Guide Bash du débutant

Version française du livre *Bash Guide for Beginners*

Machtelt Garrels

Xalasys.com

<tille ne veut pas de spam CHEZ xalasys POINT com>

Ykerb

Adaptation française

Marc Blanc

Relecture de la version française <arsace CHEZ wanadoo POINT fr>

Jerome Blondel

Relecture de la version française <jeromeblondel CHEZ yahoo POINT fr>

Jean-Philippe Guérard

Préparation de la publication de la v.f. <fevrier CHEZ tigreraye POINT org>

Version 1.9.fr.1.1

2007-04-23

Table des matières

Introduction
- 1. Pourquoi ce guide ?
- 2. Qui devrait lire ce guide?
- 3. Nouvelles versions, traductions et disponibilité
- 4. Historique des révisions
- 5. Contributions
- 6. Observations et retours variés
- 7. information de Copyright
- 8. De quoi avez-vous besoin ?
- 9. Conventions employées dans ce document
- 10. Organisation de ce document

1. Bash et scripts Bash
- 1. Les langages de contrôle (Shell) courants
  - 1.1. Les fonctions du Shell en général
  - 1.2. Types de Shell
- 2. Avantages du Bourne Again SHell
  - 2.1. Bash est le Shell GNU
  - 2.2. Fonctionnalités offertes seulement par le Bash
- 3. L'exécution de commandes
  - 3.1. Généralité
  - 3.2. Les commandes intégrées du Shell

        3.3. Exécuter un programme dans un script.
    4. Construction de blocs
        4.1. Construction de blocs Shell
    5. Ecrire de bons scripts
        5.1. Caractéristiques d'un bon script
        5.2. Structure
        5.3. Terminologie
        5.4. Un mot sur l'ordre et la logique
        5.5. Un exemple Bash script : mysystem.sh
        5.6. Exemple : init script (NdT d'initialisation)
    6. Résumé
    7. Exercices
2. Ecrire et corriger des scripts
    1. Créer et lancer un script
        1.1. Écrire et nommer
        1.2. script1.sh
        1.3. Exécuter le script
    2. Les bases du script
        2.1. Quel Shell exécutera le script ?
        2.2. Ajout de commentaires
    3. Débugger (NdT : corriger) les scripts Bash
        3.1. Débugger le script globalement
        3.2. Débugger qu'une partie du script
    4. Résumé
    5. Exercices
3. L'environnement du Bash
    1. Les fichiers d'initialisation du Shell
        1.1. Les fichiers de configuration qui agissent sur tout le système
        1.2. Les fichiers de configuration utilisateur
        1.3. Modification des fichiers de configuration du Shell
    2. Variables
        2.1. Types de variables
        2.2. Créer des variables
        2.3. Exporter les variables
        2.4. Variables réservées
        2.5. Paramètres spéciaux
        2.6. Script à finalités multiples grâce aux variables
    3. Echappement et protection de caractères
        3.1. Pourquoi protéger ou 'échapper' un caractère ?
        3.2. Le caractère Echap (escape)
        3.3. Les apostrophes
        3.4. Les guillemets
        3.5. Codage ANSI-C
        3.6. Particularités
    4. Le processus d'expansion de Shell
        4.1. Généralité
        4.2. L'expansion d'accolades
        4.3. L'expansion du tilde
        4.4. Paramètre Shell et expansion de variable
        4.5. La substitution de commande
        4.6. L'expansion arithmétique
        4.7. La substitution de processus
        4.8. Le découpage de mots
        4.9. Expansion de noms de fichier
    5. Alias
        5.1. Que sont les alias ?
        5.2. Créer et supprimer des alias
    6. Plus d'options Bash
        6.1. Afficher les options
        6.2. Changer les options

7. Résumé
8. Exercices

4. Expressions régulières
    1. Expressions régulières
        1.1. Qu'est-ce qu'une expression régulière ?
        1.2. Les métacaractères des expressions régulières
        1.3. Expressions régulières basiques versus celles étendues
    2. Exemples en utilisant grep
        2.1. Qu'est-ce que grep ?
        2.2. Grep et les expressions régulières
    3. La correspondance de patron dans les fonctionnalités Bash
        3.1. Intervalle de caractère
        3.2. Classes de caractères
    4. Résumé
    5. Exercices
5. L'éditeur de flot GNU sed
    1. Introduction
        1.1. Qu'est-ce que sed ?
        1.2. commandes sed
    2. Opérations d'édition de modification
        2.1. Afficher les lignes contenant un patron
        2.2. Exclure les lignes contenant le patron
        2.3. Intervalle de lignes
        2.4. Trouver et remplacer avec sed
    3. L'usage en mode différé de sed
        3.1. Lire des commandes sed depuis un fichier
        3.2. Ecrire des fichiers de résultat
    4. Résumé
    5. Exercices
6. Le langage de programmation GNU awk
    1. Commencer avec gawk
        1.1. Qu'est-ce que gawk ?
        1.2. Commandes Gawk
    2. Le programme d'affichage
        2.1. Afficher les champs sélectionnés
        2.2. Formater les champs
        2.3. La commande print et les expressions régulières
        2.4. Patrons particuliers
        2.5. Les scripts Gawk
    3. Les variables Gawk
        3.1. Le séparateur de champs en entrée
        3.2. Les séparateurs de résultat
        3.3. Le nombre d'enregistrements
        3.4. Les variables définies par l'utilisateur
        3.5. Plus d'exemples
        3.6. Le programme printf
    4. Résumé
    5. Exercices
7. Les instructions de condition
    1. Introduction de if
        1.1. Généralité
        1.2. Applications simples de if
    2. L'emploi avancé de if
        2.1. les blocs if/then/else
        2.2. Les blocs if/then/elif/else
        2.3. Les instructions if imbriquées
        2.4. Opérations booléennes
        2.5. Emploi de l'instruction exit et du if
    3. Utiliser les instructions case
        3.1. Les conditions simplifiées

3.2. Exemple de script d'initialisation
4. Résumé
5. Exercices
8. Ecrire des scripts interactifs
1. Afficher les messages utilisateurs
1.1. Interactif ou pas ?
1.2. Utiliser la commande intégrée echo
2. Récupérer la saisie utilisateur
2.1. L'emploi de la commande intégrée read
2.2. Demander une entrée utilisateur
2.3. Redirection et descripteurs de fichiers
2.4. Fichier d'entrée et fichier de sortie
3. Résumé
4. Exercices
9. Tâches répétitives
1. La boucle loop
1.1. Comment ça marche ?
1.2. Exemples
2. La boucle while
2.1. Qu'est-ce que c'est ?
2.2. Exemples
3. La boucle until
3.1. Qu'est-ce que c'est ?
3.2. Exemple
4. Redirection d'entrée/sortie et boucles
4.1. Redirection des entrées
4.2. Redirection des sorties
5. Break et continue
5.1. L'intégrée break
5.2. L'intégrée continue
5.3. Exemples
6. Faire des menus avec l'intégrée select
6.1. Généralité
6.2. Sous-menus
7. L'intégrée shift
7.1. Qu'est-ce qu'elle fait ?
7.2. Exemples
8. Résumé
9. Exercices
10. Un peu plus sur les variables
1. Types de variables
1.1. Affectation générale de valeur.
1.2. Utiliser l'intégrée declare
1.3. Constantes
2. Variables tableau
2.1. Créer des tableaux
2.2. Invoquer les variables d'un tableau
2.3. Supprimer des variables tableau
2.4. Exemples de tableaux
3. Opérations sur les variables
3.1. Arithmétique sur les variables
3.2. Longueur de variable
3.3. Transformation de variables
4. Résumé
5. Exercices
11. Fonctions
1. Introduction
1.1. Qu'est-ce qu'une fonction ?
1.2. La syntaxe des fonctions
1.3. Les paramètres positionnels dans les fonctions

1.4. Afficher une fonction
2. Exemples de fonctions dans des scripts
2.1. Recyclage
2.2. Définir le chemin
2.3. Sauvegarde à distance
3. Résumé
4. Exercices
12. Trapper les signaux
1. Signaux
1.1. Introduction
1.2. Utilisation de signaux avec kill
2. Piéger les signaux
2.1. Généralité
2.2. Comment Bash interprète trap
2.3. Plus d'exemples
3. Résumé
4. Exercices
A. Possibilités du Shell
1. Fonctionnalités courantes
2. Fonctionnalités spécifiques
B. GNU Free Documentation License
1. Preamble
2. Applicability and definitions
3. Verbatim copying
4. Copying in quantity
5. Modifications
6. Combining documents
7. Collections of documents
8. Aggregation with independent works
9. Translation
10. Termination
11. Future revisions of this license
12. How to use this License for your documents
Glossaire
Index

Liste des illustrations

1. Couverture du Guide Bash du Débutant
2.1. script1.sh
3.1. Différentes invites pour des utilisateurs différents
6.1. Les champs dans awk
7.1. Test d'une ligne de commande avec if
7.2. Exemple employant les opérateurs booléens

Liste des tableaux

1. Conventions typographiques et d'usage
1.1. Vue générale des termes de programmation
2.1. Aperçu des options de débug
3.1. Variables réservées Bourne Shell
3.2. Les variables réservées de Bash
3.3. Les variables Bash spéciales
3.4. Opérateurs arithmétiques
4.1. Opérateurs d'expression régulière
5.1. Commandes d'édition Sed
5.2. Options Sed
6.1. Caractères de formatage pour gawk
7.1. Expressions primitives
7.2. Combinaison d'expressions
8.1. Séquences d'échappement reconnues par la commande echo

8.2. Options de l'intégrée read
10.1. Options de l'intégrée declare
12.1. Les signaux de contrôle dans Bash
12.2. Signaux courants de kill
A.1. Fonctionnalités courantes du Shell
A.2. Différences de fonctionnalités des Shell

Introduction

Table des matières

1. Pourquoi ce guide ?
2. Qui devrait lire ce guide?
3. Nouvelles versions, traductions et disponibilité
4. Historique des révisions
5. Contributions
6. Observations et retours variés
7. information de Copyright
8. De quoi avez-vous besoin ?
9. Conventions employées dans ce document
10. Organisation de ce document

1. Pourquoi ce guide ?

La raison première de ce document est que beaucoup de gens trouvent le HOWTO trop court et incomplet, et le guide Bash Scripting trop poussé. Il n'y a rien entre ces deux extrêmes. J'ai aussi écrit ce guide selon le principe général que les guides de base devraient être gratuits, alors que peu le sont.

C'est un guide pratique qui, sans être toujours sérieux, essaye de donner des exemples d'usage plutôt que théoriques. Je l'ai en partie écrit parce que je ne suis pas emballée par les exemples dépouillés, hyper simplifiés écrits par des gens qui, sachant de quoi ils parlent, montrent de super possibilités du Bash, tellement hors contexte que vous ne pouvez vous imaginez leurs applications pratiques. Vous pouvez lire ce genre de documents après ce guide, lequel contient exercices et exemples qui aideront à survivre dans la vraie vie.

De par mon expérience en tant qu'utilisateur, administrateur et formateur sur système UNIX/Linux, je sais que des gens peuvent avoir des années d'interactions quotidiennes avec leur système sans avoir la moindre notion de l'automatisation de tâches. De sorte qu'ils pensent souvent que UNIX n'est pas convivial, et pire, ils ont l'impression que c'est lent et obsolète. Cette difficulté est de celles que peut palier ce guide.

2. Qui devrait lire ce guide?

Quiconque qui, travaillant sur un système de type UNIX, veut se simplifier la vie. Utilisateurs avancés ou administrateurs peuvent tirer bénéfice de la lecture de ce guide. Les lecteurs qui ont déjà pris en main le système via la ligne de commande apprendront les ficelles de l'écriture de 'shell' qui facilitent l'exécution des tâches quotidiennes. L'administration de système repose grandement sur l'écriture de 'shell'. Les tâches courantes sont automatisées avec de simples scripts. Ce document est plein d'exemples qui vous encourageront à écrire les vôtres et qui vous inciteront à améliorer ceux existants.

Prérequis — Ce qui n'est pas dans ce guide. Vous devriez :

- Être familiarisé avec UNIX ou Linux : les commandes de bases, les pages de manuel et de documentation.
- Être capable d'utiliser un éditeur de texte.
- Comprendre les processus d'initialisation et d'arrêt du système : init et scripts d'initialisation.

- Savoir créer des utilisateurs et des groupes, déclarer des mots de passe.
- Savoir donner des droits et des modes d'accès.
- Comprendre les conventions de nommage des périphériques, le partitionnement, ainsi que le montage et démontage des systèmes de fichiers.
- Savoir ajouter et retirer des logiciels du système.

Voir Introduction to Linux (ou votre miroir TLDP TLDP mirror) si vous ignorez l'un de ces aspects. Des informations complémentaires peuvent être trouvées dans la documentation de votre système (man ; info pages), ou là : the Linux Documentation Project.

3. Nouvelles versions, traductions et disponibilité

La dernière édition se trouve à http://tille.xalasys.com/training/bash/. Vous devriez aussi la trouver à http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html.

Ce guide est disponible imprimé chez Fultus.com.

Figure 1. Couverture du Guide Bash du Débutant

Ce guide a été traduit :

- Traduction chinoise at http://xiaowang.net/bgb-cn/, par Wang Wei.
- Traduction ukrainienne at http://docs.linux.org.ua/index.php/LDP:Bash_beginners_guide, par Yaroslav Fedevych et son équipe.

Une traduction française en cours, à relire.

4. Historique des révisions

Historique des versions		
Version 1.9.fr.1.1	2007-04-23	Y, JPG
Relectures de Marc Blanc et Jerome Blondel.		
Version 1.9.fr.1.0	2007-04-01	Y, JPG
Première version française.		
Version 1.9	2006-10-10	MG
Remarques des lecteurs ajoutées, index ajouté en utilisant les tags DocBook.		
Version 1.8	2006-03-15	MG

Exemple clarifié au Chap 4, correction du document « ici » au Chap 9, corrections typographiques, ajout d'un lien vers les traductions chinoises et ukrainienne, note et chose à savoir au sujet de awk au Chap 6.		
Version 1.7	2005-09-05	MG
Correction de typographie au Chap 3, 6 et 7, remarques de lecteurs ajoutées, ajout d'une note au Chap 7.		
Version 1.6	2005-03-01	MG
Debuggage mineur, ajout de mots clés, note au sujet du nouveau Bash 3.0, retrait d'une image vierge.		
Version 1.5	2004-12-06	MG
Changements du fait du nouveau domaine, corrections mineures.		
Version 1.4	2004-10-18	MG
Debuggage, ajout de quelques notes au Chap 9, repositionnement de vues écran avec les sections écran. Correction de typographie.		
Version 1.3	2004-07-09	MG
Ajout d'une image de traceur 1X1 pixel http://tille.xalasys.com/images/blank-bash.png, ajout object texte pour toutes les images, réparation d'un lien mort dans l'index, amélioration de la liste des signaux.		
Version 1.2	2004-06-15	MG
Ajout index, plus de repère dans les sections écrans.		
Version 1.1	2004-05-22	MG
Dernière relecture avant la mise sous presse, ajout d'exemples, vérification du sommaire, exercices, introduction arrangée.		
Version 1.0	2004-04-27	TM
Livraison initiale pour LDP, d'autres exercices, d'autres repères, moins d'erreurs et abus, ajout du glossaire.		
Version 1.0-beta	2003-04-20	MG
Pre-version		

5. Contributions

Merci à tous les amis qui ont aidé (ou essayé) et à mon mari ; vos paroles d'encouragement ont rendu ce travail possible. Merci à tous les gens qui ont soumis anomalies, exemples et remarques — parmi plein, plein d'autres :

- Hans Bol, l'une des groupies
- Mike Sim, remarques sur le style
- Dan Richter, pour les exemples de tableaux
- Gerg Ferguson, pour les idées sur le titre
- Mendel Leo Cooper, pour avoir mis à disposition de l'espace
- #linux.be, pour m'avoir aidé à garder les pieds sur terre
- Frank Wang, pour ses remarques détaillées sur toutes mes erreurs ; -)

Remerciements special à Tabatha Marshall qui a bénévolement revu, et l'expression, et la grammaire. On forme une bonne équipe : elle travaille quand je dors. Et vice versa ; -)

6. Observations et retours variés

Informations manquantes, liens invalides, erreurs de frappe, remarques ? Envoyer un mail à

<tille ne veut pas de spam CHEZ xalasys POINT com>

La personne assurant le suivi du document.

7. information de Copyright

Copyright © 2003-2005 Machtelt Garrels.

Permission est donnée pour copier, distribuer et/ou modifier ce document selon les termes de la Licence GNU Free Documentation, Version 1.1 ou ultérieure publiée par la Free Software Foundation, avec les Sections Invariantes : « New versions of this document », « Contributions », « Feedback » et « Copyright information », sans textes de couverture de garde ni de textes de couverture de dos. Une copie de la licence est incluse dans Annexe B, *GNU Free Documentation License* intitulée « GNU Free Documentation License ».

L'auteur et l'éditeur ont fait leur possible pour s'assurer de la validité des informations de ce livre. Cependant, le contenu de ce guide est mis à disposition sans garantie, que ce soit explicite ou implicite. Ni l'auteur, ni l'éditeur, ni un distributeur ne peuvent être tenu responsable des éventuels dommages ou conséquences résultant de l'application du contenu de ce guide.

Les logos, marques déposées et les symboles utilisés dans ce guide sont la propriété de leur dépositaire respectif.

8. De quoi avez-vous besoin ?

Bash, téléchargeable à http://www.gnu.org/directory/GNU/. Le Bash accompagne à peu près tous les systèmes Linux, et se trouve maintenant sur un large éventail de systèmes UNIX.

Se compile aisément si vous avez besoin de le personnaliser, testé sur un large éventail d'UNIX, Linux, MS Windows, et autres systèmes.

9. Conventions employées dans ce document

Les conventions typographiques et d'usage suivantes apparaissent dans le texte :

Tableau 1. Conventions typographiques et d'usage

Type de texte	**sens**
« Texte entre guillemets »	Citation de gens, texte rendu par l'ordinateur entre guillemets
`reproduction de la vue du terminal`	Capture des données saisies ou affichées sur le terminal, généralement rendue avec un fond gris clair.
commande	Nom d'une commande qui peut être tapée sur la ligne de commande.
`VARIABLE`	Nom d'une variable ou pointeur vers le contenu d'une variable, comme `$VARNAME`.
`option`	Option d'une commande, comme « l'option `-a` de la commande **ls** ».
`argument`	Argument d'une commande, comme dans « read **man** ***`ls`*** ».
`commande options` *`paramètres`*	Synopsis de commande ou emploi habituel, sur une ligne séparée.

Type de texte	sens
`NomDeFichier`	Nom d'un fichier ou d'un répertoire, par exemple « se positionner dans le répertoire `/usr/bin` . »
Touche	Touches à frapper sur le clavier, exemple « taper **Q** pour quitter ».
Bouton	Bouton graphique sur lequel cliquer comme le bouton OK .
Menu → Choix	Options à choisir dans un menu graphique, par exemple : « Choisir Aide → A propos de Mozilla dans votre navigateur. »
Terminologie	Terme important ou concept : « Le *noyau* est le coeur du système. »
\	La barre oblique inversée affichée dans un terminal ou dans un synopsis de commande indique que la ligne n'est pas finie. (NdT : nous appelerons ce symbole l'échappement). En d'autres mots, si vous voyez une longue commande qui est découpée en plusieurs lignes, \ signifie « Ne pressez pas encore la touche **Entrée** encore ! »
Voir Chapitre 1, *Bash et scripts Bash*	Lien vers sujets connexes dans ce guide.
L'auteur	Lien vers une ressource WEB externe.

10. Organisation de ce document

Ce guide expose des concepts utiles dans la vie de tous les jours de l'utilisateur Bash assidu. Bien qu'une connaissance basique du shell soit requise, nous commençons par aborder les composants et pratiques de base dans les 3 premiers chapitres.

Les chapitres 4 à 6 abordent les outils de base qui sont utilisés régulièrement dans les scripts.

Les chapitres 8 à 12 abordent les constructions les plus courantes dans les scripts.

Tous les chapitres sont accompagnés d'exercices qui testent votre aptitude à aborder le chapitre suivant.

- Chapitre 1, *Bash et scripts Bash* : Les bases de Bash : pourquoi Bash est si bon, construction de blocs, premières consignes d'écriture de bons scripts.
- Chapitre 2, *Ecrire et corriger des scripts* : Les bases du script : écrire et débugger.
- Chapitre 3, *L'environnement du Bash* : L'environnement Bash : les fichiers d'initialisation, les variables, les expressions littérales, l'ordre d'expansion, les alias, les options.
- Chapitre 4, *Expressions régulières* : Expressions régulières : une introduction.
- Chapitre 5, *L'éditeur de flot GNU sed* : Sed : une introduction à l'éditeur ligne à ligne.
- Chapitre 6, *Le langage de programmation GNU awk* : Awk : introduction à awk le langage de progammation.
- Chapitre 7, *Les instructions de condition* : Les instructions conditionnelles : constructions utilisées en Bash pour tester des conditions.
- Chapitre 8, *Ecrire des scripts interactifs* : Les scripts interactifs : faire des scripts conviviaux, intégrer la saisie de l'utilisateur.

- Chapitre 9, *Tâches répétitives* : Exécuter des commandes récursivement : constructions utilisées en Bash pour automatiser l'exécution de commandes.
- Chapitre 10, *Un peu plus sur les variables* : Variables complexes : spécifier des types de variables, introduction aux tableaux de variables, opérations sur variables.
- Chapitre 11, *Fonctions* : Fonctions : une introduction.
- Chapitre 12, *Trapper les signaux* : Capturer des signaux : introduction aux signaux de processus, capturer les signaux envoyés par l'utilisateur.

Chapitre 1. Bash et scripts Bash

Table des matières

1. Les langages de contrôle (Shell) courants
    1.1. Les fonctions du Shell en général
    1.2. Types de Shell
2. Avantages du Bourne Again SHell
    2.1. Bash est le Shell GNU
    2.2. Fonctionnalités offertes seulement par le Bash
3. L'exécution de commandes
    3.1. Généralité
    3.2. Les commandes intégrées du Shell
    3.3. Exécuter un programme dans un script.
4. Construction de blocs
    4.1. Construction de blocs Shell
5. Ecrire de bons scripts
    5.1. Caractéristiques d'un bon script
    5.2. Structure
    5.3. Terminologie
    5.4. Un mot sur l'ordre et la logique
    5.5. Un exemple Bash script : mysystem.sh
    5.6. Exemple : init script (NdT d'initialisation)
6. Résumé
7. Exercices

Résumé

Dans ce module d'introduction nous

- Décrivons quelques Shell courants
- Mettons en avant les avantages et possibilités du Bash GNU
- Décrivons les blocs de constructions du Shell
- Abordons les fichiers d'initialisation du Bash
- Voyons comment le Shell exécute les commandes
- Examinons quelques exemples simples de scripts

1. Les langages de contrôle (Shell) courants

1.1. Les fonctions du Shell en général

Le Shell UNIX interprète les commandes de l'utilisateur, qui sont soit directement entrées par celui-ci, ou qui peuvent être lues depuis un fichier appelé un script shell ou programme. Ces scripts sont interprétés, donc non compilés. Le Shell lit les commandes de chaque ligne du script et cherche

ces commandes dans le système (voir Section 2, « Avantages du Bourne Again SHell »), alors qu'un compilateur convertit un programme en une forme lisible par la machine, un fichier exécutable - lequel peut alors être employé dans un script.

A part de passer des commandes au noyau, la tâche principale du Shell est de mettre en place un environnement utilisateur qui peut être configuré individuellement par le biais de fichiers de configuration.

