteur de ce document reste sceptique).

Exemple 33.6. Perl inclus dans un script *Bash*

```
#!/bin/bash

# Les commandes shell peuvent précéder un script Perl.
echo "Ceci précède le script Perl embarqué à l'intérieur de \"$0\"."
echo "============================================================="

perl -e 'print "Ceci est un script Perl embarqué.\n";'
# Comme sed, Perl utilise aussi l'option "-e".

echo "============================================================="
echo "Néanmoins, le script peut aussi contenir des commandes shell et système."

exit 0
```

Il est même possible de combiner un script Bash et un script Perl dans le même fichier. Dépendant de la façon dont le script est invoqué, soit la partie Bash soit la partie Perl sera exécutée.

Exemple 33.7. Combinaison de scripts Bash et Perl

```
#!/bin/bash
# bashandperl.sh

echo "Bienvenue dans la partie Bash de ce script."
# Plus de commandes Bash peuvent suivre ici.

exit 0
# Fin de la partie Bash de ce script.

# ======================================================

#!/usr/bin/perl
# Cette partie du script doit être appelé avec l'option -x.

print "Bienvenue de la partie Perl de ce script.\n";
# Plus de commandes Perl peuvent suivre ici.

# Fin de la partie Perl de ce script.
```

```
bash$ bash bashandperl.sh
Bienvenue de la partie Bash du script.


bash$ perl -x bashandperl.sh
Bienvenue de la partie Perl du script.
```

33.4. Tests et comparaisons : alternatives

Pour les tests, la construction [[ ]]50 peut être plus appropriée que **[ ]**. De même, les comparaisons arithmétiques pourraient bénéficier de la construction (( )).

```
a=8

# Toutes les comparaisons ci-dessous sont équivalentes.
test "$a" -lt 16 && echo "oui, $a < 16"         # "liste et"
/bin/test "$a" -lt 16 && echo "oui, $a < 16"
[ "$a" -lt 16 ] && echo "oui, $a < 16"
[[ $a -lt 16 ]] && echo "oui, $a < 16"          # Placer les variables entre
(( a < 16 )) && echo "oui, $a < 16"             # [[ ]] et (( )) n'est pas
                                                # nécessaire.

cite="New York"
# Encore une fois, toutes les comparaisons ci-dessous sont équivalentes.
test "$cite" \< Paris && echo "Oui, Paris est plus grand que $cite"
    # Ordre ASCII du plus grand.
/bin/test "$cite" \< Paris && echo "Oui, Paris est plus grand que $cite"
[ "$cite" \< Paris ] && echo "Oui, Paris est plus grand que $cite"
[[ $cite < Paris ]] && echo "Oui, Paris est plus grand que $cite"
    # Pas besoin de mettre $cite entre double quote.

# Merci, S.C.
```

33.5. Un script s'appelant lui-même (récursion)

Un script peut-il s'appeler récursivement365 ? En fait, oui.

Exemple 33.8. Un script (inutile) qui s'appelle récursivement

```
#!/bin/bash
# recurse.sh

# Un script peut-il s'appeler récursivement ?
# Oui, mais est-ce d'une utilité quelconque ?
# Voir le script suivant.

ECHELLE=10
VALMAX=9

i=$RANDOM
let "i %= $ECHELLE"  # Génère un nombre aléatoire entre 0 et $VALMAX.

if [ "$i" -lt "$VALMAX" ]
then
  echo "i = $i"
  ./$0       #  Le script lance récursivement une nouvelle instance de lui-même.
fi           #  Chaque fil du script fait de même jusqu'à ce que la valeur
             #+ générée $i soit égale à $VALMAX.

# Utiliser une boucle "while" au lieu d'un test "if/then" pose des problèmes.
# Expliquez pourquoi.

exit 0

# Note :
```

```
# -----
# Ce script doit avoir le droit d'exécution pour fonctionner correctement.
# C'est le cas même s'il est appelé par une commande "sh".
# Expliquez pourquoi.
```

Exemple 33.9. Un script (utile) qui s'appelle récursivement

```
#!/bin/bash
# pb.sh : carnet de téléphones.

# Écrit par Rick Boivie et utilisé avec sa permission.
# Modifications par l'auteur du guide ABS

MINARGS=1       # Le script a besoin d'au moins un argument.
FICHIERDONNEES=./carnet_telephone
          #  Un fichier de données du répertoire courant, nommé
          #+ "carnet_telephone", doit exister.
NOMPROG=$0
E_SANSARGS=70   # Erreur lorsque sans arguments.

if [ $# -lt $MINARGS ]; then
      echo "Usage : "$NOMPROG" donnees-a-rechercher"
      exit $E_SANSARGS
fi


if [ $# -eq $MINARGS ]; then
      grep $1 "$FICHIERDONNEES"
      # 'grep' affiche un message d'erreur si $FICHIERDONNEES n'existe pas.
else
      ( shift; "$NOMPROG" $* ) | grep $1
      # Le script s'appelle récursivement.
fi

exit 0        #  Le script sort ici.
              #  On peut mettre des commentaires sans '#' et des données après
              #+ ce point.

# ------------------------------------------------------------------------
# Exemple d'un carnet d'adresses :

John Doe        1555 Main St., Baltimore, MD 21228          (410) 222-3333
Mary Moe        9899 Jones Blvd., Warren, NH 03787          (603) 898-3232
Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678
Zoe Zenobia     4481 N. Baker St., San Francisco, SF 94338   (415) 501-1631
# ------------------------------------------------------------------------

$bash pb.sh Roe
Richard Roe     856 E. 7th St., New York, NY 10009          (212) 333-4567
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

$bash pb.sh Roe Sam
Sam Roe         956 E. 8th St., New York, NY 10009          (212) 444-5678

#  Lorsqu'au moins un argument est passé au script, celui-ci n'affiche *que*
#+ le(s) ligne(s) contenant tous les arguments.
```

Exemple 33.10. Un autre script (utile) qui s'appelle récursivement

```
#!/bin/bash
# usrmnt.sh, écrit par Anthony Richardson
# Utilisé avec sa permission.

# usage :       usrmnt.sh
```

```
# description : monte un périphérique, l'utilisateur l'appelant doit être listé
#               dans le groupe MNTUSERS dans le fichier /etc/sudoers.

# ------------------------------------------------------------------
# C'est un script usermount qui se relance lui-même en utilisant sudo.
# Un utilisateur avec seulement les bons droits doit taper

#   usermount /dev/fd0 /mnt/floppy

# au lieu de

#   sudo usermount /dev/fd0 /mnt/floppy

#  J'utilise cette même technique pour tous mes scripts sudo
#+ parce que je la trouve convenable.
# ------------------------------------------------------------------

#  Si la variable SUDO_COMMAND n'est pas initialisée, nous ne sommes pas exécutés
#+ à partir de sudo, donc nous le relançons nous-même. Passez les identifiants
#+ réels de l'utilisateur et du groupe...

if [ -z "$SUDO_COMMAND" ]
then
   mntusr=$(id -u) grpusr=$(id -g) sudo $0 $*
   exit 0
fi

# Nous arriverons ici que si le script a été exécuté via sudo
/bin/mount $* -o uid=$mntusr,gid=$grpusr

exit 0

# Notes supplémentaires (de l'auteur de ce script) :
# --------------------------------------------------

# 1) Linux permet l'option "users" dans le fichier /etc/fstab
#    de façon à ce que tout utilisateur puisse monter un media modifiable.
#    Mais, sur un serveur, j'aime autoriser seulement quelques accès
#    individuels au média modifiable. Je trouve qu'utiliser sudo me donne
#    plus de contrôle.

# 2) Je trouve aussi sudo plus convenable pour accomplir cette tâche plutôt
#    qu'utiliser les groupes.

# 3) Cette méthode donne à chacun les bons droits (accès root) pour la commande
#    mount, donc faites bien attention à qui vous donnez accès. Vous pouvez
#    obtenir un contrôle plus fin sur les accès de montage en utilisant cette
#    même technique dans des scripts séparés : mntfloppy, mntcdrom et mntsamba.
```

Attention

Trop de niveaux de récursivité peut surcharger la pile du script, causant une erreur de segmentation (segfault).

33.6. « Coloriser » des scripts

Les séquences d'échappement d'ANSI [3] permettent de régler les attributs de l'écran, tels que le texte en gras et la couleur d'affichage et de fond. Les fichiers batch DOS659 utilisaient communément les séquences d'échappement ANSI pour les affichages *couleur*, comme peuvent le faire les scripts Bash.

Exemple 33.11. Une base de données d'adresses « colorisée »

```
#!/bin/bash
# ex30a.sh : Version "colorisée" de ex30.sh.
```

[3] ANSI est, bien sûr, l'acronyme pour « American National Standards Institute ». Ce corps auguste établit et maintient différents standards techniques et industriels.

```
#            Base de données d'adresses.


clear                                   # Efface l'écran.

echo -n "          "
echo -e '\E[37;44m'"\033[1mListe de contacts\033[0m"
                                        # Blanc sur fond bleu
echo; echo
echo -e "\033[1mChoisissez une des personnes suivantes :\033[0m"
                                        # Bold
tput sgr0
echo "(Entrez seulement les premières lettres du nom)"
echo
echo -en '\E[47;34m'"\033[1mE\033[0m"   # Bleu
tput sgr0                               #  Réinitialise les couleurs à la
                                        #+ "normale."
echo "vans, Roland"                     # "[E]vans, Roland"
echo -en '\E[47;35m'"\033[1mJ\033[0m"   # Magenta
tput sgr0
echo "ones, Mildred"
echo -en '\E[47;32m'"\033[1mS\033[0m"   # Vert
tput sgr0
echo "mith, Julie"
echo -en '\E[47;31m'"\033[1mZ\033[0m"   # Rouge
tput sgr0
echo "ane, Morris"
echo

read personne

case "$personne" in
# Notez que la variable est entre guillemets.

  "E" | "e" )
  # Accepte une entrée en majuscule ou minuscule.
  echo
  echo "Roland Evans"
  echo "4321 Floppy Dr."
  echo "Hardscrabble, CO 80753"
  echo "(303) 734-9874"
  echo "(303) 734-9892 fax"
  echo "revans@zzy.net"
  echo "Business partner & old friend"
  ;;

  "J" | "j" )
  echo
  echo "Mildred Jones"
  echo "249 E. 7th St., Apt. 19"
  echo "New York, NY 10009"
  echo "(212) 533-2814"
  echo "(212) 533-9972 fax"
  echo "milliej@loisaida.com"
  echo "Girlfriend"
  echo "Birthday: Feb. 11"
  ;;

# Ajoutez de l'info pour Smith & Zane plus tard.

          * )
   # Option par défaut.
   # Une entrée vide (en appuyant uniquement sur RETURN) vient ici aussi.
   echo
   echo "Pas encore dans la base de données."
  ;;

esac
```

```
tput sgr0                                   #  Réinitialisation des couleurs à la
                                            #+ "normale".

echo

exit 0
```

Exemple 33.12. Dessiner une boîte

```
#!/bin/bash
# Draw-box.sh : Dessine une boîte en utilisant des caractères ASCII.

#  Script de Stefano Palmeri, avec quelques modifications mineures par
#+ l'auteur de ce document.
#  Modifications mineures suggérées par Jim Angstadt.
#  Utilisé dans le guide ABS avec sa permission.


######################################################################
###  doc de la fonction dessine_une_boite  ###

#  La fonction "dessine_une_boite" permet à l'utilisateur de dessiner une boîte
#+ dans un terminal.
#
#  Usage : dessine_une_boite LIGNE COLONNE HAUTEUR LARGEUR [COULEUR]
#  LIGNE et COLONNE représente la position de l'angle gauche en haut pour la
#+ boîte que vous dessinez.
#  LIGNE et COLONNE doivent être supérieurs à 0 et inférieurs aux dimensions
#+ actuelles du terminal.
#  HAUTEUR est le nombre de lignes de la boîte et doit être positif.
#  HAUTEUR + LIGNE doit être inférieur à la hauteur actuelle du terminal.
#  LARGEUR est le nombre de colonnes de la boîte et doit être positif.
#  LARGEUR + COLONNE doit être inférieur ou égale à la largeur actuelle du
#+ terminal.
#
# C'est-à-dire que si la dimension de votre terminal est de 20x80,
#  dessine_une_boite 2 3 10 45 est bon
#  dessine_une_boite 2 3 19 45 n'a pas une bonne HAUTEUR (19+2 > 20)
#  dessine_une_boite 2 3 18 78 n'a pas une bonne LARGEUR (78+3 > 80)
#
#  COULEUR est la couleur du cadre de la boîte.
#  Ce cinquième argument est optionnel.
#  0=noir 1=rouge 2=vert 3=tan 4=bleu 5=violet 6=cyan 7=blanc.
#  Si vous passez de mauvais arguments à la fonction,
#+ elle quittera avec le code 65
#+ et aucun message ne sera affiché sur stderr.
#
#  Effacez le terminal avant de dessiner une boîte.
#  La commande clear n'est pas contenue dans la fonction.
#  Cela permet à l'utilisateur de dessiner plusieurs boîtes, y compris en les
#  entre-mêlant.

###  fin de la doc sur la fonction dessine_une_boite  ###
######################################################################

dessine_une_boite(){

#=============#
HORZ="-"
VERT="|"
CARACTERE_DU_COIN="+"

MINARGS=4
E_MAUVAISARGS=65
#=============#
```

```
if [ $# -lt "$MINARGS" ]; then       # Si moins de quatre arguments, quitte.
    exit $E_MAUVAISARGS
fi

# Recherche de caractères non numériques dans les arguments.
# Cela pourrait être mieux fait (exercice pour le lecteur ?).
if echo $@ | tr -d [:blank:] | tr -d [:digit:] | grep . &> /dev/null; then
   exit $E_MAUVAISARGS
fi

HAUTEUR_BOITE=`expr $3 - 1`   #  -1, correction nécessaire car le caractère
                              #+ de l'angle, "+", fait partie à la fois de
LARGEUR_BOITE=`expr $4 - 1`   #+ la hauteur et de la largeur.
T_LIGNES=`tput lines`         #  Définit les dimensions actuels du terminal
T_COLONNES=`tput cols`        #+ en lignes et colonnes.

if [ $1 -lt 1 ] || [ $1 -gt $T_LIGNES ]; then    #  Commence la vérification des
   exit $E_MAUVAISARGS                           #+ arguments.
fi
if [ $2 -lt 1 ] || [ $2 -gt $T_COLONNES ]; then
   exit $E_MAUVAISARGS
fi
if [ `expr $1 + $HAUTEUR_BOITE + 1` -gt $T_LIGNES ]; then
   exit $E_MAUVAISARGS
fi
if [ `expr $2 + $LARGEUR_BOITE + 1` -gt $T_COLONNES ]; then
   exit $E_MAUVAISARGS
fi
if [ $3 -lt 1 ] || [ $4 -lt 1 ]; then
   exit $E_MAUVAISARGS
fi                                # Fin de la vérification des arguments.

plot_char(){                      # Fonction à l'intérieur d'une fonction.
   echo -e "\E[${1};${2}H"$3
}

echo -ne "\E[3${5}m"              #  Initialise la couleur du cadre de la boîte
                                  #+ si elle est définie.

# start drawing the box

compteur=1                                       #  Dessine les lignes verticales
for (( r=$1; compteur<=$HAUTEUR_BOITE; r++)); do #+ en utilisant la fonction
  plot_char $r $2 $VERT                          #+ plot_char.
  let compteur=compteur+1
done

compteur=1
c=`expr $2 + $LARGEUR_BOITE`
for (( r=$1; compteur<=$HAUTEUR_BOITE; r++)); do
  plot_char $r $c $VERT
  let compteur=compteur+1
done

compteur=1                                       #  Dessine les lignes horizontales
for (( c=$2; compteur<=$LARGEUR_BOITE; c++)); do #+  en utilisant la fonction
  plot_char $1 $c $HORZ                          #+ plot_char.
  let compteur=compteur+1
done

compteur=1
r=`expr $1 + $HAUTEUR_BOITE`
for (( c=$2; compteur<=$LARGEUR_BOITE; c++)); do
  plot_char $r $c $HORZ
  let compteur=compteur+1
done

plot_char $1 $2 $CARACTERE_DU_COIN          # Dessine les angles de la boîte.
plot_char $1 `expr $2 + $LARGEUR_BOITE` $CARACTERE_DU_COIN
```

```
plot_char `expr $1 + $HAUTEUR_BOITE` $2 $CARACTERE_DU_COIN
plot_char `expr $1 + $HAUTEUR_BOITE` `expr $2 + $LARGEUR_BOITE` $CARACTERE_DU_COIN

echo -ne "\E[0m"               #  Restaure les anciennes couleurs.

P_ROWS=`expr $T_LIGNES - 1`    #  Place l'invite au bas du terminal.

echo -e "\E[${P_ROWS};1H"
}


# Maintenant, essayons de dessiner une boîte.
clear                                # Efface le terminal.
R=2      # Ligne
C=3      # Colonne
H=10     # Hauteur
W=45     # Largeur
col=1    # Couleur (rouge)
dessine_une_boite $R $C $H $W $col   # Dessine la boîte.

exit 0

# Exercice :
# ---------
# Ajoutez l'option d'impression de texte dans la boîte dessinée.
```

La séquence d'échappement ANSI la plus simple et peut-être la plus utile est du texte gras, **\033[1m ... \033[0m**. \033 représente un escape39, « [1 » active l'attribut gras, alors que « [0 » la désactive. « m » termine chaque terme de la séquence d'échappement.

```
bash$ echo -e "\033[1mCeci est un texte en gras.\033[0m"
```

Une séquence d'échappement similaire active l'attribut de soulignement (sur un *rxvt* et un *aterm*).

```
bash$ echo -e "\033[4mCe texte est souligné.\033[0m"
```

Note

Avec un **echo**, l'option `-e` active les séquences d'échappement.

D'autres séquences d'échappement modifie le texte et/ou la couleur du fond.

```
bash$ echo -e '\E[34;47mCeci est affiché en bleu.'; tput sgr0


bash$ echo -e '\E[33;44m'"texte jaune sur fond bleu"; tput sgr0

bash$ echo -e '\E[1;33;44m'"texte jaune en gras sur
fond bleu"; tput sgr0
```

Note

Il est généralement conseillé d'initialiser l'attribut *gras* pour le texte coloré avec des teintes claires.

tput sgr0 restaure les paramétrages du terminal en normal. L'omettre laisse toute sortie ultérieure à partir de ce terminal en bleu.

Note

Comme **tput sgr0** échoue lors de la restauration des paramètres du terminal sous certaines circonstances, **echo -ne \E[0m** pourrait être un meilleur choix.

Utiliser le modèle suivant pour écrire du texte coloré sur un fond coloré.

```
echo -e '\E[COLOR1;COLOR2mDu texte vient ici.'
```

Les caractères « \E[» commencent la séquence d'échappement. Les nombres « COLOR1 » et « COLOR2 » séparés par le point-virgule spécifient une couleur de texte et de fond, suivant la table ci-dessous (l'ordre des nombres importe peu car les nombres d'avant et d'arrière-plan tombent dans des plages qui ne se couvrent pas). « m » termine la séquence d'échappement et le texte commence immédiatement après ça.

Notez aussi que les guillemets simples enferment le reste de la séquence de commandes suivant le **echo -e**.

Les nombres dans la table suivante fonctionnent pour un terminal *rxvt*. Les résultats peuvent varier pour d'autres émulateurs de terminaux.

Tableau 33.2. Nombres représentant les couleurs des séquences d'échappement

Couleur	Avant-plan	Arrière-plan
noir	30	40
rouge	31	41
vert	32	42
jaune	33	43
bleu	34	44
magenta	35	45
cyan	36	46
blanc	37	47

Exemple 33.13. Afficher du texte coloré

```
#!/bin/bash
# color-echo.sh : Affiche des messages texte en couleur.

# Modifier ce script pour vos besoins propres.
# C'est plus facile que de coder manuellement les couleurs.

noir='\E[30;47m'
rouge='\E[31;47m'
vert='\E[32;47m'
jaune='\E[33;47m'
bleu='\E[34;47m'
magenta='\E[35;47m'
cyan='\E[36;47m'
blanc='\E[37;47m'


alias init="tput sgr0"       #  Initialise les attributs texte à la normale
                             #+ sans effacer l'écran.


cecho ()                     # Echo couleur.
                             # Argument $1 = message
                             # Argument $2 = couleur
{
local msg_par_defaut="Pas de message."
                             # N'a pas réellement besoin d'être une variable
                             # locale.
```

```
message=${1:-$msg_par_defaut}# Message par défaut.
couleur=${2:-$noir}          # Noir par défaut si non spécifié.

  echo -e "$color"
  echo "$message"
  init                       # Retour à la normale.

  return
}


# Maintenant, essayons-le.
# ------------------------------------------------
cecho "Je me sens bleu..." $bleu
cecho "Le magenta ressemble plus à du violet." $magenta
cecho "Vert avec envie." $vert
cecho "Vous voyez rouge ?" $rouge
cecho "Cyan, mieux connu sous le nom d'aqua." $cyan
cecho "Pas de couleur précisée (noir par défaut)."
       # Argument $color manquant.
cecho "Couleur \"vide\" passée (noir par défaut)." ""
       # Argument $color vide.
cecho
       # Arguments $message et $color manquants.
cecho "" ""
       # Arguments $message et $color vides.
# ------------------------------------------------

echo

exit 0

# Exercices :
# ----------
# 1) Ajouter l'attribut "gras" à la fonction 'cecho ()'.
# 2) Ajouter des options pour des fonds colorés.
```

Exemple 33.14. Un jeu de « courses de chevaux »

```
#!/bin/bash
# horserace.sh : simulation très simple d'une course de chevaux.
# Auteur : Stefano Palmeri
# Utilisé avec sa permission.

##############################################################################
#  But du script :
#  jouer avec les séquences d'échappement et les couleurs du terminal.
#
#  Exercice :
#  Modifiez le script pour qu'il fonctionne de façon moins aléatoire,
#+ construisez un faux magasin de paris...
#  Hum... Hum... cela me rappelle un film...
#
#  Le script donne un handicap (au hasard) à chaque cheval.
#  Les chances sont calculées suivant le handicap du cheval
#+ et sont exprimées dans le style européen (?).
#  Par exemple : odds=3.75 signifie que si vous pariez 1$ et que vous gagnez,
#+ vous recevrez $3.75.
#
#  Le script a été testé avec un système d'exploitation GNU/Linux,
#+ en utilisant xterm, rxvt et konsole.
#  Sur une machine disposant d'un processeur AMD 900 MHz,
#+ le temps moyen d'une course est de 75 secondes...
#  Sur des ordinateurs plus rapides, le temps serait encore plus faible.
#  Donc, si vous voulez plus de suspens, réinitialisez la variable ARG_USLEEP.
```

```
#
#  Script de Stefano Palmeri.
################################################################################

E_ERREXEC=65

# Vérifie si md5sum et bc sont installés.
if ! which bc &> /dev/null; then
   echo "bc n'est pas installé."
   echo "Impossible de continuer..."
   exit $E_ERREXEC
fi
if ! which md5sum &> /dev/null; then
   echo "md5sum n'est pas installé."
   echo "Impossible de continuer..."
   exit $E_ERREXEC
fi

#  Configurez la variable suivante pour ralentir l'exécution du script.
#  Elle sera passée comme argument de usleep (man usleep)
#+ et est exprimée en microsecondes (500000 = une demi-seconde).
ARG_USLEEP=0

#  Nettoie le répertoire temporaire, restaure le curseur du terminal et
#+ ses couleurs - si le script a été interrompu par Ctl-C.
trap 'echo -en "\E[?25h"; echo -en "\E[0m"; stty echo;\
tput cup 20 0; rm -fr  $REP_TEMP_COURSE_CHEVAUX'  TERM EXIT
#  Voir le chapitre sur le débogage pour une explication de 'trap.'

#  Configure un nom unique (paranoïaque) pour le répertoire temporaire
#+ dont a besoin le script.
REP_TEMP_COURSE_CHEVAUX=$HOME/.horserace-`date +%s`-`head -c10 /dev/urandom \
| md5sum | head -c30`

# Crée le répertoire temporaire et s'y place.
mkdir $REP_TEMP_COURSE_CHEVAUX
cd $REP_TEMP_COURSE_CHEVAUX


#  Cette fonction déplace le curseur sur la ligne $1, colonne $2 puis affiche $3.
#  Par exemple :  "deplace_et_affiche 5 10 linux" est équivalent à
#+ "tput cup 4 9; echo linux", mais avec une seule commande au lieu de deux.
#  Note : "tput cup" définit 0 0 comme étant l'angle en haut à gauche du terminal,
#+ echo définit 1 1 comme étant l'angle en haut à gauche du terminal.
deplace_et_affiche() {
          echo -ne "\E[${1};${2}H""$3"
}

# Fonction pour générer un nombre pseudo-aléatoire entre 1 et 9.
hasard_1_9 ()
{
               head -c10 /dev/urandom | md5sum | tr -d [a-z] | tr -d 0 | cut -c1
}

#  Deux fonctions simulant un "mouvement" lors de l'affichage des chevaux.
dessine_cheval_un() {
              echo -n " "//$DEPLACE_CHEVAL//
}
dessine_cheval_deux(){
             echo -n " "\\\\$DEPLACE_CHEVAL\\\\
}


# Définit les dimensions actuelles du terminal.
N_COLS=`tput cols`
N_LIGNES=`tput lines`

# A besoin d'un terminal avec au moins 20 lignes et 80 colonnes. Vérifiez-le.
if [ $N_COLS -lt 80 ] || [ $N_LIGNES -lt 20 ]; then
```

```
   echo "`basename $0` a besoin d'un terminal à 80 colonnes et 20 lignes."
   echo "Votre terminal fait ${N_COLS} colonnes sur ${N_LIGNES} lignes."
   exit $E_ERREXEC
fi


# Commence par le dessin du champ de course.

# A besoin d'une chaîne de 80 caractères. Voir ci-dessous.
ESPACES80=`seq -s "" 100 | head -c80`

clear

# Configure les couleurs en avant et en arrière-plan à blanc.
echo -ne '\E[37;47m'

# Déplace le curseur à l'angle en haut à gauche du terminal.
tput cup 0 0

# Dessine six lignes blanches.
for n in `seq 5`; do
      echo $ESPACES80        # Utilise les 80 caractères pour coloriser le terminal.
done

# Configure la couleur en avant-plan à noir.
echo -ne '\E[30m'

deplace_et_affiche 3 1 "START  1"
deplace_et_affiche 3 75 FINISH
deplace_et_affiche 1 5 "|"
deplace_et_affiche 1 80 "|"
deplace_et_affiche 2 5 "|"
deplace_et_affiche 2 80 "|"
deplace_et_affiche 4 5 "|  2"
deplace_et_affiche 4 80 "|"
deplace_et_affiche 5 5 "V  3"
deplace_et_affiche 5 80 "V"

# Configure la couleur en avant-plan à rouge.
echo -ne '\E[31m'

# Un peu d'art ASCII.
deplace_et_affiche 1 8 "..@@@..@@@@@...@@@@@.@...@..@@@@..."
deplace_et_affiche 2 8 ".@...@...@.......@...@...@.@......."
deplace_et_affiche 3 8 ".@@@@@...@.......@...@@@@@.@@@@...."
deplace_et_affiche 4 8 ".@...@...@.......@...@...@.@......."
deplace_et_affiche 5 8 ".@...@...@.......@...@...@..@@@@..."
deplace_et_affiche 1 43 "@@@@...@@@...@@@@..@@@@..@@@@."
deplace_et_affiche 2 43 "@...@.@...@.@.....@.....@....."
deplace_et_affiche 3 43 "@@@@..@@@@@.@.....@@@@...@@@.."
deplace_et_affiche 4 43 "@..@..@...@.@.....@.........@."
deplace_et_affiche 5 43 "@...@.@...@..@@@@..@@@@.@@@@.."


# Configure la couleur en avant-plan et en arrière-plan à vert.
echo -ne '\E[32;42m'

# Dessine onze lignes vertes.
tput cup 5 0
for n in `seq 11`; do
      echo $ESPACES80
done

# Configure la couleur en avant-plan à noir.
echo -ne '\E[30m'
tput cup 5 0

# Dessine les limites.
echo "++++++++++++++++++++++++++++++++++++\
```

```
++++++++++++++++++++++++++++++++++++++++"

tput cup 15 0
echo "++++++++++++++++++++++++++++++++++++\
++++++++++++++++++++++++++++++++++++++++"

# Configure la couleur en avant et en arrière-plan à blanc.
echo -ne '\E[37;47m'

# Dessine trois lignes blanches.
for n in `seq 3`; do
      echo $ESPACES80
done

# Configure la couleur en avant-plan à noir.
echo -ne '\E[30m'

# Crée neuf fichiers pour stocker les handicaps.
for n in `seq 10 7 68`; do
      touch $n
done

# Configure le premier type de "cheval" que le script dessinera.
TYPE_CHEVAL=2

#  Crée un fichier position et un fichier chance pour chaque "cheval".
#+ Dans ces fichiers, stocke la position actuelle du cheval,
#+ le type et les chances.
for HN in `seq 9`; do
      touch position_${HN}_cheval
      touch chances_${HN}
      echo \-1 > position_${HN}_cheval
      echo $TYPE_CHEVAL >>  position_${HN}_cheval
      # Définit un handicap au hasard pour un cheval.
       HANDICAP=`hasard_1_9`
      # Vérifie si la fonction hasard_1_9 a renvoyé une bonne valeur.
      while ! echo $HANDICAP | grep [1-9] &> /dev/null; do
                HANDICAP=`hasard_1_9`
      done
      # Définit la dernière position du handicap pour le cheval.
      LHP=`expr $HANDICAP \* 7 + 3`
      for FILE in `seq 10 7 $LHP`; do
            echo $HN >> $FILE
      done

      # Calcule les chances.
      case $HANDICAP in
              1) CHANCES=`echo $HANDICAP \* 0.25 + 1.25 | bc`
                 echo $CHANCES > chances_${HN}
              ;;
              2 | 3) CHANCES=`echo $HANDICAP \* 0.40 + 1.25 | bc`
                 echo $CHANCES > chances_${HN}
              ;;
              4 | 5 | 6) CHANCES=`echo $HANDICAP \* 0.55 + 1.25 | bc`
                 echo $CHANCES > chances_${HN}
              ;;
              7 | 8) CHANCES=`echo $HANDICAP \* 0.75 + 1.25 | bc`
                 echo $CHANCES > chances_${HN}
              ;;
              9) CHANCES=`echo $HANDICAP \* 0.90 + 1.25 | bc`
                 echo $CHANCES > chances_${HN}
      esac


done


# Affiche les chances.
affiche_chances() {
```

```
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
      echo "#$HN odds->" `cat chances_${HN}`
done
}

# Dessine les chevaux sur la ligne de départ.
dessine_chevaux() {
tput cup 6 0
echo -ne '\E[30;42m'
for HN in `seq 9`; do
      echo /\\$HN/\\"                                    "
done
}

affiche_chances

echo -ne '\E[47m'
# Attend l'appui sur la touche Enter pour commencer la course.
# La séquence d'échappement '\E[?25l' désactive le curseur.
tput cup 17 0
echo -e '\E[?25l'Appuyez sur la touche [enter] pour lancer la course...
read -s

#  Désactive l'affichage normal sur le terminal.
#  Ceci évite qu'une touche appuyée "contamine" l'écran pendant la course...
stty -echo

# ----------------------------------------------------------
# Début de la course.

dessine_chevaux
echo -ne '\E[37;47m'
deplace_et_affiche 18 1 $ESPACES80
echo -ne '\E[30m'
deplace_et_affiche 18 1 Starting...
sleep 1

# Configure la colonne de la ligne finale.
POS_GAGNANTE=74

# Définit le moment où la course a commencé.
HEURE_DEBUT=`date +%s`

# Variable COL nécessaire pour la construction "while".
COL=0

while [ $COL -lt $POS_GAGNANTE ]; do

          DEPLACE_CHEVAL=0

          # Vérifie si la fonction hasard_1_9 a renvoyé une bonne valeur.
          while ! echo $DEPLACE_CHEVAL | grep [1-9] &> /dev/null; do
                DEPLACE_CHEVAL=`hasard_1_9`
          done

          # Définit l'ancien type et position du "cheval au hasard".
          TYPE_CHEVAL=`cat  position_${DEPLACE_CHEVAL}_cheval | tail -n 1`
          COL=$(expr `cat  position_${DEPLACE_CHEVAL}_cheval | head -n 1`)

          ADD_POS=1
          # Vérifie si la position actuelle est une position de handicap.
          if seq 10 7 68 | grep -w $COL &> /dev/null; then
                if grep -w $DEPLACE_CHEVAL $COL &> /dev/null; then
                      ADD_POS=0
                      grep -v -w  $DEPLACE_CHEVAL $COL > ${COL}_new
                      rm -f $COL
                      mv -f ${COL}_new $COL
```

```
                    else ADD_POS=1
              fi
          else ADD_POS=1
          fi
          COL=`expr $COL + $ADD_POS`
          echo $COL >  position_${DEPLACE_CHEVAL}_cheval  # Stocke la nouvelle
position.

          # Choisit le type de cheval à dessiner.
          case $TYPE_CHEVAL in
              1) TYPE_CHEVAL=2; DRAW_HORSE=dessine_cheval_deux
              ;;
              2) TYPE_CHEVAL=1; DRAW_HORSE=dessine_cheval_un
          esac
          echo $TYPE_CHEVAL >>  position_${DEPLACE_CHEVAL}_cheval
          # Stocke le type actuel.

          # Configure l'avant et l'arrière-plan à vert.
          echo -ne '\E[30;42m'

          # Déplace le curseur à la nouvelle position du cheval.
          tput cup `expr $DEPLACE_CHEVAL + 5` \
          `cat  position_${DEPLACE_CHEVAL}_cheval | head -n 1`

          # Dessine le cheval.
          $DRAW_HORSE
          usleep $ARG_USLEEP

          #  Quand tous les chevaux ont passé la ligne du champ 15,
          #+ affiche de nouveau les chances.
          touch champ15
          if [ $COL = 15 ]; then
            echo $DEPLACE_CHEVAL >> champ15
          fi
          if [ `wc -l champ15 | cut -f1 -d " "` = 9 ]; then
              affiche_chances
              : > champ15
          fi

          # Définit le cheval en tête.
          MEILLEURE_POS=`cat *position | sort -n | tail -1`

          # Configure la couleur de l'arrière-plan à blanc.
          echo -ne '\E[47m'
          tput cup 17 0
          echo -n Current leader: `grep -w $MEILLEURE_POS *position | cut -c7`\
          "                              "

done

# Définit le moment où la course s'est terminée.
HEURE_FIN=`date +%s`

# Configure la couleur de l'arrière blanc à vert et active le clignotement du texte.
echo -ne '\E[30;42m'
echo -en '\E[5m'

# Fait en sorte que le gagnant clignotte.
tput cup `expr $DEPLACE_CHEVAL + 5` \
`cat  position_${DEPLACE_CHEVAL}_cheval | head -n 1`
$DESSINE_CHEVAL

# Désactive le clignotement du texte.
echo -en '\E[25m'

# Configure la couleur d'avant et d'arrière-plan à blanc.
echo -ne '\E[37;47m'
deplace_et_affiche 18 1 $ESPACES80
```

```
# Configure la couleur d'avant-plan à noir.
echo -ne '\E[30m'

# Fait que le gagnant clignotte.
tput cup 17 0
echo -e "\E[5mWINNER: $DEPLACE_CHEVAL\E[25m""  Odds: `cat odds_${DEPLACE_CHEVAL}`"\
"  Race time: `expr $HEURE_FIN - $HEURE_DEBUT` secs"

# Restaure le curseur et les anciennes couleurs.
echo -en "\E[?25h"
echo -en "\E[0m"

# Restaure l'affiche normal.
stty echo

# Supprime le répertoire temporaire de la course.
rm -rf $REP_TEMP_COURSE_CHEVAUX

tput cup 19 0

exit 0
```

Voir aussi l'Exemple A.23, « Coloriser du texte en utilisant les fonctions de hachage », Exemple A.42, « Un outil de résolution général » et Exemple A.40, « Pétales autour d'une rose ».

Attention

Il existe néanmoins un problème majeur avec tout ceci. *Les séquences d'échappement ANSI sont généralement non portables.* Ce qui fonctionne bien sur certains émulateurs de terminaux (ou la console) peut fonctionner différemment, ou pas du tout, sur les autres. Un script « coloré » ayant une excellente forme sur la machine de l'auteur du script peut produire une sortie illisible chez quelqu'un d'autre. Ceci compromet grandement l'utilité de la « colorisation » des scripts, et relègue cette technique au statut de gadget, voire de « jeu ».

L'utilitaire **color** de Moshe Jacobson (*http://runslinux.net/projects.html#color*) simplifie considérablement l'utilisation des séquences d'échappement ANSI. Il substitue une syntaxe claire et logique aux constructions bizarres dont on a discutées.

Henry/teikedvl a créé un outil (*http://scriptechocolor.sourceforge.net/*) pour simplifier la création de scripts colorisés.

33.7. Optimisations

La plupart des scripts shell sont des solutions rapides et sales pour des problèmes non complexes. Du coup, les optimiser en rapidité n'est pas vraiment un problème. Considérez le cas où un script réalise une tâche importante, le fait bien mais fonctionne trop lentement. Le réécrire avec un langage compilé peut ne pas être une option très agréable. La solution la plus simple serait de réécrire les parties du script qui le ralentissent. Est-il possible d'appliquer les principes de l'optimisation de code même à un script lent ?

Vérifiez les boucles dans le script. Le temps consommé par des opérations répétitives s'ajoute rapidement. Si c'est possible, supprimez les opérations consommatrices de temps des boucles.

Utilisez les commandes internes159 plutôt que les commandes système. Ces commandes intégrées s'exécutent plus rapidement et ne lancent habituellement pas un sous-shell lors de leur appel.

Évitez les commandes non nécessaires, particulièrement dans un tuyau.

```
cat "$fichier" | grep "$mot"

grep "$mot" "$fichier"

#  Les lignes de commandes ci-dessus ont un effet identique, mais le deuxième
#+ tourne plus vite comme il est lancé sur moins de processus.
```

La commande cat semble particulièrement sujette à une sur-utilisation dans les scripts.

Utilisez les outils time et times pour vérifier les commandes particulièrement intensives. Considérez la réécriture des sections critiques en code C, voire en assembleur.

Essayez de minimiser les entrées/sorties fichier. Bash n'est pas particulièrement efficace sur la gestion des fichiers, donc considérez l'utilisation d'outils plus appropriés pour ceci dans le script, tels que awk634 ou Perl437.

Écrivez vos scripts d'une façon modulaire et cohérente [4] , ainsi ils peuvent être réorganisés et sécurisés selon les besoins. Quelques unes des techniques d'optimisation applicables aux langages de haut niveau peuvent fonctionner pour des scripts mais d'autres, tels que le *déroulement de boucles*, sont pratiquement impossibles. Par dessus tout, utilisez votre bon sens.

Pour une excellente démonstration du fait qu'une optimisation réduit dramatiquement le temps d'exécution d'un script, voir l'Exemple 15.46, « Paiement mensuel sur une hypothèque ».

33.8. Astuces assorties

33.8.1. Idées pour des scripts plus puissants

- Vous avez un problème que vous voulez résoudre en écrivant un script Bash. Malheureusement, vous ne savez pas comment vous lancer. Une méthode est de commencer directement en codant les parties du script qui viennent facliement et d'écrire les parties difficiles en *pseudo-code*.

```
#!/bin/bash

NBARG=1                     # A besoin d'un nom comme argument.
E_MAUVAISARGS=65

if [ nombre-d-arguments différent-de "$NBARG" ]
#    ^^^^^^^^^^^^^^^^^^ ^^^^^^^^^^^^
#  Impossible de trouver comment coder ceci
#+ ... donc écrivez-le en pseudo-code.

then
  echo "Usage: nom-du-script nom"
  #            ^^^^^^^^^^^^^     Encore du pseudo-code.
  exit $E_MAUVAISARGS
fi

. . .

exit 0


# Plus tard, remplacez le pseudo-code par du code fonctionnel.

# La ligne 6 devient :
if [ $# -ne "$NBARG" ]

# La ligne 12 devient :
  echo "Usage: `basename $0` nom"
```

Pour un exemple d'utilisation de pseudo-code, voir l'exercice Square Root.

- Pour conserver un enregistrement des scripts utilisateur lancés lors de certaines sessions ou lors d'un certain nombre de sessions, ajoutez les lignes suivantes à chaque script dont vous voulez garder la trace. Ceci va conserver un fichier d'enregistrement des noms de script et des heures d'appel.

```
# Ajoute (>>) ce qui suit à la fin de chaque script tracé.

whoami>> $FICHIER_SAUVEGARDE    # Utilisateur appelant le script.
echo $0>> $FICHIER_SAUVEGARDE   # Nom du script.
date>> $FICHIER_SAUVEGARDE      # Date et heure.
echo>> $FICHIER_SAUVEGARDE      # Ligne blanche comme séparateur.

#  Bien sûr, FICHIER_SAUVEGARDE défini et exporté comme variable d'environnement
#+ dans ~/.bashrc (quelque chose comme ~/.scripts-run)
```

[4] Cela signifie habituellement une utilisation libérale des fonctions352.

- L'opérateur >> *ajoute* des lignes dans un fichier. Qu'en est-il si vous voulez *ajouter* une ligne *au début* d'un fichier existant, c'est-à-dire la coller au tout début ?

```
fichier=donnees.txt
titre="***Ceci est la ligne de titre des fichiers texte de données***"

echo $titre | cat - $fichier >$fichier.new
# "cat -" concatène stdout dans $fichier.
#  Le résultat final est l'écriture d'un nouveau fichier avec $titre ajouté au
#+ *début*.
```

  C'est une variante simplifiée du script de l'Exemple 18.13, « Ajouter une ligne au début d'un fichier » donnée plus tôt. Bien sûr, sed631 peut aussi le faire.

- Un script shell peut agir comme une commande interne à l'intérieur d'un autre script shell, d'un script *Tcl* ou d'un script *wish*, voire même d'un Makefile244. Il peut être appelé comme une commande shell externe dans un programme C en utilisant l'appel `system()`, c'est-à-dire `system("nom_du_script");`.

- Configurer une variable avec le contenu d'un script *sed* ou *awk* embarqué accroît la lisibilité de l'emballage shell434 qui l'entoure. Voir l'Exemple A.1, « *mailformat* : Formater un courrier électronique » et l'Exemple 14.20, « Utiliser *export* pour passer une variable à un script *awk* embarqué ».

- Réunissez les fichiers contenant vos définitions et vos fonctions les plus utiles. Quand nécessaire, « incluez » un ou plus de ces « fichiers bibliothèque » dans des scripts avec soit le point (**.**) soit la commande source.

```
# BIBLIOTHEQUE SCRIPT
# ------ -------

# Note :
# Pas de "#!" ici.
# Pas de code exécuté immédiatement non plus.


# Définition de variables ici

ROOT_UID=0             # Root a l'identifiant utilisateur ($UID) 0.
E_NOTROOT=101          # Pas d'erreur de l'utilisateur root.
MAXRETVAL=255          # Code de retour (positif) maximum d'une fonction.
SUCCESS=0
FAILURE=-1



# Fonctions

Usage ()               # Message "Usage :".
{
  if [ -z "$1" ]       # Pas d'argument passé.
  then
    msg=nom_du_fichier
  else
    msg=$@
  fi

  echo "Usage: `basename $0` "$msg""
}


Verifier_si_root ()       # Vérifier si le script tourne en tant que root.
{                         # À partir de l'exemple "ex39.sh".
  if [ "$UID" -ne "$ROOT_UID" ]
  then
    echo "Doit être root pour lancer ce script."
    exit $E_NOTROOT
  fi
}
```

```
Creer_Nom_Fichier_Temporaire ()  # Crée un nom de fichier temporaire "unique".
{                                # À partir de l'exemple "ex51.sh".
  prefixe=temp
  suffixe=`eval date +%s`
  Tempfilename=$prefixe.$suffixe
}


est_alpha2 ()           # Teste si la chaine de caractères *entière* est
                        # alphabétique.
{                       # À partir de l'exemple "isalpha.sh".
  [ $# -eq 1 ] || return $FAILURE

  case $1 in
  *[!a-zA-Z]*|"") return $FAILURE;;
  *) return $SUCCESS;;
  esac                  # Merci, S.C.
}


abs ()                          # Valeur absolue.
{                               # Attention : Valeur de retour maximum = 255.
  E_ARGERR=-999999

  if [ -z "$1" ]                # Il est nécessaire de passer un argument.
  then
    return $E_ARGERR            # Code d'erreur évident renvoyé.
  fi

  if [ "$1" -ge 0 ]             # Si non-négatif,
  then                          #
    absval=$1                   # reste tel quel,
  else                          # Sinon,
    let "absval = (( 0 - $1 ))"  # change son signe.
  fi

  return $absval
}


tolower ()             #  Convertit le(s) chaîne(s) de caractères passées comme
{                      #+ argument(s) en minuscule.

  if [ -z "$1" ]       #  Si aucun argument n'est passé,
  then                 #+ envoyez un message d'erreur
    echo "(null)"      #+ (message d'erreur étant un pointeur null style C)
    return             #+ et sort de la fonction.
  fi

  echo "$@" | tr A-Z a-z
  # Transforme tous les arguments passés ($@).

  return

#  Utilisez la substituion de commande pour initialiser une variable à la sortie
#+ d'une commande.
#  Par exemple :
#    anciennevar="Un EnseMBle dE LetTres miNusCuleS Et MaJuscuLeS"
#    nouvellevar=`tolower "$anciennevar"`
#    echo "$nouvellevar"    # un ensemble de lettre minuscules et majuscules
#
#  Exercice : Réécrire cette fonction pour changer le(s) argument(s) minuscule(s)
#+            en majuscules ... toupper() [facile].
}
```

- Utiliser des en-têtes de commentaires pour accroître la clarté et la compréhension des scripts.

```
## Attention.
```

```
rm -rf *.zzy   ##  Les options "-rf" de "rm" sont très dangereux,
               ##+ spécialement avec des caractères joker.

#+ Suite de la ligne.
#  Ceci est la ligne 1
#+ d'un commentaire multi-ligne.
#+ et ceci est la ligne finale.

#* Note.

#o Elément d'une liste.

#> Autre point de vue.
while [ "$var1" != "end" ]    #> while test "$var1" != "end"
```

- Une utilisation particulièrement intelligente des constructions if-test permet de mettre en commentaires des blocs de code.

```
#!/bin/bash

BLOC_COMMENTAIRE=
#  Essayez d'initialiser la variable ci-dessus autrement pour une
#+ surprise peu plaisante.

if [ $BLOC_COMMENTAIRE ]; then

Bloc de commentaires --
================================
Ceci est une ligne de commentaires.
Ceci est une autre ligne de commentaires.
Ceci est encore une autre ligne de commentaires.
================================

echo "Ceci ne s'affichera pas."

Les blocs de commentaires sont sans erreur ! Youpi !

fi

echo "Sans commentaires, merci."

exit 0
```

  Comparez ceci avec l'utilisation de documents en lignes pour commenter des blocs de code.

- En utilisant la variable d'état de sortie $?, un script peut tester si un paramètre contient seulement des chiffres, ainsi il peut être traité comme un entier.

```
#!/bin/bash

SUCCESS=0
E_BADINPUT=65

test "$1" -ne 0 -o "$1" -eq 0 2>/dev/null
# Un entier est soit égal à 0 soit différent de 0.
# 2>/dev/null supprime les messages d'erreur.

if [ $? -ne "$SUCCESS" ]
then
  echo "Usage: `basename $0` integer-input"
  exit $E_BADINPUT
fi

let "sum = $1 + 25"            # Donnera une erreur si $1 n'est pas un entier.
echo "Sum = $sum"

#  Toute variable, pas simplement un paramètre de ligne de commande, peut être
#+ testé de cette façon.
```

```
exit 0
```

- L'échelle 0 - 255 des valeurs de retour des fonctions est une limitation importante. Les variables globales et autres moyens de contourner ce problème sont souvent des problèmes eux-même. Une autre méthode, pour que la fonction communique une valeur de retour au corps principal du script, est que la fonction écrive sur `stdout` la « valeur de sortie » (habituellement avec un echo) et de l'affecter à une variable. C'est une variante de la substitution de commandes141.

Exemple 33.15. Astuce de valeur de retour

```
#!/bin/bash
# multiplication.sh

multiplie ()                    # Multiplie les paramètres passés.
{                               # Acceptera un nombre variable d'arguments.

  local produit=1

  until [ -z "$1" ]             # Jusqu'à la fin de tous les arguments...
  do
    let "produit *= $1"
    shift
  done

  echo $produit                 #  N'affichera pas sur stdout
}                               #+ car cela va être affecté à une variable.

mult1=15383; mult2=25211
val1=`multiplie $mult1 $mult2`
echo "$mult1 X $mult2 = $val1"
                                # 387820813

mult1=25; mult2=5; mult3=20
val2=`multiplie $mult1 $mult2 $mult3`
echo "$mult1 X $mult2 X $mult3 = $val2"
                                # 2500

mult1=188; mult2=37; mult3=25; mult4=47
val3=`multiplie $mult1 $mult2 $mult3 $mult4`
echo "$mult1 X $mult2 X $mult3 X $mult4 = $val3"
                                # 8173300

exit 0
```

La même technique fonctionne aussi pour les chaînes de caractères alphanumériques. Ceci signifie qu'une fonction peut « renvoyer » une valeur non-numérique.

```
capitaliser_ichar ()                #  Capitaliser le premier caractère
{                                   #+ de(s) chaîne(s) de caractères passées.

  chaine0="$@"                      # Accepte plusieurs arguments.

  premiercaractere=${chaine0:0:1}   # Premier caractère.
  chaine1=${chaine0:1}              # Reste de(s) chaîne(s) de caractères.

  PremierCaractere=`echo "$premiercaractere" | tr a-z A-Z`
                                    # Capitalise le premier caractère.

  echo "$PremierCaractere$chaine1"  # Sortie vers stdout.

}

nouvellechaine=`capitalize_ichar "toute phrase doit commencer avec une lettre
majuscule."`
echo "$nouvellechaine"          # Toute phrase doit commencer avec une lettre
majuscule.
```

Il est même possible pour une fonction de « renvoyer » plusieurs valeurs avec cette méthode.

Exemple 33.16. Une astuce permettant de renvoyer plus d'une valeur de retour

```
#!/bin/bash
# sum-product.sh
# Une fonction peut "renvoyer" plus d'une valeur.

somme_et_produit ()   # Calcule à la fois la somme et le produit des arguments.
{
  echo $(( $1 + $2 )) $(( $1 * $2 ))
# Envoie sur stdout chaque valeur calculée, séparée par une espace.
}

echo
echo "Entrez le premier nombre "
read premier

echo
echo "Entrez le deuxième nombre "
read second
echo

valretour=`somme_et_produit $premier $second` #  Affecte à la variable la sortie
                                              #+ de la fonction.
somme=`echo "$valretour" | awk '{print $1}'`  # Affecte le premier champ.
produit=`echo "$valretour" | awk '{print $2}'`# Affecte le deuxième champ.

echo "$premier + $second = $somme"
echo "$premier * $second = $produit"
echo

exit 0
```

Attention

Il ne peut y avoir qu' **une seule** instruction *echo* dans le fonction pour que cela fonctionne. Si vous modifiez l'exemple précédent :

```
somme_et_produit ()
{
  echo "Ceci est la fonction somme_et_produit." # Ceci casse tout !
  echo $(( $1 + $2 )) $(( $1 * $2 ))
}
...
retval=`somme_et_produit $first $second`      # Affecte la sortie de la
fonction.
# Maintenant, cela ne fonctionnera plus correctement.
```

- Ensuite dans notre liste d'astuces se trouvent les techniques permettant de passer un tableau375 à une fonction352, « renvoyant » alors un tableau en retour à la fonction principale du script.

  Le passage d'un tableau nécessite de charger des éléments séparés par une espace d'un tableau dans une variable avec la substitution de commandes141. Récupérer un tableau comme « valeur de retour » à partir d'une fonction utilise le stratagème mentionné précédemment de la sortie (echo) du tableau dans la fonction, puis d'invoquer la substitution de commande et l'opérateur **(...)** pour l'assigner dans un tableau.

Exemple 33.17. Passer et renvoyer un tableau

```
#!/bin/bash
# array-function.sh : Passer un tableau à une fonction et...
#                     "renvoyer" un tableau à partir d'une fonction
```

```
Passe_Tableau ()
{
  local tableau_passe   # Variable locale.
  tableau_passe=( `echo "$1"` )
  echo "${tableau_passe[@]}"
  #  Liste tous les éléments du nouveau tableau déclaré
  #+ et initialisé dans la fonction.
}


tableau_original=( element1 element2 element3 element4 element5 )

echo
echo "tableau_original = ${tableau_original[@]}"
#                       Liste tous les éléments du tableau original.


# Voici une astuce qui permet de passer un tableau à une fonction.
# *********************************
argument=`echo ${tableau_original[@]}`
# *********************************
#  Emballer une variable
#+ avec tous les éléments du tableau original séparés avec une espace.
#
# Notez que d'essayer de passer un tableau en lui-même ne fonctionnera pas.


# Voici une astuce qui permet de récupérer un tableau comme "valeur de retour".
# ************************************
tableau_renvoye=( `Passe_Tableau "$argument"` )
# ************************************
# Affecte une sortie de la fonction à une variable de type tableau.

echo "tableau_renvoye = ${tableau_renvoye[@]}"

echo "============================================================="

#  Maintenant, essayez encore d'accèder au tableau en dehors de la
#+ fonction.
Passe_Tableau "$argument"

# La fonction liste elle-même le tableau, mais...
#+ accèder au tableau de l'extérieur de la fonction est interdit.
echo "Tableau passé (de l'intérieur de la fonction) = ${tableau_passe[@]}"
# Valeur NULL comme il s'agit d'une variable locale.

echo

exit 0
```

Pour un exemple plus élaboré du passage d'un tableau dans les fonctions, voir l'Exemple A.10, « « *life* : Jeu de la Vie » ».

- En utilisant la construction en double parenthèses, il est possible d'utiliser la syntaxe style C pour initialiser et incrémenter des variables ainsi que dans des boucles for et while. Voir l'Exemple 10.12, « Une boucle *for* à la C » et l'Exemple 10.17, « Syntaxe à la C pour une boucle *while* ».
- Initialiser path et umask au début d'un script le rend plus « portable » -- il est plus probable qu'il fonctionne avec des machines « étrangères » dont l'utilisateur a pu modifier `$PATH` et **umask**.

```
#!/bin/bash
PATH=/bin:/usr/bin:/usr/local/bin ; export PATH
umask 022   # Les fichiers que le script crée auront les droits 755.

# Merci à Ian D. Allen pour ce conseil.
```

- Une technique de scripts utiles est d'envoyer *de manière répétée* la sortie d'un filtre (par un tuyau) vers le *même filtre*, mais avec un ensemble différent d'arguments et/ou options. Ceci est spécialement intéressant pour tr et grep.

```
# De l'exemple "wstrings.sh".

wlist=`strings "$1" | tr A-Z a-z | tr '[:space:]' Z | \
tr -cs '[:alpha:]' Z | tr -s '\173-\377' Z | tr Z ' '`
```

Exemple 33.18. Un peu de fun avec des anagrammes

```
#!/bin/bash
# agram.sh: Jouer avec des anagrammes.

# Trouver les anagrammes de...
LETTRES=etaoinshrdlu
FILTRE='.......'       # Combien de lettres au minimum ?
#       1234567

anagram "$LETTRES" |   # Trouver tous les anagrammes de cet ensemble de lettres...
grep '$FILTRE' |       # Avec au moins sept lettres,
grep '^is' |           # commençant par 'is',
grep -v 's$' |         # sans les puriels,
grep -v 'ed$'          # sans verbe au passé ("ed" en anglais)
#  Il est possible d'ajouter beaucoup de combinaisons
#+ dans les conditions et les filtres.

#  Utilise l'utilitaire "anagram"
#+ qui fait partie du paquetage de liste de mots "yawl" de l'auteur.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz

exit 0                 # Fin du code.

bash$ sh agram.sh
islander
isolate
isolead
isotheral



#  Exercices :
#  ----------
#  Modifiez ce script pour configurer LETTRES via la ligne de commande.
#  Transformez les filtres en paramètres dans les lignes 11 à 13
#+ (comme ce qui a été fait pour $FILTRE),
#+ de façon à ce qu'ils puissent être indiqués en passant les arguments
#+ à une fonction.

#  Pour une approche légèrement différente de la construction d'anagrammes,
#+ voir le script agram2.sh.
```

Voir aussi l'Exemple 27.3, « État de la connexion », l'Exemple 15.25, « Générer des énigmes « Crypto-Citations » » et l'Exemple A.9, « *soundex* : Conversion phonétique ».

- Utiliser des « documents anonymes324 » pour mettre en commentaire des blocs de code, pour ne pas avoir à mettre en commentaire chaque ligne avec un #. Voir Exemple 18.11, « Décommenter un bloc de code ».
- Lancer sur une machine un script dépendant de la présence d'une commande qui peut être absente est dangereux. Utilisez whatis pour éviter des problèmes potentiels avec ceci.

```
CMD=commande1                # Premier choix.
PlanB=commande2              # Option en cas de problème.

commande_test=$(whatis "$CMD" | grep 'nothing appropriate')
#  Si 'commande1' n'est pas trouvé sur ce système, 'whatis' renverra
#+ "commande1: nothing appropriate."
#
```

```
#  Une alternative plus saine est :
#     commande_test=$(whereis "$CMD" | grep \/)
#  Mais, du coup, le sens du test suivant devrait être inversé
#+ car la variable $commande_test détient le contenu si et seulement si
#+ $CMD existe sur le système.
#     (Merci bojster.)

if [[ -z "$command_test" ]]  # Vérifie si la commande est présente.
then
  $CMD option1 option2       #  Lancez commande1 avec ses options.
else                         #  Sinon,
  $PlanB                     #+ lancez commande2.
fi
```

- Un test if-grep pourrait ne pas renvoyer les résultats attendus dans un cas d'erreur lorsque le texte est affiché sur `stderr` plutôt que sur `stdout`.

```
if ls -l fichier_inexistant | grep -q 'No such file or directory'
  then echo "Le fichier \"fichier_inexistant\" n'existe pas."
fi
```

  Rediriger `stderr` sur `stdout` corrige ceci.

```
if ls -l fichier_inexistant 2>&1 | grep -q 'No such file or directory'
#                              ^^^^
  then echo "Le \"fichier_inexistant\" n'existe pas."
fi

# Merci à Chris Martin de nous l'avoir indiqué.
```

- Si vous devez vraiment accéder à une variable d'un sous-shell en dehors de ce sous-shell, voici une façon de le faire.

```
FICHIERTEMP=fichiertemp          # Crée un fichier temporaire pour stocker la
variable.

(   # À l'intérieur du sous-shell...
variable_interne=interne
echo $variable_interne
echo $variable_interne >>$FICHIERTEMP  # Ajout dans le fichier temporaire.
)

    # En dehors du sous-shell...

echo; echo "-----"; echo
echo $variable_interne             # Null, comme on s'y attendait.
echo "-----"; echo

# Maintenant...
read variable_interne <$FICHIERTEMP # Lecture de variable shell.
rm -f "$FICHIERTEMP"                # Supprime le fichier temporaire.
echo "$variable_interne"            # C'est un hack assez sale mais fonctionnel.
```

- La commande run-parts est utile pour exécuter un ensemble de scripts dans l'ordre, particulièrement en combinaison avec cron ou at.

- Pour avoir plusieurs révisions d'un script complexe, utilisez le package de contrôle de version appelé *rcs*.

  Parmi les avantages de cet outil, il modifie automatiquement les balises en en-tête. La commande **co** dans *rcs* fait un remplacement de paramètres pour certains mots clés réservés, par exemple remplacer *`#$Id: abs-part5.xml,v 1.8 2008-06-12 17:11:55 gleu Exp $`* dans un script avec quelque-chose comme :

```
#$Id: abs-part5.xml,v 1.8 2008-06-12 17:11:55 gleu Exp $
```

33.8.2. Widgets

- Il serait bien d'être capable d'invoquer les objets X-Windows à partir d'un script shell. Il existe plusieurs paquets qui disent le faire, à savoir *Xscript*, *Xmenu* et *widtools*. Les deux premiers ne semblent plus maintenus. Heureusement, il est toujours pos-

sible d'obtenir *widtools ici*.

Attention

Le paquet *widtools* (widget tools, outils pour objets) nécessite que la bibliothèque *XForms* soit installée. De plus, le Makefile244 a besoin d'être édité de façon judicieuse avant que le paquet ne soit construit sur un système Linux typique. Finalement, trois des six objets offerts ne fonctionnent pas (en fait, ils génèrent un défaut de segmentation).

La famille d'outils *dialog* offre une méthode d'appel des widgets « dialog » à partir d'un script shell. L'utilitaire original *dialog* fonctionne dans une console texte mais ses successeurs, *gdialog*, *Xdialog* et *kdialog* utilisent des ensembles de widgets basés sur X-Windows.

Exemple 33.19. Widgets appelés à partir d'un script shell

```
#!/bin/bash
# dialog.sh : Utiliser les composants graphiques de 'gdialog'.

#  Vous devez avoir installé 'gdialog' sur votre système pour lancer ce script.
#  Ou vous pouvez remplacer toutes les instances de 'gdialog' ci-dessous avec
#+ 'kdialog' ...
# Version 1.1 (corrigée le 04/05/05).

# Ce script s'inspire de l'article suivant.
#     "Scripting for X Productivity" de Marco Fioretti,
#      LINUX JOURNAL, numéro 113, septembre 2003, pp. 86-9.
# Merci à toutes ces braves âmes chez LJ.


# Erreur d'entrée dans la boîte de saisie.
E_ENTREE=65
# Dimensions de l'affichage des composants graphiques de saisie.
HAUTEUR=50
LARGEUR=60

# Nom du fichier de sortie (construit à partir du nom du script).
FICHIER_SORTIE=$0.sortie

# Affiche ce script dans un composant texte.
gdialog --title "Affichage : $0" --textbox $0 $HAUTEUR $LARGEUR



# Maintenant, nous allons essayer de sauvegarder l'entrée dans un fichier.
echo -n "VARIABLE=" > $FICHIER_SORTIE
gdialog --title "Entrée utilisateur" \
        --inputbox "Entrez une variable, s'il-vous-plaît :" \
$HAUTEUR $LARGEUR 2>> $FICHIER_SORTIE


if [ "$?" -eq 0 ]
# Une bonne pratique consiste à vérifier le code de sortie.
then
  echo "Exécution de \"dialog box\" sans erreurs."
else
  echo "Erreur(s) lors de l'exécution de \"dialog box\"."
        # Ou clic sur "Annuler" au lieu du bouton "OK".
  rm $FICHIER_SORTIE
  exit $E_ENTREE
fi


# Maintenant, nous allons retrouver et afficher la variable sauvée.
. $FICHIER_SORTIE   # 'Source'r le fichier sauvé.
echo "La variable d'entrée dans \"input box\" était : "$VARIABLE""

rm $FICHIER_SORTIE  # Nettoyage avec la suppression du fichier temporaire.
```

```
# Quelques applications pourraient avoir besoin de réclamer ce fichier.

exit $?

#  Exercice : Ré-écrivez ce script en utilisant l'ensemble de composants
#+ 'zenity'.
```

La commande xmessage est une méthode simple d'affichage d'une fenêtre contenant un message. Par exemple :

```
xmessage Erreur fatale dans le script! -button exit
```

Le dernier venu dans ce domaine est zenity. Cet outil affiche un dialogue *GTK+* et fonctionne très bien dans un script.

```
get_info ()
{
  zenity --entry       #  Montre une fenêtre de saisie
                       #+ et affiche la saisie de l'utilisateur en sortie.

                       #  Essayez aussi les options --calendar et --scale.
}

reponse=$( get_info )   #  Capture stdout dans la variable $reponse.

echo "L'utilisateur a saisi : "$answer""
```

Pour d'autres méthodes d'écriture des scripts utilisant des widgets, essayez *Tk* ou *wish* (des dérivés de *Tcl*), *PerlTk* (*Perl* avec des extensions *Tk*), *tksh* (*ksh* avec des extensions *Tk*), *XForms4Perl* (*Perl* avec des extensions *XForms*), *Gtk-Perl* (*Perl* avec des extensions *Gtk*) ou *PyQt* (*Python* avec des extensions *Qt*).

33.9. Problèmes de sécurité

33.9.1. Scripts shell infectés

Un bref message d'avertissement sur la sécurité des scripts est indiqué. Un script shell peut contenir un *ver (worm)*, un *troyen (trojan)* ou même un *virus*. Pour cette raison, ne lancez jamais un script en tant que *root* (ou ne permettez jamais son insertion dans les scripts de démarrage du système `/etc/rc.d`) à moins que vous n'ayez obtenu ledit script d'une source de confiance ou que vous l'ayez consenscieusement analysé pour vous assurer qu'il ne fait rien de nuisible.

De nombreux chercheurs chez Bell Labs et d'autres sites, incluant M. Douglas McIlroy, Tom Duff et Fred Cohen ont étudié les implications des virus de scripts shell. Ils concluent qu'il est tout à fait facile même pour un novice, un « script kiddie », d'en écrire un. [5]

Voici encore une autre raison d'apprendre les scripts. Être capable de regarder et de comprendre les scripts peut protéger votre système d'être compromis par un script.

33.9.2. Cacher le source des scripts shell

Pour des raisons de sécurité, il pourrait être nécessaire de rendre un script illisible. Si seulement il existait un outil pour créer un binaire exécutable à partir d'un script. *shc -- le compilateur générique de scripts shell de Francisco Rosales* fait exactement cela.

Malheureusement, d'après un *article* dans le numéro d'octobre 2005 du *Linux Journal*, le binaire peut, au moins dans certains cas, être décrypté pour retrouver le source original du script. Malgré tout, cette méthode pourrait être utile pour conserver une certaine sécurité dans les scripts.

33.9.3. Écrire des scripts shell sécurisés

Dan Stromberg suggère les lignes de conduite suivante pour écrire des scripts shell (relativement) sécurisés.

- Ne placez pas des données secrètes dans des variables d'environnement.
- Ne passez pas des données secrètes en arguments de commandes externes (à la place, utilisez un tube ou une redirection).
- Configurez votre $PATH avec attention. Ne faites pas confiance aux chemins dont vous héritez par l'appelant sur votre script

[5] Voir l'article de Marius van Oers, *UNIX Shell Scripting Malware*, et aussi la référence *Denning476* dans la *bibliographie*.

fonctionne en tant qu'utilisateur *root*. En fait, à chaque fois que vous utilisez une variable d'environnement héritée d'un appelant, pensez à ce qui peut arriver dans le cas où l'appelant a modifié la variable. Par exemple, l'appelant peut avoir modifié la valeur de $HOME avec `/etc`.

33.10. Problèmes de portabilité

Ce livre s'occupe principalement des scripts Bash sur un système GNU/Linux. De la même façon, les utilisateurs de **sh** et **ksh** y trouveront beaucoup d'idées de grande valeur.

Un grand nombre de shells et de langages de scripts semble converger vers le standard POSIX 1003.2. Appeler Bash avec l'option `--posix` ou insérer un **set -o posix** au début d'un script fait que Bash se conforme très étroitement à ce standard. Une autre alternative consiste à utiliser, dans le script, un en-tête sha-bang6 *#!/bin/sh* plutôt que *#!/bin/bash* [6] Notez que `/bin/sh` est un lien vers `/bin/bash` pour Linux ainsi que dans certaines autres versions d'UNIX et qu'un script appelé de cette façon désactive les fonctionnalités étendues de Bash.

La plupart des scripts Bash fonctionneront directement avec **ksh**, et vice-versa, car Chet Ramey a beaucoup travaillé sur le portage des fonctionnalités de **ksh** aux dernières versions de Bash.

Sur un UNIX commercial, les scripts utilisant les fonctionnalités spécifiques aux commandes standards GNU peuvent ne pas fonctionner. Ceci devient de moins en moins un problème ces dernières années car les outils GNU ont petit à petit remplacé les versions propriétaires même sur les UNIX « solides ». *La publication des sources* de nombreux outils de Caldera ne fera qu'accélérer la tendance.

Bash dispose de certaines fonctionnalités manquant au shell Bourne. Parmi celles-ci :

- Certaines options étendues d'appel419
- La substitution de commandes141 utilisant la notation **$( )**
- Certaines opérations de manipulations de chaînes
- La substitution de processus349
- Les commandes intégrées159 de Bash

Voir la FAQ de *Bash* pour une liste complète.

33.11. Scripts sous Windows

Même les utilisateurs sous *d'autres* OS peuvent exécuter des scripts shell de type UNIX et donc bénéficier d'un grand nombre des leçons de ce livre. Le paquet *Cygwin* de Cygnus et les *utilitaires MKS* de Mortice Kern Associates ajoutent des fonctionnalités de scripts à Windows.

Il y a eu des rumeurs comme quoi une future version de Windows contiendrait des fonctionnalités de scripts de commandes style Bash mais cela reste à voir.

[6] Or, better yet, #!/bin/env sh309.

Chapitre 34. Bash, version 2 et 3

34.1. Bash, version 2

La version actuelle de *Bash*, celle que vous avez sur votre machine, est la version 2.xx.y ou 3.xx.y.

```
bash$ echo $BASH_VERSION
3.2.25(1)-release
```

La mise à jour, version 2, du langage de script Bash classique ajoute les variables de type tableau, [1] l'expansion de chaînes de caractères et de paramètres, et une meilleure méthode pour les références de variables indirectes, parmi toutes les fonctionnalités.

Exemple 34.1. Expansion de chaîne de caractères

```
#!/bin/bash

# Expansion de chaînes de caractères.
# Introduit avec la version 2 de Bash.

# Les chaînes de caractères de la forme $'xxx' ont les caractères d'échappement
# standard interprétés.

echo $'Trois cloches sonnant à la fois \a \a \a'
      # Pourrait sonner seulement une fois sur certains terminaux.
echo $'Trois retours chariot \f \f \f'
echo $'10 retours chariot \n\n\n\n\n\n\n\n\n\n'
echo $'\102\141\163\150'   # Bash
                           # Équivalent en octal des caractères.

exit 0
```

Exemple 34.2. Références de variables indirectes - la nouvelle façon

```
#!/bin/bash

# Référencement de variables indirectes.
# Ceci a quelques-uns des attributs du C++.


a=lettre_de_l_alphabet
lettre_de_l_alphabet=z

echo "a = $a"           # Référence directe.

echo "Maintenant a = ${!a}"    # Référence indirecte.
# La notation ${!variable} est bien supérieure à l'ancien "eval var1=\$$var2"

echo

t=cellule_table_3
cellule_table_3=24
echo "t = ${!t}"        # t = 24
cellule_table_3=387
echo "La valeur de t a changé en ${!t}"    # 387

# Ceci est utile pour référencer les membres d'un tableau ou d'une table,
# ou pour simuler un tableau multi-dimensionnel.
#  Une option d'indexage (analogue à un pointeur arithmétique) aurait été bien.
#+ Sigh.

exit 0
```

[1] Chet Ramey a promit des tableaux associatifs (une fonctionnalité Perl intéressante) dans une future version de Bash. La version 3.2 n'en dispose toujours pas.

Exemple 34.3. Simple application de base de données, utilisant les références de variables indirectes

```
#!/bin/bash
# resistor-inventory.sh
# Simple base de données utilisant le référencement indirecte de variables.

# ============================================================== #
# Données

B1723_value=470                                 # Ohms
B1723_powerdissip=.25                           # Watts
B1723_colorcode="yellow-violet-brown"           # Bandes de couleurs
B1723_loc=173                                   # Où elles sont
B1723_inventory=78                              # Combien

B1724_value=1000
B1724_powerdissip=.25
B1724_colorcode="brown-black-red"
B1724_loc=24N
B1724_inventory=243

B1725_value=10000
B1725_powerdissip=.25
B1725_colorcode="brown-black-orange"
B1725_loc=24N
B1725_inventory=89

# ============================================================== #


echo

PS3='Entrez le numéro du catalogue : '

echo

select numero_catalogue in "B1723" "B1724" "B1725"
do
  Inv=${numero_catalogue}_inventory
  Val=${numero_catalogue}_value
  Pdissip=${numero_catalogue}_powerdissip
  Loc=${numero_catalogue}_loc
  Ccode=${numero_catalogue}_colorcode

  echo
  echo "Catalogue numéro $numero_catalogue :"
  echo "Il existe ${!Inv} résistances de [${!Val} ohm / ${!Pdissip} watt] en stock."
  echo "Elles sont situées dans bin # ${!Loc}."
  echo "Leur code couleur est \"${!Ccode}\"."

  break
done

echo; echo

# Exercice :
# ---------
#   Réécrire ce script en utilisant des tableaux, plutôt qu'en utilisant le
#+ référencement indirecte des variables.
# Quelle méthode est plus logique et intuitive ?


# Notes :
# ------
#  Les scripts shells sont inappropriés pour tout, sauf des applications simples
#+ de base de données, et, même là, cela implique des astuces.
#  Il est bien mieux d'utiliser un langage supportant nativement les structures
#+ de données, tels que C++ ou Java (voire même Perl).
```

```
exit 0
```

Exemple 34.4. Utiliser des tableaux et autres astuces pour gérer quatre mains aléatoires dans un jeu de cartes

```
#!/bin/bash

# Cartes :
# Gère quatre mains d'un jeu de cartes.

NON_RECUPERE=0
RECUPERE=1

DUPE_CARD=99

LIMITE_BASSE=0
LIMITE_HAUTE=51
CARTES_DANS_SUITE=13
CARTES=52

declare -a Jeu
declare -a Suites
declare -a Cartes
#  Le script aurait été plus simple à implémenter et plus intuitif
#+ avec un seul tableau à trois dimensions.
# Peut-être qu'une future version de Bash gèrera des tableaux multi-dimensionnels.


initialise_Jeu ()
{
i=$LIMITE_BASSE
until [ "$i" -gt $LIMITE_HAUTE ]
do
  Jeu[i]=$NON_RECUPERE   # Initialise chaque carte d'un "Jeu" comme non récupérée.
  let "i += 1"
done
echo
}

initialise_Suites ()
{
Suites[0]=C #Carreaux
Suites[1]=D #Piques
Suites[2]=H #Coeurs
Suites[3]=S #Trèfles
}

initialise_Cartes ()
{
Cartes=(2 3 4 5 6 7 8 9 10 J Q K A)
# Autre méthode pour initialiser un tableau.
}

recupere_une_carte ()
{
numero_carte=$ALEATOIRE
let "numero_carte %= $CARTES"
if [ "${Jeu[numero_carte]}" -eq $NON_RECUPERE ]
then
  Jeu[numero_carte]=$RECUPERE
  return $numero_carte
else
  return $DUPE_CARD
fi
}

analyse_carte ()
```

```
{
nombre=$1
let "suit_nombre = nombre / CARTES_DANS_SUITE"
suite=${Suites[suit_nombre]}
echo -n "$suit-"
let "no_carte = nombre % CARTES_DANS_SUITE"
Carte=${Cartes[no_carte]}
printf %-4s $Carte
# Affiche proprement les cartes.
}

recherche_nombre_aleatoire ()  # Générateur de nombres aléatoires.
{                              # Que se passe-t'il si vous ne faites pas cela ?
recherche=`eval date +%s`
let "recherche %= 32766"
ALEATOIRE=$recherche
# Quelles sont les autres méthodes de génération de nombres aléatoires ?
}

gere_cartes ()
{
echo

cartes_recuperees=0
while [ "$cartes_recuperees" -le $LIMITE_HAUTE ]
do
  recupere_une_carte
  t=$?

  if [ "$t" -ne $DUPE_CARD ]
  then
    analyse_carte $t

    u=$cartes_recuperees+1
    # Retour à un indexage simple (temporairement). Pourquoi ?
    let "u %= $CARTES_DANS_SUITE"
    if [ "$u" -eq 0 ]   # Condition if/then imbriquée.
    then
     echo
     echo
    fi
    # Mains séparées.

    let "cartes_recuperees += 1"
  fi
done

echo

return 0
}


# Programmation structurée :
# La logique entière du programme est modularisée en fonctions.

#================
recherche_nombre_aleatoire
initialise_Jeu
initialise_Suites
initialise_Cartes
gere_cartes
#================

exit 0



# Exercice 1 :
```

```
# Ajouter des commentaires détaillées de ce script.

# Exercice 2 :
# Ajouter une routine (fonction) pour afficher chaque main triée par suite.
# Vous pouvez ajouter d'autres fonctionnalités suivant vos souhaits.

# Exercice 3 :
# Simplifier et améliorer la logique du script.
```

34.2. Bash, version 3

Le 27 juillet 2004, Chet Ramey a sorti la version 3 de Bash. Cette mise à jour a corrigé un bon nombre de bogues et a ajouté quelques nouvelles fonctionnalités.

Voici quelques-unes des nouvelles fonctionnalités les plus importantes :

- Un nouvel opérateur, plus général, **{a..z}** expansion d'accolades.

```
#!/bin/bash

for i in {1..10}
#  Plus simple et direct que
#+ for i in $(seq 10)
do
  echo -n "$i "
done

echo

# 1 2 3 4 5 6 7 8 9 10


# Ou simplement...

echo {a..z}    #  a b c d e f g h i j k l m n o p q r s t u v w x y z
echo {z..a}    #  z y x w v u t s r q p o n m l k j i h g f e d c b a
               #  Fonctionne aussi à l'envers.
echo {3..-2}   #  3 2 1 0 -1 -2
echo {X..d}    #  X Y Z [  ] ^ _ ` a b c d
               #  Montre (certains) des caractères ASCII entre Z et a,
               #+ mais ne vous basez pas sur ce type de comportement parce que...
echo {]..a}    #  {]..a}
               #  Pourquoi ?
```

- L'opérateur **${!tableau[@]}**, qui s'étend sur tous les indices d'un tableau375 donné.

```
#!/bin/bash

Tableau=(élément-zéro élément-un élément-deux élément-trois)

echo ${Tableau[0]}   # élément-zéro
                     # Premier élément du tableau.

echo ${!Tableau[@]}  # 0 1 2 3
                     # Tous les indices de Tableau.

for i in ${!Tableau[@]}
do
  echo ${Tableau[i]} # élément-zéro
                     # élément-un
                     # élément-deux
                     # élément-trois
                     #
                     # Tous les éléments de Tableau.
done
```

- L'opérateur de correspondance =~ d'une expression rationnelle313 à l'intérieur d'une expression de tests à crochets double50. (Perl a un opérateur similaire.)

```
#!/bin/bash

variable="C'est un joyeux bazar."

echo "$variable"

# Opérateur d'expression rationnelle =~ à l'intérieur d'un [[ crochet double ]].
if [[ "$variable" =~ "C........joy*za*" ]]
#                     ^                ^
# NOTE: Mise entre guillemets non nécessaire à partir de la version 3.2 de Bash.
then
  echo "correspondance vraie"
      # correspondance vraie
fi
```

Ou, de façon plus utile :

```
#!/bin/bash

entree=$1


if [[ "$entree" =~ "[1-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]" ]]
#                  ^ NOTE: Mise entre guillemets non nécessaire
#                          à partir de la version 3.2 de Bash.
# NNN-NN-NNNN (où chaque N est un chiffre). L'entier initial ne doit pas être 0.
then
  echo "Numéro de sécurité sociale."
  # Traitement du NSS.
else
  echo "Ce n'est pas un numéro de sécurité sociale !"
  # Ou, demandez une saisie correcte.
fi
```

Pour d'autres exemples d'utilisation de l'opérateur =~, voir l'Exemple A.31, « Chasse aux spammeurs », Exemple 18.14, « Analyser une boîte mail », Exemple A.37, « Localise les paragraphes de division dans un fichier texte » et Exemple A.26, « Convertir en HTML ».

- La nouvelle option `set -o pipefail` est utile pour le débogage des tubes. Si cette option est activée, alors le code de sortie44 d'un tube est le code de sortie de la dernière commande du tube qui a *échoué* (qui a renvoyé un code de sortie différent de zéro), plutôt que la dernière commande réelle du tube.

  Voir Exemple 15.43, « Mettre à jour FC4 ».

Attention

La mise à jour à la version 3 de Bash casse quelques scripts qui fonctionnaient avec les anciennes versions. *Testez les scripts critiques pour vous assurer qu'ils fonctionnent toujours !*

Quelques scripts du *Guide ABS* ont dû être corrigés (voir l'Exemple A.21, « **obj-oriented**: Bases de données orientées objet » et l'Exemple 9.4, « *read* avec délai », par exemple).

34.2.1. Bash, version 3.1

La version 3.1 de Bash introduit un certain nombre de corrections et quelques modifications mineures.

- L'utilisation de l'opérateur += est maintenant autorisé à des endroits où seul l'opérateur d'affectation = était reconnu.

```
a=1
echo $a        # 1

a+=5           # Ne fonctionnera pas avec les versions Bash précédant la 3.1.
```

```
echo $a         # 15

a+=Hello
echo $a         # 15Hello
```

Ici, += fonctionne comme l'opérateur de *concaténation de chaîne*. Notez que ce comportement dans ce contexte particulier est différent de celui d'une construction let.

```
a=1
echo $a         # 1

let a+=5        # Arithmétique entière, plutôt que concaténation de chaînes.
echo $a         # 6

let a+=Hello    # N' "ajoute" rien à a.
echo $a         # 6
```

34.2.2. Bash, version 3.2

C'est principalement une mise à jour corrective.

- Dans la substitution de paramètres *globaux*, le modèle ne s'ancre plus au début de la chaîne.
- L'option `--wordexp` désactive la substitution de processus349.
- L'opérateur de correspondance des expressions rationnelles470 =~ ne demande plus la mise entre guillemets37 du *motif* à l'intérieur de [[...]]50.

Attention

En fait, mettre entre guillemets dans ce contexte n'est *pas* conseillé car cela pourrait causer un échec de l'évaluation de l'expression rationnelle. Voir la *liste des bogues sur Ubuntu* et *Wikinerds on Bash syntax*.

Avec Bash version 3.2.25(1), sur Fedora Core, la mise entre guillemets fonctionne mais ne supposez pas que cela sera le cas sur *votre* machine.

Chapitre 35. Notes finales

35.1. Note de l'auteur

doce ut discas

(Enseignez, afin que vous-même puissiez apprendre.)

Comment en suis-je venu à écrire un livre sur l'écriture de scripts Bash ? C'est une étrange histoire. Il semble qu'il y a quelques années, j'avais besoin d'apprendre à écrire des scripts shell -- et quelle meilleure façon de le faire que de lire un bon livre sur le sujet ? J'ai cherché à acheter un tutoriel et une référence couvrant tous les aspects du sujet. Je cherchais un livre qui prendrait tous les concepts difficiles, les expliquerait dans un soucis du détail avec des exemples bien commentés. [1] En fait, je recherchais *exactement ce livre*. Malheureusement, il n'existait pas et, si je le voulais, je devais l'écrire. Et donc nous en sommes là.

Cela me rappelle l'histoire apocryphe du professeur fou. Il était complètement fou. À la vue d'un livre, de tout livre -- à la bibliothèque, à la librairie, partout -- il devenait complètement obsédé avec l'idée qu'il pourrait l'avoir écrit, devrait l'avoir écrit et fait un meilleur travail pour commencer. Il aurait foncé chez lui et fait simplement cela, écrire un livre avec exactement le même titre. À sa mort quelques années après, il aurait eu plusieurs milliers de livre à son actif, plaçant Asimov lui-même dans la honte. Les livres pouvaient ne pas être bon -- qui sait -- mais est-ce que cela comptait ? Voici un brave homme qui a vécu son rêve, même s'il l'a obsédé, et je ne peux m'empêcher d'admirer ce vieux fou...

35.2. À propos de l'auteur

Mais qui est ce gars ?

L'auteur ne se prétend aucun crédit ou qualifications spéciales, [2] en dehors d'une certaine compulsion pour l'écriture. [3] Ce livre est un peu à l'opposé de son autre travail majeur, *HOW-2 Meet Women: The Shy Man's Guide to Relationships* (NdT : Comment rencontrer les femmes : le guide des relations à l'usage de l'homme timide). Il a aussi écrit le *Software-Building HOWTO*. Dernièrement, il s'essaie aux nouvelles.

Utilisateur Linux depuis 1995 (Slackware 2.2, noyau 1.2.1), l'auteur a produit quelques perles, incluant l'utilitaire de cryptage en une passe *cruft* , le calculateur *mcalc* , l'*arbitre* pour le Scrabble(r) et le paquetage d'une liste de jeux de mots *yawl*. Il a débuté en programmant en FORTRAN IV sur un CDC 3800 mais il n'est pas le moins du monde nostalgique de ces jours.

Vivant dans une communauté reculée du désert avec son épouse et son chat orange, il chérit la faiblesse humaine, et tout spécialement la sienne. [4]

35.3. Où trouver de l'aide

L'*auteur* répondra quelque fois aux questions générales sur l'écriture de script s'il n'est pas trop occupé (et s'il est plein de bonnes volontés). [5] Néanmoins, si vous avez un problème pour faire fonctionner un script spécifique, il vous est conseillé de poster votre problème sur le groupe Usenet *comp.os.unix.shell*.

Si vous avez besoin d'aide dans un travail pour l'école, lisez les sections pertinentes sur ce point et sur les autres références. Faites de votre mieux pour résoudre le problème en utilisant votre intelligence et vos ressources propres. Merci de ne pas gaspiller le temps de l'auteur. Vous n'obtiendrez ni aide ni sympathie. [6]

35.4. Outils utilisés pour produire ce livre

35.4.1. Matériel

Un IBM Thinkpad usé, modèle 760XL (P166, 104 meg RAM) sous Red Hat 7.1/7.3. Ok, il est lent et a un drôle de clavier, mais il bat un bloc-notes et une plume sergent major.

[1] C'est la technique très connue du *flagellez-le à mort*.
[2] En fait, l'auteur a été viré de l'école et n'a ni crédit ni qualifications.
[3] Ceux qui le peuvent le font. Ceux qui ne le peuvent pas... prenez un MCSE.
[4] Quelques fois, il semble qu'il a passé sa vie entière à faire fi de la sagesse conventionnelle et en défiant la Voix de l'Autorité disant « *Hé, vous ne pouvez pas faire cela !* »
[5] Les mail provenant de certains TLD infestés de spams (61, 202, 211, 218, 220, etc.) seront récupérés par les filtres anti-spams et détruits sans avoir été lus. Si votre ISP en fait partie, merci d'utiliser un compte Webmail pour contacter l'auteur.
[6] Bon, si vous insistez *vraiment*, vous pouvez tenter de modifier Exemple A.42, « Un outil de résolution général » pour qu'il réponde à vos besoins.

Mise à jour : passé à un 770Z Thinkpad (P2-366, 192 Mo de RAM) avec FC3. Quelqu'un souhaite donner un portable dernière génération à un écrivant en manque <g>?

Mise à jour : passé à un A31 Thinkpad (P4-1.6, 512 meg RAM) sous FC8. Plus de manque, plus besoin de solliciter des donations <g>.

35.4.2. Logiciel et impression

i. L'éditeur de texte de Bram Moolenaar, avec sa puissante connaissance de SGML, *vim*.

ii. *OpenJade*, un moteur de rendu DSSSL pour convertir des documents SGML en d'autres formats.

iii *Les feuilles de style DSSSL de Norman Walsh*.

iv. *DocBook, The Definitive Guide*, par Norman Walsh et Leonard Muellner (O'Reilly, ISBN 1-56592-580-7). C'est toujours la référence standard pour tout ceux qui essaient d'écrire un document avec le format Docbook SGML.

35.5. Remerciements

35.5.1. Pour la version originale

La participation de la communauté a rendu ce projet possible. L'auteur reconnait qu'écrire ce livre aurait été une tâche impossible sans l'aide et les retours de toutes ces personnes.

Philippe Martin a traduit la première version (0.1) de ce document en DocBook/SGML. Alors que ce n'est pas son travail dans cette petite compagnie française où il est développeur, il aime travailler sur la documentation et le logiciel GNU/Linux, lire de la littérature, jouer de la musique et rendre heureux ses amis. Vous pouvez le rencontrer en France ou dans le pays Basque, ou lui envoyer un courrier électronique à *feloy@free.fr*.

Philippe Martin m'a aussi indiqué que les paramètres positionnels après $9 sont possibles en utilisant la notation des {accolades} (voir l'Exemple 4.5, « Paramètres positionnels »).

Stéphane Chazelas a envoyé une longue liste de corrections, ajouts et exemples de scripts. Plus qu'un contributeur, il a, dans les faits, pendant un moment, pris le rôle de *co-éditeur* pour ce document. *Merci beaucoup !* (NdT : en français dans le texte)

Je voudrais spécialement remercier *Patrick Callahan*, *Mike Novak* et *Pal Domokos* pour avoir trouvé des bogues, indiqué les ambiguités et suggéré des clarifications et des modifications. Leurs discussions vivantes m'ont inspiré pour essayer de rendre ce document lisible.

Je suis reconnaissant à Jim Van Zandt d'avoir pointé les erreurs et omissions dans la version 0.2 de ce document. Il a aussi contribué à un script d'exemple496 instructif.

Un grand remerciement à *Jordi Sanfeliu* pour m'avoir donné la permission d'utiliser son script (Exemple A.17, « *tree*: Afficher l'arborescence d'un répertoire ») et à Rick Boivie pour l'avoir relu.

De même, merci à *Michel Charpentier* pour sa permission d'utiliser son script de factorisation dc (Exemple 15.51, « Factorisation »).

Merci à *Noah Friedman* pour sa permission d'utiliser sa fonction sur les chaînes de caractères (Exemple A.19, « *string*: Manipuler les chaînes de caractères comme en C »).

Emmanuel Rouat a suggéré des corrections et ajouts sur la substitution de commandes141 et sur les alias370. Il a aussi contribué à un très joli exemple de fichier `.bashrc` (Annexe K, Un exemple de fichier `.bashrc`).

Heiner Steven m'a gentimment donné la permission d'utiliser son script de conversion de base, Exemple 15.47, « Conversion de base ». Il a aussi fait un certain nombre de corrections et de suggestions d'une grande aide. Grands mercis.

Rick Boivie a contribué au script délicieusement récursif *pb.sh* (Exemple 33.9, « Un script (utile) qui s'appelle récursivement »), a revu le script *tree.sh* (Exemple A.17, « *tree*: Afficher l'arborescence d'un répertoire ») et aux améliorations de performances pour le script *monthlypmt.sh* (Exemple 15.46, « Paiement mensuel sur une hypothèque »).

Florian Wisser m'a montré des points très fin sur les tests des chaînes de caractères (voir Exemple 7.6, « Vérification si une chaîne est *nulle* »), mais aussi sur d'autres points.

Oleg Philon a envoyé des suggestions concernant cut et pidof.

Michael Zick a amélioré l'exemple du tableau vide pour démontrer des propriétés étonnantes sur les tableaux. Il a aussi contribué aux scripts *isspammer* (Exemple 15.41, « Analyser le domaine d'un courrier indésirable » et Exemple A.30, « Identification d'un spammer »).

Marc-Jano Knopp a envoyé des corrections et des clarifications sur les fichiers batch DOS.

Hyun Jin Cha a trouvé plusieurs erreurs dans le document en effectuant une traduction coréenne. Merci de me les avoir indiquées.

Andreas Abraham a envoyé une longue liste d'erreurs de typographie et d'autres corrections. Un grand merci !

D'autres ont contribué aux scripts, fait des suggestions nous ayant bien aidés et pointé des erreurs. Il s'agit de Gabor Kiss, Leopold Toetsch, Peter Tillier, Marcus Berglof, Tony Richardson, Nick Drage (idées de script !), Rich Bartell, Jess Thrysoee, Adam Lazur, Bram Moolenaar, Baris Cicek, Greg Keraunen, Keith Matthews, Sandro Magi, Albert Reiner, Dim Segebart, Rory Winston, Lee Bigelow, Wayne Pollock, « jipe », « bojster », « nyal », « Hobbit », « Ender », « Little Monster » (Alexis), « Mark », « Patsie », Emilio Conti, Ian. D. Allen, Hans-Joerg Diers, Arun Giridhar, Dennis Leeuw, Dan Jacobson, Aurelio Marinho Jargas, Edward Scholtz, Jean Helou, Chris Martin, Lee Maschmeyer, Bruno Haible, Wilbert Berendsen, Sebastien Godard, Björn Eriksson, John MacDonald, Joshua Tschida, Troy Engel, Manfred Schwarb, Amit Singh, Bill Gradwohl, E. Choroba, David Lombard, Jason Parker, Steve Parker, Bruce W. Clare, William Park, Vernia Damiano, Mihai Maties, Mark Alexander, Jeremy Impson, Ken Fuchs, Jared Martin, Frank Wang, Sylvain Fourmanoit, Matthew Sage, Matthew Walker, Kenny Stauffer, Filip Moritz, Andrzej Stefanski, Daniel Albers, Stefano Palmeri, Serghey Rodin, Jeroen Domburg, Alfredo Pironti, Phil Braham, Bruno de Oliveira Schneider, Stefano Falsetto, Chris Morgan, Walter Dnes, Linc Fessenden, Michael Iatrou, Pharis Monalo, Jesse Gough, Fabian Kreutz, Mark Norman, Harald Koenig, Dan Stromberg, Peter Knowles, Francisco Lobo, Mariusz Gniazdowski, Sebastian Arming, Benno Schulenberg, Tedman Eng, Jochen DeSmet, Juan Nicolas Ruiz, Oliver Beckstein, Achmed Darwish, Richard Neill, Albert Siersema, Omair Eshkenazi, Geoff Lee, JuanJo Ciarlante, Cliff Bamford, Nathan Coulter, Andreas Kühne et David Lawyer (lui-même auteur de quatre guides pratiques.

Ma gratitude pour *Chet Ramey* et Brian Fox pour avoir écrit et construit un élégant et puissant outil de scripts, *Bash*.

Et un très grand merci pour les volontaires qui ont durement travaillé au *Linux Documentation Project*. Le LDP contient un dépôt de connaissances Linux et a, pour une grande partie, permis la publication de ce livre.

Remerciements à IBM, Red Hat, la *Free Software Foundation* et à toutes les personnes se battant justement pour garder les logiciels libres, libres et ouverts.

Merci en particulier à ma femme, Anita, pour ses encouragements et pour son support émotionnel.

35.5.2. Pour la version traduite

Toute une équipe a travaillé à la traduction et à la relecture de ce document. Cela n'a pas toujours été simple. Il a parfois fallu débattre sur le meilleur moyen de faire de ce guide un excellent manuel sur Bash en français. Mais le résultat est là.

Voici la liste des personnes, par ordre alphabétique, ayant participé à :

- la traduction/relecture des fichiers SGML :
    - Adrien Rebollo
    - Charles Duysinx
    - Dodo
    - Dup
    - Gabriel Giovannetti
    - Guillaume Lelarge
    - Ilan Bloch
    - Isabelle Hurbain (Balise)
    - Joris de Pooter
    - Josée Caumartin
    - Marc Chantreux
    - Pascal Gosse
    - Philippe Dexemple
    - Ph. Rimbault
    - ykerb
- la traduction/relecture des scripts :

- Adrien Rebollo
- Damien Dubedat
- Guillaume Lelarge
- Joris De Pooter
- Pascal Gosse

Qu'ils en soient tous remerciés chaleureusement.

Évidemment, il est possible que des erreurs ou des fautes se soient glissées dans ce guide. N'hésitez pas à nous indiquer tout problème à l'adresse mail suivante : traduc AT traduc.org. N'hésitez pas non plus à venir rejoindre l'association *traduc.org*.

35.6. Avis de non-responsabilité

(Ceci est une variante de l'avis de non-responsabilité du *LDP*.)

Aucune responsabilité sur le contenu de ce document ne sera acceptée. Utilisez les concepts, exemples et informations à vos risques et périls. Il peut y avoir des erreurs ou des inexactitudes qui pourraient être la source de perte de données ou de dommages sur votre système, donc *faites particulièrement attention*. L'auteur n'est pas responsable des dommages causés.

Il serait très étonnant que vous ou votre système ayez un problème avec ce livre. En fait, la *raison d'être* de ce livre est de permettre aux lecteurs d'analyser des scripts shell et de déterminer si ces derniers peuvent avoir des *conséquences inattendues*.

Bibliographie

Those who do not understand UNIX are condemned to reinvent it, poorly.

-- Henry Spencer

Peter Denning. *Computers Under Attack: Intruders, Worms, and Viruses*. ACM Press. Copyright © 1990. 0-201-53067-8.

Ken Burtch. *Linux Shell Scripting with Bash*. première édition. Sams Publishing (Pearson). Copyright © 2004. 0672326426.

Dale Dougherty et Arnold Robbins. *Sed and Awk*. 2e édition. O'Reilly and Associates. Copyright © 1997. 1-156592-225-5.

Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly and Associates. Copyright © 2002. 0-596-00289-0.

Aeleen Frisch. *Essential System Administration*. 3e édition. O'Reilly and Associates. Copyright © 2002. 0-596-00343-9.

Aeleen Frisch. *Les bases de l'administration système*. 3e édition. O'Reilly and Associates. Copyright © 2002. 2-84177-222-5.

Stephen Kochan et Patrick Wood. *UNIX Shell Programming*. Hayden. Copyright © 1990. 067248448X.

Neil Matthew et Richard Stones. *Beginning Linux Programming*. Wrox Press. Copyright © 1996. 1874416680.

[mayerref] Herbert Mayer. *Advanced C Programming on the IBM PC*. Windcrest Books. Copyright © 1989. 0830693637.

David Medinets. *UNIX Shell Programming Tools*. McGraw-Hill. Copyright © 1999. 0070397333.

Cameron Newham et Bill Rosenblatt. *Learning the Bash Shell*. 2e édition. O'Reilly and Associates. Copyright © 1998. 1-56592-347-2.

Anatole Olczak. *Bourne Shell Quick Reference Guide*. ASP, Inc.. Copyright © 1991. 093573922X.

Jerry Peek, Tim O'Reilly, et Mike Loukides. *UNIX Power Tools*. 3e édition. O'Reilly and Associates. Random House. Copyright © 2002. 0-596-00330-7.

Clifford Pickover. *Computers, Pattern, Chaos, and Beauty*. St. Martin's Press. Copyright © 1990. 0-312-04123-3.

George Polya. *How To Solve It*. Princeton University Press. Copyright © 1973. 0-691-02356-5.

Chet Ramey et Brian Fox. *The GNU Bash Reference Manual*. Network Theory Ltd. Copyright © 2003. 0-9541617-7-7.

Arnold Robbins. *Bash Reference Card*. SSC. Copyright © 1998. 1-58731-010-5.

Arnold Robbins. *Effective Awk Programming*. Free Software Foundation / O'Reilly and Associates. Copyright © 2000. 1-882114-26-4.

Bill Rosenblatt. *Learning the Korn Shell*. O'Reilly and Associates. Copyright © 1993. 1-56592-054-6.

Paul Sheer. *LINUX: Rute User's Tutorial and Exposition*. 1ère édition. . Copyright © 2002. 0-13-033351-4.

Ellen Siever et l'équipe d'O'Reilly and Associates. *Linux in a Nutshell*. 2e édition. O'Reilly and Associates. Copyright © 1999. 1-56592-585-8.

Dave Taylor. *Wicked Cool Shell Scripts: 101 Scripts for Linux, Mac OS X, and Unix Systems*. 1ère édition. No Starch Press. Copyright © 2004. 1-59327-012-7.

The UNIX CD Bookshelf. 2e édition. O'Reilly and Associates. Copyright © 2000. 1-56592-815-6.

Annexe A. Contribution de scripts

Ces scripts, bien que ne rentrant pas dans le texte de ce document, illustrent quelques techniques intéressantes de programmation shell. Ils sont aussi utiles. Amusez-vous à les analyser et à les lancer.

Exemple A.1. *mailformat* : Formater un courrier électronique

```
#!/bin/bash
#  mail-format.sh (ver. 1.1) : Formate les courriers électroniques.

#  Supprime les caractères '>', les tabulations et coupe aussi les lignes
#+ excessivement longues.

# ================================================================
#                 Vérification standard des argument(s) du script
ARGS=1
E_MAUVAISARGS=65
E_PASDEFICHIER=66

if [ $# -ne $ARGS ]  # Le bon nombre d'arguments a-t'il été passé au script?
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

if [ -f "$1" ]       # Vérifie si le fichier existe.
then
    nomfichier=$1
else
    echo "Le fichier \"$1\" n'existe pas."
    exit $E_PASDEFICHIER
fi
# ================================================================

LONGUEUR_MAX=70
# Longueur à partir de laquelle on coupe les lignes excessivement longues.

# --------------------------------
# Une variable peut contenir un script sed.
scriptsed='s/^>//
s/^  *>//
s/^  *//
s/         *//'
# --------------------------------

#  Supprime les caractères '>' et tabulations en début de lignes,
#+ puis coupe les lignes à $LONGUEUR_MAX caractères.
sed "$scriptsed" $1 | fold -s --width=$LONGUEUR_MAX
#  option -s pour couper les lignes à une espace blanche, si possible.


#  Ce script a été inspiré par un article d'un journal bien connu
#+ proposant un utilitaire Windows de 164Ko pour les mêmes fonctionnalités.
#
#  Un joli ensemble d'utilitaires de manipulation de texte et un langage de
#+ scripts efficace apportent une alternative à des exécutables gonflés.

exit 0
```

Exemple A.2. *rn* : Un utilitaire simple pour renommer des fichiers

Ce script est une modification de l'Exemple 15.22, « *lowercase* : Change tous les noms de fichier du répertoire courant en minuscule. ».

```
#! /bin/bash
#
# Un très simplifié "renommeur" de fichiers (basé sur "lowercase.sh").
#
#  L'utilitaire "ren", par Vladimir Lanin (lanin@csd2.nyu.edu),
#+ fait un bien meilleur travail que ceci.


ARGS=2
E_MAUVAISARGS=65
UN=1                    # Pour avoir correctement singulier ou pluriel
                        # (voir plus bas.)

if [ $# -ne "$ARGS" ]
then
  echo "Usage: `basename $0` ancien-modele nouveau-modele"
  #  Comme avec "rn gif jpg", qui renomme tous les fichiers gif du répertoire
  #+ courant en jpg.
  exit $E_MAUVAISARGS
fi

nombre=0                # Garde la trace du nombre de fichiers renommés.


for fichier in *$1*     # Vérifie tous les fichiers correspondants du répertoire.
do
   if [ -f "$fichier" ]  # S'il y a correspondance...
   then
     fname=`basename $fichier`             # Supprime le chemin.
     n=`echo $fname | sed -e "s/$1/$2/"`   # Substitue ancien par nouveau dans
                                           # le fichier.
     mv $fname $n                          # Renomme.
     let "nombre += 1"
   fi
done

if [ "$nombre" -eq "$UN" ]                # Pour une bonne grammaire.
then
  echo "$nombre fichier renommé."
else
  echo "$nombre fichiers renommés."
fi

exit 0


# Exercices:
# ---------
# Avec quel type de fichiers cela ne fonctionnera pas?
# Comment corriger cela?
#
#  Réécrire ce script pour travailler sur tous les fichiers d'un répertoire,
#+ contenant des espaces dans leur noms, et en les renommant après avoir
#+ substitué chaque espace par un tiret bas.
```

Exemple A.3. *blank-rename* : Renommer les fichiers dont le nom contient des espaces

C'est une version encore plus simple du script précédent.

```
#! /bin/bash
# blank-rename.sh
#
#  Substitue les tirets soulignés par des blancs dans tous les fichiers d'un
#+ répertoire.

UN=1                     # Pour obtenir le singulier/pluriel correctement (voir
                         # plus bas).
```

```
nombre=0                # Garde trace du nombre de fichiers renommés.
TROUVE=0                # Valeur de retour en cas de succès.

for fichier in *        #Traverse tous les fichiers du répertoire.
do
     echo "$fichier" | grep -q " "         #  Vérifie si le nom du fichier
     if [ $? -eq $TROUVE ]                #+ contient un (des) espace(s).
     then
       nomf=$fichier                      # Oui, ce nom de fichier doit être
travaillé.
       n=`echo $nomf | sed -e "s/ /_/g"`  # Remplace l'espace par un tiret.
       mv "$nomf" "$n"                    # Réalise le renommage.
       let "nombre += 1"
     fi
done

if [ "$nombre" -eq "$UN" ]                # Pour une bonne grammaire.
then
  echo "$nombre fichier renommé."
else
  echo "$nombre fichiers renommés."
fi

exit 0
```

Exemple A.4. *encryptedpw* : Charger un fichier sur un site ftp, en utilisant un mot de passe crypté en local

```
#!/bin/bash

# Exemple "ex72.sh" modifié pour utiliser les mots de passe cryptés.

#  Notez que c'est toujours moyennement sécurisé, car le mot de passe décrypté
#+ est envoyé en clair.
#  Utilisez quelque chose comme "ssh" si cela vous préoccupe.

E_MAUVAISARGS=65

if [ -z "$1" ]
then
  echo "Usage: `basename $0` nomfichier"
  exit $E_MAUVAISARGS
fi

NomUtilisateur=bozo      # Changez suivant vos besoins.
motpasse=/home/bozo/secret/fichier_avec_mot_de_passe_crypte
# Le fichier contient un mot de passe crypté.

Nomfichier=`basename $1` # Supprime le chemin du fichier

Serveur="XXX"            #  Changez le nom du serveur et du répertoire suivant
Repertoire="YYY"         #+ vos besoins.


MotDePasse=`cruft <$motpasse`          # Décrypte le mot de passe.
#  Utilise le paquetage de cryptage de fichier de l'auteur,
#+ basé sur l'algorithme classique "onetime pad",
#+ et disponible à partir de:
#+ Site primaire:  ftp://ibiblio.org/pub/Linux/utils/file
#+                 cruft-0.2.tar.gz [16k]


ftp -n $Serveur <<Fin-de-Session
user $NomUtilisateur $MotDePasse
binary
bell
cd $Repertoire
put $Nomfichier
```

```
bye
Fin-de-Session
# L'option -n de "ftp" désactive la connexion automatique.
# Notez que "bell" fait sonner une cloche après chaque transfert.

exit 0
```

Exemple A.5. *copy-cd* : Copier un CD de données

```
#!/bin/bash
# copy-cd.sh: copier un CD de données

CDROM=/dev/cdrom                            # périphérique CD ROM
OF=/home/bozo/projects/cdimage.iso          # fichier de sortie
#       /xxxx/xxxxxxx/                      A modifier suivant votre système.
TAILLEBLOC=2048
VITESSE=2                                   # Utiliser une vitesse supèrieure
                                            #+ si elle est supportée.
PERIPHERIQUE=cdrom
#PERIPHERIQUE="0,0" pour les anciennes versions de cdrecord

echo; echo "Insérez le CD source, mais ne le montez *pas*."
echo "Appuyez sur ENTER lorsque vous êtes prêt. "
read pret                                   # Attendre une entrée, $pret n'est
                                            # pas utilisé.

echo; echo "Copie du CD source vers $OF."
echo "Ceci peut prendre du temps. Soyez patient."

dd if=$CDROM of=$OF bs=$TAILLEBLOC          # Copie brute du périphérique.


echo; echo "Retirez le CD de données."
echo "Insérez un CDR vierge."
echo "Appuyez sur ENTER lorsque vous êtes prêt. "
read pret                                   # Attendre une entrée, $pret n'est
                                            # pas utilisé.

echo "Copie de $OF vers CDR."

cdrecord -v -isosize speed=$VITESSE dev=$PERIPHERIQUE $OF
# Utilise le paquetage "cdrecord" de Joerg Schilling's (voir sa doc).
# http://www.fokus.gmd.de/nthp/employees/schilling/cdrecord.html


echo; echo "Copie terminée de $OF vers un CDR du périphérique $CDROM."

echo "Voulez-vous écraser le fichier image (o/n)? "  # Probablement un fichier
                                                     # immense.
read reponse

case "$reponse" in
[oO]) rm -f $OF
      echo "$OF supprimé."
      ;;
*)    echo "$OF non supprimé.";;
esac

echo

#  Exercice:
#  Modifiez l'instruction "case" pour aussi accepter "oui" et "Oui" comme
#+ entrée.

exit 0
```

Exemple A.6. *collatz* : Séries de Collatz

```
#!/bin/bash
# collatz.sh

#  Le célèbre "hailstone" ou la série de Collatz.
#  ----------------------------------------------
#  1) Obtenir un entier "de recherche" à partir de la ligne de commande.
#  2) NOMBRE &lt;--- seed
#  3) Afficher NOMBRE.
#  4)  Si NOMBRE est pair, divisez par 2, ou
#  5)+ si impair, multiplier par 3 et ajouter 1.
#  6) NOMBRE &lt;--- résultat
#  7) Boucler à l'étape 3 (pour un nombre spécifié d'itérations).
#
#  La théorie est que chaque séquence, quelle soit la valeur initiale,
#+ se stabilisera éventuellement en répétant des cycles "4,2,1...",
#+ même après avoir fluctuée à travers un grand nombre de valeurs.
#
#  C'est une instance d'une "itération", une opération qui remplit son
#+ entrée par sa sortie.
#  Quelque fois, le résultat est une série "chaotique".


MAX_ITERATIONS=200
# Pour une grande échelle de nombre (&gt;32000), augmenter MAX_ITERATIONS.

h=${1:-$$}                      #  Nombre de recherche
                                #  Utiliser $PID comme nombre de recherche,
                                #+ si il n'est pas spécifié en argument de la
                                #+ ligne de commande.

echo
echo "C($h) --- $MAX_ITERATIONS Iterations"
echo

for ((i=1; i<=MAX_ITERATIONS; i++))
do

echo -n "$h "
#          ^^^^^
#           tab

  let "reste = h % 2"
  if [ "$reste" -eq 0 ]       # Pair?
  then
    let "h /= 2"              # Divise par 2.
  else
    let "h = h*3 + 1"         # Multiplie par 3 et ajoute 1.
  fi


COLONNES=10                   # Sortie avec 10 valeurs par ligne.
let "retour_ligne = i % $COLONNES"
if [ "$retour_ligne" -eq 0 ]
then
  echo
fi

done

echo

#  Pour plus d'informations sur cette fonction mathématique,
#+ voir _Computers, Pattern, Chaos, and Beauty_, par Pickover, p. 185 ff.,
#+ comme listé dans la bibliographie.

exit 0
```

Exemple A.7. *days-between* : Calculer le nombre de jours entre deux dates

```
#!/bin/bash
# days-between.sh:    Nombre de jours entre deux dates.
# Usage: ./days-between.sh [M]M/[D]D/AAAA [M]M/[D]D/AAAA
#
# Note: Script modifié pour tenir compte des changements dans Bash 2.05b +
#+      qui ont fermé la "fonctionnalité" permettant de renvoyer des valeurs
#+      entières négatives grandes.

ARGS=2                # Deux arguments attendus en ligne de commande.
E_PARAM_ERR=65        # Erreur de paramètres.

ANNEEREF=1600         # Année de référence.
SIECLE=100
JEA=365
AJUST_DIY=367         # Ajusté pour l'année bissextile + fraction.
MEA=12
JEM=31
CYCLE=4

MAXRETVAL=256         #  Valeur de retour positive la plus grande possible
                      #+ renvoyée par une fonction.

diff=                #  Déclaration d'une variable globale pour la différence
                      #+ de date.
value=                #  Déclaration d'une variable globale pour la valeur
                      #+ absolue.
jour=                 #  Déclaration de globales pour jour, mois, année.
mois=
annee=


Erreur_Param ()        # Mauvais paramètres en ligne de commande.
{
  echo "Usage: `basename $0` [M]M/[D]D/YYYY [M]M/[D]D/YYYY"
  echo "       (la date doit être supérieure au 1/3/1600)"
  exit $E_PARAM_ERR
}


Analyse_Date ()               #  Analyse la date à partir des paramètres en
{                             #+ ligne de commande.
  mois=${1%%/**}
  jm=${1%/**}                 # Jour et mois.
  jour=${dm#*/}
  let "annee = `basename $1`"  #  Pas un nom de fichier mais fonctionne de la
                              #+ même façon.
}


verifie_date ()                # Vérifie la validité d'une date.
{
  [ "$jour" -gt "$JEM" ] || [ "$mois" -gt "$MEA" ] ||
  [ "$annee" -lt "$ANNEEREF" ] && Erreur_Param
  # Sort du script si mauvaise(s) valeur(s).
  # Utilise une liste-ou ou une liste-et.
  #
  # Exercice: Implémenter une vérification de date plus rigoureuse.
}


supprime_zero_devant () #  Il est préférable de supprimer les zéros possibles
{                       #+ du jour et/ou du mois sinon Bash va les
  val=${1#0}            #+ interpréter comme des valeurs octales
```

```
  return $val           #+ (POSIX.2, sect 2.9.2.1).
}


index_jour ()         # Formule de Gauss:
{                     # Nombre de jours du 1er mars 1600 jusqu'à la date passée
                      # en arguments.

  jour=$1
  mois=$2
  annee=$3

  let "mois = $mois - 2"
  if [ "$mois" -le 0 ]
  then
    let "mois += 12"
    let "annee -= 1"
  fi

  let "annee -= $ANNEEREF"
  let "indexyr = $annee / $SIECLE"


  let "Jours = $JEA*$annee + $annee/$CYCLE - $indexyr \
              + $indexyr/$CYCLE + $AJUST_DIY*$mois/$MEA + $jour - $JEM"
  # Pour une explication en détails de cet algorithme, voir
  #+   http://weblogs.asp.net/pgreborio/archive/2005/01/06/347968.aspx

  echo $Days

}


calcule_difference ()             # Différence entre les indices de deux jours.
{
  let "diff = $1 - $2"            # Variable globale.
}


abs ()                            #  Valeur absolue.
{                                 #  Utilise une variable globale "valeur".
  if [ "$1" -lt 0 ]               #  Si négatif
  then                            #+ alors
    let "value = 0 - $1"          #+ change de signe,
  else                            #+ sinon
    let "value = $1"              #+ on le laisse.
  fi
}



if [ $# -ne "$ARGS" ]            # Requiert deux arguments en ligne de commande.
then
  Erreur_Param
fi

Analyse_Date $1
verifie_date $jour $mois $annee      #  Vérifie si la date est valide.

supprime_zero_devant $jour           #  Supprime tout zéro débutant
jour=$?                              #+ sur le jour et/ou le mois.
supprime_zero_devant $mois
mois=$?

let "date1 = `day_index $jour $mois $annee`"

Analyse_Date $2
verifie_date $jour $mois $annee
```

```
supprime_zero_devant $jour
jour=$?
supprime_zero_devant $mois
mois=$?

date2 = $(day_index $jour $mois $annee)  # Substitution de commande


calcule_difference $date1 $date2

abs $diff                           # S'assure que c'est positif.
diff=$value

echo $diff

exit 0
#  Comparez ce script avec l'implémentation de la formule de Gauss en C sur
#+ http://buschencrew.hypermart.net/software/datedif
```

Exemple A.8. *makedict* : Créer un *dictionnaire*

```
#!/bin/bash
# makedict.sh  [make dictionary]

# Modification du script /usr/sbin/mkdict (/usr/sbin/cracklib-forman).
# Script original copyright 1993, par Alec Muffett.
#
#  Ce script modifié inclus dans ce document d'une manière consistente avec le
#+ document "LICENSE" du paquetage "Crack" dont fait partie le script original.

#  Ce script manipule des fichiers texte pour produire une liste triée de mots
#+ trouvés dans les fichiers.
#  Ceci pourrait être utile pour compiler les dictionnaires et pour d'autres
#+ buts lexicographiques.


E_MAUVAISARGS=65

if [ ! -r "$1" ]                    #  Au moins un argument, qui doit être
then                                #+ un fichier valide.
        echo "Usage: $0 fichiers-à-manipuler"
  exit $E_MAUVAISARGS
fi


# SORT="sort"                     #  Plus nécessaire de définir des options
                                  #+ pour sort. Modification du script
                                  #+ original.

cat $* |                          # Contenu des fichiers spécifiés vers stdout.
        tr A-Z a-z |              # Convertion en minuscule.
        tr ' ' '\012' |           #  Nouveau: modification des espaces en
                                  #+ retours chariot.
#       tr -cd '\012[a-z][0-9]' | #  Suppression de tout ce qui n'est pas
                                  #  alphanumérique
                                  #+ (dans le script original).
        tr -c '\012a-z'  '\012' | #  Plutôt que de supprimer les caractères
                                  #+ autres qu'alphanumériques,
                                  #+ les modifie en retours chariot.
        sort |                    #  Les options $SORT ne sont plus
                                  #+ nécessaires maintenant.
        uniq |                    # Suppression des mots dupliqués.
        grep -v '^#' |            #  Suppression des lignes commençant avec
                                  #+ le symbole '#'.
        grep -v '^$'              # Suppression des lignes blanches.

exit 0
```

Exemple A.9. *soundex* : Conversion phonétique

```
#!/bin/bash
# soundex.sh: Calcule le code "soundex" pour des noms

# ======================================================
#        Script soundex
#              par
#         Mendel Cooper
#     thegrendel@theriver.com
#       23 Janvier 2002
#
#   Placé dans le domaine public.
#
#  Une version légèrement différente de ce script est apparu dans
#+ la colonne "Shell Corner" d'Ed Schaefer en juillet 2002
#+ du magazine en ligne "Unix Review",
#+ http://www.unixreview.com/documents/uni1026336632258/
# ======================================================


NBARGS=1                      # A besoin du nom comme argument.
E_MAUVAISARGS=70

if [ $# -ne "$NBARGS" ]
then
  echo "Usage: `basenom $0` nom"
  exit $E_MAUVAISARGS
fi


affecte_valeur ()             #  Affecte une valeur numérique
{                             #+ aux lettres du nom.

  val1=bfpv                   # 'b,f,p,v' = 1
  val2=cgjkqsxz               # 'c,g,j,k,q,s,x,z' = 2
  val3=dt                     #  etc.
  val4=l
  val5=mn
  val6=r

# Une utilisation particulièrement intelligente de 'tr' suit.
# Essayez de comprendre ce qui se passe ici.

valeur=$( echo "$1" \
| tr -d wh \
| tr $val1 1 | tr $val2 2 | tr $val3 3 \
| tr $val4 4 | tr $val5 5 | tr $val6 6 \
| tr -s 123456 \
| tr -d aeiouy )

# Affecte des valeurs aux lettres.
# Supprime les numéros dupliqués, sauf s'ils sont séparés par des voyelles.
# Ignore les voyelles, sauf en tant que séparateurs, donc les supprime à la fin.
# Ignore 'w' et 'h', même en tant que séparateurs, donc les supprime au début.
#
# La substitution de commande ci-dessus utilise plus de tube qu'un plombier
# <g>.

}


nom_en_entree="$1"
echo
echo "Nom = $nom_en_entree"
```

```
# Change tous les caractères en entrée par des minuscules.
# ------------------------------------------------
nom=$( echo $nom_en_entree | tr A-Z a-z )
# ------------------------------------------------
# Au cas où cet argument est un mélange de majuscules et de minuscules.


# Préfixe des codes soundex: première lettre du nom.
# ------------------------------------------


pos_caract=0                    # Initialise la position du caractère.
prefixe0=${nom:$pos_caract:1}
prefixe=`echo $prefixe0 | tr a-z A-Z`
                                # Met en majuscule la première lettre de soundex.

let "pos_caract += 1"           # Aller directement au deuxième caractères.
nom1=${nom:$pos_caract}


# ++++++++++++++++++++++++++ Correctif Exception ++++++++++++++++++++++++++++++++
#  Maintenant, nous lançons à la fois le nom en entrée et le nom décalé d'un
#+ caractère vers la droite au travers de la fonction d'affectation de valeur.
#  Si nous obtenons la même valeur, cela signifie que les deux premiers
#+ caractères du nom ont la même valeur et que l'une d'elles doit être annulée.
#  Néanmoins, nous avons aussi besoin de tester si la première lettre du nom est
#+ une voyelle ou 'w' ou 'h', parce que sinon cela va poser problème.

caract1=`echo $prefixe | tr A-Z a-z`    # Première lettre du nom en minuscule.

affecte_valeur $nom
s1=$valeur
affecte_valeur $nom1
s2=$valeur
affecte_valeur $caract1
s3=$valeur
s3=9$s3                             #  Si la première lettre du nom est une
                                    #+ voyelle ou 'w' ou 'h',
                                    #+ alors sa "valeur" sera nulle (non
                                    #+ initialisée).
                                    #+ Donc, positionnons-la à 9, une autre
                                    #+ valeur non utilisée, qui peut être
                                    #+ vérifiée.


if [[ "$s1" -ne "$s2" || "$s3" -eq 9 ]]
then
  suffixe=$s2
else
  suffixe=${s2:$pos_caract}
fi
# ++++++++++++++++++++++ fin Correctif Exception ++++++++++++++++++++++++++++++++


fin=000                   # Utilisez au moins 3 zéro pour terminer.


soun=$prefixe$suffixe$fin  # Terminez avec des zéro.

LONGUEURMAX=4              # Tronquer un maximum de 4 caractères
soundex=${soun:0:$LONGUEURMAX}

echo "Soundex = $soundex"

echo

#  Le code soundex est une méthode d'indexage et de classification de noms
```

```
#+ en les groupant avec ceux qui sonnent de le même façon.
#  Le code soundex pour un nom donné est la première lettre de ce nom, suivi par
#+ un code calculé sur trois chiffres.
#  Des noms similaires devraient avoir les mêmes codes soundex

#   Exemples:
#   Smith et Smythe ont tous les deux le soundex "S-530"
#   Harrison = H-625
#   Hargison = H-622
#   Harriman = H-655

#  Ceci fonctionne assez bien en pratique mais il existe quelques anomalies.
#
#
#  Certaines agences du gouvernement U.S. utilisent soundex, comme le font les
#  généalogistes.
#
#  Pour plus d'informations, voir
#+ "National Archives and Records Administration home page",
#+ http://www.nara.gov/genealogy/soundex/soundex.html



# Exercice:
# --------
# Simplifier la section "Correctif Exception" de ce script.

exit 0
```

Exemple A.10. « *life* : Jeu de la Vie »

```
#!/bin/bash
# life.sh: "Life in the Slow Lane"
# Version 2: Corrigé par Daniel Albers
#+           pour permettre d'avoir en entrée des grilles non carrées.

# ##################################################################### #
# Ce script est la version Bash du "Jeu de la vie" de John Conway.      #
# "Life" est une implémentation simple d'automatisme cellulaire.        #
# --------------------------------------------------------------------- #
# Sur un tableau rectangulaire, chaque "cellule" sera soit "vivante"    #
# soit "morte". On désignera une cellule vivante avec un point et une   #
# cellule morte avec une espace.                                        #
#  Nous commençons avec un tableau composé aléatoirement de points et   #
#+ d'espaces. Ce sera la génération de départ, "génération 0".         #
# Déterminez chaque génération successive avec les règles suivantes :   #
# 1) Chaque cellule a huit voisins, les cellules voisines (gauche,      #
#+   droite, haut, bas ainsi que les quatre diagonales.                 #
#                                                                       #
#                      123                                              #
#                      4*5       L'étoile est la cellule en question.   #
#                      678                                              #
#                                                                       #
# 2) Une cellule vivante avec deux ou trois voisins vivants reste       #
#+   vivante.                                                           #
SURVIE=2                                                                #
# 3) Une cellule morte avec trois cellules vivantes devient vivante     #
#+   (une "naissance").                                                 #
NAISSANCE=3                                                             #
# 4) Tous les autres cas concerne une cellule morte pour la prochaine   #
#+   génération.                                                        #
# ##################################################################### #


fichier_de_depart=gen0   # Lit la génération de départ à partir du fichier "gen0".
                         #  Par défaut, si aucun autre fichier n'est spécifié à
```

```
                          #+ l'appel de ce script.
                          #
if [ -n "$1" ]            # Spécifie un autre fichier "génération 0".
then
  fichier_de_depart="$1"
fi


########################################################
#  Annule le script si fichier_de_depart non spécifié
#+ et
#+ gen0 non présent.

E_PASDEFICHIERDEPART=68

if [ ! -e "$fichier_de_depart" ]
then
  echo "Fichier de départ \""$fichier_de_depart"\" manquant !"
  exit $E_PASDEFICHIERDEPART
fi
########################################################

VIVANT1=.
MORT1=_
        # Représente des cellules vivantes et "mortes" dans le fichier de départ.

#  ---------------------------------------------------------- #
#  Ce script utilise un tableau 10 sur 10 (pourrait être augmenté
#+ mais une grande grille ralentirait de beaucoup l'exécution).
LIGNES=10
COLONNES=10
#  Modifiez ces deux variables pour correspondre à la taille
#+ de la grille, si nécessaire.
#  ---------------------------------------------------------- #

GENERATIONS=10          #  Nombre de générations pour le cycle.
                        #  Ajustez-le en l'augmentant si vous en avez le temps.

AUCUNE_VIVANTE=80       #  Code de sortie en cas de sortie prématurée,
                        #+ si aucune cellule n'est vivante.
VRAI=0
FAUX=1
VIVANTE=0
MORTE=1

avar=                   #  Global; détient la génération actuelle.
generation=0            # Initialise le compteur des générations.

# ==================================================================


let "cellules = $LIGNES * $COLONNES"
                        # Nombre de cellules.

declare -a initial      # Tableaux contenant les "cellules".
declare -a current

affiche ()
{

alive=0                 # Nombre de cellules "vivantes" à un moment donné.
                        # Initialement à zéro.

declare -a tab
tab=( `echo "$1"` )     # Argument convertit en tableau.

nombre_element=${#tab[*]}

local i
```

```
local verifligne

for ((i=0; i<$nombre_element; i++))
do

# Insère un saut de ligne à la fin de chaque ligne.
  let "verifligne = $i % COLONNES"
  if [ "$verifligne" -eq 0 ]
  then
    echo                # Saut de ligne.
    echo -n "      "    # Indentation.
  fi

  cellule=${tab[i]}

  if [ "$cellule" = . ]
  then
    let "vivante += 1"
  fi

  echo -n "$cellule" | sed -e 's/_/ /g'
  # Affiche le tableau et modifie les tirets bas en espaces.
done

return

}

EstValide ()                            # Teste si les coordonnées sont valides.
{

  if [ -z "$1"  -o -z "$2" ]          # Manque-t'il des arguments requis ?
  then
    return $FAUX
  fi

local ligne
local limite_basse=0                   # Désactive les coordonnées négatives.
local limite_haute
local gauche
local droite

let "limite_haute = $LIGNES * $COLONNES - 1" # Nombre total de cellules.


if [ "$1" -lt "$limite_basse" -o "$1" -gt "$limite_haute" ]
then
  return $FAUX                       # En dehors des limites.
fi

ligne=$2
let "gauche = $ligne * $COLONNES"            # Limite gauche.
let "droite = $gauche + $COLONNES - 1"       # Limite droite.

if [ "$1" -lt "$gauche" -o "$1" -gt "$droite" ]
then
  return $FAUX                       # En dehors des limites.
fi

return $VRAI                          # Coordonnées valides.

}


EstVivante ()           # Teste si la cellule est vivante.
                        #  Prend un tableau, un numéro de cellule et un état de
                        #+ cellule comme arguments.
{
  ObtientNombre "$1" $2 # Récupère le nombre de cellules vivantes dans le voisinage.
```

```
  local voisinage=$?


  if [ "$voisinage" -eq "$NAISSANCE" ]  # Vivante dans tous les cas.
  then
    return $VIVANTE
  fi

  if [ "$3" = "." -a "$voisinage" -eq "$SURVIE" ]
  then                  # Vivante uniquement si précédemment vivante.
    return $VIVANTE
  fi

  return $MORTE          # Par défaut.

}


ObtientNombre ()        # Compte le nombre de cellules vivantes dans le
                        # voisinage de la cellule passée en argument.
                        # Deux arguments nécessaires :
                        # $1) tableau contenant les variables
                        # $2) numéro de cellule
{
  local numero_cellule=$2
  local tableau
  local haut
  local centre
  local bas
  local l
  local ligne
  local i
  local t_hau
  local t_cen
  local t_bas
  local total=0
  local LIGNE_NHBD=3

  tableau=( `echo "$1"` )

  let "haut = $numero_cellule - $COLONNES - 1"  #  Initialise le voisinage de la
                                                #+ cellule.
  let "centre = $numero_cellule - 1"
  let "bas = $numero_cellule + $COLONNES - 1"
  let "l = $numero_cellule / $COLONNES"

  for ((i=0; i<$LIGNE_NHBD; i++))     # Parcours de gauche à droite.
  do
    let "t_hau = $haut + $i"
    let "t_cen = $centre + $i"
    let "t_bas = $bas + $i"


    let "ligne = $l"                  # Calcule la ligne centrée du voisinage.
    EstValide $t_cen $ligne           # Position de la cellule valide ?
    if [ $? -eq "$VRAI" ]
    then
      if [ ${tableau[$t_cen]} = "$VIVANT1" ] # Est-elle vivante ?
      then                            # Oui ?
        let "total += 1"              # Incrémenter le total.
      fi
    fi

    let "ligne = $l - 1"              # Compte la ligne du haut.
    EstValide $t_haut $haut
    if [ $? -eq "$VRAI" ]
    then
      if [ ${tableau[$t_haut]} = "$VIVANT1" ]  # Redondance.
      then                                     # Cela peut-il être optimisé ?
```

```
        let "total += 1"
      fi
    fi

    let "ligne = $l + 1"             # Compte la ligne du bas.
    EstValide $t_bas $ligne
    if [ $? -eq "$VRAI" ]
    then
      if [ ${tableau[$t_bas]} = "$VIVANT1" ]
      then
        let "total += 1"
      fi
    fi

  done


  if [ ${tableau[$numero_cellule]} = "$VIVANT1" ]
  then
    let "total -= 1"        #  S'assurer que la valeur de la cellule testée
  fi                        #+ n'est pas elle-même comptée.


  return $total

}

prochaine_gen ()              # Mise à jour du tableau des générations.
{

local tableau
local i=0

tableau=( `echo "$1"` )       # Argument passé converti en tableau.

while [ "$i" -lt "$cellules" ]
do
  EstVivante "$1" $i ${tableau[$i]}  # La cellule est-elle vivante ?
  if [ $? -eq "$VIVANTE" ]
  then                        #  Si elle l'est, alors
    tableau[$i]=.             #+ représente la cellule avec un point.
  else
    tableau[$i]="_"           #  Sinon, avec un tiret bas.
   fi                         #+ (sera transformé plus tard en espace).
  let "i += 1"
done


# let "generation += 1"   # Incrémente le nombre de générations.
# Pourquoi cette ligne a-t'elle été mise en commentaire ?

# Initialise la variable à passer en tant que paramètre à la fonction
# "affiche".
une_var=`echo ${tableau[@]}` # Convertit un tableau en une variable de type chaîne.
affiche "$une_var"           # L'affiche.
echo; echo
echo "Génération $generation  -  $vivante vivante"

if [ "$alive" -eq 0 ]
then
  echo
  echo "Sortie prématurée : aucune cellule encore vivante !"
  exit $AUCUNE_VIVANTE    #  Aucun intérêt à continuer
fi                        #+ si aucune cellule n'est vivante.

}


# ============================================================
```

```
# main ()

# Charge un tableau initial avec un fichier de départ.
initial=( `cat "$fichier_de_depart" | sed -e '/#/d' | tr -d '\n' |\
sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Supprime les lignes contenant le symbole de commentaires '#'.
# Supprime les retours chariot et insère des espaces entre les éléments.

clear          # Efface l'écran.

echo #         Titre
echo "======================"
echo "    $GENERATIONS générations"
echo "           du"
echo "   \"Jeu de la Vie\""
echo "======================"


# -------- Affiche la première génération. --------
Gen0=`echo ${initial[@]}`
affiche "$Gen0"           # Affiche seulement.
echo; echo
echo "Génération $generation  -  $alive vivante"
# ----------------------------------------


let "generation += 1"     # Incrémente le compteur de générations.
echo

# ------- Affiche la deuxième génération. -------
Actuelle=`echo ${initial[@]}`
prochaine_gen "$Actuelle"          # Mise à jour & affichage.
# ----------------------------------------

let "generation += 1"     # Incrémente le compteur de générations.

# ------ Boucle principale pour afficher les générations conséquentes ------
while [ "$generation" -le "$GENERATIONS" ]
do
  Actuelle="$une_var"
  prochaine_gen "$Actuelle"
  let "generation += 1"
done
# ============================================================

echo

exit 0 # FIN

# Le tableau dans ce script a un "problème de bordures".
# Les bordures haute, basse et des côtés avoisinent une absence de cellules mortes.
# Exercice: Modifiez le script pour avoir la grille
# +         de façon à ce que les côtés gauche et droit se touchent,
# +         comme le haut et le bas.
#
# Exercice: Créez un nouveau fichier "gen0" pour ce script.
#           Utilisez une grille 12 x 16, au lieu du 10 x 10 original.
#           Faites les modifications nécessaires dans le script,
#+          de façon à ce qu'il s'exécute avec le fichier modifié.
#
# Exercice: Modifiez ce script de façon à ce qu'il puisse déterminer la taille
#+          de la grille à partir du fichier "gen0" et initialiser toute variable
#+          nécessaire au bon fonctionnement du script.
#           Ceci rend inutile la modification des variables dans le script
#+          suite à un modification de la taille de la grille.
#
# Exercice : Optimisez ce script.
#            Le code est répétitif et redondant,
```

```
#+           par exemple aux lignes 335-336.
```

Exemple A.11. Fichier de données pour le *Jeu de la Vie*

```
# gen0
#
# This is an example "generation 0" start-up file for "life.sh".
# ---------------------------------------------------------------
#  The "gen0" file is a 10 x 10 grid using a period (.) for live cells,
#+ and an underscore (_) for dead ones. We cannot simply use spaces
#+ for dead cells in this file because of a peculiarity in Bash arrays.
#  [Exercise for the reader: explain this.]
#
# Lines beginning with a '#' are comments, and the script ignores them.
__.__..___
___._._____
____.___..
_._______.
____._____
..__...___
____._____
___...____
__.._..___
_..___..__
```

+++

Les deux scripts suivants sont de Mark Moraes de l'Université de Toronto. Voir le fichier `Moraes-COPYRIGHT` pour les droits. Ce fichier est inclus dans l'archive tar HTML/source2 du *guide ABS*.

Exemple A.12. *behead*: Supprimer les en-têtes des courriers électroniques et des nouvelles

```
#! /bin/sh
# Supprime l'entête d'un message mail/news jusqu'à la première ligne vide.
# Mark Moraes, Université de Toronto

# ==> Ces commentaires sont ajoutés par l'auteur de ce document.

if [ $# -eq 0 ]; then
# ==> Si pas d'arguments en ligne de commande, alors fonctionne avec un
# ==> fichier redirigé vers stdin.
        sed -e '1,/^$/d' -e '/^[         ]*$/d'
        # --> Supprime les lignes vides et les autres jusqu'à la première
        # --> commençant avec une espace blanche.
else
# ==> Si des arguments sont présents en ligne de commande, alors fonctionne avec
# ==> des fichiers nommés.
        for i do
                sed -e '1,/^$/d' -e '/^[         ]*$/d' $i
                # --> De même.
        done
fi

# ==> Exercice: Ajouter la vérification d'erreurs et d'autres options.
# ==>
# ==> Notez que le petit script sed se réfère à l'exception des arguments
# ==> passés.
# ==> Est-il intéressant de l'embarquer dans une fonction? Pourquoi?
```

Exemple A.13. *ftpget*: Télécharger des fichiers via ftp

```
#! /bin/sh
```

```
# $Id: ftpget.sh,v 1.8 2008-05-10 08:36:14 gleu Exp $
# Script pour réaliser une suite d'actions avec un ftp anonyme. Généralement,
# convertit une liste d'arguments de la ligne de commande en entrée vers ftp.
# ==> Ce script n'est rien de plus qu'un emballage shell autour de "ftp"...
# Simple et rapide - écrit comme compagnon de ftplist
# -h spécifie l'hôte distant (par défaut prep.ai.mit.edu)
# -d spécifie le répertoire distant où se déplacer - vous pouvez spécifier une
# séquence d'options -d - elles seront exécutées chacune leur tour. Si les
# chemins sont relatifs, assurez-vous d'avoir la bonne séquence. Attention aux
# chemins relatifs, il existe bien trop de liens symboliques de nos jours.
# (par défaut, le répertoire distant est le répertoire au moment de la connexion)
# -v active l'option verbeux de ftp et affiche toutes les réponses du serveur
# ftp
# -f fichierdistant[:fichierlocal] récupère le fichier distant et le renomme en
# localfile
# -m modele fait un mget suivant le modèle spécifié. Rappelez-vous de mettre
# entre guillemets les caractères shell.
# -c fait un cd local vers le répertoire spécifié
# Par exemple example,
#         ftpget -h expo.lcs.mit.edu -d contrib -f xplaces.shar:xplaces.sh \
#         -d ../pub/R3/fixes -c ~/fixes -m 'fix*'
# récupèrera xplaces.shar à partir de ~ftp/contrib sur expo.lcs.mit.edu et
# l'enregistrera sous xplaces.sh dans le répertoire actuel, puis obtiendra
# tous les correctifs de ~ftp/pub/R3/fixes et les placera dans le répertoire
# ~/fixes.
# De façon évidente, la séquence des options est importante, car les commandes
# équivalentes sont exécutées par ftp dans le même ordre.
#
# Mark Moraes (moraes@csri.toronto.edu), Feb 1, 1989
#


# ==> Ces commentaires ont été ajoutés par l'auteur de ce document.

# PATH=/local/bin:/usr/ucb:/usr/bin:/bin
# export PATH
# ==> Les deux lignes ci-dessus faisaient parti du script original et étaient
# ==> probablement inutiles

E_MAUVAISARGS=65

FICHIER_TEMPORAIRE=/tmp/ftp.$$
# ==> Crée un fichier temporaire, en utilisant l'identifiant du processus du
# ==> script ($$) pour construire le nom du fichier.

SITE=`domainname`.toronto.edu
# ==> 'domainname' est similaire à 'hostname'
# ==> Ceci pourrait être réécrit en ajoutant un paramètre ce qui rendrait son
# ==> utilisation plus générale.

usage="Usage: $0 [-h hotedistant] [-d repertoiredistant]... [-f
fichierdistant:fichierlocal]... \
                [-c repertoirelocal] [-m modele] [-v]"
optionsftp="-i -n"
verbflag=
set -f                  # So we can use globbing in -m
set x `getopt vh:d:c:m:f: $*`
if [ $? != 0 ]; then
        echo $usage
        exit $E_MAUVAISARGS
fi
shift
trap 'rm -f ${FICHIER_TEMPORAIRE} ; exit' 0 1    2            3     15
# ==>                           Signaux:     HUP  INT (Ctl-C)  QUIT  TERM
# ==> Supprimer FICHIER_TEMPORAIRE dans le cas d'une sortie anormale du script.
echo "user anonymous ${USER-gnu}@${SITE} > ${FICHIER_TEMPORAIRE}"
# ==> Ajout des guillemets (recommandé pour les echo complexes).
echo binary >> ${FICHIER_TEMPORAIRE}
for i in $*   # ==> Analyse les arguments de la ligne de commande.
```

```
do
        case $i in
        -v) verbflag=-v; echo hash >> ${FICHIER_TEMPORAIRE}; shift;;
        -h) hotedistant=$2; shift 2;;
        -d) echo cd $2 >> ${FICHIER_TEMPORAIRE};
            if [ x${verbflag} != x ]; then
                echo pwd >> ${FICHIER_TEMPORAIRE};
            fi;
            shift 2;;
        -c) echo lcd $2 >> ${FICHIER_TEMPORAIRE}; shift 2;;
        -m) echo mget "$2" >> ${FICHIER_TEMPORAIRE}; shift 2;;
        -f) f1=`expr "$2" : "\([^:]*\).*"`; f2=`expr "$2" : "[^:]*:\(.*\)"`;
            echo get ${f1} ${f2} >> ${FICHIER_TEMPORAIRE}; shift 2;;
        --) shift; break;;
        esac
# ==> 'lcd' et 'mget' sont des commandes ftp. Voir "man ftp"...
done
if [ $# -ne 0 ]; then
        echo $usage
        exit $E_MAUVAISARGS
        # ==> Modifié de l'"exit 2" pour se conformer avec le standard du style.
fi
if [ x${verbflag} != x ]; then
        optionsftp="${optionsftp} -v"
fi
if [ x${hotedistant} = x ]; then
        hotedistant=prep.ai.mit.edu
        # ==> À modifier pour utiliser votre site ftp favori.
fi
echo quit >> ${FICHIER_TEMPORAIRE}
# ==> Toutes les commandes sont sauvegardées dans fichier_temporaire.

ftp ${optionsftp} ${hotedistant} < ${FICHIER_TEMPORAIRE}
# ==> Maintenant, exécution par ftp de toutes les commandes contenues dans le
# ==> fichier fichier_temporaire.

rm -f ${FICHIER_TEMPORAIRE}
# ==> Enfin, fichier_temporaire est supprimé (vous pouvez souhaiter le copier
# ==> dans un journal).


# ==> Exercices:
# ==> ---------
# ==> 1) Ajouter une vérification d'erreurs.
# ==> 2) Ajouter des tas de trucs.
```

+

Antek Sawicki a contribué avec le script suivant, qui fait une utilisation très intelligente des opérateurs de substitution de paramètres discutés dans la Section 9.3, « Substitution de paramètres ».

Exemple A.14. *password*: Générer des mots de passe aléatoires de 8 caractères

```
#!/bin/bash
#  Pourrait nécessiter d'être appelé avec un #!/bin/bash2 sur les anciennes
#+ machines.
#
#  Générateur de mots de passe aléatoires pour Bash 2.x +
#+ par Antek Sawicki <tenox@tenox.tc>,
#  qui a généreusement permis à l'auteur du guide ABS de l'utiliser ici.
#
# ==> Commentaires ajoutés par l'auteur du document ==>


MATRICE="0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
# ==> Les mots de passe seront constitués de caractères alphanumériques.
LONGUEUR="8"
```

```
# ==> Modification possible de 'LONGUEUR' pour des mots de passe plus longs.


while [ "${n:=1}" -le "$LONGUEUR" ]
# ==> Rappelez-vous que := est l'opérateur de "substitution par défaut".
# ==> Donc, si 'n' n'a pas été initialisé, l'initialiser à 1.
do
        PASS="$PASS${MATRICE:$(($RANDOM%${#MATRICE})):1}"
        # ==> Très intelligent, pratiquement trop astucieux.

        # ==> Commençons par le plus intégré...
        # ==> ${#MATRICE} renvoie la longueur du tableau MATRICE.

        # ==> $RANDOM%${#MATRICE} renvoie un nombre aléatoire entre 1 et la
        # ==> longueur de MATRICE - 1.

        # ==> ${MATRICE:$(($RANDOM%${#MATRICE})):1}
        # ==> renvoie l'expansion de MATRICE à une position aléatoire, par
        # ==> longueur 1.
        # ==> Voir la substitution de paramètres {var:pos:len}, section 3.3.1
        # ==> et les exemples suivants.

        # ==> PASS=... copie simplement ce résultat dans PASS (concaténation).

        # ==> Pour mieux visualiser ceci, décommentez la ligne suivante
        # ==>             echo "$PASS"
        # ==> pour voir la construction de PASS, un caractère à la fois,
        # ==> à chaque itération de la boucle.

        let n+=1
        # ==> Incrémentez 'n' pour le prochain tour.
done

echo "$PASS"      # ==> Ou, redirigez le fichier, comme voulu.

exit 0
```

+

James R. Van Zandt a contribué avec ce script, qui utilise les tubes nommés et, ce sont ses mots, « really exercises quoting and escaping ».

Exemple A.15. *fifo*: Faire des sauvegardes journalières, en utilisant des tubes nommés

```
#!/bin/bash
# ==> Script de James R. Van Zandt, et utilisé ici avec sa permission.

# ==> Commentaires ajoutés par l'auteur de ce document.


  ICI=`uname -n`    # ==> nom d'hôte
  LA_BAS=bilbo
  echo "début de la sauvegarde distante vers $LA_BAS à `date +%r`"
  # ==> `date +%r` renvoie l'heure en un format sur 12 heures, par exempe
  # ==> "08:08:34 PM".

  #  Assurez-vous que /pipe est réellement un tube et non pas un fichier
  #+ standard.
  rm -rf /tube
  mkfifo /tube       # ==> Crée un fichier "tube nommé", nommé "/tube".

  # ==> 'su xyz' lance les commandes en tant qu'utilisateur "xyz".
  # ==> 'ssh' appele le shell sécurisé (client de connexion à distance).
  su xyz -c "ssh $LA_BAS \"cat > /home/xyz/sauve/${ICI}-jour.tar.gz\" < /tube"&
  cd /
  tar -czf - bin boot dev etc home info lib man root sbin share usr var > /tube
  # ==> Utilise un tube nommé, /tube, pour communiquer entre processus:
```

```
  # ==> 'tar/gzip' écrit dans le tube et 'ssh' lit /tube.

  #  ==> Le résultat final est que cela sauvegarde les répertoires principaux;
  #+ ==> à partir de /.

  # ==>  Quels sont les avantages d'un "tube nommé" dans cette situation,
  # ==>+ en opposition avec le "tube anonyme", avec |?
  # ==> Est-ce qu'un tube anonyme pourrait fonctionner ici?

  # ==>  Est-il nécessaire de supprimer le tube avant de sortir du script ?
  # ==>  Comment le faire ?

  exit 0
```

+

Stéphane Chazelas a utilisé avec le script suivant pour démontrer la génération de nombres premiers sans tableaux.

Exemple A.16. *primes*: Générer des nombres premiers en utilisant l'opérateur modulo

```
#!/bin/bash
# primes.sh: Génère des nombres premiers, sans utiliser des tableaux.
# Script contribué par Stephane Chazelas.

#  Il n'utilise *pas* l'algorithme classique du crible d'Ératosthène,
#+ mais utilise à la place la méthode plus intuitive de test de chaque nombre
#+ candidat pour les facteurs (diviseurs), en utilisant l'opérateur modulo "%".


LIMITE=1000                   # Premiers de 2 à 1000

Premiers()
{
 (( n = $1 + 1 ))             # Va au prochain entier.
 shift                        # Prochain paramètre dans la liste.
#  echo "_n=$n i=$i_"

 if (( n == LIMITE ))
 then echo $*
 return
 fi

 for i; do                    #  "i" est initialisé à "@", les précédentes
                              #+ valeurs de $n.
#   echo "-n=$n i=$i-"
   (( i * i > n )) && break   # Optimisation.
   (( n % i )) && continue    #  Passe les non premiers en utilisant l'opérateur
                              #+ modulo.
   Premiers $n $@             # Récursion à l'intérieur de la boucle.
   return
   done

   Premiers $n $@ $n          # Récursion à l'extérieur de la boucle.
                              #  Accumule successivement les paramètres de
                              #+ position.
                              # "$@" est la liste des premiers accumulés.
}

Premiers 1

exit  $?  # Envoyer la sortie du script à 'fmt' pour un affichage plus joli.

# Décommentez les lignes 16 et 24 pour vous aider à comprendre ce qui se passe.

#  Comparez la vitesse de cet algorithme de génération des nombres premiers avec
#+ celui de "Sieve of Eratosthenes" (ex68.sh).
```

```
#  Exercice: Réécrivez ce script sans récursion, pour une exécution plus rapide.
```

+

C'est la version de Rick Boivie du script de Jordi Sanfeliu, qui a donné sa permission pour utiliser son script élégant sur les *arborescences*.

Exemple A.17. *tree*: Afficher l'arborescence d'un répertoire

```
#!/bin/sh
# tree.sh

#  Écrit par Rick Boivie.
#  Utilisé avec sa permission.
#  Ceci est une version revue et simplifiée d'un script
#+ par Jordi Sanfeliu (et corrigée par Ian Kjos).
#  Ce script remplace la version précédente utilisée dans
#+ les précédentes versions du Guide d'écriture avancé de scripts Bash.

# ==> Commentaires ajoutés par l'auteur de ce document.


search () {
   for dir in `echo *`
   # ==> `echo *` affiche tous les fichiers du répertoire actuel sans retour à
   # ==> la ligne.
   # ==> Même effet que     for dir in *
   # ==> mais "dir in `echo *`" ne gère pas les noms de fichiers comprenant des
   # ==> espaces blancs.
   do
      if [ -d "$dir" ] ; then   # ==> S'il s'agit d'un répertoire (-d)...
         zz=0   # ==> Variable temporaire, pour garder trace du niveau du
                # ==> répertoire.
         while [ $zz != $1 ]    # Conserve la trace de la boucle interne.
         do
            echo -n "|   "    # ==> Affiche le symbole du connecteur vertical
                              # ==> avec 2 espaces mais pas de retour à la ligne
                              # ==> pour l'indentation.
            zz=`expr $zz + 1` # ==> Incrémente zz.
         done

         if [ -L "$dir" ] ; then   # ==> Si le répertoire est un lien symbolique...
            echo "+---$dir" `ls -l $dir | sed 's/^.*'$dir' //'`
            # ==> Affiche le connecteur horizontal et affiche le nom du
            # ==> répertoire mais...
            # ==> supprime la partie date/heure des longues listes.
         else
            echo "+---$dir"      # ==> Affiche le symbole du connecteur
                                 # ==> horizontal et le nom du répertoire.
            numdirs=`expr $numdirs + 1` # ==> Incrémente le compteur de répertoire.
            if cd "$dir" ; then # ==> S'il peut se déplacer dans le sous-répertoire...
              search `expr $1 + 1` # avec la récursivité ;-)
              # ==> La fonction s'appelle elle-même.
              cd ..
            fi
         fi
      fi
   done
}

if [ $# != 0 ] ; then
  cd $1 # se déplace au répertoire indiqué.
  #else # reste dans le répertoire actuel.
fi
```

```
echo "Répertoire initial = `pwd`"
numdirs=0

search 0
echo "Nombre total de répertoires = $numdirs"

exit 0
```

+

La version de Patsie du script *tree*.

Exemple A.18. *tree2* : autre script tree

```
#!/bin/bash
# tree2.sh

# Lightly modified/reformatted by ABS Guide author.
# Included in ABS Guide with permission of script author (thanks!).

## Recursive file/dirsize checking script, by Patsie
##
## This script builds a list of files/directories and their size (du -akx)
## and processes this list to a human readable tree shape
## The 'du -akx' is only as good as the permissions the owner has.
## So preferably run as root* to get the best results, or use only on
## directories for which you have read permissions. Anything you can't
## read is not in the list.

#* ABS Guide author advises caution when running scripts as root!


##########  THIS IS CONFIGURABLE  ##########

TOP=5                   # Top 5 biggest (sub)directories.
MAXRECURS=5             # Max 5 subdirectories/recursions deep.
E_BL=80                 # Blank line already returned.
E_DIR=81                # Directory not specified.


##########  DON'T CHANGE ANYTHING BELOW THIS LINE  ##########

PID=$$                              # Our own process ID.
SELF=`basename $0`                  # Our own program name.
TMP="/tmp/${SELF}.${PID}.tmp"       # Temporary 'du' result.

# Convert number to dotted thousand.
function dot { echo "            $*" |
               sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta' |
               tail -c 12; }

# Usage: tree &lt;recursion&gt; &lt;indent prefix&gt; &lt;min size&gt;
&lt;directory&gt;
function tree {
  recurs="$1"           # How deep nested are we?
  prefix="$2"           # What do we display before file/dirname?
  minsize="$3"          # What is the minumum file/dirsize?
  dirname="$4"          # Which directory are we checking?

# Get ($TOP) biggest subdirs/subfiles from TMP file.
  LIST=`egrep "[[:space:]]${dirname}/[^/]*$" "$TMP" |
        awk '{if($1>'$minsize') print;}' | sort -nr | head -$TOP`
  [ -z "$LIST" ] && return        # Empty list, then go back.

  cnt=0
  num=`echo "$LIST" | wc -l`      # How many entries in the list.

  ## Main loop
```

```
  echo "$LIST" | while read size name; do
    ((cnt+=1))                    # Count entry number.
    bname=`basename "$name"`      # We only need a basename of the entry.
    [ -d "$name" ] && bname="$bname/"
                                  # If it's a directory, append a slash.
    echo "`dot $size`$prefix +-$bname"
                                  # Display the result.
    #  Call ourself recursively if it's a directory
    #+ and we're not nested too deep ($MAXRECURS).
    #  The recursion goes up: $((recurs+1))
    #  The prefix gets a space if it's the last entry,
    #+ or a pipe if there are more entries.
    #  The minimum file/dirsize becomes
    #+ a tenth of his parent: $((size/10)).
    # Last argument is the full directory name to check.
    if [ -d "$name" -a $recurs -lt $MAXRECURS ]; then
      [ $cnt -lt $num ] \
        || (tree $((recurs+1)) "$prefix  " $((size/10)) "$name") \
        && (tree $((recurs+1)) "$prefix |" $((size/10)) "$name")
    fi
  done

  [ $? -eq 0 ] && echo "           $prefix"
  # Every time we jump back add a 'blank' line.
  return $E_BL
  # We return 80 to tell we added a blank line already.
}

###                ###
###  main program  ###
###                ###

rootdir="$@"
[ -d "$rootdir" ] ||
  { echo "$SELF: Usage: $SELF &lt;directory&gt;" &gt;&2; exit $E_DIR; }
  # We should be called with a directory name.

echo "Building inventory list, please wait ..."
     # Show "please wait" message.
du -akx "$rootdir" 1>"$TMP" 2>/dev/null
     # Build a temporary list of all files/dirs and their size.
size=`tail -1 "$TMP" | awk '{print $1}'`
     # What is our rootdirectory's size?
echo "`dot $size` $rootdir"
     # Display rootdirectory's entry.
tree 0 "" 0 "$rootdir"
     # Display the tree below our rootdirectory.

rm "$TMP" 2>/dev/null
     # Clean up TMP file.

exit $?
```

Noah Friedman a donné sa permission pour utiliser son script contenant des *fonctions sur les chaînes de caractères*, qui reproduit les fonctions de manipulations de la bibliothèque *C* string.