1.2. Types de Shell

Tout comme les gens connaissent une variété de langages, votre système UNIX généralement offre une variété de types de Shell :

- **sh** ou Bourne Shell : le Shell originel toujours en vigueur sur les systèmes UNIX et sur les environnements de type UNIX. C'est le Shell de base, un petit programme avec peu de possibilités. Bien que ce ne soit pas le Shell standard, il est toujours disponible sur les systèmes Linux par souci de compatibilité des programmes UNIX.
- **bash** ou Bourne Again shell : le Shell standard GNU , intuitif et souple. Probablement celui à conseiller aux débutants tout en étant un outil puissant pour un usage poussé et professionnel. Sur Linux, **bash** est le Shell standard pour l'utilisateur courant. Ce Shell est réputé être un *sur-ensemble* du Bourne Shell, un ensemble d'ajouts et d'extensions. Ce qui veut dire que le Bourne Again Shell est compatible avec le Bourne Shell : les commandes reconnues par **sh**, le sont aussi par **bash**. Cependant, l'inverse n'est pas toujours vrai. Tous les exemples et exercices de ce livre utilisent **bash**.
- **csh** ou C shell : la syntaxe de ce Shell ressemble à celle du langage de programmation C. Parfois demandée par les programmeurs.
- **tcsh** ou TENEX C Shell : un surensemble du répandu Shell C, implémentant convivialité et rapidité. C'est pourquoi certains l'appellent aussi le Turbo Shell C.
- **ksh** ou le Korn shell : quelques fois apprécié des gens venant du monde UNIX. Un sur-ensemble du Bourne Shell ; avec une configuration - le cauchemar des débutants - standard.

Le fichier `/etc/shells` donne un aperçu des Shells connus du système Linux :

```
mia:~> cat /etc/shells
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

Votre Shell par défaut est déclaré dans le fichier `/etc/passwd` , comme cette ligne pour l'utilisateur *mia* :

```
mia:L2NOfqdlPrHwE:504:504:Mia Maya:/home/mia:/bin/bash
```

Pour permuter d'un Shell à un autre, simplement entrez le nom du nouveau Shell actif dans le terminal. Le système trouve le répertoire où le nom apparaît au moyen des paramètres de `PATH`, et puisqu'un Shell est un fichier exécutable (programme), le Shell courant l'active et il s'exécute. Une nouvelle invite est souvent affichée, du fait que chaque Shell a une interface propre :

```
mia:~> tcsh
[mia@post21 ~]$
```

2. Avantages du Bourne Again SHell

2.1. Bash est le Shell GNU

Le projet GNU (ne pas confondre GNU et UNIX) offre des outils pour l'administration de système de type UNIX qui sont libres et qui respectent les standards UNIX.

Bash est un Shell compatible avec sh qui incorpore des spécificités utiles du Korn Shell (ksh) et du C Shell (csh). Il est censé se conformer à la norme IEEE POSIX P1003.2/ISO 9945.2 Standards des Shell et Outils. Il offre des améliorations fonctionnelles par rapport à sh pour la programmation et l'utilisation interactive ; ce qui inclut l'édition de commande en ligne, historique illimité des commandes, contrôle des travaux, fonctions Shell et alias, tableau indexé de taille illimitée, et l'arithmétique d'entiers dans toutes les bases depuis la base 2 jusqu'à la base 64. Bash peut exécuter la plupart des scripts sh sans modification.

Comme les autres projets GNU, le projet bash a été lancé pour préserver, protéger et promouvoir la liberté d'utiliser, étudier, copier, modifier et redistribuer les logiciels. Il est généralement admis que de telles conditions stimulent la créativité. Cela a été le cas avec le programme Bash, qui a beaucoup de fonctionnalités que les autres Shells n'offrent pas.

2.2. Fonctionnalités offertes seulement par le Bash

2.2.1. Invocation

En plus de l'option permettant des commandes Shell à un caractère qui peut être configuré généralement avec la commande intégrée **set**, il y a plusieurs options multi-caractères que vous pouvez employer. Nous verrons quelques unes de ces options les plus usitées dans les chapitres suivants ; la liste complète peut être trouvée dans les pages info de Bash, Bash features → Invoking Bash.

2.2.2. Fichiers de démarrage de Bash

Les fichiers de démarrage sont des scripts qui sont lus et exécutés par Bash quand il démarre. Les sous-sections suivantes décrivent diverses façons de démarrer le Shell, et le fichier de démarrage lu en conséquence.

2.2.2.1. Invoqué pour être le Shell d'interaction, ou avec l'option `–login'

Interactif signifie que vous pouvez entrer des commandes. Le Shell n'est pas lancé parce qu'un script a été activé. Un Shell de connection vous donne accès au Shell après qu'il vous ait authentifié, généralement en contrôlant le nom d'utilisateur et le mot de passe.

Fichiers lus :

- `/etc/profile`
- `~/.bash_profile`, `~/.bash_login` ou `~/.profile` : le premier fichier lisible trouvé est lu
- `~/.bash_logout` à la déconnexion.

Des messages d'erreur s'affichent si les fichiers de configuration existent mais sont illisibles. Si un fichier n'existe pas, Bash cherche le suivant.

2.2.2.2. Invoqué comme Shell interactif sans étape de connexion

Un Shell sans connexion signifie que l'accès ne nécessite pas d'authentification par le système. Par exemple, quand vous ouvrez un terminal par le biais d'une icone, ou d'un menu.

Fichiers lus :

- `~/.bashrc`

Ce fichier est habituellement référencé dans `~/.bash_profile` :

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

Voir Chapitre 7, *Les instructions de condition* pour plus d'informations sur la construction **if**.

2.2.2.3. Invoqué non interactivement

Tous les scripts utilisent un Shell non-interactif. Ils sont programmés pour faire certaines tâches et ne peuvent être utilisés pour faire autre chose que ce pour quoi ils ont été prévus.

Fichiers lus :

- définis par `BASH_ENV`

`PATH` n'est pas utilisé pour la recherche de ces fichiers, donc mettre le chemin complet dans la variable si vous souhaitez en faire usage.

2.2.2.4. Invoqué avec la commande sh

Bash essaye de se comporter comme le programme historique Bourne **sh** tout en se conformant à la norme POSIX.

Fichiers lus :

- `/etc/profile`
- `~/.profile`

Quand il est invoqué de façon interactive, la variable `ENV` peut pointer vers des informations de démarrage suplémentaires.

2.2.2.5. Mode POSIX

Cette option est activée soit en employant l'intégrée **set** :

```
set -o posix
```

ou en appelant le **Bash** avec l'option `--posix` option. Bash essayera alors de respecter autant que possible la norme POSIX des Shell. Déclarer la variable `POSIXLY_CORRECT` fait la même chose.

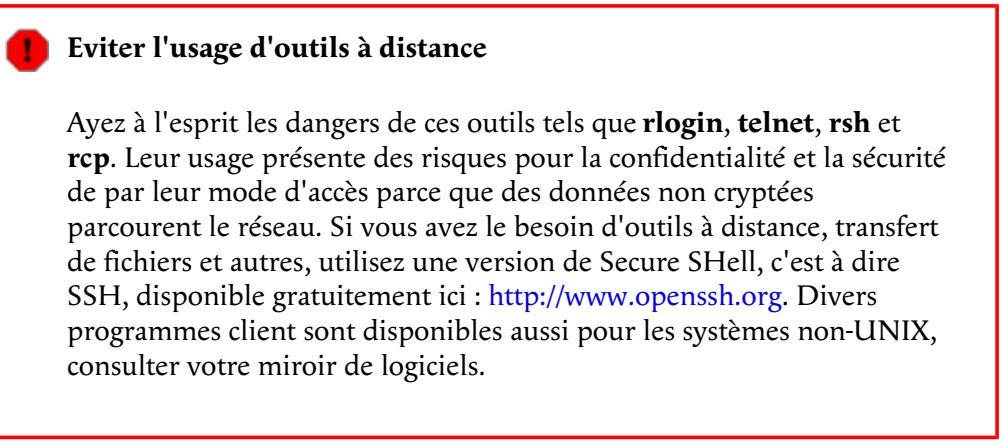
Fichiers lus :

- définis par la variable `ENV`

2.2.2.6. Invoqué à distance

Fichiers lus quand le Shell est invoqué par **rshd** :

- `~/.bashrc`



2.2.2.7. Invoqué alors que UID est différent de EUID

Aucun fichier de démarrage n'est lu dans ce cas.

2.2.3. Shell interactif

2.2.3.1. Qu'est-ce qu'un Shell interactif

Un Shell interactif généralement lit et écrit depuis et sur un terminal utilisateur : les flux d'entrée et de sortie sont dirigés vers le terminal. Le mode interactif de Bash est activé quand la commande **bash** est invoquée sans les options rendant inactif, et sauf avec l'option qui permet de prendre l'entrée depuis une chaîne de caractères ou quand le Shell est invoqué de façon à lire l'entrée standard, ce qui autorise les paramètres positionnels (voir Chapitre 3, *L'environnement du Bash*).

2.2.3.2. Ce Shell est-il interactif ?

Test en examinant la variable spéciale `-`, il contient un 'i' quand le Shell est interactif :

```
eddy:~> echo $-
himBH
```

Dans un Shell non interactif, l'invite `PS1`, n'est pas paramétrée.

2.2.3.3. Le comportement d'un Shell interactif

Différences dans le mode interactif :

- Bash lit les fichiers de démarrage.
- Le contrôle de travail est actif par défaut.
- Les invites sont établies, `PS2` est déclaré pour des commandes multi-lignes , il est souvent déclaré à « > ». C'est aussi l'invite que l'on obtient quand le Shell trouve que la commande entrée n'est pas finie, par exemple si vous oubliez des guillemets, une structure de commande non finie, etc.
- Les commandes sont par défaut lues depuis la ligne de commande en utilisant **readline**.
- Bash interprète l'option Shell `ignoreeof` plutôt que de sortir immédiatement à la réception de EOF (Fin de Fichier).
- L'historique des commandes avec leur expansion est activé par défaut. L'historique est enregistré dans le fichier désigné par `HISTFILE` quand le Shell est quitté. Par defaut, `HISTFILE` pointe vers `~/.bash_history`.
- L'expansion d'alias est actif.
- En l'absence de 'trap', le signal `SIGTERM` est ignoré.
- En l'absence de 'trap', `SIGINT` est capturé et exploité. Donc, les touches **Ctrl**+**C**, par exemple, ne feront pas quitter votre Shell interactif.
- Le signal `SIGHUP` est configuré pour être envoyé à tous les travaux quand Bash se termine, avec l'option `huponexit` option.
- Les commandes sont exécutées à la lecture.
- Bash vérifie régulièrement le courrier électronique.

- Bash peut être configuré pour quitter quand il trouve des variables non déclarées. En mode interactif ce comportement est désactivé.
- Quand les commandes intégrées du Shell trouvent des erreurs de redirection, cela n'a pas pour effet de quitter le Shell.
- Les commandes intégrées utilisées selon le mode POSIX et qui renvoient des erreurs n'ont pas pour effet de quitter le Shell. Les commandes intégrées sont listées à la Section 3.2, « Les commandes intégrées du Shell ».
- Un échec de **exec** ne fait pas quitter le Shell.
- Des erreurs produites par l'analyse de syntaxe ne font pas quitter le Shell.
- Le contrôle simple de nom des arguments de la commande intégrée **cd** est activé par défaut.
- La sortie automatique, après le laps de temps spécifié par la variable `TMOUT`, est activée.

Plus d'informations :

- Section 2, « Variables »
- Section 6, « Plus d'options Bash »
- Voir Chapitre 12, *Trapper les signaux* au sujet des signaux.
- Section 4, « Le processus d'expansion de Shell » aborde divers processus d'expansion sur une commande saisie.

2.2.4. Les conditions

Les expressions conditionnelles sont utilisées dans la commande composée **[[** et par les intégrées **test** et **[**.

Les expressions peuvent être unaire ou binaire. Une expression unaire est souvent utilisée pour examiner le statut d'un fichier. Vous avez seulement besoin d'un objet, exemple un fichier, pour tester une condition dessus.

Il y a aussi des opérateurs de comparaison de textes et de nombres ; ils sont binaires, puisqu'ils requièrent 2 objets pour effectuer le test. Si l'option `FICHIER` d'une expression est de la forme `/dev/fd/N`, alors le descripteur de fichier N est utilisé. Si l'option `FICHIER` d'une expression est de la forme `/dev/stdin`, `/dev/stdout` ou `/dev/stderr`, alors le descripteur de fichier 0, 1 ou 2 respectivement est utilisé.

Les conditions sont discutées en détail au Chapitre 7, *Les instructions de condition*.

Plus d'informations au sujet des descripteurs de fichiers à la Section 2.3, « Redirection et descripteurs de fichiers ».

2.2.5. L'arithmétique avec Shell

Le Shell permet aux expressions arithmétiques d'être évaluées, en tant que processus d'expansion ou par l'intégrée **let**.

L'évaluation utilise des entiers de longueur fixe sans vérification de possible débordement de capacité, mais avec un contrôle de la division par 0 qui renvoie une erreur. Les opérateurs, leur ordre et leur associativité, sont pareil que dans le langage C, voir Chapitre 3, *L'environnement du Bash*.

2.2.6. Alias

Les alias permettent à une chaîne d'être substituée par un mot quand il est utilisé comme le premier

mot d'une commande simple. Le Shell maintient une liste d'alias qui peuvent être déclarés ou supprimés avec les commandes **alias** et **unalias**.

Bash lit toujours au moins une ligne complète saisie avant d'exécuter une des commandes de cette ligne. L'alias est interprété quand la commande est lue, non pas quand elle est exécutée. De ce fait, une définition d'alias apparaissant sur la même ligne qu'une autre commande ne prendra effet qu'à la lecture de la ligne suivante. Les commandes suivant la définition de l'alias sur la ligne ne seront pas affectées par le nouvel alias.

Un alias est interprété quand la définition d'une fonction est lue, pas quand la fonction est exécutée, parce que la définition de fonction est elle-même une commande composée. En conséquence, l'alias défini dans une fonction n'est pas utilisable tant que la fonction n'a pas été exécutée.

Nous aborderons les alias en détail à la Section 5, « Alias ».

2.2.7. Tableaux

Bash fournit les variables sous la forme d'un tableau à une dimension . Toute variable peut être utilisée comme un tableau ; l'intégrée **declare** déclarera explicitement un tableau. Il n'y a pas de limite supérieure à la taille d'un tableau, ni de besoin de trier ou d'assigner de façon contiguë les valeurs. Les tableaux sont basés sur le zéro. Voir Chapitre 10, *Un peu plus sur les variables*.

2.2.8. Pile de répertoires

La pile de répertoires est une liste des répertoires récemment visités. L'intégrée **pushd** ajoute des répertoires à la pile tout en changeant le répertoire courant, et l'intégrée **popd** enlève les répertoires spécifiés de la pile en positionant le répertoire courant comme étant celui qui vient d'être enlevé.

Le contenu peut être affiché avec la commande **dirs** ou en visualisant la variable `DIRSTACK`.

Plus d'informations au sujet du fonctionnement de ces mécanismes se trouvent dans les pages Bash info.

2.2.9. L'invite

Bash permet de jouer avec l'invite de façon amusante. Voir la section *Controlling the Prompt* dans les pages info de Bash.

2.2.10. Le Shell restreint

Quand il est invoqué avec **rbash** ou avec `--restricted` ou l'option `-r` , il se produit ceci :

- La commande intégrée **cd** est indisponible.
- Déclarer ou invalider `SHELL`, `PATH`, `ENV` ou `BASH_ENV` n'est pas possible.
- Les noms de commande ne peuvent plus comporter de slashes.
- Un nom de fichier contenant un slash n'est pas permis avec l'intégrée **.** (**source**) .
- L'intégrée **hash** n'accepte pas des slashs avec l'option `-p`.
- L'Import de fonctions au démarrage est désactivé.
- `SHELLOPTS` est ignoré au démarrage.
- La redirection des résultats avec >, >|, ><, >**&**, **&**> et >> est désactivée.
- La commande intégrée **exec** est indisponible.

- L'option `-f` et `-d` Les options sont désactivées pour l'intégrée **enable** .
- Un chemin par défaut `PATH` ne peut pas être spécifié avec l'intégrée **command**.
- Annuler le mode restrictif n'est pas possible.

Quand une commande qui appele un script Shell est exécutée, **rbash** annule toute restriction dans le Shell lancé dans lequel s'exécute ce script.

Plus d'informations :

- Section 2, « Variables »
- Section 6, « Plus d'options Bash »
- Info Bash → Basic Shell Features → Redirections
- Section 2.3, « Redirection et descripteurs de fichiers » : redirection poussée

3. L'exécution de commandes

3.1. Généralité

Bash determine le type de programme qui doit être exécuté. Les programmes standards sont les commandes système qui existent sous forme compilée dans le système. Quand un tel programme est exécuté, un nouveau processus est créé parce que Bash lance une copie exacte de lui-même. Ce processus fils a le même environnement que son parent, seul l'identifiant est différent. Cette procédure est appellée *forking*.

Une fois que le processus a fourché, l'espace d'adresse du processus fils est renseigné avec ses propres informations. Ceci est fait grâce à un appel système par *exec*.

Le mécanisme *fork-and-exec* cependant substitue une ancienne commande par une nouvelle, tandis que l'environnement dans lequel le nouveau programme est exécuté reste le même, y compris la configuration des entrées/sorties, des variables d'environnement et des priorités. Ce mécanisme est employé pour créer tous les processus UNIX, donc il s'applique aussi au système d'opération Linux. Même le premier processus, **init**, avec l'ID 1, fourche dans la procédure de démarrage appelée *bootstrapping*.

3.2. Les commandes intégrées du Shell

Les commandes intégrées sont parties intégrantes du Shell lui-même. Quand le nom d'une commande intégrée est employé comme le premier mot d'une commande simple, le Shell exécute la commande directement, sans créer un nouveau processus. Les commandes intégrées sont nécessaires pour implanter des fonctionnalités impossibles ou difficiles à mettre en oeuvre par des outils externes.

Bash possède 3 types de commandes intégrées :

- Les intégrées Bourne :

  :, **.**, **break**, **cd**, **continue**, **eval**, **exec**, **exit**, **export**, **getopts**, **hash**, **pwd**, **readonly**, **return**, **set**, **shift**, **test**, **[**, **times**, **trap**, **umask** et **unset**.

- Les intégrées Bash :

  alias, **bind**, **builtin**, **command**, **declare**, **echo**, **enable**, **help**, **let**, **local**, **logout**, **printf**, **read**, **shopt**, **type**, **typeset**, **ulimit** et **unalias**.

- Les intégrées speciales :

Quand Bash est exécuté en mode POSIX, les commandes spéciales diffèrent des autres selon 3 aspects :

1. Les commandes spéciales sont rencontrées avant les fonctions Shell pendant la localisation de la commande.
2. Si une commande spéciale renvoie un statut en erreur, un Shell non-interactif quitte.
3. Les variables affectées avant la commande existent toujours dans l'environnement Shell après que la commande se soit terminée.

Les commandes spéciales POSIX sont **:**, **.**, **break**, **continue**, **eval**, **exec**, **exit**, **export**, **readonly**, **return**, **set**, **shift**, **trap** et **unset**.

La plupart de ces intégrées seront abordées dans les chapitres suivants. Pour les commandes qui ne le seront pas, se référer aux pages Info.

3.3. Exécuter un programme dans un script.

Quand le programme en exécution est un script Shell, Bash créera un nouveau processus Bash en activant un *fork*. Ce sous-Shell lit les lignes du script une par une. Les commandes de chaque ligne sont lues, interprétées et exécutées comme si elles avaient été entrées au clavier.

Tandis que le sous-Shell opère sur chaque ligne du script, le Shell parent attend que le processus fils ait fini. Quand il n'y a plus de ligne à lire dans le script, le sous-Shell se termine. Le Shell parent s'active et affiche l'invite de nouveau.

4. Construction de blocs

4.1. Construction de blocs Shell

4.1.1. La syntaxe Shell

Si la saisie n'est pas commentée, le Shell la lit et la divise en mots et opérateurs, selon les règles d'analyse qui déterminent la signification de chaque caractère saisi. Alors ces mots et opérateurs sont transformés en commandes et autres constructions, lesquels retournent un statut d'exécution qui peut être exploité. Le schéma fork-and-exec ci-dessus est appliqué seulement après que le Shell ait analysé la saisie selon le processus suivant :

- Le Shell lit le texte en entrée dans un fichier, ou une chaîne (de caractère), ou depuis le périphérique de saisie.
- Le texte est découpé en mots et opérateurs, selon les règles de syntaxe, voir Chapitre 3, *L'environnement du Bash*. Ces éléments sont séparés par des *métacaractères*. Les alias sont remplacés par leur équivalent.
- Le Shell *parses* (analyse et transforme) les éléments en commandes simples ou composées.
- Bash procède à diverses expansions d'éléments, les décomposant en listes de fichiers et commandes avec arguments.
- Au besoin il est procédé à des redirections, les opérateurs de redirection et leurs opérandes sont éliminés de la liste des arguments.
- Les commandes sont exécutées.
- Optionnellement le Shell attend que la commande s'achève pour récupérer son statut d'exécution.

4.1.2. Les commandes Shell

Une simple commande Shell telle que **touch `file1 file2 file3`** consiste en la commande elle-même suivie par des arguments, séparés par des espaces.

Les commandes Shell plus complexes sont des compositions variées de commandes simples : en tube lequel délivre le résultat d'une commande sur le canal d'entrée de la suivante, en boucle ou en construction de conditions, et encore d'autres façons. Quelques exemples :