Exemple A.19. *string*: Manipuler les chaînes de caractères comme en C

```
#!/bin/bash

# string.bash --- bash emulation of string(3) library routines
# Author: Noah Friedman &lt;friedman@prep.ai.mit.edu&gt;
# ==>     Used with his kind permission in this document.
# Created: 1992-07-01
# Last modified: 1993-09-29
# Public domain
```

```
# Conversion to bash v2 syntax done by Chet Ramey

# Commentary:
# Code:

#:docstring strcat:
# Usage: strcat s1 s2
#
# Strcat appends the value of variable s2 to variable s1.
#
# Example:
#    a="foo"
#    b="bar"
#    strcat a b
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload   ==> Autoloading of function commented out.
function strcat ()
{
    local s1_val s2_val

    s1_val=${!1}                        # indirect variable expansion
    s2_val=${!2}
    eval "$1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strncat:
# Usage: strncat s1 s2 $n
#
# Line strcat, but strncat appends a maximum of n characters from the value
# of variable s2.  It copies fewer if the value of variabl s2 is shorter
# than n characters.  Echoes result on stdout.
#
# Example:
#    a=foo
#    b=barbaz
#    strncat a b 3
#    echo $a
#    => foobar
#
#:end docstring:

###;;;autoload
function strncat ()
{
    local s1="$1"
    local s2="$2"
    local -i n="$3"
    local s1_val s2_val

    s1_val=${!s1}                       # ==> indirect variable expansion
    s2_val=${!s2}

    if [ ${#s2_val} -gt ${n} ]; then
       s2_val=${s2_val:0:$n}            # ==> substring extraction
    fi

    eval "$s1"=\'"${s1_val}${s2_val}"\'
    # ==> eval $1='${s1_val}${s2_val}' avoids problems,
    # ==> if one of the variables contains a single quote.
}

#:docstring strcmp:
# Usage: strcmp $s1 $s2
```

```
#
# Strcmp compares its arguments and returns an integer less than, equal to,
# or greater than zero, depending on whether string s1 is lexicographically
# less than, equal to, or greater than string s2.
#:end docstring:

###;;;autoload
function strcmp ()
{
    [ "$1" = "$2" ] && return 0

    [ "${1}" '<' "${2}" ] > /dev/null && return -1

    return 1
}

#:docstring strncmp:
# Usage: strncmp $s1 $s2 $n
#
# Like strcmp, but makes the comparison by examining a maximum of n
# characters (n less than or equal to zero yields equality).
#:end docstring:

###;;;autoload
function strncmp ()
{
    if [ -z "${3}" -o "${3}" -le "0" ]; then
       return 0
    fi

    if [ ${3} -ge ${#1} -a ${3} -ge ${#2} ]; then
       strcmp "$1" "$2"
       return $?
    else
       s1=${1:0:$3}
       s2=${2:0:$3}
       strcmp $s1 $s2
       return $?
    fi
}

#:docstring strlen:
# Usage: strlen s
#
# Strlen returns the number of characters in string literal s.
#:end docstring:

###;;;autoload
function strlen ()
{
    eval echo "\${#${1}}"
    # ==> Returns the length of the value of the variable
    # ==> whose name is passed as an argument.
}

#:docstring strspn:
# Usage: strspn $s1 $s2
#
# Strspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters from string s2.
#:end docstring:

###;;;autoload
function strspn ()
{
    # Unsetting IFS allows whitespace to be handled as normal chars.
    local IFS=
    local result="${1%%[!${2}]*}"
```

```
    echo ${#result}
}

#:docstring strcspn:
# Usage: strcspn $s1 $s2
#
# Strcspn returns the length of the maximum initial segment of string s1,
# which consists entirely of characters not from string s2.
#:end docstring:

###;;;autoload
function strcspn ()
{
    # Unsetting IFS allows whitspace to be handled as normal chars.
    local IFS=
    local result="${1%%[${2}]*}"

    echo ${#result}
}

#:docstring strstr:
# Usage: strstr s1 s2
#
# Strstr echoes a substring starting at the first occurrence of string s2 in
# string s1, or nothing if s2 does not occur in the string.  If s2 points to
# a string of zero length, strstr echoes s1.
#:end docstring:

###;;;autoload
function strstr ()
{
    # if s2 points to a string of zero length, strstr echoes s1
    [ ${#2} -eq 0 ] && { echo "$1" ; return 0; }

    # strstr echoes nothing if s2 does not occur in s1
    case "$1" in
    *$2*) ;;
    *) return 1;;
    esac

    # use the pattern matching code to strip off the match and everything
    # following it
    first=${1/$2*/}

    # then strip off the first unmatched portion of the string
    echo "${1##$first}"
}

#:docstring strtok:
# Usage: strtok s1 s2
#
# Strtok considers the string s1 to consist of a sequence of zero or more
# text tokens separated by spans of one or more characters from the
# separator string s2.  The first call (with a non-empty string s1
# specified) echoes a string consisting of the first token on stdout. The
# function keeps track of its position in the string s1 between separate
# calls, so that subsequent calls made with the first argument an empty
# string will work through the string immediately following that token.  In
# this way subsequent calls will work through the string s1 until no tokens
# remain.  The separator string s2 may be different from call to call.
# When no token remains in s1, an empty value is echoed on stdout.
#:end docstring:

###;;;autoload
function strtok ()
{
 :
}
```

```
#:docstring strtrunc:
# Usage: strtrunc $n $s1 {$s2} {$...}
#
# Used by many functions like strncmp to truncate arguments for comparison.
# Echoes the first n characters of each string s1 s2 ... on stdout.
#:end docstring:

###;;;autoload
function strtrunc ()
{
    n=$1 ; shift
    for z; do
        echo "${z:0:$n}"
    done
}

# provide string

# string.bash ends here


# ========================================================================== #
# ==> Everything below here added by the document author.

# ==> Suggested use of this script is to delete everything below here,
# ==> and "source" this file into your own scripts.

# strcat
string0=one
string1=two
echo
echo "Testing \"strcat\" function:"
echo "Original \"string0\" = $string0"
echo "\"string1\" = $string1"
strcat string0 string1
echo "New \"string0\" = $string0"
echo

# strlen
echo
echo "Testing \"strlen\" function:"
str=123456789
echo "\"str\" = $str"
echo -n "Length of \"str\" = "
strlen str
echo



# Exercise:
# --------
# Add code to test all the other string functions above.


exit 0
```

L'exemple de tableaux complexes par Michael Zick utilise la commande de vérification de sommes md5sum pour coder les informations sur le répertoire.

Exemple A.20. Informations sur un répertoire

```
#! /bin/bash
# directory-info.sh
# Analyse et affiche des informations sur le répertoire.

# NOTE: Modification des lignes 273 et 353 suivant le fichier "README".
```

```
# Michael Zick est l'auteur de ce script.
# Utilisé ici avec son autorisation.

# Contrôles
# Si outrepassé par les arguments de la commande, ils doivent être dans l'ordre:
#   Arg1: "Descripteur du répertoire"
#   Arg2: "Chemins à exclure"
#   Arg3: "Répertoires à exclure"
#
# Les variables d'environnement outrepassent les valeurs par défaut.
# Les arguments de la commande outrepassent les variables d'environnement.

# Emplacement par défaut du contenu des descripteurs de fichiers.
MD5UCFS=${1:-${MD5UCFS:-'/tmpfs/ucfs'}}

# Répertoires à exclure
declare -a \
  CHEMINS_A_EXCLURE=${2:-${CHEMINS_A_EXCLURE:-'(/proc /dev /devfs /tmpfs)'}}

# Répertoires à exclure
declare -a \
  REPERTOIRES_A_EXCLURE=${3:-${REPERTOIRES_A_EXCLURE:-'(ucfs lost+found tmp wtmp)'}}

# Fichiers à exclure
declare -a \
  FICHIERS_A_EXCLURE=${3:-${FICHIERS_A_EXCLURE:-'(core "Nom avec des espaces")'}}


# Document intégré utilisé comme bloc de commentaires.
: &lt;&lt;LSfieldsDoc
# # # Affiche les informations sur les répertoires du système de fichiers # # #
#
# AfficheRepertoire "FileGlob" "Field-Array-Name"
# ou
# AfficheRepertoire -of "FileGlob" "Field-Array-Filename"
# '-of' signifiant 'sortie vers fichier'
# # # # #

Description du format de la chaîne : ls (GNU fileutils) version 4.0.36

Produit une ligne (ou plus) formattée :
inode droits    liens propriétaire groupe ...
32736 -rw-------    1 mszick   mszick

taille jour mois date hh:mm:ss année chemin
2756608 Sun Apr 20 08:53:06 2003 /home/mszick/core

Sauf, s'il est formatté :
inode  droits    liens propriétaire groupe ...
266705 crw-rw----    1    root  uucp

majeur mineur jour mois date hh:mm:ss année chemin
4,  68 Sun Apr 20 09:27:33 2003 /dev/ttyS4
NOTE: cette virgule bizarre après le nombre majeur

NOTE: le 'chemin' pourrait avoir plusieurs champs :
/home/mszick/core
/proc/982/fd/0 -> /dev/null
/proc/982/fd/1 -> /home/mszick/.xsession-errors
/proc/982/fd/13 -> /tmp/tmpfZVVOCs (deleted)
/proc/982/fd/7 -> /tmp/kde-mszick/ksycoca
/proc/982/fd/8 -> socket:[11586]
/proc/982/fd/9 -> pipe:[11588]

Si ce n'est pas suffisant pour que votre analyseur continue à deviner,
soit une soit les deux parties du chemin peuvent être relatives :
../Built-Shared -> Built-Static
../linux-2.4.20.tar.bz2 -> ../../../SRCS/linux-2.4.20.tar.bz2
```

```
Le premier caractère du champ des droits (sur 11 (10 ?) caractères) :
's' Socket
'd' Répertoire
'b' Périphérique bloc
'c' Périphérique caractère
'l' Lien symbolique
NOTE: Les liens non symboliques ne sont pas identifiés - testés pour des numéros
d'inodes identiques sur le même système de fichiers.
Toutes les informations sur les fichiers liés sont partagées sauf le nom et
l'emplacement.
NOTE: Un "lien" est connu comme un "alias" sur certains systèmes.
'-' fichier sans distinction.

Suivi par trois groupes de lettres pour l'utilisateur, le groupe et les autres.
Caractère 1: '-' non lisible; 'r' lisible
Caractère 2: '-' pas d'écriture; 'w' écriture (writable)
Caractère 3, utilisateur et groupe: Combine l'éxécution et un spécial
'-' non exécutable, non spécial
'x' exécutable, non spécial
's' exécutable, spécial
'S' non exécutable, spécial
Caractère 3, autres: Combine l'éxécution et le sticky (tacky?)
'-' non éxécutable, non tacky
'x' exécutable, non tacky
't' exécutable, tacky
'T' non exécutable, tacky

Suivi par un indicateur d'accès
Non testé, il pourrait être le onzième caractère
ou il pourrait générer un autre champ
' ' Pas d'accès autre
'+' Accès autre
LSfieldsDoc


AfficheRepertoire()
{
        local -a T
        local -i of=0         # Valeur par défaut
# OLD_IFS=$IFS         # Utilise la variable BASH par défaut ' \t\n'

        case "$#" in
        3) case "$1" in
                -of) of=1 ; shift ;;
                 * ) return 1 ;;
                esac ;;
        2) : ;;         # L'instruction "continue" du pauvre
        *) return 1 ;;
        esac

        # NOTE: la commande (ls) N'est PAS entre guillemets (")
        T=( $(ls --inode --ignore-backups --almost-all --directory \
        --full-time --color=none --time=status --sort=none \
        --format=long $1) )

        case $of in
        #  Affecte T en retour pour le tableau dont le nom a été passé
        #+ à $2
                0) eval $2=\( \"\$\{T\[@\]\}\" \) ;;
        # Ecrit T dans le nom du fichier passé à $2
                1) echo "${T[@]}" > "$2" ;;
        esac
        return 0
   }

# # # # # Est-ce que cette chaîne est un nombre légal ? # # # # # #
#
# EstNombre "Var"
# # # # # Il doit y avoir un meilleur moyen, hum...
```

```
EstNombre()
{
        local -i int
        if [ $# -eq 0 ]
        then
                return 1
        else
                (let int=$1)  2>/dev/null
                return $? # Code de sortie du thread créé pour let
        fi
}

# # # Informations sur l'index des répertoires du système de fichiers # # #
#
# AfficheIndex "Field-Array-Name" "Index-Array-Name"
# ou
# AfficheIndex -if Field-Array-Filename Index-Array-Name
# AfficheIndex -of Field-Array-Name Index-Array-Filename
# AfficheIndex -if -of Field-Array-Filename Index-Array-Filename
# # # # #

: &lt;&lt;AfficheIndexDoc
Parcourt un tableau de champs répertoire créé par AfficheRepertoire

Ayant supprimé les retours chariots dans un rapport habituellement ligne par
ligne, construit un index vers l'élement du tableau commençant à chaque ligne.

Chaque ligne obtient deux entrées de l'index, le premier élément de chaque ligne
(inode) et l'élément qui contient le chemin du fichier.

La première paire d'entrée de l'index (Numero-Ligne==0) apporte une
information :
Nom-Tableau-Index[0] : Nombre de "lignes" indexé
Nom-Tableau-Index[1] : Pointeur de la "ligne courante" vers Nom-Tableau-Index

Les paires d'index suivantes (si elles existent) contiennent les index des
éléments dans Nom-Tableau-Champ avec :
Nom-Tableau-Index[Numero-Ligne * 2] : L'élément champ "inode".
NOTE: La distance peut être de +11 ou +12 éléments.
Nom-Tableau-Index[(Numero-Ligne * 2) + 1] : L'élément "chemin".
NOTE: La distance est un nombre variable d'éléments.
La prochaine paire de lignes d'index pour Numero-Ligne+1.
AfficheIndexDoc



AfficheIndex()
{
        local -a LISTE                  # Variable locale du nom de liste
        local -a -i INDEX=( 0 0 ) # Variable locale de l'index à renvoyer
        local -i Lidx Lcpt
        local -i if=0 of=0         # Par défaut

        case "$#" in                  # Test simpliste des options
                0) return 1 ;;
                1) return 1 ;;
                2) : ;;                  # Instruction "continue" du pauvre
                3) case "$1" in
                        -if) if=1 ;;
                        -of) of=1 ;;
                         * ) return 1 ;;
                   esac ; shift ;;
                4) if=1 ; of=1 ; shift ; shift ;;
                *) return 1
        esac

        # Fait une copie locale de liste
        case "$if" in
```

```
                0) eval LISTE=\( \"\$\{$1\[@\]\}\" \) ;;
                1) LISTE=( $(cat $1) ) ;;
        esac

        # "Grok (grope?)" le tableau
        Lcpt=${#LISTE[@]}
        Lidx=0
        until (( Lidx >= Lcpt ))
        do
        if EstNombre ${LISTE[$Lidx]}
        then
                local -i inode nom
                local ft
                inode=Lidx
                local m=${LISTE[$Lidx+2]} # Champ des liens
                ft=${LISTE[$Lidx+1]:0:1}         # Stats rapides
                case $ft in
                b) ((Lidx+=12)) ;;         # Périphérique bloc
                c) ((Lidx+=12)) ;;         # Périphérique caractère
                *) ((Lidx+=11)) ;;         # Le reste
                esac
                nom=Lidx
                case $ft in
                -) ((Lidx+=1)) ;;         # Le plus simple
                b) ((Lidx+=1)) ;;         # Périphérique bloc
                c) ((Lidx+=1)) ;;         # Périphérique caractère
                d) ((Lidx+=1)) ;;         # Encore un autre
                l) ((Lidx+=3)) ;;         # Au MOINS deux autres champs
#  Un peu plus d'élégance ici permettrait de gérer des tubes, des sockets,
#+ des fichiers supprimés - plus tard.
                *) until EstNombre ${LISTE[$Lidx]} || ((Lidx >= Lcpt))
                        do
                                ((Lidx+=1))
                        done
                        ;;                 # Non requis.
                esac
                INDEX[${#INDEX[*]}]=$inode
                INDEX[${#INDEX[*]}]=$nom
                INDEX[0]=${INDEX[0]}+1         # Une "ligne" de plus
# echo "Ligne: ${INDEX[0]} Type: $ft Liens: $m Inode: \
# ${LIST[$inode]} Nom: ${LIST[$name]}"

        else
                ((Lidx+=1))
        fi
        done
        case "$of" in
                0) eval $2=\( \"\$\{INDEX\[@\]\}\" \) ;;
                1) echo "${INDEX[@]}" > "$2" ;;
        esac
        return 0                # Que pourrait'il arriver de mal ?
}

# # # # # Fichier identifié par son contenu # # # # #
#
# DigestFile Nom-Tableau-Entree Nom-Tableau-Digest
# ou
# DigestFile -if NomFichier-EnEntree Nom-Tableau-Digest
# # # # #

# Document intégré utilisé comme bloc de commentaires.
: &lt;&lt;DigestFilesDoc

La clé (no pun intended) vers un Système de Fichiers au Contenu Unifié (UCFS)
permet de distinguer les fichiers du système basés sur leur contenu.
Distinguer des fichiers par leur nom est tellement 20è siècle.

Le contenu se distingue en calculant une somme de contrôle de ce contenu.
Cette version utilise le programme md5sum pour générer une représentation de la
```

```
somme de contrôle 128 bit du contenu.
Il existe une chance pour que deux fichiers ayant des contenus différents
génèrent la même somme de contrôle utilisant md5sum (ou tout autre outil de
calcul de somme de contrôle). Si cela devait devenir un problème, alors
l'utilisation de md5sum peut être remplacée par une signature cryptographique.
Mais jusque là...

La documentation de md5sum précise que la sortie de cette commande affiche
trois champs mais, à la lecture, il apparaît comme deux champs (éléments du
tableau). Ceci se fait par le manque d'espaces blancs entre le second et le
troisième champ. Donc, cette fonction groupe la sortie du md5sum et renvoit :
        [0] Somme de contrôle sur 32 caractères en héxidecimal (nom du
                fichier UCFS)
        [1] Caractère seul : ' ' fichier texte, '*' fichier binaire
        [2] Nom système de fichiers (style 20è siècle)
        Note: Ce nom pourrait être le caractère '-' indiquant la lecture de
        STDIN

DigestFilesDoc



DigestFile()
{
        local if=0         # Par défaut.
        local -a T1 T2

        case "$#" in
        3) case "$1" in
                -if) if=1 ; shift ;;
                 * ) return 1 ;;
                esac ;;
        2) : ;;         # Instruction "continue" du pauvre
        *) return 1 ;;
        esac

        case $if in
        0) eval T1=\( \"\$\{$1\[@\]\}\" \)
           T2=( $(echo ${T1[@]} | md5sum -) )
           ;;
        1) T2=( $(md5sum $1) )
           ;;
        esac

        case ${#T2[@]} in
        0) return 1 ;;
        1) return 1 ;;
        2) case ${T2[1]:0:1} in         # SanScrit-2.0.5
           \*) T2[${#T2[@]}]=${T2[1]:1}
               T2[1]=\*
               ;;
            *) T2[${#T2[@]}]=${T2[1]}
               T2[1]=" "
               ;;
           esac
           ;;
        3) : ;; # Suppose qu'il fonctionne
        *) return 1 ;;
        esac

        local -i len=${#T2[0]}
        if [ $len -ne 32 ] ; then return 1 ; fi
        eval $2=\( \"\$\{T2\[@\]\}\" \)
}

# # # # # Trouve l'emplacement du fichier # # # # #
#
# LocateFile [-l] NomFichier Nom-Tableau-Emplacement
# ou
```

```
# LocateFile [-l] -of NomFichier NomFichier-Tableau-Emplacement
# # # # #

#  L'emplacement d'un fichier correspond à l'identifiant du système de fichiers
#+ et du numéro de l'inode.

# Document intégré comme bloc de commentaire.
: &lt;&lt;StatFieldsDoc
        Basé sur stat, version 2.2
        champs de stat -t et stat -lt
        [0] nom
        [1] Taille totale
                Fichier - nombre d'octets
                Lien symbolique - longueur de la chaîne représentant le chemin
        [2] Nombre de blocs (de 512 octets) alloués
        [3] Type de fichier et droits d'accès (hex)
        [4] ID utilisateur du propriétaire
        [5] ID groupe du propriétaire
        [6] Numéro de périphérique
        [7] Numéro de l'inode
        [8] Nombre de liens
        [9] Type de périphérique (si périphérique d'inode) Majeur
        [10] Type de périphérique (si périphérique d'inode) Mineur
        [11] Heure du dernier accès
                Pourrait être désactivé dans 'mount' avec noatime
                atime des fichiers changés par exec, read, pipe, utime, mknod
                (mmap?)
                atime des répertoires changés par ajout/suppression des fichiers
        [12] Heure de dernière modification
                mtime des fichiers changés par write, truncate, utime, mknod
                mtime des répertoires changés par ajout/suppression des fichiers
        [13] Heure de dernier changement
                ctime reflète le temps de la dernière modification de l'inode
                (propriétaire, groupe, droits, nombre de liens)
-*-*- Pour :
        Code de sortie: 0
        Taille du tableau: 14
        Contenu du tableau
        Elément 0: /home/mszick
        Elément 1: 4096
        Elément 2: 8
        Elément 3: 41e8
        Elément 4: 500
        Elément 5: 500
        Elément 6: 303
        Elément 7: 32385
        Elément 8: 22
        Elément 9: 0
        Elément 10: 0
        Elément 11: 1051221030
        Elément 12: 1051214068
        Elément 13: 1051214068

        Pour un lien de la forme nom_lien -> vrai_nom
        stat -t  nom_lien renvoit des informations sur le lien
        stat -lt nom_lien renvoit des informations sur le vrai fichier

        Champs stat -tf et stat -ltf
        [0] nom
        [1] ID-0?         # Peut-être un jour, mais la structure stat de
        [2] ID-0?         # Linux n'a ni le champ LABEL ni UUID,
                                # actuellement l'information doit provenir
                                # d'utilitaires système spécifiques
        Ceci sera transformé en :
        [1] UUID si possible
        [2] Label du volume si possible
        Note: 'mount -l' renvoit le label et pourrait renvoyer le UUID

        [3] Longueur maximum des noms de fichier
```

```
        [4] Type de système de fichiers
        [5] Nombre total de blocs dans le système de fichiers
        [6] Blocs libres
        [7] Blocs libres pour l'utilisateur non root
        [8] Taille du bloc du système de fichiers
        [9] Nombre total d'inodes
        [10] Inodes libres

-*-*- Per:
        Code de sortie: 0
        Taille du tableau : 11
        Contenu du tableau
        Elément 0: /home/mszick
        Elément 1: 0
        Elément 2: 0
        Elément 3: 255
        Elément 4: ef53
        Elément 5: 2581445
        Elément 6: 2277180
        Elément 7: 2146050
        Elément 8: 4096
        Elément 9: 1311552
        Elément 10: 1276425

StatFieldsDoc


# LocateFile [-l] NomFichier Nom-Tableau-Emplacement
# LocateFile [-l] -of NomFichier Nom-Tableau-Emplacement

LocateFile()
{
        local -a LOC LOC1 LOC2
        local lk="" of=0

        case "$#" in
        0) return 1 ;;
        1) return 1 ;;
        2) : ;;
        *) while (( "$#" > 2 ))
           do
              case "$1" in
               -l) lk=-1 ;;
              -of) of=1 ;;
                *) return 1 ;;
              esac
           shift
           done ;;
        esac

# Plus de Sanscrit-2.0.5
      # LOC1=( $(stat -t $lk $1) )
      # LOC2=( $(stat -tf $lk $1) )
      #  Supprimez le commentaire des deux lignes ci-dessus si le système
      #+ dispose de la commande "stat" installée.
        LOC=( ${LOC1[@]:0:1} ${LOC1[@]:3:11}
              ${LOC2[@]:1:2} ${LOC2[@]:4:1} )

        case "$of" in
                0) eval $2=\( \"\$\{LOC\[@\]\}\" \) ;;
                1) echo "${LOC[@]}" > "$2" ;;
        esac
        return 0
# Ce qui rend comme résultat (si vous êtes chanceux et avez installé "stat")
# -*-*- Descripteur de l'emplacement -*-*-
# Code de sortie : 0
# Taille du tableau : 15
# Contenu du tableau
# Elément 0: /home/mszick         Nom du 20è siècle
```

```
# Elément 1: 41e8                Type et droits
# Elément 2: 500                Utilisateur
# Elément 3: 500                Groupe
# Elément 4: 303                Périphérique
# Elément 5: 32385         Inode
# Elément 6: 22                 Nombre de liens
# Elément 7: 0                Numéro majeur
# Elément 8: 0                Numéro mineur
# Elément 9: 1051224608         Dernier accès
# Elément 10: 1051214068         Dernière modification
# Elément 11: 1051214068         Dernier statut
# Elément 12: 0                UUID (à faire)
# Elément 13: 0                Volume Label (à faire)
# Elément 14: ef53         Type de système de fichiers
}



# Et enfin, voici un code de test

AfficheTableau() # AfficheTableau Nom
{
        local -a Ta

        eval Ta=\( \"\$\{$1\[@\]\}\" \)
        echo
        echo "-*-*- Liste de tableaux -*-*-"
        echo "Taille du tableau $1: ${#Ta[*]}"
        echo "Contenu du tableau $1:"
        for (( i=0 ; i<${#Ta[*]} ; i++ ))
        do
            echo -e "\tElément $i: ${Ta[$i]}"
        done
        return 0
}

declare -a CUR_DIR
# Pour de petits tableaux
AfficheRepertoire "${PWD}" CUR_DIR
AfficheTableau CUR_DIR

declare -a DIR_DIG
DigestFile CUR_DIR DIR_DIG
echo "Le nouveau \"nom\" (somme de contrôle) pour ${CUR_DIR[9]} est ${DIR_DIG[0]}"

declare -a DIR_ENT
# BIG_DIR # Pour de réellement gros tableaux - utilise un fichier temporaire en
          # disque RAM
# BIG-DIR # AfficheRepertoire -of "${CUR_DIR[11]}/*" "/tmpfs/junk2"
AfficheRepertoire "${CUR_DIR[11]}/*" DIR_ENT

declare -a DIR_IDX
# BIG-DIR # AfficheIndex -if "/tmpfs/junk2" DIR_IDX
AfficheIndex DIR_ENT DIR_IDX

declare -a IDX_DIG
# BIG-DIR # DIR_ENT=( $(cat /tmpfs/junk2) )
# BIG-DIR # DigestFile -if /tmpfs/junk2 IDX_DIG
DigestFile DIR_ENT IDX_DIG
# Les petits (devraient) être capable de paralléliser AfficheIndex & DigestFile
# Les grands (devraient) être capable de paralléliser AfficheIndex & DigestFile
# & l'affectation
echo "Le \"nom\" (somme de contrôle) pour le contenu de ${PWD} est ${IDX_DIG[0]}"

declare -a FILE_LOC
LocateFile ${PWD} FILE_LOC
AfficheTableau FILE_LOC

exit 0
```

Stéphane Chazelas montre la programmation objet dans un script Bash.

Exemple A.21. obj-oriented: Bases de données orientées objet

```
#!/bin/bash
# obj-oriented.sh: programmation orientée objet dans un script shell.
# Script par Stephane Chazelas.

#  Note Importante :
#  ---- ----------
#  Si vous exécutez ce script avec une version 3 ou ultérieure de Bash,
#+ remplacez tous les points dans les noms de fonctions avec un
#+ caractère légal, par exemple un tiret bas.


person.new()        # Ressemble à la déclaration d'une classe en C++.
{
  local nom_objet=$1 nom=$2 prenom=$3 datenaissance=$4

  eval "$nom_objet.set_nom() {
          eval \"$nom_objet.get_nom() {
                  echo \$1
                }\"
        }"

  eval "$nom_objet.set_prenom() {
          eval \"$nom_objet.get_prenom() {
                  echo \$1
                }\"
        }"

  eval "$nom_objet.set_datenaissance() {
          eval \"$nom_objet.get_datenaissance() {
            echo \$1
          }\"
          eval \"$nom_objet.show_datenaissance() {
            echo \$(date -d \"1/1/1970 0:0:\$1 GMT\")
          }\"
          eval \"$nom_objet.get_age() {
            echo \$(( (\$(date +%s) - \$1) / 3600 / 24 / 365 ))
          }\"
        }"

  $nom_objet.set_nom $nom
  $nom_objet.set_prenom $prenom
  $nom_objet.set_datenaissance $datenaissance
}

echo

person.new self Bozeman Bozo 101272413
#  Crée une instance de "person.new" (en fait, passe les arguments à la
#+ fonction).

self.get_prenom             #   Bozo
self.get_nom                #   Bozeman
self.get_age                #   28
self.get_datenaissance      #   101272413
self.show_datenaissance     #   Sat Mar 17 20:13:33 MST 1973

echo

#  typeset -f
#+ pour voir les fonctions créées (attention, cela fait défiler la page).

exit 0
```

Mariusz Gniazdowski a contribué avec une bibliothèque de hachage à utiliser dans des scripts.

Exemple A.22. Bibliothèque de fonctions de hachage

```
# Hash:
# Bibliothèque de fonctions de hachage
# Auteur : Mariusz Gniazdowski &lt;mgniazd-at-gmail.com&gt;
# Date : 2005-04-07

# Fonctions rendant l'émulation de hâchage en Bash un peu moins pénible.


#    Limitations:
#  * Seules les variables globales sont supportées.
#  * Chaque instance de hâchage génère une variable globale par valeur.
#  * Les collisions de noms de variables sont possibles
#+   si vous définissez des variables comme __hash__hashname_key
#  * Les clés doivent utilisant des caractères faisant partie du nom d'une variable
Bash
#+   (pas de tirets, points, etc.).
#  * Le hâchage est créé comme une variable :
#    ... hashname_keyname
#    Donc si quelqu'un crée des hâchages ainsi :
#      myhash_ + mykey = myhash__mykey
#      myhash + _mykey = myhash__mykey
#    Alors, il y aura collision.
#    (Ceci ne devrait pas poser un problème majeur.)


Hash_config_varname_prefix=__hash__


# Émule:  hash[key]=value
#
# Paramètres:
# 1 - hash  (hachage)
# 2 - key   (clé)
# 3 - value (valeur
function hash_set {
        eval "${Hash_config_varname_prefix}${1}_${2}=\"${3}\""
}


# Émule:  value=hash[key]
#
# Paramètres:
# 1 - hash
# 2 - key
# 3 - value (nom d'une variable globale à initialiser)
function hash_get_into {
        eval "$3=\"\$${Hash_config_varname_prefix}${1}_${2}\""
}


# Émule:  echo hash[key]
#
# Paramètres:
# 1 - hash
# 2 - key
# 3 - echo params (like -n, for example)
function hash_echo {
        eval "echo $3 \"\$${Hash_config_varname_prefix}${1}_${2}\""
}


# Émule:  hash1[key1]=hash2[key2]
```

```
#
# Paramètres:
# 1 - hash1
# 2 - key1
# 3 - hash2
# 4 - key2
function hash_copy {
        eval "${Hash_config_varname_prefix}${1}_${2}\
=\"\$${Hash_config_varname_prefix}${3}_${4}\""
}


# Émule:  hash[keyN-1]=hash[key2]=...hash[key1]
#
# Copie la première clé au reste des clés.
#
# Paramètres:
# 1 - hash1
# 2 - key1
# 3 - key2
# . . .
# N - keyN
function hash_dup {
        local hashName="$1" keyName="$2"
        shift 2
        until [ ${#} -le 0 ]; do
                eval "${Hash_config_varname_prefix}${hashName}_${1}\
=\"\$${Hash_config_varname_prefix}${hashName}_${keyName}\""
                shift;
        done;
}


# Émule:  unset hash[key]
#
# Paramètres:
# 1 - hash
# 2 - key
function hash_unset {
        eval "unset ${Hash_config_varname_prefix}${1}_${2}"
}


# Emulates something similar to:  ref=&amp;hash[key]
#
# The reference is name of the variable in which value is held.
#
# Paramètres:
# 1 - hash
# 2 - key
# 3 - ref - Nom d'une variable globale à initialiser.
function hash_get_ref_into {
        eval "$3=\"${Hash_config_varname_prefix}${1}_${2}\""
}


# Émule quelque chose de similaire à:  echo &amp;hash[key]
#
# Cette référence est le nom d'une variable dans laquelle est contenue la valeur.
#
# Paramètres:
# 1 - hash
# 2 - key
# 3 - echo params (comme -n par exemple)
function hash_echo_ref {
        eval "echo $3 \"${Hash_config_varname_prefix}${1}_${2}\""
}
```

```
# Émule quelque chose de similaire à:  $$hash[key](param1, param2, ...)
#
# Paramètres:
# 1 - hash
# 2 - key
# 3,4, ... - Paramètres de fonction.
function hash_call {
        local hash key
        hash=$1
        key=$2
        shift 2
        eval "eval \"\$${Hash_config_varname_prefix}${hash}_${key} \\\"\\\$@\\\"\""
}


# Émule quelque chose de similaire à:  isset(hash[key]) ou hash[key]==NULL
#
# Paramètres:
# 1 - hash
# 2 - key
# Renvoit:
# 0 - cette clé existe
# 1 - cette clé n'existe pas
function hash_is_set {
        eval "if [[ \"\${${Hash_config_varname_prefix}${1}_${2}-a}\" = \"a\" &&
                        \"\${${Hash_config_varname_prefix}${1}_${2}-b}\" = \"b\" ]];
then return 1; else return 0; fi"
}


# Émule quelque chose de similaire à:
#   foreach($hash as $key => $value) { fun($key,$value); }
#
# Il est possible d'écrite plusieurs variations de cette fonction.
# Ici, nous utilisons un appel de fonction pour la rendre aussi "générique" que
possible.
#
# Paramètres:
# 1 - hash
# 2 - function name
function hash_foreach {
        local keyname oldIFS="$IFS"
        IFS=' '
        for i in $(eval "echo \${!${Hash_config_varname_prefix}${1}_*}"); do
                keyname=$(eval "echo \${i##${Hash_config_varname_prefix}${1}_}")
                eval "$2 $keyname \"\$$i\""
        done
        IFS="$oldIFS"
}

# NOTE : Sur les lignes 103 et 116, modification de l'arobase.
#        Mais, cela n'a pas d'importance parce qu'il s'agit de lignes de commentaires.
```

Voici un exemple de script utilisant cette bibliothèque de hachage.

Exemple A.23. Coloriser du texte en utilisant les fonctions de hachage

```
#!/bin/bash
# hash-example.sh: Colorisation de texte.
# Auteur : Mariusz Gniazdowski <mgniazd-at-gmail.com>

. Hash.lib      # Chargement de la bibliothèque des fonctions.

hash_set couleurs rouge        "\033[0;31m"
hash_set couleurs bleu         "\033[0;34m"
hash_set couleurs bleu_leger   "\033[1;34m"
```

```
hash_set couleurs rouge_leger  "\033[1;31m"
hash_set couleurs cyan         "\033[0;36m"
hash_set couleurs vert_leger   "\033[1;32m"
hash_set couleurs gris_leger   "\033[0;37m"
hash_set couleurs vert         "\033[0;32m"
hash_set couleurs jaune        "\033[1;33m"
hash_set couleurs violet_leger "\033[1;35m"
hash_set couleurs violet       "\033[0;35m"
hash_set couleurs reset_couleur "\033[0;00m"


# $1 - nom de la clé
# $2 - valeur
essaie_couleurs() {
        echo -en "$2"
        echo "Cette ligne est $1."
}
hash_foreach couleurs essaie_couleurs
hash_echo couleurs reset_couleur -en

echo -e '\nSurchargeons quelques couleurs avec du jaune.\n'
# Il est difficile de lire du texte jaune sur certains terminaux.
hash_dup couleurs jaune rouge vert_leger bleu vert gris_leger cyan
hash_foreach couleurs essaie_couleurs
hash_echo couleurs reset_color -en

echo -e '\nSupprimons-les et essayons couleurs une fois encore...\n'

for i in rouge vert_leger bleu vert gris_leger cyan; do
        hash_unset couleurs $i
done
hash_foreach couleurs essaie_couleurs
hash_echo couleurs reset_couleur -en

hash_set autre texte "Autres exemples..."
hash_echo autre texte
hash_get_into autre txt texte
echo $texte

hash_set autre my_fun essaie_couleurs
hash_call autre my_fun   purple "`hash_echo couleurs violet`"
hash_echo couleurs reset_couleur -en

echo; echo "Retour à la normale ?"; echo

exit $?

#  Sur certains terminaux, les couleurs "légères" sont affichées en gras
#  et finissent par sembler plus sombres que les normales.
#  Pourquoi ?
```

Un exemple illustrant les mécanismes de hachage à partir d'un autre point de vue.

Exemple A.24. Encore plus sur les fonctions de hachage

```
#!/bin/bash
# $Id: ha.sh,v 1.4 2008-05-10 17:43:04 gleu Exp $
# Copyright 2005 Oliver Beckstein
# Sous licence GNU Public
# L'auteur du script a donné le droit de l'inclure dans le guide ABS.
# (Merci !)

#-------------------------------------------------------------
# pseudo hachage basé sur l'expansion des paramètres indirects
# API : accès par les fonctions :
#
```

```
# création du hachage :
#
#      newhash Lovers
#
# ajout des entrées (notez les guillemets pour les espaces)
#
#      addhash Lovers Tristan Isolde
#      addhash Lovers 'Romeo Montague' 'Juliet Capulet'
#
# accès à la valeur par la clé
#
#      gethash Lovers Tristan   ---->  Isolde
#
# affichage de toutes les valeurs
#
#      keyshash Lovers         ----> 'Tristan'  'Romeo Montague'
#
#
# Convention : au lieu de la syntaxe perls foo{bar} = boing',
# utiliser
#       '_foo_bar=boing' (deux tirets bas, pas d'espaces)
#
# 1) stocke la clé    dans _NAME_keys[]
# 2) stocke la valeur dans _NAME_values[] en utilisant le même index
# L'index de la dernière entrée est _NAME_ptr
#
# NOTE : pas de vérification d'erreurs.


function _inihash () {
    # fonction privée
    # appelée au début de chaque procédure
    # définit : _keys _values _ptr
    #
    # Usage : _inihash NAME
    local name=$1
    _keys=_${name}_keys
    _values=_${name}_values
    _ptr=_${name}_ptr
}

function newhash () {
    # Usage : newhash NAME
    #         NAME ne devrait pas contenir d'espaces ou de points.
    #         En fait, il doit être un nom syntaxiquement correct pour une variable
Bash.
    # Nous nous reposons sur Bash pour reconnaître automatiquement des tableaux.
    local name=$1
    local _keys _values _ptr
    _inihash ${name}
    eval ${_ptr}=0
}


function addhash () {
    # Usage : addhash NAME KEY 'VALUE with spaces'
    #         les arguments avec espaces doivent être mis entre guillemets ''
    local name=$1 k="$2" v="$3"
    local _keys _values _ptr
    _inihash ${name}

    #echo "DEBUG(addhash): ${_ptr}=${!_ptr}"

    eval let ${_ptr}=${_ptr}+1
    eval "$_keys[${!_ptr}]=\"${k}\""
    eval "$_values[${!_ptr}]=\"${v}\""
}

function gethash () {
```

```
    # Usage: gethash NAME KEY
    #        Renvoie boing
    #        ERR=0 si le nom est trouvée, 1 sinon
    # Ce n'est pas un bon hachage -- nous cherchons simplement dans les clés
    local name=$1 key="$2"
    local _keys _values _ptr
    local k v i found h
    _inihash ${name}

    # _ptr contient l'index le plus haut dans le hachage
    found=0

    for i in $(seq 1 ${!_ptr}); do
        h="\${${_keys}[${i}]}"  #  Plus propre de le faire en deux étapes
        eval k=${h}             #+ (tout spécialement avec les guillemets pour
                                #+ les espaces)
        if [ "${k}" = "${key}" ]; then found=1; break; fi
    done;

    [ ${found} = 0 ] && return 1;
    # sinon i est l'index qui correspond à la clé
    h="\${${_values}[${i}]}"
    eval echo "${h}"
    return 0;
}

function keyshash () {
    # Usage : keyshash NAME
    # Renvoie la liste de toutes les clés définies pour le nom du hachage
    local name=$1 key="$2"
    local _keys _values _ptr
    local k i h
    _inihash ${name}

    # _ptr contient l'index le plus haut du hachage
    for i in $(seq 1 ${!_ptr}); do
        h="\${${_keys}[${i}]}"   #  Plus propre de le faire en deux étapes
        eval k=${h}              #+ (tout spécialement avec les guillemets
                                 #+ pour les espaces)
        echo -n "'${k}' "
    done;
}


# --------------------------------------------------------------------

# Maintenant, testons-le.
# (d'après les commentaires au début du script).
newhash Lovers
addhash Lovers Tristan Isolde
addhash Lovers 'Romeo Montague' 'Juliet Capulet'

# Résultats en sortie.
echo
gethash Lovers Tristan      # Isolde
echo
keyshash Lovers             # 'Tristan' 'Romeo Montague'
echo; echo


exit 0

# Exercice :
#----------
#
# Ajouter des vérifications d'erreur aux fonctions.
```

Maintenant, un script qui installe et monte ces jolies clés USB, version « disques durs ».

Exemple A.25. Monter des périphériques de stockage USB

```
#!/bin/bash
# ==> usb.sh
# ==> Script pour monter et installer les périphériques de stockage d'une clé USB.
# ==> Lancer en tant que root au démarrage du système (voir ci-dessous).
# ==>
# ==> Les nouvelles distributions Linux (2004 ou ultérieures) détectent
# ==> automatiquement et installent les clés USB.
# ==> Elles n'ont donc pas besoin de ce script.
# ==> Mais c'est toujours instructif.

#  This code is free software covered by GNU GPL license version 2 or above.
#  Please refer to http://www.gnu.org/ for the full license text.
#
#  Ce code est un logiciel libre couvert par la licence GNU GPL version 2 et
#+ ultérieure. Référez-vous à http://www.gnu.org/ pour le texte complet.
#
#  Une partie du code provient de usb-mount écrit par Michael Hamilton (LGPL)
#+ voir http://users.actrix.co.nz/michael/usbmount.html
#
#  INSTALLATION
#  ------------
#  Placez ceci dans /etc/hotplug/usb/clefusb.
#  Puis regardez dans /etc/hotplug/usb.distmap, copiez toutes les entrées de
#+ stockage USB dans /etc/hotplug/usb.usermap, en substituant "usb-storage" par
#+ "diskonkey".
#  Sinon, ce code est seulement lancé lors de l'appel/suppression du module du
#+ noyau (au moins lors de mes tests), ce qui annule le but.
#
#  A FAIRE
#  -------
#  Gère plus d'un périphérique "diskonkey" en même temps (c'est-à-dire
#+ /dev/diskonkey1 et /mnt/clefusb1), etc. Le plus gros problème ici concerne
#+ la gestion par devlabel, que je n'ai pas essayé.
#
#  AUTEUR et SUPPORT
#  -----------------
#  Konstantin Riabitsev, &lt;icon linux duke edu&gt;.
#  Envoyez tout problème via mon adresse de courrier électronique.
#
# ==> Commentaires ajoutés par l'auteur du guide ABS.



PERIPH_LIENSYMBOLIQUE=/dev/diskonkey
POINT_MONTAGE=/mnt/clefusb
LABEL_PERIPH=/sbin/devlabel
CONFIG_LABEL_PERIPH=/etc/sysconfig/devlabel
JE_SUIS=$0

##
# Fonctions pratiquement récupérées du code d'usb-mount.
#
function tousUsbScsiAttaches {
    find /proc/scsi/ -path '/proc/scsi/usb-storage*' -type f |
    xargs grep -l 'Attaché: Oui'
}
function periphScsiAPartirScsiUsb {
    echo $1 | awk -F"[-/]" '{ n=$(NF-1);
    print "/dev/sd" substr("abcdefghijklmnopqrstuvwxyz", n+1, 1) }'
}

if [ "${ACTION}" = "add" ] && [ -f "${DEVICE}" ]; then
    ##
    # Récupéré du code d'usbcam.
    #
    if [ -f /var/run/console.lock ]; then
```

```
        PROPRIETAIRE_CONSOLE=`cat /var/run/console.lock`
    elif [ -f /var/lock/console.lock ]; then
        PROPRIETAIRE_CONSOLE=`cat /var/lock/console.lock`
    else
        PROPRIETAIRE_CONSOLE=
    fi
    for entreeProc in $(tousUsbScsiAttaches); do
        scsiDev=$(periphScsiAPartirScsiUsb $entreeProc)
        #  Quelques bogues avec usb-storage?
        #  Les partitions ne sont pas dans /proc/partitions jusqu'à ce qu'elles
        #+ soient utilisées.
        /sbin/fdisk -l $scsiDev >/dev/null
        ##
        #  La plupart des périphériques ont des informations de partitionnement,
        #+ donc les données sont sur /dev/sd?1. Néanmois, quelques-uns plus
        #+ stupides n'ont pas du tout de partitions et utilisent le périphérique
        #+ complet pour du stockage de données. Il essaie de deviner si vous
        #+ avez un /dev/sd?1 et si non, il utilise le périphérique entier.
        #
        if grep -q `basename $scsiDev`1 /proc/partitions; then
            part="$scsiDev""1"
        else
            part=$scsiDev
        fi
        ##
        #  Modifie le propriétaire de la partition par l'utilisateur de la
        #+ console pour qu'ils puissent le monter.
        #
        if [ ! -z "$PROPRIETAIRE_CONSOLE" ]; then
            chown $PROPRIETAIRE_CONSOLE:disk $part
        fi
        ##
        # Ceci vérifie si nous avons déjà cet UID défini avec devlabel. Sinon,
        # il ajoute alors le périphérique à la liste.
        #
        prodid=`$LABEL_PERIPH printid -d $part`
        if ! grep -q $prodid $CONFIG_LABEL_PERIPH; then
                # croisez les doigts et espérez que cela fonctionne
            $LABEL_PERIPH add -d $part -s $PERIPH_LIENSYMBOLIQUE 2>/dev/null
        fi
        ##
        # Vérifie si le point de montage existe et le crée dans le cas contraire.
        #
        if [ ! -e $POINT_MONTAGE ]; then
            mkdir -p $POINT_MONTAGE
        fi
        ##
        # S'occupe de /etc/fstab pour faciliter le montage.
        #
        if ! grep -q "^$PERIPH_LIENSYMBOLIQUE" /etc/fstab; then
            # Ajoute une entrée fstab
            echo -e \
                "$PERIPH_LIENSYMBOLIQUE\t\t$POINT_MONTAGE\t\tauto\tnoauto,owner,kudzu 0
0" \
                >> /etc/fstab
        fi
    done
    if [ ! -z "$REMOVER" ]; then
        ##
        #  Assurez-vous que ce script est appelé lors de la suppression du
        #+ périphérique.
        #
        mkdir -p `dirname $REMOVER`
        ln -s $JE_SUIS $REMOVER
    fi
elif [ "${ACTION}" = "remove" ]; then
    ##
    # Si le périphérique est monté, le démonte proprement.
    #
```

```
    if grep -q "$POINT_MONTAGE" /etc/mtab; then
        # Démonte proprement.
        umount -l $POINT_MONTAGE
    fi
    ##
    # Le supprime à partir de /etc/fstab s'il existe.
    #
    if grep -q "^$PERIPH_LIENSYMBOLIQUE" /etc/fstab; then
        grep -v "^$PERIPH_LIENSYMBOLIQUE" /etc/fstab > /etc/.fstab.new
        mv -f /etc/.fstab.new /etc/fstab
    fi
fi

exit 0
```

Un script qui réalise une conversion texte vers HTML.

Exemple A.26. Convertir en HTML

```
#!/bin/bash
# tohtml.sh

# Convertit un fichier texte au format HTML.
# Auteur :      Mendel Cooper
# Licence :     GPL3
# Utilisation : sh tohtml.sh < fichiertexte > fichierhtml
#  Ce script est facilement modifiable pour accepter
#+ des noms de fichier source et destination.

#    Suppositions :
# 1) Les paragraphes du fichier texte (cible) sont séparés par une ligne blanche.
# 2) Les images JPEG (*.jpg) sont situées dans le sous-répertoire "images".
#    Dans le fichier cible, les noms des images sont placés entre des crochets,
#    par exemple [image01.jpg].
# 3) Les phrases importantes (en italique) commencent avec un espace suivi d'un
#+   tiret bas ou le premier caractère sur la ligne est un tiret bas
#+   et finissent avec un tiret bas suivi d'un espace ou d'une fin de ligne.


# Paramétrages
TAILLEPOLICE=2        # Taille de police.
REPIMG="images"  # Répertoire images.
# En-têtes
ENT01='&lt;!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"&gt;'
ENT02='&lt;!-- Convertit en HTML par le script ***tohtml.sh*** --&gt;'
ENT03='&lt;!-- auteur du script : M. Leo Cooper &lt;thegrendel@theriver.com&gt; --&gt;'
ENT10='&lt;html&gt;'
ENT11='&lt;head&gt;'
ENT11a='&lt;/head&gt;'
ENT12a='&lt;title&gt;'
ENT12b='&lt;/title&gt;'
ENT121='&lt;META NAME="GENERATOR" CONTENT="tohtml.sh script"&gt;'
ENT13='&lt;body bgcolor="#dddddd"&gt;'   # Modifie la couleur du fond.
ENT14a='&lt;font size='
ENT14b='&gt;'
# Bas de page
FTR10='&lt;/body&gt;'
FTR11='&lt;/html&gt;'
# Balises
GRAS="&lt;b&gt;"
CENTRE="&lt;center&gt;"
FIN_CENTRE="&lt;/center&gt;"
LF="&lt;br&gt;"


ecrire_entetes ()
  {
```

```
  echo "$ENT01"
  echo
  echo "$ENT02"
  echo "$ENT03"
  echo
  echo
  echo "$ENT10"
  echo "$ENT11"
  echo "$ENT121"
  echo "$ENT11a"
  echo "$ENT13"
  echo
  echo -n "$ENT14a"
  echo -n "$TAILLEPOLICE"
  echo "$ENT14b"
  echo
  echo "$GRAS"        # Tout en gras (plus facile à lire).
  }


traitement_texte ()
  {
  while read ligne    # Lire une ligne à la fois.
  do
    {
    if [ ! "$ligne" ] # Ligne vide ?
    then              # Alors un nouveau paragraphe doit suivre.
      echo
      echo "$LF"      # Insérer deux balises &lt;br&gt;.
      echo "$LF"
      echo
      continue        # Ignorer le test du tiret bas.
    else              # Sinon...

      if [[ "$ligne" =~ "\[*jpg\]" ]] # Une image ?
      then                            # Supprimer les crochets.
        temp=$( echo "$ligne" | sed -e 's/\[//' -e 's/\]//' )
        line=""$CENTRE" &lt;img src="\"$REPIMG"/$temp\"&gt; "$FIN_CENTRE" "
                                      # Ajouter la balise de l'image
                                      # et la centrer.
      fi

    fi


    echo "$ligne" | grep -q _
    if [ "$?" -eq 0 ]    # Si la ligne contient un tiret bas..
    then
      # ============================================================
      # Placer en italique une phrase entre tiret bas.
      temp=$( echo "$ligne" |
              sed -e 's/ _/ &lt;i&gt;/' -e 's/_ /&lt;\/i&gt; /' |
              sed -e 's/^_/&lt;i&gt;/'  -e 's/_$/&lt;\/i&gt;/' )
      #  Traiter seulement les tirets bas préfixés par un espace,
      #+ suivi par un espace ou en fin ou en début de ligne.
      #  Ne pas convertir les tirets bas contenus dans un mot !
      line="$temp"
      # Ralentit l'exécution du script. Cela peut-il être optimisé ?
      # ============================================================
    fi



    echo
    echo "$ligne"
    echo
    } # Fin while
  done
  }   # Fin traitement_texte ()
```

```
ecrire_basdepage ()  # Fin des balises.
  {
  echo "$FTR10"
  echo "$FTR11"
  }


# main () {
# =========
ecrire_entetes
traitement_texte
ecrire_basdepage
# =========
#         }

exit $?

#  Exercices :
#  ----------
#  1) Correction : vérifiez le tiret bas de fermeture avant une virgule ou
#+    un point.
#  2) Ajoutez un test sur le présence d'un tiret bas de fin dans les phrases
#+    à mettre en italique.
```

Voici quelque chose qui va réchauffer le coeur des webmasters : un script qui sauvegarde les traces du serveur web.

Exemple A.27. Préserver les weblogs

```
#!/bin/bash
# archiveweblogs.sh v1.0

# Troy Engel <tengel@fluid.com>
# Légèrement modifié par l'auteur du document
# Utilisé avec sa permission.
#
#  Ce script préservera les traces web habituellement supprimées à partir d'une
#+ installation RedHat/Apache par défaut.
#  Il sauvegardera les fichiers en indiquant la date et l'heure dans le nom du
#+ fichier, compressé avec bzip, dans un répertoire donné.
#
#  Lancez ceci avec crontab la nuit car bzip2 avale la puissance du CPU sur des
#+ journaux particulièrement gros.
#  0 2 * * * /opt/sbin/archiveweblogs.sh


PROBLEME=66

# Modifiez-le par votre répertoire de sauvegarde.
REP_SAUVEGARDE=/opt/sauvegardes/journaux_web

# Apache/RedHat par défaut
JOURS_DE_SAUVEGARDE="4 3 2 1"
REP_JOURNAUX=/var/log/httpd
JOURNAUX="access_log error_log"

# Emplacement par défaut des programmes RedHat
LS=/bin/ls
MV=/bin/mv
ID=/usr/bin/id
CUT=/bin/cut
COL=/usr/bin/column
BZ2=/usr/bin/bzip2

# Sommes-nous root?
USER=`$ID -u`
```

```
if [ "X$USER" != "X0" ]; then
  echo "PANIQUE : Seul root peut lancer ce script !"
  exit $PROBLEME
fi

# Le répertoire de sauvegarde existe-t'il ? est-il modifiable ?
if [ ! -x $REP_SAUVEGARDE ]; then
  echo "PANIQUE : $REP_SAUVEGARDE n'existe pas ou n'est pas modifiable !"
  exit $PROBLEME
fi

# Déplace, renomme et compresse avec bzip2 les journaux
for jour in $JOURS_DE_SAUVEGARDE; do
  for journal in $JOURNAUX; do
    MONFICHIER="$REP_JOURNAUX/$journal.$jour"
    if [ -w $MONFICHIER ]; then
      DTS=`$LS -lgo --time-style=+%Y%m%d $MONFICHIER | $COL -t | $CUT -d ' ' -f7`
      $MV $MONFICHIER $REP_SAUVEGARDE/$journal.$DTS
      $BZ2 $REP_SAUVEGARDE/$journal.$DTS
    else
           # Affiche une erreur seulement si le fichier existe (ne peut
           # s'écrire sur lui-même).
      if [ -f $MONFICHIER ]; then
        echo "ERREUR : $MONFICHIER n'est pas modifiable. Je passe au suivant."
      fi
    fi
  done
done

exit 0
```

Comment empêcher le shell d'étendre et de réinterpréter leschaînes ?

Exemple A.28. Protéger les chaînes littérales

```
#! /bin/bash
# protect_literal.sh

# set -vx

:<<-'_Protect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
    Revision: $ID$

    Copyright (c) Michael S. Zick, 2003; Tous droits réservés
    Licence: Utilisation non restreinte quelque soit sa forme, quelque soit le
    but.
    Garantie : Aucune
    Revision: $ID$

    Documentation redirigée vers no-operation sous Bash. Bash enverra ce bloc
    vers '/dev/null' lorsque le script sera lu la première fois.
    (Supprimez le commentaire de la commande ci-dessus pour voir cette action.)

    Supprimez la première ligne (Sha-Bang, #!) lors de l'utilisation de ce
    script en tant que procédure d'une bibliothèque. Décommentez aussi
    le code d'exemple utilisé dans les deux places indiquées.


    Usage:
        _protect_literal_str 'une chaine quelconque qui correspond à votre
        ${fantaisie}'
        Affiche simplement l'argument sur la sortie standard, les guillemets étant
        restaurés.
```

```
        $(_protect_literal_str 'une chaine quelconque qui correspond à votre
        ${fantaisie}')
        sur le côté droit d'une instruction d'affectation.

    Fait:
        Utilisé sur le côté droit d'une affectation, préserve les guillemets
        protégeant le contenu d'un littéral lors de son affectation.

    Notes:
        Les noms étranges (_*) sont utilisé pour éviter de rencontrer ceux
        choisis par l'utilisateur lorsqu'il l'utilise en tant que bibliothèque.


_Protect_Literal_String_Doc

# La fonction 'pour illustration'

_protect_literal_str() {

# Récupére un caractère inutilisé, non affichable comme IFS local.
# Non requis, mais montre ce que nous ignorons.
    local IFS=$'\x1B'               # caractère \ESC

# Entoure tous_elements_de entre guillemets lors de l'affectation.
    local tmp=$'\x27'$@$'\x27'
#    local tmp=$'\''$@$'\''         # Encore plus sale.

    local len=${#tmp}               # Info seulement.
    echo $tmp a une longueur de $len.         # Sortie ET information.
}

# Ceci est la version nom-court.
_pls() {
    local IFS=$'x1B'                # caractère \ESC (non requis)
    echo $'\x27'$@$'\x27'           # Paramètre global codé en dur
}

# :<<-'_Protect_Literal_String_Test'
# # # Supprimez le "# " ci-dessus pour désactiver ce code. # # #

# Voir à quoi ressemble ceci une fois affiché.
echo
echo "- - Test Un - -"
_protect_literal_str 'Bonjour $utilisateur'
_protect_literal_str 'Bonjour "${nom_utilisateur}"'
echo

# Ce qui donne :
# - - Test Un - -
# 'Bonjour $utilisateur' fait 13 caractères de long.
# 'Bonjour "${nom_utilisateur}"' a une taille de 21 caractères.

#  Cela ressemble à notre attente, donc pourquoi tout ceci ?
#  La différence est cachée à l'intérieur de l'ordonnancement interne des opérations
#+ de Bash.
#  Ce qui s'affiche lorsque vous l'utilisez sur le côté droit de l'affectation.

# Déclarez un tableau pour les valeurs de tests.
declare -a tableauZ

#  Affecte les éléments comprenant différents types de guillemets et de caractères
#+ d'échappement.
tableauZ=( zero "$(_pls 'Bonjour ${Moi}')" 'Bonjour ${Toi}' "\'Passe: ${pw}\'" )

# Maintenant, affiche ce tableau.
echo "- - Test Deux - -"
for (( i=0 ; i<${#tableauZ[*]} ; i++ ))
do
```

```
    echo  Elément $i: ${tableauZ[$i]} fait  ${#tableauZ[$i]} caractères de long.
done
echo

# Ce qui nous donne :
# - - Test Deux - -
# Elément 0: zero fait 4 caractères de long.           # Notre élément marqueur
# Elément 1: 'Bonjour ${Moi}' fait 13 caractères de long.# Notre "$(_pls '...' )"
# Elément 2: Bonjour ${Toi} fait 12 caractères de long.  # Les guillemets manquent
# Elément 3: \'Passe: \' fait 10 caractères de long.    # ${pw} n'affiche rien

# Maintenant, affectez ce résultat.
declare -a tableau2=( ${tableauZ[@]} )

# Et affiche ce qui s'est passé.
echo "- - Test Trois - -"
for (( i=0 ; i<${#tableau2[*]} ; i++ ))
do
    echo  Elément $i: ${tableau2[$i]} fait ${#tableau2[$i]} caractères de long.
done
echo

# Ce qui nous donne :
# - - Test Trois - -
# Elément 0: zero fait 4 caractères de long.         # Notre élément marqueur.
# Elément 1: Hello ${Moi} fait 11 caractères de long.# Résultat attendu.
# Elément 2: Hello fait 5 caractères de long.        # ${Toi} n'affiche rien.
# Elément 3: 'Passe: fait 6 caractères de long.      # Se coupe sur les espaces.
# Elément 4: ' fait 1 caractères de long.            # Le guillemet final est ici
                                                     # maintenant.

#  Les guillemets de début et de fin de notre élément 1 sont supprimés.
#  Bien que non affiché, les espaces blancs de début et de fin sont aussi supprimés.
#  Maintenant que le contenu des chaînes est initialisé, Bash placera toujours, en
interne,
#+ entre guillemets les contenus comme requis lors de ses opérations.

#  Pourquoi?
#  En considérant notre construction "$(_pls 'Hello ${Moi}')" :
#  " ... " -> Supprime les guillemets.
#  $( ... ) -> Remplace avec le resultat de ..., supprime ceci.
#  _pls ' ... ' -> appelé avec des arguments littérales, supprime les guillemets.
#  Le résultat renvoyé inclut les guillemets ; MAIS le processus ci-dessus a déjà
#+ été réalisé, donc il devient une partie de la valeur affectée.
#
#  De manière identique, lors d'une utilisation plus poussée de la variable de type
#+ chaînes de caractères, le ${Moi} fait partie du contenu (résultat) et survit à
#+ toutes les opérations.
#  (Jusqu'à une indication explicite pour évaluer la chaîne).

#  Astuce : Voir ce qui arrive lorsque les guillemets ($'\x27') sont remplacés par
#+ des caractères ($'\x22') pour les procédures ci-dessus.
#  Intéressant aussi pour supprimer l'ajout de guillemets.

# _Protect_Literal_String_Test
# # # Supprimez le caractère "# " ci-dessus pour désactiver ce code. # # #

exit 0
```

Et si vous *voulez* que le shell étende et réinterprète les chaînes ?

Exemple A.29. Ne pas protéger les chaînes littérales

```
#! /bin/bash
# unprotect_literal.sh

# set -vx
```

```
:<<-'_UnProtect_Literal_String_Doc'

    Copyright (c) Michael S. Zick, 2003; All Rights Reserved
    License: Unrestricted reuse in any form, for any purpose.
    Warranty: None
    Revision: $ID$

    Copyright (c) Michael S. Zick, 2003; Tous droits réservés
    Licence: Utilisation non restreinte quelque soit sa forme, quelque soit le
    but.
    Garantie : Aucune
    Revision: $ID$

    Documentation redirigée vers no-operation sous Bash. Bash enverra ce bloc
    vers '/dev/null' lorsque le script est lu la première fois.
    (Supprimez le commentaire de la commande ci-dessus pour voir cette action.)

    Supprimez la première ligne (Sha-Bang, #!) lors de l'utilisation de ce
    script en tant que procédure d'une bibliothèque. Dé-commentez aussi
    le code d'exemple utilisé dans les deux places indiquées.


    Utilisation:
        Complément de la fonction "$(_pls 'Chaine litterale')".
        (Voir l'exemple protect_literal.sh.)

        VarChaine=$(_upls VariableChaineProtege)

    Fait:
        Lorsqu'utilisé sur le côté droit d'une instruction d'affectation ;
        fait que la substition est intégré à la chaîne protégée.

    Notes:
        Les noms étranges (_*) sont utilisé pour éviter de rencontrer ceux
        choisis par l'utilisateur lorsqu'il l'utilise en tant que bibliothèque.


_UnProtect_Literal_String_Doc

_upls() {
    local IFS=$'x1B'                # Caractère \ESC (non requis)
    eval echo $@                    # Substitution on the glob.
}

# :<<-'_UnProtect_Literal_String_Test'
# # # Supprimez le "# " ci-dessus pour désactiver ce code. # # #


_pls() {
    local IFS=$'x1B'                # Caractère \ESC (non requis)
    echo $'\x27'$@$'\x27'           # Paramètre global codé en dur.
}

# Déclare un tableau pour les valeurs de tests.
declare -a tableauZ

# Affecte les éléments avec des types différents de guillements et échappements.
tableauZ=( zero "$(_pls 'Bonjour ${Moi}')" 'Bonjour ${Toi}' "\'Passe: ${pw}\'" )

# Maintenant, faire une affectation avec ce résultat.
declare -a tableau2=( ${tableauZ[@]} )

# Ce qui fait :
# - - Test trois - -
# Elément 0: zero est d'une longueur 4            # Notre élément marqueur.
# Elément 1: Bonjour ${Moi} est d'une longueur 11 # Résultat attendu.
# Elément 2: Bonjour est d'une longueur 5         # ${Toi} ne renvoit rien.
# Elément 3: 'Passe est d'une longueur 6          # Divisé sur les espaces.
```

```
# Elément 4: ' est d'une longueur 1               # La fin du guillemet est ici
                                                  # maintenant.

# set -vx

#  Initialise 'Moi' avec quelque-chose pour la substitution imbriqué ${Moi}.
#  Ceci a besoin d'être fait SEULEMENT avant d'évaluer la chaîne protégée.
#  (C'est pourquoi elle a été protégée.)

Moi="au gars du tableau."

# Initialise une variable de chaînes de caractères pour le résultat.
nouvelleVariable=$(_upls ${tableau2[1]})

# Affiche le contenu.
echo $nouvelleVariable

# Avons-nous réellement besoin d'une fonction pour faire ceci ?
variablePlusRecente=$(eval echo ${tableau2[1]})
echo $variablePlusRecente

#  J'imagine que non mais la fonction _upls nous donne un endroit où placer la
#+ documentation.
#  Ceci aide lorsque nous oublions une construction # comme ce que signifie
#+ $(eval echo ... ).

#  Que se passe-t'il si Moi n'est pas initialisé quand la chaîne protégée est
#+ évaluée ?
unset Moi
variableLaPlusRecente=$(_upls ${tableau2[1]})
echo $variableLaPlusRecente

# Simplement partie, pas d'aide, pas d'exécution, pas d'erreurs.

#  Pourquoi ?
#  Initialiser le contenu d'une variable de type chaîne contenant la séquence de
#+ caractères qui ont une signification dans Bash est un problème général
#+ d'écriture des scripts.
#
#  Ce problème est maintenant résolu en huit lignes de code (et quatre pages de
#+ description).

#  Où cela nous mène-t'il ?
#  Les pages web au contenu dynamique en tant que tableau de chaînes Bash.
#  Le contenu par requête pour une commande Bash 'eval' sur le modèle de page
#+ stocké.
#  Pas prévu pour remplacer PHP, simplement quelque chose d'intéressant à faire.
###
#  Vous n'avez pas une application pour serveur web ?
#  Aucun problème, vérifiez dans le répertoire d'exemples des sources Bash :
#+ il existe aussi un script Bash pour faire ça.

# _UnProtect_Literal_String_Test
# # # Supprimez le "# " ci-dessus pour désactiver ce code. # # #

exit 0
```

Ce script intéressant chasse les spammers.

Exemple A.30. Identification d'un spammer

```
#!/bin/bash

# $Id: is_spammer.bash,v 1.7 2008-05-10 08:36:14 gleu Exp $
# L'information ci-dessus est l'ID RCS.

# La dernière version de ce script est disponible sur http://www.morethan.org.
```

```
#
# Spammer-identification
# par Michael S. Zick
# Utilisé dans le guide ABS Guide avec sa permission.



###########################################################
# Documentation
# Voir aussi "Quickstart" à la fin du script.
###########################################################

:<<-'__is_spammer_Doc_'

    Copyright (c) Michael S. Zick, 2004
    Licence : Ré-utilisation non restreinte quelque soit la forme et
              le but
    Garantie: Aucune -{C'est un script; l'utilisateur est seul responsable.}-

Impatient?
    Code de l'application : Allez à "# # #  Code 'Chez le spammeur'  # # #"
    Sortie d'exemple      : ":<<-'_is_spammer_outputs_'"
    Comment l'utiliser    : Entrer le nom du script sans arguments.
                Ou allez à "Quickstart" à la fin du script.

Fournit
    Avec un nom de domaine ou une adresse IP(v4) en entrée :

    Lance un ensemble exhaustif de requêtes pour trouver les ressources réseau
    associées (raccourci pour un parcours récursif dans les TLD).

    Vérifie les adresses IP(v4) disponibles sur les serveurs de noms Blacklist.

    S'il se trouve faire partie d'une adresse IP(v4) indiquée, rapporte les
    enregistrements texte de la liste noire.
    (habituellement des liens hypertextes vers le rapport spécifique.)

Requiert
    Une connexion Internet fonctionnelle.
    (Exercice : ajoutez la vérification et/ou annulez l'opération si la
    connexion n'est pas établie lors du lancement du script.)
    Une version de Bash disposant des tableaux (2.05b+).

    Le programme externe 'dig' --
    ou outil fourni avec l'ensemble de programmes 'bind'.
    Spécifiquement, la version qui fait partie de Bind série 9.x
    Voir : http://www.isc.org

    Toutes les utilisations de 'dig' sont limitées à des fonctions d'emballage,
    qui pourraient être ré-écrites si nécessaire.
    Voir : dig_wrappers.bash pour plus de détails.
         ("Documentation supplémentaire" -- ci-dessous)

Usage
    Ce script requiert un seul argument, qui pourrait être:
    1) Un nom de domaine ;
    2) Une adresse IP(v4) ;
    3) Un nom de fichier, avec un nom ou une adresse par ligne.

    Ce script accepte un deuxième argument optionnel, qui pourrait être:
    1) Un serveur de noms Blacklist ;
    2) Un nom de fichier avec un serveur de noms Blacklist par ligne.

    Si le second argument n'est pas fourni, le script utilise un ensemble
    intégré de serveurs Blacklist (libres).

    Voir aussi la section Quickstart à la fin de ce script (après 'exit').

Codes de retour
```

```
    0 - Tout est OK
    1 - Échec du script
    2 - Quelque chose fait partie de la liste noire

Variables d'environnement optionnelles
    SPAMMER_TRACE
        S'il comprend le nom d'un fichier sur lequel le script a droit
        d'écriture, le script tracera toute l'exécution.

    SPAMMER_DATA
        S'il comprend le nom d'un fichier sur lequel le script a droit
        d'écriture, le script y enregitrera les données trouvées sous la forme
        d'un fichier GraphViz.
        Voir : http://www.research.att.com/sw/tools/graphviz

    SPAMMER_LIMIT
        Limite la profondeur des recherches de ressources.

        Par défaut à deux niveaux.

        Un paramètrage de 0 (zero) signifie 'illimité' . . .
          Attention : le script pourrait parcourir tout Internet !

        Une limite de 1 ou 2 est plus utile dans le cas d'un fichier de noms de
        domaine et d'adresses.
        Une limite encore plus haute est utile pour chasser les gangs de spam.


Documentation supplémentaire
    Téléchargez l'ensemble archivé de scripts expliquant et illustrant la
    fonction contenue dans ce script.
    http://personal.riverusers.com/mszick_clf.tar.bz2


Notes d'étude
    Ce script utilise un grand nombre de fonctions.
    Pratiquement toutes les fonctions générales ont leur propre script
    d'exemple. Chacun des scripts d'exemples ont leur commentaires (niveau
    tutoriel).

Projets pour ce script
    Ajoutez le support des adresses IP(v6).
    Les adresses IP(v6) sont reconnues mais pas gérées.

Projet avancé
    Ajoutez le détail de la recherche inverse dans les informations découvertes.

    Rapportez la chaîne de délégation et les contacts d'abus.

    Modifiez la sortie du fichier GraphViz pour inclure les informations
    nouvellement découvertes.

__is_spammer_Doc_

############################################################




#### Configuration spéciale pour l'IFS utilisée pour l'analyse des chaînes. ####

# Espace blanc == :Espace:Tabulation:Retour à la ligne:Retour chariot:
WSP_IFS=$'\x20'$'\x09'$'\x0A'$'\x0D'

# Pas d'espace blanc == Retour à la ligne:Retour chariot
NO_WSP=$'\x0A'$'\x0D'

# Séparateur de champ pour les adresses IP décimales
ADR_IFS=${NO_WSP}'.'
```

```
# Tableau de conversions de chaînes
DOT_IFS='.'${WSP_IFS}

# # # Machine à pile pour les opérations restantes # # #
# Cet ensemble de fonctions est décrite dans func_stack.bash.
# (Voir "Documentation supplémentaire" ci-dessus.)
# # #

# Pile globale des opérations restantes.
declare -f -a _pending_
# Sentinelle gloable pour les épuiseurs de pile
declare -i _p_ctrl_
# Déteneur global pour la fonction en cours d'exécution
declare -f _pend_current_

# # # Version de déboguage seulement - à supprimer pour une utilisation normale
# # #
#
# La fonction stockée dans _pend_hook_ est appellée immédiatement avant que
# chaque fonction en cours ne soit évaluée. Pile propre, _pend_current_ configuré.
#
# Ceci est démontré dans pend_hook.bash.
declare -f _pend_hook_
# # #

# La fonction ne faisant rien.
pend_dummy() { : ; }

# Efface et initialise la pile des fonctions.
pend_init() {
    unset _pending_[@]
    pend_func pend_stop_mark
    _pend_hook_='pend_dummy'  # Débogage seulement.
}

# Désactive la fonction du haut de la pile.
pend_pop() {
    if [ ${#_pending_[@]} -gt 0 ]
    then
        local -i _top_
        _top_=${#_pending_[@]}-1
        unset _pending_[$_top_]
    fi
}

# pend_func function_name [$(printf '%q\n' arguments)]
pend_func() {
    local IFS=${NO_WSP}
    set -f
    _pending_[${#_pending_[@]}]=$@
    set +f
}

# La fonction qui arrête la sortie :
pend_stop_mark() {
    _p_ctrl_=0
}

pend_mark() {
    pend_func pend_stop_mark
}

# Exécute les fonctions jusqu'à 'pend_stop_mark' . . .
pend_release() {
    local -i _top_              # Déclare _top_ en tant qu'entier.
    _p_ctrl_=${#_pending_[@]}
    while [ ${_p_ctrl_} -gt 0 ]
    do
```

```
       _top_=${#_pending_[@]}-1
       _pend_current_=${_pending_[$_top_]}
       unset _pending_[$_top_]
       $_pend_hook_            # Débogage seulement.
       eval $_pend_current_
    done
}

# Supprime les fonctions jusqu'à 'pend_stop_mark' . . .
pend_drop() {
    local -i _top_
    local _pd_ctrl_=${#_pending_[@]}
    while [ ${_pd_ctrl_} -gt 0 ]
    do
       _top_=$_pd_ctrl_-1
       if [ "${_pending_[$_top_]}" == 'pend_stop_mark' ]
       then
           unset _pending_[$_top_]
           break
       else
           unset _pending_[$_top_]
           _pd_ctrl_=$_top_
       fi
    done
    if [ ${#_pending_[@]} -eq 0 ]
    then
        pend_func pend_stop_mark
    fi
}

#### Éditeurs de tableaux ####

# Cette fonction est décrite dans edit_exact.bash.
# (Voir "Additional documentation", ci-dessus.)
# edit_exact &lt;excludes_array_name&gt; &lt;target_array_name&gt;
edit_exact() {
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ee_Excludes
    local -a _ee_Target
    local _ee_x
    local _ee_t
    local IFS=${NO_WSP}
    set -f
    eval _ee_Excludes=\( \$\{$1\[@\]\} \)
    eval _ee_Target=\( \$\{$2\[@\]\} \)
    local _ee_len=${#_ee_Target[@]}     # Longueur originale.
    local _ee_cnt=${#_ee_Excludes[@]}   # Exclut la longueur de la liste.
    [ ${_ee_len} -ne 0 ] || return 0    # Ne peut pas éditer une longueur nulle.
    [ ${_ee_cnt} -ne 0 ] || return 0    # Ne peut pas éditer une longueur nulle.
    for (( x = 0; x < ${_ee_cnt} ; x++ ))
    do
        _ee_x=${_ee_Excludes[$x]}
        for (( n = 0 ; n < ${_ee_len} ; n++ ))
        do
            _ee_t=${_ee_Target[$n]}
            if [ x"${_ee_t}" == x"${_ee_x}" ]
            then
                unset _ee_Target[$n]     # Désactive la correspondance.
                [ $# -eq 2 ] && break    # Si deux arguments, alors terminé.
            fi
        done
    done
    eval $2=\( \$\{_ee_Target\[@\]\} \)
    set +f
    return 0
}

# Cette fonction est décrite dans edit_by_glob.bash.
```

```
# edit_by_glob &lt;excludes_array_name&gt; &lt;target_array_name&gt;
edit_by_glob() {
    [ $# -eq 2 ] ||
    [ $# -eq 3 ] || return 1
    local -a _ebg_Excludes
    local -a _ebg_Target
    local _ebg_x
    local _ebg_t
    local IFS=${NO_WSP}
    set -f
    eval _ebg_Excludes=\( \$\{$1\[@\]\} \)
    eval _ebg_Target=\( \$\{$2\[@\]\} \)
    local _ebg_len=${#_ebg_Target[@]}
    local _ebg_cnt=${#_ebg_Excludes[@]}
    [ ${_ebg_len} -ne 0 ] || return 0
    [ ${_ebg_cnt} -ne 0 ] || return 0
    for (( x = 0; x < ${_ebg_cnt} ; x++ ))
    do
        _ebg_x=${_ebg_Excludes[$x]}
        for (( n = 0 ; n < ${_ebg_len} ; n++ ))
        do
            [ $# -eq 3 ] && _ebg_x=${_ebg_x}'*'  #  Do prefix edit
            if [ ${_ebg_Target[$n]:=} ]          #+ if defined & set.
            then
                _ebg_t=${_ebg_Target[$n]/#${_ebg_x}/}
                [ ${#_ebg_t} -eq 0 ] && unset _ebg_Target[$n]
            fi
        done
    done
    eval $2=\( \$\{_ebg_Target\[@\]\} \)
    set +f
    return 0
}

# Cette fonction est décrite par unique_lines.bash.
# unique_lines &lt;in_name&gt; &lt;out_name&gt;
unique_lines() {
    [ $# -eq 2 ] || return 1
    local -a _ul_in
    local -a _ul_out
    local -i _ul_cnt
    local -i _ul_pos
    local _ul_tmp
    local IFS=${NO_WSP}
    set -f
    eval _ul_in=\( \$\{$1\[@\]\} \)
    _ul_cnt=${#_ul_in[@]}
    for (( _ul_pos = 0 ; _ul_pos < ${_ul_cnt} ; _ul_pos++ ))
    do
        if [ ${_ul_in[${_ul_pos}]:=} ]      # Si définie et non vide
        then
            _ul_tmp=${_ul_in[${_ul_pos}]}
            _ul_out[${#_ul_out[@]}]=${_ul_tmp}
            for (( zap = _ul_pos ; zap < ${_ul_cnt} ; zap++ ))
            do
                [ ${_ul_in[${zap}]:=} ] &&
                [ 'x'${_ul_in[${zap}]} == 'x'${_ul_tmp} ] &&
                    unset _ul_in[${zap}]
            done
        fi
    done
    eval $2=\( \$\{_ul_out\[@\]\} \)
    set +f
    return 0
}

# Cette fonction est décrite par char_convert.bash.
# to_lower &lt;string&gt;
to_lower() {
```

```
    [ $# -eq 1 ] || return 1
    local _tl_out
    _tl_out=${1//A/a}
    _tl_out=${_tl_out//B/b}
    _tl_out=${_tl_out//C/c}
    _tl_out=${_tl_out//D/d}
    _tl_out=${_tl_out//E/e}
    _tl_out=${_tl_out//F/f}
    _tl_out=${_tl_out//G/g}
    _tl_out=${_tl_out//H/h}
    _tl_out=${_tl_out//I/i}
    _tl_out=${_tl_out//J/j}
    _tl_out=${_tl_out//K/k}
    _tl_out=${_tl_out//L/l}
    _tl_out=${_tl_out//M/m}
    _tl_out=${_tl_out//N/n}
    _tl_out=${_tl_out//O/o}
    _tl_out=${_tl_out//P/p}
    _tl_out=${_tl_out//Q/q}
    _tl_out=${_tl_out//R/r}
    _tl_out=${_tl_out//S/s}
    _tl_out=${_tl_out//T/t}
    _tl_out=${_tl_out//U/u}
    _tl_out=${_tl_out//V/v}
    _tl_out=${_tl_out//W/w}
    _tl_out=${_tl_out//X/x}
    _tl_out=${_tl_out//Y/y}
    _tl_out=${_tl_out//Z/z}
    echo ${_tl_out}
    return 0
}

#### Fonctions d'aide de l'application ####

# Tout le monde n'utilise pas de points comme séparateur (APNIC, par exemple).
# Cette fonction est décrite par to_dot.bash
# to_dot &lt;string&gt;
to_dot() {
    [ $# -eq 1 ] || return 1
    echo ${1//[#|@|%]/.}
    return 0
}

# Cette fonction est décrite par is_number.bash.
# is_number &lt;input&gt;
is_number() {
    [ "$#" -eq 1 ]    || return 1  # est-ce blanc ?
    [ x"$1" == 'x0' ] && return 0  # est-ce zéro  ?
    local -i tst
    let tst=$1 2>/dev/null         # sinon, c'est numérique !
    return $?
}

# Cette fonction est décrite par is_address.bash.
# is_address &lt;input&gt;
is_address() {
    [ $# -eq 1 ] || return 1    # Blanc ==> faux
    local -a _ia_input
    local IFS=${ADR_IFS}
    _ia_input=( $1 )
    if  [ ${#_ia_input[@]} -eq 4 ]  &&
        is_number ${_ia_input[0]}   &&
        is_number ${_ia_input[1]}   &&
        is_number ${_ia_input[2]}   &&
        is_number ${_ia_input[3]}   &&
        [ ${_ia_input[0]} -lt 256 ] &&
        [ ${_ia_input[1]} -lt 256 ] &&
        [ ${_ia_input[2]} -lt 256 ] &&
        [ ${_ia_input[3]} -lt 256 ]
```

```
    then
        return 0
    else
        return 1
    fi
}

# Cette fonction est décrite par split_ip.bash.
# split_ip &lt;IP_address&gt; &lt;array_name_norm&gt; [&lt;array_name_rev&gt;]
split_ip() {
    [ $# -eq 3 ] ||              #  Soit trois
    [ $# -eq 2 ] || return 1     #+ soit deux arguments
    local -a _si_input
    local IFS=${ADR_IFS}
    _si_input=( $1 )
    IFS=${WSP_IFS}
    eval $2=\(\ \$\{_si_input\[@\]\}\ \)
    if [ $# -eq 3 ]
    then
        # Construit le tableau de l'ordre des requêtes.
        local -a _dns_ip
        _dns_ip[0]=${_si_input[3]}
        _dns_ip[1]=${_si_input[2]}
        _dns_ip[2]=${_si_input[1]}
        _dns_ip[3]=${_si_input[0]}
        eval $3=\(\ \$\{_dns_ip\[@\]\}\ \)
    fi
    return 0
}

# Cette fonction est décrite par dot_array.bash.
# dot_array &lt;array_name&gt;
dot_array() {
    [ $# -eq 1 ] || return 1     # Un seul argument requis.
    local -a _da_input
    eval _da_input=\(\ \$\{$1\[@\]\}\ \)
    local IFS=${DOT_IFS}
    local _da_output=${_da_input[@]}
    IFS=${WSP_IFS}
    echo ${_da_output}
    return 0
}

# Cette fonction est décrite par file_to_array.bash
# file_to_array &lt;file_name&gt; &lt;line_array_name&gt;
file_to_array() {
    [ $# -eq 2 ] || return 1  # Deux arguments requis.
    local IFS=${NO_WSP}
    local -a _fta_tmp_
    _fta_tmp_=( $(cat $1) )
    eval $2=\( \$\{_fta_tmp_\[@\]\} \)
    return 0
}

# Columnized print of an array of multi-field strings.
# col_print &lt;array_name&gt; &lt;min_space&gt; &lt;tab_stop [tab_stops]&gt;
col_print() {
    [ $# -gt 2 ] || return 0
    local -a _cp_inp
    local -a _cp_spc
    local -a _cp_line
    local _cp_min
    local _cp_mcnt
    local _cp_pos
    local _cp_cnt
    local _cp_tab
    local -i _cp
    local -i _cpf
    local _cp_fld
```

```
    #  ATTENTION : LIGNE SUIVANTE NON BLANCHE -- CE SONT DES ESPACES ENTRE
    #+             GUILLEMET.
    local _cp_max='                                                                '
    set -f
    local IFS=${NO_WSP}
    eval _cp_inp=\(\ \$\{$1\[@\]\}\ \)
    [ ${#_cp_inp[@]} -gt 0 ] || return 0 # Le cas vide est simple.
    _cp_mcnt=$2
    _cp_min=${_cp_max:1:${_cp_mcnt}}
    shift
    shift
    _cp_cnt=$#
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_spc[${#_cp_spc[@]}]="${_cp_max:2:$1}" #"
        shift
    done
    _cp_cnt=${#_cp_inp[@]}
    for (( _cp = 0 ; _cp < _cp_cnt ; _cp++ ))
    do
        _cp_pos=1
        IFS=${NO_WSP}$'\x20'
        _cp_line=( ${_cp_inp[${_cp}]} )
        IFS=${NO_WSP}
        for (( _cpf = 0 ; _cpf < ${#_cp_line[@]} ; _cpf++ ))
        do
            _cp_tab=${_cp_spc[${_cpf}]:${_cp_pos}}
            if [ ${#_cp_tab} -lt ${_cp_mcnt} ]
            then
                _cp_tab="${_cp_min}"
            fi
            echo -n "${_cp_tab}"
            (( _cp_pos = ${_cp_pos} + ${#_cp_tab} ))
            _cp_fld="${_cp_line[${_cpf}]}"
            echo -n ${_cp_fld}
            (( _cp_pos = ${_cp_pos} + ${#_cp_fld} ))
        done
        echo
    done
    set +f
    return 0
}

# # # # Flux de données 'Chassez le spammeur' # # # #

# Code de retour de l'application
declare -i _hs_RC

# Entrée originale, à partir de laquelle les adresses IP sont supprimées
# Après cela, les noms de domaine à vérifier
declare -a uc_name

# Les adresses IP de l'entrée originale sont déplacées ici
# Après cela, les adresses IP à vérifier
declare -a uc_address

# Noms contre lesquels l'expansion d'adresses est lancée
# Prêt pour la recherche des détails des noms
declare -a chk_name

# Noms contre lesquelles l'expansion de noms est lancée
# Prêt pour la recherche des détails des adresses
declare -a chk_address

#  La récursion est depth-first-by-name.
#  expand_input_address maintient cette liste pour prohiber
#+ deux fois les adresses à rechercher durant la récursion
#+ des noms de domaine.
declare -a been_there_addr
```

```
been_there_addr=( '127.0.0.1' ) # Liste blanche pour localhost

# Noms que nous avons vérifié (ou abandonné)
declare -a known_name

# Adresses que nous avons vérifié (ou abandonné)
declare -a known_address

#  Liste de zéro ou plus de serveurs Blacklist pour la vérification.
#  Chaque 'known_address' vérifiera chaque serveur,
#+ avec des réponses négatives et des échecs supprimés.
declare -a list_server

# limite d'indirection - initialisée à zéro == pas de limite
indirect=${SPAMMER_LIMIT:=2}

# # # # données de sortie d'informations 'Chassez le
spammeur' # # # #

# Tout nom de domaine pourrait avoir de nombreuses adresses IP.
# Toute adresse IP pourrait avoir de multiples noms de domaines.
# Du coup, trace des paires uniques adresse-nom.
declare -a known_pair
declare -a reverse_pair

#  En plus des variables de flux de données ; known_address
#+ known_name et list_server, ce qui suit est sorti vers le fichier d'interface
#+ graphique externe.

# Chaîne d'autorité, parent -> champs SOA.
declare -a auth_chain

# Référence la chaîne, nom du parent -> nom du fils
declare -a ref_chain

# Chaîne DNS - nom de domaine -> adresse
declare -a name_address

# Paires de nom et service - nom de domaine -> service
declare -a name_srvc

# Paires de nom et ressource - nom de domaine -> enregistrement de ressource
declare -a name_resource

# Paires de parent et fils - nom de parent -> nom du fils
# Ceci POURRAIT NE PAS être identique au ref_chain qui suit !
declare -a parent_child

# Paires des correspondances d'adresses et des listes noires - adresse->serveur
declare -a address_hits

# Liste les données du fichier d'interface
declare -f _dot_dump
_dot_dump=pend_dummy   # Initialement un no-op

#  Les traces des données sont activées en initialisant la variable
#+ d'environnement SPAMMER_DATA avec le nom d'un fichier sur lequel le script
#+ peut écrire.
declare _dot_file

# Fonction d'aide pour la fonction dump-to-dot-file
# dump_to_dot &lt;array_name&gt; &lt;prefix&gt;
dump_to_dot() {
    local -a _dda_tmp
    local -i _dda_cnt
    local _dda_form='    '${2}'%04u %s\n'
    local IFS=${NO_WSP}
    eval _dda_tmp=\(\ \$\{$1\[@\]\}\ \)
    _dda_cnt=${#_dda_tmp[@]}
```

```
    if [ ${_dda_cnt} -gt 0 ]
    then
        for (( _dda = 0 ; _dda < _dda_cnt ; _dda++ ))
        do
            printf "${_dda_form}" \
                   "${_dda}" "${_dda_tmp[${_dda}]}" >>${_dot_file}
        done
    fi
}

# Qui initialise aussi _dot_dump par cette fonction . . .
dump_dot() {
    local -i _dd_cnt
    echo '# Data vintage: '$(date -R) >${_dot_file}
    echo '# ABS Guide: is_spammer.bash; v2, 2004-msz' >>${_dot_file}
    echo >>${_dot_file}
    echo 'digraph G {' >>${_dot_file}

    if [ ${#known_name[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known domain name nodes' >>${_dot_file}
        _dd_cnt=${#known_name[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    N%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_name[${_dd}]}" >>${_dot_file}
        done
    fi

    if [ ${#known_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known address nodes' >>${_dot_file}
        _dd_cnt=${#known_address[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    A%04u [label="%s"] ;\n' \
                   "${_dd}" "${known_address[${_dd}]}" >>${_dot_file}
        done
    fi

    echo                                    >>${_dot_file}
    echo '/*'                               >>${_dot_file}
    echo ' * Known relationships :: User conversion to'  >>${_dot_file}
    echo ' * graphic form by hand or program required.'  >>${_dot_file}
    echo ' *'                               >>${_dot_file}

    if [ ${#auth_chain[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Authority reference edges followed and field source.'  >>${_dot_file}
        dump_to_dot auth_chain AC
    fi

    if [ ${#ref_chain[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Name reference edges followed and field source.'  >>${_dot_file}
        dump_to_dot ref_chain RC
    fi

    if [ ${#name_address[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->address edges' >>${_dot_file}
        dump_to_dot name_address NA
    fi
```

```
    if [ ${#name_srvc[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->service edges' >>${_dot_file}
        dump_to_dot name_srvc NS
    fi

    if [ ${#name_resource[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known name->resource edges' >>${_dot_file}
        dump_to_dot name_resource NR
    fi

    if [ ${#parent_child[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known parent->child edges' >>${_dot_file}
        dump_to_dot parent_child PC
    fi

    if [ ${#list_server[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known Blacklist nodes' >>${_dot_file}
        _dd_cnt=${#list_server[@]}
        for (( _dd = 0 ; _dd < _dd_cnt ; _dd++ ))
        do
            printf '    LS%04u [label="%s"] ;\n' \
                   "${_dd}" "${list_server[${_dd}]}" >>${_dot_file}
        done
    fi

    unique_lines address_hits address_hits
    if [ ${#address_hits[@]} -gt 0 ]
    then
        echo >>${_dot_file}
        echo '# Known address->Blacklist_hit edges' >>${_dot_file}
        echo '# CAUTION: dig warnings can trigger false hits.' >>${_dot_file}
        dump_to_dot address_hits AH
    fi
    echo          >>${_dot_file}
    echo ' *'     >>${_dot_file}
    echo ' * That is a lot of relationships. Happy graphing.' >>${_dot_file}
    echo ' */'    >>${_dot_file}
    echo '}'      >>${_dot_file}
    return 0
}

# # # # Flux d'exécution 'Chassez le spammeur' # # # #

#  La trace d'exécution est activée en initialisant la variable d'environnement
#+ SPAMMER_TRACE avec le nom d'un fichier sur lequel le script peut écrire.
declare -a _trace_log
declare _log_file

# Fonction pour remplir le journal de traces
trace_logger() {
    _trace_log[${#_trace_log[@]}]=${_pend_current_}
}

# Enregistre le journal des traces vers la variable fichier.
declare -f _log_dump
_log_dump=pend_dummy   # Initialement un no-op.

# Enregistre le journal des traces vers un fichier.
dump_log() {
    local -i _dl_cnt
    _dl_cnt=${#_trace_log[@]}
```

```
    for (( _dl = 0 ; _dl < _dl_cnt ; _dl++ ))
    do
        echo ${_trace_log[${_dl}]} >> ${_log_file}
    done
    _dl_cnt=${#_pending_[@]}
    if [ ${_dl_cnt} -gt 0 ]
    then
        _dl_cnt=${_dl_cnt}-1
        echo '# # # Operations stack not empty # # #' >> ${_log_file}
        for (( _dl = ${_dl_cnt} ; _dl >= 0 ; _dl-- ))
        do
            echo ${_pending_[${_dl}]} >> ${_log_file}
        done
    fi
}

# # # Emballages de l'outil 'dig' # # #
#
#  Ces emballages sont dérivées des exemples affichés dans
#+ dig_wrappers.bash.
#
#  La différence majeur est que ceux-ci retournent leur résultat comme une liste
#+ dans un tableau.
#
#  Voir dig_wrappers.bash pour les détails et utiliser ce script pour développer
#+ toute modification.
#
# # #

# Réponse courte : 'dig' analyse la réponse.

# Recherche avant :: Nom -> Adresse
# short_fwd &lt;domain_name&gt; &lt;array_name&gt;
short_fwd() {
    local -a _sf_reply
    local -i _sf_rc
    local -i _sf_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'sfwd: '${1}
    _sf_reply=( $(dig +short ${1} -c in -t a 2>/dev/null) )
    _sf_rc=$?
    if [ ${_sf_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='## Lookup error '${_sf_rc}' on '${1}' ##'
# [ ${_sf_rc} -ne 9 ] && pend_drop
        return ${_sf_rc}
    else
        # Quelques versions de 'dig' renvoient des avertissements sur stdout.
        _sf_cnt=${#_sf_reply[@]}
        for (( _sf = 0 ; _sf < ${_sf_cnt} ; _sf++ ))
        do
            [ 'x'${_sf_reply[${_sf}]:0:2} == 'x;;' ] &&
                unset _sf_reply[${_sf}]
        done
        eval $2=\( \$\{_sf_reply\[@\]\} \)
    fi
    return 0
}

# Recherche inverse :: Adresse -> Nom
# short_rev &lt;ip_address&gt; &lt;array_name&gt;
short_rev() {
    local -a _sr_reply
    local -i _sr_rc
    local -i _sr_cnt
    IFS=${NO_WSP}
echo -n '.'
# echo 'srev: '${1}
```

```
    _sr_reply=( $(dig +short -x ${1} 2>/dev/null) )
    _sr_rc=$?
    if [ ${_sr_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='## Lookup error '${_sr_rc}' on '${1}'
##'
# [ ${_sr_rc} -ne 9 ] && pend_drop
        return ${_sr_rc}
    else
        # Quelques versions de 'dig' renvoient des avertissements sur stdout.
        _sr_cnt=${#_sr_reply[@]}
        for (( _sr = 0 ; _sr < ${_sr_cnt} ; _sr++ ))
        do
            [ 'x'${_sr_reply[${_sr}]:0:2} == 'x;;' ] &&
                unset _sr_reply[${_sr}]
        done
        eval $2=\( \$\{_sr_reply\[@\]\} \)
    fi
    return 0
}

#  Recherche du format spécial utilisé pour lancer des requêtes sur les serveurs
#+ de listes noires (blacklist).
# short_text &lt;ip_address&gt; &lt;array_name&gt;
short_text() {
    local -a _st_reply
    local -i _st_rc
    local -i _st_cnt
    IFS=${NO_WSP}
# echo 'stxt: '${1}
    _st_reply=( $(dig +short ${1} -c in -t txt 2>/dev/null) )
    _st_rc=$?
    if [ ${_st_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='## Text lookup error '${_st_rc}' on '${1}' ##'
# [ ${_st_rc} -ne 9 ] && pend_drop
        return ${_st_rc}
    else
        # Quelques versions de 'dig' renvoient des avertissements sur stdout.
        _st_cnt=${#_st_reply[@]}
        for (( _st = 0 ; _st < ${#_st_cnt} ; _st++ ))
        do
            [ 'x'${_st_reply[${_st}]:0:2} == 'x;;' ] &&
                unset _st_reply[${_st}]
        done
        eval $2=\( \$\{_st_reply\[@\]\} \)
    fi
    return 0
}

# Les formes longues, aussi connues sous le nom de versions "Analyse toi-même"

# RFC 2782   Recherche de service
# dig +noall +nofail +answer _ldap._tcp.openldap.org -t srv
# _&lt;service&gt;._&lt;protocol&gt;.&lt;domain_name&gt;
# _ldap._tcp.openldap.org. 3600   IN      SRV     0 0 389 ldap.openldap.org.
# domain TTL Class SRV Priority Weight Port Target

# Recherche avant :: Nom -> transfert de zone du pauvre
# long_fwd &lt;domain_name&gt; &lt;array_name&gt;
long_fwd() {
    local -a _lf_reply
    local -i _lf_rc
    local -i _lf_cnt
    IFS=${NO_WSP}
echo -n ':'
# echo 'lfwd: '${1}
    _lf_reply=( $(
        dig +noall +nofail +answer +authority +additional \
```

```
            ${1} -t soa ${1} -t mx ${1} -t any 2>/dev/null) )
    _lf_rc=$?
    if [ ${_lf_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='## Zone lookup error '${_lf_rc}' on
'${1}' ##'
# [ ${_lf_rc} -ne 9 ] && pend_drop
        return ${_lf_rc}
    else
        # Quelques versions de 'dig' renvoient des avertissements sur stdout.
        _lf_cnt=${#_lf_reply[@]}
        for (( _lf = 0 ; _lf < ${_lf_cnt} ; _lf++ ))
        do
            [ 'x'${_lf_reply[${_lf}]:0:2} == 'x;;' ] &&
                unset _lf_reply[${_lf}]
        done
        eval $2=\( \$\{_lf_reply\[@\]\} \)
    fi
    return 0
}
#   La recherche inverse de nom de domaine correspondant à l'adresse IPv6:
#       4321:0:1:2:3:4:567:89ab
#   pourrait donnée (en hexadécimal) :
#   b.a.9.8.7.6.5.0.4.0.0.0.3.0.0.0.2.0.0.0.1.0.0.0.0.0.0.0.1.2.3.4.IP6.ARPA.

# Recherche inverse :: Adresse -> chaîne de délégation du pauvre
# long_rev &lt;rev_ip_address&gt; &lt;array_name&gt;
long_rev() {
    local -a _lr_reply
    local -i _lr_rc
    local -i _lr_cnt
    local _lr_dns
    _lr_dns=${1}'.in-addr.arpa.'
    IFS=${NO_WSP}
echo -n ':'
# echo 'lrev: '${1}
    _lr_reply=( $(
         dig +noall +nofail +answer +authority +additional \
             ${_lr_dns} -t soa ${_lr_dns} -t any 2>/dev/null) )
    _lr_rc=$?
    if [ ${_lr_rc} -ne 0 ]
    then
        _trace_log[${#_trace_log[@]}]='## Delegation lookup error '${_lr_rc}' on '${1}'
##'
# [ ${_lr_rc} -ne 9 ] && pend_drop
        return ${_lr_rc}
    else
        # Quelques versions de 'dig' renvoient des avertissements sur stdout.
        _lr_cnt=${#_lr_reply[@]}
        for (( _lr = 0 ; _lr < ${_lr_cnt} ; _lr++ ))
        do
            [ 'x'${_lr_reply[${_lr}]:0:2} == 'x;;' ] &&
                unset _lr_reply[${_lr}]
        done
        eval $2=\( \$\{_lr_reply\[@\]\} \)
    fi
    return 0
}

## Fonctions spécifiques à l'application ##

# Récupère un nom possible ; supprime root et TLD.
# name_fixup &lt;string&gt;
name_fixup(){
    local -a _nf_tmp
    local -i _nf_end
    local _nf_str
    local IFS
    _nf_str=$(to_lower ${1})
```

```
    _nf_str=$(to_dot ${_nf_str})
    _nf_end=${#_nf_str}-1
    [ ${_nf_str:${_nf_end}} != '.' ] &&
        _nf_str=${_nf_str}'.'
    IFS=${ADR_IFS}
    _nf_tmp=( ${_nf_str} )
    IFS=${WSP_IFS}
    _nf_end=${#_nf_tmp[@]}
    case ${_nf_end} in
    0) # Pas de point, seulement des points
        echo
        return 1
    ;;
    1) # Seulement un TLD.
        echo
        return 1
    ;;
    2) # Pourrait être bon.
       echo ${_nf_str}
       return 0
       # Besoin d'une table de recherche ?
       if [ ${#_nf_tmp[1]} -eq 2 ]
       then # TLD codé suivant le pays.
           echo
           return 1
       else
           echo ${_nf_str}
           return 0
       fi
    ;;
    esac
    echo ${_nf_str}
    return 0
}

# Récupère le(s) entrée(s) originale(s).
split_input() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -i _si_cnt
    local -i _si_len
    local _si_str
    unique_lines uc_name uc_name
    _si_cnt=${#uc_name[@]}
    for (( _si = 0 ; _si < _si_cnt ; _si++ ))
    do
        _si_str=${uc_name[$_si]}
        if is_address ${_si_str}
        then
            uc_address[${#uc_address[@]}]=${_si_str}
            unset uc_name[$_si]
        else
            if ! uc_name[$_si]=$(name_fixup ${_si_str})
            then
                unset ucname[$_si]
            fi
        fi
    done
    uc_name=( ${uc_name[@]} )
    _si_cnt=${#uc_name[@]}
    _trace_log[${#_trace_log[@]}]='## Input '${_si_cnt}' unchecked name input(s). ##'
    _si_cnt=${#uc_address[@]}
    _trace_log[${#_trace_log[@]}]='## Input '${_si_cnt}' unchecked address input(s).
##'
    return 0
}

## Fonctions de découverte -- verrouillage récursif par des données externes ##
## Le début 'si la liste est vide; renvoyer 0' de chacun est requis. ##
```

```
# Limiteur de récursion
# limit_chk() &lt;next_level&gt;
limit_chk() {
    local -i _lc_lmt
    # Vérifiez la limite d'indirection.
    if [ ${indirect} -eq 0 ] || [ $# -eq 0 ]
    then
        # Le choix 'faites-à-chaque-fois'
        echo 1                # Toute valeur le fera.
        return 0              # OK pour continuer.
    else
        # La limite est effective.
        if [ ${indirect} -lt ${1} ]
        then
            echo ${1}         # Quoi que ce soit.
            return 1          # Arrêter ici.
        else
            _lc_lmt=${1}+1    # Augmenter la limite donnée.
            echo ${_lc_lmt}   # L'afficher.
            return 0          # OK pour continuer.
        fi
    fi
}

# Pour chaque nom dans uc_name:
#     Déplacez le nom dans chk_name.
#     Ajoutez les adresses à uc_address.
#     Lancez expand_input_address.
#     Répétez jusqu'à ce que rien de nouveau ne soit trouvé.
# expand_input_name &lt;indirection_limit&gt;
expand_input_name() {
    [ ${#uc_name[@]} -gt 0 ] || return 0
    local -a _ein_addr
    local -a _ein_new
    local -i _ucn_cnt
    local -i _ein_cnt
    local _ein_tst
    _ucn_cnt=${#uc_name[@]}

    if  ! _ein_cnt=$(limit_chk ${1})
    then
        return 0
    fi

    for (( _ein = 0 ; _ein < _ucn_cnt ; _ein++ ))
    do
        if short_fwd ${uc_name[${_ein}]} _ein_new
        then
            for (( _ein_cnt = 0 ; _ein_cnt < ${#_ein_new[@]}; _ein_cnt++ ))
            do
                _ein_tst=${_ein_new[${_ein_cnt}]}
                if is_address ${_ein_tst}
                then
                    _ein_addr[${#_ein_addr[@]}]=${_ein_tst}
                fi
           done
        fi
    done
    unique_lines _ein_addr _ein_addr     # Scrub duplicates.
    edit_exact chk_address _ein_addr     # Scrub pending detail.
    edit_exact known_address _ein_addr   # Scrub already detailed.
    if [ ${#_ein_addr[@]} -gt 0 ]        # Anything new?
    then
        uc_address=( ${uc_address[@]} ${_ein_addr[@]} )
        pend_func expand_input_address ${1}
        _trace_log[${#_trace_log[@]}]='## Added '${#_ein_addr[@]}' unchecked address
input(s). ##'
    fi
    edit_exact chk_name uc_name          # Scrub pending detail.
```

```
    edit_exact known_name uc_name        # Scrub already detailed.
    if [ ${#uc_name[@]} -gt 0 ]
    then
        chk_name=( ${chk_name[@]} ${uc_name[@]}  )
        pend_func detail_each_name ${1}
    fi
    unset uc_name[@]
    return 0
}

# Pour chaque adresse dans uc_address:
#     Déplacez l'adresse vers chk_address.
#     Ajoutez les noms à uc_name.
#     Lancez expand_input_name.
#     Répétez jusqu'à ce que rien de nouveau ne soit trouvé.
# expand_input_address &lt;indirection_limit&gt;
expand_input_address() {
    [ ${#uc_address[@]} -gt 0 ] || return 0
    local -a _eia_addr
    local -a _eia_name
    local -a _eia_new
    local -i _uca_cnt
    local -i _eia_cnt
    local _eia_tst
    unique_lines uc_address _eia_addr
    unset uc_address[@]
    edit_exact been_there_addr _eia_addr
    _uca_cnt=${#_eia_addr[@]}
    [ ${_uca_cnt} -gt 0 ] &&
        been_there_addr=( ${been_there_addr[@]} ${_eia_addr[@]} )

    for (( _eia = 0 ; _eia < _uca_cnt ; _eia++ ))
    do
            if short_rev ${_eia_addr[${_eia}]} _eia_new
            then
                for (( _eia_cnt = 0 ; _eia_cnt < ${#_eia_new[@]} ; _eia_cnt++ ))
                do
                    _eia_tst=${_eia_new[${_eia_cnt}]}
                    if _eia_tst=$(name_fixup ${_eia_tst})
                    then
                        _eia_name[${#_eia_name[@]}]=${_eia_tst}
                    fi
                done
            fi
    done
    unique_lines _eia_name _eia_name     # Scrub duplicates.
    edit_exact chk_name _eia_name        # Scrub pending detail.
    edit_exact known_name _eia_name      # Scrub already detailed.
    if [ ${#_eia_name[@]} -gt 0 ]        # Anything new?
    then
        uc_name=( ${uc_name[@]} ${_eia_name[@]} )
        pend_func expand_input_name ${1}
        _trace_log[${#_trace_log[@]}]='## Added '${#_eia_name[@]}' unchecked name
input(s). ##'
    fi
    edit_exact chk_address _eia_addr     # Scrub pending detail.
    edit_exact known_address _eia_addr   # Scrub already detailed.
    if [ ${#_eia_addr[@]} -gt 0 ]        # Anything new?
    then
        chk_address=( ${chk_address[@]} ${_eia_addr[@]} )
        pend_func detail_each_address ${1}
    fi
    return 0
}

# La réponse de la zone analysez-le-vous-même.
# L'entrée est la liste chk_name.
# detail_each_name &lt;indirection_limit&gt;
detail_each_name() {
```

```
    [ ${#chk_name[@]} -gt 0 ] || return 0
    local -a _den_chk       # Noms à vérifier
    local -a _den_name      # Noms trouvés ici
    local -a _den_address   # Adresses trouvées ici
    local -a _den_pair      # Paires trouvés ici
    local -a _den_rev       # Paires inverses trouvées ici
    local -a _den_tmp       # Ligne en cours d'analyse
    local -a _den_auth      # Contact SOA en cours d'analyse
    local -a _den_new       # La réponse de la zone
    local -a _den_pc        # Parent-Fils devient très rapide
    local -a _den_ref       # Ainsi que la chaîne de référence
    local -a _den_nr        # Nom-Ressource peut être gros
    local -a _den_na        # Nom-Adresse
    local -a _den_ns        # Nom-Service
    local -a _den_achn      # Chaîne d'autorité
    local -i _den_cnt       # Nombre de noms à détailler
    local -i _den_lmt       # Limite d'indirection
    local _den_who          # Named en cours d'exécution
    local _den_rec          # Type d'enregistrement en cours d'exécution
    local _den_cont         # Domaine du contact
    local _den_str          # Correction du nom
    local _den_str2         # Correction inverse
    local IFS=${WSP_IFS}

    # Copie locale, unique de noms à vérifier
    unique_lines chk_name _den_chk
    unset chk_name[@]       # Fait avec des globales.

    # Moins de noms déjà connus
    edit_exact known_name _den_chk
    _den_cnt=${#_den_chk[@]}

    # S'il reste quelque chose, ajoutez à known_name.
    [ ${_den_cnt} -gt 0 ] &&
        known_name=( ${known_name[@]} ${_den_chk[@]} )

    # pour la liste des (précédents) noms inconnus . . .
    for (( _den = 0 ; _den < _den_cnt ; _den++ ))
    do
        _den_who=${_den_chk[${_den}]}
        if long_fwd ${_den_who} _den_new
        then
            unique_lines _den_new _den_new
            if [ ${#_den_new[@]} -eq 0 ]
            then
                _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
            fi

            # Analyser chaque ligne de la réponse.
            for (( _line = 0 ; _line < ${#_den_new[@]} ; _line++ ))
            do
                IFS=${NO_WSP}$'\x09'$'\x20'
                _den_tmp=( ${_den_new[${_line}]} )
                IFS=${WSP_IFS}
                #  Si l'enregistrement est utilisable et n'est pas un message
                #+ d'avertissement . . .
                if [ ${#_den_tmp[@]} -gt 4 ] && [ 'x'${_den_tmp[0]} != 'x;;' ]
                then
                    _den_rec=${_den_tmp[3]}
                    _den_nr[${#_den_nr[@]}]=${_den_who}' '${_den_rec}
                    # Début de RFC1033 (+++)
                    case ${_den_rec} in

                         #&lt;name&gt;  [&lt;ttl&gt;]  [&lt;class&gt;]  SOA
&lt;origin&gt;  &lt;person&gt;
                    SOA) # Début de l'autorité
                        if _den_str=$(name_fixup ${_den_tmp[0]})
                        then
                            _den_name[${#_den_name[@]}]=${_den_str}
```

```
                            _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str}' SOA'
                            #  origine SOA -- nom de domaine de l'enregistrement
                            #+ de la zone maître
                            if _den_str2=$(name_fixup ${_den_tmp[4]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str2}
                                _den_achn[${#_den_achn[@]}]=${_den_who}' '${_den_str2}'
SOA.O'
                            fi
                            # Adresse mail responsable (peut-être boguée).
                            # Possibilité d'un premier.dernier@domaine.nom
                            # ignoré.
                            set -f
                            if _den_str2=$(name_fixup ${_den_tmp[5]})
                            then
                                IFS=${ADR_IFS}
                                _den_auth=( ${_den_str2} )
                                IFS=${WSP_IFS}
                                if [ ${#_den_auth[@]} -gt 2 ]
                                then
                                     _den_cont=${_den_auth[1]}
                                     for (( _auth = 2 ; _auth < ${#_den_auth[@]}
; _auth++ ))
                                     do

_den_cont=${_den_cont}'.'${_den_auth[${_auth}]}
                                     done
                                     _den_name[${#_den_name[@]}]=${_den_cont}'.'
                                     _den_achn[${#_den_achn[@]}]=${_den_who}'
'${_den_cont}'. SOA.C'
                                fi
                            fi
                            set +f
                        fi
                    ;;


                    A) # Enregistrement d'adresse IP(v4)
                        if _den_str=$(name_fixup ${_den_tmp[0]})
                        then
                            _den_name[${#_den_name[@]}]=${_den_str}
                            _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
                            _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
                            _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' A'
                        else
                            _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
                            _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
                            _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain A'
                        fi
                        _den_address[${#_den_address[@]}]=${_den_tmp[4]}
                        _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
                    ;;

                    NS) #  Enregistrement du nom de serveur
                        #  Nom de domaine en cours de service (peut être autre
                        #+ chose que l'actuel)
                        if _den_str=$(name_fixup ${_den_tmp[0]})
                        then
                            _den_name[${#_den_name[@]}]=${_den_str}
                            _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' NS'

                            # Nom du domaine du fournisseur de services
                            if _den_str2=$(name_fixup ${_den_tmp[4]})
                            then
                                _den_name[${#_den_name[@]}]=${_den_str2}
                                _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}'
NSH'
                                _den_ns[${#_den_ns[@]}]=${_den_str2}' NS'
                                _den_pc[${#_den_pc[@]}]=${_den_str}' '${_den_str2}
```

```
                        fi
                    fi
                ;;

                MX) # Enregistrement du serveur de mails
                    # Nom de domaine en service (jokers non gérés ici)
                    if _den_str=$(name_fixup ${_den_tmp[0]})
                    then
                        _den_name[${#_den_name[@]}]=${_den_str}
                        _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MX'
                    fi
                    # Nom du domaine du fournisseur de service
                    if _den_str=$(name_fixup ${_den_tmp[5]})
                    then
                        _den_name[${#_den_name[@]}]=${_den_str}
                        _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' MXH'
                        _den_ns[${#_den_ns[@]}]=${_den_str}' MX'
                        _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                    fi
                ;;

                PTR) # Enregistrement de l'adresse inverse
                     # Nom spécial
                    if _den_str=$(name_fixup ${_den_tmp[0]})
                    then
                        _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' PTR'
                        # Nom d'hôte (pas un CNAME)
                        if _den_str2=$(name_fixup ${_den_tmp[4]})
                        then
                            _den_rev[${#_den_rev[@]}]=${_den_str}' '${_den_str2}
                            _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str2}'
PTRH'
                            _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                        fi
                    fi
                ;;

                AAAA) # Enregistrement de l'adresse IP(v6)
                    if _den_str=$(name_fixup ${_den_tmp[0]})
                    then
                        _den_name[${#_den_name[@]}]=${_den_str}
                        _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' '${_den_str}
                        _den_na[${#_den_na[@]}]=${_den_str}' '${_den_tmp[4]}
                        _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' AAAA'
                    else
                        _den_pair[${#_den_pair[@]}]=${_den_tmp[4]}' unknown.domain'
                        _den_na[${#_den_na[@]}]='unknown.domain '${_den_tmp[4]}
                        _den_ref[${#_den_ref[@]}]=${_den_who}' unknown.domain'
                    fi
                    # Aucun travaux sur les adresses IPv6
                        _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_tmp[4]}
                ;;

                CNAME) # Enregistrement du nom de l'alias
                       # Pseudo
                    if _den_str=$(name_fixup ${_den_tmp[0]})
                    then
                        _den_name[${#_den_name[@]}]=${_den_str}
                        _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CNAME'
                        _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                    fi
                    # Nom d'hôte
                    if _den_str=$(name_fixup ${_den_tmp[4]})
                    then
                        _den_name[${#_den_name[@]}]=${_den_str}
                        _den_ref[${#_den_ref[@]}]=${_den_who}' '${_den_str}' CHOST'
                        _den_pc[${#_den_pc[@]}]=${_den_who}' '${_den_str}
                    fi
                ;;
```

```
#                   TXT)
#                   ;;
                    esac
                fi
            done
        else # Erreur de recherche == enregistrement 'A' 'adresse inconnue'
            _den_pair[${#_den_pair[@]}]='0.0.0.0 '${_den_who}
        fi
    done

    # Tableau des points de contrôle grandit.
    unique_lines _den_achn _den_achn      # Fonctionne mieux, tout identique.
    edit_exact auth_chain _den_achn       # Fonctionne mieux, éléments uniques.
    if [ ${#_den_achn[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        auth_chain=( ${auth_chain[@]} ${_den_achn[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_ref _den_ref      # Fonctionne mieux, tout identique.
    edit_exact ref_chain _den_ref       # Fonctionne mieux, éléments uniques.
    if [ ${#_den_ref[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        ref_chain=( ${ref_chain[@]} ${_den_ref[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_na _den_na
    edit_exact name_address _den_na
    if [ ${#_den_na[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_address=( ${name_address[@]} ${_den_na[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_ns _den_ns
    edit_exact name_srvc _den_ns
    if [ ${#_den_ns[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_srvc=( ${name_srvc[@]} ${_den_ns[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_nr _den_nr
    edit_exact name_resource _den_nr
    if [ ${#_den_nr[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        name_resource=( ${name_resource[@]} ${_den_nr[@]} )
        IFS=${WSP_IFS}
    fi

    unique_lines _den_pc _den_pc
    edit_exact parent_child _den_pc
    if [ ${#_den_pc[@]} -gt 0 ]
    then
        IFS=${NO_WSP}
        parent_child=( ${parent_child[@]} ${_den_pc[@]} )
        IFS=${WSP_IFS}
    fi

    # Mise à jour de la liste known_pair (adresse et nom).
    unique_lines _den_pair _den_pair
    edit_exact known_pair _den_pair
    if [ ${#_den_pair[@]} -gt 0 ]  # Rien de nouveau?
```

```
    then
        IFS=${NO_WSP}
        known_pair=( ${known_pair[@]} ${_den_pair[@]} )
        IFS=${WSP_IFS}
    fi

    # Mise à jour de la liste des pairs inversés.
    unique_lines _den_rev _den_rev
    edit_exact reverse_pair _den_rev
    if [ ${#_den_rev[@]} -gt 0 ]   # Rien de nouveau ?
    then
        IFS=${NO_WSP}
        reverse_pair=( ${reverse_pair[@]} ${_den_rev[@]} )
        IFS=${WSP_IFS}
    fi

    # Vérification de la limite d'indirection -- abandon si elle est atteinte.
    if ! _den_lmt=$(limit_chk ${1})
    then
        return 0
    fi

    #  Le moteur d'exécution est LIFO. L'ordre des opérations en attente est
    #+ important.
    # Avons-nous défini de nouvelles adresses ?
    unique_lines _den_address _den_address    # Scrub duplicates.
    edit_exact known_address _den_address     # Scrub already processed.
    edit_exact un_address _den_address        # Scrub already waiting.
    if [ ${#_den_address[@]} -gt 0 ]          # Anything new?
    then
        uc_address=( ${uc_address[@]} ${_den_address[@]} )
        pend_func expand_input_address ${_den_lmt}
        _trace_log[${#_trace_log[@]}]='## Added '${#_den_address[@]}' unchecked
address(s). ##'
    fi

    # Avons-nous trouvé de nouveaux noms ?
    unique_lines _den_name _den_name          # Scrub duplicates.
    edit_exact known_name _den_name           # Scrub already processed.
    edit_exact uc_name _den_name              # Scrub already waiting.
    if [ ${#_den_name[@]} -gt 0 ]             # Anything new?
    then
        uc_name=( ${uc_name[@]} ${_den_name[@]} )
        pend_func expand_input_name ${_den_lmt}
        _trace_log[${#_trace_log[@]}]='## Added '${#_den_name[@]}' unchecked name(s).
##'
    fi
    return 0
}

# Réponse de délégation analysez-le-vous-même
# L'entrée est la liste chk_address.
# detail_each_address &amp;lt;indirection_limit&amp;gt;
detail_each_address() {
    [ ${#chk_address[@]} -gt 0 ] || return 0
    unique_lines chk_address chk_address
    edit_exact known_address chk_address
    if [ ${#chk_address[@]} -gt 0 ]
    then
        known_address=( ${known_address[@]} ${chk_address[@]} )
        unset chk_address[@]
    fi
    return 0
}

## Fonctions de sortie spécifiques à l'application ##

# Affiche joliment les pairs connues.
report_pairs() {
```

```
    echo
    echo 'Known network pairs.'
    col_print known_pair 2 5 30

    if [ ${#auth_chain[@]} -gt 0 ]
    then
        echo
        echo 'Known chain of authority.'
        col_print auth_chain 2 5 30 55
    fi

    if [ ${#reverse_pair[@]} -gt 0 ]
    then
        echo
        echo 'Known reverse pairs.'
        col_print reverse_pair 2 5 55
    fi
    return 0
}

#  Vérifie une adresse contre la liste des serveurs
#+ faisant partie de la liste noire.
# Un bon endroit pour capturer avec GraphViz :
# address-&gt;status(server(reports))
# check_lists &lt;ip_address&gt;
check_lists() {
    [ $# -eq 1 ] || return 1
    local -a _cl_fwd_addr
    local -a _cl_rev_addr
    local -a _cl_reply
    local -i _cl_rc
    local -i _ls_cnt
    local _cl_dns_addr
    local _cl_lkup

    split_ip ${1} _cl_fwd_addr _cl_rev_addr
    _cl_dns_addr=$(dot_array _cl_rev_addr)'.'
    _ls_cnt=${#list_server[@]}
    echo '    Checking address '${1}
    for (( _cl = 0 ; _cl < _ls_cnt ; _cl++ ))
    do
        _cl_lkup=${_cl_dns_addr}${list_server[${_cl}]}
        if short_text ${_cl_lkup} _cl_reply
        then
            if [ ${#_cl_reply[@]} -gt 0 ]
            then
                echo '        Records from '${list_server[${_cl}]}
                address_hits[${#address_hits[@]}]=${1}' '${list_server[${_cl}]}
                _hs_RC=2
                for (( _clr = 0 ; _clr < ${#_cl_reply[@]} ; _clr++ ))
                do
                    echo '            '${_cl_reply[${_clr}]}
                done
            fi
        fi
    done
    return 0
}

## La colle habituelle de l'application ##

# Qui l'a fait ?
credits() {
   echo
   echo "Guide d'écriture avancée des scripts Bash : is_spammer.bash, v2,
2004-msz"
}

# Comment l'utiliser ?
```

```
# (Voir aussi, "Quickstart" à la fin de ce script.)
usage() {
    cat <<-'_usage_statement_'
    Le script is_spammer.bash requiert un ou deux arguments.

    arg 1) Pourrait être :
        a) Un nom de domaine
        b) Une adresse IPv4
        c) Le nom d'un fichier avec des noms et adresses mélangés, un par ligne.

    arg 2) Pourrait être :
        a) Un nom de domaine d'un serveur Blacklist
        b) Le nom d'un fichier contenant une liste de noms de domaine Blacklist,
           un domaine par ligne.
        c) Si non présent, une liste par défaut de serveurs Blacklist (libres)
           est utilisée.
        d) Si un fichier vide, lisible, est donné, la recherche de serveurs
           Blacklist est désactivée.

    Toutes les sorties du script sont écrites sur stdout.

    Codes de retour: 0 -> Tout est OK, 1 -> Échec du script,
                     2 -> Quelque chose fait partie de la liste noire.

    Requiert le programme externe 'dig' provenant des programmes DNS de 'bind-9'
    Voir http://www.isc.org

    La limite de la profondeur de recherche du nom de domaine est par défaut de
    deux niveaux.
    Initialisez la variable d'environnement SPAMMER_LIMIT pour modifier ceci.
    SPAMMER_LIMIT=0 signifie 'illimité'

    La limite peut aussi être initialisée sur la ligne de commande.
    Si arg#1 est un entier, la limite utilise cette valeur
    puis les règles d'arguments ci-dessus sont appliquées.

    Initialiser la variable d'environnemnt 'SPAMMER_DATA' à un nom de fichier
    demandera au script d'écrire un fichier graphique GraphViz.

    Pour la version de développement ;
    Initialiser la variable d'environnement 'SPAMMER_TRACE' avec un nom de
    fichier demandera au moteur d'exécution de tracer tous les appels de
    fonction.

_usage_statement_
}

# La liste par défaut des serveurs Blacklist :
# Plusieurs choix, voir : http://www.spews.org/lists.html

declare -a default_servers
# Voir : http://www.spamhaus.org (Conservateur, bien maintenu)
default_servers[0]='sbl-xbl.spamhaus.org'
# Voir : http://ordb.org (Relais mail ouverts)
default_servers[1]='relays.ordb.org'
# Voir : http://www.spamcop.net/ (Vous pouvez rapporter les spammeurs ici)
default_servers[2]='bl.spamcop.net'
# Voir : http://www.spews.org (Un système de détection rapide)
default_servers[3]='l2.spews.dnsbl.sorbs.net'
# Voir : http://www.dnsbl.us.sorbs.net/using.shtml
default_servers[4]='dnsbl.sorbs.net'
# Voir : http://dsbl.org/usage (Différentes listes de relai de mail)
default_servers[5]='list.dsbl.org'
default_servers[6]='multihop.dsbl.org'
default_servers[7]='unconfirmed.dsbl.org'

# Argument utilisateur #1
setup_input() {
    if [ -e ${1} ] && [ -r ${1} ]  # Nom d'un fichier lisible
```

```
    then
        file_to_array ${1} uc_name
        echo 'Using filename >'${1}'< as input.'
    else
        if is_address ${1}          # Adresse IP ?
        then
            uc_address=( ${1} )
            echo 'Starting with address >'${1}'<'
        else                        # Doit être un nom.
            uc_name=( ${1} )
            echo 'Starting with domain name >'${1}'<'
        fi
    fi
    return 0
}

# Argument utilisateur #2
setup_servers() {
    if [ -e ${1} ] && [ -r ${1} ]  # Nom d'un fichier lisible
    then
        file_to_array ${1} list_server
        echo 'Using filename >'${1}'< as blacklist server list.'
    else
        list_server=( ${1} )
        echo 'Using blacklist server >'${1}'<'
    fi
    return 0
}

# Variable d'environnement utilisateur SPAMMER_TRACE
live_log_die() {
    if [ ${SPAMMER_TRACE:=} ]    # Journal de trace ?
    then
        if [ ! -e ${SPAMMER_TRACE} ]
        then
            if ! touch ${SPAMMER_TRACE} 2>/dev/null
            then
                pend_func echo $(printf '%q\n' \
                'Unable to create log file >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        else
            if [ ! -w ${SPAMMER_TRACE} ]
            then
                pend_func echo $(printf '%q\n' \
                'Unable to write log file >'${SPAMMER_TRACE}'<')
                pend_release
                exit 1
            fi
            _log_file=${SPAMMER_TRACE}
            echo '' > ${_log_file}
            _pend_hook_=trace_logger
            _log_dump=dump_log
        fi
    fi
    return 0
}

# Variable d'environnement utilisateur SPAMMER_DATA
data_capture() {
    if [ ${SPAMMER_DATA:=} ]    # Tracer les données ?
    then
        if [ ! -e ${SPAMMER_DATA} ]
        then
            if ! touch ${SPAMMER_DATA} 2>/dev/null
```

```
            then
                pend_func echo $(printf '%q]n' \
                'Unable to create data output file >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
        else
            if [ ! -w ${SPAMMER_DATA} ]
            then
                pend_func echo $(printf '%q\n' \
                'Unable to write data output file >'${SPAMMER_DATA}'<')
                pend_release
                exit 1
            fi
            _dot_file=${SPAMMER_DATA}
            _dot_dump=dump_dot
        fi
    fi
    return 0
}

# Réunir les arguments spécifiés par l'utilisateur.
do_user_args() {
    if [ $# -gt 0 ] && is_number $1
    then
        indirect=$1
        shift
    fi

    case $# in                 # L'utilisateur nous traite-t'il correctement?
        1)
            if ! setup_input $1    # Vérification des erreurs.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            list_server=( ${default_servers[@]} )
            _list_cnt=${#list_server[@]}
            echo 'Using default blacklist server list.'
            echo 'Search depth limit: '${indirect}
            ;;
        2)
            if ! setup_input $1    # Vérification des erreurs.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            if ! setup_servers $2  # Vérification des erreurs.
            then
                pend_release
                $_log_dump
                exit 1
            fi
            echo 'Search depth limit: '${indirect}
            ;;
        *)
            pend_func usage
            pend_release
            $_log_dump
            exit 1
            ;;
    esac
    return 0
}
```

```
# Un outil à but général de déboguage.
# list_array &amp;lt;array_name&amp;gt;
list_array() {
    [ $# -eq 1 ] || return 1  # Un argument requis.

    local -a _la_lines
    set -f
    local IFS=${NO_WSP}
    eval _la_lines=\(\ \$\{$1\[@\]\}\ \)
    echo
    echo "Element count "${#_la_lines[@]}" array "${1}
    local _ln_cnt=${#_la_lines[@]}

    for (( _i = 0; _i < ${_ln_cnt}; _i++ ))
    do
        echo 'Element '$_i' >'${_la_lines[$_i]}'<'
    done
    set +f
    return 0
}

## Code 'Chez le spammeur' ##
pend_init                               # Initialisation du moteur à pile.
pend_func credits                       # Dernière chose à afficher.

## Gérer l'utilisateur ##
live_log_die                            #  Initialiser le journal de trace de
                                        #+ déboguage.
data_capture                            #  Initialiser le fichier de capture de
                                        #+ données.
echo
do_user_args $@

## N'a pas encore quitté - Il y a donc un peu d'espoir ##
# Groupe de découverte - Le moteur d'exécution est LIFO - queue en ordre
# inverse d'exécution.
_hs_RC=0                                # Code de retour de Chassez le spammeur
pend_mark
    pend_func report_pairs              # Paires nom-adresse rapportées.

    # Les deux detail_* sont des fonctions mutuellement récursives.
    # Elles mettent en queue les fonctions expand_* functions si nécessaire.
    # Ces deux (les dernières de ???) sortent de la récursion.
    pend_func detail_each_address       #  Obtient toutes les ressources
                                        #+ des adresses.
    pend_func detail_each_name          #  Obtient toutes les ressources
                                        #+ des noms.

    #  Les deux expand_* sont des fonctions mutuellement récursives,
    #+ qui mettent en queue les fonctions detail_* supplémentaires si
    #+ nécessaire.
    pend_func expand_input_address 1    #  Étend les noms en entrées par des
                                        #+ adresses.
    pend_func expand_input_name 1       #  Étend les adresses en entrées par des
                                        #+ noms.

    # Commence avec un ensemble unique de noms et d'adresses.
    pend_func unique_lines uc_address uc_address
    pend_func unique_lines uc_name uc_name

    # Entrée mixe séparée de noms et d'adresses.
    pend_func split_input
pend_release

## Paires rapportées -- Liste unique d'adresses IP trouvées
echo
_ip_cnt=${#known_address[@]}
if [ ${#list_server[@]} -eq 0 ]
then
```

```
    echo 'Blacklist server list empty, none checked.'
else
    if [ ${_ip_cnt} -eq 0 ]
    then
        echo 'Known address list empty, none checked.'
    else
        _ip_cnt=${_ip_cnt}-1   # Start at top.
        echo 'Checking Blacklist servers.'
        for (( _ip = _ip_cnt ; _ip >= 0 ; _ip-- ))
        do
            pend_func check_lists $( printf '%q\n' ${known_address[$_ip]} )
        done
    fi
fi
pend_release
$_dot_dump                  # Fichier graphique
$_log_dump                  # Trace d'exécution
echo


##########################################
# Exemple de sortie provenant du script #
##########################################
:<<-'_is_spammer_outputs_'

./is_spammer.bash 0 web4.alojamentos7.com

Starting with domain name >web4.alojamentos7.com<
Using default blacklist server list.
Search depth limit: 0
.:...:::::..:::..:::.....::..::..:::.....::
Known network pairs.
    66.98.208.97            web4.alojamentos7.com.
    66.98.208.97            ns1.alojamentos7.com.
    69.56.202.147           ns2.alojamentos.ws.
    66.98.208.97            alojamentos7.com.
    66.98.208.97            web.alojamentos7.com.
    69.56.202.146           ns1.alojamentos.ws.
    69.56.202.146           alojamentos.ws.
    66.235.180.113          ns1.alojamentos.org.
    66.235.181.192          ns2.alojamentos.org.
    66.235.180.113          alojamentos.org.
    66.235.180.113          web6.alojamentos.org.
    216.234.234.30          ns1.theplanet.com.
    12.96.160.115           ns2.theplanet.com.
    216.185.111.52          mail1.theplanet.com.
    69.56.141.4             spooling.theplanet.com.
    216.185.111.40          theplanet.com.
    216.185.111.40          www.theplanet.com.
    216.185.111.52          mail.theplanet.com.

Checking Blacklist servers.
    Checking address 66.98.208.97
        Records from dnsbl.sorbs.net
            "Spam Received See: http://www.dnsbl.sorbs.net/lookup.shtml?66.98.208.97"
    Checking address 69.56.202.147
    Checking address 69.56.202.146
    Checking address 66.235.180.113
    Checking address 66.235.181.192
    Checking address 216.185.111.40
    Checking address 216.234.234.30
    Checking address 12.96.160.115
    Checking address 216.185.111.52
    Checking address 69.56.141.4

Advanced Bash Scripting Guide: is_spammer.bash, v2, 2004-msz

_is_spammer_outputs_
```

```
exit ${_hs_RC}

######################################################################
#  Le script ignore tout ce qui se trouve entre ici et la fin #
#+ à cause de la commande 'exit' ci-dessus.                       #
######################################################################



Quickstart
==========

 Prérequis

  Bash version 2.05b ou 3.00 (bash --version)
  Une version de Bash supportant les tableaux. Le support des tableaux est
  inclus dans les configurations par défaut de Bash.

  'dig,' version 9.x.x (dig $HOSTNAME, voir la première ligne en sortie)
  Une version de dig supportant les options +short.
  Voir dig_wrappers.bash pour les détails.


 Prérequis optionnels

  'named', un programme de cache DNS local. N'importe lequel conviendra.
  Faites deux fois : dig $HOSTNAME
  Vérifier près de la fin de la sortie si vous voyez:
    SERVER: 127.0.0.1#53
  Ceci signifie qu'il fonctionne.


 Support optionnel des graphiques

  'date', un outil standard *nix. (date -R)

  dot un programme pour convertir le fichier de description graphique en
  un diagramme. (dot -V)
  Fait partie de l'ensemble des programmes Graph-Viz.
  Voir [http://www.research.att.com/sw/tools/graphviz||GraphViz]

  'dotty', un éditeur visuel pour les fichiers de description graphique.
  Fait aussi partie de l'ensemble des programmes Graph-Viz.




 Quick Start

Dans le même répertoire que le script is_spammer.bash;
Lancez : ./is_spammer.bash

 Détails d'utilisation

1. Choix de serveurs Blacklist.

  (a) Pour utiliser les serveurs par défaut, liste intégrée : ne rien faire.

  (b) Pour utiliser votre propre liste :

    i. Créez un fichier avec un seul serveru Blacklist par ligne.

    ii. Indiquez ce fichier en dernier argument du script.

  (c) Pour utiliser un seul serveur Blacklist : Dernier argument de ce script.

  (d) Pour désactiver les recherches Blacklist :

    i. Créez un fichier vide (touch spammer.nul)
```

```
       Le nom du fichier n'a pas d'importance.

    ii. Indiquez ce nom en dernier argument du script.

2. Limite de la profondeur de recherche.

  (a) Pour utiliser la valeur par défaut de 2 : ne rien faire.

  (b) Pour configurer une limite différente :
      Une limite de 0 signifie illimitée.

    i. export SPAMMER_LIMIT=1
       ou tout autre limite que vous désirez.

    ii. OU indiquez la limite désirée en premier argument de ce script.

3. Journal de trace de l'exécution (optionnel).

  (a) Pour utiliser la configuration par défaut (sans traces) : ne rien faire.

  (b) Pour écrire dans un journal de trace :
      export SPAMMER_TRACE=spammer.log
      ou tout autre nom de fichier que vous voulez.

4. Fichier de description graphique optionnel.

  (a) Pour utiliser la configuration par défaut (sans graphique) : ne rien
      faire.

  (b) Pour écrire un fichier de description graphique Graph-Viz :
      export SPAMMER_DATA=spammer.dot
      ou tout autre nom de fichier que vous voulez.

5. Où commencer la recherche.

  (a) Commencer avec un simple nom de domaine :

    i. Sans limite de recherche sur la ligne de commande : Premier
       argument du script.

    ii. Avec une limite de recherche sur la ligne de commande : Second
        argument du script.

  (b) Commencer avec une simple adresse IP :

    i. Sans limite de recherche sur la ligne de commande : Premier
       argument du script.

    ii. Avec une limite de recherche sur la ligne de commande : Second
        argument du script.

  (c) Commencer avec de nombreux noms et/ou adresses :
      Créer un fichier avec un nom ou une adresse par ligne.
      Le nom du fichier n'a pas d'importance.

    i. Sans limite de recherche sur la ligne de commande : Fichier comme premier
       argument du script.

    ii. Avec une limite de recherche sur la ligne de commande : Fichier comme
        second argument du script.

6. Que faire pour l'affichage en sortie.

  (a) Pour visualiser la sortie à l'écran : ne rien faire.

  (b) Pour sauvegarder la sortie dans un fichier : rediriger stdout vers un
      fichier.

  (c) Pour désactiver la sortie : rediriger stdout vers /dev/null.
```

```
7. Fin temporaire de la phase de décision.
   appuyez sur RETURN
   attendez (sinon, regardez les points et les virgules).

8. De façon optionnelle, vérifiez le code de retour.

  (a) Code de retour 0: Tout est OK

  (b) Code de retour 1: Échec du script de configuration

  (c) Code de retour 2: Quelque chose était sur la liste noire.

9. Où est mon graphe (diagramme) ?

Le script ne produit pas directement un graphe (diagramme).
Il produit seulement un fichier de description graphique. Vous pouvez utiliser
ce fichier qui a été créé par le programme 'dot'.

Jusqu'à l'édition du fichier de description pour décrire les relations que vous
souhaitez montrer, tout ce que vous obtenez est un ensemble de noms et de noms
d'adresses.

Toutes les relations découvertes par le script font partie d'un bloc en
commentaires dans le fichier de description graphique, chacun ayant un en-tête
descriptif.

L'édition requise pour tracer une ligne entre une paire de noeuds peut se faire
avec un éditeur de texte à partir des informations du fichier descripteur.

Avec ces lignes quelque part dans le fichier descripteur :

# Known domain name nodes

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;



# Known address nodes

A0000 [label="61.141.32.197"] ;



/*

# Known name->address edges

NA0000 third.guardproof.info. 61.141.32.197



# Known parent->child edges

PC0000 guardproof.info. third.guardproof.info.

 */

Modifiez ceci en les lignes suivantes après avoir substitué les identifiants de
noeuds avec les relations :

# Known domain name nodes

N0000 [label="guardproof.info."] ;

N0002 [label="third.guardproof.info."] ;
```

```
# Known address nodes

A0000 [label="61.141.32.197"] ;


# PC0000 guardproof.info. third.guardproof.info.

N0000->N0002 ;


# NA0000 third.guardproof.info. 61.141.32.197

N0002->A0000 ;


/*

# Known name->address edges

NA0000 third.guardproof.info. 61.141.32.197


# Known parent->child edges

PC0000 guardproof.info. third.guardproof.info.

 */

Lancez le programme 'dot' et vous avez votre premier diagramme réseau.

En plus des formes graphiques habituelles, le fichier de description inclut des
paires/données de format similaires, décrivant les services, les enregistrements
de zones (sous-graphe ?), des adresses sur liste noire et d'autres choses
pouvant être intéressante à inclure dans votre graphe. Cette information
supplémentaire pourrait être affichée comme différentes formes de noeuds,
couleurs, tailles de lignes, etc.

Le fichier de description peut aussi être lu et édité par un script Bash (bien
sûr). Vous devez être capable de trouver la plupart des fonctions requises à
l'intérieur du script "is_spammer.bash".

# Fin de Quickstart.


Note Supplémentaire
==== ==============

Michael Zick indique qu'il existe un "makeviz.bash" interactif sur
le site Web rediris.es. Impossible de donner le lien complet car
ce n'est pas un site accessible publiquement.
```

Un autre script anti-spam.

Exemple A.31. Chasse aux spammeurs

```
#!/bin/bash
# whx.sh : recherche d'un spammeur via "whois"
# Auteur: Walter Dnes
# Révisions légères (première section) par l'auteur du guide ABS.
# Utilisé dans le guide ABS avec sa permission.
```

```
#  Nécessite la version 3.x ou ultérieure de Bash pour fonctionner
#+ (à cause de l'utilisation de l'opérateur =~).
#  Commenté par l'auteur du script et par l'auteur du guide ABS.



E_MAUVAISARGS=65    # Argument manquant en ligne de commande.
E_SANSHOTE=66       # Hôte introuvable.
E_DELAIDEPASSE=67   # Délai dépassée pour la recherche de l'hôte.
E_NONDEF=68         # D'autres erreurs (non définies).
ATTENTEHOTE=10      # Spécifiez jusqu'à 10 secondes pour la réponse à la requête.
                    # L'attente réelle pourrait être un peu plus longue.
FICHIER_RESULTAT=whois.txt   # Fichier en sortie.
PORT=4321


if [ -z "$1" ]      # Vérification de l'argument (requis) en ligne de commande.
then
  echo "Usage: $0 nom de domaine ou adresse IP"
  exit $E_MAUVAISARGS
fi


if [[ "$1" =~ "[a-zA-Z][a-zA-Z]$" ]]  # Se termine avec deux caractères alphabetiques ?
then                                  # C'est un nom de domaine et nous devons faire
une recherche d'hôte.
  ADR_IP=$(host -W $ATTENTEHOTE $1 | awk '{print $4}')
                                      # Recherche d'hôte pour récupérer l'adresse IP.
                                      # Extraction du champ final.
else
  ADR_IP="$1"                         # L'argument en ligne de commande était une
adresse IP.
fi

echo; echo "L'adresse IP est "ADR_IP""; echo

if [ -e "$FICHIER_RESULTAT" ]
then
  rm -f "$FICHIER_RESULTAT"
  echo "Ancien fichier résultat \"$FICHIER_RESULTAT\" supprimé."; echo
fi


#  Vérification.
#  (Cette section nécessite plus de travail.)
#  ========================================
if [ -z "$ADR_IP" ]
# Sans réponse.
then
  echo "Hôte introuvable !"
  exit $E_SANSHOTE    # Quitte.
fi

if [[ "$ADR_IP" =~ "^[;;]" ]]
#  ;; connection timed out; no servers could be reached
then
  echo "Délai de recherche dépassé !"
  exit $E_DELAIDEPASSE   # On quitte.
fi

if [[ "$ADR_IP" =~ "[(NXDOMAIN)]$" ]]
#  Host xxxxxxxxx.xxx not found: 3(NXDOMAIN)
then
  echo "Hôte introuvable !"
  exit $E_SANSHOTE    # On quitte.
fi

if [[ "$ADR_IP" =~ "[(SERVFAIL)]$" ]]
```

```
#  Host xxxxxxxxx.xxx not found: 2(SERVFAIL)
then
  echo "Hôte introuvable !"
  exit $E_SANSHOTE    # On quitte.
fi



# ======================== Corps principal du script ========================

AFRINICquery() {
#  Définit la fonction qui envoit la requête à l'AFRINIC.
#+ Affiche une notification à l'écran, puis exécute la requête
#+ en redirigeant la sortie vers $FICHIER_RESULTAT.

  echo "Recherche de $ADR_IP dans whois.afrinic.net"
  whois -h whois.afrinic.net "$ADR_IP" > $FICHIER_RESULTAT

#  Vérification de la présence de la référence à un rwhois.
#  Avertissement sur un serveur rwhois.infosat.net non fonctionnel
#+ et tente une requête rwhois.
  if grep -e "^remarks: .*rwhois\.[^ ]\+" "$FICHIER_RESULTAT"
  then
    echo " " >> $FICHIER_RESULTAT
    echo "***" >> $FICHIER_RESULTAT
    echo "***" >> $FICHIER_RESULTAT
    echo "Avertissement : rwhois.infosat.net ne fonctionnait pas le 2005/02/02" >>
$FICHIER_RESULTAT
    echo "               lorsque ce script a été écrit." >> $FICHIER_RESULTAT
    echo "***" >> $FICHIER_RESULTAT
    echo "***" >> $FICHIER_RESULTAT
    echo " " >> $FICHIER_RESULTAT
    RWHOIS=`grep "^remarks: .*rwhois\.[^ ]\+" "$FICHIER_RESULTAT" | tail -n 1 |\
    sed "s/\(^.*\)\(rwhois\..*\)\(:4.*\)/\2/"`
    whois -h ${RWHOIS}:${PORT} "$ADR_IP" >> $FICHIER_RESULTAT
  fi
}

APNICquery() {
  echo "Recherche de $ADR_IP dans whois.apnic.net"
  whois -h whois.apnic.net "$ADR_IP" > $FICHIER_RESULTAT

#  Just  about  every  country has its own internet registrar.
#  I don't normally bother consulting them, because the regional registry
#+ usually supplies sufficient information.
#  There are a few exceptions, where the regional registry simply
#+ refers to the national registry for direct data.
#  These are Japan and South Korea in APNIC, and Brasil in LACNIC.
#  The following if statement checks $FICHIER_RESULTAT (whois.txt) for the presence
#+ of "KR" (South Korea) or "JP" (Japan) in the country field.
#  If either is found, the query is re-run against the appropriate
#+ national registry.

  if grep -E "^country:[ ]+KR$" "$FICHIER_RESULTAT"
  then
    echo "Recherche de $ADR_IP dans whois.krnic.net"
    whois -h whois.krnic.net "$ADR_IP" >> $FICHIER_RESULTAT
  elif grep -E "^country:[ ]+JP$" "$FICHIER_RESULTAT"
  then
    echo "Recherche de $ADR_IP dans whois.nic.ad.jp"
    whois -h whois.nic.ad.jp "$ADR_IP"/e >> $FICHIER_RESULTAT
  fi
}

ARINquery() {
  echo "Recherche de $ADR_IP dans whois.arin.net"
  whois -h whois.arin.net "$ADR_IP" > $FICHIER_RESULTAT

#  Several large internet providers listed by ARIN have their own
```

```
#+ internal whois service, referred to as "rwhois".
#  A large block of IP addresses is listed with the provider
#+ under the ARIN registry.
#  To get the IP addresses of 2nd-level ISPs or other large customers,
#+ one has to refer to the rwhois server on port 4321.
#  I originally started with a bunch of "if" statements checking for
#+ the larger providers.
#  This approach is unwieldy, and there's always another rwhois server
#+ that I didn't know about.
#  A more elegant approach is to check $FICHIER_RESULTAT for a reference
#+ to a whois server, parse that server name out of the comment section,
#+ and re-run the query against the appropriate rwhois server.
#  The parsing looks a bit ugly, with a long continued line inside
#+ backticks.
#  But it only has to be done once, and will work as new servers are added.
#@   ABS Guide author comment: it isn't all that ugly, and is, in fact,
#@+  an instructive use of Regular Expressions.

  if grep -E "^Comment: .*rwhois.[^ ]+" "$FICHIER_RESULTAT"
  then
    RWHOIS=`grep -e "^Comment:.*rwhois\.[^ ]\+" "$FICHIER_RESULTAT" | tail -n 1 |\
    sed "s/^\(.*\)\(rwhois\.[^ ]\+\)\(.*$\)/\2/"`
    echo "Recherche de $ADR_IP dans ${RWHOIS}"
    whois -h ${RWHOIS}:${PORT} "$ADR_IP" >> $FICHIER_RESULTAT
  fi
}

LACNICquery() {
  echo "Recherche de $ADR_IP dans whois.lacnic.net"
  whois -h whois.lacnic.net "$ADR_IP" > $FICHIER_RESULTAT

#  The  following if statement checks $FICHIER_RESULTAT (whois.txt) for the presence of
#+ "BR" (Brasil) in the country field.
#  If it is found, the query is re-run against whois.registro.br.

  if grep -E "^country:[ ]+BR$" "$FICHIER_RESULTAT"
  then
    echo "Recherche de $ADR_IP dans whois.registro.br"
    whois -h whois.registro.br "$ADR_IP" >> $FICHIER_RESULTAT
  fi
}

RIPEquery() {
  echo "Recherche de $ADR_IP dans whois.ripe.net"
  whois -h whois.ripe.net "$ADR_IP" > $FICHIER_RESULTAT
}

#  Initialise quelques variables.
#  * slash8 est l'octet le plus significatif
#  * slash16 consiste aux deux octets les plus significatifs
#  * octet2 est le deuxième octet le plus significatif




slash8=`echo $IPADDR | cut -d. -f 1`
  if [ -z "$slash8" ]  # Encore une autre vérification.
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi
slash16=`echo $IPADDR | cut -d. -f 1-2`
#                          ^ Point spécifié comme délimiteur pour cut.
  if [ -z "$slash16" ]
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi
octet2=`echo $slash16 | cut -d. -f 2`
```

```
  if [ -z "$octet2" ]
  then
    echo "Undefined error!"
    exit $E_UNDEF
  fi


#  Vérification de différentes étrangetés.
#  Il n'y a pas d'intérêts à chercher ces adresses.

if [ $slash8 == 0 ]; then
  echo $ADR_IP est l\'espace '"This Network"' \; Pas de requêtes
elif [ $slash8 == 10 ]; then
  echo $ADR_IP est l\'espace RFC1918 \; Pas de requêtes
elif [ $slash8 == 14 ]; then
  echo $ADR_IP est l\'espace '"Public Data Network"' \; Pas de requêtes
elif [ $slash8 == 127 ]; then
  echo $ADR_IP est l\'espace loopback \; Pas de requêtes
elif [ $slash16 == 169.254 ]; then
  echo $ADR_IP est l\'espace link-local \; Pas de requêtes
elif [ $slash8 == 172 ] && [ $octet2 -ge 16 ] && [ $octet2 -le 31 ];then
  echo $ADR_IP est l\'espace RFC1918 \; Pas de requêtes
elif [ $slash16 == 192.168 ]; then
  echo $ADR_IP est l\'espace RFC1918 \; Pas de requêtes
elif [ $slash8 -ge 224 ]; then
  echo $ADR_IP est l\'espace Multicast ou réservé \; Pas de requêtes
elif [ $slash8 -ge 200 ] && [ $slash8 -le 201 ]; then LACNICquery "$ADR_IP"
elif [ $slash8 -ge 202 ] && [ $slash8 -le 203 ]; then APNICquery "$ADR_IP"
elif [ $slash8 -ge 210 ] && [ $slash8 -le 211 ]; then APNICquery "$ADR_IP"
elif [ $slash8 -ge 218 ] && [ $slash8 -le 223 ]; then APNICquery "$ADR_IP"

#  Si nous sommes arrivés ici sans prendre de décision, demander à l'ARIN.
#  Si une référence est trouvée dans $FICHIER_RESULTAT à l'APNIC, l'AFRINIC, LACNIC ou
RIPE,
#+ alors envoyez une requête au serveur whois approprié.

else
  ARINquery "$ADR_IP"
  if grep "whois.afrinic.net" "$FICHIER_RESULTAT"; then
    AFRINICquery "$ADR_IP"
  elif grep -E "^OrgID:[ ]+RIPE$" "$FICHIER_RESULTAT"; then
    RIPEquery "$ADR_IP"
  elif grep -E "^OrgID:[ ]+APNIC$" "$FICHIER_RESULTAT"; then
    APNICquery "$ADR_IP"
  elif grep -E "^OrgID:[ ]+LACNIC$" "$FICHIER_RESULTAT"; then
    LACNICquery "$ADR_IP"
  fi
fi

#@  ----------------------------------------------------------------
#   Essayez aussi :
#   wget http://logi.cc/nw/whois.php3?ACTION=doQuery&amp;DOMAIN=$ADR_IP
#@  ----------------------------------------------------------------

#  Nous avons fini maintenant toutes les requêtes.
#  Affiche une copie du résultat final à l'écran.

cat $FICHIER_RESULTAT
# Ou "less $FICHIER_RESULTAT" . . .


exit 0

#@  Commentaires de l'auteur du guide ABS :
#@  Rien de particulièrement intéressant ici,
#@+ mais quand même un outil très utile pour chasser les spammeurs.
#@  Bien sûr, le script peut être un peu nettoyé et il est encore un peu bogué
#@+ (exercice pour le lecteur) mais, en fait, c'est un joli code de
#@+ Walter Dnes.
```

```
#@  Merci !
```

L'interface de « Little Monster » pour wget.