```
ls | more
```

```
gunzip file.tar.gz | tar xvf -
```

4.1.3. La fonction Shell

Les fonctions Shell sont un moyen de grouper des commandes pour une exécution ultérieure par l'appel d'un nom pour le groupe. Elles sont exécutées tout comme des commandes « régulières ». Quand le nom de la fonction Shell est employé comme le nom d'une commande simple, la liste des commandes associées à cette fonction est exécutée.

Les fonctions Shell sont exécutées dans le contexte en cours du Shell, elles ne sont pas interprétées dans un nouveau processus.

Les fonctions sont expliquées au Chapitre 11, *Fonctions*.

4.1.4. Les paramètres Shell

Un paramètre est une entité qui mémorise une valeur. Cela peut être un nom, un nombre ou une valeur spéciale. Pour les besoins du Shell, une variable est un paramètre qui mémorise un nom. Une variable a une valeur et zéro ou plus attributs. Les variables sont créées avec l'intégrée **declare**.

Si aucune valeur ne lui est assignée, une variable prend la valeur nulle. Une variable peut être invalidée seulement avec l'intégrée **unset**.

L'assignation de variables est traitée à la Section 2, « Variables », l'utilisation poussée de variables au Chapitre 10, *Un peu plus sur les variables*.

4.1.5. Le processus d'expansion de Shell

L'expansion par le Shell est effectuée après que chaque ligne de commande ait été découpée en jetons ou morceaux. Voici les opérations d'expansion :

- L'expansion d'accolades
- L'expansion du tilde
- L'expansion de paramètres et de variables
- La substitution de commande
- L'expansion arithmétique
- Le découpage de mots
- L'expansion de nom de fichiers

Nous traiterons ces types d'expansion à la Section 4, « Le processus d'expansion de Shell ».

4.1.6. Redirections

Avant qu'une commande soit exécutée, ses flux d'entrée et de sortie peuvent être redirigés en employant un symbole spécial interprété par le Shell. La redirection peut aussi être employée pour

ouvrir et fermer des fichiers dans l'environnement d'exécution du Shell.

4.1.7. L'exécution de commandes

A l'exécution d'une commande, les mots que l'analyse syntaxique a marqué comme assignation de variables (précédant le nom de commande) et comme redirection sont conservés pour y faire référence ultérieurement. Les mots qui ne sont pas des assignations de variables ou des redirections sont analysés ; le premier mot restant après cette analyse est considéré comme étant le nom de la commande et le reste ses arguments. Alors les opérations de redirections sont effectuées, puis les valeurs assignées aux variables sont interprétées (expansion). Si le résultat ne donne aucun nom de commande, les variables sont affectées dans l'environnement en cours.

Une part importante du travail du Shell est de localiser les commandes. Bash le fait de cette façon :

- Recherche du caractère slash dans la commande. Si aucun, d'abord parcourir la liste de fonctions pour voir si elle contient le nom de commande cherché.
- Si la commande n'est pas une fonction, la chercher dans la liste des intégrées.
- Si la commande n'est ni une fonction ni une intégrée, la chercher dans les répertoires listés dans PATH. Bash se sert d'une *table de hachage* (zone de stockage en mémoire) pour récupérer le chemin complet des exécutables de sorte qu'une recherche extensive est évitée.
- Si la recherche est un échec, Bash affiche un message d'erreur et retourne le statut d'exécution 127.
- Si la recherche donne un résultat ou si la commande contient des slashs, le Shell exécute la commande dans un environnement d'exécution propre.
- Si l'exécution échoue parce que le fichier n'est pas exécutable et qu'il n'est pas un répertoire, il est alors considéré comme étant un script Shell.
- Si la commande n'a pas été lancée de façon asynchrone, le Shell attend que la commande se termine et récupère son statut d'exécution.

4.1.8. Les scripts Shell

Quand un fichier contenant des commandes Shell est utilisé comme le premier argument n'étant pas une option à l'invocation de Bash (sans l'option `-c` ou l'option `-s`) un Shell non-interactif est lancé. Ce Shell cherche d'abord le fichier de script dans le répertoire en cours, puis cherche dans ceux de PATH si le fichier ne peut pas y être trouvé.

5. Ecrire de bons scripts

5.1. Caractéristiques d'un bon script

Ce guide traite principalement du dernier bloc de construction Shell : les scripts. Quelques considerations générales avant de continuer :

1. Un script devrait s'exécuter sans erreurs.
2. Il devrait accomplir la tâche pour laquelle il a été conçu.
3. La logique du programme est clairement définie et apparente.
4. Un script n'exécute pas des instructions inutiles.
5. Les scripts devraient être réutilisables.

5.2. Structure

La structure d'un script est très flexible. Même si Bash permet beaucoup de liberté, vous devez mettre en oeuvre une logique rigoureuse, un contrôle des données, une efficacité qui permet à l'utilisateur qui exécute le script de le faire facilement et correctement.

Au moment d'écrire un nouveau script, posez-vous les questions suivantes :

- Aurai-je besoin d'informations de la part de l'utilisateur ou de son environnement ?
- Comment vais-je mémoriser ces données ?
- Des fichiers doivent-ils être créés ? Où et avec quel propriétaire et quelles permissions ?
- Quelles commandes utiliserais-je ? Si le script est exécuté sur différents systèmes, est-ce que tous ces systèmes ont les commandes dans la version requise ?
- L'utilisateur a-t-il besoin que le script lui renvoie des informations ? Quand et pourquoi ?

5.3. Terminologie

La table ci-dessous donne un aperçu des termes de programmation avec lesquels vous devez vous familiariser :

Tableau 1.1. Vue générale des termes de programmation

Termes	**Qu'est-ce que c'est ?**
Contrôle de commande	Test du statut d'exécution (NdT : code retour) d'une commande pour déterminer si une portion du code doit être exécutée ou pas.
Branchement conditionnel	Une instruction logique du programme qui détermine quelle alternative du programme exécuter ensuite.
Enchaînement logique	La conception du programme dans ses grandes lignes. Détermine la séquence logique des opérations de sorte que cela aboutisse à un résultat contrôlé.
Boucle	Partie de code qui s'exécute zéro fois ou plus.
Saisie de l'utilisateur	Donnée fournie par une source externe(NdT périphérique de saisie) pendant que le programme tourne, qui peut être mémorisée et exploitée au besoin.

5.4. Un mot sur l'ordre et la logique

Afin d'accélérer les phases de développement, l'ordre logique du programme devrait être pensé à l'avance. C'est votre première étape quand vous développez un script.

Diverses méthodes peuvent être utilisées ; une des plus courantes est la constitution de listes. Lister les opérations nécessaires au programme vous permet de décrire chaque tâche. Les opérations unitaires peuvent être référencées par leur numéro dans la liste.

En utilisant vos propres mots pour déterminer les opérations à exécuter par votre programme il vous sera plus facile de créer un programme compréhensible. Ensuite, vous écrivez le langage compris par Bash.

L'exemple ci-dessous montre un tel enchaînement logique. Il décrit la rotation des fichiers journaux.

Cet exemple montre la possible réitération d'une boucle, en fonction du nombre de fichiers journaux sur lesquels vous voulez paramétrer une rotation.

1. Voulez-vous paramétrer la rotation de journaux ?
    a. Si oui :
        i. Indiquez le répertoire contenant les journaux sur lesquels la rotation se fera.
        ii. Entrez le nom générique du fichier journal.
        iii. Entrez le nombre de jours durant lesquels le journal doit être conservé.
        iv. Enregistrer le paramétrage dans le fichier utilisateur 'crontab'.
    b. Si non, aller à l'étape 3.
2. Voulez-vous paramétrer la rotation d'un autre ensemble de journaux ?
    a. Si oui : répéter étape 1.
    b. Si non, aller à l'étape 3.
3. Fin

L'utilisateur va devoir saisir des données pour que le programme effectue quelque chose. La saisie de l'utilisateur doit être sollicitée et mémorisée. L'utilisateur devrait être informé que 'son' crontab va être changé.

5.5. Un exemple Bash script : mysystem.sh

Le script `mysystem.sh` ci-dessous exécute des commandes bien connues (**date**, **w**, **uname**, **uptime**) pour afficher des informations au sujet de la machine et sur vous.

```
tom:~> cat -n mysystem.sh
     1  #!/bin/bash
     2  clear
     3  echo "This is information provided by mysystem.sh.  Le programme démarre maintenant."
     4
     5  echo "Bonjour, $USER"
     6  echo
     7
     8  echo "Nous sommes le `date`, semaine `date +"%V"`."
     9  echo
    10
    11  echo "Ces utilisateurs sont actuellement connectés :"
    12  w | cut -d " " -f 1 - | grep -v USER | sort -u
    13  echo
    14
    15  echo "`uname -s` est le système, `uname -m` le processeur."
    16  echo
    17
    18  echo "Le système fonctionne depuis :"
    19  uptime
    20  echo
    21
    22  echo "C'est pas plus compliqué !"
```

Un script commence toujours par ces 2 caractères : « #! ». Suit le nom du Shell qui exécutera les commandes suivant. Ce script commence en effaçant l'écran à la ligne 2. La ligne 3 fait afficher un message pour informer l'utilisateur de ce qui va se passer. La ligne 5 salue l'utilisateur. Les lignes 6, 9, 13, 16 et 20 ne sont là que pour aérer l'affichage des résultats. La ligne 8 affiche la date du jour et le numéro de la semaine. Ligne 11 encore un message informatif, ainsi que ligne 3, 18 et 22. La ligne 12 formate le résultat de la commande **w** ; la ligne 15 affiche le nom du système d'exploitation et le type de processeur. La ligne 19 donne la durée de fonctionnement du système ainsi que sa charge.

echo et **printf** sont des commandes intégrées de Bash. La première retourne toujours un statut à 0, et affiche simplement ses arguments terminés par un caractère de fin de ligne sur la sortie standard, tandis que la deuxième autorise des options de formatage de la chaîne et renvoie un statut différent de 0 en cas d'échec.

Voici le même script avec l'intégrée **printf** :

```
tom:~> cat mysystem.sh
#!/bin/bash
clear
printf "This is information provided by mysystem.sh.  Le programme démarre maintenant."

printf "Bonjour, $USER.\n\n"

printf "Nous sommes le `date`, semaine `date +"%V"`.\n\n"

printf "Ces utilisateurs sont actuellement connectés :\n"
w | cut -d " " -f 1 - | grep -v USER | sort -u
printf "\n"

printf "`uname -s` est le système, `uname -m` le processeur.\n\n"

printf "Le système fonctionne depuis :\n"
uptime
printf "\n"

printf "C'est pas plus compliqué\n"
```

L'écriture d'un script convivial en insérant des messages est traité au Chapitre 8, *Ecrire des scripts interactifs*.

La localisation standard du Bourne Again Shell

Ceci implique que le programme **bash** est installé dans `/bin`.

Si stdout n'est pas disponible

Si vous exécutez un script par cron, fournir le chemin complet et rediriger les résultats et les erreurs. Du fait que le Shell tourne en mode non-interactif, toute erreur provoquera la fin du script prématurément si vous n'y songez pas.

Les chapitres suivants traiteront en détail les scripts ci-dessus.

5.6. Exemple : init script (NdT d'initialisation)

Un script init démarre les services système sur des machines UNIX et Linux. Les démons de journalisation du système, de gestion des ressources, de contrôle d'accès et de mails en sont des exemples. Ces scripts, aussi appelés scripts de démarrage, sont stockés dans un endroit particulier de votre système, tel que `/etc/rc.d/init.d` ou `/etc/init.d`. Init, le processus initial, lit ses fichiers de configuration et décide quels services il démarre ou arrête en fonction du niveau d'exécution système. Le niveau d'exécution est un état de configuration du système (NdT qui correspond à une utilisation particulière du système) ; chaque système a un niveau d'exécution qui autorise un unique utilisateur, par exemple, pour exécuter des tâches administratives, pour lesquelles le système doit être dans un état aussi stable que possible. Comme par exemple récupérer un fichier système important depuis une sauvegarde. Les niveaux d'exécution de démarrage et d'arrêt sont habituellement aussi configurés.

Les tâches à exécuter au démarrage ou à l'arrêt d'un service sont listées dans le script de lancement. C'est l'une des tâches de l'administrateur système de configurer **init**, de façon que les services soient lancés et stoppés au bon moment. Quand vous êtes confrontés à cette tâche, vous avez besoin d'une bonne compréhension des procédures de démarrage et d'arrêt du système. Nous vous conseillons donc de lire les pages man pour **init** et `inittab` avant de vous lancer dans les scripts d'initialisation.

Voici un exemple très simple qui joue un son au démarrage et à l'arrêt de la machine :

```
#!/bin/bash

# This script is for /etc/rc.d/init.d
# Link in rc3.d/S99audio-greeting and rc0.d/K01audio-greeting
```

```
case "$1" in
'start')
  cat /usr/share/audio/at_your_service.au > /dev/audio
  ;;
'stop')
  cat /usr/share/audio/oh_no_not_again.au > /dev/audio
  ;;
esac
exit 0
```

L'instruction **case** souvent utilisée dans ce genre de script est décrite à la Section 2.5, « Emploi de l'instruction exit et du if ».

6. Résumé

Bash est le Shell GNU, compatible avec Bourne shell, il incorpore beaucoup de fonctionnalités pratiques issues d'autres Shells. Quand le Shell est lancé, il lit ses fichiers de configuration. Les plus importants sont :

- `/etc/profile`
- `~/.bash_profile`
- `~/.bashrc`

Bash agit différemment en mode interactif ; il a un mode à la norme POSIX et un autre restreint.

Les commandes Shell peuvent être classées en 3 groupes : les fonctions Shell, les intégrées Shell et les commandes réparties dans les répertoires du système. Bash admet des intégrées supplémentaires qui ne sont pas connues du classique Bourne Shell.

Les scripts Shell comportent ces commandes agencées selon la syntaxe dictée par Shell. Les scripts sont lus et exécutés ligne par ligne et devraient avoir une structure logique.

7. Exercices

Ces exercices vont vous entraîner pour le prochain chapitre :

1. Où le programme **bash** est localisé sur le système ?
2. Utilisez l'option `--version` pour déterminer quelle version tourne.
3. Quels fichiers de configuration du Shell sont lus quand vous vous faites authentifier par le système au moyen de l'interface graphique puis en ouvrant une fenêtre de console ?
4. Les Shell suivants sont-ils interactifs ? Sont-ils des Shell d'authentification ?
    - Un Shell ouvert en cliquant dans l'arrière plan de votre bureau graphique sur l'icône « Terminal » ou l'équivalent dans un menu.
    - Un Shell que vous obtenez après avoir lancé la commande **ssh *`localhost`***.
    - Un Shell que vous activez en vous connectant à la console en mode texte.
    - Un Shell activé par la commande **xterm &**.
    - Un Shell activé par le script **mysystem.sh**.
    - Un Shell que vous activez sur un système distant, pour lequel vous n'aviez pas besoin de vous authentifier parce que vous utilisez SSH et peut être des clés SSH.
5. Pouvez-vous expliquer pourquoi **bash** ne quitte pas quand vous frapper les touches **Ctrl**+**C** sur la ligne de commande ?
6. Afficher le contenu de la pile des répertoires.

7. Si ce n'est pas déjà le cas, paramétrez l'invite de sorte qu'elle affiche votre localisation dans la hiérarchie système, par exemple ajoutez cette ligne à `~/.bashrc` :

   ```
   export PS1="\u@\h \w> "
   ```

8. Affichez les commandes mémorisées dans la table 'hash' de votre session de Shell en cours.

9. Combien de processus sont en train de tourner sur votre système ? Utilisez **ps** et **wc**, la première ligne de résultat de **ps** n'est pas un processus !

10. Comment afficher le nom du système ? Seulement le nom, rien de plus !

Chapitre 2. Ecrire et corriger des scripts

Table des matières

1. Créer et lancer un script
   - 1.1. Écrire et nommer
   - 1.2. script1.sh
   - 1.3. Exécuter le script
2. Les bases du script
   - 2.1. Quel Shell exécutera le script ?
   - 2.2. Ajout de commentaires
3. Débugger (NdT : corriger) les scripts Bash
   - 3.1. Débugger le script globalement
   - 3.2. Débugger qu'une partie du script
4. Résumé
5. Exercices

Résumé

A la fin de ce chapitre vous serez capable de :

- Ecrire un script simple
- Définir le type de Shell qui doit exécuter le script
- Ajouter des commentaires
- Changer les permissions du script
- Exécuter et débugger un script

1. Créer et lancer un script

1.1. Écrire et nommer

Un script Shell est une séquence de commandes dont vous avez un usage répété. Cette séquence est en principe exécutée en entrant le nom du script sur la ligne de commande. Alternativement, vous pouvez utiliser des scripts pour automatiser des tâches via l'outil cron. Un autre usage des scripts est celui fait par la procédure de démarrage et d'arrêt d'UNIX où les opérations des services et démons sont définies dans des scripts « init ».

Pour créer un script Shell, ouvrez un nouveau fichier avec l'éditeur. N'importe quel éditeur fera l'affaire : **vim**, **emacs**, **gedit**, **dtpad** et cetera sont tous valides. Vous pouvez songer à utiliser un éditeur sophistiqué comme **vim** ou **emacs**, parce qu'ils peuvent être configurés pour reconnaître la syntaxe Shell et Bash et donc peuvent être d'une grande aide en évitant ces erreurs que les débutants font, tel que oublier un crochet ou un point-virgule.

Le vidéo-inverse dans vim

Pour activer le vidéo-inverse dans **vim**, passer la commande

```
:set syntax enable
```

Vous pouvez ajouter ce paramètre à votre fichier `.vimrc` pour rendre permanent cette configuration.

Entrez des commandes UNIX dans ce nouveau fichier, comme vous le feriez sur la ligne de commande. Ainsi que nous l'avons vu dans le chapitre précédent (voir Section 3, « L'exécution de commandes »), les commandes peuvent être des fonctions Shell, des commandes intégrées, des commandes UNIX et le nom d'un autre script.

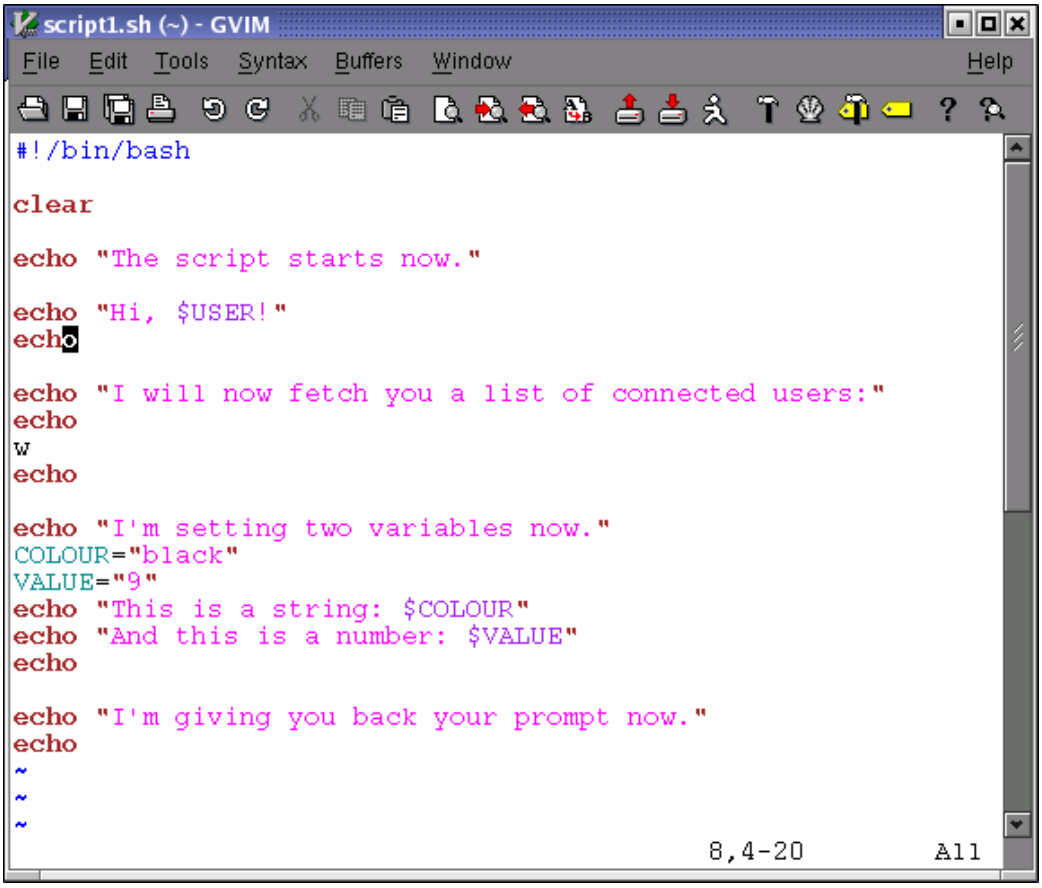
Donnez à votre script un nom significatif qui donne une idée de ce qu'il fait. Assurez vous que ce nom ne soit pas en conflit avec une commande existante. Afin d'éviter des confusions, les noms de scripts souvent finissent par `.sh` ; mais même dans ce cas, il peut y avoir un autre script dans votre système qui porte le même nom. Vérifier avec **which**, **whereis** et les autres commandes qui renvoyent des informations sur les programmes et les fichiers :

```
which -a script_name

whereis script_name

locate script_name
```

1.2. script1.sh

Dans cet exemple nous employons l'intégrée **echo** pour informer l'utilisateur sur ce qui va se dérouler, avant que la tâche qui créera le résultat s'exécute. Il est fortement recommandé d'informer les utilisateurs sur ce que fait le script, de façon à éviter qu'ils deviennent anxieux *parce que le script ne fait rien*. Nous reviendrons sur le sujet de la notification aux utilisateurs au Chapitre 8, *Ecrire des scripts interactifs*.

Figure 2.1. script1.sh



Ecrivez ce script. Ce peut être une bonne idée de créer un répertoire `~/scripts` pour ranger vos scripts. Ajoutez le répertoire au contenu de la variable `PATH` variable :

```
export PATH="$PATH:~/scripts"
```

Si vous êtes tout nouveau avec Bash, utilisez un éditeur de texte qui emploie différentes couleurs pour les différentes constructions syntaxiques. Le code de couleur est une fonction de **vim**, **gvim**, **(x)emacs**, **kwrite** et de beaucoup d'autres éditeurs. Se référer à la documentation de votre éditeur.

> **Des invites différentes**
>
> L'invite varie au long de ce guide selon l'humeur de l'auteur. Ce qui ressemble plus à la vie réelle que l'invite classique *$*. La seule convention que nous avons gardé est que l'invite de *root* finit par *#*.

1.3. Exécuter le script

Le script doit avoir les permissions d'exécution pour le propriétaire afin d'être exécutable. Quand vous définissez des permissions, contrôlez que vous avez obtenu les permissions voulues. Une fois fait, le script peut être lancé comme toute autre commande :

```
willy:~/scripts> chmod u+x script1.sh

willy:~/scripts> ls -l script1.sh
-rwxrw-r--    1 willy   willy          456 Dec 24 17:11 script1.sh

willy:~> script1.sh
Le script démarre.
Salut, willy !

Je vais afficher une liste des utilisateurs connectés :

  3:38pm  up 18 days,  5:37,  4 users,  load average: 0.12, 0.22, 0.15
USER     TTY      FROM              LOGIN@   IDLE   JCPU   PCPU  WHAT
root     tty2     -                Sat 2pm  4:25m  0.24s  0.05s  -bash
willy    :0       -                Sat 2pm   ?     0.00s   ?     -
```

```
willy    pts/3    -               Sat 2pm  3:33m 36.39s 36.39s  BitchX willy ir
willy    pts/2    -               Sat 2pm  3:33m  0.13s  0.06s  /usr/bin/screen

Je définis 2 variables, maintenant.
Ceci est une chaîne : noir
Et ceci est un nombre : 9

Je vous rends la main, maintenant.

willy:~/scripts> echo $COLOUR

willy:~/scripts> echo $VALUE

willy:~/scripts>
```

C'est la façon la plus courante d'exécuter un script. C'est préférable d'exécuter le script comme ça dans un sous-Shell. Les variables, fonctions et alias créés dans le sous-Shell sont seulement connus dans la session Bash de ce sous-Shell. Quand le sous-Shell finit et que le parent reprend le contrôle, tout est réinitialisé et les changements faits dans l'environnement du Shell par le script sont oubliés.

Si vous ne mettez pas le répertoire de `scripts` dans votre `PATH`, et si `.` (le répertoire courant) n'est pas dans le `PATH` non plus, vous pouvez lancer le script comme ça :

```
./script_name.sh
```

Un script peut aussi être explicitement exécuté par un Shell particulier, mais généralement on ne fait ça que pour obtenir un comportement spécial, comme vérifier que le script tourne avec un autre Shell ou afficher une trace pour debugger.