Exemple A.32. Rendre *wget* plus facile à utiliser

```
#!/bin/bash
# wgetter2.bash

# Auteur : Little Monster [monster@monstruum.co.uk]
# ==> Utilisé dans le guide ABS avec la permission de l'auteur du script.
# ==> Ce script a toujours besoin de débogage et de corrections (exercice
# ==> laissé au lecteur).
# ==> Il pourrait aussi bénéficier de meilleurs commentaires.


#  Ceci est wgetter2 --
#+ un script Bash rendant wget un peu plus facile à utiliser
#+ et évitant de la frappe clavier.

#  Écrit avec attention par Little Monster.
#  Plus ou moins complet le 02/02/2005.
#  Si vous pensez que ce script est améliorable,
#+ envoyez-moi un courrier électronique à : monster@monstruum.co.uk
# ==> et mettez en copie l'auteur du guide ABS.
#  Ce script est sous licence GPL.
#  Vous êtes libre de le copier, modifier, ré-utiliser,
#+ mais, s'il-vous-plait, ne dites pas que vous l'avez écrit.
#  À la place, indiquez vos changements ici.

# ======================================================================
# journal des modifications :

# 07/02/2005.  Corrections par Little Monster.
# 02/02/2005.  Petits ajouts de Little Monster.
#              (Voir après # ++++++++++++ )
# 29/01/2005.  Quelques petites modifications de style et nettoyage de l'auteur
#              du guide ABS.
#              Ajout des codes d'erreur.
# 22/11/2004.  Fin de la version initiale de la seconde version de wgetter :
#              wgetter2 est né.
# 01/12/2004.  Modification de la fonction 'runn' de façon à ce qu'il
#              fonctionne de deux façons --
#              soit en demandant le nom d'un fichier soit en le récupérant sur
#              la ligne de commande.
# 01/12/2004.  Gestion sensible si aucune URL n'est fournie.
# 01/12/2004.  Boucle des options principales, de façon à ne pas avoir à
#              rappeller wgetter 2 tout le temps.
#              À la place, fonctionne comme une session.
# 01/12/2004.  Ajout d'une boucle dans la fonction 'runn'.
#              Simplifié et amélioré.
# 01/12/2004.  Ajout de state au paramètrage de récursion.
#              Active la ré-utilisation de la valeur précédente.
# 05/12/2004.  Modification de la routine de détection de fichiers dans la
#              fonction 'runn' de façon à ce qu'il ne soit pas gêné par des
#              valeurs vides et pour qu'il soit plus propre.
# 01/02/2004.  Ajout de la routine de récupération du cookie à partir de
#              la dernière version (qui n'est pas encore prête), de façon à ne
#              pas avoir à codé en dur les chemins.
# ======================================================================

# Codes d'erreur pour une sortie anormale.
E_USAGE=67                  # Message d'usage, puis quitte.
E_SANS_OPTS=68              # Aucun argument en ligne de commande.
E_SANS_URLS=69              # Aucune URL passée au script.
E_SANS_FICHIERSAUVEGARDE=70  # Aucun nom de fichier de sortie passé au script.
E_SORTIE_UTILISATEUR=71      # L'utilisateur a décidé de quitter.
```

```
#  Commande wget par défaut que nous voulons utiliser.
#  C'est l'endroit où la changer, si nécessaire.
#  NB: si vous utilisez un proxy, indiquez http_proxy = yourproxy dans .wgetrc.
#  Sinon, supprimez --proxy=on, ci-dessous.
# =========================================================================
CommandeA="wget -nc -c -t 5 --progress=bar --random-wait --proxy=on -r"
# =========================================================================



# -------------------------------------------------------------------
# Initialisation de quelques autres variables avec leur explications.

pattern=" -A .jpg,.JPG,.jpeg,.JPEG,.gif,.GIF,.htm,.html,.shtml,.php"
                    #  Options de wget pour ne récupérer que certain types de
                    #+ fichiers. Mettre en commentaire si inutile
today=`date +%F`    # Utilisé pour un nom de fichier.
home=$HOME          # Utilise HOME pour configurer une variable interne.
                    #  Au cas où d'autres chemins sont utilisés, modifiez cette
                    #+ variable.
depthDefault=3      # Configure un niveau de récursion sensible.
Depth=$depthDefault # Sinon, le retour de l'utilisateur ne sera pas intégré.
RefA=""             # Configure la page blanche de référence.
Flag=""             #  Par défaut, ne sauvegarde rien,
                    #+ ou tout ce qui pourrait être voulu dans le futur.
lister=""           # Utilisé pour passer une liste d'url directement à wget.
Woptions=""         # Utilisé pour passer quelques options à wget.
inFile=""           # Utilisé pour la fonction run.
newFile=""          # Utilisé pour la fonction run.
savePath="$home/w-save"
Config="$home/.wgetter2rc"
                    #  Quelques variables peuvent être stockées,
                    #+ si elles sont modifiées en permanence à l'intérieur de ce
                    #+ script.
Cookie_List="$home/.cookielist"
                    # Pour que nous sachions où sont conservés les cookies...
cFlag=""            # Une partie de la routine de sélection du cookie.

#  Définissez les options disponibles. Lettres faciles à modifier ici si
#+ nécessaire.
#  Ce sont les options optionnelles ; vous n'avez pas besoin d'attendre
#+ qu'elles vous soient demandées.

save=s   # Sauvegarde la commande au lieu de l'exécuter.
cook=c   # Modifie le cookie pour cette session.
help=h   # Guide d'usage.
list=l   # Passe à wget l'option -i et la liste d'URL.
runn=r   # Lance les commandes sauvegardées comme argument de l'option.
inpu=i   # Lance les commandes sauvegardées de façon interactive.
wopt=w   # Autorise la saisie d'options à passer directement à wget.
# -------------------------------------------------------------------


if [ -z "$1" ]; then   # Soyons sûr de donner quelque chose à manger à wget.
   echo "Vous devez entrer au moins une RLS ou une option!"
   echo "-$help pour l'utilisation."
   exit $E_SANS_OPTS
fi



# +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# ajout ajout ajout ajout ajout ajout ajout ajout ajout ajout ajout ajout

if [ ! -e "$Config" ]; then   #  Vérification de l'existence du fichier de
                              #+ configuration.
   echo "Création du fichier de configuration, $Config"
```

```
   echo "# Ceci est le fichier de configuration pour wgetter2" > "$Config"
   echo "# Vos paramètres personnalisés seront sauvegardés dans ce fichier" \
     >> "$Config"
else
   source $Config             #  Import des variables que nous avons initialisé
                              #+ en dehors de ce script.
fi

if [ ! -e "$Cookie_List" ]; then
   # Configure une liste de cookie, si elle n'existe pas.
   echo "Recherche des cookies..."
   find -name cookies.txt >> $Cookie_List   # Crée une liste des cookies.
fi #  Isole ceci dans sa propre instruction 'if',
   #+ au cas où nous serions interrompu durant la recherche.

if [ -z "$cFlag" ]; then # Si nous n'avons pas encore fait ceci...
   echo                    # Ajoute un espacement après l'invite de la commande.
   echo "Il semble que vous n'avez pas encore configuré votre source de cookies."
   n=0                     # S'assure que le compteur ne contient pas de valeurs.
   while read; do
      Cookies[$n]=$REPLY #  Place les cookies que nous avons trouvé dans un
                         #+ tableau.
      echo "$n) ${Cookies[$n]}"  # Crée un menu.
      n=$(( n + 1 ))     # Incrémente le comteur.
   done < $Cookie_List   # Remplit l'instruction read.
   echo "Saisissez le nombre de cookies que vous souhaitez utiliser."
   echo "Si vous ne voulez pas utiliser de cookie, faites simplement RETURN."
   echo
   echo "Je ne vous demanderais plus ceci. Éditez $Config"
   echo "si vous décidez de le changer ultérieurement"
   echo "ou utilisez l'option -${cook} pour des modifications sur une session."
   read
   if [ ! -z $REPLY ]; then   # L'utilisateur n'a pas seulement faire ENTER.
      Cookie=" --load-cookies ${Cookies[$REPLY]}"
      # Initialise la variable ici ainsi que dans le fichier de configuration.

      echo "Cookie=\" --load-cookies ${Cookies[$REPLY]}\"" >> $Config
   fi
   echo "cFlag=1" >> $Config  #  Pour que nous nous rappelions de ne pas le
                              #+ demander de nouveau.
fi

# fin section ajoutée fin section ajoutée fin section ajoutée
# ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++



# Une autre variable.
# Celle-ci pourrait être ou pas sujet à variation.
# Un peu comme le petit affichage.
CookiesON=$Cookie
# echo "cookie file is $CookiesON" # Pour débogage.
# echo "home is ${home}"           # Pour débogage. Faites attention à celui-ci!


wopts()
{
echo "Entrer les options à fournir à wget."
echo "Il est supposé que vous savez ce que vous faites."
echo
echo "Vous pouvez passer leurs arguments ici aussi."
# C'est-à-dire que tout ce qui est saisi ici sera passé à wget.

read Wopts
# Lire les options à donner à wget.

Woptions=" $Wopts"
#         ^  Pourquoi cet espace initial ?
# Affecter à une autre variable.
```

```
# Pour le plaisir, ou pour tout autre chose...

echo "options ${Wopts} fournies à wget"
# Principalement pour du débogage.
# Est joli.

return
}


save_func()
{
echo "Les paramètres vont être sauvegardés."
if [ ! -d $savePath ]; then  #  Vérifie si le répertoire existe.
   mkdir $savePath           #  Crée le répertoire pour la sauvegarde
                             #+ si ce dernier n'existe pas.
fi

Flag=S
# Indique au dernier bout de code ce qu'il faut faire.
# Positionne un drapeau car le boulot est effectué dans la partie principale.

return
}


usage() # Indique comment cela fonctionne.
{
    echo "Bienvenue dans wgetter. C'est une interface pour wget."
    echo "Il lancera en permanence wget avec ces options :"
    echo "$CommandeA"
    echo "et le modèle de correspondance: $modele (que vous pouvez changer en"
    echo "haut du script)."
    echo "Il vous demandera aussi une profondeur de récursion depth et si vous"
    echo "souhaitez utiliser une page de référence."
    echo "Wgetter accepte les options suivantes :"
    echo ""
    echo "-$help : Affiche cette aide."
    echo "-$save : Sauvegarde la commande dans un fichier"
    echo "$savePath/wget-($today) au lieu de l'exécuter."
    echo "-$runn : Exécute les commandes wget sauvegardées au lieu d'en"
    echo "commencer une nouvelle --"
    echo "Saisissez le nom du fichier comme argument de cette option."
    echo "-$inpu : Exécute les commandes wget sauvegardées, de façon"
    echo "interactive -- "
    echo "Le script vous demandera le nom du fichier."
    echo "-$cook : Modifie le fichier des cookies pour cette session."
    echo "-$list : Indique à wget d'utiliser les URL à partir d'une liste"
    echo "plutôt que sur la ligne de commande."
    echo "-$wopt : Passe toute autre option directement à wget."
    echo ""
    echo "Voir la page man de wget pour les options supplémentaires que vous"
    echo "pouvez lui passer."
    echo ""

    exit $E_USAGE  # Fin ici. Ne rien exécuter d'autre.
}



list_func() #  Donne à l'utilisateur l'option pour utiliser l'option -i de wget,
            #+ et une liste d'URL.
{
while [ 1 ]; do
   echo "Saisissez le nom du fichier contenant les URL (appuyez sur q si vous"
   echo "avez changé d'idée)."
   read urlfile
   if [ ! -e "$urlfile" ] && [ "$urlfile" != q ]; then
       # Recherche un fichier ou l'option de sortie.
```

```
       echo "Ce fichier n'existe pas!"
   elif [ "$urlfile" = q ]; then   # Vérifie l'option de sortie.
       echo "N'utilise pas de liste d'URL."
       return
   else
      echo "Utilisation de $urlfile."
      echo "Si vous m'avez fourni des URL sur la ligne de commandes,"
      echo "je les utiliserais en premier."
                  # Indique le comportement standard de wget à l'utilisateur.
      lister=" -i $urlfile" # C'est ce que nous voulons fournir à wget.
      return
   fi
done
}


cookie_func()  #  Donne à l'utilisateur l'option d'utiliser un fichier
               #+ cookie différent.
{
while [ 1 ]; do
   echo "Modification du fichier cookie. Appuyez sur return si vous ne voulez "
   echo "pas le changer."
   read Cookies
   # NB: Ceci n'est pas la même chose que Cookie, un peu plus tôt.
   # Il y a un 's' à la fin.
   if [ -z "$Cookies" ]; then                  # Clause d'échappement.
      return
   elif [ ! -e "$Cookies" ]; then
      echo "Le fichier n'existe pas. Essayez de nouveau." # On continue...
   else
       CookiesON=" --load-cookies $Cookies"  # Le fichier est bon -- utilisons-le!
       return
   fi
done
}


run_func()
{
if [ -z "$OPTARG" ]; then
# Teste pour voir si nous utilisons les options en ligne ou la requête.
   if [ ! -d "$savePath" ]; then  # Au cas où le répertoire n'existe pas...
      echo "$savePath ne semble pas exister."
      echo "Merci de fournir un chemin et un nom de fichiers pour les commandes"
      echo "wget sauvegardées :"
      read newFile
         until [ -f "$newFile" ]; do  #  Continue jusqu'à ce que nous obtenions
                                      #+ quelque chose.
            echo "Désolé, ce fichier n'existe pas. Essayez de nouveau."
            # Essaie réellement d'avoir quelque chose.
            read newFile
         done

# ------------------------------------------------------------------------
#         if [ -z ( grep wget ${newfile} ) ]; then
          # Suppose qu'ils n'ont pas encore le bon fichier.
#         echo "Désolé, ce fichier ne contient pas de commandes wget.
#         echo "Annulation."
#         exit
#         fi
#
# Ce code est bogué.
# Il ne fonctionne réellement pas.
# Si vous voulez le corriger, n'hésitez pas !
# ------------------------------------------------------------------------

      filePath="${newFile}"
   else
   echo "Le chemin de sauvegarde est $savePath"
```

```
      echo "Merci de saisir le nom du fichier que vous souhaitez utiliser."
      echo "Vous avez le choix entre :"
      ls $savePath                               # Leur donne un choix.
      read inFile
         until [ -f "$savePath/$inFile" ]; do     # Continuez jusqu'à obtention.
            if [ ! -f "${savePath}/${inFile}" ]; then
                                                 # Si le fichier n'existe pas.
               echo "Désolé, ce fichier n'existe pas."
               echo " Faites votre choix à partir de :"
               ls $savePath                          # Si une erreur est faite.
               read inFile
            fi
         done
      filePath="${savePath}/${inFile}"  # En faire une variable...
   fi
else filePath="${savePath}/${OPTARG}"   # qui peut être beaucoup de choses...
fi

if [ ! -f "$filePath" ]; then           # Si nous obtenons un fichier bogué.
   echo "Vous n'avez pas spécifié un fichier convenable."
   echo "Lancez tout d'abord ce script avec l'option -${save}."
   echo "Annulation."
   exit $E_SANS_FICHIERSAUVEGARDE
fi
echo "Utilisation de : $filePath"
while read; do
    eval $REPLY
    echo "Fin : $REPLY"
done < $filePath  # Remplit le fichier que nous utilisons avec une boucle while.

exit
}



# Récupération de toute option que nous utilisons pour ce script.
# Ceci est basé sur la démo de "Learning The Bash Shell" (O'Reilly).
while getopts ":$save$cook$help$list$runn:$inpu$wopt" opt
do
  case $opt in
     $save) save_func;;   #  Sauvegarde de quelques sessions wgetter pour plus
                          #  tard.
     $cook) cookie_func;; #  Modifie le fichier cookie.
     $help) usage;;       #  Obtient de l'aide.
     $list) list_func;;   #  Autorise wget à utiliser une liste d'URL.
     $runn) run_func;;    #  Utile si vous appelez wgetter à partir d'un script
                          #+ cron par exemple.
     $inpu) run_func;;    #  Lorsque vous ne connaissez pas le nom des fichiers.
     $wopt) wopts;;       #  Passe les options directement à wget.
        \?) echo "Option invalide."
            echo "Utilisez -${wopt} si vous voulez passer les options "
            echo "directement à to wget,"
            echo "ou -${help} pour de l'aide";;      # Récupère quelque chose.
  esac
done
shift $((OPTIND - 1))     # Opérations magiques avec $#.


if [ -z "$1" ] && [ -z "$lister" ]; then
                          #  Nous devrions laisser au moins une URL sur la
                          #+ ligne de commande à moins qu'une liste ne soit
                          #+ utilisée - récupère les lignes de commandes vides.
   echo "Aucune URL fournie ! Vous devez les saisir sur la même ligne "
   echo "que wgetter2."
   echo "Par exemple,  wgetter2 http://somesite http://anothersite."
   echo "Utilisez l'option $help pour plus d'informations."
   exit $E_SANS_URLS        # Quitte avec le bon code d'erreur.
fi
```

```
URLS=" $@"
#  Utilise ceci pour que la liste d'URL puisse être modifié si nous restons dans
#+ la boucle d'option.

while [ 1 ]; do
   # C'est ici que nous demandons les options les plus utilisées.
   # (Pratiquement pas changées depuis la version 1 de wgetter)
   if [ -z $curDepth ]; then
      Current=""
   else Current=" La valeur courante est $curDepth"
   fi
       echo "A quelle profondeur dois-je aller ? "
       echo "(entier: valeur par défaut $depthDefault.$Current)"
       read Depth   # Récursion -- A quelle profondeur allons-nous ?
       inputB=""    # Réinitialise ceci à rien sur chaque passe de la boucle.
       echo "Saisissez le nom de la page de référence (par défaut, aucune)."
       read inputB  # Nécessaire pour certains sites.

       echo "Voulez-vous que la sortie soit tracée sur le terminal"
       echo "(o/n, par défaut, oui) ?"
       read noHide  # Sinon, wget le tracera simplement dans un fichier.

       case $noHide in
          # Maintenant, vous me voyez, maintenant, vous ne me voyez plus.
          o|O ) hide="";;
          n|N ) hide=" -b";;
            * ) hide="";;
       esac

       if [ -z ${Depth} ]; then       #  L'utilisateur a accepté la valeur par
                                      #+ défaut ou la valeur courante,
                                      #+ auquel cas Depth est maintenant vide.
          if [ -z ${curDepth} ]; then #  Vérifie si Depth a été configuré
                                      #+ sur une précédente itération.
             Depth="$depthDefault"    #  Configure la profondeur de récursion
                                      #+ par défaut si rien de défini
                                      #+ sinon, l'utilise.
          else Depth="$curDepth"      #  Sinon, utilisez celui configuré
                                      #+ précédemment.
          fi
       fi
   Recurse=" -l $Depth"               #  Initialise la profondeur.
   curDepth=$Depth                    #  Se rappeler de ce paramètrage la
                                      #+ prochaine fois.

       if [ ! -z $inputB ]; then
          RefA=" --referer=$inputB"   #  Option à utiliser pour la page de
                                      #+ référence.
       fi


WGETTER="${CommandeA}${modele}${hide}${RefA}${Recurse}${CookiesON}${lister}${Woptions}${URLS
   #  Crée une chaîne contenant le lot complet...
   #  NB: pas d'espace imbriqués.
   #  Ils sont dans les éléments individuels si aucun n'est vide,
   #+ nous n'obtenons pas d'espace supplémentaire.

   if [ -z "${CookiesON}" ] && [ "$cFlag" = "1" ] ; then
       echo "Attention -- impossible de trouver le fichier cookie."
       #  Ceci pourrait changer, au cas où l'utilisateur aurait choisi de ne
       #+ pas utiliser les cookies.
   fi

   if [ "$Flag" = "S" ]; then
      echo "$WGETTER" >> $savePath/wget-${today}
      #  Crée un nom de fichier unique pour aujourd'hui
      #+ ou y ajoute les informations s'il existe déjà.
      echo "$inputB" >> $savePath/site-list-${today}
      #  Crée une liste pour qu'il soit plus simple de s'y référer plus tard,
```

```
      #+ car la commande complète est un peu confuse.
      echo "Commande sauvegardée dans le fichier $savePath/wget-${today}"
           # Indication pour l'utilisateur.
      echo "URL de la page de référence sauvegardé dans le fichier "
      echo "$savePath/site-list-${today}"
           # Indication pour l'utilisateur.
      Saver=" avec les options sauvegardées"
      #  Sauvegarde ceci quelque part, de façon à ce qu'il apparaisse dans la
      #+ boucle si nécessaire.
   else
       echo "**********************"
       echo "*****Récupération*****"
       echo "**********************"
       echo ""
       echo "$WGETTER"
       echo ""
       echo "**********************"
       eval "$WGETTER"
   fi

       echo ""
       echo "Continue avec$Saver."
       echo "Si vous voulez stopper, appuyez sur q."
       echo "Sinon, saisissez des URL :"
       # Laissons-les continuer. Indication sur les options sauvegardées.

       read
       case $REPLY in        # Nécessaire de changer ceci par une clause 'trap'.
          q|Q ) exit $E_SORTIE_UTILISATEUR;;  # Exercice pour le lecteur ?
            * ) URLS=" $REPLY";;
       esac

       echo ""
done


exit 0
```

Exemple A.33. Un script de *podcasting*

```
#!/bin/bash

# bashpodder.sh:
# Par Linc 10/1/2004
# Trouve le dernier script sur :
# http://linc.homeunix.org:8080/scripts/bashpodder
# Dernière révision 14/12/2004 - Beaucoup de contributeurs !
# Si vous l'utilisez et y avez ajouté quelques améliorations ou commentaires,
#+ envoyez-moi un courrier électronique (linc POINT fessenden CHEZ gmail POINT com)
# J'apprécierais beaucoup !

# ==>  Commentaires supplémentaires du guide ABS.

# ==>  L'auteur de ce script a donné gentimment sa permission
# ==>+ pour son ajout dans le guide ABS.


# ==> ################################################################
#
# ==> Qu'est-ce que "podcasting" ?

# ==> C'est l'envoi d'émissions de radio sur Internet.
# ==> Ces émissions peuvent être écoutées sur des iPod ainsi que sur
#+==> d'autres lecteurs de fichiers musicaux.

# ==> Ce script rend ceci possible.
# ==> Voir la documentation sur le site de l'auteur du script.
```

```
# ==> ######################################################################


# Rend ce script compatible avec crontab :
cd $(dirname $0)
# ==> Change de répertoire par celui où ce script réside.

# repdonnees est le répertoire où les fichiers podcasts ont été sauvegardés :
repdonnees=$(date +%Y-%m-%d)
# ==> Créera un répertoire de nom : YYYY-MM-DD

# Vérifie et crée repdonnees si nécessaire :
if test ! -d $repdonnees
        then
        mkdir $repdonnees
fi

# Supprime tout fichier temporaire :
rm -f temp.log

# Lit le fichier bp.conf et récupère toute URL qui ne se trouve pas dans le fichier
podcast.log :
while read podcast
        do # ==> L'action principale suit.
        fichier=$(wget -q $podcast -O - | tr '\r' '\n' | tr \' \" | \
        sed -n 's/.*url="\([^"]*\)".*/\1/p')
        for url in $fichier
                do
                echo $url >> temp.log
                if ! grep "$url" podcast.log > /dev/null
                        then
                        wget -q -P $repdonnees "$url"
                fi
                done
        done < bp.conf

# Déplace le journal créé dynamiquement dans le journal permanent :
cat podcast.log >> temp.log
sort temp.log | uniq > podcast.log
rm temp.log
# Crée une liste musicale m3u :
ls $repdonnees | grep -v m3u > $repdonnees/podcast.m3u


exit 0

##################################################
Pour une approche différente de l'écriture de script pour le Podcasting,
voir l'article de Phil Salkie,
"Internet Radio to Podcast with Shell Tools"
dans le numéro de septembre 2005 du LINUX JOURNAL,
http://www.linuxjournal.com/article/8171
##################################################
```

Exemple A.34. Sauvegarde de nuit pour un disque firewire

```
#!/bin/bash
# nightly-backup.sh
# http://www.richardneill.org/source.php#nightly-backup-rsync
# Copyright (c) 2005 Richard Neill &lt;backup@richardneill.org&gt;.
# Ce logiciel libre est sous licence GNU GPL.
# ==> Inclus dans le guide ABS avec l'aimable autorisation de l'auteur du script.
# ==> (Merci !)

#  Ceci réalise une sauvegarde de l'ordinateur hôte vers un disque dur firewire
#+ connecté localement en utilisant rsync et ssh.
```

```
#  Il exécute ensuite une rotation des sauvegardes.
#  Exécutez-la via cron tous les jours à 5h du matin.
#  Cela ne sauvegarde que le répertoire principal.
#  Si le propriétaire (autre que l'utilisateur) doit être conservé,
#+ alors exécutez le processus rsync en tant que root (et ajoutez le -o).
#  Nous sauvegardons tous les jours pendant sept jours,
#+ puis chaque semaine pendant quatre semaines,
#+ puis chaque mois pendant trois mois.

#  Voir http://www.mikerubel.org/computers/rsync_snapshots/
#+ pour plus d'informations sur la théorie.
#  À sauvegarder sous : $HOME/bin/nightly-backup_firewire-hdd.sh

#  Bogues connus :
#  ---------------
#  i)  Idéalement, nous voulons exclure ~/.tmp et les caches du navigateur.

#  ii) Si l'utilisateur est devant son ordinateur à 5h du matin
#+     et que les fichiers sont modifiés alors que le rsync est en cours,
#+     alors la branche SAUVEGARDE_AUCASOU est appelée.
#      D'une certaine façon, c'est une fonctionnalité
#+     mais cela cause aussi une "fuite d'espace disque".




##### DÉBUT DE LA SECTION DE CONFIGURATION ####################################
UTILISATEUR_LOCAL=rjn         #  Utilisateur dont le répertoire principal sera
                              #+ sauvegardé.
POINT_MONTAGE=/backup         #  Point de montage du répertoire de sauvegarde.
                              #  Pas de slash à la fin !
                              #  Il doit être unique
                              #+ (par exemple en utilisant un lien symbolique udev)
REP_SOURCE=/home/$UTILISATEUR_LOCAL  # Pas de slash à la fin - important pour rsync.
REP_DEST_SAUVE=$POINT_MONTAGE/backup/`hostname -s`.${UTILISATEUR_LOCAL}.nightly_backup
ESSAI_A_BLANC=false           #  Si vrai, appelle rsync avec -n, réalisant un test.
                              #  Commentez ou configurez à faux pour une utilisation
                              #+ normale.
VERBEUX=false                 #  Si vrai, rend rsync verbeux.
                              #  Commentez ou configurez à faux sinon.
COMPRESSIONION=false          #  Si vrai, compresse.
                              #  Bon pour internet, mauvais sur LAN.
                              #  Commentez ou configurez à faux sinon.

### Codes d'erreur ###
E_VAR_NON_CONF=64
E_LIGNECOMMANDE=65
E_ECHEC_MONTAGE=70
E_PASREPSOURCE=71
E_NONMONTE=72
E_SAUVE=73
##### FIN DE LA SECTION DE CONFIGURATION ######################################


# Vérifie que toutes les variables importantes sont configurées :
if [ -z "$UTILISATEUR_LOCAL" ] ||
   [ -z "$REP_SOURCE" ] ||
   [ -z "$POINT_MONTAGE" ]  ||
   [ -z "$REP_DEST_SAUVE" ]
then
   echo "Une des variables n'est pas configurée ! Modifiez le fichier $0. ÉCHEC DE LA
SAUVEGARDE."
   exit $E_VAR_NON_CONF
fi

if [ "$#" != 0 ]  # Si des paramètres en ligne de commande...
then              # Document(ation) en ligne.
  cat <<-FINDUTEXTE
```

```
    Sauvegarde quotienne automatique exécutée par cron.
    Lisez les sources pour plus de détails : $0
    Le répertoire de sauvegarde est $REP_DEST_SAUVE .
    Il sera créé si nécessaire ; une initialisation est inutile.

    ATTENTION : le contenu de $REP_DEST_SAUVE est l'objet de rotation.
    Les répertoires nommés 'backup.\$i' seront éventuellement supprimés.
    Nous conservons des répertoires pour chaque jour sur sept jours (1-8),
    puis pour chaque semaine sur quatre semaines (9-12),
    puis pour chaque mois sur trois mois (13-15).

    Vous pouvez ajouter ceci à votre crontab en utilisant 'crontab -e'
    #  Fichiers sauvegardés : $REP_SOURCE dans $REP_DEST_SAUVE
    #+ chaque nuit à 3:15 du matin
         15 03 * * * /home/$UTILISATEUR_LOCAL/bin/nightly-backup_firewire-hdd.sh

    N'oubliez pas de vérifier que les sauvegardes fonctionnent, surtout si vous
    ne lisez pas le mail de cron !"
        FINDUTEXTE
   exit $E_LIGNECOMMANDE
fi


# Analyse des options.
# ====================

if [ "$ESSAI_A_BLANC" == "true" ]; then
  ESSAI_A_BLANC="-n"
  echo "ATTENTION"
  echo "CECI EST UN TEST SIMPLE !"
  echo "Aucune donnée ne sera réellement transférée !"
else
  ESSAI_A_BLANC=""
fi

if [ "$VERBEUX" == "true" ]; then
  VERBEUX="-v"
else
  VERBEUX=""
fi

if [ "$COMPRESSION" == "true" ]; then
  COMPRESSION="-z"
else
  COMPRESSION=""
fi


#  Chaque semaine (en fait tous les huit jours) et chaque mois,
#+ des sauvegardes supplémentaires seront effectuées.
JOUR_DU_MOIS=`date +%d`            # Jour du mois (01..31).
if [ $JOUR_DU_MOIS = 01 ]; then    # Premier du mois.
  DEBUTMOIS=true
elif [ $JOUR_DU_MOIS = 08 \
    -o $JOUR_DU_MOIS = 16 \
    -o $JOUR_DU_MOIS = 24 ]; then
    # Jour 8,16,24
    # (on utilise 8 et non pas 7 pour mieux gérer les mois à 31 jours)
      DEBUTSEMAINE=true
fi



#  Vérifie que le disque est monté.
#  En fait, vérifie que *quelque chose* est monté ici !
#  Nous pouvons utiliser quelque chose d'unique sur le périphérique
#+ plutôt que de simplement deviner l'ID SCSI en utilisant la bonne règle udev
#+ dans /etc/udev/rules.d/10-rules.local
#+ et en plaçant une entrée adéquate dans /etc/fstab.
```

```
#  Par exemple, cette règle udev :
# BUS="scsi", KERNEL="sd*", SYSFS{vendor}="WDC WD16",
# SYSFS{model}="00JB-00GVA0     ", NAME="%k", SYMLINK="lacie_1394d%n"

if mount | grep $POINT_MONTAGE >/dev/null; then
  echo "Le point de montage $POINT_MONTAGE est déjà utilisé. OK"
else
  echo -n "Tentative de montage de $POINT_MONTAGE..."
           # S'il n'est pas monté, essaie de le monter.
  sudo mount $POINT_MONTAGE 2>/dev/null

  if mount | grep $POINT_MONTAGE >/dev/null; then
    DEMONTE_APRES=TRUE
    echo "OK"
    #  Note : s'assure qu'il sera aussi démonté
    #+ si nous quittons prématurément avec erreur.
  else
    echo "ÉCHEC"
    echo -e "Rien n'est monté sur $POINT_MONTAGE. ÉCHEC DE LA SAUVEGARDE!"
    exit $E_ECHEC_MONTAGE
  fi
fi


# Vérifie que le répertoire source existe et est lisible.
if [ ! -r  $REP_SOURCE ] ; then
  echo "$REP_SOURCE n'existe pas ou ne peut être lu. ÉCHEC DE LA SAUVEGARDE."
  exit $E_PASREPSOURCE
fi


# Vérifie que la structure du répertoire de sauvegarde est bonne.
# Sinon, il la crée.
# Crée les sous-répertoires.
# Notez que backup.0 sera créé si nécessaire par rsync.

for ((i=1;i<=15;i++)); do
  if [ ! -d $REP_DEST_SAUVE/backup.$i ]; then
    if /bin/mkdir -p $REP_DEST_SAUVE/backup.$i ; then
    #  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  Pas de tests entre crochets. Pourquoi
?
      echo "Attention : le répertoire $REP_DEST_SAUVE/backup.$i n'existe pas"
      echo "ou n'a pas été initialisé. (Re-)creation du répertoire."
    else
      echo "ERREUR : le répertoire $REP_DEST_SAUVE/backup.$i"
      echo "n'existe pas et n'a pas pu être créé."
    if  [ "$DEMONTE_APRES" == "TRUE" ]; then
        # Avant de quitter, démonte le point de montage si nécessaire.
        cd
        sudo umount $POINT_MONTAGE &&
        echo "Démontage de $POINT_MONTAGE. Abandon."
    fi
      exit $E_NONMONTE
  fi
fi
done


#  Configure les droits à 700 pour de la sécurité
#+ sur un système multi-utilisateur.
if ! /bin/chmod 700 $REP_DEST_SAUVE ; then
  echo "ERREUR : n'a pas pu configurer les droits du répertoire $REP_DEST_SAUVE à 700."

  if  [ "$DEMONTE_APRES" == "TRUE" ]; then
  # Avant de quitter, démonte le point de montage si nécessaire.
     cd ; sudo umount $POINT_MONTAGE && echo "Démontage de $POINT_MONTAGE. Abandon."
  fi

  exit $E_NONMONTE
```

```
fi

# Création du lien symbolique : current -> backup.1 si nécessaire.
# Un échec ici n'est pas critique.
cd $REP_DEST_SAUVE
if [ ! -h current ] ; then
  if ! /bin/ln -s backup.1 current ; then
    echo "Attention : n'a pas pu créer le lien symbolique current -> backup.1"
  fi
fi


# Maintenant, exécute le rsync.
echo "Sauvegarde en cours avec rsync..."
echo "Répertoire source : $REP_SOURCE"
echo -e "Répertoire destination : $REP_DEST_SAUVE\n"


/usr/bin/rsync $ESSAI_A_BLANC $VERBEUX -a -S --delete --modify-window=60 \
--link-dest=../backup.1 $REP_SOURCE $REP_DEST_SAUVE/backup.0/

#  Avertit seulement, plutôt que de quitter, si rsync a échoué,
#+ car cela pourrait n'être qu'un problème mineur.
#  Par exemple, si un fichier n'est pas lisible, rsync échouera.
#  Ceci ne doit pas empêcher la rotation.
#  Ne pas utiliser, par exemple, `date +%a` car ces répertoires
#+ sont plein de liens et ne consomment pas *tant* d'espace.

if [ $? != 0 ]; then
  SAUVEGARDE_AUCASOU=backup.`date +%F_%T`.justincase
  echo "ATTENTION : le processus rsync n'a pas complètement réussi."
  echo "Quelque chose s'est mal passé. Sauvegarde d'une copie supplémentaire dans :
$SAUVEGARDE_AUCASOU"
  echo "ATTENTION : si cela arrive fréquemment, BEAUCOUP d'espace sera utilisé,"
  echo "même si ce ne sont que des liens !"
fi

# Ajoute un fichier readme dans le répertoire principal de la sauvegarde.
# En sauvegarde un autre dans le sous-répertoire recent.
echo "La sauvegarde de $REP_SOURCE sur `hostname` a été exécuté le \
`date`" > $REP_DEST_SAUVE/README.txt
echo "Cette sauvegarde de $REP_SOURCE sur `hostname` a été créé le \
`date`" > $REP_DEST_SAUVE/backup.0/README.txt

# Si nous n'avons pas fait un test, exécute une rotation des sauvegardes.
[ -z "$ESSAI_A_BLANC" ] &&

  #  Vérifie l'espace occupé du disque de sauvegarde.
  #  Avertissement si 90%.
  #  Si 98% voire plus, nous échouerons probablement, donc abandon.
  #  (Note : df peut afficher plus d'une ligne.)
  #  Nous le testons ici plutôt qu'avant pour donner une chance à rsync.
  DISK_FULL_PERCENT=`/bin/df $REP_DEST_SAUVE |
  tr "\n" ' ' | awk '{print $12}' | grep -oE [0-9]+ `
  echo "Vérification de l'espace disque sur la partition de sauvegarde \
  remplie à $POINT_MONTAGE $DISK_FULL_PERCENT%."
  if [ $DISK_FULL_PERCENT -gt 90 ]; then
    echo "Attention : le disque est rempli à plus de 90%."
  fi
  if [ $DISK_FULL_PERCENT -gt 98 ]; then
    echo "Erreur : le disque est rempli complètement ! Abandon."
      if  [ "$DEMONTE_APRES" == "TRUE" ]; then
        # Avant de quitter, démonte le point de montage si nécessaire.
        cd; sudo umount $POINT_MONTAGE &&
        echo "Démontage de $POINT_MONTAGE. Abandon."
      fi
    exit $E_NONMONTE
  fi
```

```
 # Crée une sauvegarde supplémentaire.
 # Si cette copie échoue, abandonne.
 if [ -n "$SAUVEGARDE_AUCASOU" ]; then
   if ! /bin/cp -al $REP_DEST_SAUVE/backup.0 $REP_DEST_SAUVE/$SAUVEGARDE_AUCASOU
   then
     echo "ERREUR : échec lors de la création de la copie de sauvegarde \
     $REP_DEST_SAUVE/$SAUVEGARDE_AUCASOU"
     if  [ "$DEMONTE_APRES" == "TRUE" ]; then
       # Avant de quitter, démonte le point de montage si nécessaire.
       cd ;sudo umount $POINT_MONTAGE &&
       echo "Démontage de $POINT_MONTAGE. Abandon."
     fi
     exit $E_NONMONTE
   fi
 fi


 # Au début du mois, exécute une rotation des huit plus anciens.
 if [ "$DEBUTMOIS" == "true" ]; then
   echo -e "\nDébut du mois. \
   Suppression de l'ancienne sauvegarde : $REP_DEST_SAUVE/backup.15"  &&
   /bin/rm -rf  $REP_DEST_SAUVE/backup.15  &&
   echo "Rotation mensuelle, sauvegardes hebdomadaires : \
   $REP_DEST_SAUVE/backup.[8-14] -> $REP_DEST_SAUVE/backup.[9-15]"  &&
     /bin/mv $REP_DEST_SAUVE/backup.14 $REP_DEST_SAUVE/backup.15  &&
     /bin/mv $REP_DEST_SAUVE/backup.13 $REP_DEST_SAUVE/backup.14  &&
     /bin/mv $REP_DEST_SAUVE/backup.12 $REP_DEST_SAUVE/backup.13  &&
     /bin/mv $REP_DEST_SAUVE/backup.11 $REP_DEST_SAUVE/backup.12  &&
     /bin/mv $REP_DEST_SAUVE/backup.10 $REP_DEST_SAUVE/backup.11  &&
     /bin/mv $REP_DEST_SAUVE/backup.9 $REP_DEST_SAUVE/backup.10  &&
     /bin/mv $REP_DEST_SAUVE/backup.8 $REP_DEST_SAUVE/backup.9

 # Au début de la semaine, exécute une rotation des quatre seconds plus anciens.
 elif [ "$DEBUTSEMAINE" == "true" ]; then
   echo -e "\nDébut de semaine. \
   Suppression de l'ancienne sauvegarde hebdomadaire : $REP_DEST_SAUVE/backup.12"  &&
   /bin/rm -rf  $REP_DEST_SAUVE/backup.12  &&

   echo "Rotation des sauvegardes hebdomadaires : \
   $REP_DEST_SAUVE/backup.[8-11] -> $REP_DEST_SAUVE/backup.[9-12]"  &&
     /bin/mv $REP_DEST_SAUVE/backup.11 $REP_DEST_SAUVE/backup.12  &&
     /bin/mv $REP_DEST_SAUVE/backup.10 $REP_DEST_SAUVE/backup.11  &&
     /bin/mv $REP_DEST_SAUVE/backup.9 $REP_DEST_SAUVE/backup.10  &&
     /bin/mv $REP_DEST_SAUVE/backup.8 $REP_DEST_SAUVE/backup.9

 else
   echo -e "\nSuppression de l'ancienne sauvegarde quotidienne :
$REP_DEST_SAUVE/backup.8"  &&
     /bin/rm -rf  $REP_DEST_SAUVE/backup.8

 fi  &&

 # Chaque jour, rotation de huit plus anciens.
 echo "Rotation des sauvegardes quotidiennes : \
 $REP_DEST_SAUVE/backup.[1-7] -> $REP_DEST_SAUVE/backup.[2-8]"  &&
     /bin/mv $REP_DEST_SAUVE/backup.7 $REP_DEST_SAUVE/backup.8  &&
     /bin/mv $REP_DEST_SAUVE/backup.6 $REP_DEST_SAUVE/backup.7  &&
     /bin/mv $REP_DEST_SAUVE/backup.5 $REP_DEST_SAUVE/backup.6  &&
     /bin/mv $REP_DEST_SAUVE/backup.4 $REP_DEST_SAUVE/backup.5  &&
     /bin/mv $REP_DEST_SAUVE/backup.3 $REP_DEST_SAUVE/backup.4  &&
     /bin/mv $REP_DEST_SAUVE/backup.2 $REP_DEST_SAUVE/backup.3  &&
     /bin/mv $REP_DEST_SAUVE/backup.1 $REP_DEST_SAUVE/backup.2  &&
     /bin/mv $REP_DEST_SAUVE/backup.0 $REP_DEST_SAUVE/backup.1  &&

 SUCCES=true


if  [ "$DEMONTE_APRES" == "TRUE" ]; then
```

```
  # Démonte le point de montage s'il n'était pas monté au début.
  cd ; sudo umount $POINT_MONTAGE && echo "$POINT_MONTAGE de nouveau démonté."
fi


if [ "$SUCCES" == "true" ]; then
  echo 'SUCCÈS !'
  exit 0
fi

# Nous devrions avoir déjà quitté si la sauvegarde a fonctionné.
echo 'ÉCHEC DE LA SAUVEGARDE ! Est-ce un test ? Le disque est-il plein ?) '
exit $E_SAUVE
```

Exemple A.35. Une commande *cd* étendue

```
###############################################################################
#
#       cdll
#       par Phil Braham
#
#       ############################################################
#       La dernière version de ce script est disponible à partir de
#       http://freshmeat.net/projects/cd/
#       ############################################################
#
#       .cd_new
#
#       Une amélioration de la commande Unix cd
#
#       Il y a une pile illimitée d'entrées et d'entrées spéciales. Les
#       entrées de la pile conservent les cd_maxhistory derniers répertoires
#       qui ont été utilisés. Les entrées spéciales peuvent être affectées aux
#       répertoires fréquemment utilisés.
#
#       Les entrées spéciales pourraient être préaffectées en configurant les
#       variables d'environnement CDSn ou en utilisant la commande -u ou -U.
#
#       Ce qui suit est une suggestion pour le fichier .profile :
#
#               . cdll              #  Configure la commande cd
#       alias cd='cd_new'           #  Remplace la commande cd
#               cd -U               #  Charge les entrées pré-affectées pour
#                                   #+ la pile et les entrées spéciales
#               cd -D               #  Configure le mode pas par défaut
#               alias @="cd_new @"  #  Autorise l'utilisation de @ pour récupérer
#                                   #+ l'historique
#
#       Pour une aide, saisissez :
#
#               cd -h ou
#               cd -H
#
#
###############################################################################
#
#       Version 1.2.1
#
#       Écrit par Phil Braham - Realtime Software Pty Ltd
#       (realtime@mpx.com.au)
#       Merci d'envoyer vos suggestions ou améliorations à l'auteur
#       (phil@braham.net)
#
###############################################################################

cd_hm ()
{
```

```
        ${PRINTF} "%s" "cd [dir] [0-9] [@[s|h] [-g [&lt;dir&gt;]] [-d] [-D]
[-r&lt;n&gt;] [dir|0-9] [-R&lt;n&gt;] [&lt;dir&gt;|0-9]
   [-s&lt;n&gt;] [-S&lt;n&gt;] [-u] [-U] [-f] [-F] [-h] [-H] [-v]
    &lt;dir&gt; Se place sous le répertoire
    0-n         Se place sous le répertoire précedent (0 est le précédent, 1 est
l'avant-dernier, etc)
                n va jusqu'au bout de l'historique (par défaut, 50)
    @           Liste les entrées de l'historique et les entrées spéciales
    @h          Liste les entrées de l'historique
    @s          Liste les entrées spéciales
    -g [&lt;dir&gt;]  Se place sous le nom littéral (sans prendre en compte les noms
spéciaux)
                Ceci permet l'accès aux répertoires nommés '0','1','-h' etc
    -d          Modifie l'action par défaut - verbeux. (Voir note)
    -D          Modifie l'action par défaut - silencieux. (Voir note)
    -s&lt;n&gt;       Se place sous l'entrée spéciale &lt;n&gt;*
    -S&lt;n&gt;       Se place sous l'entrée spéciale &lt;n&gt; et la remplace avec le
répertoire en cours*
    -r&lt;n&gt; [&lt;dir&gt;] Se place sous le répertoire &lt;dir&gt; and then put it
on special entry &lt;n&gt;*
    -R&lt;n&gt; [&lt;dir&gt;] Se place sous le répertoire &lt;dir&gt; et place le
répertoire en cours dans une entrée spéciale &lt;n&gt;*
    -a&lt;n&gt;       Autre répertoire suggéré. Voir la note ci-dessous.
    -f [&lt;file&gt;] Fichier des entrées &lt;file&gt;.
    -u [&lt;file&gt;] Met à jour les entrées à partir de &lt;file&gt;.
                Si aucun nom de fichier n'est fourni, utilise le fichier par défaut
(${CDPath}${2:-"$CDFile"})
                -F et -U sont les versions silencieuses
    -v          Affiche le numéro de version
    -h          Aide
    -H          Aide détaillée

    *Les entrées spéciales (0 - 9) sont conservées jusqu'à la déconnexion, remplacées
par une autre entrée
    ou mises à jour avec la commande -u

    Autres répertoires suggérés :
    Si un répertoire est introuvable, alors CD suggèrera des possibilités. Ce sont les
répertoires
    commençant avec les mêmes lettres et si des résultats sont disponibles, ils sont
affichés avec
    le préfixe -a&lt;n&gt; où &lt;n&gt; est un numéro.
    Il est possible de se placer dans le répertoire en saisissant cd -a&lt;n&gt; sur la
ligne de commande.

    Le répertoire pour -r&lt;n&gt; ou -R&lt;n&gt; pourrait être un numéro. Par exemple
:
        $ cd -r3 4  Se place dans le répertoire de l'entrée 4 de l'historique et la
place
                    sur l'entrée spéciale 3
        $ cd -R3 4  Place le répertoire en cours sur l'entrée spéciale 3 et se déplace
dans l'entrée 4
                    de l'historique
        $ cd -s3    Se déplace dans l'entrée spéciale 3

    Notez que les commandes R,r,S et s pourraient être utilisées sans numéro et faire
ainsi référence à 0:
        $ cd -s     Se déplace dans l'entrée spéciale 0
        $ cd -S     Se déplace dans l'entrée spéciale 0 et fait de l'entrée spéciale 0
le répertoire courant
        $ cd -r 1   Se déplace dans l'entrée spéciale 1 et la place sur l'entrée
spéciale 0
        $ cd -r     Se déplace dans l'entrée spéciale 0 et la place sur l'entrée
spéciale 0
    "
        if ${TEST} "$CD_MODE" = "PREV"
        then
                ${PRINTF} "$cd_mnset"
        else
```

```
                ${PRINTF} "$cd_mset"
        fi
}

cd_Hm ()
{
        cd_hm
        ${PRINTF} "%s" "
        Les répertoires précédents (0-$cd_maxhistory) sont stockés dans les variables
        d'environnement CD[0] - CD[$cd_maxhistory]
        De façon similaire, les répertoires spéciaux S0 - $cd_maxspecial sont dans la
        variable d'environnement CDS[0] - CDS[$cd_maxspecial]
        et pourraient être accédés à partir de la ligne de commande

        Le chemin par défaut pour les commandes -f et -u est $CDPath
        Le fichier par défaut pour les commandes est -f et -u est $CDFile

        Configurez les variables d'environnement suivantes :
            CDL_PROMPTLEN  - Configuré à la longueur de l'invite que vous demandez.
                La chaîne de l'invite est configurée suivant les caractères de droite
du
                répertoire en cours.
                Si non configuré, l'invite n'est pas modifiée.
            CDL_PROMPT_PRE - Configuré avec la chaîne pour préfixer l'invite.
                La valeur par défaut est:
                    standard:  \"\\[\\e[01;34m\\]\"  (couleur bleu).
                    root:      \"\\[\\e[01;31m\\]\"  (couleur rouge).
            CDL_PROMPT_POST    - Configuré avec la chaîne pour suffixer l'invite.
                La valeur par défaut est:
                    standard:  \"\\[\\e[00m\\]$\"   (réinitialise la couleur et affiche
$).
                    root:      \"\\[\\e[00m\\]#\"   (réinitialise la couleur et affiche
#).
            CDPath - Configure le chemin par défaut des options -f & -u.
                     Par défaut, le répertoire personnel de l'utilisateur
            CDFile - Configure le fichier par défaut pour les options -f & -u.
                     Par défaut, cdfile

"
    cd_version

}

cd_version ()
{
    printf "Version: ${VERSION_MAJOR}.${VERSION_MINOR} Date: ${VERSION_DATE}\n"
}

#
# Tronque à droite.
#
# params:
#   p1 - chaîne
#   p2 - longueur à tronquer
#
# renvoit la chaîne dans tcd
#
cd_right_trunc ()
{
    local tlen=${2}
    local plen=${#1}
    local str="${1}"
    local diff
    local filler="<--"
    if ${TEST} ${plen} -le ${tlen}
    then
        tcd="${str}"
    else
        let diff=${plen}-${tlen}
```

```
        elen=3
        if ${TEST} ${diff} -le 2
        then
            let elen=${diff}
        fi
        tlen=-${tlen}
        let tlen=${tlen}+${elen}
        tcd=${filler:0:elen}${str:tlen}
    fi
}

#
# Trois versions de l'historique do :
#    cd_dohistory  - empile l'historique et les spéciaux côte à côte
#    cd_dohistoryH - Affiche seulement l'historique
#    cd_dohistoryS - Affiche seulement les spéciaux
#
cd_dohistory ()
{
    cd_getrc
        ${PRINTF} "Historique :\n"
    local -i count=${cd_histcount}
    while ${TEST} ${count} -ge 0
    do
        cd_right_trunc "${CD[count]}" ${cd_lchar}
            ${PRINTF} "%2d %-${cd_lchar}.${cd_lchar}s " ${count} "${tcd}"

        cd_right_trunc "${CDS[count]}" ${cd_rchar}
            ${PRINTF} "S%d %-${cd_rchar}.${cd_rchar}s\n" ${count} "${tcd}"
        count=${count}-1
    done
}

cd_dohistoryH ()
{
    cd_getrc
        ${PRINTF} "Historique :\n"
        local -i count=${cd_maxhistory}
        while ${TEST} ${count} -ge 0
        do
                ${PRINTF} "${count} %-${cd_flchar}.${cd_flchar}s\n" ${CD[$count]}
                count=${count}-1
        done
}

cd_dohistoryS ()
{
    cd_getrc
        ${PRINTF} "Spéciaux :\n"
        local -i count=${cd_maxspecial}
        while ${TEST} ${count} -ge 0
        do
                ${PRINTF} "S${count} %-${cd_flchar}.${cd_flchar}s\n" ${CDS[$count]}
                count=${count}-1
        done
}

cd_getrc ()
{
    cd_flchar=$(stty -a | awk -F \; '/rows/ { print $2 $3 }' | awk -F \  '{ print $4
}')
    if ${TEST} ${cd_flchar} -ne 0
    then
        cd_lchar=${cd_flchar}/2-5
        cd_rchar=${cd_flchar}/2-5
            cd_flchar=${cd_flchar}-5
    else
            cd_flchar=${FLCHAR:=75}  # cd_flchar is used for for the @s & @h history
            cd_lchar=${LCHAR:=35}
```

```
            cd_rchar=${RCHAR:=35}
    fi
}

cd_doselection ()
{
        local -i nm=0
        cd_doflag="TRUE"
        if ${TEST} "${CD_MODE}" = "PREV"
        then
                if ${TEST} -z "$cd_npwd"
                then
                        cd_npwd=0
                fi
        fi
        tm=$(echo "${cd_npwd}" | cut -b 1)
    if ${TEST} "${tm}" = "-"
    then
        pm=$(echo "${cd_npwd}" | cut -b 2)
        nm=$(echo "${cd_npwd}" | cut -d $pm -f2)
        case "${pm}" in
                a) cd_npwd=${cd_sugg[$nm]} ;;
                s) cd_npwd="${CDS[$nm]}" ;;
                S) cd_npwd="${CDS[$nm]}" ; CDS[$nm]=`pwd` ;;
                r) cd_npwd="$2" ; cd_specDir=$nm ; cd_doselection "$1" "$2";;
                R) cd_npwd="$2" ; CDS[$nm]=`pwd` ; cd_doselection "$1" "$2";;
        esac
    fi

        if ${TEST} "${cd_npwd}" != "." -a "${cd_npwd}" != ".." -a "${cd_npwd}" -le
${cd_maxhistory} >>/dev/null 2>&1
        then
                cd_npwd=${CD[$cd_npwd]}
        else
                case "$cd_npwd" in
                         @)  cd_dohistory ; cd_doflag="FALSE" ;;
                        @h) cd_dohistoryH ; cd_doflag="FALSE" ;;
                        @s) cd_dohistoryS ; cd_doflag="FALSE" ;;
                        -h) cd_hm ; cd_doflag="FALSE" ;;
                        -H) cd_Hm ; cd_doflag="FALSE" ;;
                        -f) cd_fsave "SHOW" $2 ; cd_doflag="FALSE" ;;
                        -u) cd_upload "SHOW" $2 ; cd_doflag="FALSE" ;;
                        -F) cd_fsave "NOSHOW" $2 ; cd_doflag="FALSE" ;;
                        -U) cd_upload "NOSHOW" $2 ; cd_doflag="FALSE" ;;
                        -g) cd_npwd="$2" ;;
                        -d) cd_chdefm 1; cd_doflag="FALSE" ;;
                        -D) cd_chdefm 0; cd_doflag="FALSE" ;;
                        -r) cd_npwd="$2" ; cd_specDir=0 ; cd_doselection "$1" "$2";;
                        -R) cd_npwd="$2" ; CDS[0]=`pwd` ; cd_doselection "$1" "$2";;
                        -s) cd_npwd="${CDS[0]}" ;;
                        -S) cd_npwd="${CDS[0]}"  ; CDS[0]=`pwd` ;;
                        -v) cd_version ; cd_doflag="FALSE";;
                esac
        fi
}

cd_chdefm ()
{
        if ${TEST} "${CD_MODE}" = "PREV"
        then
                CD_MODE=""
                if ${TEST} $1 -eq 1
                then
                        ${PRINTF} "${cd_mset}"
                fi
        else
                CD_MODE="PREV"
                if ${TEST} $1 -eq 1
                then
```

```
                        ${PRINTF} "${cd_mnset}"
                fi
        fi
}

cd_fsave ()
{
        local sfile=${CDPath}${2:-"$CDFile"}
        if ${TEST} "$1" = "SHOW"
        then
                ${PRINTF} "Saved to %s\n" $sfile
        fi
        ${RM} -f ${sfile}
        local -i count=0
        while ${TEST} ${count} -le ${cd_maxhistory}
        do
                echo "CD[$count]=\"${CD[$count]}\"" >> ${sfile}
                count=${count}+1
        done
        count=0
        while ${TEST} ${count} -le ${cd_maxspecial}
        do
                echo "CDS[$count]=\"${CDS[$count]}\"" >> ${sfile}
                count=${count}+1
        done
}

cd_upload ()
{
        local sfile=${CDPath}${2:-"$CDFile"}
        if ${TEST} "${1}" = "SHOW"
        then
                ${PRINTF} "Chargement de %s\n" ${sfile}
        fi
        . ${sfile}
}

cd_new ()
{
    local -i count
    local -i choose=0

        cd_npwd="${1}"
        cd_specDir=-1
        cd_doselection "${1}" "${2}"

        if ${TEST} ${cd_doflag} = "TRUE"
        then
                if ${TEST} "${CD[0]}" != "`pwd`"
                then
                        count=$cd_maxhistory
                        while ${TEST} $count -gt 0
                        do
                                CD[$count]=${CD[$count-1]}
                                count=${count}-1
                        done
                        CD[0]=`pwd`
                fi
                command cd "${cd_npwd}" 2>/dev/null
        if ${TEST} $? -eq 1
        then
            ${PRINTF} "Répertoire inconnu : %s\n" "${cd_npwd}"
            local -i ftflag=0
            for i in "${cd_npwd}"*
            do
                if ${TEST} -d "${i}"
                then
                    if ${TEST} ${ftflag} -eq 0
                    then
```

```
                        ${PRINTF} "Suggest:\n"
                        ftflag=1
                fi
                    ${PRINTF} "\t-a${choose} %s\n" "$i"
                                        cd_sugg[$choose]="${i}"
                    choose=${choose}+1
        fi
            done
        fi
        fi

        if ${TEST} ${cd_specDir} -ne -1
        then
                CDS[${cd_specDir}]=`pwd`
        fi

        if ${TEST} ! -z "${CDL_PROMPTLEN}"
        then
        cd_right_trunc "${PWD}" ${CDL_PROMPTLEN}
            cd_rp=${CDL_PROMPT_PRE}${tcd}${CDL_PROMPT_POST}
                export PS1="$(echo -ne ${cd_rp})"
        fi
}
###############################################################################
#                                                                             #
#                           Initialisation ici                                #
#                                                                             #
###############################################################################
#
VERSION_MAJOR="1"
VERSION_MINOR="2.1"
VERSION_DATE="24 MAI 2003"
#
alias cd=cd_new
#
# Configuration des commandes
RM=/bin/rm
TEST=test
PRINTF=printf              # Utilise le printf interne

###############################################################################
#                                                                             #
# Modifiez ceci pour modifier les chaînes préfixe et suffixe de l'invite.     #
# Elles ne prennent effet que si CDL_PROMPTLEN est configuré.                 #
#                                                                             #
###############################################################################
if ${TEST} ${EUID} -eq 0
then
#   CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="$HOSTNAME@"}
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;31m\\]"}    # Root est en rouge
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]#"}
else
    CDL_PROMPT_PRE=${CDL_PROMPT_PRE:="\\[\\e[01;34m\\]"}    # Les utilisateurs sont en
bleu
    CDL_PROMPT_POST=${CDL_PROMPT_POST:="\\[\\e[00m\\]$"}
fi
###############################################################################
#
# cd_maxhistory définit le nombre max d'entrées dans l'historique.
typeset -i cd_maxhistory=50

###############################################################################
#
# cd_maxspecial définit le nombre d'entrées spéciales.
typeset -i cd_maxspecial=9
#
#
###############################################################################
#
```

```
# cd_histcount définit le nombre d'entrées affichées dans la commande historique.
typeset -i cd_histcount=9
#
###############################################################################
export CDPath=${HOME}/
#  Modifiez-les pour utiliser un chemin et un nom de fichier                  #
#+ différent de la valeur par défaut                                         #
export CDFile=${CDFILE:=cdfile}                  # pour les commandes -u et -f #
#
###############################################################################
                                                                              #
typeset -i cd_lchar cd_rchar cd_flchar
                                #  Ceci est le nombre de caractères pour que
#
cd_flchar=${FLCHAR:=75}         #+ cd_flchar puisse être autorisé pour l'historique de
@s & @h #

typeset -ax CD CDS
#
cd_mset="\n\tLe mode par défaut est maintenant configuré - saisir cd sans paramètre
correspond à l'action par défaut\n\tUtilisez cd -d ou -D pour que cd aille au
répertoire précédent sans paramètres\n"
cd_mnset="\n\tL'autre mode est maintenant configuré - saisir cd sans paramètres est
identique à saisir cd 0\n\tUtilisez cd -d ou -D pour modifier l'action par défaut de
cd\n"

# ============================================================================ #



: &lt;&lt;DOCUMENTATION

Écrit par Phil Braham. Realtime Software Pty Ltd.
Sortie sous licence GNU. Libre à utiliser. Merci de passer toutes modifications
ou commentaires à l'auteur Phil Braham:

realtime@mpx.com.au
================================================================================

cdll est un remplacement pour cd et incorpore des fonctionnalités similaires
aux commandes pushd et popd de bash mais est indépendent.

Cette version de cdll a été testée sur Linux en utilisant Bash. Il fonctionnera
sur la plupart des versions Linux mais ne fonctionnera probablement pas sur les
autres shells sans modification.

Introduction
============

cdll permet un déplacement facile entre les répertoires. En allant dans un autre
répertoire, celui en cours est placé automatiquement sur une pile. Par défaut,
50 entrées sont conservées mais c'est configurable. Les répertoires spéciaux
peuvent être gardés pour un accès facile - par défaut jusqu'à 10, mais ceci est
configurable. Les entrées les plus récentes de la pile et les entrées spéciales
peuvent être facilement visualisées.

La pile de répertoires et les entrées spéciales peuvent être sauvegardées dans
un fichier ou chargées à partir d'un fichier. Ceci leur permet d'être initialisé
à la connexion, sauvegardé avant la fin de la session ou déplacé en passant de
projet à projet.

En plus, cdll fournit une invite flexible permettant, par exemple, un nom de
répertoire en couleur, tronqué à partir de la gauche s'ilest trop long.


Configurer cdll
===============

Copiez cdll soit dans votre répertoire personnel soit dans un répertoire central
```

```
comme /usr/bin (ceci requiert un accès root).

Copiez le fichier cdfile dans votre répertoie personnel. Il requèrera un accès
en lecture et écriture. Ceci est un fichier par défaut contenant une pile de
répertoires et des entrées spéciales.

Pour remplacer la commande cd, vous devez ajouter les commandes à votre script
de connexion. Le script de connexion fait partie de :

    /etc/profile
    ~/.bash_profile
    ~/.bash_login
    ~/.profile
    ~/.bashrc
    /etc/bash.bashrc.local

Pour configurer votre connexion, ~/.bashrc est recommandé, pour la configuration
globale (et de root), ajoutez les commandes à /etc/bash.bashrc.local

Pour configurer la connexion, ajoutez la commande :
    . &lt;dir&gt;/cdll
Par exemple, si cdll est dans votre répertoire personnel :
    . ~/cdll
Si dans /usr/bin, alors :
    . /usr/bin/cdll

Si vous voulez utiliser ceci à la place de la commande cd interne, alors ajoutez :
    alias cd='cd_new'
Nous devrions aussi recommander les commandes suivantes :
    alias @='cd_new @'
    cd -U
    cd -D

Si vous utilisez la capacité de l'invite de cdll, alors ajoutez ce qui suit :
    CDL_PROMPTLEN=nn
Quand nn est un nombre décrit ci-dessous. Initialement, 99 serait un nombre
convenable.

Du coup, le script ressemble à ceci :

    ##############################################################################
    # CD Setup
    ##############################################################################
    CDL_PROMPTLEN=21        # Autorise une longueur d'invite d'un maximum
                            # de 21 caractères
    . /usr/bin/cdll         # Initialise cdll
    alias cd='cd_new'       # Remplace la commande cd interne
    alias @='cd_new @'      # Autorise @ sur l'invite pour affiche l'historique
    cd -U                   # Recharge le répertoire
    cd -D                   # Configure l'action par défaut en non posix
    ##############################################################################

La signification complète de ces commandes deviendra claire plus tard.

Voici quelques astuces. Si un autre programme modifie le répertoire sans appeler
cdll, alors le répertoire ne sera pas placé sur la pile et aussi si la
fonctionnalité de l'invite est utilisée, alors ceci ne sera pas mise à jour.
Deux programmes qui peuvent faire ceci sont pushd et popd. Pour mettre à jour
l'invite et la pile, saisissez simplement :

    cd .

Notez que si l'entrée précédente sur la pile est le répertoire en cours, alors
la pile n'est pas mise à jour.

Usage
=====
cd [dir] [0-9] [@[s|h] [-g &lt;dir&gt;] [-d] [-D] [-r&lt;n&gt;]
   [dir|0-9] [-R&lt;n&gt;] [&lt;dir&gt;|0-9] [-s&lt;n&gt;] [-S&lt;n&gt;]
```

```
   [-u] [-U] [-f] [-F] [-h] [-H] [-v]

    &lt;dir&gt; Se place sous le répertoire
    0-n         Se place sous le répertoire précedent (0 est le précédent, 1 est
l'avant-dernier, etc)
                n va jusqu'au bout de l'historique (par défaut, 50)
    @           Liste les entrées de l'historique et les entrées spéciales
    @h          Liste les entrées de l'historique
    @s          Liste les entrées spéciales
    -g [&lt;dir&gt;]  Se place sous le nom littéral (sans prendre en compte les noms
spéciaux)
                Ceci permet l'accès aux répertoires nommés '0','1','-h' etc
    -d          Modifie l'action par défaut - verbeux. (Voir note)
    -D          Modifie l'action par défaut - silencieux. (Voir note)
    -s&lt;n&gt;       Se place sous l'entrée spéciale &lt;n&gt;*
    -S&lt;n&gt;       Se place sous l'entrée spéciale &lt;n&gt; et la remplace avec le
répertoire en cours*
    -r&lt;n&gt; [&lt;dir&gt;] Se place sous le répertoire &lt;dir&gt; and then put it
on special entry &lt;n&gt;*
    -R&lt;n&gt; [&lt;dir&gt;] Se place sous le répertoire &lt;dir&gt; et place le
répertoire en cours dans une entrée spéciale &lt;n&gt;*
    -a&lt;n&gt;       Autre répertoire suggéré. Voir la note ci-dessous.
    -f [&lt;file&gt;] Fichier des entrées &lt;file&gt;.
    -u [&lt;file&gt;] Met à jour les entrées à partir de &lt;file&gt;.
                Si aucun nom de fichier n'est fourni, utilise le fichier par défaut
(${CDPath}${2:-"$CDFile"})
                -F et -U sont les versions silencieuses
    -v          Affiche le numéro de version
    -h          Aide
    -H          Aide détaillée



Exemples
========

Ces exemples supposent que le mode autre que celui par défaut est configuré (qui
est, cd sans paramètres ira sur le répertoire le plus récent de la pile), que
les alias ont été configurés pour cd et @ comme décrits ci-dessus et que la
fonctionnalité de l'invite de cd est active et la longueur de l'invite est de
21 caractères.

    /home/phil$ @                                                   # Liste les entrées
avec le @
    History:                                                        # Affiche la
commande @
    .....                                                           # Laissé ces
entrées pour être bref
    1 /home/phil/ummdev               S1 /home/phil/perl            # Les deux entrées
les plus récentes de l'historique
    0 /home/phil/perl/eg              S0 /home/phil/umm/ummdev      # et deux entrées
spéciales sont affichées

    /home/phil$ cd /home/phil/utils/Cdll                            # Maintenant,
modifie les répertoires
    /home/phil/utils/Cdll$ @                                        # L'invite reflète
le répertoire.
    History:                                                        # Nouvel historique
    .....
    1 /home/phil/perl/eg              S1 /home/phil/perl            # L'entrée 0 de
l'historique a été déplacé dans 1
    0 /home/phil                      S0 /home/phil/umm/ummdev      # et la plus
récente a été entrée

Pour aller dans une entrée de l'historique :

    /home/phil/utils/Cdll$ cd 1                                     # Va dans l'entrée
1 de l'historique.
    /home/phil/perl/eg$                                             # Le répertoire en
```

```
cours est maintenant celui du 1

Pour aller dans une entrée spéciale :

    /home/phil/perl/eg$ cd -s1                                          # Va dans l'entrée
spéciale 1
    /home/phil/umm/ummdev$                                              # Le répertoire en
cours est S1

Pour aller dans un répertoire nommé, par exemple, 1 :

    /home/phil$ cd -g 1                                                 # -g ignore la
signification spéciale de 1
    /home/phil/1$

Pour placer le répertoire en cours sur la liste spéciale en tant que S1 :
    cd -r1 .        #  OU
    cd -R1 .        #  Elles ont le même effet si le répertoire est
                    #+ . (le répertoire en cours)

Pour aller dans un répertoire et l'ajouter comme entrée spéciale
    Le répertoire pour -r&lt;n&gt; ou -R&lt;n&gt; pourrait être un nombre. Par exemple
:
        $ cd -r3 4  Va dans l'entrée 4 de l'historique et placez-la dans l'entrée
spéciale 3
        $ cd -R3 4  Placez le répertoire en cours sur l'entrée spéciale 3 et allez dans
l'entrée spéciale 4
        $ cd -s3    Allez dans l'entrée spéciale 3

    Notez que les commands R,r,S et s pourraient être utilisées sans un numéro et faire
référence à 0 :
        $ cd -s     Va dans l'entrée spéciale 0
        $ cd -S     Va dans l'entrée spéciale 0 et fait de l'entrée spéciale 0 le
répertoire en cours
        $ cd -r 1   Va dans l'entrée 1 de l'historique et la place sur l'entrée
spéciale 0
        $ cd -r     Va dans l'entrée 0 de l'historique et la place sur l'entrée
spéciale 0


    Autres répertoires suggérés :

    Si un répertoire est introuvable, alors CD suggèrera toute possibilité.
    Il s'agit des répertoires commençant avec les mêmes lettres et si des
    correspondances sont trouvées, ils sont affichés préfixés avec -a&lt;n&gt;
    où &lt;n&gt; est un numéro. Il est possible d'aller dans un répertoire
    de saisir cd -a&lt;n&gt; sur la ligne de commande.

        Utilisez cd -d ou -D pour modifier l'action par défaut de cd. cd -H
        affichera l'action en cours.

        Les entrées de l'historique (0-n) sont stockées dans les variables
        d'environnement CD[0] - CD[n]
        De façon similaire, les répertoires spéciaux S0 - 9 sont dans la variable
        d'environnement CDS[0] - CDS[9] et pourraient être accédés à partir de
        la ligne de commande, par exemple :

            ls -l ${CDS[3]}
            cat ${CD[8]}/file.txt

        Le chemin par défaut pour les commandes -f et -u est ~
        Le nom du fichier par défaut pour les commandes -f et -u est cdfile


Configuration
=============

    Les variables d'environnement suivantes peuvent être configurées :
```

```
            CDL_PROMPTLEN  - Configuré à la longueur de l'invite que vous demandez.
                La chaîne de l'invite est configurée suivant les caractères de droite
du
                répertoire en cours. Si non configuré, l'invite n'est pas modifiée.
                Notez que ceci est le nombre de caractères raccourcissant le
répertoire,
                pas le nombre de caractères total dans l'invite.

            CDL_PROMPT_PRE - Configure une chaîne pour préfixer l'invite.
                Default is:
                    non-root:  "\\[\\e[01;34m\\]"  (initialise la couleur à bleu).
                    root:      "\\[\\e[01;31m\\]"  (initialise la couleur à rouge).

            CDL_PROMPT_POST    - Configure une chaîne pour suffixer l'invite.
                Default is:
                    non-root:  "\\[\\e[00m\\]$"    (réinitialise la couleur et affiche
$).
                    root:      "\\[\\e[00m\\]#"    (réinitialise la couleur et affiche
#).

        Note:
            CDL_PROMPT_PRE & _POST only t

        CDPath - Configure le chemin par défaut pour les options -f & -u.
                 La valeur par défaut est le répertoire personnel
        CDFile - Configure le nom du fichier pour les options -f & -u.
                 La valeur par défaut est cdfile


    Il existe trois variables définies dans le fichier cdll qui contrôle le nombre
    d'entrées stockées ou affichées. Elles sont dans la sectioon labellées
    'Initialisation ici' jusqu'à la fin du fichier.

        cd_maxhistory       - Le nombre d'entrées stockées dans l'historique.
                              Par défaut, 50.
        cd_maxspecial       - Le nombre d'entrées spéciale autorisées.
                              Par défaut, 9.
        cd_histcount        - Le nombre d'entrées de l'historique et d'entrées
spéciales
                              affichées. Par défaut, 9.

    Notez que cd_maxspecial devrait être >= cd_histcount pour afficher des entrées
    spéciales qui ne peuvent pas être initialisées.


Version: 1.2.1 Date: 24-MAY-2003

DOCUMENTATION
```

Exemple A.36. Un script de configuration d'une carte son

```
#!/bin/bash
# soundcard-on.sh

#  Auteur du script : Mkarcher
#  http://www.thinkwiki.org/wiki  ...
#  /Script pour la configuration du composant CS4239 en mode PnP.
#  L'auteur du guide ABS a réalisé quelques modifications mineures
#+ et ajouté des commentaires.
#  N'a pas pu contacter l'auteur du script pour demander une autorisation de
#+ publication mais... le script était disponible sous licence FDL,
#+ donc son utilisation ici ne devrait pas poser de problèmes légaux ou éthiques.

#  Sound-via-pnp-script pour Thinkpad 600E
#+ et aussi pour d'autres ordinateurs disposant du CS4239/CS4610
#+ qui ne fonctionne pas avec le piltoe PCI
#+ et ne sont pas reconnus par le code PnP de snd-cs4236.
```

```
#  Aussi pour certains Thinkpads 770, comme le 770x.
#  À exécuter en tant que root bien sûr.
#
#  Ce sont des portables vieux et largement obsolètes,
#+ mais ce script particulier est très instructif
#+ car il montre comment configurer et modifier des fichiers périphériques.



#  Recherche du périphérique correspondant à la carte son PnP :

for dev in /sys/bus/pnp/devices/*
do
  grep CSC0100 $dev/id > /dev/null && WSSDEV=$dev
  grep CSC0110 $dev/id > /dev/null && CTLDEV=$dev
done
# Sur 770x :
# WSSDEV = /sys/bus/pnp/devices/00:07
# CTLDEV = /sys/bus/pnp/devices/00:06
# Ce sont des liens symboliques vers /sys/devices/pnp0/ ...


#  Activation des périphériques :
#  Thinkpad démarre avec les périphériques désactivés
#+ sauf si le démarrage rapide est désactivé (dans le BIOS).
echo activate > $WSSDEV/resources
echo activate > $CTLDEV/resources


# Analyse du paramétrage de la ressource.

{ read # Ignore "state = active" (voir ci-dessous).
  read bla port1
  read bla port2
  read bla port3
  read bla irq
  read bla dma1
  read bla dma2
 # Les "bla" sont des labels dans le premier champ : "io", "state", etc.
 # Ils sont ignorés.

 #  Hack : avec PnPBIOS : les ports sont : port1: WSS, port2:
 #+ OPL, port3: sb (unneeded)
 #       avec ACPI-PnP : les ports sont: port1: OPL, port2: sb, port3: WSS
 #  (le BIOS ACPI semble être erroné ici, le code de la carte PnP dans snd-cs4236.c
 #+  utilise l'ordre des ports de PnPBIOS)
 #  Détecter l'ordre des ports en utilisant le port OPL fixe comme référence.
  if [ ${port2%%-*} = 0x388 ]
 #            ^^^^  Supprime tout ce qui suit le trait d'union dans l'adresse du port.
 #                  Donc, si port1 vaut 0x530-0x537
 #+                 il nous reste 0x530 -- l'adresse de début du port.
 then
   # PnPBIOS : ordre habituel
   port=${port1%%-*}
   oplport=${port2%%-*}
 else
   # ACPI : ordre mixe
   port=${port3%%-*}
   oplport=${port1%%-*}
 fi
 } < $WSSDEV/resources
# Pour voir ce qu'il se passe ici :
# --------------------------------
#   cat /sys/devices/pnp0/00:07/resources
#
#   state = active
#   io 0x530-0x537
#   io 0x388-0x38b
#   io 0x220-0x233
```

```
#   irq 5
#   dma 1
#   dma 0
#   ^^^   labels "bla" dans le premier champ (ignoré).


{ read # Ignore la première ligne, comme ci-dessus.
  read bla port1
  cport=${port1%%-*}
  #            ^^^^
  # On veut seulement l'adresse de _début_ du port.
} < $CTLDEV/resources


# Chargement du module :

modprobe --ignore-install snd-cs4236 port=$port cport=$cport\
fm_port=$oplport irq=$irq dma1=$dma1 dma2=$dma2 isapnp=0 index=0
# Voir la page man de modprobe.

exit $?
```

Exemple A.37. Localise les paragraphes de division dans un fichier texte

```
#!/bin/bash
# find-splitpara.sh
#  Recherche les paragraphes de séparation dans un fichier texte
#+ et numérote les lignes.


NB_ARG=1         # Un seul argument attendu.
E_MAUVAISARG=65

fichier="$1"     # Cible le nom du fichier.
no_ligne=1       # Numéro de ligne. Commence à 1.
Drapeau=0        # Blank ligne Drapeau.

if [ $# -ne "$NB_ARG" ]
then
  echo "Usage: `basename $0` NOMFICHIER"
  exit $E_MAUVAISARG
fi

# Parcourt le fichier à la recherche d'un modèle puis affiche la ligne.
lecture_fichier ()
{
while read ligne
do

  if [[ "$ligne" =~ ^[a-z] && $Drapeau -eq 1 ]]
     then  # ligne commence avec un caractère alphabétique, précédée d'une ligne
blanche.
     echo -n "$no_ligne::   "
     echo "$line"
  fi


  if [[ "$ligne" =~ "^$" ]]
     then        #  Si ligne vide,
     Drapeau=1   #+ Drapeau activé.
  else
     Drapeau=0
  fi

  ((no_ligne++))
```

```
done
} < $fichier  # Redirige fichier dans le stdin de la fonction.

lecture_fichier


exit $?


# ---------------------------------------------------------------
Ceci est la ligne une du paragraphe exemple, bla, bla, bla.
Ceci est la ligne deux et la ligne trois doit suivre mais...

une ligne blanche sépare les deux parties du paragraphe.
# ---------------------------------------------------------------

Exécuter ce script sur un fichier contenant le paragraphe ci-dessus
ramène :

4::   une ligne blanche sépare les deux parties du paragraphe.


Il y aura d'autres affichages pour tous les paragraphes séparées contenus
dans le fichier cible.
```

Exemple A.38. Tri d'insertion

```
#!/bin/bash
# insertion-sort.bash: Implémentation du tri d'insertion dans Bash
#                      Grosse utilisation des fonctionnalités tableau de Bash :
#+                     découpage (chaîne), assemblage, etc
# URL: http://www.lugmen.org.ar/~jjo/jjotip/insertion-sort.bash.d
#+          /insertion-sort.bash.sh
#
# Auteur : JuanJo Ciarlante &lt;jjo@irrigacion.gov.ar&gt;
# Légèrement reformaté par l'auteur du guide ABS.
# Licence : GPLv2
# Utilisé dans le guide ABS avec la permission de l'auteur (merci !).
#
# Testez avec    ./insertion-sort.bash -t
# Ou  :          bash insertion-sort.bash -t
# Ce qui suit *ne fonctionne pas* :
#              sh insertion-sort.bash -t
#  Pourquoi pas ? Astuce : quelles fonctionnalités spécifiques de Bash sont
#+ désactivées quand un script est exécuté par 'sh script.sh'?
#
: ${DEBUG:=0}  # Debug, surchargé avec :  DEBUG=1 ./nomscript . . .
# Substitution de paramètres -- configurer DEBUG à 0 si non initialisé auparavant.

# Tableau global : "liste"
typeset -a liste
# Chargement de nombres séparés par des espaces blancs à partir de stdin.
if [ "$1" = "-t" ]; then
DEBUG=1
        read -a liste < <( od -Ad -w24 -t u2 /dev/urandom ) # Liste aléatoire.
#                       ^ ^  substitution de processus
else
        read -a liste
fi
numelem=${#liste[*]}

#  Affiche la liste, marquant l'élément dont l'index est $1
#+ en la surchargeant avec les deux caractères passés à $2.
#  La ligne est préfixée par $3.
afficherliste()
  {
```

```
  echo "$3"${liste[@]:0:$1} ${2:0:1}${liste[$1]}${2:1:1} ${liste[@]:$1+1};
  }

# Boucle _pivot_ -- à partir du second élément jusqu'à la fin de la liste.
for(( i=1; i&amp;lt;numelem; i++ )) do
        ((DEBUG))&amp;amp;&amp;amp;showlist i "[]" " "
        # À partir du _pivot_ actuel, retour au premier élément.
        for(( j=i; j; j-- )) do
                # Recherche du premier élément inférieur au "pivot" actuel...
                [[ "${list[j-1]}" -le "${list[i]}" ]] && break
        done
        (( i==j )) && continue ## Aucune insertion n'était nécessaire pour cet élément.
        # . . . Déplacer liste[i] (pivot) à la gauche de liste[j] :
        liste=(${liste[@]:0:j} ${liste[i]} ${liste[j]}\
        #        {0,j-1}       {i}       {j}
              ${liste[@]:j+1:i-(j+1)} ${liste[@]:i+1})
        #          {j+1,i-1}              {i+1,last}
        ((DEBUG))&amp;amp;&amp;amp;afficherliste j "&amp;lt;&amp;gt;" "*"
done


echo
echo  "------"
echo $'Résultat :\n'${liste[@]}

exit $?
```

Exemple A.39. Un générateur de fichiers *pad* pour les auteurs de shareware

```
#!/bin/bash
# pad.sh

###########################################################
#               PAD (xml) file creator
#+ Written by Mendel Cooper &amp;lt;thegrendel@theriver.com&amp;gt;.
#+ Released to the Public Domain.
#
#  Generates a "PAD" descriptor file for shareware
#+ packages, according to the specifications
#+ of the ASP.
#  http://www.asp-shareware.org/pad
###########################################################


# Accepts (optional) save filename as a command-line argument.
if [ -n "$1" ]
then
  savefile=$1
else
  savefile=save_file.xml               # Default save_file name.
fi


# ===== PAD file headers =====
HDR1="&amp;lt;?xml version=\"1.0\" encoding=\"Windows-1252\" ?&amp;gt;"
HDR2="&amp;lt;XML_DIZ_INFO&amp;gt;"
HDR3="&amp;lt;MASTER_PAD_VERSION_INFO&amp;gt;"
HDR4="\t&amp;lt;MASTER_PAD_VERSION&amp;gt;1.15&amp;lt;/MASTER_PAD_VERSION&amp;gt;"
HDR5="\t&amp;lt;MASTER_PAD_INFO&amp;gt;Portable Application Description, or PAD
for short, is a data set that is used by shareware authors to
disseminate information to anyone interested in their software products.
To find out more go to http://www.asp-shareware.org/pad&amp;lt;/MASTER_PAD_INFO&amp;gt;"
HDR6="&amp;lt;/MASTER_PAD_VERSION_INFO&amp;gt;"
# ============================


fill_in ()
```

```
{
  if [ -z "$2" ]
  then
    echo -n "$1? "     # Get user input.
  else
    echo -n "$1 $2? "  # Additional query?
  fi

  read var             # May paste to fill in field.
                       # This shows how flexible "read" can be.

  if [ -z "$var" ]
  then
    echo -e "\t\t<$1 />" >>$savefile    # Indent with 2 tabs.
    return
  else
    echo -e "\t\t<$1>$var</$1>" >>$savefile
    return ${#var}     # Return length of input string.
  fi
}

check_field_length ()  # Check length of program description fields.
{
  # $1 = maximum field length
  # $2 = actual field length
  if [ "$2" -gt "$1" ]
  then
    echo "Warning: Maximum field length of $1 characters exceeded!"
  fi
}

clear                  # Clear screen.
echo "PAD File Creator"
echo "--- ---- -------"
echo

# Write File Headers to file.
echo $HDR1 >$savefile
echo $HDR2 >>$savefile
echo $HDR3 >>$savefile
echo -e $HDR4 >>$savefile
echo -e $HDR5 >>$savefile
echo $HDR6 >>$savefile


# Company_Info
echo "COMPANY INFO"
CO_HDR="Company_Info"
echo "<$CO_HDR>" >>$savefile

fill_in Company_Name
fill_in Address_1
fill_in Address_2
fill_in City_Town
fill_in State_Province
fill_in Zip_Postal_Code
fill_in Country

# If applicable:
# fill_in ASP_Member "[Y/N]"
# fill_in ASP_Member_Number
# fill_in ESC_Member "[Y/N]"

fill_in Company_WebSite_URL

clear   # Clear screen between sections.

   # Contact_Info
echo "CONTACT INFO"
```

```
CONTACT_HDR="Contact_Info"
echo "<$CONTACT_HDR>" >>$savefile
fill_in Author_First_Name
fill_in Author_Last_Name
fill_in Author_Email
fill_in Contact_First_Name
fill_in Contact_Last_Name
fill_in Contact_Email
echo -e "\t</$CONTACT_HDR>" >>$savefile
   # END Contact_Info

clear

   # Support_Info
echo "SUPPORT INFO"
SUPPORT_HDR="Support_Info"
echo "<$SUPPORT_HDR>" >>$savefile
fill_in Sales_Email
fill_in Support_Email
fill_in General_Email
fill_in Sales_Phone
fill_in Support_Phone
fill_in General_Phone
fill_in Fax_Phone
echo -e "\t</$SUPPORT_HDR>" >>$savefile
   # END Support_Info

echo "</$CO_HDR>" >>$savefile
# END Company_Info

clear

# Program_Info
echo "PROGRAM INFO"
PROGRAM_HDR="Program_Info"
echo "<$PROGRAM_HDR>" >>$savefile
fill_in Program_Name
fill_in Program_Version
fill_in Program_Release_Month
fill_in Program_Release_Day
fill_in Program_Release_Year
fill_in Program_Cost_Dollars
fill_in Program_Cost_Other
fill_in Program_Type "[Shareware/Freeware/GPL]"
fill_in Program_Release_Status "[Beta, Major Upgrade, etc.]"
fill_in Program_Install_Support
fill_in Program_OS_Support "[Win9x/Win2k/Linux/etc.]"
fill_in Program_Language "[English/Spanish/etc.]"

echo; echo

  # File_Info
echo "FILE INFO"
FILEINFO_HDR="File_Info"
echo "<$FILEINFO_HDR>" >>$savefile
fill_in Filename_Versioned
fill_in Filename_Previous
fill_in Filename_Generic
fill_in Filename_Long
fill_in File_Size_Bytes
fill_in File_Size_K
fill_in File_Size_MB
echo -e "\t</$FILEINFO_HDR>" >>$savefile
  # END File_Info

clear

  # Expire_Info
echo "EXPIRE INFO"
```

```
EXPIRE_HDR="Expire_Info"
echo "<$EXPIRE_HDR>" >>$savefile
fill_in Has_Expire_Info "Y/N"
fill_in Expire_Count
fill_in Expire_Based_On
fill_in Expire_Other_Info
fill_in Expire_Month
fill_in Expire_Day
fill_in Expire_Year
echo -e "\t</$EXPIRE_HDR>" >>$savefile
  # END Expire_Info

clear

  # More Program_Info
echo "ADDITIONAL PROGRAM INFO"
fill_in Program_Change_Info
fill_in Program_Specific_Category
fill_in Program_Categories
fill_in Includes_JAVA_VM "[Y/N]"
fill_in Includes_VB_Runtime "[Y/N]"
fill_in Includes_DirectX "[Y/N]"
  # END More Program_Info

echo "</$PROGRAM_HDR>" >>$savefile
# END Program_Info

clear

# Program Description
echo "PROGRAM DESCRIPTIONS"
PROGDESC_HDR="Program_Descriptions"
echo "<$PROGDESC_HDR>" >>$savefile

LANG="English"
echo "<$LANG>" >>$savefile

fill_in Keywords "[comma + space separated]"
echo
echo "45, 80, 250, 450, 2000 word program descriptions"
echo "(may cut and paste into field)"
#  It would be highly appropriate to compose the following
#+ "Char_Desc" fields with a text editor,
#+ then cut-and-paste the text into the answer fields.
echo
echo "              |---------------45 characters---------------|"
fill_in Char_Desc_45
check_field_length 45 "$?"
echo
fill_in Char_Desc_80
check_field_length 80 "$?"

fill_in Char_Desc_250
check_field_length 250 "$?"

fill_in Char_Desc_450
fill_in Char_Desc_2000

echo "</$LANG>" >>$savefile
echo "</$PROGDESC_HDR>" >>$savefile
# END Program Description

clear
echo "Done."; echo; echo
echo "Save file is:  \""$savefile"\""

exit 0
```

Exemple A.40. Pétales autour d'une rose

```
#!/bin/bash -i
# petals.sh

##########################################################################
# Petals Around the Rose                                                 #
#                                                                        #
# Version 0.1 Created by Serghey Rodin                                   #
# Version 0.2 Modded by ABS Guide Author                                 #
#                                                                        #
# License: GPL3                                                          #
# Used in ABS Guide with permission.                                     #
# ###################################################################### #

hits=0      # Correct guesses.
WIN=6       # Mastered the game.
ALMOST=5    # One short of mastery.
EXIT=exit   # Give up early?

RANDOM=$$   # Seeds the random number generator from PID of script.


# Bones (ASCII graphics for dice)
bone1[1]="|         |"
bone1[2]="|       o |"
bone1[3]="|       o |"
bone1[4]="| o     o |"
bone1[5]="| o     o |"
bone1[6]="| o     o |"
bone2[1]="|    o    |"
bone2[2]="|         |"
bone2[3]="|    o    |"
bone2[4]="|         |"
bone2[5]="|    o    |"
bone2[6]="| o     o |"
bone3[1]="|         |"
bone3[2]="| o       |"
bone3[3]="| o       |"
bone3[4]="| o     o |"
bone3[5]="| o     o |"
bone3[6]="| o     o |"
bone="+---------+"



# Functions

instructions () {

  clear
  echo -n "Do you need instructions? (y/n) "; read ans
  if [ "$ans" = "y" -o "$ans" = "Y" ]; then
    clear
    echo -e '\E[34;47m'  # Blue type.

#  "cat document"
    cat <<INSTRUCTIONSZZZ
The name of the game is Petals Around the Rose,
and that name is significant.
Five dice will roll and you must guess the "answer" for each roll.
It will be zero or an even number.
After your guess, you will be told the answer for the roll, but . . .
that's ALL the information you will get.

Six consecutive correct guesses admits you to the
Fellowship of the Rose.
INSTRUCTIONSZZZ
```

```
    echo -e "\033[0m"    # Turn off blue.
    else clear
  fi

}


fortune ()
{
  RANGE=7
  FLOOR=0
  number=0
  while [ "$number" -le $FLOOR ]
  do
    number=$RANDOM
    let "number %= $RANGE"   # 1 - 6.
  done

  return $number
}



throw () { # Calculate each individual die.
  fortune; B1=$?
  fortune; B2=$?
  fortune; B3=$?
  fortune; B4=$?
  fortune; B5=$?

  calc () { # Function embedded within a function!
    case "$1" in
       3   ) rose=2;;
       5   ) rose=4;;
       *   ) rose=0;;
    esac    # Simplified algorithm.
            # Doesn't really get to the heart of the matter.
    return $rose
  }

  answer=0
  calc "$B1"; answer=$(expr $answer + $(echo $?))
  calc "$B2"; answer=$(expr $answer + $(echo $?))
  calc "$B3"; answer=$(expr $answer + $(echo $?))
  calc "$B4"; answer=$(expr $answer + $(echo $?))
  calc "$B5"; answer=$(expr $answer + $(echo $?))
}



game ()
{ # Generate graphic display of dice throw.
  throw
    echo -e "\033[1m"    # Bold.
  echo -e "\n"
  echo -e "$bone\t$bone\t$bone\t$bone\t$bone"
  echo -e \
 "${bone1[$B1]}\t${bone1[$B2]}\t${bone1[$B3]}\t${bone1[$B4]}\t${bone1[$B5]}"
  echo -e \
 "${bone2[$B1]}\t${bone2[$B2]}\t${bone2[$B3]}\t${bone2[$B4]}\t${bone2[$B5]}"
  echo -e \
 "${bone3[$B1]}\t${bone3[$B2]}\t${bone3[$B3]}\t${bone3[$B4]}\t${bone3[$B5]}"
  echo -e "$bone\t$bone\t$bone\t$bone\t$bone"
  echo -e "\n\n\t\t"
    echo -e "\033[0m"    # Turn off bold.
  echo -n "There are how many petals around the rose? "
}
```

```
# ============================================================ #

instructions

while [ "$petal" != "$EXIT" ]    # Main loop.
do
  game
  read petal
  echo "$petal" | grep [0-9] >/dev/null  # Filter response for digit.
                                         # Otherwise just roll dice again.
  if [ "$?" -eq 0 ]   # If-loop #1.
  then
    if [ "$petal" == "$answer" ]; then    # If-loop #2.
          echo -e "\nCorrect. There are $petal petals around the rose.\n"
        (( hits++ ))

        if [ "$hits" -eq "$WIN" ]; then   # If-loop #3.
          echo -e '\E[31;47m'  # Red type.
          echo -e "\033[1m"    # Bold.
          echo "You have unraveled the mystery of the Rose Petals!"
          echo "Welcome to the Fellowship of the Rose!!!"
          echo "(You are herewith sworn to secrecy.)"; echo
          echo -e "\033[0m"    # Turn off red and bold.
          break               # Exit!
        else echo "You have $hits correct so far."; echo

        if [ "$hits" -eq "$ALMOST" ]; then
          echo "Just one more gets you to the heart of the mystery!"; echo
        fi

      fi                                  # Close if-loop #3.

    else
      echo -e "\nWrong. There are $answer petals around the rose.\n"
      hits=0   # Reset number of correct guesses.
    fi                                    # Close if-loop #2.

    echo -n "Hit ENTER for the next roll, or type \"exit\" to end. "
    read
    if [ "$REPLY" = "$EXIT" ]; then exit
    fi

  fi                  # Close if-loop #1.

  clear
done                  # End of main (while) loop.

###

exit $?

# Resources:
# ---------
# 1) http://en.wikipedia.org/wiki/Petals_Around_the_Rose
#    (Wikipedia entry.)
# 2) http://www.borrett.id.au/computing/petals-bg.htm
#    (How Bill Gates coped with the Petals Around the Rose challenge.)
```

Exemple A.41. Quacky : un jeu de mots de type Perquackey

```
#!/bin/bash
# qky.sh

##################################################################
# QUACKEY: a somewhat simplified version of Perquackey [TM]. #
```

```
#                                                                     #
# Author: Mendel Cooper  <thegrendel@theriver.com>                    #
# version 0.1.02      03 May, 2008                                    #
# License: GPL3                                                       #
#######################################################################

WLIST=/usr/share/dict/word.lst
#                     ^^^^^^^^  Word list file found here.
#  ASCII word list, one word per line, UNIX format.
#  A suggested list is the script author's "yawl" word list package.
#  http://ibiblio.org/pub/Linux/libs/yawl-0.3.2.tar.gz
#    or
#  http://personal.riverusers.com/~thegrendel/yawl-0.3.2.tar.gz

NONCONS=0     # Word not constructable from letter set.
CONS=1        # Constructable.
SUCCESS=0
NG=1
FAILURE=''
NULL=0        # Zero out value of letter (if found).
MINWLEN=3     # Minimum word length.
MAXCAT=5      # Maximum number of words in a given category.
PENALTY=200   # General-purpose penalty for unacceptable words.
total=
E_DUP=70      # Duplicate word error.

TIMEOUT=10    # Time for word input.

NVLET=10      # 10 letters for non-vulnerable.
VULET=13      # 13 letters for vulnerable (not yet implemented).

declare -a Words
declare -a Status
declare -a Score=( 0 0 0 0 0 0 0 0 0 0 0 )


letters=( a n s r t m l k p r b c i d s i d z e w u e t f
e y e r e f e g t g h h i t r s c i t i d i j a t a o l a
m n a n o v n w o s e l n o s p a q e e r a b r s a o d s
t g t i t l u e u v n e o x y m r k )
#  Letter distribution table shamelessly borrowed from "Wordy" game,
#+ ca. 1992, written by a certain fine fellow named Mendel Cooper.

declare -a LS

numelements=${#letters[@]}
randseed="$1"

instructions ()
{
  clear
  echo "Welcome to QUACKEY, the anagramming word construction game."; echo
  echo -n "Do you need instructions? (y/n) "; read ans

   if [ "$ans" = "y" -o "$ans" = "Y" ]; then
     clear
     echo -e '\E[31;47m'  # Red foreground. '\E[34;47m' for blue.
     cat <<INSTRUCTION1

QUACKEY is a variant of Perquackey [TM].
The rules are the same, but the scoring is simplified
and plurals of previously played words are allowed.
"Vulnerable" play is not yet implemented,
but it is otherwise feature-complete.

As the game begins, the player gets 10 letters.
The object is to construct valid dictionary words
of at least 3-letter-length from the letterset.
Each word-length category
```

```
-- 3-letter, 4-letter, 5-letter, ... --
fills up with the fifth word entered,
and no further words in that category are accepted.

The penalty for too-short (two-letter), duplicate, unconstructable,
and invalid (not in dictionary) words is -200. The same penalty applies
to attempts to enter a word in a filled-up category.

INSTRUCTION1

  echo -n "Hit ENTER for next page of instructions. "; read az1

     cat <<INSTRUCTION2

The scoring mostly corresponds to classic Perquackey:
The first 3-letter word scores    60, plus   10 for each additional one.
The first 4-letter word scores   120, plus   20 for each additional one.
The first 5-letter word scores   200, plus   50 for each additional one.
The first 6-letter word scores   300, plus  100 for each additional one.
The first 7-letter word scores   500, plus  150 for each additional one.
The first 8-letter word scores   750, plus  250 for each additional one.
The first 9-letter word scores  1000, plus  500 for each additional one.
The first 10-letter word scores 2000, plus 2000 for each additional one.

Category completion bonuses are:
3-letter words   100
4-letter words   200
5-letter words   400
6-letter words   800
7-letter words  2000
8-letter words 10000
This is a simplification of the absurdly complicated Perquackey bonus
scoring system.

INSTRUCTION2

  echo -n "Hit ENTER for final page of instructions. "; read az1

     cat <<INSTRUCTION3


Hitting just ENTER for a word entry ends the game.

Individual word entry is timed to a maximum of 10 seconds.
*** Timing out on an entry ends the game. ***
Other than that, the game is untimed.

-------------------------------------------------
Game statistics are automatically saved to a file.
-------------------------------------------------

For competitive ("duplicate") play, a previous letterset
may be duplicated by repeating the script's random seed,
command-line parameter \$1.
For example, "qky 7633" specifies the letterset
c a d i f r h u s k ...
INSTRUCTION3

  echo; echo -n "Hit ENTER to begin game. "; read az1

       echo -e "\033[0m"    # Turn off red.
     else clear
  fi

  clear

}
```

```
seed_random ()
{                             #  Seed random number generator.
  if [ -n "$randseed" ]       #  Can specify random seed.
  then                        #+ for play in competitive mode.
#   RANDOM="$randseed"
    echo "RANDOM seed set to "$randseed""
  else
    randseed="$$"             # Or get random seed from process ID.
    echo "RANDOM seed not specified, set to Process ID of script ($$)."
  fi

  RANDOM="$randseed"

  echo
}


get_letset ()
{
  element=0
  echo -n "Letterset:"

  for lset in $(seq $NVLET)
  do  # Pick random letters to fill out letterset.
    LS[element]="${letters[$((RANDOM%numelements))]}"
    ((element++))
  done

  echo
  echo "${LS[@]}"

}


add_word ()
{
  wrd="$1"
  local idx=0

  Status[0]=""
  Status[3]=""
  Status[4]=""

  while [ "${Words[idx]}" != '' ]
  do
    if [ "${Words[idx]}" = "$wrd" ]
    then
      Status[3]="Duplicate-word-PENALTY"
      let "Score[0]= 0 - $PENALTY"
      let "Score[1]-=$PENALTY"
      return $E_DUP
    fi

    ((idx++))
  done

  Words[idx]="$wrd"
  get_score

}

get_score()
{
  local wlen=0
  local score=0
  local bonus=0
  local first_word=0
  local add_word=0
```

```
  local numwords=0

  wlen=${#wrd}
  numwords=${Score[wlen]}
  Score[2]=0
  Status[4]=""   # Initialize "bonus" to 0.

  case "$wlen" in
    3) first_word=60
       add_word=10;;
    4) first_word=120
       add_word=20;;
    5) first_word=200
       add_word=50;;
    6) first_word=300
       add_word=100;;
    7) first_word=500
       add_word=150;;
    8) first_word=750
       add_word=250;;
    9) first_word=1000
       add_word=500;;
   10) first_word=2000
       add_word=2000;;   # This category modified from original rules!
      esac

  ((Score[wlen]++))
  if [ ${Score[wlen]} -eq $MAXCAT ]
  then   # Category completion bonus scoring simplified!
    case $wlen in
      3 ) bonus=100;;
      4 ) bonus=200;;
      5 ) bonus=400;;
      6 ) bonus=800;;
      7 ) bonus=2000;;
      8 ) bonus=10000;;
    esac  # Needn't worry about 9's and 10's.
    Status[4]="Category-$wlen-completion***BONUS***"
    Score[2]=$bonus
  else
    Status[4]=""   # Erase it.
  fi


    let "score =  $first_word +   $add_word * $numwords"
    if [ "$numwords" -eq 0 ]
    then
      Score[0]=$score
    else
      Score[0]=$add_word
    fi   #  All this to distinguish last-word score
         #+ from total running score.
  let "Score[1] += ${Score[0]}"
  let "Score[1] += ${Score[2]}"

}



get_word ()
{
  local wrd=''
  read -t $TIMEOUT wrd   # Timed read.
  echo $wrd
}

is_constructable ()
{ # This was the most complex and difficult-to-write function.
  local -a local_LS=( "${LS[@]}" )  # Local copy of letter set.
```

```
  local is_found=0
  local idx=0
  local pos
  local strlen
  local local_word=( "$1" )
  strlen=${#local_word}

  while [ "$idx" -lt "$strlen" ]
  do
    is_found=$(expr index "${local_LS[*]}" "${local_word:idx:1}")
    if [ "$is_found" -eq "$NONCONS" ] # Not constructable!
    then
      echo "$FAILURE"; return
    else
      ((pos = ($is_found - 1) / 2))   # Compensate for spaces betw. letters!
      local_LS[pos]=$NULL             # Zero out used letters.
      ((idx++))                       # Bump index.
    fi
  done

  echo "$SUCCESS"
  return
}

is_valid ()
{ # Surprisingly easy to check if word in dictionary ...
  fgrep -qw "$1" "$WLIST"   # ... thanks to 'grep' ...
  echo $?
}

check_word ()
{
  if [ -z "$1" ]
  then
    return
  fi

  Status[1]=""
  Status[2]=""
  Status[3]=""
  Status[4]=""

  iscons=$(is_constructable "$1")
  if [ "$iscons" ]
  then
    Status[1]="constructable"
    v=$(is_valid "$1")
    if [ "$v" -eq "$SUCCESS" ]
    then
      Status[2]="valid"
      strlen=${#1}

      if [ ${Score[strlen]} -eq "$MAXCAT" ]   # Category full!
      then
        Status[3]="Category-$strlen-overflow-PENALTY"
        return $NG
      fi

      case "$strlen" in
        1 | 2 )
        Status[3]="Two-letter-word-PENALTY"
        return $NG;;
        * )
        Status[3]=""
        return $SUCCESS;;
      esac
    else
      Status[3]="Not-valid-PENALTY"
      return $NG
```

```
    fi
  else
    Status[3]="Not-constructable-PENALTY"
      return $NG
  fi

  ### FIXME: Streamline the above code.

}


display_words ()
{
  local idx=0
  local wlen0

  clear
  echo "Letterset:   ${LS[@]}"
  echo "Threes:    Fours:    Fives:     Sixes:    Sevens:    Eights:"
  echo "------------------------------------------------------------"



  while [ "${Words[idx]}" != '' ]
  do
   wlen0=${#Words[idx]}
   case "$wlen0" in
     3) ;;
     4) echo -n "           " ;;
     5) echo -n "                      " ;;
     6) echo -n "                                 " ;;
     7) echo -n "                                            " ;;
     8) echo -n "                                                       " ;;
   esac
   echo "${Words[idx]}"
   ((idx++))
  done

  ### FIXME: The word display is pretty crude.
}


play ()
{
  word="Start game"   # Dummy word, to start ...

  while [ "$word" ]   #  If player just hits return (blank word),
  do                  #+ then game ends.
    echo "$word: "${Status[@]}""
    echo -n "Last score: [${Score[0]}]   TOTAL score: [${Score[1]}]:     Next word: "
    total=${Score[1]}
    word=$(get_word)
    check_word "$word"

    if [ "$?" -eq "$SUCCESS" ]
    then
      add_word "$word"
    else
      let "Score[0]= 0 - $PENALTY"
      let "Score[1]-=$PENALTY"
    fi

  display_words
  done   # Exit game.

  ### FIXME: The play () function calls too many other functions.
  ### This is perilously close to "spaghetti code" ...
}
```

```
end_of_game ()
{ # Save and display stats.

  ######################Autosave#############################
  savefile=qky.save.$$
  #                 ^^ PID of script
  echo `date` >> $savefile
  echo "Letterset # $randseed  (random seed) ">> $savefile
  echo -n "Letterset: " >> $savefile
  echo "${LS[@]}" >> $savefile
  echo "---------" >> $savefile
  echo "Words constructed:" >> $savefile
  echo "${Words[@]}" >> $savefile
  echo >> $savefile
  echo "Score: $total" >> $savefile

  echo "Statistics for this round saved in \""$savefile"\""
  ###########################################################

  echo "Score for this round: $total"
  echo "Words:  ${Words[@]}"
}

# ---------#
instructions
seed_random
get_letset
play
end_of_game
# ---------#

exit $?

# TODO:
#
# 1) Clean up code!
# 2) Prettify the display_words () function (maybe with widgets?).
# 3) Improve the time-out ... maybe change to untimed entry,
#+   but with a time limit for the overall round.
# 4) An on-screen countdown timer would be nice.
# 5) Implement "vulnerable" mode of play.
# 6) Improve save-to-file capability (and maybe make it optional).
# 7) Fix bugs!!!
```

Exemple A.42. Un outil de résolution général

```
#/bin/bash
# homework.sh: General purpose homework assignment solution.
#  Author: M. Leo Cooper
#  If you substitute your own name as author, then it is plagiarism,
#+ probably a lesser sin than cheating on your homework.
#  License: Public Domain

#  This script may be turned in to your instructor
#+ in fulfillment of ALL Bash scripting homework assignments.
#  It's sparsely commented, but you, the student, can easily remedy that.
#  The script author repudiates all responsibility!

DLA=1
P1=2
P2=4
P3=7
PP1=0
PP2=8
MAXL=9
E_LZY=99
```

```
declare -a L
L[0]="3 4 0 17 29 8 13 18 19 17 20 2 19 14 17 28"
L[1]="8 29 12 14 18 19 29 4 12 15 7 0 19 8 2 0 11 11 24 29 17 4 6 17 4 19"
L[2]="29 19 7 0 19 29 8 29 7 0 21 4 29 13 4 6 11 4 2 19 4 3"
L[3]="19 14 29 2 14 12 15 11 4 19 4 29 19 7 8 18 29"
L[4]="18 2 7 14 14 11 22 14 17 10 29 0 18 18 8 6 13 12 4 13 19 26"
L[5]="15 11 4 0 18 4 29 0 2 2 4 15 19 29 12 24 29 7 20 12 1 11 4 29"
L[6]="4 23 2 20 18 4 29 14 5 29 4 6 17 4 6 8 14 20 18 29"
L[7]="11 0 25 8 13 4 18 18 27"
L[8]="0 13 3 29 6 17 0 3 4 29 12 4 29 0 2 2 14 17 3 8 13 6 11 24 26"
L[9]="19 7 0 13 10 29 24 14 20 26"

declare -a \
alph=( A B C D E F G H I J K L M N O P Q R S T U V W X Y Z . , : ' ' )


pt_lt ()
{
  echo -n "${alph[$1]}"
  echo -n -e "\a"
  sleep $DLA
}

b_r ()
{
 echo -e '\E[31;48m\033[1m'
}

cr ()
{
 echo -e "\a"
 sleep $DLA
}

restore ()
{
  echo -e '\033[0m'            # Bold off.
  tput sgr0                    # Normal.
}


p_l ()
{
  for ltr in $1
  do
    pt_lt "$ltr"
  done
}

# ----------------------
b_r

for i in $(seq 0 $MAXL)
do
 p_l "${L[i]}"
 if [[ "$i" -eq "$P1" || "$i" -eq "$P2" || "$i" -eq "$P3" ]]
 then
   cr
   elif [[ "$i" -eq "$PP1" || "$i" -eq "$PP2" ]]
   then
     cr; cr
fi
done

restore
# ----------------------

echo
```

```
exit $E_LZY

#  Un exemple de script "obfusqué" volontairement pour être difficile à
#+ comprendre et frustrant à maintenir.
#  Dans votre carrière d'aministrateur système, vous rencontrerez ceci
#+ trop souvent.
```

Pour finir cette section, une revue des bases... et plus encore.

Exemple A.43. Basics Reviewed

```
#!/bin/bash
# basics-reviewed.bash

# Extension du fichier == *.bash == spécifique à Bash

#   Copyright (c) Michael S. Zick, 2003; All rights reserved.
#   License: Use in any form, for any purpose.
#   Revision: $ID$
#
#             Édité pour la présentation par M.C.
#   (auteur du "Guide d'écriture avancée des scripts Bash")
#   Corrections et mises à jour (04/08) par Cliff Bamford.


#  Ce script a été testé sous Bash version 2.04, 2.05a et
#+ 2.05b.
#  Il pourrait ne pas fonctionner avec les versions précédentes.
#  Ce script de démonstration génère une erreur "command not found"
#+ --intentionnelle--. Voir ligne 436.

#  Le mainteneur actuel de Bash maintainer, Chet Ramey, a corrigé les éléments
#+ notés pour les versions ultérieures de Bash.



        ###-----------------------------------------###
        ###  Envoyez la sortie de ce script à 'more'  ###
        ###+ sinon cela dépassera la page.            ###
        ###                                          ###
        ###  Vous pouvez aussi rediriger sa sortie    ###
        ###+ vers un fichier pour l'examiner.         ###
        ###-----------------------------------------###



#  La plupart des points suivants sont décrit en détail dans
#+ le guide d'écriture avancé du script Bash.
#  Ce script de démonstration est principalement une présentation réorganisée.
#      -- msz

# Les variables ne sont pas typées sauf cas indiqués.

#  Les variables sont nommées. Les noms doivent contenir un caractère qui
#+ n'est pas un chiffre.
#  Les noms des descripteurs de fichiers (comme dans, par exemple, 2>&1)
#+ contiennent UNIQUEMENT des chiffres.

# Les paramètres et les éléments de tavbleau Bash sont numérotés.
# (Les paramètres sont très similaires aux tableaux Bash.)

# Un nom de variable pourrait être indéfini (référence nulle).
unset VarNullee

# Un nom de variable pourrait être défini mais vide (contenu nul).
VarVide=''                          # Deux guillemets simples, adjacents.
```

```
# Un nom de variable pourrait être défini et non vide.
VarQuelquechose='Littéral'

# Une variable pourrait contenir:
#   * Un nombre complet, entier signé sur 32-bit (voire plus)
#   * Une chaîne
# Une variable pourrait aussi être un tableau.

#  Une chaîne pourrait contenir des espaces et pourrait être traitée
#+ comme s'il s'agissait d'un nom de fonction avec des arguments optionnelles.

#  Les noms des variables et les noms des functions sont dans différents
#+ espaces de noms.


#  Une variable pourrait être défini comme un tableau Bash soit explicitement
#+ soit implicitement par la syntaxe de l'instruction d'affectation.
#  Explicite:
declare -a VarTableau



# La commande echo est intégrée.
echo $VarQuelquechose

# La commande printf est intégrée.
# Traduire %s comme "Format chaîne"
printf %s $VarQuelquechose      #  Pas de retours chariot spécifiés,
                                #+ aucune sortie.
echo                            # Par défaut, seulement un retour chariot.




#  L'analyseur de mots de Bash s'arrête sur chaque espace blanc mais son
#+ manquement est significatif.
#  (Ceci reste vrai en général ; Il existe évidemment des exceptions.)



# Traduire le signe SIGNE_DOLLAR comme Contenu-de.

# Syntaxe étendue pour écrire Contenu-de :
echo ${VarQuelquechose}

#  La syntaxe étendue ${ ... } permet de spécifier plus que le nom de la
#+ variable.
#  En général, $VarQuelquechose peut toujours être écrit ${VarQuelquechose}.

# Appelez ce script avec des arguments pour visualiser l'action de ce qui suit.



#  En dehors des doubles guillemets, les caractères spéciaux @ et *
#+ spécifient un comportement identique.
#  Pourrait être prononcé comme Tous-Éléments-De.

#  Sans spécifier un nom, ils réfèrent un paramètre prédéfini Bash-Array.



# Références de modèles globaux
echo $*                     # Tous les paramètres du script ou de la fonction
echo ${*}                   # Pareil

# Bash désactive l'expansion de nom de fichier pour les modèles globaux.
# Seuls les caractères correspondants sont actifs.
```

```
# Références de Tous-Éléments-De
echo $@                         # Identique à ci-dessus
echo ${@}                       # Identique à ci-dessus



#  À l'intérieur des guillemets doubles, le comportement des références de
#+ modèles globaux dépend du paramètrage de l'IFS (Input Field Separator, soit
#+ séparateur de champ d'entrée).
#  À l'intérieur des guillemets doubles, les références à Tous-Éléments-De
#+ se comportent de façon identique.


#  Spécifier uniquement le nom de la variable contenant une chaîne réfère tous
#+ les éléments (caractères) d'une chaîne.


#  Spécifier un élément (caractère) d'une chaîne,
#+ la notation de référence de syntaxe étendue (voir ci-dessous) POURRAIT être
#+ utilisée.



#  Spécifier uniquement le nom d'un tableau Bash référence l'élément 0,
#+ PAS le PREMIER DÉFINI, PAS le PREMIER AVEC CONTENU.

#  Une qualification supplémentaire est nécessaire pour référencer d'autres
#+ éléments, ce qui signifie que la référence DOIT être écrite dans la syntaxe
#+ étendue. La forme générale est ${nom[indice]}.

#  Le format de chaîne pourrait aussi être utilisé ${nom:indice}
#+ pour les tableaux Bash lors de la référence de l'élément zéro.


#  Les tableaux Bash sont implémentés en interne comme des listes liés,
#+ pas comme une aire fixe de stockage comme le font certains langages de
#+ programmation.


#   Caractéristiques des tableaux Bash (Bash-Arrays):
#   -------------------------------------------------

#   Sans autre indication, les indices des tableaux Bash
#+  commencent avec l'indice numéro 0. Littéralement : [0]
#   Ceci s'appelle un indice base 0.
###
#   Sans autre indication, les tableaux Bash ont des indices continus
#+  (indices séquentielles, sans trou/manque).
###
#   Les indices négatifs ne sont pas autorisés.
###
#   Les éléments d'un tableau Bash n'ont pas besoin de tous être du même type.
###
#   Les éléments d'un tableau Bash pourraient être indéfinis (référence nulle).
#       C'est-à-dire qu'un tableau Bash pourrait être "subscript sparse."
###
#   Les éléments d'un tableau Bash pourraient être définis et vides
#+  (contenu nul).
###
#   Les éléments d'un tableau Bash pourraient être :
#     * Un entier codé sur 32 bits (ou plus)
#     * Une chaîne
#     * Une chaîne formattée de façon à ce qu'elle soit en fait le nom d'une
#       fonction avec des arguments optionnelles
###
#   Les éléments définis d'un tableau Bash pourraient ne pas être définis
```

```
#   (unset).
#       C'est-à-dire qu'un tableau Bash à indice continu pourrait être modifié
#       en un tableau Bash à indice disparate.
###
#   Des éléments pourraient être ajoutés dans un tableau Bash en définissant un
#   élément non défini précédemment.
###
# Pour ces raisons, je les ai appelé des tableaux Bash ("Bash-Arrays").
# Je retourne maintenant au terme générique "tableau".
#     -- msz



echo "============================================================"

#  Lignes 202 à 334 fournies par Cliff Bamford. (Merci !)
#  Démo --- Interaction avec les tableaux, les guillemets, IFS, echo, * et @
#+ --- tous modifient la façon dont cela fonctionne

ArrayVar[0]='zero'                    # 0 normal
ArrayVar[1]=one                       # 1 valeur litérale sans guillemet
ArrayVar[2]='two'                     # 2 normal
ArrayVar[3]='three'                   # 3 normal
ArrayVar[4]='I am four'               # 4 normal avec des espaces
ArrayVar[5]='five'                    # 5 normal
unset ArrayVar[6]                     # 6 indéfini
ArrayValue[7]='seven'                 # 7 normal
ArrayValue[8]=''                      # 8 défini mais vide
ArrayValue[9]='nine'                  # 9 normal


echo '--- Voici le tableau que nous utilisons pour ce test'
echo
echo "ArrayVar[0]='zero'              # 0 normal"
echo "ArrayVar[1]=one                 # 1 valeur litérale sans guillemet"
echo "ArrayVar[2]='two'               # 2 normal"
echo "ArrayVar[3]='three'             # 3 normal"
echo "ArrayVar[4]='I am four'         # 4 normal avec des espaces"
echo "ArrayVar[5]='five'              # 5 normal"
echo "unset ArrayVar[6]               # 6 indéfini"
echo "ArrayValue[7]='seven'           # 7 normal"
echo "ArrayValue[8]=''                # 8 défini mais vide"
echo "ArrayValue[9]='nine'            # 9 normal"
echo


echo
echo '---Cas 0 : Sans double guillemets, IFS par défaut (espace, tabulation, retour à
la ligne) ---'
IFS=$'\x20'$'\x09'$'\x0A'             # Exactement dans cet ordre.
echo 'Voici : printf %q {${ArrayVar[*]}'
printf %q ${ArrayVar[*]}
echo
echo 'Voici : printf %q {${ArrayVar[@]}'
printf %q ${ArrayVar[@]}
echo
echo 'Voici : echo ${ArrayVar[*]}'
echo  ${ArrayVar[@]}
echo 'Voici : echo {${ArrayVar[@]}'
echo ${ArrayVar[@]}

echo
echo '---Cas 1 : Dans des guillemets doubles - IFS par défaut ---'
IFS=$'\x20'$'\x09'$'\x0A'     #  These three bytes,
echo 'Voici : printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Voici : printf %q "{${ArrayVar[@]}"'
```

```
printf %q "${ArrayVar[@]}"
echo
echo 'Voici : echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Voici : echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Cas 2 : Dans des guillemets doubles - IFS vaut q'
IFS='q'
echo 'Voici : printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Voici : printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Voici : echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Voici : echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Cas 3 : Dans des guillemets doubles - IFS vaut ^'
IFS='^'
echo 'Voici : printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Voici : printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Voici : echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Voici : echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Cas 4 : Dans des guillemets doubles - IFS vaut ^ suivi par
espace, tabulation, retour à la ligne'
IFS=$'^'$'\x20'$'\x09'$'\x0A'       # ^ + espace tabulation retour à la ligne
echo 'Voici : printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Voici : printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Voici : echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Voici : echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Cas 6 : Dans des guillemets doubles - IFS configuré mais vide'
IFS=''
echo 'Voici : printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
echo
echo 'Voici : printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Voici : echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Voici : echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Cas 7 : Dans des guillemets doubles - IFS indéfini'
unset IFS
echo 'Voici : printf %q "{${ArrayVar[*]}"'
printf %q "${ArrayVar[*]}"
```

```
echo
echo 'Voici : printf %q "{${ArrayVar[@]}"'
printf %q "${ArrayVar[@]}"
echo
echo 'Voici : echo "${ArrayVar[*]}"'
echo  "${ArrayVar[@]}"
echo 'Voici : echo "{${ArrayVar[@]}"'
echo "${ArrayVar[@]}"

echo
echo '---Fin des cas---'
echo "============================================================"; echo



# Remettre la valeur par défaut d'IFS.
# Par défaut, il s'agit exactement de ces trois octets.
IFS=$'\x20'$'\x09'$'\x0A'           # Dans cet ordre.

# Interprétation des affichages précédents :
#   Un modèle global est de l'entrée/sortie ; le paramètrage de l'IFS est pris
en compte.
###
#   Un Tous-Éléments-De ne prend pas en compte le paramètrage de l'IFS.
###
#   Notez les affichages différents en utilisant la commande echo et l'opérateur
#+ de format entre guillemets de la commande printf.


#  Rappel :
#   Les paramètres sont similaires aux tableaux et ont des comportements
similaires.
###
#  Les exemples ci-dessous démontrent les variantes possibles.
#  Pour conserver la forme d'un tableau à indice non continu, un supplément au
script
#+ est requis.
###
#  Le code source de Bash dispose d'une routine d'affichage du format
#+ d'affectation [indice]=valeur   .
#  Jusqu'à la version 2.05b, cette routine n'est pas utilisée
#+ mais cela pourrait changer dans les versions suivantes.



# La longueur d'une chaîne, mesurée en éléments non nuls (caractères) :
echo
echo '- - Références sans guillemets - -'
echo 'Nombre de caractères non nuls : '${#VarQuelquechose}' caractères.'

# test='Lit'$'\x00''eral'           # $'\x00' est un caractère nul.
# echo ${#test}                     # Vous avez remarqué ?



#  La longueur d'un tableau, mesurée en éléments définis,
#+ ceci incluant les éléments à contenu nul.
echo
echo 'Nombre de contenu défini : '${#VarTableau[@]}' éléments.'
# Ce n'est PAS l'indice maximum (4).
# Ce n'est PAS l'échelle des indices (1...4 inclus).
# C'EST la longueur de la liste chaînée.
###
#  L'indice maximum et l'échelle d'indices pourraient être trouvées avec
#+ un peu de code supplémentaire.

# La longueur d'une chaîne, mesurée en éléments non nuls (caractères):
echo
echo '- - Références du modèle global, entre guillemets - -'
```

```
echo 'Nombre de caractères non nuls : '"${#VarQuelquechose}"'.'

#  La longueur d'un tableau, mesuré avec ses éléments définis,
#+ ceci incluant les éléments à contenu nul.
echo
echo "Nombre d'éléments définis: '"${#VarTableau[*]}"' éléments."

#  Interprétation : la substitution n'a pas d'effet sur l'opération ${# ... }.
#  Suggestion :
#  Toujours utiliser le caractère Tous-Éléments-De
#+ si cela correspond au comportement voulu (indépendence par rapport à l'IFS).



#  Définir une fonction simple.
#  J'inclus un tiret bas dans le nom pour le distinguer des exemples ci-dessous.
###
#  Bash sépare les noms de variables et les noms de fonctions
#+ grâce à des espaces de noms différents.
#  The Mark-One eyeball isn't that advanced.
###
_simple() {
    echo -n 'FonctionSimple'$@      #  Les retours chariots disparaissent dans
le résultat.
}


# La notation ( ... ) appelle une commande ou une fonction.
# La notation $( ... ) est prononcée Résultat-De.


# Appelle la fonction _simple
echo
echo '- - Sortie de la fonction _simple - -'
_simple                             # Essayez de passer des arguments.
echo
# or
(_simple)                           # Essayez de passer des arguments.
echo

echo "- Existe-t'il une variable de ce nom ? -"
echo $_simple indéfinie             # Aucune variable de ce nom.

# Appelle le résultat de la fonction _simple (message d'erreur attendu)

###
$(_simple)                          # Donne un message d'erreur :
#                          line 436: FonctionSimple: command not found
#                          ----------------------------------------

echo
###

#  Le premier mot du résultat de la fonction _simple
#+ n'est ni une commande Bash valide ni le nom d'une fonction définie.
###
# Ceci démontre que la sortie de _simple est sujet à évaluation.
###
# Interprétation :
#   Une fonction peut être utilisée pour générer des commandes Bash en ligne.


# Une fonction simple où le premier mot du résultat EST une commande Bash :
###
_print() {
    echo -n 'printf %q '$@
}

echo '- - Affichage de la fonction _print - -'
```

```
_print parm1 parm2                  # Une sortie n'est PAS une commande.
echo

$(_print parm1 parm2)               #  Exécute : printf %q parm1 parm2
                                    #  Voir ci-dessus les exemples IFS
                                    #+ pour les nombreuses possibilités.
echo

$(_print $VarQuelquechose)             # Le résultat prévisible.
echo



# Variables de fonctions
# ----------------------

echo
echo '- - Variables de fonctions - -'
# Une variable pourrait représenter un entier signé, une chaîne ou un tableau.
# Une chaîne pourrait être utilisée comme nom de fonction avec des arguments
optionnelles.

# set -vx                           #  À activer si désiré
declare -f funcVar                  #+ dans l'espace de noms des fonctions

funcVar=_print                      # Contient le nom de la fonction.
$funcVar parm1                      # Identique à _print à ce moment.
echo

funcVar=$(_print )                  # Contient le résultat de la fonction.
$funcVar                            # Pas d'entrée, pas de sortie.
$funcVar $VarQuelquechose           # Le résultat prévisible.
echo

funcVar=$(_print $VarQuelquechose)  #  $VarQuelquechose remplacé ICI.
$funcVar                            #  L'expansion fait parti du contenu
echo                                #+ des variables.

funcVar="$(_print $VarQuelquechose)" #  $VarQuelquechose remplacé ICI.
$funcVar                             #  L'expansion fait parti du contenu
echo                                #+ des variables.

#  La différence entre les versions sans guillemets et avec double guillemets
#+ ci-dessus est rencontrée dans l'exemple "protect_literal.sh".
#  Le premier cas ci-dessus est exécuté comme deux mots Bash sans guillemets.
#  Le deuxième cas est exécuté comme un mot Bash avec guillemets.




# Remplacement avec délai
# -----------------------

echo
echo '- - Remplacement avec délai - -'
funcVar="$(_print '$VarQuelquechose')" # Pas de remplacement, simple mot Bash.
eval $funcVar                          # $VarQuelquechose remplacé ICI.
echo

VarQuelquechose='NouvelleChose'
eval $funcVar                       # $VarQuelquechose remplacé ICI.
echo

# Restaure la configuration initiale.
VarQuelquechose=Literal

#  Il existe une paire de fonctions démontrées dans les exemples
#+ "protect_literal.sh" et "unprotect_literal.sh".
#  Il s'agit de fonctions à but général pour des littérales à remplacements avec
```

```
délai
#+ contenant des variables.




# REVUE :
# ------

#  Une chaîne peut être considérée comme un tableau classique d'éléments de type
#+ caractère.
#  Une opération sur une chaîne s'applique à tous les éléments (caractères) de
#+ la chaîne (enfin, dans son concept).
###
#  La notation ${nom_tableau[@]} représente tous les éléments du tableau Bash
#+ nom_tableau.
###
#  Les opérations sur les chaînes de syntaxe étendue sont applicables à tous les
#+ éléments d'un tableau.
###
#  Ceci peut être pensé comme une boucle For-Each sur un vecteur de chaînes.
###
#  Les paramètres sont similaires à un tableau.
#  L'initialisation d'un paramètre de type tableau pour un script
#+ et d'un paramètre de type tableau pour une fonction diffèrent seulement
#+ dans l'initialisation de ${0}, qui ne change jamais sa configuration.
###
#  L'indice zéro du tableau, paramètre d'un script, contient le nom du script.
###
#  L'indice zéro du tableau, paramètre de fonction, NE CONTIENT PAS le nom de la
#+ fonction.
#  Le nom de la fonction courante est accédé par la variable $NOM_FONCTION.
###
#  Une liste rapide et revue suit (rapide mais pas courte).

echo
echo '- - Test (mais sans changement) - -'
echo '- référence nulle -'
echo -n ${VarNulle-'NonInitialisée'}' '  # NonInitialisée
echo ${VarNulle}                         # NewLine only
echo -n ${VarNulle:-'NonInitialisée'}' ' # NonInitialisée
echo ${VarNulle}                         # Newline only

echo '- contenu nul -'
echo -n ${VarVide-'Vide'}' '             # Seulement l'espace
echo ${VarVide}                          # Nouvelle ligne seulement
echo -n ${VarVide:-'Vide'}' '            # Vide
echo ${VarVide}                          # Nouvelle ligne seulement

echo '- contenu -'
echo ${VarQuelquechose-'Contenu'}        # Littéral
echo ${VarQuelquechose:-'Contenu'}       # Littéral

echo '- Tableau à indice non continu -'
echo ${VarTableau[@]-'non initialisée'}

# Moment ASCII-Art
# État               O==oui, N==non
#                    -       :-
# Non initialisé     O       O       ${# ... } == 0
# Vide               N       O       ${# ... } == 0
# Contenu            N       N       ${# ... } > 0

#  Soit la première partie des tests soit la seconde pourrait être une chaîne
#+ d'appel d'une commande ou d'une fonction.
echo
echo '- - Test 1 pour indéfini - -'
declare -i t
```

```
_decT() {
    t=$t-1
}

# Référence nulle, initialisez à t == -1
t=${#VarNulle}                          # Résultats en zéro.
${VarNulle- _decT }                     # La fonction s'exécute, t vaut maintenant -1.
echo $t

# Contenu nul, initialisez à t == 0
t=${#VarVide}                          # Résultats en zéro.
${VarVide- _decT }                     # Fontion _decT NON exécutée.
echo $t

# Contenu, initialisez à t == nombre de caractères non nuls
VarQuelquechose='_simple'                  # Initialisez avec un nom de fonction
valide.
t=${#VarQuelquechose}                      # longueur différente de zéro
${VarQuelquechose- _decT }                 # Fonction _simple exécutée.
echo $t                                    # Notez l'action Append-To.

# Exercice : nettoyez cet exemple.
unset t
unset _decT
VarQuelquechose=Literal

echo
echo '- - Test et modification - -'
echo '- Affectation si référence nulle -'
echo -n ${VarNulle='NonInitialisée'}' '          # NonInitialisée NonInitialisée
echo ${VarNulle}
unset VarNulle

echo '- Affectation si référence nulle -'
echo -n ${VarNulle:='NonInitialisée'}' '         # NonInitialisée NonInitialisée
echo ${VarNulle}
unset VarNulle

echo "- Pas d'affectation si contenu nul -"
echo -n ${VarVide='Vide'}' '          # Espace seulement
echo ${VarVide}
VarVide=''

echo "- Affectation si contenu nul -"
echo -n ${VarVide:='Vide'}' '         # Vide Vide
echo ${VarVide}
VarVide=''

echo "- Aucun changement s'il a déjà un contenu -"
echo ${VarQuelquechose='Contenu'}          # Littéral
echo ${VarQuelquechose:='Contenu'}         # Littéral


# Tableaux Bash à indice non continu
###
#  Les tableaux Bash ont des indices continus, commençant à zéro
#+  sauf indication contraire.
###
#  L'initialisation de VarTableau était une façon de le "faire autrement".
#+ Voici un autre moyen :
###
echo
declare -a TableauNonContinu
TableauNonContinu=( [1]=un [2]='' [4]='quatre' )
# [0]=référence nulle, [2]=contenu nul, [3]=référence nulle

echo '- - Liste de tableaux à indice non continu - -'
# À l'intérieur de guillemets doubles, IFS par défaut, modèle global
```

```
IFS=$'\x20'$'\x09'$'\x0A'
printf %q "${TableauNonContinu[*]}"
echo

#  Notez que l'affichage ne distingue pas entre "contenu nul" et "référence nulle".
#  Les deux s'affichent comme des espaces blancs échappés.
###
#  Notez aussi que la sortie ne contient PAS d'espace blanc échappé
#+ pour le(s) "référence(s) nulle(s)" avant le premier élément défini.
###
# Ce comportement des versions 2.04, 2.05a et 2.05b a été rapporté et
#+ pourrait changer dans une prochaine version de Bash.

#  Pour afficher un tableau sans indice continu et maintenir la relation
#+ [indice]=valeur sans changement requiert un peu de programmation.
#  Un bout de code possible :
###
# local l=${#TableauNonContinu[@]}  # Nombre d'éléments définis
# local f=0                         # Nombre d'indices trouvés
# local i=0                         # Indice à tester
(                                   # Fonction anonyme en ligne
    for (( l=${#TableauNonContinu[@]}, f = 0, i = 0 ; f < l ; i++ ))
    do
        # 'si défini alors...'
        ${TableauNonContinu[$i]+ eval echo '\ ['$i']='${TableauNonContinu[$i]} ; (( f++
)) }
    done
)

# Le lecteur arrivant au fragment de code ci-dessus pourrait vouloir voir
#+ la liste des commandes et les commandes multiples sur une ligne dans le texte
#+ du guide de l'écriture avancée de scripts shell Bash.
###
#  Note :
#  La version "read -a nom_tableau" de la commande "read" commence à remplir
#+ nom_tableau à l'indice zéro.
#  TableauNonContinu ne définit pas de valeur à l'indice zéro.
###
#  L'utilisateur ayant besoin de lire/écrire un tableau non contigu pour soit
#+ un stockage externe soit une communication par socket doit inventer une paire
#+ de code lecture/écriture convenant à ce but.
###
# Exercice : nettoyez-le.

unset TableauNonContinu

echo
echo '- - Alternative conditionnel (mais sans changement)- -'
echo "- Pas d'alternative si référence nulle -"
echo -n ${VarNulle+'NonInitialisee'}' '
echo ${VarNulle}
unset VarNulle

echo "- Pas d'alternative si référence nulle -"
echo -n ${VarNulle:+'NonInitialisee'}' '
echo ${VarNulle}
unset VarNulle

echo "- Alternative si contenu nul -"
echo -n ${VarVide+'Vide'}' '              # Vide
echo ${VarVide}
VarVide=''

echo "- Pas d'alternative si contenu nul -"
echo -n ${VarVide:+'Vide'}' '             # Espace seul
echo ${VarVide}
VarVide=''

echo "- Alternative si contenu déjà existant -"
```

```
# Alternative littérale
echo -n ${VarQuelquechose+'Contenu'}' '        # Contenu littéral
echo ${VarQuelquechose}

# Appelle une fonction
echo -n ${VarQuelquechose:+ $(_simple) }' '    # Littéral FonctionSimple
echo ${VarQuelquechose}
echo

echo '- - Tableau non contigu - -'
echo ${VarTableau[@]+'Vide'}                   # Un tableau de 'vide'(s)
echo

echo '- - Test 2 pour indéfini - -'

declare -i t
_incT() {
    t=$t+1
}

#  Note:
#  C'est le même test utilisé dans le fragment de code
#+  pour le tableau non contigu.

# Référence nulle, initialisez : t == -1
t=${#VarNulle}-1                    # Les résultats dans moins-un.
${VarNulle+ _incT }                 # Ne s'exécute pas.
echo $t' Null reference'

# Contenu nul, initialisez : t == 0
t=${#VarVide}-1                    # Les résultats dans moins-un.
${VarVide+ _incT }                 # S'exécute.
echo $t'  Null content'

# Contenu, initialisez : t == (nombre de caractères non nuls)
t=${#VarQuelquechose}-1               # longueur non nul moins un
${VarQuelquechose+ _incT }            # S'exécute.
echo $t'  Contents'

# Exercice : nettoyez cet exemple.
unset t
unset _incT

# ${name?err_msg} ${name:?err_msg}
#  Ceci suit les mêmes règles mais quitte toujours après
#+ si une action est spécifiée après le point d'interrogation.
#  L'action suivant le point d'interrogation pourrait être un littéral
#+ ou le résultat d'une fonction.
###
#  ${nom?} ${nom:?} sont seulement des tests, le retour peut être testé.




# Opérations sur les éléments
# --------------------------

echo
echo '- - Sélection du sous-élément de queue - -'

#  Chaînes, tableaux et paramètres de position

#  Appeler ce script avec des arguments multiples
#+ pour voir les sélections du paramètre.

echo '- Tous -'
echo ${VarQuelquechose:0}           # tous les caractères non nuls
echo ${VarTableau[@]:0}             # tous les éléments avec contenu
```

```
echo ${@:0}                           # tous les paramètres avec contenu
                                      # ignore paramètre[0]

echo
echo '- Tous après -'
echo ${VarQuelquechose:1}             # tous les non nuls après caractère[0]
echo ${VarTableau[@]:1}               # tous après élément[0] avec contenu
echo ${@:2}                           # tous après param[1] avec contenu

echo
echo '- Intervalle après -'
echo ${VarQuelquechose:4:3}           # ral
                                      # trois caractères après
                                      # caractère[3]

echo '- Sparse array gotch -'
echo ${VarTableau[@]:1:2}   #  quatre - le premier élément avec contenu.
                            #  Deux éléments après (s'ils existent).
                            #  le PREMIER AVEC CONTENU
                            #+ (le PREMIER AVEC CONTENU doit être
                            #+ considéré comme s'il s'agissait de
                            #+ l'indice zéro).
#  Éxécuté comme si Bash considère SEULEMENT les éléments de tableau avec CONTENU
#  printf %q "${VarTableau[@]:0:3}"    # Essayez celle-ci

#  Dans les versions 2.04, 2.05a et 2.05b,
#+ Bash ne gère pas les tableaux non contigu comme attendu avec cette notation.
#
#  Le mainteneur actuel de Bash, Chet Ramey, a corrigé ceci.


echo '- Tableaux contigus -'
echo ${@:2:2}               # Deux paramètres suivant paramètre[1]

# Nouvelles victimes des exemples de vecteurs de chaînes :
chaineZ=abcABC123ABCabc
tableauZ=( abcabc ABCABC 123123 ABCABC abcabc )
noncontiguZ=( [1]='abcabc' [3]='ABCABC' [4]='' [5]='123123' )

echo
echo ' - - Chaîne victime - -'$chaineZ'- - '
echo ' - - Tableau victime - -'${tableauZ[@]}'- - '
echo ' - - Tableau non contigu - -'${noncontiguZ[@]}'- - '
echo ' - [0]==réf. nulle, [2]==réf. nulle, [4]==contenu nul - '
echo ' - [1]=abcabc [3]=ABCABC [5]=123123 - '
echo ' - nombre de références non nulles : '${#noncontiguZ[@]}' elements'

echo
echo "- - Suppression du préfixe d'un sous élément - -"
echo '- - la correspondance de modèle globale doit inclure le premier caractère. - -'
echo "- - Le modèle global doit être un littéral ou le résultat d'une fonction. - -"
echo


# Fonction renvoyant un modèle global simple, littéral
_abc() {
    echo -n 'abc'
}

echo '- Préfixe court -'
echo ${chaineZ#123}                   # Non modifié (pas un préfixe).
echo ${chaineZ#$(_abc)}               # ABC123ABCabc
echo ${tableauZ[@]#abc}               # Appliqué à chaque élément.

# echo ${noncontiguZ[@]#abc}            # Version-2.05b quitte avec un « core dump ».
# Corrigé depuis par Chet Ramey.

# Le -it serait sympa- Premier-Indice-De
# echo ${#noncontiguZ[@]#*}             # Ce n'est PAS du Bash valide.
```

```
echo
echo '- Préfixe le plus long -'
echo ${chaineZ##1*3}                # Non modifié (pas un préfixe)
echo ${chaineZ##a*C}                # abc
echo ${tableauZ[@]##a*c}            # ABCABC 123123 ABCABC

# echo ${noncontiguZ[@]##a*c}           # Version-2.05b quitte avec un « core dump ».
# Corrigé depuis par Chet Ramey.

echo
echo '- - Suppression du sous-élément suffixe - -'
echo '- - La correspondance du modèle global doit inclure le dernier caractère. - -'
echo '- - Le modèle global pourrait être un littéral ou un résultat de fonction. - -'
echo
echo '- Suffixe le plus court -'
echo ${chaineZ%1*3}                 # Non modifié (pas un suffixe).
echo ${chaineZ%$(_abc)}             # abcABC123ABC
echo ${tableauZ[@]%abc}             # Appliqué à chaque élément.

# echo ${noncontiguZ[@]%abc}            # Version-2.05b quitte avec un « core dump ».
# Corrigé depuis par Chet Ramey.

# Le -it serait sympa- Dernier-Indice-De
# echo ${#noncontiguZ[@]%*}             # Ce n'est PAS du Bash valide.

echo
echo '- Suffixe le plus long -'
echo ${chaineZ%%1*3}                # Non modifié (pas un suffixe)
echo ${chaineZ%%b*c}                # a
echo ${tableauZ[@]%%b*c}            # a ABCABC 123123 ABCABC a

# echo ${noncontiguZ[@]%%b*c}           # Version-2.05b quitte avec un « core dump ».
# Corrigé depuis par Chet Ramey.

echo
echo '- - Remplacement de sous-élements - -'
echo "- - Sous-élément situé n'importe où dans la chaîne. - -"
echo '- - La première spécification est un modèle global. - -'
echo '- - Le modèle global pourrait être un littéral ou un résultat de fonction de
modèle global. - -'
echo '- - La seconde spécification pourrait être un littéral ou un résultat de
fonction. - -'
echo '- - La seconde spécification pourrait être non spécifiée. Prononcez-ça comme :'
echo '    Remplace-Avec-Rien (Supprime) - -'
echo



# Fonction renvoyant un modèle global simple, littéral
_123() {
    echo -n '123'
}

echo '- Remplace la première occurrence -'
echo ${chaineZ/$(_123)/999}         # Modifié (123 est un composant).
echo ${chaineZ/ABC/xyz}             # xyzABC123ABCabc
echo ${tableauZ[@]/ABC/xyz}         # Appliqué à chaque élément.
echo ${noncontiguZ[@]/ABC/xyz}      # Fonctionne comme attendu.

echo
echo '- Supprime la première first occurrence -'
echo ${chaineZ/$(_123)/}
echo ${chaineZ/ABC/}
echo ${tableauZ[@]/ABC/}
echo ${noncontiguZ[@]/ABC/}

#  Le remplacement ne doit pas être un littéral,
#+ car le résultat de l'appel d'une fonction est permis.
```

```
#  C'est général pour toutes les formes de remplacement.
echo
echo '- Remplace la première occurence avec Résultat-De -'
echo ${chaineZ/$(_123)/$(_simple)}   # Fonctionne comme attendu.
echo ${tableauZ[@]/ca/$(_simple)}    # Appliqué à chaque élément.
echo ${noncontiguZ[@]/ca/$(_simple)} # Fonctionne comme attendu.

echo
echo '- Remplace toutes les occurrences -'
echo ${chaineZ//[b2]/X}              # X-out b et 2
echo ${chaineZ//abc/xyz}             # xyzABC123ABCxyz
echo ${tableauZ[@]//abc/xyz}         # Appliqué à chaque élément.
echo ${noncontiguZ[@]//abc/xyz}      # Fonctionne comme attendu.

echo
echo '- Supprime toutes les occurrences -'
echo ${chaineZ//[b2]/}
echo ${chaineZ//abc/}
echo ${tableauZ[@]//abc/}
echo ${noncontiguZ[@]//abc/}

echo
echo '- - Remplacement du sous-élément préfixe - -'
echo '- - La correspondance doit inclure le premier caractère. - -'
echo

echo '- Remplace les occurrences du préfixe -'
echo ${chaineZ/#[b2]/X}              # Non modifié (n'est pas non plus un préfixe).
echo ${chaineZ/#$(_abc)/XYZ}         # XYZABC123ABCabc
echo ${tableauZ[@]/#abc/XYZ}         # Appliqué à chaque élément.
echo ${noncontiguZ[@]/#abc/XYZ}      # Fonctionne comme attendu.

echo
echo '- Supprime les occurrences du préfixe -'
echo ${chaineZ/#[b2]/}
echo ${chaineZ/#$(_abc)/}
echo ${tableauZ[@]/#abc/}
echo ${noncontiguZ[@]/#abc/}

echo
echo '- - Remplacement du sous-élément suffixe - -'
echo '- - La correspondance doit inclure le dernier caractère. - -'
echo

echo '- Remplace les occurrences du suffixe -'
echo ${chaineZ/%[b2]/X}              # Non modifié (n'est pas non plus un suffixe).
echo ${chaineZ/%$(_abc)/XYZ}         # abcABC123ABCXYZ
echo ${tableauZ[@]/%abc/XYZ}         # Appliqué à chaque élément.
echo ${noncontiguZ[@]/%abc/XYZ}      # Fonctionne comme attendu.

echo
echo '- Supprime les occurrences du suffixe -'
echo ${chaineZ/%[b2]/}
echo ${chaineZ/%$(_abc)/}
echo ${tableauZ[@]/%abc/}
echo ${noncontiguZ[@]/%abc/}

echo
echo '- - Cas spéciaux du modèle global nul - -'
echo

echo '- Tout préfixe -'
# modèle de sous-chaîne nul, signifiant 'préfixe'
echo ${chaineZ/#/NEW}                # NEWabcABC123ABCabc
echo ${tableauZ[@]/#/NEW}            # Appliqué à chaque élément.
echo ${noncontiguZ[@]/#/NEW}         # Aussi appliqué au contenu nul.
                                     # Cela semble raisonnable.

echo
```

```
echo '- Tout suffixe -'
# modèle de sous-chaîne nul, signifiant 'suffixe'
echo ${chaineZ/%/NEW}               # abcABC123ABCabcNEW
echo ${tableauZ[@]/%/NEW}           # Appliqué à chaque élément.
echo ${noncontiguZ[@]/%/NEW}        # Aussi appliqué au contenu nul.
                                    # Cela semble raisonnable.

echo
echo '- - Cas spécial pour le modèle global For-Each - -'
echo '- - - - Ceci est un rêve - - - -'
echo

_GenFunc() {
    echo -n ${0}                    # Illustration seulement.
    # Actuellement, ce serait un calcul arbitraire.
}

#  Toutes les occurrences, correspondant au modèle NImporteQuoi.
#  Actuellement, //*/ n'établit pas une correspondance avec un modèle nul
#+ ainsi qu'avec une référence nulle.
#  /#/ et /%/ correspondent à un contenu nul mais pas à une référence nulle.
echo ${noncontiguZ[@]//*/$(_GenFunc)}


#  Une syntaxe possible placerait la notation du paramètre utilisé
#+ à l'intérieur du moyen de construction.
#   ${1} - L'élément complet
#   ${2} - Le préfixe, S'il existe, du sous-élément correspondant
#   ${3} - Le sous-élément correspondant
#   ${4} - Le suffixe, S'il existe, du sous-élément correspondant
#
# echo ${noncontiguZ[@]//*/$(_GenFunc ${3})}   # Pareil que ${1}, ici.
# Cela sera peut-être implémenté dans une future version de Bash.


exit 0
```

Annexe B. Cartes de référence

Les cartes de référence suivantes apportent un *résumé* utile de certains concepts dans l'écriture des scripts. Le texte suivant traite de ces sujets avec plus de profondeur et donne des exemples d'utilisation.

Tableau B.1. Variables spéciales du shell

Variable	Signification
`$0`	Nom du script
`$1`	Paramètre de position #1
`$2 - $9`	Paramètres de position #2 - #9
`${10}`	Paramètre de position #10
`$#`	Nombre de paramètres de position
`"$*"`	Tous les paramètres de position (en un seul mot) *
`"$@"`	Tous les paramètres de position (en des chaînes séparées)
`${#*}`	Nombre de paramètres sur la ligne de commande passés au script
`${#@}`	Nombre de paramètres sur la ligne de commande passés au script
`$?`	Code de retour
`$$`	Numéro d'identifiant du processus (PID) généré par le script
`$-`	Options passées au script (utilisant *set*)
`$_`	Dernier argument de la commande précédente
`$!`	Identifiant du processus (PID) du dernier job exécuté en tâche de fond

***** *Doit être entre guillemet*, sinon il vaudra par défaut « `$@` ».

Tableau B.2. Opérateurs de test : comparaison binaire

Opérateur	Signification	-----	Opérateur	Signification
Comparaison arithmétique			Comparaison de chaînes	
`-eq`	Égal à		`=`	Égal à
			`==`	Égal à
`-ne`	Différent de		`!=`	Différent de
`-lt`	Plus petit que		`\<`	Plus petit que (ASCII) *
`-le`	Plus petit que ou égal à			
`-gt`	Plus grand que		`\>`	Plus grand que (ASCII) *
`-ge`	Plus grand que ou égal à			
			`-z`	Chaîne vide
			`-n`	Chaîne non vide
Comparaison arithmétique	Entre des parenthèses doubles ((...))			
`>`	Plus grand que			
`>=`	Plus grand que ou égal à			

Opérateur	Signification	-----	Opérateur	Signification
<	Plus petit que			
<=	Plus petit que ou égal à			

***** *Si à l'intérieur d'une construction de tests à double crochets* [[...]] *, alors l'échappement \ n'est pas nécessaire*.

Tableau B.3. Opérateurs de test : fichiers

Opérateur	Tests si	-----	Opérateur	Tests si
`-e`	Le fichier existe		`-s`	Le fichier est vide
`-f`	Le fichier est un fichier *standard*			
`-d`	Le fichier est un *répertoire*		`-r`	Le fichier a un droit de *lecture*
`-h`	Le fichier est un lien symbolique194		`-w`	Le fichier a un droit en *écriture*
`-L`	Le fichier est un *lien symbolique*		`-x`	Le fichier a le droit d'*exécution*
`-b`	Le fichier est un périphérique bloc53			
`-c`	Le fichier est un périphérique caractère53		`-g`	L'option *sgid* est positionnée
`-p`	Le fichier est un tube		`-u`	L'option *suid* est positionnée
`-S`	Le fichier est un socket399		`-k`	L'option « sticky bit » est positionnée
`-t`	Le fichier est associé à un *terminal*			
`-N`	Le fichier a été modifié depuis sa dernière lecture		`F1 -nt F2`	Le fichier F1 est *plus récent* que F2 *
`-O`	Vous êtes le propriétaire du fichier		`F1 -ot F2`	Le fichier F1 est *plus ancien* que F2 *
`-G`	L'*identifiant du groupe* du fichier est le même que vous		`F1 -ef F2`	Les fichiers F1 et F2 sont des *liens* vers le même fichier *
`!`	NOT (inverse le résultat des tests ci-dessus)			

***** Opérateur *binaire* (nécessite deux opérandes).

Tableau B.4. Substitution et expansion de paramètres

Expression	Signification
`${var}`	Valeur de *`var`*, identique à *`$var`*
`${var-DEFAUT}`	Si *`var`* n'est pas initialisé, évalue l'expression *`$DEFAUT`* *
`${var:-DEFAUT}`	Si *`var`* n'est pas initialisé ou est vide, évalue l'expression *`$DEFAUT`* *

Expression	Signification
`${var=DEFAUT}`	Si *`var`* n'est pas initialisé, évalue l'expression *`$DEFAUT`* *
`${var:=DEFAUT}`	Si *`var`* n'est pas initialisé, évalue l'expression *`$DEFAUT`* *
`${var+AUTRE}`	Si *`var`* est initialisé, évalue l'expression *`$AUTRE`*, sinon est une chaîne null
`${var:+AUTRE}`	Si *`var`* est initialisé, évalue l'expression *`$AUTRE`*, sinon est une chaîne null
`${var?ERR_MSG}`	Si *`var`* n'est pas initialisé, affiche *`$ERR_MSG`* *
`${var:?ERR_MSG}`	Si *`var`* n'est pas initialisé, affiche *`$ERR_MSG`* *
`${!varprefix*}`	Correspond à toutes les variables déclarées précédemment et commençant par *`varprefix`*
`${!varprefix@}`	Correspond à toutes les variables déclarées précédemment et commençant par *`varprefix`*

***** Bien sûr, si *`var`* *est* initialisé, évalue l'expression comme *`$var`*.

Tableau B.5. Opérations sur les chaînes

Expression	Signification
`${#chaine}`	Longueur de *`$chaine`*
`${chaine:position}`	Extrait la sous-chaîne à partir de *`$chaine`* jusqu'à *`$position`*
`${chaine:position:longueur}`	Extrait *`$longueur`* caractères dans la sous-chaîne à partir de *`$chaine`* jusqu'à *`$position`*
`${chaine#sous-chaine}`	Supprime la plus petite correspondance de *`$sous-chaine`* à partir du début de *`$chaine`*
`${chaine##sous-chaine}`	Supprime la plus grande correspondance de *`$sous-chaine`* à partir du début de *`$chaine`*
`${chaine%sous-chaine}`	Supprime la plus courte correspondance de *`$sous-chaine`* à partir de la fin de *`$chaine`*
`${chaine%%sous-chaine}`	Supprime la plus longue correspondance de *`$sous-chaine`* à partir de la fin de *`$chaine`*
`${chaine/sous-chaine/remplacement}`	Remplace la première correspondance de *`$sous-chaine`* avec *`$remplacement`*
`${chaine//sous-chaine/remplacement}`	Remplace *toutes* les correspondances de *`$sous-chaine`* avec *`$remplacement`*
`${chaine/#sous-chaine/remplacement}`	Si *`$sous-chaine`* correspond au *début* de *`$chaine`*, substitue *`$remplacement`* par *`$sous-chaine`*
`${chaine/%sous-chaine/remplacement}`	Si *`$sous-chaine`* correspond à la *fin* de *`$chaine`*, substitue *`$remplacement`* par *`$sous-chaine`*
`expr match "$chaine" '$sous-chaine'`	Longueur de *`$sous-chaine`** correspondant au début de *`$chaine`*
`expr "$chaine" : '$sous-chaine'`	Longueur de *`$sous-chaine`** correspondant au début de

Expression	Signification
	$chaine
`expr index "$chaine" $sous-chaine`	Position numérique dans *$chaine* du premier caractère correspondant dans *$sous-chaine*
`expr substr $chaine $position $longueur`	Extrait *$longueur* caractères à partir de *$chaine* commençant à *$position*
`expr match "$chaine" '\($sous-chaine\)'`	Extrait *$sous-chaine** au début de *$chaine*
`expr "$chaine" : '\($sous-chaine\)'`	Extrait *$sous-chaine** au début de *$chaine*
`expr match "$chaine" '.*\($sous-chaine\)'`	Extrait *$sous-chaine** à la fin de *$chaine*
`expr "$chaine" : '.*\($sous-chaine\)'`	Extrait *$sous-chaine** à la fin de *$chaine*

***** Où *$sous-chaine* est une expression rationnelle313.

Tableau B.6. Constructions diverses

Expression	Interprétation
Crochets313	
`if [ CONDITION ]`	Construction de tests
`if [[ CONDITION ]]`	Construction de tests étendue50
`Tableau[1]=element1`	Initialisation d'un tableau375
`[a-z]`	Ensemble de caractères se suivant à l'intérieur d'une expression rationnelle313
Accolades	
`${variable}`	Substitution de paramètres
`${!variable}`	Référence de variable indirecte
`{ commande1; commande2; ... commandeN; }`	Bloc de code
`{chaine1,string2,string3,...}`	Expansion
`{a..z}`	Expansion d'accolades469
`{}`	Remplacement de texte, après find195 et xargs199
Parenthèses	
`( commande1; commande2 )`	Groupe de commandes exécutées dans un sous-shell342
`Tableau=(element1 element2 element3)`	Initialisation d'un tableau
`result=$(COMMANDE)`	Exécute la commande dans un sous-shell et affecte le résultat dans une variable
`>(COMMANDE)`	Substitution de processus349
`<(COMMANDE)`	Substitution de processus
Double parenthèses	
`(( var = 78 ))`	Arithmétique entière
`var=$(( 20 + 5 ))`	Arithmétique entière, avec affectation de variables
`(( var++ ))`	Incrément de variables118 *style C*
`(( var-- ))`	Incrément de variables118 *style C*
`(( var0 = var1<98?9:21 ))`	Opération à trois arguments14

Expression	**Interprétation**
Guillemets37	
`"$variable"`	Guillemets « faibles »
`'chaine'`	Guillemets « forts »
Guillemets inversés141	
`result=`COMMANDE``	Exécute la commande dans un sous-shell et affecte le résultat à une variable

Annexe C. Petit guide sur Sed et Awk

Ceci est une brève introduction aux utilitaires de traitement de texte **sed** et **awk**. Nous allons voir seulement quelques commandes basiques ici, mais cela devrait suffire pour comprendre des constructions sed et awk simples à l'intérieur de scripts shell.

sed : un éditeur de fichiers texte non interactif

awk : un langage d'examen et de traitement de motifs orienté champs avec une syntaxe C

Malgré toutes leurs différences, les deux utilitaires partagent une syntaxe d'appel similaire, utilisent tous les deux les expressions rationnelles313, lisent tous les deux l'entrée à partir de `stdin` par défaut, et envoient leur sortie sur `stdout`. Ce sont de bons outils UNIX et ils travaillent bien ensemble. La sortie de l'un peut être envoyée via un tube vers l'autre, et leurs capacités combinées donnent aux scripts shell un peu de la puissance de Perl.

Note

Une différence importante entre ces utilitaires est que si les scripts shell peuvent passer des arguments facilement à sed, c'est plus compliqué avec awk (voir l'Exemple 33.5, « Un *script d'appel* autour d'un autre script awk » et l'Exemple 9.25, « Passer une référence indirecte à *awk* »).

C.1. Sed

Sed est un éditeur ligne non interactif [1] . Il reçoit du texte en entrée, que ce soit à partir de `stdin` ou d'un fichier, réalise certaines opérations sur les lignes spécifiées de l'entrée, une ligne à la fois, puis sort le résultat vers `stdout` ou vers un fichier. À l'intérieur d'un script shell, sed est habituellement un des différents outils composant un tube.

Sed détermine sur quelles lignes de son entrée il va opérer à partir d'*une plage d'adresses* qui lui aura été fourni. [2] Cette plage d'adresses est définie soit par des numéros de ligne soit par un motif à rechercher. Par exemple, *`3d`* indique à sed qu'il doit supprimer la ligne 3 de l'entrée et *`/windows/d`* dit à sed que vous voulez que toutes les lignes de l'entrée contenant « windows » soient supprimées.

De toutes les opérations de la boîte à outils sed, nous nous occuperons principalement des trois les plus communément utilisées. Il s'agit de **p**rinting (NdT : affichage vers `stdout`), **d**eletion (NdT : suppression) et **s**ubstitution (NdT : euh... substitution :).