```
rbash script_name.sh

sh script_name.sh

bash -x script_name.sh
```

Le Shell spécifié démarrera en tant que sous-Shell de votre Shell actif et exécutera le script. On le fait quand on veut que le script démarre avec des options ou des conditions spécifiques qui ne sont pas indiquées dans le script.

Si vous ne voulez pas démarrer un nouveau Shell mais exécuter le script dans le Shell courant, vous faites *source* :

```
source script_name.sh
```

> **source = .**
>
> L'intégrée Bash **source** est un synonyme de la commande Bourne shell **.** (dot).

Le script n'a pas besoin de permission d'exécution dans ce cas. Les commandes sont exécutées dans l'environnement du Shell actif, par conséquent tout changement restera tel quel quand le script aura terminé :

```
willy:~/scripts> source script1.sh
--output ommitted--

willy:~/scripts> echo $VALUE
9

willy:~/scripts>
```

2. Les bases du script

2.1. Quel Shell exécutera le script ?

Quand un script s'exécute dans un sous-Shell, vous devriez définir quel Shell doit exécuter ce script.

Le type de Shell pour lequel vous avez écrit le script peut ne pas être celui par défaut de votre système, alors les commandes peuvent ne pas être interprétées par un Shell inadéquat.

La première ligne du script définit le Shell à lancer. Les 2 premiers caractères de la première ligne devraient être *#!*, puis suit le chemin vers le Shell qui doit interpréter les commandes qui suivent. Les lignes blanches sont aussi prises en compte, donc ne commencez pas votre script par une ligne vide.

Dans ce guide, tous les scripts commenceront par la ligne

```
#!/bin/bash
```

Comme indiqué auparavant, ceci implique que le programme Bash doit se trouver dans `/bin`.

2.2. Ajout de commentaires

Vous devriez vous rappeler que vous ne serez peut-être pas la seule personne à lire votre code. Beaucoup d'utilisateurs et d'administrateurs système lancent des scripts qui ont été écrits par d'autres. Si ils veulent voir comment vous avez fait, les commentaires sont utiles pour éclairer le lecteur.

Les commentaires vous rendent aussi la vie plus facile. Par exemple vous avez lu beaucoup de pages man pour obtenir de certaines commandes de votre script un résultat donné. Vous ne vous souviendrez plus de ce qu'il fait après quelques semaines, à moins d'avoir commenté ce que vous avez fait, comment et pourquoi.

Prenez en exemple `script1.sh` et copiez le sur `commented-script1.sh`, que vous éditez de sorte que les commentaires réflètent ce que le script fait. Tout ce qui apparaît après le # sur une ligne est ignoré par le Shell et n'apparaît qu'à l'édition.

```
#!/bin/bash
# Ce script efface le terminal, affiche un message d'accueil et donne des informations
# sur les utilisateurs connectés.  Les 2 exemples de variables sont définis et affichés.

clear                           # efface le terminal

echo "Le script démarre."

echo "Salut, $USER !"           # le signe dollar est employé pour obtenir le contenu d'une variable
echo

echo "Je vais maintenant vous afficher une liste des utilisateurs connectés :"
echo
w                               # montre qui est connecté
echo                            # et ce qu'ils font

echo "Je définis 2 variables maintenant."
COLOUR="noir"                             # définit une variable locale Shell
VALUE="9"                                 # définit une variable locale Shell
echo "Ceci est une chaîne : $COLOUR"      # affiche le contenu de la variable
echo "Et ceci est un nombre : $VALUE"     # affiche le contenu de la variable
echo

echo "Je vous redonne la main maintenant."
echo
```

Dans un script correct, les premières lignes sont habituellement des commentaires sur son but. Puis chaque portion importante de code devrait être commentée autant que la clarté le demande. Les scripts d'init Linux, par exemple, dans le répertoire `init.d` sont généralement bien documentés puisque ils doivent être lisibles et éditables par quiconque utilise Linux.

3. Débugger (NdT : corriger) les scripts Bash

3.1. Débugger le script globalement

Quand les choses ne vont pas comme attendues, vous devez déterminer qu'est-ce qui provoque cette situation. Bash fournit divers moyens pour débugger. La plus commune est de lancer le sous-Shell avec l'option `-x` ce qui fait s'exécuter le script en mode débug. Une trace de chaque commande avec ses arguments est affichée sur la sortie standard après que la commande ait été interprétée mais avant son exécution.

Ceci est le script `commented-script1.sh` exécuté en mode débug. Notez que les commentaires ne sont pas visibles dans la sortie du script.

```
willy:~/scripts> bash -x script1.sh
+ clear

+ echo 'Le script démarre.'
Le script démarre.
+ echo 'Salut, willy !'
Salut, willy !
+ echo

+ echo 'Je vais maintenant vous afficher une liste des utilisateurs connectés :'
Je vais maintenant vous afficher une liste des utilisateurs connectés :
+ echo

+ w
  4:50pm  up 18 days,  6:49,  4 users,  load average: 0.58, 0.62, 0.40
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
root     tty2     -                Sat 2pm  5:36m  0.24s  0.05s  -bash
willy    :0       -                Sat 2pm   ?     0.00s   ?     -
willy    pts/3    -                Sat 2pm 43:13  36.82s 36.82s  BitchX willy ir
willy    pts/2    -                Sat 2pm 43:13   0.13s  0.06s  /usr/bin/screen
+ echo

+ echo 'Je définis 2 variables maintenant.'
Je définis 2 variables maintenant.
+ COLOUR=noir
+ VALUE=9
+ echo 'Ceci est une chaîne : '
Ceci est une chaîne :
+ echo 'Et ceci est un nombre : '
Et ceci est un nombre :
+ echo

+ echo 'Je vous redonne la main maintenant.'
Je vous redonne la main maintenant.
+ echo
```

Fonctionnalités à venir du Bash

Il y a maintenant un débugger complet pour Bash, disponible sur SourceForge. Cependant, cela nécessite une version modifiée de bash-2.05. Cette fonctionnalité devrait être disponible dans la version bash-3.0.

3.2. Débugger qu'une partie du script

Avec l'intégrée **set** vous pouvez exécuter en mode normal ces portions de code où vous êtes sûr qu'il n'y a pas d'erreurs, et afficher les informations de débuggage seulement pour les portions douteuses. Admettons que nous ne soyons pas sûr de ce que fait la commande **w** dans l'exemple `commented-script1.sh`, alors nous pouvons l'entourer dans le script comme ceci :

```
set -x                  # active le mode débug
w
set +x                  # stoppe le mode débug
```

La sortie affiche alors ceci :

```
willy: ~/scripts> script1.sh
Le script démarre.
Salut, willy !

Je vais maintenant vous afficher une liste des utilisateurs connectés :

+ w
  5:00pm  up 18 days,  7:00,  4 users,  load average: 0.79, 0.39, 0.33
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU  WHAT
root     tty2     -                Sat 2pm  5:47m  0.24s  0.05s  -bash
willy    :0       -                Sat 2pm   ?     0.00s   ?     -
willy    pts/3    -                Sat 2pm 54:02  36.88s 36.88s  BitchX willyke
willy    pts/2    -                Sat 2pm 54:02   0.13s  0.06s  /usr/bin/screen
+ set +x

Je définis 2 variables maintenant.
Ceci est une chaîne :
Et ceci est un nombre :
```

```
Je vous redonne la main maintenant.
willy: ~/scripts>
```

On peut basculer du mode activé à désactivé autant de fois que l'on veut dans le script.

La table ci-dessous donne un aperçu d'autres options Bash utiles :

Tableau 2.1. Aperçu des options de débug

Syntaxe abrégée	Syntaxe longue	Effet
set -f	set -o noglob	Désactive la génération de noms de fichiers à partir des métacaractères (globbing).
set -v	set -o verbose	Affiche les lignes fournies au Shell telles qu'elles ont été lues.
set -x	set -o xtrace	Affiche la trace des commandes avant leur exécution.

Le signe - est utilisé pour activer une option Shell et le + pour la désactiver. Ne vous faites pas avoir !

Dans l'exemple qui suit, nous montrons l'usage de ces options depuis la ligne de commande :

```
willy:~/scripts> set -v

willy:~/scripts> ls
ls
commented-scripts.sh    script1.sh

willy:~/scripts> set +v
set +v

willy:~/scripts> ls *
commented-scripts.sh    script1.sh

willy:~/scripts> set -f

willy:~/scripts> ls *
ls: *: No such file or directory

willy:~/scripts> touch *

willy:~/scripts> ls
*   commented-scripts.sh    script1.sh

willy:~/scripts> rm *

willy:~/scripts> ls
commented-scripts.sh    script1.sh
```

De façon alternative, ces modes peuvent être indiqués dans le script lui-même, en ajoutant l'option voulue sur la première ligne de déclaration du Shell. Les options peuvent être combinées, comme c'est généralement le cas pour les commandes UNIX :

```
#!/bin/bash -xv
```

Une fois que vous avez localisé la partie douteuse, vous pouvez ajouter des instructions **echo** devant chaque commande douteuse, de sorte que vous verrez exactement où et pourquoi le résultat n'est pas satisfaisant. Dans le script `commented-script1.sh`, ça pourrait être fait comme ça, toujours en supposant que l'affichage des utilisateurs nous cause des soucis :

```
echo "debug message : avant exécution de la commande w"; w
```

Dans des scripts plus élaborés **echo** peut être inséré pour faire afficher le contenu de variables à différentes étapes du script, afin de détecter les erreurs :

```
echo "Variable VARNAME a la valeur $VARNAME."
```

4. Résumé

Un script Shell est une série réutilisable de commandes saisies dans un fichier de texte exécutable. Tout type d'éditeur de texte peut être utilisé pour écrire des scripts.

Un script commence par*#!* suivi par le chemin vers le Shell qui exécutera les commandes qui viennent après. Les commentaires sont ajoutés au script pour vos propres besoins ultérieurs, et aussi pour le rendre compréhensible par les autres. Mieux vaut avoir trop d'explications que pas assez.

Corriger un script peut être fait grâce aux options Shell de débug. Ces options peuvent être utilisées sur une partie ou sur la totalité du script. Ajouter des commandes **echo** à des endroits judicieusement choisis est aussi un bon moyen de traquer l'erreur.

5. Exercices

Cet exercice vous aidera à créer votre premier script.

1. Ecrire un script au moyen de votre éditeur favori. Le script devrait afficher le chemin de votre répertoire utilisateur et le type de terminal que vous utilisez. De plus il montrera tous les services lancés par le niveau d'exécution 3 de votre système. (tuyau : employez `HOME`, `TERM` et **ls** `/etc/rc3.d/S*`)

2. Ajoutez des commentaires.

3. Ajoutez des informations à destination de l'utilisateur.

4. Changer les permissions de sorte que vous puissiez le lancer.

5. Exécuter le script en mode normal puis en mode débug. Il doit s'exécuter sans erreurs.

6. Faites que le script fasse une erreur : voyez ce qui arrive si la première ligne n'est pas correcte ou si vous libellez mal une commande ou une variable - par exemple déclarez une variable par un nom en majuscule et référencez-la avec le nom en minuscule. Voyez ce que les commentaires de débug affichent dans ce cas.

Chapitre 3. L'environnement du Bash

Table des matières

1. Les fichiers d'initialisation du Shell
    1.1. Les fichiers de configuration qui agissent sur tout le système
    1.2. Les fichiers de configuration utilisateur
    1.3. Modification des fichiers de configuration du Shell
2. Variables
    2.1. Types de variables
    2.2. Créer des variables
    2.3. Exporter les variables
    2.4. Variables réservées
    2.5. Paramètres spéciaux
    2.6. Script à finalités multiples grâce aux variables
3. Echappement et protection de caractères
    3.1. Pourquoi protéger ou 'échapper' un caractère ?
    3.2. Le caractère Echap (escape)
    3.3. Les apostrophes
    3.4. Les guillemets
    3.5. Codage ANSI-C
    3.6. Particularités
4. Le processus d'expansion de Shell
    4.1. Généralité
    4.2. L'expansion d'accolades
    4.3. L'expansion du tilde
    4.4. Paramètre Shell et expansion de variable

4.5. La substitution de commande
4.6. L'expansion arithmétique
4.7. La substitution de processus
4.8. Le découpage de mots
4.9. Expansion de noms de fichier
5. Alias
5.1. Que sont les alias ?
5.2. Créer et supprimer des alias
6. Plus d'options Bash
6.1. Afficher les options
6.2. Changer les options
7. Résumé
8. Exercices

Résumé

Dans ce chapitre nous traiterons des diverses façons de modifier l'environnement du Bash :

- En modifiant les fichiers d'initialisation du Shell
- En utilisant des variables
- En utilisant divers modes d'échappement
- En effectuant des calculs arithmétiques
- En déclarant des alias
- En employant l'expansion et la substitution

1. Les fichiers d'initialisation du Shell

1.1. Les fichiers de configuration qui agissent sur tout le système

1.1.1. /etc/profile

Quand il est invoqué interactivement avec l'option `--login` ou si il est invoqué en tant que **sh**, Bash lit les instructions de `/etc/profile`. Ceci habituellement définit les variables Shell `PATH`, `USER`, `MAIL`, `HOSTNAME` et `HISTSIZE`.

Sur certains systèmes la valeur **umask** est définie dans `/etc/profile` ; sur d'autres ce fichier indique d'autres fichiers de configuration tels que :

- `/etc/inputrc`, le fichier d'initialisation du système Readline où vous pouvez configurer le style de son de la ligne de commande.
- le répertoire `/etc/profile.d`, qui contient les fichiers configurant le comportement de programmes spécifiques dans tout le système.

Tous les paramétrages de l'environnement des utilisateurs devraient être dans ce fichier. Cela peut ressembler à ça :

```
# /etc/profile

# Environnement général du système et programmes de démarrage pour le paramétrage du login

PATH=$PATH:/usr/X11R6/bin

# No core files by default
ulimit -S -c 0 > /dev/null 2>&1

USER="`id -un`"
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"
```

```
HOSTNAME=`/bin/hostname`
HISTSIZE=1000

# Clavier, sonnerie, style d'affichage : le fichier de configuration readline :
if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
    INPUTRC=/etc/inputrc
fi

PS1="\u@\h \W"

export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC PS1

# Source des fichiers d'init de programmes spécifiques (ls, vim, less, ...)
for i in /etc/profile.d/*.sh ; do
    if [ -r "$i" ]; then
        . $i
    fi
done

# Paramètrage de programmes d'initialisation
source /etc/java.conf
export NPX_PLUGIN_PATH="$JRE_HOME/plugin/ns4plugin/:/usr/lib/netscape/plugins"

PAGER="/usr/bin/less"

unset i
```

Ce fichier de configuration définit quelques variables de base de l'environnement du Shell ainsi que quelques variables requises par les utilisateurs lançant Java et/ou des applications Java depuis leur navigateur. Voir Section 2, « Variables ».

Voir Chapitre 7, *Les instructions de condition* pour en savoir plus sur les structures **if** employées dans ce fichier ; Chapitre 9, *Tâches répétitives* traite des boucles tel que **for**.

Le source Bash contient des exemples de fichiers `profile` pour un usage courant ou particulier. Ceux-ci et celui donné en exemple ci-dessus nécessitent des adaptations propres à les faire fonctionner dans votre environnement.

1.1.2. /etc/bashrc

Sur des systèmes qui offrent divers types de Shell, il peut être judicieux de mettre la configuration spécifique à Bash dans ce fichier, parce que `/etc/profile` est aussi lu par d'autres Shell, comme Bourne. Pour éviter que se produisent des erreurs de syntaxe par des Shells qui ne comprennent pas la syntaxe de Bash le fichier de configuration de chaque Shell est différent. Dans un tel cas, le fichier de l'utilisateur `~/.bashrc` devrait pointer sur `/etc/bashrc` afin de l'inclure dans le processus d'initialisation du Shell à la connexion.

Vous pouvez voir aussi que `/etc/profile` sur votre système contient seulement l'environnement Shell et le paramètrage du programme de démarrage, tandis que `/etc/bashrc` contient des définitions pour des fonctions Shell et des alias qui s'appliquent à tout le système . Le fichier `/etc/bashrc` peut être appelé dans `/etc/profile` ou dans les fichiers d'initialisation Shell de chaque utilisateur.

Le source contient des exemples de fichiers `bashrc`, ou vous pouvez en trouver une copie dans `/usr/share/doc/bash-2.05b/startup-files`. Ceci fait partie de `bashrc` et vient avec la documentation Bash :

```
alias ll='ls -l'
alias dir='ls -ba'
alias c='clear'
alias ls='ls --color'

alias mroe='more'
alias pdw='pwd'
alias sl='ls --color'

pskill()
{
        local pid

        pid=$(ps -ax | grep $1 | grep -v grep | gawk '{ print $1 }')
        echo -n "killing $1 (process $pid)..."
        kill -9 $pid
        echo "slaughtered."
}
```

A part les alias généraux, il contient des alias pratiques qui rendent correctes des commandes même

mal libellées. Nous traiterons des alias à la Section 5.2, « Créer et supprimer des alias ». Ce fichier contient une fonction **pskill**, qui sera étudiée en détail au Chapitre 11, *Fonctions*.

1.2. Les fichiers de configuration utilisateur

> **Je n'ai pas ces fichiers ? !**
>
> Ces fichiers peuvent être absent de votre répertoire racine ; créer les au besoin.

1.2.1. ~/.bash_profile

C'est le fichier de configuration principal pour définir l'environnement personnel. Dans ce fichier l'utilisateur peut ajouter des options de configuration supplémentaires ou changer le paramétrage par défaut :

```
franky~> cat .bash_profile
#####################################################################
#                                                                   #
#   .bash_profile file                                              #
#                                                                   #
#   Exécuté depuis le Shell Bash quand vous vous loggez.            #
#                                                                   #
#####################################################################

source ~/.bashrc
source ~/.bash_login
case "$OS" in
  IRIX)
    stty sane dec
    stty erase
    ;;
#  SunOS)
#    stty erase
#    ;;
  *)
    stty sane
    ;;
esac
```

Cet utilisateur configure le caractère retour arrière selon le système d'exploitation sur lequel il se connecte. A part ça, `.bashrc` et `.bash_login` sont lus.

1.2.2. ~/.bash_login

Ce fichier contient des ordres de paramétrage spécifiques qui sont normalement exécutés seulement quand vous vous connectez au système. Dans l'exemple nous l'utilisons pour définir la valeur de **umask** et pour afficher une liste des utilisateurs connectés. Cet utilisateur obtient aussi le calendrier du mois :

```
#########################################################################
#                                                                       #
#   Bash_login file                                                     #
#                                                                       #
#   instructions à exécuter par le Shell Bash à l'étape de connection   #
#   (sourced from .bash_profile)                                        #
#                                                                       #
#########################################################################
#   paramétrer les permissions par défaut
umask 002       # Toutes pour moi, lecture pour le groupe et les autres
#   diverses actions
w
cal `date +"%m"` `date +"%Y"`
```

En l'absence de `~/.bash_profile`, ce fichier sera lu.

1.2.3. ~/.profile

En l'absence de `~/.bash_profile` et `~/.bash_login`, `~/.profile` est lu. Il peut définir le même

paramétrage, qui alors peut être accessible aux autres Shells. Rappelez-vous que les autres Shells peuvent mal interpréter la syntaxe Bash.

1.2.4. ~/.bashrc

Aujourd'hui il est plus courant d'utiliser un Shell hors connection, par exemple quand vous vous connectez via une interface graphique de type X terminal. A l'ouverture de la fenêtre, l'utilisateur n'a pas besoin de fournir un nom et un mot de passe : pas d'authentification. Bash cherche `~/.bashrc` dans ce cas, et de même à la connection ce fichier est référencé dans le fichier de configuration de connection, ce qui évite d'entrer le même paramétrage dans différents fichiers.

Dans ce fichier utilisateur, `.bashrc`, un ensemble de variables pour des programmes spécifiques et d'alias est défini après que le fichier à usage global `/etc/bashrc` ait été lu :

```
franky ~> cat .bashrc
# /home/franky/.bashrc

# Source de définitions globales
if [ -f /etc/bashrc ]; then
       . /etc/bashrc

fi

# shell options

set -o noclobber

# mes variables Shell

export PS1="\[\033[1;44m\]\u \w\[\033[0m\] "
export PATH="$PATH:~/bin:~/scripts"

# mes alias

alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias ss='ssh octarine'
alias ll='ls -la'

# paramétrage de mozilla

MOZILLA_FIVE_HOME=/usr/lib/mozilla
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
MOZ_DIST_BIN=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
export MOZILLA_FIVE_HOME LD_LIBRARY_PATH MOZ_DIST_BIN MOZ_PROGRAM

# paramétrage des fontes
alias xt='xterm -bg black -fg white &'

# paramétrage de BitchX
export IRCNAME="frnk"

# La fin
franky ~>
```

Plus d'exemples se trouvent dans le paquetage Bash. Rappelez-vous que les fichiers exemples peuvent nécessiter des adaptations afin de les faire fonctionner dans votre environnement.

Les alias sont traités à la Section 5, « Alias ».

1.2.5. ~/.bash_logout

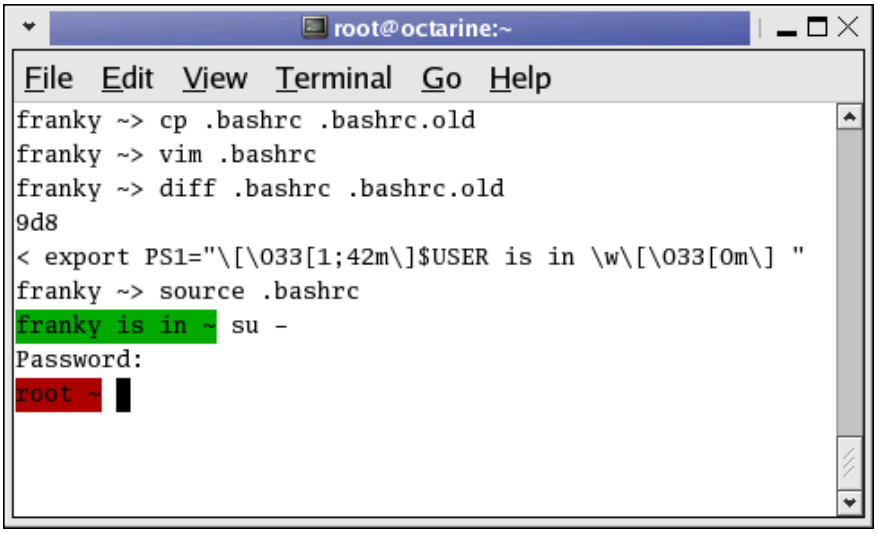
Ce fichier contient des instructions spécifiques pour la procédure de déconnexion. Dans cet exemple, la fenêtre du terminal est effacée à la déconnexion. C'est utile pour les connections à distance qui de cette façon laisse une fenêtre vide après la déconnexion.