Tableau C.1. Opérateurs sed basiques

Opérateur	Nom	Effet
`[plage-d-adresses]/p`	print	Affiche la [plage d'adresse] spécifiée
`[plage-d-adresses]/d`	delete	Supprime la [plage d'adresse] spécifiée
`s/motif1/motif2/`	substitute	Substitue motif2 à la première instance de motif1 sur une ligne
`[plage-d-adresses]/s/motif1/motif2/`	substitute	Substitue motif2 à la première instance de motif1 sur une ligne, en restant dans la *`plage-d-adresses`*
`[plage-d-adresses]/y/motif1/motif2/`	transform	remplace tout caractère de motif1 avec le caractère correspondant dans motif2, en restant dans la *`plage-d-adresses`* (équivalent de **tr**)
`g`	global	Opère sur *chaque* correspondance du motif à l'intérieur de chaque ligne d'entrée de la plage d'adresses concernée

Note

La substitution opère seulement sur la première instance de la correspondance d'un motif à l'intérieur de chaque ligne, sauf si l'opérateur `g` (*global*) est ajouté à la commande *substitute*.

[1] Sed s'exécute sans l'intervention de l'utilisateur.
[2] Si aucune plage d'adresses n'est spécifiée, il s'agit par défaut de *toutes* les lignes.

À partir de la ligne de commande et dans un script shell, une opération sed peut nécessiter de mettre entre guillemets et d'utiliser certaines options.

```
sed -e '/^$/d' $nomfichier
#  L'option -e fait que la chaîne de caractère suivante est interprétée comme
#+ une instruction d'édition.
#  (Si vous passez une seule instruction à sed, le "-e" est optionnel.)
#  Les guillemets "forts" ('') empêchent les caractères de l'ER compris dans
#+ l'instruction d'être interprétés comme des caractères spéciaux par le corps
#+ du script.
# (Ceci réserve l'expansion de l'ER de l'instruction à sed.)
#
# Opère sur le texte contenu dans le fichier $nomfichier.
```

Dans certains cas, une commande d'édition *sed* ne fonctionnera pas avec des guillemets simples.

```
nomfichier=fichier1.txt
motif=BEGIN

  sed "/^$motif/d" "$nomfichier"  # fonctionne comme indiqué
# sed '/^$motif/d' "$nomfichier"    a des résultats inattendus.
#        Dans cette instance, avec des guillemets forts (' ... '),
#+      "$motif" ne sera pas étendu en "BEGIN".
```

Note

sed utilise l'option `-e` pour spécifier que la chaîne suivante est une instruction ou un ensemble d'instructions. Si la chaîne ne contient qu'une seule instruction, alors cette option peut être omise.

```
sed -n '/xzy/p' $nomfichier
#  L'option -n indique à sed d'afficher seulement les lignes correspondant au
#+ motif.
# Sinon toutes les lignes en entrée s'afficheront.
# L'option -e est inutile ici car il y a une seule instruction d'édition.
```

Tableau C.2. Exemples d'opérateurs sed

Notation	Effet
`8d`	Supprime la 8è ligne de l'entrée.
`/^$/d`	Supprime toutes les lignes vides.
`1,/^$/d`	Supprime les lignes du début à la première ligne vide (inclue).
`/Jones/p`	Affiche seulement les lignes contenant « Jones » (avec l'option -n).
`s/Windows/Linux/`	Substitue « Linux » à chaque première instance de « Windows » trouvée dans chaque ligne d'entrée.
`s/BSOD/stability/g`	Substitue « stability » à chaque instance de « BSOD » trouvée dans chaque ligne d'entrée.
`s/ *$//`	Supprime tous les espaces à la fin de toutes les lignes.
`s/00*/0/g`	Compresse toutes les séquences consécutives de zéros en un seul zéro.
`/GUI/d`	Supprime toutes les lignes contenant « GUI ».
`s/GUI//g`	Supprime toutes les instances de « GUI », en laissant le reste de la ligne intact.

Substituer une chaîne vide (taille zéro) à une autre est équivalent à supprimer cette chaîne dans une ligne de l'entrée. Le reste de la ligne reste intact. Appliquer **`s/GUI//`** à la ligne

```
The most important parts of any application are its GUI and sound effects
```

donne

```
The most important parts of any application are its  and sound effects
```

L'antislash force la continuité sur la ligne suivante pour la commande **sed**. Ceci a pour effet d'utiliser la *nouvelle ligne* comme fin de ligne de la *chaîne de remplacement*.

```
s/^  */\
/g
```

Cette substitution remplace les espaces débutant les lignes par un retour chariot. Le résultat final est le remplacement des indentations de paragraphes par une ligne vide entre les paragraphes.

Une plage d'adresses suivie par une ou plusieurs opérations peut nécessiter des accolades ouvrantes et fermantes avec les retours chariot appropriés.

```
/[0-9A-Za-z]/,/^$/{
/^$/d
}
```

Ceci supprime seulement la première de chaque ensemble de lignes vides consécutives. Ceci peut être utile pour espacer de manière égale un fichier texte mais en conservant les lignes vides entre paragraphes.

Note

Le délimiteur habituellement utilisé par *sed* est /. Néanmoins, *sed* autorise d'autres délimiteurs, comme %. C'est utile quand / fait partie d'une chaîne de remplacement, comme dans le chemin d'un fichier. Voir Exemple 10.9, « Rechercher les auteurs de tous les binaires d'un répertoire » et Exemple 15.32, « Supprimer les commentaires des programmes C ».

Astuce

Une façon rapide de doubler les espaces dans un fichier texte est **`sed G nomfichier`**.

Pour des exemples illustrant l'usage de sed à l'intérieur de scripts shell, jetez un oeil sur les exemples suivants :

1. Exemple 33.1, « *Script d'appel* »
2. Exemple 33.2, « Un *script d'appel* légèrement plus complexe »
3. Exemple 15.3, « *Badname* élimine dans le répertoire courant les fichiers dont le nom contient des caractères incorrects et des espaces blancs. »
4. Exemple A.2, « *rn* : Un utilitaire simple pour renommer des fichiers »
5. Exemple 15.17, « Émuler *grep* dans un script »
6. Exemple 15.27, « Utiliser *column* pour formater l'affichage des répertoires »
7. Exemple A.12, « *behead*: Supprimer les en-têtes des courriers électroniques et des nouvelles »
8. Exemple A.17, « *tree*: Afficher l'arborescence d'un répertoire »
9. Exemple A.18, « *tree2* : autre script tree »
10. Exemple 15.32, « Supprimer les commentaires des programmes C »
11. Exemple 10.9, « Rechercher les auteurs de tous les binaires d'un répertoire »
12. Exemple 15.47, « Conversion de base »
13. Exemple A.1, « *mailformat* : Formater un courrier électronique »
14. Exemple 15.14, « Générer des nombres aléatoires de dix chiffres »
15. Exemple 15.12, « Analyse de fréquence d'apparition des mots »
16. Exemple A.10, « « *life* : Jeu de la Vie » »

.

. Exemple 18.12, « Un script auto-documenté »

18 Exemple 15.19, « Rechercher des définitions dans *le dictionnaire Webster de 1913* »

19 Exemple A.31, « Chasse aux spammeurs »

20 Exemple A.26, « Convertir en HTML »

.

Pour un traitement plus en profondeur de *sed*, vérifiez les références appropriées dans la Bibliographie.

C.2. Awk

Awk est un langage de manipulation de texte plein de fonctionnalités avec une syntaxe proche du *C*. Alors qu'il possède un ensemble impressionnant d'opérateurs et de fonctionnalités, nous n'en couvrirons que quelques-uns, les plus utiles pour l'écriture de scripts shell.

Awk casse chaque ligne d'entrée en *champs*. Par défaut, un champ est une chaîne de caractères consécutifs délimités par des espaces, bien qu'il existe des options pour changer le délimiteur. Awk analyse et opère sur chaque champ. Ceci le rend idéal pour gérer des fichiers texte structurés, particulièrement des tableaux, des données organisées en ensembles cohérents tels que des lignes et des colonnes.

Desguillemets forts (guillemets simples) et des accolades entourent les segments de code awk dans un script shell.

```
# $1 est le premier champ, $2 est le second champ, etc.

echo un deux | awk '{print $1}'
# un

echo un deux | awk '{print $2}'
# deux

# Mais quel est le champ #0 ($0)?
echo un deux | awk '{print $0}'
# one two
# Tous les champs !


awk '{print $3}' $nomfichier
# Affiche le champ 3 du fichier $nomfichier sur stdout.

awk '{print $1 $5 $6}' $nomfichier
# Affiche les champs 1, 5 et 6 du fichier $nomfichier.

awk '{print $0}' $nomfichier
# Affiche le fichier entier the entire file!
# Même effet que :   cat $nomfichier . . . or . . . sed '' $nomfichier
```

Nous venons juste de voir la commande awk *print* en action. Les seules autres fonctionnalités de awk que nous avons besoin de gérer ici sont des variables. Awk gère les variables de façon similaire aux scripts shell, quoiqu'avec un peu plus de flexibilité.

```
{ total += ${numero_colonne} }
```

Ceci ajoute la valeur de *numero_colonne* au total *total*. Finalement, pour afficher « total », il existe un bloc de commandes **END**, exécuté après que le script ait opéré sur toute son entrée.

```
END { print total }
```

Correspondant au **END**, il existe **BEGIN** pour un bloc de code à exécuter avant que awk ne commence son travail sur son entrée.

L'exemple suivant illustre comment **awk** ajoute des outils d'analyse de texte dans un script shell.

Exemple C.1. Compteur sur le nombre d'occurences des lettres

```
#! /bin/sh
# letter-count2.sh : Compter les occurrences des lettres d'un fichier texte.
#
# Script de nyal [nyal@voila.fr].
# Utilisé dans le guide ABS avec sa permission.
```

```
# Nouveaux commentaires et reformatage par l'auteur du guide ABS.
# Version 1.1 : Modifié pour fonctionner avec gawk 3.1.3.
#               (Fonctionnera toujours avec les anciennes versions)


INIT_TAB_AWK=""
# Paramètre pour initialiser le script awk.
compteur=0
FICHIER_A_ANALYSER=$1

E_PARAMERR=65

usage()
{
    echo "Usage : letter-count.sh fichier lettres" 2>&1
    # Par exemple :   ./letter-count2.sh nom_fichier a b c
    exit $E_PARAMERR  # Pas assez d'arguments passés au script.
}

if [ ! -f "$1" ] ; then
    echo "$1 : Fichier inconnu." 2>&1
    usage      # Affiche le message d'usage et quitte.
fi

if [ -z "$2" ] ; then
    echo "$2 : Aucune lettre spécifiée." 2>&1
    usage
fi

shift                       # Lettres spécifiées.
for lettre in `echo $@`     # Pour chacune...
  do
  INIT_TAB_AWK="$INIT_TAB_AWK tableau_recherche[${compteur}] = \
  \"$lettre\"; tableau_final[${compteur}] = 0; "
  # A passer comme paramètres au script awk ci-dessous.
  compteur=`expr $compteur + 1`
done

# DEBUG :
# echo $INIT_TAB_AWK;

cat $FICHIER_A_ANALYSER |
# Envoyer le fichier cible au script awk suivant.

#
-------------------------------------------------------------------------------------------------
# L'ancienne version du script utilisait
# awk -v tableau_recherche=0 -v tableau_final=0 -v tab=0 -v \
# nb_letter=0 -v chara=0 -v caractere2=0 \

awk \
"BEGIN { $INIT_TAB_AWK } \
{ split(\$0, tab, \"\"); \
for (caractere in tab) \
{ for (caractere2 in tableau_recherche) \
{ if (tableau_recherche[caractere2] == tab[caractere]) { tableau_final[caractere2]++ }
} } } \
END { for (caractere in tableau_final) \
{ print tableau_recherche[caractere] \" => \" tableau_final[caractere] } }"
# ------------------------------------------------------------------------------
#  Rien de très compliqué, seulement...
#+ boucles for, tests if et quelques fonctions spécialisées.

exit $?

# Comparez ce script à letter-count.sh.
```

Pour des exemples simples de **awk** à l'intérieur de scripts shell, jetez un oeil sur :

1. Exemple 14.13, « Forcer une déconnexion »
2. Exemple 19.8, « Boucle *for* redirigée »
3. Exemple 15.32, « Supprimer les commentaires des programmes C »
4. Exemple 33.5, « Un *script d'appel* autour d'un autre script awk »
5. Exemple 9.25, « Passer une référence indirecte à *awk* »
6. Exemple 14.20, « Utiliser *export* pour passer une variable à un script *awk* embarqué »
7. Exemple 27.2, « Trouver le processus associé à un PID »
8. Exemple 27.3, « État de la connexion »
9. Exemple 10.3, « *Fileinfo* : opérer sur une liste de fichiers contenue dans une variable »
10 Exemple 15.59, « Effacer les fichiers de façon sûre »
11 Exemple 9.31, « Réinitialiser RANDOM »
12 Exemple 15.4, « Effacer un fichier par son numéro d'*inode* »
13 Exemple 9.15, « Autres moyens d'extraire des sous-chaînes »
14 Exemple 33.16, « Une astuce permettant de renvoyer plus d'une valeur de retour »
15 Exemple 10.8, « Afficher tous les utilisateurs du système »
16 Exemple 33.4, « Un *script d'appel* autour d'un script awk »
17 Exemple 15.52, « Calculer l'hypoténuse d'un triangle »

.

C'est tout ce que nous allons voir sur awk mais il existe bien plus à apprendre. Voyez les références appropriées dans la Bibliographie.

Annexe D. Codes de sortie ayant une signification particulière

Tableau D.1. Codes de sortie *réservés*

Code de sortie	Signification	Exemple	Commentaires
1	standard pour les erreurs générales	let "var1 = 1/0"	erreurs diverses, comme une « division par zéro » et autres opérations interdites
2	mauvaise utilisation de commandes intégrées, d'après la documentation de Bash		Rarement vue, généralement utilisation du code de sortie 1
126	la commande appelée ne peut s'exécuter		problème de droits ou commande non exécutable
127	« commande introuvable »		problème possible avec $PATH ou erreur de frappe
128	argument invalide pour exit44	exit 3.14159	**exit** prend seulement des arguments de type entier compris entre 0 et 255 (voir la première note de bas de page)
128+n	signal « n » d'erreur fatale	*kill -9* $PPID d'un script	**$?** renvoie 137 (128 + 9)
130	script terminé avec Control-C		Control-C est le signal 2 d'erreur fatale (130 = 128 + 2, voir ci-dessus)
255*	code de sortie en dehors de la limite	exit -1	**exit** prend seulement des arguments de type entier compris entre 0 et 255

D'après la table ci-dessus, les codes de sortie 1 - 2, 126 - 165, et 255 [1] ont une signification particulière et devraient donc être évités pour les paramètres de sortie définis par l'utilisateur. Finir un script avec *exit 127* va certainement causer une certaine confusion lors du débogage (est-ce que le code d'erreur est « commande introuvable » ou une erreur définie par l'utilisateur ?). Néanmoins, beaucoup de scripts utilisent un *exit 1* comme code de sortie générique en cas d'erreur. Le code de sortie exit 1 est utilisé dans tellement de cas d'erreur que cela ne sera pas très utile pour le débogage.

Il y a eu un essai de normalisation des codes de sortie (voir `/usr/include/sysexits.h`) mais il avait pour cible les programmeurs C et C++. Un standard similaire pour la programmation de script pourrait être approprié. L'auteur de ce document propose de restreindre les codes de sortie définis par l'utilisateur à l'intervalle 64 - 113 (en plus du code 0 en cas de succès) pour se conformer au standard C/C++. Ceci permettrait 50 codes valides et faciliterait le débogage des scripts. [2] Tous les codes de sortie définis par l'utilisateur dans les exemples accompagnant ce document se conforment maintenant à ce standard, sauf dans les cas de redéfinition comme dans l'Exemple 9.2, « Saisie avec délai ».

Note

Lancer un $? à partir de la ligne de commande après un script shell donne des résultats cohérents avec la table ci-dessus seulement à partir de l'invite Bash ou *sh*. L'utilisation de cette commande dans un shell *C* ou *tcsh* peut donner d'autres valeurs dans certains cas.

[1] Des valeurs de sortie en dehors de limites peuvent donner des codes de sortie inattendus. Un code de sortie plus grand que 255 renvoie ce code modulo 256. Par exemple, *exit 3809* donne un code de sortie 225 (3809 % 256 = 225).

[2] Une mise à jour de `/usr/include/sysexits.h` alloue des codes de sortie auparavant inutilisés, de 64 à 78. Il semble que l'ensemble des codes de sortie pourront être encore plus restreints dans le futur. L'auteur de ce document ne fera *pas* de corrections sur les exemples de scripts pour se conformer au standard changeant. Ceci ne doit pas causer de problème car il n'y a pas de surcharge ou de conflit dans les codes de sorties utilisés entre des binaires C/C++ compilés et les scripts shell.

Annexe E. Une introduction détaillée sur les redirections d'entrées/sorties

Écrit par Stéphane Chazelas et relu par l'auteur du document

Une commande s'attend à ce que les trois premiers descripteurs de fichier330 (fd) soient disponibles. Le premier, *fd 0* (l'entrée standard, `stdin`), concerne la lecture. Les deux autres (*fd 1*, `stdout` et *fd 2*, `stderr`) concernent l'écriture.

Il existe un `stdin`, `stdout` et un `stderr` associés à chaque commande. **`ls 2>&1`** connecte temporairement le `stderr` de la commande **ls** à la même « ressource » que le `stdout` du shell.

Par convention, une commande lit l'entrée à partir de fd 0 (`stdin`), affiche sur la sortie normale, fd 1 (`stdout`) et sur la sortie des erreurs, fd 2 (`stderr`). Si un des trois fd n'est pas ouvert, vous pouvez rencontrer des problèmes:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

Par exemple, lorsque **xterm** est lancé, il commence par s'initialiser soi-même. Avant de lancer le shell de l'utilisateur, **xterm** ouvre le périphérique du terminal (/dev/pts/<n> ou quelque chose de similaire) trois fois.

À ce moment, Bash hérite de ces trois descripteurs de fichiers et chaque commande (processus fils) lancée par Bash en hérite à leur tour sauf quand vous redirigez la commande. La redirection signifie la réaffectation d'un des descripteurs de fichier à un autre fichier (ou tube, ou tout autre chose permise). Les descripteurs de fichiers peuvent être réaffectés (pour une commande, un groupe de commande, un sous-shell342, une boucle while ou if ou case ou for336...) ou, globalement, pour le reste du script (en utilisant exec).

`ls > /dev/null` lance **ls** avec fd 1 connecté à `/dev/null`.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1


bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    426 root    1w  FIFO    0,0      7520 pipe
 lsof    426 root    2w  FIFO    0,0      7520 pipe
```

Ceci fonctionne avec différents types de redirection.

`Exercice :` Analyser le script suivant.

```
#! /usr/bin/env bash
```

```
mkfifo /tmp/fifo1 /tmp/fifo2
while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 & exec 7> /tmp/fifo1
exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)

exec 3>&1
(
 (
  (
   while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr | \
        tee /dev/fd/4 | tee /dev/fd/5 | tee /dev/fd/6 >&7 &
   exec 3> /tmp/fifo2

   echo 1st, to stdout
   sleep 1
   echo 2nd, to stderr >&2
   sleep 1
   echo 3rd, to fd 3 >&3
   sleep 1
   echo 4th, to fd 4 >&4
   sleep 1
   echo 5th, to fd 5 >&5
   sleep 1
   echo 6th, through a pipe | sed 's/.*/PIPE: &, to fd 5/' >&5
   sleep 1
   echo 7th, to fd 6 >&6
   sleep 1
   echo 8th, to fd 7 >&7
   sleep 1
   echo 9th, to fd 8 >&8

  ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
 ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-

rm -f /tmp/fifo1 /tmp/fifo2


# Pour chaque commande et sous-shell, cherchez vers quoi est lié chaque fd.
# Bonne chance.

exit 0
```

Annexe F. Options en ligne de commande

Un grand nombre d'exécutables, qu'ils soient binaires ou scripts, acceptent des options pour modifier leur comportement en exécution. Par exemple, à partir de la ligne de commande, saisissez **commande -o** appelera *commande* avec l'option `o`.

F.1. Options standards en ligne de commande

Avec le temps, une norme lâche sur la signification des options en la ligne de commande a évolué. Les outils GNU se conforment plus nettement à ce « standard » que les autres outils UNIX, plus anciens.

Traditionellement, les options UNIX en ligne de commande consiste en un tiret suivi d'une ou plusieurs lettres minuscules. Les outils GNU ajoutent un double tiret suivi par un mot complet ou un mot composé.

Les deux options les plus acceptées sont :

- `-h`

  `--help`

  Aide : donne le message d'utilisation et quitte.
- `-v`

  `--version`

  Version : affiche la version du programme et quitte.

Les autres options communes sont :

- `-a`

  `--all`

  Tous : affiche *toutes* les informations ou opère sur *tous* les arguments.
- `-l`

  `--list`

  Liste : liste les fichiers ou arguments sans effectuer d'autres actions.
- `-o`

  Fichier de *sortie*
- `-q`

  `--quiet`

  Silencieux : supprime `stdout`.
- `-r`

  `-R`

  `--recursive`

  Récursif : opère récursivement (sur tout le répertoire).
- `-v`

  `--verbose`

  Verbeux : affiche des informations supplémentaires sur `stdout` ou `stderr`.
- `-z`

  `--compress`

  Compresse : applique une compression (habituellement avec gzip).

Néanmoins :

- Avec **tar** et **gawk** :

  `-f`

  `--file`

  Fichier : le nom du fichier suit.

- Avec **cp**, **mv**, **rm** :

  `-f`

  `--force`

  Force : force l'écrasement du fichier cible.

Attention

Beaucoup d'outils UNIX et Linux dévient de ce « standard », donc il est dangereux d'*assumer* qu'une option donnée se comportera d'une façon standard. Vérifiez toujours la page man pour la commande en question lors d'un doute.

Un tableau complet des options recommandées pour les outils GNU est disponible sur *la page des standards GNU*.

F.2. Options en ligne de commande de Bash

Bash a lui-aussi quelques options en ligne de commande. Voici les plus utiles.

- `-c`

  Lit les commandes à partir de la chaîne suivante et affecte tous les arguments aux paramètres de position.

```
bash$ bash -c 'set a b c d; IFS="+-;"; echo "$*"'
a+b+c+d
```

- `-r`

  `--restricted`

  Exécute le shell ou un script dans le mode restreint.

- `--posix`

  Force Bash à se conformer au mode POSIX.

- `--version`

  Affiche les informations de version de Bash et quitte.

- `--`

  Fin des options. Tout ce qui suit sur la ligne de commande est un argument, pas une option.

Annexe G. Fichiers importants

Fichiers de démarrage

Ces fichiers contiennent les alias et variables d'environnement rendus accessibles au Bash exécuté en tant qu'utilisateur shell et à tous les scripts Bash appelés après l'initialisation du système.

`/etc/profile`
Défauts valables pour le système entier, configure essentiellement l'environnement (tous les shells de type Bourne, pas seulement Bash [1])

`/etc/bashrc`
Fonctions valables pour le système entier et alias370 pour Bash

`$HOME/.bash_profile`
Configuration de l'environnement par défaut spécifique à l'utilisateur, trouvée dans chaque répertoire personnel des utilisateurs (la contre-partie locale de `/etc/profile`)

`$HOME/.bashrc`
Fichier d'initialisation Bash spécifique à l'utilisateur, trouvé dans chaque répertoire personnel des utilisateurs (la contre-partie locale de `/etc/bashrc`). Seuls les shells interactifs et les scripts utilisateurs lisent ce fichier. Voir l'Annexe K, Un exemple de fichier `.bashrc` pour un fichier `.bashrc` d'exemple.

Fichier de déconnexion

`$HOME/.bash_logout`
Fichier d'instructions spécifique à l'utilisateur, trouvé dans chaque répertoire personnel des utilisateurs. En sortie d'un shell login (Bash), les commandes de ce fichier sont exécutées.

Fichiers de configuration système

`/etc/sysconfig/hwconf`
Liste et description des périphériques matériels attachés. Cette information est sous la forme de texte et peut être extraite et analysée.

```
bash$ grep -A 5 AUDIO /etc/sysconfig/hwconf
class: AUDIO
 bus: PCI
 detached: 0
 driver: snd-intel8x0
 desc: "Intel Corporation 82801CA/CAM AC'97 Audio Controller"
 vendorId: 8086
```

Note

Ce fichier est présent sur les installations à base de Red Hat et Fedora Core mais pourrait être absent sur d'autres distributions.

[1] Ceci ne s'applique pas à **csh**, **tcsh**, et d'autres shells non liés ou descendant du classique shell Bourne (**sh**).

Annexe H. Répertoires système importants

Les administrateurs système et tout autre personne écrivant des scripts d'administration devraient être intimement familier avec les répertoires système suivants.

- `/bin`

  Binaires (exécutables). Les programmes basiques du système et les outils (tels que **bash**).

- `/usr/bin` [1]

  Des binaires système supplémentaires.

- `/usr/local/bin`

  Divers binaires locaux sur cette machine.

- `/sbin`

  Binaires système. Programmes basiques pour l'administration du système et des outils (tels que **fsck**).

- `/usr/sbin`

  Encore plus de programmes et d'outils d'administration du système.

- `/etc`

  Et cetera. Scripts de configuration pour tout le système.

  Les fichiers `/etc/fstab` (table des systèmes de fichiers), `/etc/mtab` (table des systèmes de fichiers montés) et `/etc/inittab`299 sont d'un intérêt particulier.

- `/etc/rc.d`

  Scripts de démarrage sur Red Hat et les distributions dérivées de Linux.

- `/usr/share/doc`

  Documentation des paquets installés.

- `/usr/man`

  Les pages man pour tout le système.

- `/dev`

  Répertoire des périphériques. Les entrées (mais *pas* les points de montage) des périphériques physiques et virtuels. Voir le Chapitre 27, `/dev` et `/proc`.

- `/proc`

  Répertoire des processus. Contient des informations et des statistiques sur les processus en cours d'exécution et les paramètres du noyau. Voir le Chapitre 27, `/dev` et `/proc`.

- `/sys`

  Répertoire des périphériques du système. Contient des informations et des statistiques sur les périphériques et les noms des périphériques. Cela a été ajouté à Linux avec les noyaux 2.6.X.

- `/mnt`

  Monte. Répertoire pour monter les partitions des disques durs, tels que `/mnt/dos` et les périphériques physiques. Dans les nouvelles distributions Linux, le répertoire `/media` a pris la place comme point de montage préféré pour les périphériques d'entrées/sorties.

- `/media`

  Dans les nouvelles distributions Linux, le point de montage préféré pour les périphériques d'entrées/sorties tels que lecteurs CD/DVD ou clés USB.

- `/var`

  Système de fichiers *variables* (modifiables). C'est un immense répertoire de « notes » pour les données générées lors de l'utilisation d'une machine Linux/UNIX.

[1] Quelques systèmes UNIX des premiers temps avaient un disque rapide de petite capacité (contenant `/`, la partition root), et un second disque plus important mais plus rapide (contenant `/usr` et d'autres partitions). Les programmes les plus fréquemment utilisés résidaient donc dans le petit disque rapide, c'est-à-dire dans `/bin`, et les autres dans le disque lent, `/usr/bin`.

Ceci est aussi vrai pour `/sbin` et `/usr/sbin`, `/lib` et `/usr/lib`, etc.

- `/var/log`

  Fichiers de trace du système.
- `/var/spool/mail`

  Répertoire des courriers électroniques des utilisateurs.
- `/lib`

  Bibliothèques pour tout le système.
- `/usr/lib`

  Encore plus de bibliothèques pour tout le système.
- `/tmp`

  Fichiers système temporaires.
- `/boot`

  Répertoire *boot* système. Le noyau, les liens de module, la carte système et le gestionnaire de démarrage résident ici.

Avertissement

Modifier les fichiers de ce répertoire pourrait empêcher le redémarrage du système.

Annexe I. Localisation

L'adaptation à la région géographique est une fonctionnalité non documentée de Bash.

Un script shell adapté affiche son texte dans la langue définie par le paramètre système. Un utilisateur Linux à Berlin, Allemagne, aura une sortie en allemand alors que son cousin à Berlin, Maryland, aura une sortie en anglais avec le même script.

Pour créer un script autochtone, utilisez le modèle suivant pour écrire tous les messages pour l'utilisateur (messages d'erreur, invite, etc.).

```
#!/bin/bash
# localized.sh
# Script écrit par Stéphane Chazelas,
# modifié par Bruno Haible et corrigé par Alfredo Pironti

. gettext.sh

E_CDERROR=65

error()
{
  printf "$@" >&2
  exit $E_CDERROR
}

cd $var || error "`eval_gettext \"Ne peut pas entrer dans \$var.\"`"
#  Le triple antislash (échappements) en face de $var est nécessaire
#+ "parce que eval_gettext attend une chaîne là où les valeurs des
#+ variables n'ont pas encore été substituées."
#    -- par Bruno Haible
read -p "`gettext \"Entrez la valeur : \"`" var
# ...


#  ----------------------------------------------------------------
#  Alfredo Pironti commente :

#  Ce script a été modifié pour ne pas utiliser la syntaxe $"..."
#+ en faveur de la syntaxe "`gettext \"...\"`".
#  C'est OK mais, avec le nouveau programme localized.sh, les commandes
#+ "bash -D fichier" et "bash --dump-po-string fichier" ne produiront
#+ aucune sortie
#+ (car ces commandes recherchent seulement les chaînes $"...") !
#  L'UNIQUE façon d'extraire les chaînes du nouveau fichier est d'utiliser
# le programme 'xgettext'. Néanmoins, le programme xgettext est boguée.

# Notez qu'xgettext a un autre bogue.
#
# Le fragment de shell :
#    gettext -s "I like Bash"
# sera correctement extrait mais...
#    xgettext -s "I like Bash"
# ... échoue!
#  'xgettext' extraiera "-s" parce que
#+ la commande extrait seulement le tout premier argument
#+ après le mot 'gettext'.


#  Caractère d'échappement :
#
#  Pour adapter une phrase comme
#     echo -e "Bonjour\tmonde!"
#+ vous devez utiliser
#     echo -e "`gettext \"Bonjour\\tmonde\"`"
#  Le "caractère d'échappement double" avant le `t' est nécessaire parce que
#+ 'gettext' cherchera une chaîne identique à : 'Bonjour\tmonde'
#  Ceci est dû au fait que gettext lira un littéral `\')
#+ et affichera une chaîne comme "Bonjour\tmonde",
```

```
#+ donc la commande 'echo' affichera le message correctement.
#
#  Vous ne pouvez pas utiliser
#     echo "`gettext -e \"Bonjour\tmonde\"`"
#+ à cause du bogue d'xgettext expliqué ci-dessus.


# Localisons le fragment de shell suivant :
#     echo "-h display help and exit"
#
# Tout d'abord, vous pourriez faire ceci :
#     echo "`gettext \"-h display help and exit\"`"
#  De cette façon, 'xgettext' fonctionnera bien
#+ mais le programme 'gettext' lira "-h" comme une option !
#
# Une solution serait
#     echo "`gettext -- \"-h display help and exit\"`"
#  De cette façon, 'gettext' fonctionnera
#+ mais 'xgettext' extraiera "--" comme indiqué ci-dessus.
#
#  Le contournement que vous pourriez utiliser
#+ pour obtenir l'adaptation de la chaîne est
#     echo -e "`gettext \"\\0-h display help and exit\"`"
#  Nous avons ajouté un \0 (NULL) au début de la phrase.
#  De cette façon, 'gettext' fonctionnera bien ainsi que 'xgettext.'
#  De plus, le caractère NULL ne modifiera pas le comportement de la commande
#+ 'echo'.
#  ------------------------------------------------------------------
```

```
bash$ bash -D localized.sh
"Can't cd to %s."
 "Enter the value: "
```

Ceci liste tout le texte adapté (l'option `-D` liste les chaînes de caractères mises entre double guillemets préfixées par un $ sans exécuter le script).

```
bash$ bash --dump-po-strings localized.sh
#: a:6
 msgid "Can't cd to %s."
 msgstr ""
 #: a:7
 msgid "Enter the value: "
 msgstr ""
```

L'option `--dump-po-strings` de Bash ressemble à l'option `-D` mais utilise le format « po » de gettext.

Note

Bruno Haible précise :

À partir de gettext-0.12.2, **xgettext -o - localized.sh** est recommandé à la place de **bash --dump-po-strings localized.sh** parce que **xgettext** . . .

1. comprend les commandes gettext et eval_gettext (alors que bash --dump-po-strings comprend seulement la syntaxe obsolète $"...")

2. peut extraire les commentaires placés par le développeur à l'intention du traducteur.

Ce code shell n'est donc pas spécifique à Bash ; il fonctionne de la même façon avec Bash 1.x et sous les autres implémentations de /bin/sh.

Maintenant, construisez un fichier `langage.po` pour chaque langage dans lequel le script sera traduit, en spécifiant le *`msgstr`*. Alfredo Pironti donne l'exemple suivant :

fr.po:

```
#: a:6
msgid "Can't cd to $var."
msgstr "Impossible de se positionner dans le répertoire $var."
```

```
#: a:7
msgid "Enter the value: "
msgstr "Entrez la valeur : "

#  Les chaînes sont affichées avec les noms de variable, et non pas avec la
#+ syntaxe %s similaire aux programmes C.
#+ C'est une fonctionnalité géniale si le développeur utilise des noms de
#+ variables qui ont un sens !
```

Ensuite, lancez msgfmt.

`msgfmt -o localized.sh.mo fr.po`

Placez le fichier résultant `localized.sh.mo` dans le répertoire `/usr/local/share/locale/fr/LC_MESSAGES` et ajoutez les lignes suivantes au début du script :

```
TEXTDOMAINDIR=/usr/local/share/locale
TEXTDOMAIN=localized.sh
```

Si un utilisateur d'un système français lance le script, il obtiendra des messages en français.

Note

Avec les anciennes versions de Bash ou d'autres shells, gettext avec l'option `-s` est obligatoire. Dans ce cas, le script devient :

```
#!/bin/bash
# localized.sh

E_CDERROR=65

error() {
  local format=$1
  shift
  printf "$(gettext -s "$format")" "$@" >&2
  exit $E_CDERROR
}
cd $var || error "Can't cd to %s." "$var"
read -p "$(gettext -s "Enter the value: ")" var
# ...
```

Les variables `TEXTDOMAIN` et `TEXTDOMAINDIR` doivent être initialisées et exportées dans l'environnement. Cela doit être fait à l'intérieur du script.

Cette annexe a été écrite par Stéphane Chazelas avec quelques améliorations suggérées par Alfredo Pironti et Bruno Haible, le mainteneur de gettext.

Annexe J. Commandes d'historique

Le shell Bash apporte des outils en ligne de commande pour éditer et manipuler l'*historique des commandes* d'un utilisateur. C'est principalement du confort, un moyen d'économiser des frappes de touches.

Commandes d'historique de Bash :

1. **history**
2. **fc**

```
bash$ history
    1  mount /mnt/cdrom
    2  cd /mnt/cdrom
    3  ls
     ...
```

Variables internes associées aux commandes d'historique de Bash :

1. $HISTCMD
2. $HISTCONTROL
3. $HISTIGNORE
4. $HISTFILE
5. $HISTFILESIZE
6. $HISTTIMEFORMAT (Bash, version 3.0 et suivantes)
7. $HISTSIZE
8. !!
9. !$

10 !#

11 !N

12 !-N

13 !STRING

14 !?STRING?

15 ^STRING^string^

.

Malheureusement, les outils d'historique de Bash n'ont pas d'utilité dans un script.

```
#!/bin/bash
# history.sh
# Essai d'utilisation de la commande 'history' dans un script.

history                     # Aucun affichage.

var=$(history); echo "$var"  # $var est vide.

# Les commandes d'historique ne fonctionnent pas à l'intérieur d'un script.
```

```
bash$ ./history.sh
(pas de sortie)
```

Le site *Advancing in the Bash Shell* donne une bonne introduction à l'utilisation de l'historique des commandes avec Bash.

Annexe K. Un exemple de fichier `.bashrc`

Le fichier `~/.bashrc` détermine le comportement des shells interactifs. Une étude de ce fichier peut amener une meilleure compréhension de Bash.

Emmanuel Rouat a fourni le fichier `.bashrc` suivant, très élaboré et écrit pour un système Linux. Il accepte volontiers des commentaires des lecteurs.

Étudiez le fichier avec attention et n'hésitez pas à réutiliser certaines parties du code pour votre propre `.bashrc`, voire même dans vos scripts.

Exemple K.1. Exemple de fichier `.bashrc`

```
#=============================================================
#
# PERSONAL $HOME/.bashrc FILE for bash-2.05a (or later)
#
# Last modified: Tue Apr 15 20:32:34 CEST 2003
#
# This file is read (normally) by interactive shells only.
# Here is the place to define your aliases, functions and
# other interactive features like your prompt.
#
# This file was designed (originally) for Solaris but based
# on Redhat's default .bashrc file
# --> Modified for Linux.
# The majority of the code you'll find here is based on code found
# on Usenet (or internet).
# This bashrc file is a bit overcrowded - remember it is just
# just an example. Tailor it to your needs
#
#
#=============================================================

# --> Comments added by HOWTO author.
# --> And then edited again by ER :-)

#-----------------------------------
# Source global definitions (if any)
#-----------------------------------

if [ -f /etc/bashrc ]; then
        . /etc/bashrc   # --> Read /etc/bashrc, if present.
fi

#--------------------------------------------------------------
# Automatic setting of $DISPLAY (if not set already)
# This works for linux - your mileage may vary....
# The problem is that different types of terminals give
# different answers to 'who am i'......
# I have not found a 'universal' method yet
#--------------------------------------------------------------

function get_xserver ()
{
    case $TERM in
        xterm )
            XSERVER=$(who am i | awk '{print $NF}' | tr -d ')''(' )
            # Ane-Pieter Wieringa suggests the following alternative:
            # I_AM=$(who am i)
            # SERVER=${I_AM#*(}
            # SERVER=${SERVER%*)}

            XSERVER=${XSERVER%%:*}
            ;;
        aterm | rxvt)
         # find some code that works here.....
```

```
            ;;
    esac
}

if [ -z ${DISPLAY:=""} ]; then
    get_xserver
    if [[ -z ${XSERVER}  || ${XSERVER} == $(hostname) ||
          ${XSERVER} == "unix" ]]; then
        DISPLAY=":0.0"         # Display on local host
    else
        DISPLAY=${XSERVER}:0.0 # Display on remote host
    fi
fi

export DISPLAY

#---------------
# Some settings
#---------------

ulimit -S -c 0        # Don't want any coredumps
set -o notify
set -o noclobber
set -o ignoreeof
set -o nounset
#set -o xtrace        # Useful for debuging

# Enable options:
shopt -s cdspell
shopt -s cdable_vars
shopt -s checkhash
shopt -s checkwinsize
shopt -s mailwarn
shopt -s sourcepath
shopt -s no_empty_cmd_completion  # bash>=2.04 only
shopt -s cmdhist
shopt -s histappend histreedit histverify
shopt -s extglob      # Necessary for programmable completion

# Disable options:
shopt -u mailwarn
unset MAILCHECK       # I don't want my shell to warn me of incoming mail


export TIMEFORMAT=$'\nreal %3R\tuser %3U\tsys %3S\tpcpu %P\n'
export HISTIGNORE="&:bg:fg:ll:h"
export HOSTFILE=$HOME/.hosts # Put a list of remote hosts in ~/.hosts



#------------------------
# Greeting, motd etc...
#------------------------

# Define some colors first:
red='\e[0;31m'
RED='\e[1;31m'
blue='\e[0;34m'
BLUE='\e[1;34m'
cyan='\e[0;36m'
CYAN='\e[1;36m'
NC='\e[0m'              # No Color
# --> Nice. Has the same effect as using "ansi.sys" in DOS.

# Looks best on a black background.....
echo -e "${CYAN}This is BASH ${RED}${BASH_VERSION%.*}\
${CYAN} - DISPLAY on ${RED}$DISPLAY${NC}\n"
date
if [ -x /usr/games/fortune ]; then
```

```
    /usr/games/fortune -s     # makes our day a bit more fun.... :-)
fi

function _exit() # function to run upon exit of shell
{
    echo -e "${RED}Hasta la vista, baby${NC}"
}
trap _exit EXIT

#---------------
# Shell Prompt
#---------------

if [[ "${DISPLAY#$HOST}" != ":0.0" &&  "${DISPLAY}" != ":0" ]]; then
    HILIT=${red}   # remote machine: prompt will be partly red
else
    HILIT=${cyan}  # local machine: prompt will be partly cyan
fi

#  --> Replace instances of \W with \w in prompt functions below
#+ --> to get display of full path name.

function fastprompt()
{
    unset PROMPT_COMMAND
    case $TERM in
        *term | rxvt )
           PS1="${HILIT}[\h]$NC \W > \[\033]0;\${TERM} [\u@\h] \w\007\]" ;;
        linux )
           PS1="${HILIT}[\h]$NC \W > " ;;
        *)
           PS1="[\h] \W > " ;;
    esac
}

function powerprompt()
{
    _powerprompt()
    {
        LOAD=$(uptime|sed -e "s/.*: \([^,]*\).*/\1/" -e "s/ //g")
    }

 PROMPT_COMMAND=_powerprompt
 case $TERM in
   *term | rxvt  )
      PS1="${HILIT}[\A \$LOAD]$NC\n[\h \#] \W > \
           \[\033]0;\${TERM} [\u@\h] \w\007\]" ;;
        linux )
           PS1="${HILIT}[\A - \$LOAD]$NC\n[\h \#] \w > " ;;
        * )
           PS1="[\A - \$LOAD]\n[\h \#] \w > " ;;
 esac
}

powerprompt     # This is the default prompt -- might be slow.
                # If too slow, use fastprompt instead.

#==============================================================
#
# ALIASES AND FUNCTIONS
#
# Arguably, some functions defined here are quite big
# (ie 'lowercase') but my workstation has 512Meg of RAM, so ...
# If you want to make this file smaller, these functions can
# be converted into scripts.
#
# Many functions were taken (almost) straight from the bash-2.04
# examples.
#
```

```
#=============================================================

#-------------------
# Personnal Aliases
#-------------------

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'
# -> Prevents accidentally clobbering files.
alias mkdir='mkdir -p'

alias h='history'
alias j='jobs -l'
alias r='rlogin'
alias which='type -all'
alias ..='cd ..'
alias path='echo -e ${PATH//:/\\n}'
alias print='/usr/bin/lp -o nobanner -d $LPDEST'
      # Assumes LPDEST is defined
alias pjet='enscript -h -G -fCourier9 -d $LPDEST'
      # Pretty-print using enscript
alias background='xv -root -quit -max -rmode 5'
      # Put a picture in the background
alias du='du -kh'
alias df='df -kTh'

# The 'ls' family (this assumes you use the GNU ls)
alias la='ls -Al'               # show hidden files
alias ls='ls -hF --color' # add colors for filetype recognition
alias lx='ls -lXB'              # sort by extension
alias lk='ls -lSr'              # sort by size
alias lc='ls -lcr'         # sort by change time
alias lu='ls -lur'         # sort by access time
alias lr='ls -lR'               # recursive ls
alias lt='ls -ltr'              # sort by date
alias lm='ls -al |more'         # pipe through 'more'
alias tree='tree -Csu'         # nice alternative to 'ls'

# tailoring 'less'
alias more='less'
export PAGER=less
export LESSCHARSET='latin1'
export LESSOPEN='|/usr/bin/lesspipe.sh %s 2>&-'
       # Use this if lesspipe.sh exists.
export LESS='-i -N -w  -z-4 -g -e -M -X -F -R -P%t?f%f \
:stdin .?pb%pb\%:?lbLine %lb:?bbByte %bb:-...'

# spelling typos - highly personnal :-)
alias xs='cd'
alias vf='cd'
alias moer='more'
alias moew='more'
alias kk='ll'

#---------------
# a few fun ones
#---------------

function xtitle ()
{
    case "$TERM" in
        *term | rxvt)
            echo -n -e "\033]0;$*\007" ;;
        *)
            ;;
    esac
}
```

```
# aliases...
alias top='xtitle Processes on $HOST && top'
alias make='xtitle Making $(basename $PWD) ; make'
alias ncftp="xtitle ncFTP ; ncftp"

# .. and functions
function man ()
{
    for i ; do
        xtitle The $(basename $1|tr -d .[:digit:]) manual
        command man -F -a "$i"
    done
}

function ll()
{ ls -l "$@"| egrep "^d" ; ls -lXB "$@" 2>&-| egrep -v "^d|total "; }

function te()  # wrapper around xemacs/gnuserv
{
    if [ "$(gnuclient -batch -eval t 2>&-)" == "t" ]; then
        gnuclient -q "$@";
    else
        ( xemacs "$@" &);
    fi
}

#----------------------------------
# File & strings related functions:
#----------------------------------

# Find a file with a pattern in name:
function ff()

{ find . -type f -iname '*'$*'*' -ls ; }
# Find a file with pattern $1 in name and Execute $2 on it:

function fe()
{ find . -type f -iname '*'$1'*' -exec "${2:-file}" {} \;  ; }
# find pattern in a set of filesand highlight them:

function fstr()
{
    OPTIND=1
    local case=""
    local usage="fstr: find string in files.
Usage: fstr [-i] \"pattern\" [\"filename pattern\"] "
    while getopts :it opt
    do
        case "$opt" in
        i) case="-i " ;;
        *) echo "$usage"; return;;
        esac
    done
    shift $(( $OPTIND - 1 ))
    if [ "$#" -lt 1 ]; then
        echo "$usage"
        return;
    fi
    local SMSO=$(tput smso)
    local RMSO=$(tput rmso)
    find . -type f -name "${2:-*}" -print0 |
    xargs -0 grep -sn ${case} "$1" 2>&- | \
    sed "s/$1/${SMSO}\0${RMSO}/gI" | more
}

function cuttail() # Cut last n lines in file, 10 by default.
{
    nlines=${2:-10}
    sed -n -e :a -e "1,${nlines}!{P;N;D;};N;ba" $1
```

```
}

function lowercase()  # move filenames to lowercase
{
    for file ; do
        filename=${file##*/}
        case "$filename" in
        */*) dirname==${file%/*} ;;
        *) dirname=.;;
        esac
        nf=$(echo $filename | tr A-Z a-z)
        newname="${dirname}/${nf}"
        if [ "$nf" != "$filename" ]; then
            mv "$file" "$newname"
            echo "lowercase: $file --> $newname"
        else
            echo "lowercase: $file not changed."
        fi
    done
}

function swap()         # swap 2 filenames around
{
    local TMPFILE=tmp.$$
    mv "$1" $TMPFILE
    mv "$2" "$1"
    mv $TMPFILE "$2"
}


#-----------------------------------
# Process/system related functions:
#-----------------------------------

function my_ps()
{ ps $@ -u $USER -o pid,%cpu,%mem,bsdtime,command ; }

function pp()
{ my_ps f | awk '!/awk/ && $0~var' var=${1:-".*"} ; }

# This function is roughly the same as 'killall' on linux
# but has no equivalent (that I know of) on Solaris
function killps()   # kill by process name
{
   local pid pname sig="-TERM"   # default signal
   if [ "$#" -lt 1 ] || [ "$#" -gt 2 ]; then
       echo "Usage: killps [-SIGNAL] pattern"
       return;
   fi
   if [ $# = 2 ]; then sig=$1 ; fi
   for pid in $(my_ps| awk '!/awk/ && $0~pat { print $1 }' pat=${!#} ) ; do
       pname=$(my_ps | awk '$1~var { print $5 }' var=$pid )
       if ask "Kill process $pid <$pname> with signal $sig?"
           then kill $sig $pid
       fi
   done
}

function my_ip() # get IP adresses
{
    MY_IP=$(/sbin/ifconfig ppp0 | awk '/inet/ { print $2 } ' | \
sed -e s/addr://)
    MY_ISP=$(/sbin/ifconfig ppp0 | awk '/P-t-P/ { print $3 } ' | \
sed -e s/P-t-P://)
}

function ii()   # get current host related info
{
  echo -e "\nYou are logged on ${RED}$HOST"
```

```
  echo -e "\nAdditionnal information:$NC " ; uname -a
  echo -e "\n${RED}Users logged on:$NC " ; w -h
  echo -e "\n${RED}Current date :$NC " ; date
  echo -e "\n${RED}Machine stats :$NC " ; uptime
  echo -e "\n${RED}Memory stats :$NC " ; free
  my_ip 2>&- ;
  echo -e "\n${RED}Local IP Address :$NC" ; echo ${MY_IP:-"Not connected"}
  echo -e "\n${RED}ISP Address :$NC" ; echo ${MY_ISP:-"Not connected"}
  echo
}

# Misc utilities:

function repeat()       # repeat n times command
{
    local i max
    max=$1; shift;
    for ((i=1; i <= max ; i++)); do  # --> C-like syntax
        eval "$@";
    done
}

function ask()
{
    echo -n "$@" '[y/n] ' ; read ans
    case "$ans" in
        y*|Y*) return 0 ;;
        *) return 1 ;;
    esac
}

#=============================================================================
#
# PROGRAMMABLE COMPLETION - ONLY SINCE BASH-2.04
# Most are taken from the bash 2.05 documentation and from Ian McDonalds
# 'Bash completion' package
#  (http://www.caliban.org/bash/index.shtml#completion)
# You will in fact need bash-2.05a for some features
#
#=============================================================================

if [ "${BASH_VERSION%.*}" \< "2.05" ]; then
   echo "You will need to upgrade to version 2.05 \
for programmable completion"
   return
fi

shopt -s extglob        # necessary
set +o nounset          # otherwise some completions will fail

complete -A hostname   rsh rcp telnet rlogin r ftp ping disk
complete -A export     printenv
complete -A variable   export local readonly unset
complete -A enabled    builtin
complete -A alias      alias unalias
complete -A function   function
complete -A user       su mail finger

complete -A helptopic  help     # currently same as builtins
complete -A shopt      shopt
complete -A stopped -P '%' bg
complete -A job -P '%'     fg jobs disown

complete -A directory  mkdir rmdir
complete -A directory   -o default cd

# Compression
complete -f -o default -X '*.+(zip|ZIP)'  zip
complete -f -o default -X '!*.+(zip|ZIP)' unzip
```

```
complete -f -o default -X '*.+(z|Z)'      compress
complete -f -o default -X '!*.+(z|Z)'     uncompress
complete -f -o default -X '*.+(gz|GZ)'    gzip
complete -f -o default -X '!*.+(gz|GZ)'   gunzip
complete -f -o default -X '*.+(bz2|BZ2)'  bzip2
complete -f -o default -X '!*.+(bz2|BZ2)' bunzip2
# Postscript,pdf,dvi.....
complete -f -o default -X '!*.ps'  gs ghostview ps2pdf ps2ascii
complete -f -o default -X '!*.dvi' dvips dvipdf xdvi dviselect dvitype
complete -f -o default -X '!*.pdf' acroread pdf2ps
complete -f -o default -X '!*.+(pdf|ps)' gv
complete -f -o default -X '!*.texi*' makeinfo texi2dvi texi2html texi2pdf
complete -f -o default -X '!*.tex' tex latex slitex
complete -f -o default -X '!*.lyx' lyx
complete -f -o default -X '!*.+(htm*|HTM*)' lynx html2ps
# Multimedia
complete -f -o default -X '!*.+(jp*g|gif|xpm|png|bmp)' xv gimp
complete -f -o default -X '!*.+(mp3|MP3)' mpg123 mpg321
complete -f -o default -X '!*.+(ogg|OGG)' ogg123



complete -f -o default -X '!*.pl'  perl perl5

# This is a 'universal' completion function - it works when commands have
# a so-called 'long options' mode , ie: 'ls --all' instead of 'ls -a'

_get_longopts ()
{
    $1 --help | sed  -e '/--/!d' -e 's/.*--\([^[:space:].,]*\).*/--\1/'| \
grep ^"$2" |sort -u ;
}

_longopts_func ()
{
    case "${2:-*}" in
        -*) ;;
        *) return ;;
    esac

    case "$1" in
        \~*) eval cmd="$1" ;;
        *) cmd="$1" ;;
    esac
    COMPREPLY=( $(_get_longopts ${1} ${2} ) )
}
complete  -o default -F _longopts_func configure bash
complete  -o default -F _longopts_func wget id info a2ps ls recode


_make_targets ()
{
    local mdef makef gcmd cur prev i

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    # if prev argument is -f, return possible filename completions.
    # we could be a little smarter here and return matches against
    # `makefile Makefile *.mk', whatever exists
    case "$prev" in
        -*f)    COMPREPLY=( $(compgen -f $cur ) ); return 0;;
    esac

    # if we want an option, return the possible posix options
    case "$cur" in
        -)      COMPREPLY=(-e -f -i -k -n -p -q -r -S -s -t); return 0;;
    esac
```

```
    # make reads `makefile' before `Makefile'
    if [ -f makefile ]; then
        mdef=makefile
    elif [ -f Makefile ]; then
        mdef=Makefile
    else
        mdef=*.mk               # local convention
    fi

    # before we scan for targets, see if a makefile name was specified
    # with -f
    for (( i=0; i < ${#COMP_WORDS[@]}; i++ )); do
        if [[ ${COMP_WORDS[i]} == -*f ]]; then
            eval makef=${COMP_WORDS[i+1]}      # eval for tilde expansion
            break
        fi
    done

        [ -z "$makef" ] && makef=$mdef

    # if we have a partial word to complete, restrict completions to
    # matches of that word
    if [ -n "$2" ]; then gcmd='grep "^$2"' ; else gcmd=cat ; fi

    # if we don't want to use *.mk, we can take out the cat and use
    # test -f $makef and input redirection
    COMPREPLY=( $(cat $makef 2>/dev/null | \
    awk 'BEGIN {FS=":"} /^[^.#   ][^=]*:/ {print $1}' \
    | tr -s ' ' '\012' | sort -u | eval $gcmd ) )
}

complete -F _make_targets -X '+($*|*.[cho])' make gmake pmake


# cvs(1) completion
_cvs ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}
    prev=${COMP_WORDS[COMP_CWORD-1]}

    if [ $COMP_CWORD -eq 1 ] || [ "${prev:0:1}" = "-" ]; then
        COMPREPLY=( $( compgen -W 'add admin checkout commit diff \
        export history import log rdiff release remove rtag status \
        tag update' $cur ))
    else
        COMPREPLY=( $( compgen -f $cur ))
    fi
    return 0
}
complete -F _cvs cvs

_killall ()
{
    local cur prev
    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    # get a list of processes (the first sed evaluation
    # takes care of swapped out processes, the second
    # takes care of getting the basename of the process)
    COMPREPLY=( $( /usr/bin/ps -u $USER -o comm  | \
        sed -e '1,1d' -e 's#[]\[]##g' -e 's#^.*/##'| \
        awk '{if ($0 ~ /^'$cur'/) print $0}' ))

    return 0
}
```

```
complete -F _killall killall killps


# A meta-command completion function for commands like sudo(8), which
# need to first complete on a command,
# then complete according to that command's own
# completion definition - currently not quite foolproof
# (e.g. mount and umount don't work properly),
# but still quite useful --
# By Ian McDonald, modified by me.

_my_command()
{
    local cur func cline cspec

    COMPREPLY=()
    cur=${COMP_WORDS[COMP_CWORD]}

    if [ $COMP_CWORD = 1 ]; then
        COMPREPLY=( $( compgen -c $cur ) )
    elif complete -p ${COMP_WORDS[1]} &>/dev/null; then
        cspec=$( complete -p ${COMP_WORDS[1]} )
        if [ "${cspec%%-F *}" != "${cspec}" ]; then
            # complete -F &lt;function&gt;
            #
            # COMP_CWORD and COMP_WORDS() are not read-only,
            # so we can set them before handing off to regular
            # completion routine

            # set current token number to 1 less than now
            COMP_CWORD=$(( $COMP_CWORD - 1 ))
            # get function name
            func=${cspec#*-F }
            func=${func%% *}
            # get current command line minus initial command
            cline="${COMP_LINE#$1 }"
            # split current command line tokens into array
                COMP_WORDS=( $cline )
            $func $cline
        elif [ "${cspec#*-[abcdefgjkvu]}" != "" ]; then
          # complete -[abcdefgjkvu]
          #func=$( echo $cspec | sed -e 's/^.*\(-[abcdefgjkvu]\).*$/\1/' )
          func=$( echo $cspec | sed -e 's/^complete//' -e 's/[^ ]*$//' )
            COMPREPLY=( $( eval compgen $func $cur ) )
        elif [ "${cspec#*-A}" != "$cspec" ]; then
            # complete -A &lt;type&gt;
            func=${cspec#*-A }
        func=${func%% *}
        COMPREPLY=( $( compgen -A $func $cur ) )
        fi
    else
        COMPREPLY=( $( compgen -f $cur ) )
    fi
}


complete -o default -F _my_command nohup exec eval \
trace truss strace sotruss gdb
complete -o default -F _my_command command type which man nice

# Local Variables:
# mode:shell-script
# sh-shell:bash
# End:
```

Annexe L. Convertir des fichiers batch DOS en scripts shell

De nombreux programmeurs ont appris la programmation de scripts sur un PC avec DOS. Même le langage limité de fichiers batch sous DOS a permis l'écriture de scripts et d'applications assez puissantes, bien que cela nécessitait souvent des astuces assez importantes. Occasionnellement, le besoin de convertir un ancien fichier batch DOS en script shell UNIX se fait encore sentir. Ce n'est généralement pas difficile car les opérateurs d'un fichier batch DOS ne sont qu'un sous-ensemble limité des équivalents en shell.

Tableau L.1. Mots clés / variables / opérateurs des fichiers batch, et leur équivalent shell

Opérateur de fichier batch	Équivalent en script shell	Signification
%	$	préfixe d'un paramètre en ligne de commande
/	-	préfixe d'option d'une commande
\	/	séparateur d'un chemin de répertoire
==	=	(égal-à) test de comparaison de chaînes de caractères
!==!	!=	(non égal-à) test de comparaison de chaînes de caractères
|	|	tube
@	set +v	n'affiche pas la commande actuelle
*	*	« joker » dans un nom de fichier
>	>	redirection de fichier (écrasement)
>>	>>	redirection de fichier (ajout)
<	<	redirection de `stdin`
%VAR%	$VAR	variable d'environnement
REM	#	commentaire
NOT	!	négation du test suivant
NUL	/dev/null	« trou noir » pour supprimer la sortie des commandes
ECHO	echo	echo (bien plus d'options avec Bash)
ECHO.	echo	affiche une ligne vide
ECHO OFF	set +v	n'affiche pas la commande suivante
FOR %%VAR IN (LIST) DO	for var in [liste]; do	boucle « for »
:LABEL	none (inutile)	label
GOTO	none (utiliser une fonction)	saute à un autre emplacement du script
PAUSE	sleep	pause ou attente pendant un intervalle de temps
CHOICE	case ou select	choix ou menu
IF	if	test if
IF EXIST *NOMFICHIER*	if [-e nomfichier]	teste si le fichier existe
IF !%N==!	if [-z "$N"]	si le paramètre « N » n'est pas présent
CALL	source ou . (opérateur point)	« inclut » un autre script
COMMAND /C	source ou . (opérateur point)	« inclut » un autre script (identique à CALL)
SET	export	affecte une valeur à une variable d'environnement

Opérateur de fichier batch	Équivalent en script shell	Signification
SHIFT	shift	décalage gauche de la liste des arguments en ligne de commande
SGN	-lt ou -gt	signe (d'un entier)
ERRORLEVEL	$?	code de sortie
CON	stdin	« console » (stdin)
PRN	/dev/lp0	périphérique imprimante (générique)
LPT1	/dev/lp0	premier périphérique imprimante
COM1	/dev/ttyS0	premier port série

Les fichiers batch contiennent habituellement des commandes DOS. Elles doivent être remplacées par leur équivalent UNIX pour convertir un fichier batch en script shell.

Tableau L.2. Commandes DOS et leur équivalent UNIX

Commande DOS	Équivalent UNIX	Effet
ASSIGN	ln	lie un fichier ou un répertoire
ATTRIB	chmod	change les droits d'un fichier
CD	cd	change de répertoire
CHDIR	cd	change de répertoire
CLS	clear	efface l'écran
COMP	diff, comm, cmp	compare des fichiers
COPY	cp	copie des fichiers
Ctl-C	Ctl-C	break (signal)
Ctl-Z	Ctl-D	EOF (end-of-file, fin de fichier)
DEL	rm	supprime le(s) fichier(s)
DELTREE	rm -rf	supprime le répertoire récursivement
DIR	ls -l	affiche le contenu du répertoire
ERASE	rm	supprime le(s) fichier(s)
EXIT	exit	sort du processus courant
FC	comm, cmp	compare des fichiers
FIND	grep	cherche des chaînes de caractères dans des fichiers
MD	mkdir	crée un répertoire
MKDIR	mkdir	crée un répertoire
MORE	more	affiche un fichier page par page
MOVE	mv	déplace le(s) fichier(s)
PATH	$PATH	chemin vers les exécutables
REN	mv	renomme (ou déplace)
RENAME	mv	renomme (ou déplace)
RD	rmdir	supprime un répertoire
RMDIR	rmdir	supprime un répertoire
SORT	sort	trie un fichier
TIME	date	affiche l'heure système
TYPE	cat	envoie le fichier vers stdout
XCOPY	cp	copie de fichier (étendue)

Note

Virtuellement, tous les opérateurs et commandes shell et UNIX ont beaucoup plus d'options et de fonctionnalités que leur équivalent DOS et fichier batch. Beaucoup de fichiers batch DOS reposent sur des utilitaires supplémentaires, tel que **ask.com**, un équivalent limité de read.

DOS supporte un sous-ensemble très limité et incompatible de caractères d'expansion pour les noms de fichier, reconnaissant seulement les caractères * et ?.

Convertir un fichier batch DOS en script shell est généralement assez simple et le résultat est souvent bien meilleur que l'original.

Exemple L.1. VIEWDATA.BAT : Fichier Batch DOS

```
REM VIEWDATA

REM INSPIRED BY AN EXAMPLE IN "DOS POWERTOOLS"
REM                           BY PAUL SOMERSON


@ECHO OFF

IF !%1==! GOTO VIEWDATA
REM  IF NO COMMAND-LINE ARG...
FIND "%1" C:\BOZO\BOOKLIST.TXT
GOTO EXIT0
REM  PRINT LINE WITH STRING MATCH, THEN EXIT.

:VIEWDATA
TYPE C:\BOZO\BOOKLIST.TXT | MORE
REM  SHOW ENTIRE FILE, 1 PAGE AT A TIME.

:EXIT0
```

La conversion de ce script en est une belle amélioration. [1]

Exemple L.2. *viewdata.sh* : Conversion du script shell VIEWDATA.BAT

```
#!/bin/bash
# viewdata.sh
# Conversion de VISUDONNEES.BAT en script shell.

FICHIERDONNEES=/home/bozo/datafiles/book-collection.data
SANSARGUMENT=1

# @ECHO OFF       Commande inutile ici.

if [ $# -lt "$SANSARGUMENT" ]   # IF !%1==! GOTO VIEWDATA
then
  less $FICHIERDONNEES          # TYPE C:\MYDIR\BOOKLIST.TXT | MORE
else
  grep "$1" $FICHIERDONNEES     # FIND "%1" C:\MYDIR\BOOKLIST.TXT
fi

exit 0                  # :EXIT0

#  Les GOTOs, labels, smoke-and-mirrors et flimflam sont inutiles.
#  Le script converti est court, joli et propre, ce qu'on ne peut pas dire de
#+ l'original.
```

[1] Plusieurs lecteurs ont suggéré des modifications sur le fichier batch ci-dessus pour le rendre plus beau et pour qu'il soit plus compact et efficace. L'auteur du *Guide ABS* pense que cet effort est inutile. Un script Bash peut accéder à un système de fichiers DOS ou même une partition NTFS (avec l'aide de *ntfs-3g*) pour réaliser des opérations en groupe ou scriptées.

Le site *Shell Scripts on the PC* de Ted Davis contient un ensemble complet de tutoriels sur l'art démodé de la programmation des fichiers batch. Certaines de ses techniques ingénieuses peuvent raisonnablement être utilisées dans des scripts shell.

Annexe M. Exercices

Les exercices qui suivent testent et étendent votre connaissance de l'écriture de scripts. Pensez-y comme à autant de challenges, comme à un moyen de vous amuser pour aller plus loin sur le chemin de l'expertise UNIX.

Sur une petite rue de Hoboken, New Jersey, se trouve un squat difficilement descriptible dans un bâtiment en brique de deux étages et contenant sur sa façade une inscription gravée sur une plaque de marbre : *Bash Scripting Hall of Fame*. À l'intérieur, entre différents tableaux poussiéreux exposés, se trouve une plaque en laiton avec une courte, très courte liste de ces rares personnes qui ont réussi à maîtriser le contenu du *Guide sur l'écriture avancée de scripts Bash*, comme en témoigne leur réussite aux exercices suivants...

M.1. Analyse de scripts

Examinez le script suivant. Lancez-le, puis expliquez ce qu'il fait. Annotez le script, puis ré-écrivez-le d'une façon plus compacte et plus élégante.

```
#!/bin/bash

MAX=10000


  for((nr=1; nr<$MAX; nr++))
  do

    let "t1 = nr % 5"
    if [ "$t1" -ne 3 ]
    then
      continue
    fi

    let "t2 = nr % 7"
    if [ "$t2" -ne 4 ]
    then
      continue
    fi

    let "t3 = nr % 9"
    if [ "$t3" -ne 5 ]
    then
      continue
    fi

  break   # Que se passe-t-il quand vous mettez cette ligne en commentaire ?
          # Pourquoi ?

  done

  echo "Nombre = $nr"


exit 0
```

Expliquez ce que fait le script suivant. C'est réellement seulement un tube paramétré en ligne de commande.

```
#!/bin/bash

REPERTOIRE=/usr/bin
TYPEFICHIER="shell script"
JOURNAL=logfile

file "$REPERTOIRE"/* | fgrep "$TYPEFICHIER" | tee $JOURNAL | wc -l

exit 0
```

Examinez et expliquez le script suivant. Pour vous aider : find et stat.

```
#!/bin/bash

# Auteur :  Nathan Coulter
# Ce code est dans le domaine public.
# L'auteur a donné sa permission d'utiliser ce code dans le guide ABS.

find -maxdepth 1 -type f -printf '%f\000'  | {
   while read -d $'\000'; do
      mv "$REPLY" "$(date -d "$(stat -c '%y' "$REPLY") " '+%Y%m%d%H%M%S'
      )-$REPLY"
   done
}

# Attention : testez ce script dans un répertoire de tests.
# Car il touche aux fichiers qui en font partie.
```

Un lecteur a envoyé le code suivant.

```
while read LIGNE
do
  echo $LIGNE
done < `tail -f /var/log/messages`
```

Il souhaitait écrire un script traçant les modifications dans le journal système, `/var/log/messages`. Malheureusement, le bloc de code ci-dessus se bloque et ne fait rien d'utile. Pourquoi ? Corrigez-le pour qu'il fonctionne (indice : plutôt que de rediriger l'entrée standard `stdin` de la boucle336, essayez un tube).

Analyser le « code sur une ligne » (ici séparé sur deux lignes pour plus de clareté) contribué par Rory Winston :

```
export SUM=0; for f in $(find src -name "*.java");
do export SUM=$(($SUM + $(wc -l $f | awk '{ print $1 }'))); done; echo $SUM
```

Astuce : tout d'abord, diviser le script en petites sections. Puis, examinez avec attention son utilisation de l'arithmétique par parenthèses doubles, et des commandes export, find, wc et awk634.

Analyser l'Exemple A.10, « « *life* : Jeu de la Vie » » et ré-organisez-le en suivant un style simplifié et plus logique. Cherchez combien de ses variables peuvent être éliminées et essayez d'optimiser le temps d'exécution du script.

Modifiez le script pour qu'il accepte n'importe quel fichier texte ASCII pour sa « génération » initiale. Le script lira les *`$ROW*$COL`* premiers caractères et initialisera les occurences de voyelles comme des cellules « vivantes ». Indice : assurez-vous de remplacer les espaces dans le fichier d'entrée par des caractères 'tiret bas'.

M.2. Écriture de scripts

Écrivez un script pour réaliser chacune des tâches suivantes.

Facile

Liste des répertoires à partir du répertoire personnel de l'utilisateur
: Listez les répertoires et les sous rèpertoires à partir du répertoire personnel de l'utilisateur et sauvegardez le résultat dans un fichier. Compressez le fichier, demandez à l'utilisateur d'insérer une disquette et d'appuyer sur **ENTER**. Enfin, sauvegardez le fichier sur la disquette.

Script auto-reproductible
: Écrivez un script qui se sauvegarde lui-même, c'est-à-dire qu'il se copie dans un fichier nommé `backup.sh`.

  Astuce : utilisez la commande cat et les paramètres de position appropriés33.

Conversion de boucles for en boucle while et en boucles until
Convertissez les *boucles for* de l'Exemple 10.1, « Des boucles *for* simples » en *boucles while*. Indice : stockez les données dans un tableau375 et parcourez les éléments du tableau.

Après avoir effectué ce « gros lifting », convertissez maintenant les boucles de l'exemple en *boucles until*.

Modification de l'espacement des lignes d'un fichier texte
Écrivez un script qui lit chaque ligne d'un fichier cible puis écrit la ligne sur `stdout` suivie d'une ligne vide. Ceci a pour effet de *doubler l'espacement des lignes* du fichier.

Incluez tout le code nécessaire pour vérifier que le script obtient l'argument nécessaire en ligne de commande (un nom de fichier), et que le fichier spécifié existe.

Quand le script s'exécute correctement, modifiez-le pour *tripler l'espacement des lignes* du fichier cible.

Enfin, écrivez un script pour supprimer toutes les lignes vides du fichier cible.

Liste inverse
Écrivez un script qui s'affiche lui-même sur `stdout` mais *à l'envers*.

Décompression automatique de fichiers
À partir d'une liste de noms de fichiers donnée en entrée, cherchez sur chaque fichier cible le type de compression utilisé en analysant la sortie de la commande file. Puis, appellez automatiquement la commande de décompression appropriée (**gunzip**, **bunzip2**, **unzip**, **uncompress** ou autre). Si un fichier cible n'est pas compressé, affichez un message d'erreur mais ne faites aucune autre action sur ce fichier particulier.

Identifiant unique
Générez un identifiant hexadécimal à six chiffres « unique » pour votre ordinateur. N'utilisez *pas* la commande défectueuse hostid. Indice : **md5sum `/etc/passwd`**, puis sélectionnez les six premiers chiffres de la sortie.

Sauvegarde
Archivez dans une « archive tar » (fichier `*.tar.gz`) tous les fichiers de votre répertoire personnel (`/home/votre-nom`) qui ont été modifiés dans les dernières 24 heures. Indice : utilisez find.

Vérifier si un processus est toujours en cours d'exécution
Avec un identifiant de processus (*PID*) comme argument, ce script vérifiera, à un interval spécifié par l'utilisateur, si le processus donné est toujours en cours d'exécution. Vous pouvez utiliser les commandes ps et sleep.

Premiers
Afficher (sur stdout) tous les nombres premiers entre 60000 et 63000. La sortie doit être joliment formatée en colonnes (indice : utilisez printf).

Numéros de loterie
Un type de loterie implique de choisir cinq numéros différents entre 1 et 50. Écrivez un script générant cinq numéros pseudo-aléatoires dans cet intervalle *sans doublons*. Le script donnera la possibilité d'afficher les nombres sur `stdout` ou de les sauvegarder dans un fichier avec la date et l'heure où cet ensemble de nombres a été généré.

Intermédiaire

Entier ou chaîne
Écrivez une fonction352 de script déterminant si un argument passé est un entier ou une chaîne. La fonction renverra TRUE (0) s'il s'agit d'un entier et FALSE (1) s'il s'agit d'une chaîne.

Astuce : que renvoie l'expression suivante quand `$1` n'est *pas* un entier ?

```
expr $1 + 0
```

Gestion de l'espace disque
Listez, un par un, tous les fichiers faisant plus de 100 Ko dans l'arborescence du répertoire `/home/utilisateur`. Donnez à l'utilisateur la possibilité de supprimer ou de compresser le fichier, puis continuez en affichant le suivant. Écrivez un journal avec le nom de tous les fichiers supprimés et l'heure de leur suppression.

Supprimer les comptes inactifs
Les comptes inactifs sur un réseau font perdre de l'espace disque et pourraient devenir un risque de sécurité. Écrivez un script d'administration (appelé par *root* ou le démon cron) vérifiant et supprimant les comptes utilisateurs qui n'ont pas été utilisés pendant les 90 derniers jours.

Forcer les quotas disque
Écrivez un script pour un système multi-utilisateurs vérifiant l'utilisation du disque par les utilisateurs. Si un utilisateur dé-

passe la limite fixée (100 Mo par exemple) dans leur répertoire personnel (`/home/utilisateur`), alors le script lui enverra automatiquement un message d'avertissement par mail.

Le script utilisera les commandes du et mail. Comme option, il permettra de configurer et de forcer les quotas en utilisant les commandes quota et setquota.

Informations sur les utilisateurs connectés

Pour tous les utilisateurs connectés, affichez leur noms réels et l'heure et la date de leur dernière connexion.

Astuce : utilisez who, lastlog et analysez `/etc/passwd`.

Suppression sécurisée

Implémenter, sous la forme d'un script, une commande « sécurisée » de suppression, `sdel.sh`. Les noms de fichiers passés en tant qu'arguments de la ligne de commande à ce script ne sont pas supprimés, mais, à la place, compressés avec gzip s'ils ne sont pas déjà compressés (utilisez file pour le vérifier), puis déplacés dans le répertoire `~/TRASH`. Une fois lancé, le script vérifie si le répertoire `~/TRASH` contient des fichiers datant de plus de 48 heures et les supprime de façon permanente. (Une meilleure alternative pourraît demander l'utilisation d'un deuxième script pour gérer cela --- périodiquement appelé par le démon cron.)

Crédit supplémentaire : Écrivez le script pour qu'il gère fichiers et répertoires récursivement. Ceci donnera la possibilité de « supprimer en toute sécurité » des structures complètes de répertoire.

Faire la monnaie

Quel est le moyen le plus efficace de faire la monnaie sur 1,68 euros en utilisant seulement les pièces en circulation courante (jusqu'à 50 centimes) ? Ce sont trois pièces de 50 centimes, une de 10, une de 5 et trois de un.

À partir d'une entrée arbitraire en ligne de commande en euros et centimes ($*.??), calculez la monnaie en utilisant le plus petit nombre de pièces. Si la monnaie de votre pays n'est pas l'euro, vous pouvez utiliser votre monnaie locale à la place. Le script devra analyser l'entrée en ligne de commande, puis la modifier en multiple de la plus petite unité monétaire (centime ou autre). Indice : jetez un oeil sur l'Exemple 23.8, « Convertir des nombres en chiffres romains ».

Equations quadratiques

Résolvez une équation « quadratique » de la forme *`Ax^2 + Bx + C = 0`*. Créez un script qui prend comme arguments les coefficients **`A`**, **`B`** et **`C`**, et renvoie les solutions avec jusqu'à quatre chiffres après la virgule.

Indice : envoyez les coefficients via un tube à bc en utilisant la formule bien connue *`x = ( -B +/- sqrt( B^2 - 4AC ) ) / 2A`*.

Somme des nombres correspondants

Trouvez la somme de tous les nombres de cinq chiffres (dans l'intervalle 10000-99999) contenant *exactement deux* des chiffres de l'ensemble suivant : { 4, 5, 6 }. Ils peuvent se répéter à l'intérieur du même nombre et, si c'est le cas, ils sont comptés une fois à chaque occurence.

Quelques exemples de nombres correspondants : 42057, 74638 et 89515.

Nombres porte-bonheur

Un "nombre porte-bonheur" est un de ces nombres dont les chiffres, pris individuellement, additionnés valent 7. Par exemple, 62431 est un "nombre porte-bonheur" (6 + 2 + 4 + 3 + 1 = 16, 1 + 6 = 7). Trouvez tous les "nombres porte-bonheur" entre 1000 et 10000.

Classer par ordre alphabétique une chaîne de caractères

Classez par ordre alphabétique (suivant l'ordre ASCII) une chaîne de caractères arbitraire lue sur la ligne de commande.

Analyse

Analysez `/etc/passwd` et affichez son contenu sous la forme d'un joli tableau, facile à lire.

Tracer les connexions

Analyser `/var/log/messages` pour produire un fichier joliment formaté des connexions des utilisateurs avec l'heure de connexion. Le script peut devoir se lancer en tant que *root* (conseil : cherchez la chaîne « LOGIN »).

Mise en forme de l'affichage d'un fichier de données

Certaines bases de données et tableurs utilisent des fichiers de sauvegarde avec *des valeurs séparées par des virgules* (CSV). D'autres applications ont souvent besoin d'analyser ces fichiers.

À partir d'un fichier de données au format CSV, de la forme:

```
Jones,Bill,235 S. Williams St.,Denver,CO,80221,(303) 244-7989
Smith,Tom,404 Polk Ave.,Los Angeles,CA,90003,(213) 879-5612
...
```

Reformatez les données et affichez-les sur `stdout` avec des colonnes disposant d'un titre et de même largeur.

Justification
À partir d'une entrée texte au format ASCII provenant soit de `stdin` soit d'un fichier, justifiez à droite chaque ligne suivant une longueur de ligne spécifiée par l'utilisateur en ajustant l'espacement entre les mots et envoyez la sortie sur `stdout`.

Liste de diffusion
En utilisant la commande mail, écrivez un script gérant une liste de diffusion simple. Le script envoie automatiquement par courrier électronique la lettre d'informations mensuelle de la compagnie, lue à partir d'un fichier texte spécifié aux adresses de la liste de diffusion que le script lit à partir d'un autre fichier spécifié.

Générer des mots de passe
Générez des mots de passe de huit caractères pseudo-aléatoires en utilisant les caractères contenus dans les intervalles [0-9], [A-Z], [a-z]. Chaque mot de passe doit contenir au moins deux chiffres.

Surveiller un utilisateur
Vous suspectez un utilisateur particulier sur le réseau d'abuser de ces droits et certainement de tenter de corrompre le système. Écrivez un script qui surveille et trace automatiquement son activité de la semaine et supprime les entrées qui ont plus de sept jours.

Vous pouvez utiliser last, lastlog et lastcomm pour vous aider dans la surveillance du suspect.

Vérifier les liens cassés
En utilisant lynx avec l'option `-traversal`, écrivez un script qui vérifie les liens cassés d'un site web.

Difficile

Tester des mots de passe
Écrire un script pour vérifier et valider les mots de passe. Le but est de marquer les candidats « faibles » ou facilement devinables.

Un mot de passe en test sera en entrée du script via un paramètre de la ligne de commande. Pour être considéré comme acceptable, un mot de passe doit satisfaire les qualifications minimums suivantes :

- Longueur minimum de huit caractères
- Contenir au moins un caractère numérique
- Contenir au moins un caractère non alphabétique, numérique et figurant : @, #, $, %, &, *, +, -, =

Optionnel :

- Faire une vérification de type dictionnaire sur chaque séquence d'au moins quatre caractères consécutifs du mot de passe en test. Ceci éliminera les mots de passe contenant des « mots » disponibles dans un dictionnaire standard.
- Autoriser le script à vérifier tous les mots de passe du système. Ils pourraient ou non résider dans `/etc/passwd`.

Cet exercice a pour but de tester la maîtrise des expressions rationnelles313.

Référence croisée
Écrire un script qui génère une *référence croisée* (*concordance*) sur un fichier cible. La sortie sera un listing de toutes les occurences de mots dans le fichier cible avec les numéros de ligne où ils apparaissent. Habituellement, les constructions de *liste liée* seraient utilisées dans un tel cas. Du coup, vous avez intérêt à utiliser des tableaux375 pour cet exercice. Exemple 15.12, « Analyse de fréquence d'apparition des mots » n'est probablement *pas* un bon début.

Racine carrée
Écrire un script qui calcule les racines carrées de nombres en utilisant la *méthode de Newton*.

L'algorithme pour cela, exprimé en pseudo-code453 Bash, est le suivant :

```
#  Méthode d'Isaac Newton pour une extraction rapide
#+ des racines carrées.

tentative = $argument
#  $argument est le nombre dont on veut trouver la racine carrée.
#  $tentative est la tentative calculée -- ou une solution triviale --
#+ de la racine carrée.
#  Notre première tentative de racine carrée est l'argument lui-même.

ancienne_tentative = 0
```

```
# $ancienne_tentative est l'ancien $tentative.

tolerance = .000001
# La tolérance acceptée pour notre calcul.

nb_boucles = 0
# Gardons trace du nombre de fois passé dans la boucle.
# Certains arguments demanderont plus d'itérations que d'autres.


while [ ABS( $tentative $ancienne_tentative ) -gt $tolerance ]
#       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Corrigez la syntaxe.

#      "ABS" est une fonction à virgule flottante pour trouver la valeur absolue
#+      de la différence entre les deux termes.
#             Donc, tant que la différence entre la solution actuelle et ancienne
#+            ne dépasse pas la tolérance, on continue.

do
   ancienne_tentative = $tentative  # Met $ancienne_tentative avec la valeur de
$tentative.

#  ========================================================
   tentative = ( $ancienne_tentative + ( $argument / $ancienne_tentative ) ) / 2.0
#        = 1/2 ( ($ancienne_tentative **2 + $argument) / $ancienne_tentative )
#  équivalent à :
#        = 1/2 ( $ancienne_tentative + $argument / $ancienne_tentative )
#  c'est-à-dire une moyenne de la solution tentée et la proportion d'une déviation
#+ de l'argument
#+ (donc division par deux de l'erreur).
#  Ceci converge sur une solution précise
#+ avec étonnament peu d'itérations de boucle...
#+ pour arguments > $tolerance, bien sûr.
#  ========================================================

   (( nb_boucles++ ))     # Mise à jour du compteur de boucle.
done
```

C'est une recette assez simple et dont la conversion en Bash *semble* au premier regard être assez facile. Cela étant dit, le problème est que Bash n'a pas de support natif pour les nombres à virgules flottantes. Donc, le rédacteur du script aura besoin d'utiliser bc ou peut-être awk634 pour convertir les nombres et faire les calculs. Cela peut devenir un gros bazar...

Journal des accès aux fichiers
Tracez tous les accès aux fichiers dans `/etc` sur une journée. Ce journal devra inclure le nom du fichier, le nom de l'utilisateur et l'heure d'accès. Il doit aussi être indiqué si le fichier est modifié. Écrivez cette information en enregistrements bien mis en forme dans un fichier journal.

Traces sur les processus
Écrivez un script pour tracer de façon continue tous les processus en cours d'exécution et pour garder trace de tous les processus fils que chaque parent lance. Si un processus lance plus de cinq fils, alors le script envoie un mail à l'administrateur système (ou *root*) avec toutes les informations intéressantes, ceci incluant l'heure, le PID du père, les PID des enfants, etc. Le script ajoute un rapport dans un journal toutes les dix minutes.

Suppression des commentaires
Supprimez tous les commentaires d'un script shell dont le nom est spécifié en ligne de commande. Notez que la « ligne #! » ne doit pas subir le même traitement.

Suppression des balises HTML
Supprimez toutes les balises HTML d'un fichier, puis reformatez-le en lignes de 60 à 75 caractères de longueur. Réajustez les espacements de paragraphes et de blocs de façon appropriée, et convertissez les tables HTML en leur équivalent texte approximatif.

Conversion XML
Convertissez un fichier XML à la fois en HTML et en texte.

Chasse aux spammeurs
Écrivez un script qui analyse un courrier électronique spam en faisant des recherches DNS à partir de l'adresse IP spécifiée dans les en-têtes pour identifier les hôtes relais ainsi que le fournisseur d'accès internet (FAI) d'origine. Le script renverra le

message de spam non modifié aux FAI responsables. Bien sûr, il sera nécessaire de filtrer les *adresses IP de votre propre FAI* si vous ne voulez pas vous plaindre de vous-même.

Si nécessaire, utilisez les commandes d'analyse réseau appropriées.

Pour quelques idées, voir l'Exemple 15.41, « Analyser le domaine d'un courrier indésirable » et l'Exemple A.30, « Identification d'un spammer ».

En option : écrire un script qui recherche dans une liste de mails et supprime le pourriel suivant des filtres spécifiés.

Créer des pages man
Écrire un script automatisant le processus d'écriture de pages man.

Avec un fichier texte contenant l'information à formater en une *page man*, le script lira le fichier puis les commandes groff appropriées pour sortir les *pages man* sur `stdout`. Le fichier texte contient des blocs d'information sous des en-têtes standards pour en-têtes de *pages man*, c'est-à-dire « NAME », « SYNOPSIS », « DESCRIPTION, », etc.

See Exemple 15.29, « *manview* : Visualisation de pages man formatées ».

Code Morse
Convertissez un fichier texte en code Morse. Chaque caractère du fichier texte sera représenté par le code Morse correspondant, un ensemble de points et de tirets bas, séparé par une espace blanche du suivant. Par exemple :

```
Appelle le script "morse.sh" avec "script" en argument.


$ sh morse.sh script

... _._. ._. .. .__. _
s   c    r   i   p   t
```

Dump Hexa
Faites une sortie hexadécimale d'un fichier binaire donné en argument. La sortie devra être en colonnes, le premier champ indiquant l'adresse, chacun des huit champs suivants un nombre hexa de quatre octets, et le dernier champ l'équivalent ASCII des huit champs précédents.

La suite évidente à tout ceci est d'étendre la sortie hexadécimal à un désassembleur. En utilisant une table de recherche ou une autre stuce, convertir les valeurs hexadécimales des codes op du 80x86.

Emulation d'un registre à décalage
En s'inspirant de l'Exemple 26.14, « Émuler une pile », écrivez un script qui émule un registre à décalage de 64 bits par un tableau375. Implémentez des fonctions pour *charger* le registre, *décaler à gauche*, *décaler à droite* et *faire une rotation*. Enfin, écrivez une fonction qui interprète le contenu du registre comme huit caractères ASCII sur 8 bits.

Déterminant
Résolvez un déterminant 4 x 4.

Mots cachés
Écrivez un générateur de puzzle de « recherche de mots », un script qui cache dix mots d'entrée dans une matrice de 10 x 10 lettres choisies au hasard. Les mots peuvent être cachés en horizontal, en vertical et en diagonale.

Optionnel : écrivez un script qui *résout* des puzzles de mots. Pour ne pas rendre ceci trop complexe, le script trouvera seulement les mots horizontaux et verticaux (astuce : traitez chaque ligne et colonne comme une chaîne et recherchez les sous-chaînes).

Anagramme
Faites un anagramme des quatre lettres d'entrée. Par exemple, les anagrammes de *word* sont : *do or rod row word*. Vous pouvez utiliser `/usr/share/dict/linux.words` comme liste de référence.

Suite de mots
Une « suite de mots » est une séquence de mots, chacun différant du précédent d'une seule lettre.

Par exemple, pour faire « suivre » *vase* à *mark* :

```
mark --> park --> part --> past --> vast --> vase
         ^           ^       ^      ^           ^
```

Écrivez un script résolvant des puzzles de type « suite de mots ». Étant donné un mot de départ et un mot de fin, le script donnera toutes les étapes intermédiaires de la « suite ». Notez que *tous* les mots de la séquence doivent être des mots existants dans le dictionnaire.

Index 'brouillard'

L'« index brouillard » d'un passage de texte permet d'estimer sa difficulté de lecture, par un nombre correspondant grossièrement à un niveau d'école. Par exemple, un passage d'index 12 devrait être compréhensible par toute personne ayant plus de 12 ans d'école.

La version 'Gunning' de cet index utilise l'algorithme suivant.

1. Choisissez une section de texte d'au moins 100 mots de longueur.
2. Comptez le nombre de phrases (une portion d'une phrase tronquée par les limites de la section de texte compte pour un).
3. Trouvez le nombre moyen de mots par phrase.

   MOY_MOT_PHRASE = TOTAL_MOTS / PHRASES
4. Comptez le nombre de mots « difficiles » dans le segment -- ceux contenant au moins trois syllabes. Divisez cette quantité par le nombre total de mots pour obtenir la proportion de mots difficiles.

   PRO_MOTS_DIFFIC = MOTS_LONGS / TOTAL_MOTS
5. L'index 'Gunning' est la somme des deux quantités ci-dessus, multiplié par 0,4, arrondie à l'entier le plus proche.

   G_FOG_INDEX = int (0.4 * (MOY_MOT_PHRASE + PRO_MOTS_DIFFIC))

L'étape 4 est de loin la partie la plus difficile de cet exercice. Il existe différents algorithmes pour estimer le nombre de syllabes dans un mot. Un moyen empirique consiste à considérer le nombre de lettres dans un mot et le mélange voyelles - consonnes.

Une stricte interprétation de l'index 'Gunning' ne compte pas les mots composés et les noms propres comme des mots « difficiles », mais cela compliquerait considérablement le script.

Calculer PI en utilisant l'aiguille de Buffon

Le mathématicien français du 18è siècle Buffon a conçu une expérimentation nouvelle. Lâchez de manière répétée une aiguille de longueur « n » sur un sol en bois composé de planches longues et étroites. Les trous séparant les planches de largeur égale sont à une distance fixe « d » les uns des autres. Gardez une trace du nombre de lâchés et du nombre de fois où l'aiguille fait une intersection avec un trou sur le sol. Le ratio de ces deux nombres se trouve être un multiple fractionnel de PI.

Dans l'esprit de l'Exemple 15.49, « Calculer PI », écrivez un script qui lance une simulation de Monte Carlo de l'aiguille de Buffon. Pour simplifier le problème, initialisez la longueur de l'aiguille à la distance entre les trous, $n = d$.

Indice : il existe en fait deux variables critiques, la distance du centre de l'aiguille au trou le plus proche et l'angle formé par l'aiguille et le trou. Vous pouvez utiliser bc pour réaliser les calculs.

Cryptage Playfair

Implémentez le cryptage Playfair (Wheatstone) dans un script.

Le cryptage Playfair crypte le texte par substitution de chaque « diagramme » de deux lettres (groupe). Traditionnellement, on utiliserait un carré de cinq lettres sur cinq composant un alphabet pour le cryptage et le décriptage.

```
   C O D E S
   A B F G H
   I K L M N
   P Q R T U
   V W X Y Z

Chaque lettre de l'alphabet apparaît une fois. Le I représente aussi le J.
Le mot clé choisi arbitrairement, "CODES" vient en premier, ensuite le
reste de l'alphabet, de gauche à droite, en évitant les lettres déjà utilisées.

Pour crypter, séparez le message texte en groupes de deux lettres. Si un groupe
a deux lettres identiques, supprimez la seconde et formez un nouveau groupe.
Si il reste une seule lettre à la fin, insérez le caractère "null", typiquement
un "X".

THIS IS A TOP SECRET MESSAGE

TH IS IS AT OP SE CR ET ME SA GE

Pour chaque groupe, il existe trois possibilités.
-------------------------------------------------
1) Les deux lettres sont sur la même ligne du carré
   Pour chaque lettre, substituez celle immédiatement à droite sur la ligne. Si
```

```
   nécessaire, retournez au début de la ligne.
ou

2) Les deux lettres sont dans la même colonne du carré
   Pour chaque lettre, substituez celle immédiatement en dessous sur cette colonne.
   Si nécessaire, retournez en haut de la colonne.

ou

3) Les deux lettres sont aux coins d'un rectangle à l'intérieur du carré.
   Pour chaque lettre, substituez celle à l'autre coin du rectangle se trouvant
   sur la même ligne.


Le groupe "TH" fait partie du cas #3.
G H
M N
T U           (Rectangle avec "T" et "H" aux coins)

T --&gt; U
H --&gt; G


Le groupe "SE" fait partie du cas #1.
C O D E S     (Ligne contenant "S" et "E")

S --&gt; C  (on retourne au début de la ligne)
E --&gt; S

========================================================================

Pour décrypter un texte, inversez la procédure ci-dessus pour #1 et #2
(déplacez-vous dans la direction opposée pour la substitution). Pour le cas #3,
prenez juste les deux coins restants du rectangle.


Le travail classique d'Helen Fouche Gaines, ELEMENTARY CRYPTANALYSIS (1939),
donne les étapes détaillées sur le cryptage Playfair et ses méthodes de
résolution.
```

Ce script aura trois sections principales

I. Génération du « carré de clé » basé sur un mot-clé saisi par l'utilisateur.

II. Cryptage d'un message « texte ».

III Décryptage du texte crypté.

Ce script utilisera de façon poussée les tableaux375 et les fonctions352.

--

Merci de ne pas envoyer à l'auteur vos solutions pour ces exercices. Il existe de bien meilleures façons de l'impressionner avec votre intelligence comme, par exemple, l'envoi de corrections de bogues et des suggestions pour améliorer ce livre.

Annexe N. Historique des révisions

Ce document est tout d'abord apparu sous la forme d'un guide pratique de 60 pages à la fin du printemps 2000. Depuis, il a reçu un grand nombre de mises à jour et de révisions. Ce livre n'aurait pas pu être écrit sans l'assistance de la communauté Linux et spécialement celle des volontaires du *Linux Documentation Project*.

Voici l'email envoyé au LDP pour proposer la version 0.1.

```
From thegrendel@theriver.com Sat Jun 10 09:05:33 2000 -0700
Date: Sat, 10 Jun 2000 09:05:28 -0700 (MST)
From: "M. Leo Cooper" <thegrendel@theriver.com>
X-Sender: thegrendel@localhost
To: ldp-discuss@lists.linuxdoc.org
Subject: Permission to submit HOWTO

Dear HOWTO Coordinator,

I am working on and would like to submit to the LDP a HOWTO on the subject
of "Bash Scripting" (shell scripting, using 'bash'). As it happens,
I have been writing this document, off and on, for about the last eight
months or so, and I could produce a first draft in ASCII text format in
a matter of just a few more days.

I began writing this out of frustration at being unable to find a
decent book on shell scripting. I managed to locate some pretty good
articles on various aspects of scripting, but nothing like a complete,
beginning-to-end tutorial.  Well, in keeping with my philosophy, if all
else fails, do it yourself.

As it stands, this proposed "Bash-Scripting HOWTO" would serve as a
combination tutorial and reference, with the heavier emphasis on the
tutorial. It assumes Linux experience, but only a very basic level
of programming skills. Interspersed with the text are 79 illustrative
example scripts of varying complexity, all liberally commented. There
are even exercises for the reader.

At this stage, I'm up to 18,000+ words (124k), and that's over 50 pages of
text (whew!).


I haven't mentioned that I've previously authored an LDP HOWTO, the
"Software-Building HOWTO", which I wrote in Linuxdoc/SGML. I don't know
if I could handle Docbook/SGML, and I'm glad you have volunteers to do
the conversion. You people seem to have gotten on a more organized basis
these last few months. Working with Greg Hankins and Tim Bynum was nice,
but a professional team is even nicer.

Anyhow, please advise.


Mendel Cooper
thegrendel@theriver.com
```

Tableau N.1. Historique des révisions

Sortie	Date	Commentaires
0.1	14 juin 2000	Sortie initiale.
0.2	30 octobre 2000	Correction de bogues, quelques ajouts d'informations et des scripts d'exemples supplémentaires.
0.3	12 février 2001	Mise à jour majeure.

Sortie	Date	Commentaires
0.4	08 juillet 2001	Révision complète et extension du livre.
0.5	03 septembre 2001	Mise à jour majeure : corrections, ajouts d'informations, réorganisation des sections.
1.0	14 octobre 2001	Sortie stable : corrections, reorganisation, ajouts d'informations.
1.1	06 janvier 2002	Corrections, ajouts d'informations et de scripts.
1.2	31 mars 2002	Corrections, ajouts d'informations et de scripts.
1.3	02 juin 2002	Sortie de TANGERINE : quelques corrections, plus d'informations et quelques scripts ajoutés.
1.4	16 juin 2002	Sortie de MANGO : quelques erreurs de frappe corrigées, plus d'informations et de scripts.
1.5	13 juillet 2002	Sortie de PAPAYA : quelques corrections de bogues, encore plus d'informations et de scripts ajoutés.
1.6	29 septembre 2002	Sortie de POMEGRANATE : corrections de bogues, plus d'informations et un script supplémentaire.
1.7	05 janvier 2003	Sortie de COCONUT : quelques corrections de bogues, plus d'informations et un script de plus.
1.8	10 mai 2003	Sortie de BREADFRUIT : un certain nombre de corrections de bogues, plus de scripts et d'informations.
1.9	21 juin 2003	Sortie de PERSIMMON : correctifs et informations supplémentaires.
2.0	24 août 2003	Sortie de GOOSEBERRY : mise à jour majeure.
2.1	14 septembre 2003	Sortie de HUCKLEBERRY : corrections de bogues et ajouts d'informations.
2.2	31 octobre 2003	Sortie de CRANBERRY : mise à jour majeure.
2.3	03 janvier 2004	Sortie de STRAWBERRY : corrections de bogues et ajout d'informations.
2.4	25 janvier 2004	Sortie de MUSKMELON : corrections de bogues.
2.5	15 février 2004	Sortie de STARFRUIT : corrections de bogues et ajout d'informations.
2.6	15 mars 2004	Sortie de SALAL : mise à jour mineure.
2.7	18 avril 2004	Sortie de MULBERRY : mise à jour mineure.
2.8	11 juillet 2004	Sortie de ELDERBERRY : mise à jour mineure.
3.0	3 octobre 2004	Sortie de LOGANBERRY : mise à jour majeure.
3.1	14 novembre 2004	Sortie de 'BAYBERRY' : corrections de bogues.
3.2	6 février 2005	Sortie de BLUEBERRY : petite mise à jour.

Sortie	Date	Commentaires
3.3	20 mars 2005	Sortie de RASPBERRY : correctifs et plus de matériels.
3.4	8 mai 2005	Sortie de TEABERRY : corrections, révisions au niveau du style.
3.5	5 juin 2005	Sortie de BOXBERRY : correction de bogues, quelques informations ajoutées.
3.6	28 août 2005	Sortie de POKEBERRY : corrections de bogues, quelques informations ajoutées.
3.7	23 octobre 2005	Sortie de WHORTLEBERRY : corrections de bogues, quelques informations ajoutées.
3.8	26 février 2006	Sortie de BLAEBERRY : corrections de bogues, quelques ajouts.
3.9	15 mai 2006	Sortie de SPICEBERRY : corrections de bogues, quelques ajouts.
4.0	18 juin 2006	Sortie de WINTERBERRY : réorganisation majeure.
4.1	8 octobre 2006	Sortie de WAXBERRY : mise à jour mineure.
4.2	10 décembre 2006	Sortie de SPARKLEBERRY : mise à jour importante.
4.3	29 avril 2007	Sortie de INKBERRY : corrections de bogues, ajout d'informations.
5.0	24 juin 2007	Sortie de SERVICEBERRY : mise à jour majeure.
5.1	10 novembre 2007	Sortie de LINGONBERRY : mise à jour mineure.
5.2	16 mars 2008	Sortie de SILVERBERRY : mise à jour importante.
5.3	11 mai 2008	Sortie de GOLDENBERRY : mise à jour mineure.

Annexe O. Sites miroirs

La dernière mise à jour de ce document, en tant qu' « archive tar » incluant à la fois les sources SGML et le HTML rendu, sont téléchargeables depuis le *site de l'auteur*.

Le principal site miroir de ce document est le *projet de documentation Linux*, qui maintient d'autres livres et guides pratiques.

Il existe encore un autre miroir pour ce document sur *morethan.org*.

Vous pouvez télécharger la version *PDF ici*.

Annexe P. Liste de choses à faire

- Une étude complète des incompatibilités entre le shell Bash et le shell Bourne classique464.
- Identique à ci-dessus mais pour le shell Korn (*ksh*).
- Une introduction à la programmation de CGI en utilisant Bash.

  Voici un exemple de script CGI pour commencer.

  Exemple P.1. Afficher l'environnement du serveur

```
#!/bin/bash
# Vous pourriez avoir à changer l'emplacement sur votre site
# (sur les serveurs des ISP, Bash pourrait ne pas être au même endroit).
# Autres emplacements : /usr/bin ou /usr/local/bin
# Pourrait même être testé sans sha-bang.

# test-cgi.sh
# par Michael Zick
# Utilisé avec sa permission


# Désactive la recherche de fichiers.
set -f

# Les en-têtes indiquent au navigateur quoi attendre.
echo Content-type: text/plain
echo

echo CGI/1.0 rapport du script de test :
echo

echo "configuration de l'environnement :"
set
echo

echo où se trouve bash ?
whereis bash
echo


echo qui sommes-nous ?
echo ${BASH_VERSINFO[*]}
echo

echo argc vaut $#. argv vaut "$*".
echo

# Variables d'environnement attendues avec CGI/1.0.

echo SERVER_SOFTWARE = $SERVER_SOFTWARE
echo SERVER_NAME = $SERVER_NAME
echo GATEWAY_INTERFACE = $GATEWAY_INTERFACE
echo SERVER_PROTOCOL = $SERVER_PROTOCOL
echo SERVER_PORT = $SERVER_PORT
echo REQUEST_METHOD = $REQUEST_METHOD
echo HTTP_ACCEPT = "$HTTP_ACCEPT"
echo PATH_INFO = "$PATH_INFO"
echo PATH_TRANSLATED = "$PATH_TRANSLATED"
echo SCRIPT_NAME = "$SCRIPT_NAME"
echo QUERY_STRING = "$QUERY_STRING"
echo REMOTE_HOST = $REMOTE_HOST
echo REMOTE_ADDR = $REMOTE_ADDR
echo REMOTE_USER = $REMOTE_USER
echo AUTH_TYPE = $AUTH_TYPE
echo CONTENT_TYPE = $CONTENT_TYPE
```

```
echo CONTENT_LENGTH = $CONTENT_LENGTH

exit 0

# Document en ligne donnant des instructions brèves.
:<<-'_test_CGI_'

1) Placez ce script dans le répertoire http://domain.name/cgi-bin.
2) Puis, ouvrez http://domain.name/cgi-bin/test-cgi.sh.

_test_CGI_
```

Des volontaires ?

Annexe Q. Droits d'utilisation

Important

Le texte ci-dessous est la version française de la mise en garde de ce document. Seule la version originale de cette mise en garde, présentée dans la section suivante, fait foi.

Le *Guide avancé d'écriture des scripts Bash* est sous le copyright© 2000, de Mendel Cooper. L'auteur détient aussi le copyright sur les versions précédentes de ce document. [1]

Cet encart de copyright reconnaît et protège les droits de tous les contributeurs à ce document.

```
A.  La distribution de versions substantiellement
    modifiées de ce document est seulement autorisée sous les conditions suivantes :

A1. Le document modifié doit clairement indiqué qu'il s'agit d'un travail dérivé du
    Advanced Bash Scripting Guide original, et l'auteur original, Mendel Cooper, doit
    être indiqué comme auteur principal.

A2. Le document modifié ou dérivé doit clairement indiqué les portions de texte qui
    diffèrent ou qui dévient du document original. Un avertissement doit être
    présent, indiquant que l'auteur original n'assume pas forcément les
    modifications réalisées.

A3. Le document modifié ou dérivé doit être distribué sous cette licence et les
    droits de l'auteur original, si applicable, ne doivent pas être modifiés.

B.  Ce document, ou toute version modifiée, dérivée, ne peut PAS être distribué
    chiffré ou avec tout type de DRM (Digital Rights Management) ou de contrôle
    de contenu intégré. Ce document ou toute version dérivée ne peut pas non
    plus être livré avec d'autres travaux sous DRM.

C.  Si ce document (ou toute version précédente ou dérivée) est rendu disponible
    sur un site web ou FTP, alors le fichier doit être accessible à tout public.
    Aucun mot de passe ou autre type de restriction ne doit empêché son
    téléchargement.

D.  La distribution du travail original quelle que soit sa
    forme papier est interdite sans l'accord explicite du détenteur du copyright.

E.  Au cas où l'auteur ou le mainteneur de ce document ne peut pas être contacté,
    le Linux Documentation Project est autorisé à prendre possession du
    document et à nommer un nouveau mainteneur, qui aura donc le droit de mettre
    à jour et de modifier le document.
```

Sans *accords écrits et explicites* de l'auteur, les distributeurs et éditeurs (incluant les éditeurs en ligne) se voient interdits l'ajout de conditions supplémentaires sur ce document, sur les versions précédentes et sur les versions dérivées. Au moment de cette mise à jour, l'auteur garantit qu'il n'a *pas* d'obligations contractuelles qui modifieraient ces déclarations.

En résumé, vous pouvez distribuer librement ce livre ou toute version dérivée sous sa forme électronique.

Si vous affichez ou distribuez ce document, toute version précédente ou toute version dérivée, quelque soit la license sauf celle ci-dessus, alors il vous est demandé d'obtenir les droits écrits de l'auteur. Si ceci n'est pas fait, vos droits de distribution s'en verraient annulés.

De plus, l'exemption suivant s'applique :

```
En copiant ou en distribuant ce livre, vous perdez le
        droit d'utiliser ceci lors de poursuites contre la communauté OpenSource,
        ses développeurs ou tout autre logiciel ou documentation associé, ceci
        incluant par exemple le noyau Linux, OpenOffice, Samba et Wine, mais pas
        seulement eux. Vous perdez aussi le droit d'exercer en tant qu'expert
```

[1] L'auteur souhaite que ce livre soit dans le domaine public après une période de 14 ans, donc en 2014. Au début de la république américaine, c'était la durée accordée à un travail sous copyright.

```
       dans toute poursuite contre la communauté OpenSource, ses développeurs
       out tout logiciel ou documentation associé.
```

Ce sont des termes très libéraux et ils ne doivent pas empêcher une diffusion ou utilisation légitime de ce livre. L'auteur encourage particulièrement l'utilisation de ce livre en classe ou dans un but éducatif.

Note

Certains des scripts contenus dans ce document sont indiqués dans le domaine public. Ces scripts ne sont concernés ni par la licence ci-dessous ni par les restrictions du copyright.

Les droits d'impression commerciale de ce livre mais aussi les autres droits sont disponibles. Merci de contacter l'*auteur* si vous êtes intéressé.

L'auteur a créé ce livre dans l'esprit du *LDP Manifesto*.

Linux est une marque enregistrée par Linus Torvalds.

Fedora est une marque enregistrée par Red Hat.

UNIX et UNIX sont des marques enregistrées par l'Open Group.

MS Windows est une marque enregistrée par Microsoft Corp.

Solaris est une marque enregistrée par Sun, Inc.

OSX est une marque enregistrée par Apple, Inc.

Yahoo est une marque enregistrée par Yahoo, Inc.

Pentium est une marque enregistrée par Intel, Inc.

Thinkpad est une marque enregistrée par Lenovo, Inc.

Scrabble est une marque enregistrée par Hasbro, Inc.

Librie, PRS-500 et PRS-505 sont des marques enregistrées par Sony, Inc.

Toute autre marque commerciale mentionnée dans le corps de ce texte est enregistrée par son propriétaire.

Hyun Jin Cha a réalisé une *traduction koréenne* de la version 1.0.11 de ce livre. Des traductions espagnole, portuguaise, *française*, allemande, *italienne*, *russe*, *tchèque*, *chinoise*, indonésienne et hollandaise sont aussi disponibles ou en cours. Si vous souhaitez traduire ce document dans une autre langue, n'hésitez pas à le faire, suivant les termes indiqués ci-dessus. L'auteur souhaite être averti de tels efforts.

Annexe R. Copyright

The *Advanced Bash Scripting Guide* is copyright © 2000, by Mendel Cooper. The author also asserts copyright on all previous versions of this document. [1]

This blanket copyright recognizes and protects the rights of all contributors to this document.

This document may only be distributed subject to the terms and conditions set forth in the Open Publication License (version 1.0 or later), *http://www.opencontent.org/openpub/*. The following license options also apply.

```
A.  Distribution of substantively modified versions of this document
    is permitted only under the following provisions.

A1. The modified document must clearly indicate that it is derivative
    of the original Advanced Bash Scripting Guide, and the original
    author, Mendel Cooper, must be listed as the primary author.

A2. The modified or derivative document must clearly indicate which portions
    of the text differ or deviate from the original document. A notice must
    be present, stating that the original author does not necessarily
    endorse the changes to the original.

A3. The modified or derivative document must be distributed under this
    license, and the original author's copyright, as applicable, may
    not be modified.


B.  This document, or any modified or derivative version thereof, may
    NOT be distributed encrypted or with any form of DRM (Digital Rights
    Management) or content control mechanism embedded in it. Nor may this
    document or any derivative thereof be bundled with other DRM-ed works.

C.  If this document (or any previous version or derivative thereof)
    is made available on a Web or ftp site, then the file(s) must be
    publicly accessible. No password or other access restrictions to
    its download may be imposed.

D.  Distribution of the original work in any standard (paper) book form
    requires permission from the copyright holder.

E.  In the event that the author or maintainer of this document cannot
    be contacted, the Linux Documentation Project is authorized to
    take over custodianship of the document and name a new maintainer,
    who would then have the right to update and modify the document.
```

Without *explicit written permission* from the author, distributors and publishers (including on-line publishers) are prohibited from imposing any additional conditions, strictures, or provisions on this document, any previous versions, or any derivative versions. As of this update, the author asserts that he has *not* entered into any contractual obligations that would alter the foregoing declarations.

Essentially, you may freely distribute this book or any derivatives thereof in electronic form.

If you display or distribute this document, any previous versions thereof, or any derivatives thereof under any license except the one above, then you are required to obtain the author's written permission. Failure to do so may terminate your distribution rights.

Additionally, the following waiver applies:

```
By copying or distributing this book YOU WAIVE THE RIGHT
to use the materials within or any portion thereof in a patent or copyright
lawsuit against the Open Source community, its developers, or against any
of its associated software or documentation including, but not limited
to, the Linux kernel, Open Office, Samba, and Wine. You further waive
the right to use any of the materials within this book in testimony or
```

[1] The author intends that this book be released into the Public Domain after a period of 14 years, that is, in 2014. In the early years of the American republic this was the duration statutorily granted to a copyrighted work.

```
depositions as a plaintiff's "expert witness" in any lawsuit against the
Open Source community, any of its developers, or any of its associated
software or documentation.
```

These are very liberal terms, and they should not hinder any legitimate distribution or use of this book. The author especially encourages its use for classroom and instructional purposes.

Note

Certain of the scripts contained in this document are, where noted, in the Public Domain. These scripts are exempt from the foregoing license and copyright restrictions.

The commercial print and other rights to this book are available. Please contact *the author* if interested.

The author produced this book in a manner consistent with the spirit of the *LDP Manifesto*.

Linux is a trademark registered to Linus Torvalds.

Fedora is a trademark registered to Red Hat.

Unix and UNIX are trademarks registered to the Open Group.

MS Windows is a trademark registered to the Microsoft Corp.

Solaris is a trademark registered to Sun, Inc.

OSX is a trademark registered to Apple, Inc.

Yahoo is a trademark registered to Yahoo, Inc.

Pentium is a trademark registered to Intel, Inc.

Thinkpad is a trademark registered to Lenovo, Inc.

Scrabble is a trademark registered to Hasbro, Inc.

Librie, PRS-500, and PRS-505 are trademarks registered to Sony, Inc.

All other commercial trademarks mentioned in the body of this work are registered to their respective owners.

Hyun Jin Cha has done a *Korean translation* of version 1.0.11 of this book. Spanish, Portuguese, *French*, *(another French)*, German, *Italian*, *Russian*, *Czech*, *Chinese*, Indonesian, and Dutch translations are also available or in progress. If you wish to translate this document into another language, please feel free to do so, subject to the terms stated above. The author wishes to be notified of such efforts.

Annexe S. Table ASCII

Dans un livre de ce type, il est traditionnel d'avoir une annexe comprenant une table ASCII. Pas dans ce livre. À la place, voici un petit script shell qui génère une table ASCII complète et l'écrit dans le fichier `ASCII.txt`.

Exemple S.1. Un script qui génère une table ASCII

```
#!/bin/bash
# ascii.sh

# Script de Sebastian Arming.
# Légèrement modifié par l'auteur du guide ABS.
# Utilisé avec sa permission (merci !).

exec >ASCII.txt         #  Sauvegarder stdout dans un fichier,
                        #+ comme dans les scripts exemples
                        #+ reassign-stdout.sh et upperconv.sh.

MAXNUM=256
COLUMNS=5
OCT=8
OCTSQU=64
LITTLESPACE=-3
BIGSPACE=-5

i=1 # Compteur décimal
o=1 # Compteur octal

while [ "$i" -lt "$MAXNUM" ]; do
        paddi="    $i"
        echo -n "${paddi: $BIGSPACE}  "    # Espacement de colonnes.
        paddo="00$o"
        echo -ne "\\${paddo: $LITTLESPACE}"
        echo -n "     "
        if (( i % $COLUMNS == 0)); then    # Nouvelle ligne.
           echo
        fi
        ((i++, o++))
        # La notation octale pour 8 est 10 et pour 80 est 100.
        (( i % $OCT == 0))    && ((o+=2))
        (( i % $OCTSQU == 0)) && ((o+=20))
        # Nous ne devons pas compter après 0777.
done

exit 0
```

Annexe T. Index

Cet index / glossaire / référence liste un grand nombre des thèmes abordés dans ce livre. Les termes sont ordonnés de façon *approximative*, dans l'ordre ASCII, *réordonnés si nécessaire* pour une meilleure compréhension.

Notez que les *commandes* sont indexées dans la quatrième partie149.

* * *

^ (caret) Début de ligne, dans une Expression Rationnelle313

~ Tilde

- ~ répertoire personnel, correspond à `$HOME`
- ~/ Répertoire personnel de l'*utilisateur* courant
- ~+ Répertoire de travail *actuel*
- ~- Répertoire de travail *précédent*

= Signe *égal*

- = Opérateur d'Affectation de variable
- = Opérateur de comparaison de chaîne

  == Opérateur de comparaison de chaîne
- =~ Opérateur de correspondance470 dans une *Expression Rationnelle*

  Script d'exemple 593

< Signe inférieur

- Est inférieur à

  Comparaison de chaîne

  Comparaison d'entiers à l'intérieur de parenthèses doubles
- Redirection

  < `stdin`331

  << *Document en ligne*

  <<< *Chaîne en ligne*

  <> Ouvrir un fichier18 en lecture *et* écriture

> Signe supérieur

- Est plus grand que

  Comparaison de chaîne

  Comparaison d'entiers, à l'intérieur de *parenthèses doubles*
- Redirection

  > Redirige `stdout` vers un fichier

  >> Redirige `stdout` vers un fichier mais en *ajout*

  i>&j Redirige le *descripteur de fichier* `i`331 vers le *descripteur de fichier* `j`

  >&j Redirige `stdout`331 vers le *descripteur de fichier* `j`

  >&2 Redirige `stdout`18 d'une commande vers `stderr`

  2>&1 Redirige `stderr`331 vers `stdout`

&> Redirige `stdout` *et* `stderr`18 d'une commande vers un fichier

:> **`fichier`** Tronque le fichier pour qu'il ait une taille nulle

| Tube, un périphérique pour faire passer la sortie d'une commande à une autre commande ou au shell

|| Opérateur de test OU logique

- (dash)

- Préfixe d'un *paramètre par défaut*, dans une *substitution de paramètre*
- Préfixe d'*option*
- Indique la *redirection* de `stdin` ou `stdout`
- **--** (tiret double)
  Préfixe des options *longues*21
  Décrémentation *style C* d'une variable118 à l'intérieur d'une *parenthèse double*

; (point-virgule)

- En tant que séparateur de commandes
- **\;** Point-virgule *échappé*195, termine une commande find
- **;;** Double point-virgule, termine une option case
- Requis quand...
  le mot clé *do* se trouve sur la première ligne d'une boucle (*loop*)119
  termine un bloc de code entre *accolades* 421

: double-points, commande *null*, équivalent à la commande interne Bash true

- **:>** **`fichier`** Tronque le fichier pour qu'il ait une taille nulle

! Opérateur de négation, inverse le code de sortie44 d'un test ou d'une commande

- **!=** différent de Opérateur de comparaison de chaîne

? (point d'interrogation)

- Correspond à zéro ou un caractère314, dans une Expression Rationnelle Étendue
- Caractère *joker* simple, dans un remplacement
- Dans un opérateur trinaire style *C* 14

// Double slash, comportement de la commande cd

. (point)

- **.** Charge un fichier (dans un script), équivalent à la commande source
- **.** Correspond à un seul caractère313, dans une Expression Rationnelle313
- **.** Répertoire de travail actuel11
  ./ Répertoire de travail actuel
- **..** Répertoire *Parent* 11

' ... ' (guillemets simples) guillemets *forts*

" ... " (guillemets doubles) guillemets *faibles* 28

() Parenthèses

- **(...)** Groupe de commandes ; lance un sous-shell342
- **(...)** Ferme un groupe315 d'*Expressions Rationnelles Étendues*
- **>(...)**

  <(...) Substitution de processus349
- **...)** Termine la condition d'un test134 dans une construction *case*
- **((...))** Double parenthèses117, dans une expansion arithmétique

[Crochet gauche, construction de *test*

Crochets **[ ]**

- Élément d'un *tableau*375
- Ferme un ensemble de caractères de correspondance313 dans une *Expression Rationnelle*
- Construction de *test*

[[...]] Double crochets50, construction étendue pour *test*

$ *ancre*313, dans une Expression Rationnelle313

$ Préfixe pour un nom de variable

$(...) Substitution de commande31, configurer une variable avec la sortie d'une commande, en utilisant la notation des parenthèses

` ... ` Substitution de commande141, en utilisant la notation des guillemets inversés

${ ... } Manipulation / évaluation de variables

- **${var}** Valeur d'une variable
- **${#var}** Longueur d'une variable
- **${#@}**

  ${#*} Nombre de *paramètres de position*98
- **${parametre?msg_err}** Message pour paramètre non initialisé
- **${parametre-defaut}**

  ${parametre:-defaut}

  ${parametre=defaut}

  ${parametre:=defaut} Configure la valeur par défaut de ce paramètre
- **${parametre+valeur_autre}**

  ${parametre:+valeur_autre}

  Autre valeur d'un paramètre si initialisé
- **${!var}**

  Référence indirecte d'une variable, nouvelle notation
- **${!varprefix*}**

  ${!varprefix@}

  Établit une correspondance des *noms* de toutes les variables précédemment déclarées commençant avec `varprefix`
- **${chaine:position}**

  ${chaine:position:longueur} Extraction de la sous-chaîne
- **${var#Modele}**

  ${var##Modele} Suppression de la sous-chaîne

- **${var%Modele}**
  ${var%%Modele} Suppression de la sous-chaîne
- **${chaine/souschaine/remplacement}**
  ${chaine//souschaine/remplacement}
  ${chaine/#souschaine/remplacement}
  ${chaine/%souschaine/remplacement} Remplacement de la sous-chaîne

\ Échappe39 le caractère suivant

- **\< ... \>** Signes inférieur et supérieur314, *échappés*, limite d'un mot dans une Expression Rationnelle313
- **\{ N \}** « Accolades », *échappés*, nombre d'ensemble de caractères dans une Expression Rationnelle Étendue
- **\;** *Point-virgule*195, *échappé*, termine une commande find
- **\$$** Référence indirecte d'une variable, ancien style de notation
- Échappement d'un *retour à la ligne*42 pour écrire une commande sur plusieurs lignes

&

- **&>** Redirige les sorties `stdout` *et* `stderr`18 d'une commande dans un fichier
- **>&j** Redirige la sortie `stdout`331 vers le *descripteur de fichier j*
  >&2 Redirige la sortie `stdout`18 d'une commande vers `stderr`
- **i>&j** Redirige le *descripteur de fichier*331 *i* vers le *descripteur de fichier j*
  2>&1 Redirige la sortie `stderr`331 vers `stdout`
- Fermer les *descripteurs de fichier*
  n<&- Ferme le descripteur de fichier *n* en entrée
  0<&-, **<&-** Ferme `stdin`
  n>&- Ferme le descripteur de fichier *n* en sortie
  1>&-, **>&-** Ferme `stdout`
- **&&** Opérateur de test ET logique
- **Command &** Exécute un processus en *tâche de fond*

dièse, symbole spécial débutant un *commentaire* dans un script

#! Sha-bang6, chaîne spécial indiquant le début d'un script shell

***** Astérisque

- *Joker*, dans un remplacement
- N'importe quel nombre de caractères13 dans une Expression Rationnelle313
- ****** Opérateur arithmétique Exponentiel

% Pourcentage

- Modulo, Opération arithmétique obtenant le reste de la division
- Opérateur (comparaison de modèle) pour la suppression d'une sous-chaîne

+ Signe plus

- *Correspondance d'un caractère*315, dans une Expression Rationnelle Étendue
- Préfixe d'un *paramètre alternatif* dans une *substitution de paramètre*

- **++** Incrément *style C* d'une variable118, dans des *parenthèses doubles*

* * *

Variables du shell

$_ Dernier argument de la commande précédente

$- Options passées au script, en utilisant set

$! *PID* du dernier job en tâche de fond

$? *Code de sortie* d'une commande44

$@ Tous les *paramètres de position*, en tant que mots *séparés*

$* Tous les *paramètres de position*, en un *seul* mot

$$ PID du script

$# Nombre d'arguments passés à la fonction352 ou au script lui-même

$0 Nom de fichier du script33

$1 Premier argument passé au script

$9 Neuvième argument passé au script

Table des *variables shell*

* * * * * *

-a ET logique tes de comparaison

Guide avancé d'écriture des scripts Bash, où le télécharger2

Alias370

- Supprimer un *alias*371 en utilisant *unalias*

Anagramme

Liste *Et*

- Pour fournir un argument par défaut en ligne de commande373

Signes inférieur et supérieur314, *échappés*, \< **. . .** \> limite de mot dans une Expression Rationnelle313

Document en ligne anonyme324 en utilisant **:**

Archivage

- rpm
- tar

Expansion arithmétique147

- variations de

Opérateurs arithmétiques

- Opérateurs de combinaison, style *C*
  += -= *= /= %=

Note

Dans certains contextes471, **+=** peut aussi fonctionner comme un opérateur de *concaténation de chaîne*.

Tableaux375

- Notation entre crochets375
- Concaténation384, *script d'exemple*
- Copie382
- Déclaration
  `declare -a nom_tableau`
- Tableaux embarqués389
- Tableaux vides, éléments vides380, *script d'exemple*
- Références indirectes389
- Initialisation376
  `tableau=( element1 element2 ... elementN)`
  Script d'exemple 383
  En utilisant la substitution de commandes
- Charger un fichier dans un tableau
- Multidimensionnel396, simulation
- Embarqué389
- Notation et usage375
- Nombre d'éléments dans380
  `${#nom_tableau[@]}`
  `${#nom_tableau[*]}`
- Opérations377
- Passer un *tableau* à une fonction
- En tant que *valeur de retour* provenant d'une fonction458
- Propriétés spéciales379, *script d'exemple*
- Opérations sur les chaînes378, *script d'exemple*
- *unset* supprime les éléments d'un tableau379

Touches de direction165, détection

Table ASCII

awk634 langage de traitement de texte orienté champs

- `rand()`117, fonction de hasard
- Manipulation de chaîne93
- Utiliser *export*177 pour passer une variable à un script *awk* embarqué

* * *

Guillemets inverses, utilisés dans une substitution de commande141

Conversion de base263, *script d'exemple*

Bash3

- Mauvaises pratiques421
- Les bases610, *exemple de script*
- Options en ligne de commande641
  Table

- Fonctionnalités manquantes au shell *Bourne* classique464
- Variables internes
- Version 2465
- Version 3469

.bashrc

$BASH_SUBSHELL

Commandes basiques, externes

Traitement en flot

bc, outil de calcul

- Dans un *document en ligne*264
- Modèle261 pour calculer une variable de script

Bibliographie

Outil Bison228

Opérateurs de bit

Périphériques blocs53, tests

Blocs de code

- Redirection15

  Exemple de script 16 : rediriger la sortie d'un bloc de code

Expansion des accolades

- Étendu, *{a..z}*

Brackets, **[ ]**

- Élément d'un *tableau* 375
- Enferme un ensemble de caractères pour une correspondance313 dans une *Expression Rationnelle*
- Construction de *test*

Crochets, *accolades*, **{}**, utilisées dans

- Bloc de code
- *find*195
- *Expressions Régulières Étendues*
- *Paramètres de position*
- *xargs*199

break commande de contrôle d'une *boucle*

- Paramètre131 (optionel)

Intégrés159 dans *Bash*

- Ne pas créer un sous-processus159

* * *

Construction *case*

- Gestion des paramètres en ligne de commande135,
- Remplacement136, filtrer des chaînes avec

cat, con*cat*énation de fichier(s)

- Abus de452
- Scripts *cat* 319
- Moins efficace que la redirection `stdin`191
- Envoie de la sortie de168, vers un read
- Utilisation de

Périphériques caractères53, tests

Processus fils

Deux-points, **:** , équivalent à la commande interne true de Bash

Scripts colorisés440

- **Table** de séquences d'échappement pour les couleurs
- Modèle, texte coloré sur fond coloré

Opérateur virgule, lie des commands ou operations

Options en ligne de commande

Substitution de commandes141

- **$(...)**145, notation préférée
- *Guillemets inverses*141
- Étendre les outils de *Bash*144
- Crée un *sous-shell*141
- Intégration145
- Supprime les nouvelles lignes142
- Configurer une variable à partir de la sortie d'une boucle144
- Séparation de mots141

En-tête de commentaires, but spécial

Commenter des blocs de code

- En utilisant un document en ligne *anonyme*
- En utilisant une construction *if-then*

Opérateurs de comparaison composée

Outils de compression

- bzip2
- compress
- gzip
- zip

continue commande de contrôle de boucle
Caractères de contrôle

- Ctrl-C25, *arrêt*
- Ctrl-D, fin / déconnexion / suppression
- Control-G, **BEL** (*beep*)
- Ctrl-H25, *redémarrage*
- Control-J, *retour à la ligne*
- Ctrl-M26, retour chariot

cron, *démon* d'exécution planifiée
Syntaxe style *C* pour la gestion de variables
Solutionneur de mots croisés
Accolades {}

- dans la commande *find*195
- dans une *Expression Rationnelle Étendue*
- dans *xargs*199

* * *
Démons, pour un système d'exploitation type UNIX
date
dc, outil de calculs
dd, commande de *duplication de données*

- Conversions
- Copie de données277 vers ou en provenance de périphériques
- Suppression *sécurisée* de fichiers277,
- Capture des touches frappées276
- Options275
- Accès aléatoire277 à un flux de données
- Fichiers swap277, initialisation
- Thread sur *www.linuxquestions.org*

Déboguer des scripts

- Outils411
- *Récupération des erreurs* à la sortie413
- *Récupération des signaux*

Nombres décimaux67, Bash interprète les nombres en tant que
Commande intégrée declare102

- options

Paramètres par défaut
Répertoire `/dev`399

- Pseudo-fichier périphérique `/dev/null`
- Pseudo-fichier périphérique `/dev/urandom`, utilisé pour générer des nombres pseudo-aléatoires
- Pseudo-fichier périphérique `/dev/zero`

*dialog*462, outil pour générer des boîtes de *dialog*ue dans un script
`$DIRSTACK` *pile de répertoire*
Commandes désactivées, dans des *shells restreints*
Mot-clé do119, commence l'exécution des commandes à l'intérieur d'une boucle119
Mot-clé done119, termine une boucle
Fichiers batch *DOS* 659, à convertir en scripts shell
Commandes *DOS* 660, équivalents UNIX (**table**)
fichiers dor, fichiers de configuration « cachés »
Double crochets50 **[[...]]** test46 construct

- and évaluation de constantes *octales/hexadécimales*51

Double quotes28 " **...** " mise entre guillemets *faible*
Doubler les espaces d'un fichier633, en utilisant sed631
* * *
-e Test d'existence d'un fichier
echo

- Envoyer des données aux commandes via un *tube*
- Initialiser une variable160 en utilisant la substitution de commandes141
- `/bin/echo`161, commande *echo* externe

elif, Contraction de *else* et *if*
esac, mot-clé terminant une construction *case*
Variables d'*environnement*
-eq , *est égal à* test de comparaison d'entiers
Siège d'Eratosthène390, algorithme pour générer des nombres premiers
Caractères d'échappement, significations spéciales
`$EUID`, *ID réel de l'utilisateur*
eval, Combine et *évalue* le(s) expression(s), avec expansion de la variable

- Effets de171, *Script d'exemple*
- Force la *ré-évaluation*171 des arguments
- Rique d'utilisation173

Évaluation de constantes *octales/hexadécimales* à l'intérieur de [[...]]51
Commande exec333, utiliser dans la redirection
Exercices
Sortie et code de sortie

- Commande exit44
- Statut de sortie44 (*code de sortie, return* statut d'une commande)

Table, *Codes de sortie* ayant une signification spéciale
En dehors de l'échelle
Spécifié lors *du retour d'une fonction*
*Avec succès*44, **0**

Export177, pour rendre disponible les variables auprès des processus fils

- Passer une variable à un script *awk* embarqué 177

expr, Évaluation d'*expression*

- Extraction de sous-chaînes
- *index* d'une sous-chaîne (position numérique dans la chaîne)
- Correspondance de sous-chaîne

Expressions Rationnelles Étendues

- **?** (point d'interrogation) Correspond à aucun caractère ou à un seul 314
- **(...)** Groupe d'expressions315
- **\{ N \}** « Accolades », *échappées*, nombre d'ensembles de caractères à correspondre
- **+** *Correspondance de caractères* 315

* * *

false, renvoit un code de sortie44 *d'erreur* (1)
Descripteurs de fichiers330

- Fermeture
  n<&- Fermer le descripteur de fichier *n* en entrée
  0<&-, **<&-** Fermer `stdin`
  n>&- Fermer le descripteur de fichier *n* en sortie
  1>&-, **>&-** Fermer `stdout`
- Descripteur de fichier en *C*, similarité

find

- **{}** Accolades 195
- **\;** Point-virgule *échappé* semicolon195

Filtre

- En utilisant - comme outil de traitement de fichier filtrant22
- Envoyer la sortie d'un filtre au *même* filtre

Nombres à virgule flottante, Bash ne les reconnaît pas
fold, un filtre pour fermer les lignes de texte
Créer un processus *fils*
Boucles *for*
Fonctions352

- Arguments passés355 référencés par position
- Capturer le code de retour361 d'une fonction en utilisant echo
- La définition doit précéder353 le premier appel à la fonction
- Code de sortie
- Variables locales
  et récursion
- Passer un *tableau* à une fonction
- Passer des pointeurs356 à une fonction
- Récursion365
- Rediriger le `stdin` d'une fonction
- return358
  Renvoyer un *tableau*458 à partir d'une fonction
  limites de *return* 457, contournements
- Arguments d'une fonction passés à *shift*

* * *

getopt, commande *externe* pour analyser les arguments en *ligne de commande*

- Émulé dans un script92

getopts, *Commande interne* Bash pour analyser les arguments en *ligne de commande* d'un script

- `$OPTIND` / `$OPTARG`178

Remplacement316, expansion de noms de fichiers
-ge , *plus grand ou égal à* test de comparaison d'entiers
-gt , *plus grand que* test de comparaison d'entiers
`$GROUPS`, *Groupes* auxquels l'utilisateur appartient
gzip, outil de compression
* * *
Hachage, créer des index de recherche dans une table

- *Script d'exemple*517

head, *echo* sur `stdout` les lignes composant le début d'un fichier texte
help, donne un résumé sur l'utilisation d'une commande interne159 Bash
Documents *en ligne* 318

- Documents en ligne *anonyme*324, en utilisant `:`
  des blocs de code en commentaire
  des scripts auto-documentés325
- *bc* dans un *document en ligne*264
- des scripts *cat* 319
- de la substitution de commandes 323
- des scripts *ex* 319
- *Fonction*323, fournir une entrée à

- Chaînes *en ligne* 327
  Ajouter du texte en début327
  En utilisant *read*327
- Chaîne *limite* 318
  La *chaîne limite* fermante pourrait ne pas être indentée
  Option tiret320 pour limiter une chaîne, `<<-ChaîneLimite`
- Affichage de texte littéral322, pour générer du code
- Substitution de paramètres321
  Désactiver322 la *substitution de paramètres*
- Passer des paramètres320
- Fichiers temporaires
- Utiliser *vi* de façon non interactive318

Historique des commandes

`$HOME`, *répertoire personnel de l'utilisateur*

Résout des problèmes

`$HOSTNAME`, *nom d'hôte* du système

* * *

construction de *test* if [condition]; then ...46

- if-grep, *if* et grep combinés
  Correction pour un test461 *if-grep*

`$IFS`, Variable *Séparateur interne de champs*

- Par défaut des *espaces blancs*72

Opérateurs de comparaison d'entiers

Mot clé in119, précédant une `[liste]` dans une boucle *for*

Initialisation de table299, `/etc/inittab`

Groupe en ligne, par exemple un bloc de code

Script interactif, test pour

Redirection d'entrée/sortie

Référencement indirect de variables

- Nouvelle notation, introduite dans la version 2 de Bash
- Script d'exemple

Itération

* * *

ID de job188, table

jot, Émet une séquence d'entiers. Équivalent à seq.

- Génération d'une séquence aléatoire272

* * *

Mots-clés159

kill, termine le processus qui a cet ID

- Options (`-l`, `-9`)

killall, termine un processus *par nom*

Script killall 310 dans `/etc/rc.d/init.d`

* * *

-le , *plus petit ou égal à* test de comparaison d'entiers

let, paramétrer et exécuter des opérations arithmétiques sur des variables

Chaîne limite318, dans un document en ligne318

`$LINENO`, variable indiquant le *numéro de ligne* où il apparaît dans un script

Link, file (using *ln* command)

- Appeler un script avec plusieurs noms194, en utilisant *ln*
- Liens *symboliques* 194, *ln -s*

Constructions de liste372

- Liste *Et*
- Liste *Ou*

Variables locales

- et récursion

Traduction

Opérateurs logiques (`&&`, `||`, etc.)

Fichier de déconnexion, `~/.bash_logout` file

Boucles

- break commande de contrôle de boucle
- Commande de contrôle de boucle
  continue
- Boucle style *C* à l'intérieur de parenthèses doubles117

  Boucle *for*125

  Boucle *while*128
- do119 (mot clé), commence l'exécution de commandes dans une boucle
- done119 (mot clé), termine une boucle
- Boucles *for*

  for `arg` *in* `[liste]`; *do*

  Substitution de commande pour générer une `[liste]`122

  Expansion de noms de fichier dans une `[liste]`121

  Multiple paramètres dans chaque élément d'une `[liste]`120

  Omettre une `[liste]`121, par défaut des paramètres de position

  Paramétrer une `[liste]`120

  Redirection124

- in119, (mot clé) précédant [liste] dans boucle *for*
- Boucles imbriquées
- Exécuter une boucle *en tâche de fond*19, *exemple de script*
- Point-virgule requis, quand *do* est sur la première ligne d'une boucle

  Boucle *for* 119

  Boucle *while* 127
- Boucle until

  `until [ condition-est-vraie ]; do`
- Boucle while

  `while [ condition ]; do`

  Appel de fonction128 dans des crochets de test

  Conditions multiple127

  Omettre les *crochets de test*

  Redirection129

  Construction *while read*129
- Quel type de boucles utiliser130

Périphériques loopback

- Dans le répertoire `/dev`399
- Monter une image ISO302

-lt , test de comparaison d'entiers *plus petit que*

* * *

m4, langage de macros

`$MACHTYPE`, *Type de machine*

Nombre magique, marqueur placé au début d'un fichier indiquant son type

`Makefile`244, fichier contenant la liste des dépendances utilisée par la commande make

Signification méta

Modulo, Opérateur *reste* arithmétique

- Application: Générer les nombres premiers497

Calculs d'échéances261, *script exemple*

* * *

-n Test de Chaîne non *nulle*

Tube nommé274, un tampon FIFO temporaire

- *Script exemple*496

nc, *netcat*, un ensemble d'outils pour les ports TCP et UDP

-ne, test de comparaison d'entiers *différent de*

Opérateur de négation, **!**, inverse le sens d'un test46

netstat, Outil statistique réseau

nl, un filtre pour compter le nombre de lignes

Noclobber, Option `-C` de Bash pour empêcher l'écrasement de fichiers
Affectation de *null* à une variable, à éviter
* * *
-o Test de comparaison composé OU logique
od, *dump octal*
`$OLDPWD` Précédent répertoire de travail
Opérateur

- Définition
- Précédence432

Options419, passées au shell, à un script sur la ligne de commande ou via la commande set
Liste *Ou*
Opérateur logique Ou, ||
* * *
Substitution de paramètres

- *${paramètre+autre_valeur}*
  ${paramètre:+autre_valeur}
  Autre valeur d'un paramètre, si initialisée
- *${paramètre-valeur_par_défaut}*
  ${paramètre:-valeur_par_défaut}
  ${paramètre=valeur_par_défaut}
  ${paramètre:=valeur_par_défaut}
  Default parameters
- *${!prefixe_var*}*
  ${!prefixe_var@}
  Correspondance du *nom* du paramètre
- *${paramètre?message_erreur}*
  Message si paramètre non initialisé
- *${paramètre}*
  Valeur d'un *paramètre*
- *Script exemple*495
- **Table** des *substitution de paramètres*

Problème de processus père / fils 424, un processus *fils* ne peut exporter des variables vers un processus père
Parenthèses

- Groupe de commandes
- Fermer un groupe315 d'*Expressions Rationnelles Étendues*
- Double parenthèses117, dans une expansion arithmétique

`$PATH`, le *chemin* (emplacement des outils systèmes)
Perl437, langage de programmation

- Intégré437 dans le même fichier que le script *Bash*
- Intégré437 dans un script *Bash*
- Utiliser *eval* avec173

Anagramme du type *Perquackey* (script *Quackey*)

Pétales autour de la rose

PID, ID du processus (acronyme de *Process ID*), un numéro d'identifiant affecté à un processus en cours d'exécution.

Tube, | , un périphérique pour passer la sortie d'une commande à une autre commande ou au shell

- Éviter les commandes inutiles452 dans un *tube*
- Pipefail470, *set -o pipefail* option indiquant le code de sortie44 à l'intérieur d'un *tube*
- `$PIPESTATUS`, *code de sortie* du dernier tube
- Envoyer la sortie d'une commande19 à un script
- Rediriger `stdin`191, plutôt qu'utiliser cat avec un *tube*

Astuces

- **-** (le tiret) n'est *pas* un opérateur de redirection
- **//** (double slash), comportement de la commande cd
- #!/bin/sh l'en-tête de script désactive les fonctionnalités *Bash* étendues464
- Abus de *cat*452
- Programmation *CGI* , en utilisant des scripts
- Fermer une *chaîne limite* dans un *document en ligne* indenté
- Les retours chariot style DOS (\r\n) arrêtent brutalement un script
- eval173, risque à l'utilisation
- Droits d'exécution manquants pour les commandes à l'intérieur d'un script
- Problème d'*export* 424, processus *fils* vers processus *père*
- Fonctionnalités *Bash* étendues non disponibles
- Échec pour *mettre entre guillemets* des variables à l'intérieur d'un *test* entre crochets
- Ensemble de commandes *GNU*, dans des scripts cross-platformes
- Plusieurs instructions echo 424 dans une fonction dont la sortie est capturée457
- Affectation de *null* à une variable
- Les Opérateurs de comparaison d'entiers et de chaînes ne sont *pas* équivalents
  = et **-eq** ne sont *pas* interchangeables
- Omettre le *point-virgule* finale421, dans un bloc de code *entre accolades*
- Envoyer dans un tube
  echo vers une boucle426
  echo vers la commande *read*425
  tail `-f` vers *grep*427
- Préserver les *espaces blancs* dans une variable, conséquences inattendues38
- Les commandes *suid* à l'intérieur d'un script
- Fonctionnalités *Bash* non documentées, danger des
- Variables non initialisées

- Noms de variable, non appropriés
- Variables dans un *sous-shell*, *étendue* limitée
- Espaces blancs421, mauvaise utilisation des

Pointeurs

- et descripteurs de fichiers
- et fonctions356
- et *références indirectes*105
- et *variables*28

Problèmes de portabilité dans l'écriture de scripts shell

- Configurer le *PATH* et *umask*
- Utilisation de *whatis*

Paramètres de position

- `$@`, pour les mots *séparés*
- `$*`, comme mot *simple*

POSIX, *Interface Portable pour les Systèmes d'ExplOitation/ UNIX*

- Classes de caractères

`$PPID`, *ID du processus* père

Précédence432, opérateur

*Ajouter*454 des lignes au début d'un fichier at head of a file, *script exemple*

Nombres premiers

- Générer des nombres premiers sans utiliser de tableaux497
- Siège d'Eratosthène390

printf, commande *print avec formatage*

Répertoire `/proc`400

- Exécuter des processus403, ficheirs décrivant
- Écrire des fichiers de `/proc`, *avertissement*

Processus14

- ID du processus (PID)

Substitution de processus349

- Pour comparer le contenu de répertoires 349
- Pour fournir le `stdin` d'une commande349
- Modèle

Prompt

- `$PS1`, *Invite principale*, vue sur la ligne de commande

- `$PS2`, Invite secondaire

Pseudo-code453, comme méthode de résolution de problèmes

`$PWD`, Répertoire de travail courant

* * *

Quackey, un anagramme de type *Perquackey* (script)

Point d'interrogation, **?**

- Correspondance sur un caractère314 dans une *Expression Rationnelle Étendue*
- Caractère *joker* dans un remplacement
- Dans un opérateur *C* trinaire14

Entre guillemets37

- Chaîne de caractères37
- Variables
  à l'intérieur de crochets de *test*
- *Espace blanc*37, en utilisant les *guillemets* pour le préserver

* * *

Nombres aléatoires

- `/dev/urandom`
- `rand()`117, fonction aléatoire dans awk634
- `$RANDOM`107, fonction Bash qui renvoit un entier pseudo-aléatoire
- Génération de séquences aléatoires206 en utilisant la commande date
- Génération de séquences aléatoires272 en utilisant jot
- Chaîne aléatoire89, génération

rcs461

read, initialise la valeur d'une variable à partir de `stdin`638

- Détecter des touches de *direction*165
- Options164
- Envoyer la sortie de *cat*168 vers *read*
- « Ajouter au début » d'un texte327
- Problèmes en envoyant la sortie de *echo*425 vers *read* via un tube
- Redirection d'un fichier166 vers *read*
- `$REPLY`, variable *read* par défaut
- Saisie avec délai 166
- Construction *while read*129

Récursion365

- Démonstration de365
- Factorielle366
- Séquence de Fibonacci367

- Variables locales
- Script s'appelant lui-même récursivement438
- Tours d'Hanoi368

Redirection

- Blocs de code 336
- exec <`nom_fichier`333,
  pour réaffecter les descripteurs de fichier330
- Explications d'introduction de la *redirection des entrées/sorties*
- Ouvrir un fichier331 pour la lecture *et* l'écriture
  `<>nom_fichier`
- Saisie redirigé vers *read* 166 à partir d'un fichier
- `stderr` vers `stdout`331
  `2>&1`
- `stdin` / `stdout`22, en utilisant `-`
- `stdin` d'une *fonction*
- `stdout` vers un fichier
  `>` ... `>>`
- `stdout` vers le *descripteur de fichier*331 *j*
  `>&j`
- descripteur de fichier`i` vers *descripteur de fichier*331 *j*
  `i>&j`
- `stdout` d'une commande18 vers `stderr`
  `>&2`
- `stdout` *et* `stderr` d'une commande18 vers un fichier
  `&>`
- tee, redirection de la sortie d'une commande vers un fichier en utilisant un tube

Cartes de référence

- Constructions diverses
- Substitution/expansion de paramètres
- Variables spéciales du shell
- Opérations sur les chaînes
- Opérateurs de test
  Comapraison de binaires
  Fichiers

*Expressions Rationnelles*313

- ^ Début-de-ligne
- **$** (dollar) *Ancre*313
- **.** (point) Correspond à un seul caractère 313

- ***** (astérisque) Tout nombre de caractères 13
- **[ ]** (crochets) Enferme un ensemble de caractères dont la correspondance est nécessaire313
- \ (antislash) Échappement314, interprète un caractère suivant litérallement
- \< **...** \> (inférieur/supérieur, *échappé*) Limite de mot314
- Expressions Rationnelles Étendues
  + *Correspondance d'un caractère*315
  \{ \} « Accolades » échappées
  [: :] Classes de caractères POSIX

$REPLY, Valeur par défaut associé avec la commande read
Shell restreint, shell (ou script) avec certaines commandes désactivées
return358, commande qui termine une fonction352
run-parts

- Exécuter des scripts en séquence, sans intervention de l'utilisateur

* * *

Étendue344 d'une variable, définition
Options d'un script419, initialisées sur la ligne de commande
Routines d'un script, bibliothèques de définitions et de fonctions352 utiles
Invite secondaire, **$PS2**
Problèmes de sécurité

- nmap, *cartographeur de réseau* / scanner de ports
- sudo
- *suid* commandes à l'intérieur d'un script
- Virus, troyens, et vers463 dans des scripts
- Écriture de scripts sécurisés463

sed631, langage de programmation basé sur des modèles

- **Table**, Opérateurs de base
- **Table**, Exemple d'opérateurs

select, construction pour la génération de menu

- **`in liste`** omise139

Sémaphore
Point-virgule requis119, lorsque le *mot clé* do119 est sur la première ligne d'une boucle

- À la fin d'un bloc de code entre *accolades* 421

seq, Génère une séquence d'entiers. Équivalent à jot.
set, Modifie la valeur des variables internes de scripts
Script shell , définition d'un
Emballage shell434, script intégrant une commande ou un outil
shift35, raffecter les *paramètres de position*

`$SHLVL`, *niveau de shell*, profondeur d'imbrication du shell (ou du script)
shopt, modifie les *options du shell*
Signal, un message envoyé à un processus
Simulations

- Brownian motion110
- Carte de Galton110
- Course de chevaux446
- *Vie*487, jeu de
- PI266, approximation en lançant des balles de canon
- *Pile* pushdown393

Guillemets simples (' **...** ') guillemets37 *forts*
Socket399, un noeud de communication associé à un port d'entrée/sortie
Tri

- Algorithme « Bubble sort »387
- Tri d'insertion 594

source, exécute un script ou, à l'intérieur d'un script, import un fichier

- Passer des paramètres de position181

Spam, gestion du

- *Script d'exemple*246
- *Script d'exemple*247
- *Script d'exemple*529
- *Script d'exemple*561

Caractères spéciaux
Pile, émuler une pule de type « push-down », *Script d'exemple*393
Fichiers de démarrage, Bash
`stdin` et `stdout`638
Chaînes

- =~ Opérateur de correspondance de chaîne470
- Comparaison
- Longueur
  `${#string}`
- Manipulation
- Manipulation93, en utilisant awk634
- Protéger des chaînes525 à partir de l'expansion et/ou de la réinterprétation, *script exemple*
  Déprotéger des chaînes527, *script exemple*
- *strchr()*, équivalent de
- *strlen()*, équivalent de87

- Extraction de sous-chaînes
  ${chaine:position}
  ${chaine:position:longueur}
  En utilisant *expr*
- *index* d'une sous-chaîne (position numérique dans la chaîne)
- *Correspondance* de sous-chaîne, en utilisant expr
- *Suppression* de sous-chaînes
  ${var#Modele}98
  ${var##Modele}98
  ${var%Modele}99
  ${var%%Modele}99
- Remplacement de sous-chaînes
  ${chaine/souschaine/remplacement}
  ${chaine//souschaine/remplacement}
  ${chaine/#souschaine/remplacement}
  ${chaine/%souschaine/remplacement}
  *Script exemple*482
- **Table** des opérateurs de manipulation et extraction de *chaîne/sous-chaîne*

Guillemets *forst* ' **...** '
Feuille de style pour l'écriture de scripts
Sous-shell342

- Liste de commandes entre parenthèses
- Variables, `$BASH_SUBSHELL` et `$SHLVL`
- Variables dans un *sous-shell*
  étendue limitée mais ...
  ... peut être accédées en dehors du sous-shell ?461

su *Substitue l'utilisateur*, se connecte en tant qu'un autre utilisateur ou en tant que *root*
suid53 (*set user id*)

- commandes *suid* à l'intérieur d'un script, non recommandé

Liens symboliques194
Fichiers swap406
* * *
tail, *echo* vers `stdout` des lignes de la fin d'un fichier texte
tar, outil d'archivage
tee, rediriger la sortie d'une commande vers un fichier via un tube
Terminaux

- setserial
- setterm

- stty
- wall

Commande *test*

- *commande intégrée* Bash49
- commande externe49, `/usr/bin/test` (équivalent à `/usr/bin/[`)

Construction de tests

Opérateurs de tests

- **-a** comparaison ET logique
- **-e** Le fichier existe
- **-eq** est-égal-à (comparaison d'entiers)
- **-f** Fichier *standard*
- **-ge** plus-grand-ou-égal-à (comparaison d'entiers)
- **-gt** plus-grand (comparaison d'entiers)
- **-le** plus-petit-ou-égal (comparaison d'entiers)
- **-lt** plus-petit (comparaison d'entiers)
- **-n** longueur-non-nulle (comparaison de chaîne)
- **-ne** différent-de (comparaison d'entiers)
- **-o** comparaison OU logique
- **-u** test du drapeau *suid* 53
- **-z** taille-nulle (comparaison de chaîne)
- = est-égal-à (comparaison de chaîne)
  == est-égal-à (comparaison de chaîne)
- < plus-petit (comparaison de chaîne)
- < plus-petit, (comparaison d'entiers, à l'intérieur de parenthèses doubles)
- <= plus-petit-ou-égal, (comparaison d'entiers, à l'intérieur de *parenthèses doubles*)
- > plus-grand (comparaison de chaîne)
- > plus-grand, (comparaison d'entiers, à l'intérieur de *parenthèses doubles*)
- >= plus-grand-ou-égal, (comparaison d'entiers, à l'intérieur de *parenthèses doubles*)
- || OU logique
- **&&** ET logique
- **!** Opérateur de négation, inverse le code de sortie44 d'un test
  != différent-de (comparaison de chaîne)
- **Tables** d'opérateurs de *test*
  Comparaison binaire
  Fichier

Saisie avec délai

- En utilisant *read -t*166
- En utilisant *stty*79

- En utilisant une boucle chronométrée78
- En utilisant $TMOUT

Conseils et astuces pour les scripts Bash
who, informations sur les utilisateurs connectés

- w
- whoami283
- logname

- Tableau, en tant *valeur de retour* à partir d'une fonction458
- Capturer la valeur en sortie361 d'une fonction, en utilisant *echo*
- Blocs de commentaire
  En utilisant les *documents en ligne anonymes*
  En utilisant les constructions *if-then*
- En-tête de commentaires, but spécial
- Syntaxe style *C* pour manipuler des variables
- Double espacement d'un fichier texte633
- Noms de fichiers préfixés avec un tiret, suppression
- Filtrer, envoyer la sortie au *même* filtre
- Contournement de la valeur de *retour* 457 d'une fonction
- Correction du test *if-grep*461
- Bibliothèques de définitions et de *fonctions* utiles
- Affection de *null* à des variables, éviter de
- Passer un *tableau* à une fonction
- *Ajouter*454 des lignes en en-tête d'un fichier
- Pseudo-code453
- Script en tant que commande embarquée
- rcs461
- Exécuter des scripts en séquence sans intervention de l'utilisateur en utilisant run-parts
- *Portabilité* des scripts
  Configurer le *path* et *umask*
  En utilisant *whatis*
- Initialiser la variable d'un script à partir d'un bloc de code embarqué *sed* ou *awk*
- Variable du sous-shell, accéder à l'extérieur du sous-shell461
- Test d'une variable pour vérifier qu'elle ne contient que des entiers
- Tracer l'utilisation d'un script
- Widgets, appeler à partir d'un script

$TMOUT, Délai
tr, filtre de traduction des caractères

- conversion de fichiers texte DOS vers Unix224

- Options222
- Soundex485, *script d'exemple*
- Variants

Trap, spécifie une action à réaliser à la réception d'un signal
Opérateur trinaire14, style *C*
true, renvoit le code de sortie44 de *succès* (0)
commande interne typeset102

- options

* * *

`$UID`, numéro d'identifiant de l'utilisateur
unalias371, pour supprimer un alias370
uname, affiche des informations sur le système
Variables on initialisées
uniq, filtre pour supprimer des lignes dupliquées à partir d'un fichier trié
unset, supprime une variable shell
Boucle until
until [condition-est-vraie]; do
* * *
Variables

- Opérations sur des tableaux
- Affectation
  *Script exemple*30
  *Script exemple*31
  *Script exemple*28
- Variables internes *Bash*
- Bloc de code *sed* ou *awk*, initialiser une variable à
- Opérations d'incrément/de décrement/trinaire118 *style C*
- Modifie la valeur de variables internes du script en utilisant *set*
- declare102, pour modifier les propriétés des variables
- Supprimer une variable shell en utilisant *unset*
- Environnement
- Opérateurs d'expansion / remplacement de sous-chaînes
- Références indirectes
  `eval variable1=\$$variable2`
  Nouvelle notation
  `${!variable}`
- Longueur
  `${#var}`
- lvalue

- Manipulation et extension
- *Nom* et *valeur* d'une variable28, distinguer entre
- Affectation de *null* à une variable, éviter de
- Entre guillemets
  à l'intérieur de crochets de *test*
  pour préserver les *espaces blancs*37
- rvalue
- Configurer une valeur *null* 28
- Dans un *sous-shell*342 non visible par le shell parent
- Tester une variable si elle ne contient que des chiffres
- Saisie, restreindre les propriétés d'une variable
- Message d'erreur si non déclarée412
- Non initialisée30
- Variable sans guillemets38, *diviser*
- Désinitialiser
- Non typée32

* * *

wait, suspend l'exécution du script

- Pour remédier à l'arrêt d'un script

Guillemets *faibles*28 " **...** "
Boucle while
while [condition]; do

- Construction *while read*129

Espace blanc, espaces, tabulations et retours chariot

- Valeur par défaut de `$IFS`72
- Utilisation inappropriée de421
- Avant la fermeture d'une *chaîne limite* dans un *document en ligne*, erreur
- Avant les commentaires d'un script10
- *Guillemets*37, pour préserver les *espaces blancs* à l'intérieur de chaînes ou variables

Composants

* * *

xargs, Filtre pour grouper des arguments

- Accolades199
- Limiter les arguments passés
- Options
- Traiter les arguments un à la fois198
- Espace blanc, handling

* * *

yes

* * *

-z La chaîne est *vide*

Zombie, un processus terminé, mais pas encore tué par son parent