```
franky ~> cat .bash_logout
#####################################################################
#                                                                   #
#   Bash_logout file                                                #
#                                                                   #
#  instructions exécutées par le Shell Bash à la déconnexion          #
#                                                                   #
#####################################################################
clear
franky ~>
```

1.3. Modification des fichiers de configuration du Shell

Quand vous modifiez n'importe lequel des fichiers ci-dessus, les utilisateurs doivent soit se reconnecter, soit exécuter (**source**) le fichier modifié afin que prennent effet les modifications. De la deuxième manière, les modifications sont appliquées à la session active du Shell :

Figure 3.1. Différentes invites pour des utilisateurs différents



La plupart des scripts Shell s'exécutent dans leur propre environnement : les processus enfants n'héritent pas des variables du parent à moins que celui-ci les exporte. Exécuter avec **source** un fichier contenant des instructions Shell est un moyen d'appliquer les changements à son propre environnement, de définir des variables dans son Shell actif.

Cet exemple montre aussi le paramétrage de diverses invites pour divers utilisateurs. Dans ce cas, rouge signifie danger. Si vous avez une invite verte, ne vous inquiétez pas trop.

Notez que **source** `resourcefile` est équivalent à **.** `resourcefile`.

Si vous vous trouvez perdu avec tous ces fichiers de configuration, et que vous ne ciblez pas où un certain paramètre est défini, employez **echo**, tout comme pour debugger un script ; voir la Section 3.2, « Débugger qu'une partie du script ». Vous pouvez ajouter des lignes comme celles-ci :

```
echo "Avant exécution de .bash_profile.."
```

ou comme celles-ci :

```
echo "Avant définition de PS1 dans .bashrc:"
export PS1="[la bonne valeur]"
echo "PS1 est défini comme ceci $PS1"
```

2. Variables

2.1. Types de variables

Comme dans l'exemple ci-dessus, les variables Shell sont en majuscule par convention. Bash garde une liste de 2 types de variables :

2.1.1. Les variables Globales

Les variables Globales ou variables d'environnement sont disponibles dans tous les Shells. Les commandes **env** ou **printenv** peuvent être employées pour afficher les variables d'environnement. Ces programmes font partie du paquetage *sh-utils*.

Ci-dessous un affichage fréquent :

```
franky ~> printenv
CC=gcc
CDPATH=.:~:/usr/local:/usr:/
CFLAGS=-O2 -fomit-frame-pointer
COLORTERM=gnome-terminal
CXXFLAGS=-O2 -fomit-frame-pointer
DISPLAY=:0
DOMAIN=hq.xalasys.com
e=
TOR=vi
FCEDIT=vi
FIGNORE=.o:~
G_BROKEN_FILENAMES=1
GDK_USE_XFT=1
GDMSESSION=Default
GNOME_DESKTOP_SESSION_ID=Default
GTK_RC_FILES=/etc/gtk/gtkrc:/nethome/franky/.gtkrc-1.2-gnome2
GWMCOLOR=darkgreen
GWMTERM=xterm
HISTFILESIZE=5000
history_control=ignoredups
HISTSIZE=2000
HOME=/nethome/franky
HOSTNAME=octarine.hq.xalasys.com
INPUTRC=/etc/inputrc
IRCNAME=franky
JAVA_HOME=/usr/java/j2sdk1.4.0
LANG=en_US
LDFLAGS=-s
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
LESSCHARSET=latin1
LESS=-edfMQ
LESSOPEN=|/usr/bin/lesspipe.sh %s
LEX=flex
LOCAL_MACHINE=octarine
LOGNAME=franky
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;
MACHINES=octarine
MAILCHECK=60
MAIL=/var/mail/franky
MANPATH=/usr/man:/usr/share/man/:/usr/local/man:/usr/X11R6/man
MEAN_MACHINES=octarine
MOZ_DIST_BIN=/usr/lib/mozilla
MOZILLA_FIVE_HOME=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
MTOOLS_FAT_COMPATIBILITY=1
MYMALLOC=0
NNTPPORT=119
NNTPSERVER=news
NPX_PLUGIN_PATH=/plugin/ns4plugin/:/usr/lib/netscape/plugins
OLDPWD=/nethome/franky
OS=Linux
PAGER=less
PATH=/nethome/franky/bin.Linux:/nethome/franky/bin:/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:/usr/b
PS1=\[\033[1;44m\]franky is in \w\[\033[0m\]
PS2=More input>
PWD=/nethome/franky
SESSION_MANAGER=local/octarine.hq.xalasys.com:/tmp/.ICE-unix/22106
SHELL=/bin/bash
SHELL_LOGIN=--login
SHLVL=2
SSH_AGENT_PID=22161
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH_AUTH_SOCK=/tmp/ssh-XXmhQ4fC/agent.22106
START_WM=twm
TERM=xterm
TYPE=type
USERNAME=franky
USER=franky
 =/usr/bin/printenv
VISUAL=vi
WINDOWID=20971661
XAPPLRESDIR=/nethome/franky/app-defaults
XAUTHORITY=/nethome/franky/.Xauthority
XENVIRONMENT=/nethome/franky/.Xdefaults
XFILESEARCHPATH=/usr/X11R6/lib/X11/%L/%T/%N%C%S:/usr/X11R6/lib/X11/%l/%T/%N%C%S:/usr/X11R6/lib/X11/%T/%
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
XMODIFIERS=@im=none
XTERMID=
XWINHOME=/usr/X11R6
X=X11R6
YACC=bison -y
```

2.1.2. Variables locales

Les variables locales ne sont visibles que dans le Shell courant. La commande intégrée **set** sans aucune option fait afficher une liste de toutes les variables (y compris les variables d'environnement) et les fonctions. L'affichage sera trié et dans un format réutilisable.

Ci-dessous un fichier diff obtenu par comparaison entre l'affichage de **printenv** et de **set**, après l'avoir expurgé des fonctions qui sont aussi affichées par **set**:

```
franky ~> diff set.sorted printenv.sorted | grep "<" | awk '{ print $2 }'
BASE=/nethome/franky/.Shell/hq.xalasys.com/octarine.aliases
BASH=/bin/bash
BASH_VERSINFO=([0]="2"
BASH_VERSION='2.05b.0(1)-release'
COLUMNS=80
DIRSTACK=()
DO_FORTUNE=
EUID=504
GROUPS=()
HERE=/home/franky
HISTFILE=/nethome/franky/.bash_history
HOSTTYPE=i686
IFS=$'
LINES=24
MACHTYPE=i686-pc-linux-gnu
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PIPESTATUS=([0]="0")
PPID=10099
PS4='+
PWD_REAL='pwd
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-comments:monitor
THERE=/home/franky
UID=504
```

Awk

L'outil GNU Awk est expliqué au Chapitre 6, *Le langage de programmation GNU awk*.

2.1.3. Variables typées selon leur contenu

A part distinguer les variables selon leur portée - locale/globale -, nous pouvons aussi les distinguer par catégories selon ce qu'elles contiennent. De ce point de vue, les variables se distinguent en 4 types :

- Variables de chaîne de caractères (NdT : nous dirons simplement « chaîne » par la suite)
- Variables d'entier
- Variables de constantes
- Variables tableau

Nous traiterons ces types au Chapitre 10, *Un peu plus sur les variables*. Dans l'immédiat, nous allons travailler avec des valeurs de variables de type chaîne et entier.

2.2. Créer des variables

Les variables sont sensibles à la casse et en majuscule par défaut. Donner des noms en minuscule aux variables locales est une convention parfois employée. Cependant, vous êtes libre de nommer comme vous voulez ou de mélanger la casse. Le nom peut aussi comprendre des chiffres, mais un nom commençant par un chiffre n'est pas admis :

```
prompt> export 1number=1
bash: export: `1number=1': not a valid identifier
```

Pour affecter une valeur dans un Shell, la commande est

```
VARNAME="value"
```

Laisser des espaces autour du signe égal causera une erreur. Il est conseillé d'entourer la valeur avec des guillemets lors de l'assignation : ça réduit les risques d'erreurs.

Divers exemples avec minuscules et majuscules, chiffres et espaces :

```
franky ~> MYVAR1="2"

franky ~> echo $MYVAR1
2

franky ~> first_name="Franky"

franky ~> echo $first_name
Franky

franky ~> full_name="Franky M. Singh"

franky ~> echo $full_name
Franky M. Singh

franky ~> MYVAR-2="2"
bash: MYVAR-2=2: command not found

franky ~> MYVAR1 ="2"
bash: MYVAR1: command not found

franky ~> MYVAR1= "2"
bash: 2: command not found

franky ~> unset MYVAR1 first_name full_name

franky ~> echo $MYVAR1 $first_name $full_name
<--no output-->

franky ~>
```

2.3. Exporter les variables

Une variable créée comme celles ci-dessus est seulement visible par le Shell actif. C'est une variable locale: les processus enfants de ce Shell actif ne connaîtront pas cette variable. Afin de faire connaître ces variables à un sous-Shell, nous avons besoin de faire un *export* avec l'intégrée **export**. Les variables exportées sont appelées les variables d'environnement. Définir et exporter sont souvent 2 actions faites en même temps.

```
export VARNAME="valeur"
```

Un sous-Shell peut changer la valeur de variables héritées de son parent, mais cette modification n'affecte pas le parent. Cet exemple en fait la démonstration :

```
franky ~> full_name="Franky M. Singh"

franky ~> bash

franky ~> echo $full_name


franky ~> exit

franky ~> export full_name

franky ~> bash

franky ~> echo $full_name
Franky M. Singh

franky ~> export full_name="Charles the Great"

franky ~> echo $full_name
Charles the Great

franky ~> exit

franky ~> echo $full_name
Franky M. Singh

franky ~>
```

Quand d'abord on essaye de lire la valeur de `full_name` dans un sous-Shell, on n'a rien (**echo** affiche une chaîne nulle). Le sous-Shell finit, et `full_name` est exporté par le parent - une variable peut être exportée après qu'elle ait été assignée. Puis un nouveau sous-Shell est lancé, dans lequel la variable exportée par le parent est visible. Le contenu de la variable est changé, mais la valeur de cette variable pour le parent reste la même.

2.4. Variables réservées

2.4.1. Variables réservées du Bourne Shell

Bash utilise certaines variables Shell de la même manière que Bourne Shell. Dans certains cas, Bash assigne une valeur par défaut à la variable. La table ci-dessous donne un aperçu de ces variables Shell de base :

Tableau 3.1. Variables réservées Bourne Shell

Nom de variable	Définition
CDPATH	Une liste des répertoires, séparés par deux points(:), utilisés comme chemin de recherche pour l'intégrée **cd**.
HOME	Le répertoire racine de l'utilisateur actif ; le chemin par défaut pour l'intégrée **cd**. La valeur de cette variable est aussi utilisée par l'expansion du tilde
IFS	Une liste de caractères qui peuvent séparer les champs ; utilisé par Shell pour découper les mots lors du processus d'expansion.
MAIL	Si à ce paramètre est affecté un nom de fichier et que la variable `MAILPATH` n'est pas définie, Bash informe l'usager de l'arrivée d'un mail dans le fichier spécifié.
MAILPATH	Une liste de fichiers, séparés par deux points (:), dont se sert le Shell régulièrement pour rechercher les nouveaux mails.
OPTARG	La valeur de l'argument de la dernière option traitée par l'intégrée **getopts**.
OPTIND	Le rang de l'argument de la dernière option traitée par l'intégré **getopts**.
PATH	Une liste de répertoires, séparés par deux points(:), dans lesquels le Shell recherche les commandes.
PS1	La principale chaîne d'invite. La valeur par défaut est « '\s-\v\$ ' ».
PS2	La chaîne alternative d'invite. La valeur par défaut est « '> ' ».

2.4.2. Les variables réservées de Bash

Ces variables sont définies ou utilisées par Bash, mais les autres Shells normalement ne les traitent pas spécialement.

Tableau 3.2. Les variables réservées de Bash

Nom de variable	Définition
auto_resume	Cette variable configure la façon dont le Shell interprète la saisie à la ligne de commande comme étant des ordres de contrôle de travaux.
BASH	Le chemin complet où se trouve l'exécutable du Bash actif.

Nom de variable	Définition
BASH_ENV	Si cette variable est définie quand Bash est invoqué pour exécuter un script Shell, sa valeur est interprétée(expansion) et utilisée comme nom de fichier de démarrage à lire avant d'exécuter le script.
BASH_VERSION	Le numéro de version du Bash actif.
BASH_VERSINFO	Un tableau en lecture dont chaque élément mémorise un niveau de la version du Bash actif.
COLUMNS	Utilisé par l'intégrée **select** pour déterminer la largeur du terminal lors de l'affichage de listes de sélection. Automatiquement défini à la réception d'un signal *SIGWINCH*.
COMP_CWORD	Un index dans `${COMP_WORDS}` qui pointe sur le mot où se trouve le curseur.
COMP_LINE	La ligne de commande courante.
COMP_POINT	Index qui point la position du curseur dans la commande courante.
COMP_WORDS	Une variable tableau dont chaque élément renvoie un mot de la commande courante.
COMPREPLY	Une variable tableau d'où Bash tire des interprétations possibles qui ont été générées par une fonction Shell dans le processus de génération.
DIRSTACK	Une variable tableau mémorisant le contenu de la pile de répertoires.
EUID	Le nombre identifiant l'utilisateur actif.
FCEDIT	L'éditeur utilisé par défaut par l'option `-e` de l'intégrée **fc**.
FIGNORE	Une liste de suffixes à ignorer, séparés par deux points (:), quand se produit la génération de noms de fichiers.
FUNCNAME	Contient le nom de fonction si une fonction Shell est en train de s'exécuter.
GLOBIGNORE	Une liste de patrons, séparés par deux points (:), qui sert à définir les fichiers à ignorer lors de la génération de nom de fichiers.
GROUPS	Un tableau qui mémorise les groupes auxquels l'utilisateur appartient.
histchars	Jusqu'à 3 caractères permettant de contrôler l'expansion d'historique, la substitution rapide, et le découpage en *mots*.
HISTCMD	Le numéro d'historique, ou le rang dans la liste d'historique de la commande en cours.
HISTCONTROL	Détermine si la commande en cours est ajoutée au fichier d'historique.
HISTFILE	Le nom de fichier dans lequel l'historique des commandes est conservé. La valeur par défaut est `~/.bash_history`.
HISTFILESIZE	Détermine le nombre maximum de lignes que mémorise le fichier d'historique ; par défaut 500.

Nom de variable	Définition
HISTIGNORE	Une liste de patrons, séparés par deux points (:), qui sert à déterminer quelles commandes sont mémorisées dans l'historique.
HISTSIZE	Détermine le nombre maximum de commandes que mémorise le fichier d'historique ; par défaut 500.
HOSTFILE	Contient le nom d'un fichier au format de `/etc/hosts` qui devrait être lu quand le Shell a besoin du nom de machine hôte.
HOSTNAME	Délivre le nom de la machine hôte.
HOSTTYPE	Une chaîne qui décrit la machine sur laquelle Bash est en train de tourner.
IGNOREEOF	Détermine l'action du Shell quand il reçoit le caractère *EOF* et uniquement celui-là.
INPUTRC	Délivre le nom du fichier d'initialisation de Readline, se substituant à `/etc/inputrc`.
LANG	Utilisé pour tenter de déterminer le particularisme local lorsque qu'aucune variable commençant par `LC_` ne spécifie la catégorie.
LC_ALL	Cette variable se substitue à `LANG` et à toute autre variable `LC_` en spécifiant une catégorie de particularisme local.
LC_COLLATE	Cette variable détermine l'ordre des lettres lors du tri du résultat de l'expansion des noms ainsi que le comportement des expressions des intervalles, des classes d'équivalences, et de la comparaison de chaînes lors de la recherche de motifs et l'expansion des noms de fichiers.
LC_CTYPE	Cette variable détermine l'interprétation des caractères et le comportement des classes de caractères [NdT : ex : [:alpha]] lors de l'expansion des noms de fichiers et de la recherche de patrons.
LC_MESSAGES	Cette variable détermine le particularisme utilisé pour traduire les chaînes entre guillemets précédés par un « $ ».
LC_NUMERIC	Cette variable détermine la catégorie des particularismes employés pour formater les nombres.
LINENO	Le numéro de la ligne en train d'être traitée dans le script ou la fonction Shell.
LINES	Utilisé par l'intégrée **select** pour déterminer la longueur de colonne lors de l'affichage de listes de sélection.
MACHTYPE	Une chaîne qui décrit complètement le type de système sur lequel Bash tourne, dans le format standard GNU CPU-COMPANY-SYSTEM.
MAILCHECK	Intervalle de temps (en secondes) entre 2 vérifications de présence de mail dans le fichier spécifié par `MAILPATH` ou `MAIL`.
OLDPWD	Contient le nom du répertoire précédent accédé par l'intégrée **cd**.

Nom de variable	Définition
OPTERR	Si défini à 1, Bash affiche les messages d'erreur générés par l'intégrée **getopts**.
OSTYPE	Une chaîne décrivant le système d'exploitation sur lequel Bash tourne.
PIPESTATUS	Un tableau contenant une liste des statuts d'exécution des processus les plus récemment exécutés en avant-plan (éventuellement une seule commande).
POSIXLY_CORRECT	Si cette variable est définie quand **bash** démarre, le Shell entre en mode POSIX.
PPID	L'identifiant du process parent du Shell.
PROMPT_COMMAND	Définie, la valeur est interprétée comme une commande à exécuter avant l'affichage de chaque invite (`PS1`).
PS3	La valeur de cette variable est utilisée comme l'invite pour la commande **select**. Par défaut « '#? ' »
PS4	La valeur est l'invite affichée avant que la commande soit affichée en echo quand l'option `-x` est activée ; par defaut « '+ ' ».
PWD	Renvoie le nom de répertoire courant défini par l'intégrée **cd**.
RANDOM	Chaque fois que cette variable est référencée, un entier entre 0 et 32767 est généré. Le fait d'assigner une valeur à cette variable réinitialise le générateur.
REPLY	Paramètre par défaut de l'intégrée **read**.
SECONDS	Renvoie le nombre de secondes écoulées depuis que le Shell est lancé.
SHELLOPTS	Une liste des options Shell activées, séparées par deux points(:).
SHLVL	Valeur augmentée de 1 chaque fois qu'une nouvelle instance de Bash est lancée.
TIMEFORMAT	Cette valeur est un paramètre utilisé pour formater le résultat de chronométrage des instructions exécutées dans un tube (pipeline) lorsque le mot réservé **time** est spécifié.
TMOUT	Défini à une valeur supérieure à zéro, `TMOUT` est considéré comme le temps imparti maximum à l'intégrée **read**. Dans un Shell interactif, la valeur est considérée comme un nombre de secondes durant lesquelles une saisie est attendue. Bash se termine après ce laps de temps si aucune entrée n'est faite.
UID	Le nombre, le véritable identifiant, de l'usagé actif.

Consultez le man Bash, les pages info et doc pour de plus amples explications. Certaines variables sont en lecture seule, d'autres sont définies automatiquement et d'autres perdent leur sens initial quand elles sont redéfinies.

2.5. Paramètres spéciaux

Le Shell considère certains paramètres spécifiquement. Ces paramètres ne peuvent qu'être référencés ; leur affectation n'est pas permise.

Tableau 3.3. Les variables Bash spéciales

Nom	Définition
`$*`	Est remplacé par tous les paramètres positionnels, sauf le premier $0. Quand l'expansion se produit entre guillemets, cela revient à avoir un seul mot avec la valeur de chaque paramètre séparée par le premier caractère de la variable spéciale `IFS`.
`$@`	Est remplacé par tous les paramètres positionnels, sauf le premier $0. Quand l'expansion se produit entre guillemets, chaque paramètre est un mot à part entière.
`$#`	Renvoie le nombre de paramètres positionnels en décimal.
`$?`	Renvoie le statut d'exécution de l'instruction la plus récemment exécutée en avant-plan dans un tube.
`$-`	Renvoie les options déclarées lors de l'invocation de l'intégrée **set**, ou celles positionnées par le Shell lui-même (tel que `-i`).
`$$`	Renvoie l'identifiant du process du Shell.
`$!`	Renvoie l'identifiant du process de la commande la plus récemment exécutée en tâche de fond (asynchrone).
`$0`	Renvoie le nom du Shell ou du script Shell actif.
`$_`	Au démarrage du Shell, contient le nom complet du fichier exécutable actif - script ou Shell - tel que passé dans la liste d'arguments. Ensuite, renvoie le dernier argument de la commande précédente après expansion. Et aussi valorisé avant chaque exécution de commande avec la valeur du chemin complet de cette commande, puis exporté dans l'environnement d'exécution. A la vérification de présence de mails, ce paramètre contient le nom du fichier de mails.

$* versus $@

L'implémentation de « $* » a toujours été un problème et aurait dû être remplacé pratiquement par le comportement de « $@ ». Dans presque tous les cas quand le programmeur utilise « $* », il veut dire « $@ ». « $* » peut être la cause de bugs et même de trou de sécurité dans votre programme.

Les paramètres positionnels sont les mots qui suivent le nom d'un script Shell. Ils définissent les variables `$1`, `$2`, `$3` etc. Autant que nécessaire, ces variables sont ajoutées dans un tableau interne. `$#` mémorise le nombre de paramètres, comme démontré dans ce simple script :

```
#!/bin/bash
```

```
# positional.sh
# Ce script lit 3 paramètres positionnels et les affiche.

POSPAR1="$1"
POSPAR2="$2"
POSPAR3="$3"

echo "$1 est le premier paramètre positionnel, \$1."
echo "$2 est le deuxième paramètre positionnel, \$2."
echo "$3 est le troisième paramètre positionnel, \$3."
echo
echo "Le nombre total de paramètres positionnels est $#."
```

A l'exécution on peut donner autant de paramètres que l'on veut :

```
franky ~> positional.sh un deux trois quatre cinq
un est le premier paramètre positionnel, $1.
deux est le deuxième paramètre positionnel, $2
trois est le troisième paramètre positionnel, $3.

Le nombre total de paramètres positionnels est 5.

franky ~> positional.sh un deux
un est le premier paramètre positionnel, $1.
deux est le deuxième paramètre positionnel, $2
 est le troisième paramètre positionnel, $3.

Le nombre total de paramètres positionnels est 2
```

Plus de précisions sur l'évaluation de ces paramètres au Chapitre 7, *Les instructions de condition* et à la Section 7, « L'intégrée shift ».

Quelques exemples sur les autres paramètres spéciaux :

```
franky ~> grep dictionary /usr/share/dict/words
dictionary

franky ~> echo $_
/usr/share/dict/words

franky ~> echo $$
10662

franky ~> mozilla &
[1] 11064

franky ~> echo $!
11064

franky ~> echo $0
bash

franky ~> echo $?
0

franky ~> ls doesnotexist
ls: doesnotexist: No such file or directory

franky ~> echo $?
1

franky ~>
```

L'utilisateur *franky* lance la commande **grep**, qui a pour effet la valorisation de la variable `_`. L'ID du processus de son Shell est 10662. Après avoir mis un travail en tâche de fond, la variable `!` renvoie l'ID du processus du travail en tâche de fond. Le Shell actif est **bash**. Quand une erreur se produit, `?` renvoie un statut d'exécution différent de 0 (zéro).

2.6. Script à finalités multiples grâce aux variables

En plus de rendre le script plus lisible, les variables vous permettent d'utiliser un même script dans divers environnements ou pour des finalités multiples. Prenez l'exemple suivant, un script très simple qui effectue une sauvegarde du répertoire racine de *franky* vers un serveur distant :

```
#!/bin/bash

# Ce script fait une sauvegarde de mon répertoire personnel.

cd /home

# Ceci crée le fichier archive
tar cf /var/tmp/home_franky.tar franky > /dev/null 2>&1
```

```
# Avant supprimer l'ancien fichier bzip2.  Redirige les erreurs parce que ceci en génère quand l'archiv
# n'existe pas.  Puis crée un nouveau fichier compressé.
rm /var/tmp/home_franky.tar.bz2 2> /dev/null
bzip2 /var/tmp/home_franky.tar

# Copie le fichier vers un autre hôte - nous avons une clé ssh pour effectuer ce travail sans intervent
scp /var/tmp/home_franky.tar.bz2 bordeaux:/opt/backup/franky > /dev/null 2>&1

# Crée un marqueur temporel dans un fichier journal..
date > /home/franky/log/home_backup.log
echo backup succeeded > /home/franky/log/home_backup.log
```

Avant tout, vous avez plus tendance à faire des erreurs si vous saisissez au clavier les noms de fichiers et de répertoires chaque fois que nécessaire. De plus supposez que *franky* veuille donner ce script à *carol*, alors carol aura à faire des modifications par l'éditeur avant de pouvoir sauvegarder son répertoire. De même si *franky* veut se servir du script pour sauvegarder d'autres répertoires. Pour une réutilisation aisée, transformer tous les fichiers, répertoires, nom d'utilisateur, nom d'hôte, etc. en variables. Ainsi, vous n'avez besoin que de modifier la variable une fois, et non pas de modifier chaque occurrence de la chaîne correspondante tout au long du script. Voici un exemple :

```
#!/bin/bash

# Ce script fait une sauvegarde de mon répertoire racine.

# Modifier les valeurs des variables pour que le script tourne pour vous :
BACKUPDIR=/home
BACKUPFILES=franky
TARFILE=/var/tmp/home_franky.tar
BZIPFILE=/var/tmp/home_franky.tar.bz2
SERVER=bordeaux
REMOTEDIR=/opt/backup/franky
LOGFILE=/home/franky/log/home_backup.log

cd $BACKUPDIR

# Ceci crée le fichier d'archive
tar cf $TARFILE $BACKUPFILES > /dev/null 2>&1

# D'abord supprimer l'ancien fichier bzip2.  Redirige les erreurs parce que ceci en génère quand l'arch
# n'existe pas.  Puis crée un nouveau fichier compressé.
rm $BZIPFILE 2> /dev/null
bzip2 $TARFILE

# Copie le fichier vers un autre hôte - nous avons une clé ssh pour effectuer ce travail sans intervent
scp $BZIPFILE $SERVER:$REMOTEDIR > /dev/null 2>&1

# Crée un marqueur temporel dans un fichier journal..
date > $LOGFILE
echo backup succeeded > $LOGFILE
```

Répertoires volumineux et faible bande passante

Tout le monde peut comprendre l'exemple ci-dessus, en utilisant un répertoire réduit et un hôte de son sous-réseau. En fonction de votre bande passante, de la taille du répertoire et de l'endroit du serveur distant, cela peut prendre un temps terriblement long de faire la sauvegarde. Pour les répertoires les plus volumineux et une bande passante faible, employez **rsync** pour garder les répertoires synchronisés entre les 2 machines.

3. Echappement et protection de caractères

3.1. Pourquoi protéger ou 'échapper' un caractère ?

Certaines touches(NdT : séquence de caractères) ont un sens spécial dans un certain contexte. La protection - ou encore citation - est utilisée pour s'échapper du sens spécial de ces caractères ou mots : en d'autres termes l'échappement peut désactiver le comportement spécial de ces caractères, il peut empêcher les mots réservés d'être reconnus comme tel et il peut désactiver l'expansion de paramètres.

3.2. Le caractère Echap (escape)

Le caractère Echap sert à inhiber la signification spéciale d'un caractère unique. Le slash inversé sans guillemets, \, est utilisé comme caractère Echap dans Bash. Il préserve le sens littéral du caractère le suivant, à l'exception de *saut de ligne.* Si un caractère 'saut de ligne' apparaît juste après le slash inversé, cela marque la continuation de la ligne quand elle est plus longue que la largeur du terminal ; le slash inversé est ôté du flot entré et donc en fait ignoré.

```
franky ~> date=20021226

franky ~> echo $date
20021226

franky ~> echo \$date
$date
```

Dans cet exemple, la variable `date` est définie avec une valeur. Le premier **echo** affiche la valeur de la variable, mais dans le second le signe $ est protégé.

3.3. Les apostrophes

Les apostrophes (') sont utilisées pour préserver la valeur littérale des caractères enfermés entre apostrophes. Une apostrophe ne peut pas être enfermée entre apostrophes, même si elle est précédée par un slash inversée.

Continuons avec l'exemple précédent :

```
franky ~> echo '$date'
$date
```

3.4. Les guillemets

Avec les guillemets la valeur littérale de tous les caractères est préservée, sauf pour le $, les apostrophes inversées (``) et le slash inversé.

Le $ et `` conservent leur sens spécial à l'intérieur de guillemets.

Le slash inversé conserve son effet seulement quand il est suivi de $, ``, ", \, et 'saut de ligne'. Au sein de guillemets, les \ sont éliminés du flot entré quand il est suivi d'un de ces caractères. Le \ qui précède les caractères sans sens spécial est laissé en l'état pour être interprété par le Shell.

Un guillemet peut être protégé à l'intérieur de guillemets en le faisant précéder par \.

```
franky ~> echo "$date"
20021226

franky ~> echo "`date`"
Sun Apr 20 11:22:06 CEST 2003

franky ~> echo "I'd say: \"Go for it!\""
I'd say: "Go for it !"

franky ~> echo "\"
More input>"

franky ~> echo "\\"
\
```

3.5. Codage ANSI-C

Les mots de la forme « $'MOT' » sont traités d'une manière spéciale. Le mot se transforme en une chaîne, avec le caractère Echap slash inversé remplacé comme spécifié dans le standard ANSI-C. La séquence d'échappement du slash inversé peut être trouvée dans la documentation Bash.

3.6. Particularités

Une chaîne entre guillemets précédée par un $ sera traitée selon la norme en vigueur. Si cette norme est celle de « C » ou de « POSIX », le $ est ignoré. Si la chaîne est transposée avec remplacement, le résultat est entre guillemets.

4. Le processus d'expansion de Shell

4.1. Généralité

Après que la commande ait été décomposée en *éléments* (voir la Section 4.1.1, « La syntaxe Shell »), ces éléments ou mots sont interprétés ou autrement dit résolus. Il y a 8 sortes d'expansion effectuées, lesquelles vont être traitées dans les sections suivantes dans l'ordre où le processus opère.

Après toutes les sortes d'expansions effectuées, guillemets et apostrophes sont éliminés.

4.2. L'expansion d'accolades

L'expansion d'accolade est un mécanisme par lequel des chaînes peuvent être arbitrairement générées. Les patrons sujets à expansion prennent la forme d'un *préfixe* optionnel, suivi d'une série de chaînes séparées par des virgules, le tout à l'intérieur d'accolades, suivi par un *suffixe* optionnel. Le préfixe enrichit chaque chaîne au début, puis à son tour le suffixe enrichit la fin, résultant en une expansion de gauche à droite.

L'expansion d'accolades peut être imbriquée. Le résultat des chaînes ainsi obtenues n'est pas trié ; l'ordre de gauche à droite est préservé.

```
franky ~> echo sp{el,il,al}l
spell spill spall
```

L'expansion d'accolade est effectuée avant tout autres, et tout caractère spécial en vue d'un autre type d'expansion est préservé dans ce résultat. C'est strictement textuel. Bash n'applique aucune interprétation syntaxique au contexte de l'expansion ou au texte entre accolades. Pour éviter des conflits avec l'expansion de paramètres, la chaîne « ${ » n'est pas éligible à l'expansion d'accolade.

Une forme correcte d'expansion d'accolades doit contenir une accolade ouvrante et fermante non protégée, et au moins une virgule non protégée. Toute forme d'expansion d'accolade incorrecte est laissée telle quelle.

4.3. L'expansion du tilde

Si un mot commence par un tilde non protégé (« ~ »), tous les caractères jusqu'au premier slash non-protégé (ou tous les caractères si il n'y a pas de slash non-protégé) sont considérés comme un *préfixe tilde*. Si aucun des caractères dans le préfixe tilde n'est protégé, ces caractères qui suivent le tilde sont considérés comme un nom de connection possible. Si ce nom de connection est la chaîne nulle, le tilde est remplacé par la valeur de la variable Shell `HOME`. Si `HOME` n'est pas défini, le répertoire racine de l'utilisateur exécutant le Shell est utilisé à la place. Sinon, le préfixe tilde est remplacé par le répertoire racine associé au nom de connection spécifié.

Si le préfixe tilde est « ~+ », la valeur de la variable Shell `PWD` remplace le préfixe tilde. Si le préfixe tilde est « ~- », la valeur de la variable Shell `OLDPWD`, si définie, s'y substitue.

Si les caractères suivant le tilde dans le préfixe consistent en un nombre N, optionnellement préfixé par « + » ou « - », le préfixe tilde est remplacé par l'élément correspondant dans la pile de répertoire, comme il serait affiché par l'intégrée **dirs** invoquée avec, comme argument, le caractère suivant le tilde dans le préfixe tilde. Si le préfixe tilde, sans le tilde, consiste en un nombre sans signe « + » ou « - », « + » est implicite.

Si le nom de connection est invalide, ou si l'expansion de tilde échoue, le mot est laissé tel quel.

Chaque assignation de variable donne lieu à un contrôle sur la présence d'un préfixe tilde non-protégé qui suit immédiatement un « : » ou un « = ». Dans ce cas l'expansion du tilde se

produit. Par conséquent, on peut utiliser un nom de fichier avec tilde dans `PATH`, `MAILPATH`, et `CDPATH`, et le Shell utilise l'expansion du nom.

Exemple :

```
franky ~> export PATH="$PATH:~/testdir"
```

`~/testdir` sera interprété en `$HOME/testdir`, donc si `$HOME` est `/var/home/franky`, le répertoire `/var/home/franky/testdir` sera ajouté au contenu de la variable `PATH`.

4.4. Paramètre Shell et expansion de variable

Le caractère « $ » introduit l'expansion de paramètre, la substitution de commande ou l'expansion arithmétique. Le nom du paramètre - ou symbole - à interpréter peut être enchâssé entre accolades. Elles sont optionnelles mais utiles à la séparation des caractères du symbole à interpréter de ceux suivant immédiatement.

Quand l'accolade est utilisée, le premier « } » - non protégé par un slash inversé ou ni à l'intérieur d'une chaîne entre guillemet, ni à l'intérieur d'une expansion arithmétique incorporée ou d'une substitution de commande ou d'expansion de paramètre - est le signal fermant correspondant.

La forme basique de l'expansion de paramètre est « ${PARAMETRE} ». La valeur de « PARAMETRE » y est substituée. Les accolades sont requises quand « PARAMETRE » est un paramètre positionnel avec un symbole de plus de 1 caractère, ou quand « PARAMETRE » est suivi par un caractère qui ne doit pas être interprété comme faisant parti du symbole.

Si le premier caractère de « PARAMETRE » est un point d'exclamation, Bash considère les caractères suivants de « PARAMETRE » comme étant le symbole de la variable ; cette variable est alors interprétée et utilisée dans la suite de la substitution, plutôt que la valeur de « PARAMETRE » lui-même. Ceci est connu sous le nom d'*expansion indirecte*.

Vous êtes certainement familier avec l'expansion de paramètre directe, parce qu'elle est fréquente même dans les cas les plus simples, tel que celui ci-dessus ou le suivant :

```
franky ~> echo $SHELL
/bin/bash
```

Voici un exemple d'expansion indirecte :

```
franky ~> echo ${!N*}
NNTPPORT NNTPSERVER NPX_PLUGIN_PATH
```

Notez que cela ne donne pas la même chose que **echo `$N*`**.

La construction suivante permet la création du nom de variable si il n'existe pas :

```
${VAR:=value}
```

Exemple :

```
franky ~> echo $FRANKY

franky ~> echo ${FRANKY:=Franky}
Franky
```

Cependant, les paramètres spéciaux, dont les paramètres positionnels, ne doivent pas être affectés par ce biais.

Nous approfondirons l'utilisation de l'accolade dans le traitement des variables au Chapitre 10, *Un peu plus sur les variables*. Les pages info de Bash fournissent aussi d'autres informations.

4.5. La substitution de commande

La substitution de commande permet de remplacer la commande elle-même par son résultat. La substitution de commande survient quand une commande est enchâssée ainsi :

```
$(commande)
```

ou ainsi avec les apostrophes inversées :

```
`commande`
```

Bash effectue l'expansion en exécutant COMMANDE et en la remplaçant par son résultat, avec tous les sauts de lignes éliminés. Les sauts de ligne incorporés ne sont pas éliminés, mais ils peuvent l'avoir été pendant le découpage en mot.

```
franky ~> echo `date`
Thu Feb 6 10:06:20 CET 2003
```

Quand l'ancien signal de substitution - l'apostrophe inversée - est utilisé, le slash inversé conserve son sens littéral sauf si il est suivi de « $ », « ` », ou « \ ». La première apostrophe inversée non précédée d'un slash inversé termine la commande de substitution. Quand la forme « $(COMMANDE) » est utilisée, tous les caractères entre parenthèses font partie de la commande ; aucun n'est traité spécifiquement.

Une substitution de commande peut être incorporée à une autre. Pour en incorporer dans la forme apostrophes inversées, protéger l'apostrophe la plus interne avec des slashs inversés.

Si la substitution apparaît entre guillemets, le découpage en mots et l'expansion de noms de fichiers ne sont pas effectués sur les résultats.

4.6. L'expansion arithmétique

L'expansion arithmétique permet l'évaluation d'une expression arithmétique et la substitution par le résultat. Le format pour l'expansion arithmétique est :

```
$(( EXPRESSION ))
```

L'expression est traitée comme si elle était entre guillemets, mais un guillemet à l'intérieur des parenthèses n'est pas traité spécifiquement. Tous les mots de l'expression font l'objet d'expansion de paramètre, de substitution de commande, et d'élimination d'apostrophe. Une substitution arithmétique peut être incorporée à une autre.

L'évaluation d'une expression arithmétique est faite en entiers de taille fixe sans contrôle de dépassement - bien que la division par zéro soit détectée comme une erreur. Les opérateurs sont à peu près les mêmes que dans le langage de programmation C. Par ordre de priorité décroissante, la liste ressemble à ceci :

Tableau 3.4. Opérateurs arithmétiques

Opérateur	**sens**
VAR++ et VAR--	variable: post-incrément et post-décrément
++VAR et --VAR	variable: pré-incrément et pré-décrément
- et +	moins et plus
! et ~	négation logique et bit à bit
**	exponentiation
*, / et %	multiplication, division, reste
+ et -	addition, soustraction
<< and >>	Décalage des bits à gauche ou à droite
<=, >=, < et >	opérateurs de comparaison
== et !=	égalité et inégalité

Opérateur	sens
&	ET logique
^	OU logique exclusif
|	OU logique
&&	ET logique
||	OU logique
expr ? expr : expr	évaluation conditionnelle
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= et |=	affectations
,	séparateur entre expressions

Les variables Shell sont autorisées comme opérandes ; l'expansion est effectuée avant l'évaluation de l'expression. Dans une expression, les variables Shell peuvent aussi être référencées par le nom sans utiliser la syntaxe d'expansion de paramètre. La valeur d'une variable est évaluée en tant qu'expression arithmétique quand elle est référencée. Une variable Shell doit avoir l'attribut 'entier' positionné pour être utilisée dans une expression.

Une constante commençant par un 0 (zéro) est considérée comme un chiffre octal. Un « 0x » ou « 0X » au début marque l'hexadécimal. Sinon, un nombre prend la forme « [BASE'#']N », où « BASE » est un nombre décimal entre 2 et 64 représentant la base arithmétique, et N un nombre dans cette base. Si « BASE'#' » est omis, alors la base 10 est utilisée. Les chiffres supérieurs à 9 sont représentés par les minuscules, les majuscules, « @ », et « _ », dans cet ordre. Si « BASE » est inférieur ou égale à 36, les minuscules et les majuscules sont interchangeables dans leur représentation des chiffres entre 10 et 35.

Les opérateurs sont évalués par ordre de priorité. Les sub-expressions entre parenthèses sont évaluées d'abord ce qui peut prévaloir sur l'ordre de priorité ci-dessus.

Autant que possible, les utilisateurs de Bash devraient essayer d'utiliser la syntaxe avec les crochets :

```
$[ EXPRESSION ]
```

Cependant, ceci ne fait qu'évaluer l'*EXPRESSION*, et ne teste pas :

```
franky ~> echo $[365*24]
8760
```

Voir Section 1.2.2, « Comparaisons numériques », entre autres, pour des exemples pratiques de scripts.

4.7. La substitution de processus

La substitution de processus est effectuée par les systèmes qui admettent les tubes (pipes) nommés (FIFO : NdT=First In First Out=premier entré, premier sorti) ou la méthode `/dev/fd` de nommage de fichiers ouverts. Ca se présente sous la forme de

```
<(LIST)
```

ou l'option

```
>(LIST)
```

Le processus `LIST` est exécuté avec ses entrées et sorties connectées à un tube ou des fichiers dans `/dev/fd`. Le nom de ce fichier est passé en argument à la commande courante comme le résultat de l'expansion. Si la forme « >(LIST) » est employée, d'écrire dans le fichier alimente l'entrée de `LIST`. Si la forme « <(LIST) » est employée, le fichier passé en argument devrait être lu pour obtenir la sortie de `LIST`. Notez qu'aucun espace ne doit apparaître entre le signe < ou > et la parenthèse gauche, sinon la construction serait interprétée comme une redirection.

Quand la substitution de processus est possible, elle est effectuée simultanément avec l'expansion de variable, la substitution de commande, et l'expansion arithmétique.

Plus de détails à la Section 2.3, « Redirection et descripteurs de fichiers ».

4.8. Le découpage de mots

Le Shell recherche dans le résultat de l'expansion de paramètre, de la substitution de commande, et de l'expansion arithmétique des mots qui n'ont pas été interprétés du fait des guillemets.

Le Shell traite chaque caractère de `$IFS` comme un délimiteur, et découpe le résultat des autres expressions en mots sur la base de ces caractères. Si `IFS` n'est pas déclaré, ou si il vaut exactement « '<space><tab><newline>' », sa valeur par défaut, alors toute suite de caractères `IFS` sert à délimiter les mots. Si `IFS` Si `IFS` a une valeur autre que celle par défaut, alors les suites d'« espace » et « Tab » sont ignorées au début et à la fin du mot, du moment que l'espace est inclut dans `IFS` (un caractère espace `IFS`). Tout caractère dans `IFS` qui n'est pas un espace `IFS`, accolé à un caractère espace `IF`, délimite un champ. Une suite de caractère espace `IFS` est aussi traitée comme un délimiteur. Si la valeur de `IFS` est nulle, le découpage en mots n'intervient pas.

Un argument vide (« "" » or « '' ») est conservé. Un argument ayant une valeur nulle, suite à l'expansion d'un paramètre, est éliminé. Si un paramètre non valorisé est interprété à l'intérieur de guillemets, il en résulte un argument nul qui est conservé.

> **Expansion et découpage en mots**
>
> Si aucune expansion ne se produit, aucun découpage n'est effectué

4.9. Expansion de noms de fichier

Après le découpage en mots, à moins que l'option `-f` ait été utilisée (voir Section 3.2, « Débugger qu'une partie du script »), Bash scanne chaque mot pour les caractères « * », « ? », et « [» Si l'un de ces caractères apparaît, alors le mot est considéré comme étant un *PATRON*, et est remplacé par la liste des noms de fichiers correspondants au patron triée par ordre alphabétique. Si aucun nom de fichier ne correspond, et que l'option Shell `nullglob` est désactivée, le mot est laissé en l'état. Si l'option `nullglob` est activée, et qu'aucune correspondance n'est trouvée, le mot est éliminé. Si l'option Shell `nocaseglob` est activée, la correspondance est recherchée sans considérer la casse des caractères alphabétiques.

Quand un patron sert à la génération de noms de fichiers, le caractère « . » au début d'un nom de fichier ou immédiatement après un slash doit trouver une correspondance explicitement, à moins que l'option Shell `dotglob` soit activée. Lors de la recherche de correspondance de noms de fichiers, le caractère slash doit toujours être explicitement indiqué. Dans les autres cas, le caractère « . » n'est pas traité spécifiquement.

La variable Shell `GLOBIGNORE` peut être utilisée pour restreindre l'ensemble de fichiers en correspondance avec le patron. Si `GLOBIGNORE` est activé, les noms qui correspondent à l'un des patrons dans `GLOBIGNORE` sont retirés de la liste de correspondance. Les noms de fichiers `.` et `..` sont toujours ignorés, même si `GLOBIGNORE` est désactivé. Cependant, déclarer`GLOBIGNORE` a pour effet d'activer l'option Shell `dotglob`, donc tous les autres fichiers commençant par « . » correspondront. Pour garder la possibilité d'ignorer les fichiers commençant par « . », indiquer « .* » comme étant un des patrons à ignorer dans `GLOBIGNORE`. L'option `dotglob` est désactivé quand `GLOBIGNORE` n'est pas déclaré.

5. Alias

5.1. Que sont les alias ?

Un alias permet de substituer un mot à une chaîne de caractère quand il est utilisé comme premier mot d'une commande simple. Le Shell maintient une liste d'alias qui sont déclarés ou invalidés avec les intégrées **alias** et **unalias** . Saisir **alias** sans options pour afficher une liste des alias connus du Shell courant.

```
franky: ~> alias
alias ..='cd ..'
alias ...='cd ../..'
alias ....='cd ../../..'
alias PAGER='less -r'
alias Txterm='export TERM=xterm'
alias XARGS='xargs -r'
alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias e='vi'
alias egrep='grep -E'
alias ewformat='fdformat -n /dev/fd0u1743; ewfsck'
alias fgrep='grep -F'
alias ftp='ncftp -d15'
alias h='history 10'
alias fformat='fdformat /dev/fd0H1440'
alias j='jobs -l'
alias ksane='setterm -reset'
alias ls='ls -F --color=auto'
alias m='less'
alias md='mkdir'
alias od='od -Ax -ta -txC'
alias p='pstree -p'
alias ping='ping -vc1'
alias sb='ssh blubber'
alias sl='ls'
alias ss='ssh octarine'
alias sss='ssh -C server1.us.xalasys.com'
alias sssu='ssh -C -l root server1.us.xalasys.com'
alias tar='gtar'
alias tmp='cd /tmp'
alias unaliasall='unalias -a'
alias vi='eval `resize`;vi'
alias vt100='export TERM=vt100'
alias which='type'
alias xt='xterm -bg black -fg white &'

franky ~>
```

Les alias sont utiles pour spécifier une version par défaut d'une commande qui existe en plusieurs versions sur le système, ou pour spécifier les options par défaut d'une commande. Un autre emploi des alias est de permettre la correction des fautes de frappes.

Le premier mot de chaque commande simple, si il n'est pas entre guillemets, est recherché dans la liste des alias. Si il y est, ce mot est remplacé par le texte correspondant. Le nom d'alias et le texte correspondant peut contenir tout caractère Shell valide, y compris les métacaractères, avec l'exception que le nom d'alias ne doit pas contenir « = ». Le premier mot du texte correspondant est recherché dans les alias, mais si ce mot est le même que celui traité il n'est pas remplacé une deuxième fois. Ceci signifie qu'on peut déclarer **ls** équivalent à **ls `-F`**, par exemple, et Bash n'essayera pas de remplacer récursivement l'alias trouvé. Si le dernier caractère de la valeur de l'alias est un espace ou une tabulation, alors le mot suivant de la commande après l'alias est aussi recherché dans les alias.

Les alias ne sont pas remplacés quand le Shell n'est pas interactif, sauf si l'option `expand_aliases` est activée par l'intégrée **shopt** shell.

5.2. Créer et supprimer des alias

Un alias est créé par l'intégrée **alias** . Pour une déclaration permanente, ajouter la commande **alias** dans l'un de vos scripts d'initialisation ; si vous l'entrez seulement sur la ligne de commande, il sera connu que durant la session.

```
franky ~> alias dh='df -h'

franky ~> dh
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda7             1.3G  272M 1018M  22% /
/dev/hda1             121M  9.4M  105M   9% /boot
/dev/hda2              13G  8.7G  3.7G  70% /home
/dev/hda3              13G  5.3G  7.1G  43% /opt
none                  243M     0  243M   0% /dev/shm
/dev/hda6             3.9G  3.2G  572M  85% /usr
/dev/hda5             5.2G  4.3G  725M  86% /var

franky ~> unalias dh
```

```
franky ~> dh
bash: dh: command not found

franky ~>
```

Bash lit toujours au moins une ligne complète saisie avant d'exécuter une des commandes de cette ligne. L'alias est interprété quand la commande est lue, non pas quand elle est exécutée. De ce fait, une définition d'alias apparaissant sur la même ligne qu'une autre commande ne prendra effet qu'à la lecture de la ligne suivante. Les commandes suivant la définition de l'alias sur la ligne ne seront pas affectées par le nouvel alias. Ce comportement joue aussi quand une fonction est exécutée. Un alias est interprété quand la définition d'une fonction est lue, pas quand la fonction est exécutée, parce que la définition de fonction est elle-même une commande composée. En conséquence, l'alias définit dans une fonction n'est pas utilisable tant que la fonction n'a pas été exécutée. En toute sécurité, toujours définir les alias sur des lignes séparées, et ne pas employer **alias** dans des commandes composées.

Les processus enfants n'héritent pas des alias. Bourne shell (**sh**) ne reconnaît pas les alias.

Plus sur les fonctions au Chapitre 11, *Fonctions*.

> **Les fonctions sont plus rapides**
>
> Les alias sont recherchés après les fonctions et donc leur résolution est plus lente. Alors que les alias sont plus faciles à comprendre, les fonctions Shell sont préférées aux alias pour la plupart des usages.

6. Plus d'options Bash

6.1. Afficher les options

Nous avons déjà abordé un certain nombre d'options Bash utiles à la correction des scripts. Dans cette section, nous aurons une vue plus approfondie des options Bash.

Utilisez l'option `-o` de **set** pour afficher toutes les options Shell :

```
willy:~> set -o
allexport               off
braceexpand             on
emacs                   on
errexit                 off
hashall                 on
histexpand              on
history                 on
ignoreeof               off
interactive-comments    on
keyword                 off
monitor                 on
noclobber               off
noexec                  off
noglob                  off
nolog                   off
notify                  off
nounset                 off
onecmd                  off
physical                off
posix                   off
privileged              off
verbose                 off
vi                      off
xtrace                  off
```

Voir les pages Bash info, section Shell Built-in Commands → The Set Built-in pour une description de chaque option. Beaucoup d'options ont un raccourci de un caractère : l'option `xtrace` , par exemple, équivaut à spécifier **set -x**.

6.2. Changer les options

Les options Shell peuvent être modifiées soit par rapport à celles par défaut à l'appel du Shell, soit au

cours des traitements Shell. Elles peuvent être aussi intégrées aux fichiers de configuration des ressources Shell.

La commande suivante exécute un script en mode compatible POSIX :

```
willy:~/scripts> bash --posix script.sh
```

Pour changer l'environnement temporairement, ou à l'usage d'un script, nous utiliserions plutôt **set**. Employez - (moins) pour activer l'option, + pour la désactiver :

```
willy:~/test> set -o noclobber

willy:~/test> touch test

willy:~/test> date > test
bash: test: cannot overwrite existing file

willy:~/test> set +o noclobber

willy:~/test> date > test
```

L'exemple ci-dessus montre l'usage de l'option `noclobber` , qui évite que les fichiers existants soient écrasés par les opérations de redirection. La même chose joue pour les options à 1 caractère, par exemple `-u`, qui, activée, traite les variables non déclarées comme des erreurs, et quitte un Shell non-interactif quand survient une telle erreur :

```
willy:~> echo $VAR


willy:~> set -u

willy:~> echo $VAR
bash: VAR: unbound variable
```

Cette option est aussi utile pour détecter des valeurs incorrectes affectées à des variables : la même erreur se produira aussi, par exemple, quand une chaîne de caractère est affectée à une variable qui a été déclarée explicitement comme devant contenir une valeur numérique.

Voici un dernier exemple qui illustre l'option `noglob` , laquelle empêche les caractères spéciaux d'être interprétés :

```
willy:~/testdir> set -o noglob

willy:~/testdir> touch *

willy:~/testdir> ls -l *
-rw-rw-r--    1 willy    willy          0 Feb 27 13:37 *
```

7. Résumé

L'environnement Bash peut être configuré globalement et pour chaque utilisateur. Divers fichiers de configuration servent à régler précisément le comportement du Shell.

Ces fichiers contiennent des options Shell, des déclarations de variables, des définitions de fonctions et diverses autres constructions qui nous permettent d'adapter notre environnement.

Mis à part les mots réservés par Bourne Shell, Bash et les paramètres spéciaux, les noms de variables peuvent être donnés assez librement.

Parce que beaucoup de caractères ont un double, voire triple, sens, selon l'environnement, Bash utilise une syntaxe appropriée pour inhiber le sens spécial de un ou plusieurs caractères quand le traitement particulier n'est pas désiré.

Bash emploie diverses méthodes pour l'expansion de la ligne de commande afin de déterminer quelle est la commande à exécuter.

8. Exercices

Pour cet exercice, vous aurez besoin de lire les pages man de **useradd**, parce que nous allons utiliser le répertoire `/etc/skel` pour stocker les fichiers de configuration Shell par défaut, lesquels sont copiés dans le répertoire racine de chaque utilisateur ajouté.

D'abord nous ferons quelques exercices de portée générale sur la déclaration et l'affichage de variables.

1. Créer 3 variables, `VAR1`, `VAR2` et `VAR3` ; les initialiser respectivement aux valeurs « thirteen », « 13 » et « Happy Birthday ».
2. Faire afficher les valeurs des ces 3 variables.
3. Sont-elles des variables locales ou globales ?
4. Supprimer `VAR3`.
5. Pouvez-vous voir les 2 variables restantes dans la nouvelle fenêtre d'un autre terminal ?
6. Modifier `/etc/profile` pour que tous les utilisateurs soient accueillis à la connection (tester).
7. Pour le compte *root*, modifier l'invite pour que s'affiche « Danger !! root travaille en mode \w », de préférence une couleur vive telle que rouge ou rose ou le mode vidéo inversé.
8. Assurez-vous que les utilisateurs nouvellement créés ont aussi une invite personnalisée qui les informe sur quel système et dans quel répertoire ils travaillent. Testez vos modifications en ajoutant un utilisateur et en se connectant avec ce compte.
9. Ecrire un script dans lequel 2 entiers sont assignés à 2 variables. Le script doit calculer la surface d'un rectangle à partir de ces valeurs. Il devrait être aéré avec des commentaires et générer un affichage plaisant.

N'oubliez pas le **chmod** de vos scripts !

Chapitre 4. Expressions régulières

Table des matières

1. Expressions régulières
    - 1.1. Qu'est-ce qu'une expression régulière ?
    - 1.2. Les métacaractères des expressions régulières
    - 1.3. Expressions régulières basiques versus celles étendues
2. Exemples en utilisant grep
    - 2.1. Qu'est-ce que grep ?
    - 2.2. Grep et les expressions régulières
3. La correspondance de patron dans les fonctionnalités Bash
    - 3.1. Intervalle de caractère
    - 3.2. Classes de caractères
4. Résumé
5. Exercices

Résumé

Dans ce chapitre nous abordons :

- L'utilisation des expressions régulières
- Les métacaractères des expressions régulières
- Trouver des patrons dans les fichiers et autres résultats
- Les intervalles de caractères et les classes en Bash

1. Expressions régulières

1.1. Qu'est-ce qu'une expression régulière ?

Une *expression régulière* est un patron qui recouvre un ensemble de chaînes de caractères. Les expressions régulières sont construites de façon analogique aux expressions arithmétiques par l'emploi de diverses opérateurs qui combinent d'autres expressions réduites.

Les briques de base sont les expressions régulières qui correspondent à un seul caractère. La plupart des caractères, incluant toutes les lettres et les chiffres, sont des expressions régulières qui correspondent exactement à elle même. Tout métacaractère portant un sens spécial peut être protégé en le précédant d'un slash inversé.

1.2. Les métacaractères des expressions régulières

Une expression régulière peut être suivi par un ou plusieurs opérateurs de répétition (métacaractère) :

Tableau 4.1. Opérateurs d'expression régulière

Opérateur	Effet
.	Correspond à tout caractère
?	L'élément précédent est optionnel et sera présent au plus une fois.
*	L'élément précédent sera présent zéro fois ou plus.
+	L'élément précédent sera présent une fois ou plus.
{N}	L'élément précédent sera présent exactement N fois.
{N,}	L'élément précédent sera présent N ou plus de fois.
{N,M}	L'élément précédent sera présent au moins N fois, mais pas plus de M fois.
-	représente l'intervalle si il n'est pas premier ou dernier dans une liste ou le dernier point d'un intervalle dans une liste.
^	Correspond à une chaîne vide au début de la ligne ; représente aussi les caractères ne se trouvant pas dans l'intervalle d'une liste.
$	Correspond à la chaîne vide à la fin d'une ligne.
\b	Correspond à la chaîne vide au début ou à la fin d'un mot.
\B	Correspond à la chaîne vide à l'intérieur d'un mot.
\<	Correspond à la chaîne vide au début d'un mot.
\>	Correspond à la chaîne vide à la fin d'un mot.

2 expressions régulières peuvent être concaténées ; l'expression régulière résultant correspond à toute chaîne formée par 2 sous-chaînes concaténées qui respectivement correspondent aux sous-expressions.

2 expressions régulières peuvent être jointes par l'opérateur « | » ; l'expression régulière résultant

correspond à toute chaîne qui correspond à l'une ou l'autre des sous-expressions.

La répétition a la préséance sur la concaténation, laquelle a la préséance sur la jointure. Toute une expression peut être mise entre parenthèses pour éviter ces règles de préséance.

1.3. Expressions régulières basiques versus celles étendues

Dans les expressions régulières basiques les métacaractères « ? », « + », « { », « | », « (», et «) » perdent leur sens spécial ; à la place employez la version avec slash inversé « \? », « \+ », « \{ », « \| », « \(», et « \) ».

Vérifiez dans la documentation de votre système si les commandes admettent les expressions étendues dans les expressions régulières.

2. Exemples en utilisant grep

2.1. Qu'est-ce que grep ?

grep cherche dans les fichiers en entrée les lignes qui contiennent une correspondance dans une liste donnée de patrons. Quand il trouve une correspondance dans une ligne, il copie la ligne sur la sortie standard (par défaut), ou sur tout autre type de sortie que vous avez requise par les options.

Bien que **grep** s'attende à établir une correspondance sur du texte, il n'a pas de limite sur la longueur de la ligne lue autre que celle de la mémoire disponible, et il trouve la correspondance sur n'importe quel caractère dans la ligne. Si le dernier octet d'un fichier en entrée n'est pas un *saut de ligne*, **grep** discrètement en ajoute un. Parce que le saut de ligne est aussi un séparateur dans la liste des patrons, il n'y a pas moyen de repérer les caractères saut de ligne dans le texte.

Quelques exemples :

```
cathy ~> grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
12:operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -v bash /etc/passwd | grep -v nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
news:x:9:13:news:/var/spool/news:
mailnull:x:47:47::/var/spool/mqueue:/dev/null
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
rpc:x:32:32:Portmapper RPC user:/:/bin/false
nscd:x:28:28:NSCD Daemon:/:/bin/false
named:x:25:25:Named:/var/named:/bin/false
squid:x:23:23::/var/spool/squid:/dev/null
ldap:x:55:55:LDAP User:/var/lib/ldap:/bin/false
apache:x:48:48:Apache:/var/www:/bin/false

cathy ~> grep -c false /etc/passwd
7

cathy ~> grep -i ps ~/.bash* | grep -v history
/home/cathy/.bashrc:PS1="\[\033[1;44m\]$USER is in \w\[\033[0m\] "
```

Avec la première commande, l'utilisateur *cathy* affiche les lignes de `/etc/passwd` contenant la chaîne *root*.

Puis elle affiche le numéro des lignes contenant cette chaîne.

Avec la troisième commande elle vérifie quels utilisateurs n'utilisent pas **bash**, mais les comptes avec **nologin** ne sont pas affichés.

Puis elle compte le nombre de comptes qui ont `/bin/false` comme Shell.

La dernière commande affiche les lignes de tous les fichiers dans son répertoire racine qui

commencent par `~/.bash`, excluant les correspondances avec *history*, afin d'exclure des correspondances de `~/.bash_history` qui pourrait contenir la même chaîne en majuscule et minuscule. Remarquez que la recherche se fait sur la *chaîne* « ps », et pas sur la *commande* **ps**.

Maintenant voyons ce que nous pouvons faire d'autre avec grep et des expressions régulières.

2.2. Grep et les expressions régulières

Si vous n'êtes pas sous Linux

Nous utilisons GNU **grep** dans ces exemples, parce qu'il admet les expressions régulières étendues. GNU **grep** se trouve par défaut sur les systèmes Linux. Si vous travaillez sur un système propriétaire, vérifiez avec l'option `-V` la version utilisée. GNU **grep** peut être téléchargé depuis http://gnu.org/directory/.

2.2.1. Ancres de lignes et de mots

A partir de l'exemple précédent, nous voulons maintenant n'afficher que les lignes commençant par la chaîne « root » :

```
cathy ~> grep ^root /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Si nous voulons voir quels comptes n'ont aucun Shell de défini, nous cherchons les lignes finissant par « : » :

```
cathy ~> grep :$ /etc/passwd
news:x:9:13:news:/var/spool/news:
```

Pour vérifier que `PATH` est exporté dans `~/.bashrc`, d'abord sélectionner les lignes « export » puis chercher les lignes commençant avec la chaîne « PATH », afin de ne pas afficher `MANPATH` et autres chaînes possibles :

```
cathy ~> grep export ~/.bashrc | grep '\<PATH'
  export PATH="/bin:/usr/lib/mh:/lib:/usr/bin:/usr/local/bin:/usr/ucb:/usr/dbin:$PATH"
```

Pareillement, \> cherche à la fin d'un mot.

Si vous voulez trouver une chaîne qui est un mot isolé (entre espaces), le mieux est d'employer `-w`, comme dans cet exemple où nous affichons des informations de la partition root :

```
cathy ~> grep -w / /etc/fstab
LABEL=/                 /                       ext3    defaults        1 1
```

Si cette option est absente, toutes les lignes de la table du système de fichiers seront affichées.

2.2.2. Classes de caractères

Une *expression entre crochet* est une liste de caractère entre « [» et «] ». Elle fera correspondre tout caractère présent dans cette liste ; si le premier caractère de la liste est le caret, « ^ », alors elle fera correspondre tout caractère NON présent dans la liste. Par exemple, l'expression régulière « [0123456789] » repère n'importe quelle chiffre.

Dans une expression entre crochet, une *expression intervalle* consiste en 2 caractères séparés par un tiret. Elle repère tout caractère qui se retrouve dans l'intervalle des 2 caractères inclus, employant pour ce faire la séquence du jeux de caractères définit localement. Par exemple, dans le jeux particulier par défaut en C, « [a-d] » est équivalent à « [abcd] ». la plupart des jeux sont ordonnés selon l'ordre du dictionnaire, et dans ces jeux « [a-d] » est classiquement pas équivalent à

« [abcd] » ; cela peut être équivalent à « [aBbCcDd] », par exemple. Pour obtenir l'interprétation classique des expressions entre crochets, vous pouvez employer le jeux de C en déclarant la variable d'environnement `LC_ALL` à la valeur « C ».

Pour finir, des jeux de caractères nommés sont prédéfinis dans des expressions entre crochets. Voir les pages man ou info de **grep** pour plus d'informations au sujet de ces expressions prédéfinies.

```
cathy ~> grep [yf] /etc/group
sys:x:3:root,bin,adm
tty:x:5:
mail:x:12:mail,postfix
ftp:x:50:
nobody:x:99:
floppy:x:19:
xfs:x:43:
nfsnobody:x:65534:
postfix:x:89:
```

Dans cet exemple, toutes les lignes contenant soit « y » ou « f » sont affichées.

2.2.3. Jokers

Employez le « . » pour cibler la correspondance avec un seul caractère. Si vous voulez obtenir une liste de tous les mots d'un dictionnaire anglais de 5 caractères et commençant par « c » et finissant par « h » (pratique pour les mots-croisés) :

```
cathy ~> grep '\<c...h\>' /usr/share/dict/words
catch
clash
cloth
coach
couch
cough
crash
crush
```

Si vous voulez afficher les lignes contenant le caractère littéral point, employez l'option `-F` de **grep**.

Pour cibler plusieurs caractères, employez l'astérisque. Cet exemple cible tous les mots commençant par « c » et finissant par « h » dans le dictionnaire du système :

```
cathy ~> grep '\<c.*h\>' /usr/share/dict/words
caliph
cash
catch
cheesecloth
cheetah
--output omitted--
```

Si vous voulez cibler le caractère astérisque littéral dans un fichier ou un résultat, employez les apostrophes. Cathy dans l'exemple ci-dessous essaye d'abord de trouver le caractère astérisque dans `/etc/profile` sans les apostrophes, ce qui ne ramène aucune ligne. Avec les apostrophes, un résultat est généré :

```
cathy ~> grep * /etc/profile

cathy ~> grep '*' /etc/profile
for i in /etc/profile.d/*.sh ; do
```

3. La correspondance de patron dans les fonctionnalités Bash

3.1. Intervalle de caractère

En plus de **grep** et des expressions régulières, il y a un bon paquet de correspondances que vous pouvez faire directement dans le Shell, sans avoir besoin d'un programme externe.

Comme vous savez déjà, l'astérisque (*) et le point d'interrogation (?) ciblent toute chaîne ou tout caractère, respectivement. Protégez ces caractères spéciaux pour les cibler littéralement :

```
cathy ~> touch "*"

cathy ~> ls "*"
*
```

Mais vous pouvez aussi employer les crochets pour cibler tout caractère ou intervalle de caractères compris, si la paire de caractères est séparée par un tiret. Un exemple :

```
cathy ~> ls -ld [a-cx-z]*
drwxr-xr-x    2 cathy    cathy          4096 Jul 20  2002 app-defaults/
drwxrwxr-x    4 cathy    cathy          4096 May 25  2002 arabic/
drwxrwxr-x    2 cathy    cathy          4096 Mar  4 18:30 bin/
drwxr-xr-x    7 cathy    cathy          4096 Sep  2  2001 crossover/
drwxrwxr-x    3 cathy    cathy          4096 Mar 22  2002 xml/
```

Ceci liste tous les fichiers dans le répertoire racine de *cathy*, commençant par « a », « b », « c », « x », « y » ou « z ».

Si le premier caractère entre crochet est « ! » ou « ^ », tout caractère non inclus sera ciblé. Pour cibler le (« - »), l'inclure en premier ou dernier caractère de l'ensemble. L'ordre dépend du paramétrage local et de la valeur de la variable `LC_COLLATE`, si elle est définie. Rappelez-vous que d'autres paramétrages pourraient interpréter « [a-cx-z] » comme « [aBbCcXxYyZz] » si le tri est fait selon l'ordre du dictionnaire. Si vous voulez être sûr d'avoir l'interprétation classique des intervalles, forcer ce paramétrage en définissant `LC_COLLATE` ou `LC_ALL` à « C ».

3.2. Classes de caractères

Les jeux de caractères peuvent être spécifiés entre crochets, avec la syntaxe **[:CLASS:]**, où CLASS est une classe définie dans le standard POSIX et a une des valeurs

« alnum », « alpha », « ascii », « blank », « cntrl », « digit », « graph », « lower », « print », « punct », « space », « upper », « word » or « xdigit ».

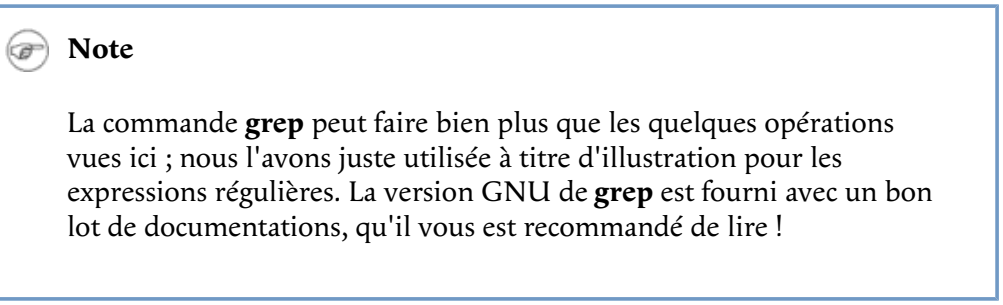
Quelques exemples :

```
cathy ~> ls -ld [[:digit:]]*
drwxrwxr-x    2 cathy   cathy          4096 Apr 20 13:45 2/

cathy ~> ls -ld [[:upper:]]*
drwxrwxr--    3 cathy   cathy          4096 Sep 30  2001 Nautilus/
drwxrwxr-x    4 cathy   cathy          4096 Jul 11  2002 OpenOffice.org1.0/
-rw-rw-r--    1 cathy   cathy        997376 Apr 18 15:39 Schedule.sdc
```

Quand l'option Shell `extglob` est activée (par l'intégrée **shopt**), plusieurs autres opérateurs de correspondance sont reconnus. Plus dans les pages Bash info, section Basic shell features → Shell Expansions → Filename Expansion → Pattern Matching.

4. Résumé

Les expressions régulières sont puissantes pour extraire des lignes particulières d'un fichier ou d'un résultat. Beaucoup de commandes UNIX emploient des expressions régulières : **vim**, **perl**, la base de données PostgreSQL etc. Elles peuvent être interprétées par tout langage et application par l'emploi de bibliothèques externes, et on les retrouve même sur des systèmes non-UNIX. Par exemple, les expressions régulières sont employées dans le tableur Excel qui est fourni avec la suite Office de Microsoft. Dans ce chapitre nous avons eu un avant-goût de la commande **grep** qui est indispensable à tout environnement UNIX.



Bash a des fonctionnalités intégrées pour cibler des patrons et peut reconnaître des classes de caractères et des intervalles.

5. Exercices

Ces exercices vous aideront à maîtriser les expressions régulières.

1. Afficher une liste de tous les utilisateurs de votre système qui se connectent avec le Shell Bash par défaut.
2. Depuis le répertoire `/etc/group`, afficher toutes les lignes commençant avec la chaîne « daemon ».
3. Imprimer toutes les lignes de ce même fichier qui ne contiennent pas la chaîne.
4. Afficher les informations du système local à partir du fichier `/etc/hosts`, afficher le numéro des lignes qui correspondent à la chaîne recherchée et compter le nombre d'occurrences de cette chaîne.
5. Afficher une liste des sous-répertoires de `/usr/share/doc` contenant des informations au sujet des Shell.
6. Combien de fichiers `README` ces sous-répertoires contiennent-ils ? Ne pas prendre en compte les fichiers de la forme « README.a_string ».
7. Faire une liste des fichiers du répertoire racine qui ont changés moins de 10 heures auparavant, en employant **grep**, mais sans prendre les répertoires.
8. Mettre ces commandes dans un script Shell qui générera un résultat lisible.
9. Pouvez-vous trouver une alternative à **wc -l**, avec **grep** ?
10. Avec la table des fichiers du système (`/etc/fstab` par exemple), lister les disques locaux.
11. Faire un script qui contrôle si un utilisateur apparaît dans `/etc/passwd`. Pour cette fois-ci, vous pouvez spécifier le nom dans le script, vous n'êtes pas tenu de travailler avec des paramètres et des conditions à ce stade.
12. Afficher les fichiers de configuration de `/etc` qui contiennent des chiffres dans leur nom.

Chapitre 5. L'éditeur de flot GNU sed

Table des matières

1. Introduction
    1.1. Qu'est-ce que sed ?
    1.2. commandes sed
2. Opérations d'édition de modification
    2.1. Afficher les lignes contenant un patron
    2.2. Exclure les lignes contenant le patron
    2.3. Intervalle de lignes
    2.4. Trouver et remplacer avec sed
3. L'usage en mode différé de sed
    3.1. Lire des commandes sed depuis un fichier
    3.2. Ecrire des fichiers de résultat
4. Résumé
5. Exercices

Résumé

À la fin de ce chapitre vous aurez connaissance des sujets suivants :

- Qu'est-ce que **sed** ?
- L'usage en mode immédiat de **sed**
- Les expressions régulières et l'édition de flot
- Utiliser des commandes **sed** dans un script

> **Ceci est une introduction**
>
> Ces explications sont loin d'être complètes et certainement pas faites pour être considérées comme le manuel utilisateur de **sed**. Ce chapitre est seulement inclus afin de montrer quelques aspects intéressants dans les chapitres suivants, et parce que tout utilisateur avancé devrait avoir une connaissance de base de cet éditeur.
>
> Pour plus d'informations, se référer aux pages man et info de **sed**.

1. Introduction

1.1. Qu'est-ce que sed ?

Un éditeur de flot (Stream EDitor) est utilisé pour effectuer des transformations simples sur du texte lu depuis un fichier ou un tube (NdT : pipe). Le résultat est envoyé sur la sortie standard. La syntaxe de la commande **sed** ne spécifie pas de fichier de sortie, mais les résultats peuvent être mémorisés dans un fichier par le biais de redirection. L'éditeur ne modifie pas le flot d'entrée.

Ce qui distingue **sed** d'autres éditeurs tel **vi** et **ed**, c'est sa faculté de filtrer le texte qu'il obtient par un tube. Vous n'avez pas à interagir avec l'éditeur pendant qu'il travaille ; c'est pourquoi **sed** est parfois appelé un *éditeur par lot*. Cette caractéristique vous permet d'utiliser des commandes d'édition dans un script, ce qui facilite grandement les tâches d'édition répétitives. Quand vous devez effectuer un remplacement de texte dans un grand nombre de fichiers, **sed** est d'une grande aide.

1.2. commandes sed

Le programme **sed** peut effectuer substitutions et suppressions avec des expressions régulières, comme celles utilisées avec la commande **grep** ; voir Section 2, « Exemples en utilisant grep ».

Les commandes d'édition sont similaires à celles utilisées dans l'éditeur **vi** :

Tableau 5.1. Commandes d'édition Sed

Commande	Effet
a\	Ajoute le texte sous la ligne courante.
c\	Remplace le texte de la ligne courante par le nouveau texte.
d	Supprime le texte.
i\	Insère le texte au dessus de la ligne courante.
p	Imprime le texte.
r	Lit un fichier.
s	Cherche et remplace du texte.
w	Ecrit dans un fichier.

A part les commandes d'édition, vous pouvez donner des options à **sed**. Un aperçu se trouve dans le tableau ci-dessous :

Tableau 5.2. Options Sed

Option	Effet
`-e SCRIPT`	Ajoute les commandes de SCRIPT au jeu de commandes à exécuter sur le flot en entrée.
`-f`	Ajoute les commandes contenues dans le fichier SCRIPT-FILE au jeu de commandes à exécuter sur le flot d'entrée.
`-n`	Mode Silencieux.
`-V`	Affiche les informations de version et s'arrête.

Les pages info de **sed** contiennent plus d'informations ; nous ne listons ici que les commandes et les options le plus fréquemment utilisé.

2. Opérations d'édition de modification

2.1. Afficher les lignes contenant un patron

Vous pouvez le faire avec **grep**, bien sûr, mais vous ne pouvez faire un « chercher-remplacer » avec cette commande. Ceci est juste un début.

Voici notre fichier texte d'exemple :

```
sandy ~> cat -n exemple
     1  This is the first line of an example text.
     2  It is a text with erors.
     3  Lots of erors.
     4  So much erors, all these erors are making me sick.
     5  This is a line not containing any errors.
     6  This is the last line.

sandy ~>
```

Nous voulons que **sed** trouve toutes les lignes contenant le patron ciblé, dans ce cas « erors ». Nous employons le **p** pour obtenir le resultat :

```
sandy ~> sed  '/erors/p' exemple
This is the first line of an example text.
It is a text with erors.
It is a text with erors.
Lots of erors.
Lots of erors.
So much erors, all these erors are making me sick.
So much erors, all these erors are making me sick.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Comme vous l'avez remarqué, **sed** affiche le fichier entier, mais les lignes contenant la cible sont affichées 2 fois. Ce n'est pas ce qu'on veut. De sorte à n'afficher que les lignes contenant la cible, employez l'option `-n` :

```
sandy ~> sed -n '/erors/p' exemple
It is a text with erors.
Lots of erors.
So much erors, all these erors are making me sick.

sandy ~>
```

2.2. Exclure les lignes contenant le patron

Nous utilisons le même exemple de fichier texte. Maintenant nous voulons seulement voir les lignes

qui ne contiennent *pas* la chaîne cible :

```
sandy ~> sed '/erors/d' exemple
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

La commande **d** a pour effet d'exclure des lignes de l'affichage.

Les lignes dont le début correspond à un patron donné et la fin à un autre sont affichées comme ça :

```
sandy ~> sed -n '/^This.*errors.$/p' exemple
This is a line not containing any errors.

sandy ~>
```

2.3. Intervalle de lignes

Cette fois nous voulons supprimer les lignes contenant les erreurs. Dans l'exemple ce sont les lignes 2 à 4. Spécifier cet intervalle avec la commande **d** :

```
sandy ~> sed '2,4d' exemple
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Pour afficher le fichier à partir d'une certaine ligne jusqu'à la fin, employez une commande de la sorte :

```
sandy ~> sed '3,$d' exemple
This is the first line of an example text.
It is a text with erors.

sandy ~>
```

Ceci affiche seulement les 2 premières lignes du fichier exemple.

Les commandes suivantes affichent la première ligne contenant le patron « a text », jusqu'à et inclus la prochaine ligne contenant « a line » :

```
sandy ~> sed -n '/a text/,/This/p' exemple
It is a text with erors.
Lots of erors.
So much erors, all these erors are making me sick.
This is a line not containing any errors.

sandy ~>
```

2.4. Trouver et remplacer avec sed

Dans cet exemple de fichier, nous allons maintenant chercher et remplacer les erreurs au lieu de seulement sélectionner les lignes contenant la cible.

```
sandy ~> sed 's/erors/errors/' exemple
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these erors are making me sick.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Comme vous le constatez, ce n'est pas exactement l'effet désiré : en ligne 4, seulement la première occurrence de la chaîne a été remplacée, et il reste encore un 'eror'. Employer la commande **g** pour indiquer à **sed** qu'il doit traiter la ligne entière plutôt que de stopper à la première occurrence trouvée :

```
sandy ~> sed 's/erors/errors/g' exemple
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Pour insérer une chaîne au début de chaque ligne du fichier, par exemple pour citer :

```
sandy ~> sed 's/^/> /' exemple
> This is the first line of an example text.
> It is a text with erors.
> Lots of erors.
> So much erors, all these erors are making me sick.
> This is a line not containing any errors.
> This is the last line.

sandy ~>
```

Insérer une chaîne à la fin de chaque ligne :

```
sandy ~> sed 's/$/EOL/' exemple
This is the first line of an example text.EOL
It is a text with erors.EOL
Lots of erors.EOL
So much erors, all these erors are making me sick.EOL
This is a line not containing any errors.EOL
This is the last line.EOL

sandy ~>
```

Commandes de recherche et remplacement multiples sont séparées avec l'option `-e` :

```
sandy ~> sed -e 's/erors/errors/g' -e 's/last/final/g' exemple
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the final line.

sandy ~>
```

Garder à l'esprit que **sed** par défaut affiche les résultats sur la sortie standard, le plus souvent sur la fenêtre de votre terminal. Si vous voulez sauvegarder le résultat dans un fichier, le rediriger :

```
sed option 'some/expression' file_to_process > sed_output_in_a_file
```

> **Plus d'exemples**
>
> Plein d'exemples **sed** se trouvent dans les scripts de démarrage de votre machine, lesquels sont d'ordinaire dans `/etc/init.d` ou `/etc/rc.d/init.d`. Placez-vous dans le répertoire contenant les scripts d'initialisation de votre système et lancer la commande suivante :
>
> `grep sed *`

3. L'usage en mode différé de sed

3.1. Lire des commandes sed depuis un fichier

De multiples commandes **sed** peuvent être mémorisées dans un fichier et exécutées avec l'option `-f`. Quand un tel fichier est créé, assurez-vous que :

- Aucun espace n'est présent à la fin des lignes.
- Aucune apostrophe n'est employée.

- Quand vous entrez du texte à ajouter ou remplacer, toutes sauf la dernière ligne finit par un slash inversé.

3.2. Ecrire des fichiers de résultat

Produire ce fichier se fait avec l'opérateur de redirection de sortie >. Ceci est un script d'exemple utile à la création de fichier HTML très simple à partir de fichiers texte simples.

```
sandy ~> cat script.sed
1i\
<html>\
<head><title>sed generated html</title></head>\
<body bgcolor="#ffffff">\
<pre>
$a\
</pre>\
</body>\
</html>

sandy ~> cat txt2html.sh
#!/bin/bash

# Ceci est un script simple pour convertir du texte en HTML.
# D'abord nous éliminons tous les caractères de saut de ligne, de sorte que l'ajout
# ne se produise qu'une fois, puis nous remplaçons les sauts de ligne.

echo "converting $1..."

SCRIPT="/home/sandy/scripts/script.sed"
NAME="$1"
TEMPFILE="/var/tmp/sed.$PID.tmp"
sed "s/\n/^M/" $1 | sed -f $SCRIPT | sed "s/^M/\n/" > $TEMPFILE
mv $TEMPFILE $NAME

echo "done."

sandy ~>
```

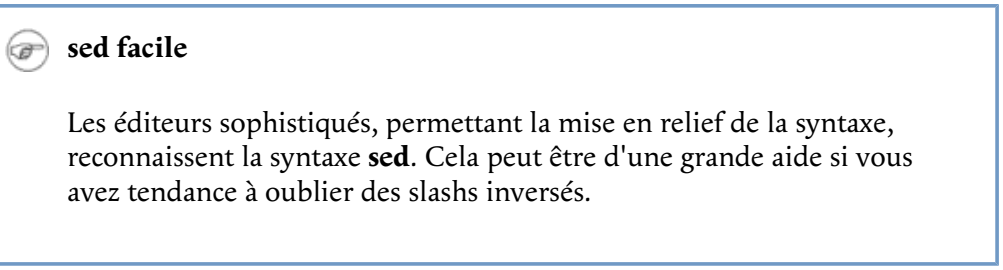
`$1` stocke le premier paramètre d'une commande donnée, dans ce cas le nom du fichier à convertir :

```
sandy ~> cat test
line1
line2
line3
```

Plus sur les paramètres positionnels au Chapitre 7, *Les instructions de condition*.

```
sandy ~> txt2html.sh test
converting test...
done.

sandy ~> cat test
<html>
<head><title>sed generated html</title></head>
<body bgcolor="#ffffff">
<pre>
line1
line2
line3
</pre>
</body>
</html>

sandy ~>
```

Ce n'est pas vraiment la bonne méthode ; c'est juste un exemple pour démontrer le potentiel de **sed**. Voir la Section 3, « Les variables Gawk » pour une solution plus décente à ce problème, avec les constructions **awk** *BEGIN* et *END*.



4. Résumé

L'éditeur par lot **sed** est un outil puissant de travail sur une ligne, lequel peut traiter des flots de données : il peut prendre en entrée les lignes depuis un tube. Ce qui le rend pratique pour un usage différé. L'**éditeur sed** utilise des commandes de type **vi** et accepte les expressions régulières.

L'outil **sed** peut lire des commandes depuis la ligne de commande ou depuis un script. Il est souvent employé pour effectuer des chercher-remplacer sur des lignes contenant un patron.

5. Exercices

Ces exercices sont faits pour montrer plus avant ce que **sed** peut faire.

1. Afficher une liste des fichiers de votre répertoire de `scripts` qui se finissent par « .sh ». Pensez que vous devrez peut-être supprimer l'alias **ls**. Mettre le résultat dans un fichier temporaire.

2. Faire une liste des fichiers de `/usr/bin` qui ont la lettre « a » en second caractère. Mettre le résultat dans un fichier temporaire.

3. Supprimer les 3 premières lignes de chaque fichier temporaire.

4. Afficher sur la sortie standard seulement les lignes contenant le patron « an ».

5. Créer un fichier contenant les commandes **sed** pour effectuer les 2 tâches précédentes. Ajouter une commande supplémentaire à ce fichier qui ajoute une chaîne comme « *** Ceci pourrait avoir quelque chose à voir avec man et les pages man *** » dans la ligne précédant chaque occurrence de la chaîne « man ». Vérifier les résultats.

6. Une liste étendue du répertoire racine, `/`, est utilisée en entrée. Créer un fichier stockant les commandes **sed** qui cherchent les liens symboliques et les fichiers simples. Si un fichier est un lien symbolique, faire précéder la ligne de « --Ceci est un symlink-- ». Si le fichier est un fichier simple, mettre un message sur la ligne, en ajoutant un commentaire du genre « <--- Ceci est un fichier simple ».

7. Créer un script qui affiche les lignes, prises dans un fichier, contenant des espaces à la fin. Ce script devrait faire appel à un script **sed** et afficher des informations utiles à l'utilisateur.

Chapitre 6. Le langage de programmation GNU awk

Table des matières

1. Commencer avec gawk
    1.1. Qu'est-ce que gawk ?
    1.2. Commandes Gawk
2. Le programme d'affichage
    2.1. Afficher les champs sélectionnés
    2.2. Formater les champs
    2.3. La commande print et les expressions régulières
    2.4. Patrons particuliers
    2.5. Les scripts Gawk
3. Les variables Gawk
    3.1. Le séparateur de champs en entrée
    3.2. Les séparateurs de résultat
    3.3. Le nombre d'enregistrements
    3.4. Les variables définies par l'utilisateur
    3.5. Plus d'exemples
    3.6. Le programme printf
4. Résumé
5. Exercices

Résumé

Dans ce chapitre nous aborderons :

- Qu'est-ce que gawk ?
- L'emploi de commandes gawk sur la ligne de commande
- Comment formater du texte avec gawk
- Comment gawk utilise les expressions régulières
- Gawk dans les scripts
- Gawk et les variables

> **Pour s'amuser un peu**
>
> Comme pour **sed**, de nombreux livres ont été écrits sur les nombreuses versions de **awk**. Cette introduction est loin d'être complète et vise seulement à faire comprendre les exemples des chapitres suivants.. Pour approfondir, le mieux est de débuter avec la documentation qui accompagne GNU awk : « GAWK: Effective AWK Programming: A User's Guide for GNU Awk ».

1. Commencer avec gawk

1.1. Qu'est-ce que gawk ?

Gawk est la version GNU du programme couramment disponible sous UNIX **awk** , un autre éditeur par lot populaire. Du fait que le programme **awk** n'est souvent qu'un lien vers **gawk**, nous nous y référerons en tant que **awk**.

La fonction de base de **awk** est de chercher dans des fichiers des lignes ou portion de texte contenant un ou des patrons. Quand on trouve dans la ligne un des patrons, des actions sont effectuées sur cette ligne.

Un programme en **awk** est différent de la plupart des programmes en d'autres langages parce que un programme **awk** est « construit sur les données » : vous décrivez les données que vous voulez traiter puis le traitement à effectuer si elles sont trouvées. La plupart des langages sont « procéduraux. » Vous devez décrire en détail chacune des étapes du processus. En travaillant avec des langages procéduraux, il est généralement plus difficile de clairement décrire les données que le programme traitera. Pour cette raison, un programme **awk** est souvent agréablement facile à lire et à écrire.

> **Qu'est-ce que cela veut dire ?**
>
> dans les années 1970, 3 programmeurs se sont joints pour créer le langage. Leurs noms étaient Aho, Kernighan et Weinberger. Ils prirent la première lettre de chacun de leur nom pour nommer ce langage. Donc le nom du langage aurait aussi bien pu être « wak ».

1.2. Commandes Gawk

Quand vous lancez **awk**, vous spécifiez un *programme* **awk** qui indique à **awk** quoi faire. Le programme consiste en une série de *règles*. (Il peut aussi contenir des définitions de fonctions, de

boucles, des conditions et autres possibilités que nous ignorerons pour l'instant.) Chaque règle spécifie un patron à cibler et une action à effectuer sur les cibles trouvées.

Il y a plusieurs façons de lancer **awk**. Si le programme est court, il est plus facile de le lancer depuis la ligne de commande :

```
awk PROGRAM inputfile(s)
```

Si de multiples changements doivent être fait, peut-être régulièrement sur de multiples fichiers, il est plus facile de mémoriser les commandes **awk** dans un script. Ce qui se lit comme ceci :

```
awk -f PROGRAM-FILE inputfile(s)
```

2. Le programme d'affichage

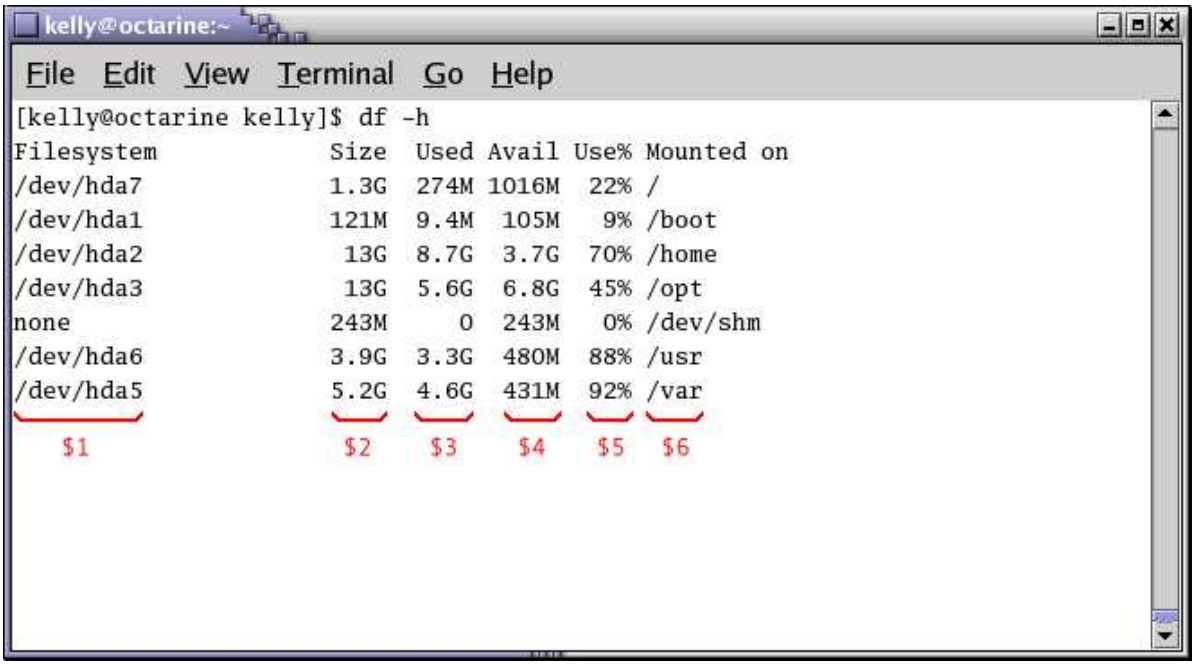
2.1. Afficher les champs sélectionnés

La commande **print** de **awk** affiche les données sélectionnées depuis le fichier d'entrée.

Quand **awk** lit une ligne d'un fichier, il divise la ligne en champs basé sur le *séparateur de champs en entrée*, `FS`, qui est une variable **awk** (voir Section 3.2, « Les séparateurs de résultat »). Cette variable est prédéfinie avec un ou plusieurs espaces et tabulations.

Les variables `$1`, `$2`, `$3`, ..., `$N` stockent les valeurs du premier, second, troisième jusqu'au dernier champ de la ligne traitée. La variable `$0` (zéro) stocke la valeur de la ligne entière. Ceci est illustré dans l'image ci-dessous, où nous voyons 6 colonnes dans l'affichage de la commande **df** :

Figure 6.1. Les champs dans awk



Dans le résultat de **ls -l**, il y a 9 colonnes. L'instruction **print** utilise ces champs comme ceci :

```
kelly@octarine ~/test> ls -l | awk '{ print $5 $9 }'
160orig
121script.sed
120temp_file
126test
120twolines
441txt2html.sh

kelly@octarine ~/test>
```

Cette commande a affiché la 5ème colonne d'une longue liste de fichiers, contenant la taille du fichier, et la dernière colonne, contenant le nom du fichier. Ce résultat est peu lisible à moins

d'utiliser le moyen ad hoc qui est de séparer les colonnes que vous voulez voir afficher par une virgule. Dans ce cas, le caractère séparateur d'affichage par défaut, souvent un espace, sera inséré entre chaque champs de résultat.

2.2. Formater les champs

Sans formater, avec seulement le séparateur de résultat, l'affichage est peu lisible. En insérant quelques tabulations et une chaîne pour indiquer la nature du champs ce sera bien mieux :

```
kelly@octarine ~/test> ls -ldh * | grep -v total | \
awk '{ print "Size is " $5 " bytes for " $9 }'
Size is 160 bytes for orig
Size is 121 bytes for script.sed
Size is 120 bytes for temp_file
Size is 126 bytes for test
Size is 120 bytes for twolines
Size is 441 bytes for txt2html.sh

kelly@octarine ~/test>
```

Notez l'effet du slash inversé, qui permet de continuer une entrée trop longue sur la ligne suivante sans que le Shell interprète cela comme des commandes distinctes. Alors qu'une ligne de commande peut être en théorie de taille illimitée, celle du moniteur ne l'est pas, et encore moins celle du papier. L'usage du slash inversé permet aussi le copier/coller de la ligne dans une fenêtre de terminal.

L'option `-h` de **ls** permet d'obtenir un format lisible de la taille des gros fichiers. L'affichage d'une longue liste avec la somme des blocs du répertoire indiquée est produite quand un répertoire est le paramètre. Cette ligne est inutile pour nous, donc nous avons mis un astérisque. Nous avons aussi ajouté l'option `-d` pour la même raison, dans le cas où l'expansion de l'astérisque donne un répertoire.

Le slash inversé dans cet exemple marque la continuation de la ligne. Voir Section 3.2, « Le caractère Echap (escape) ».

On peut prendre en compte autant de colonnes que l'on veut et même bouleverser l'ordre. Dans l'exemple ci-dessous on en trouve la démonstration qui affiche les partitions les plus critiques :

```
kelly@octarine ~> df -h | sort -rnk 5 | head -3 | \
awk '{ print "Partition " $6 "\t: " $5 " full!" }'
Partition /var  : 86% full!
Partition /usr  : 85% full!
Partition /home : 70% full!

kelly@octarine ~>
```

Le tableau ci-dessous donne un aperçu des caractères spéciaux de formatage :

Tableau 6.1. Caractères de formatage pour gawk

Séquence	sens
\a	sonnerie
\n	Saut de ligne
\t	Tabulation

L'apostrophe, le signe Dollar et autres métacaractères devraient être protégés avec un slash inversé .

2.3. La commande print et les expressions régulières

Une expression régulière peut être utilisée comme patron en l'enfermant entre slashs. L'expression régulière est alors comparée à chaque enregistrement de texte. La syntaxe est celle-ci :

```
awk 'EXPRESSION { PROGRAM }' file(s)
```

L'exemple suivant affiche seulement les informations des disques locaux, les systèmes de fichiers

réseaux n'y sont pas :

```
kelly is in ~> df -h | awk '/dev\/hd/ { print $6 "\t: " $5 }'
/       : 46%
/boot   : 10%
/opt    : 84%
/usr    : 97%
/var    : 73%
/.vol1  : 8%

kelly is in ~>
```

Le Slash doit être protégé, parce qu'il a un sens spécial pour le programme **awk**.

Ci-dessous un autre exemple où nous cherchons dans le répertoire `/etc` les fichiers qui se terminent par « .conf » et qui commencent par « a » *ou* « x », en employant des expressions régulières étendues :

```
kelly is in /etc> ls -l | awk '/\<(a|x).*\.conf$/ { print $9 }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf

kelly is in /etc>
```

Cet exemple illustre le sens spécial du point dans les expressions régulières : le premier indique que nous voulons cibler tout caractère après la première chaîne ciblée, le second est protégé parce que il fait partie d'une chaîne à cibler (la fin du nom de fichier).

2.4. Patrons particuliers

Afin de faire précéder le résultat par un commentaire, employer l'instruction **BEGIN** :

```
kelly is in /etc> ls -l | \
awk 'BEGIN { print "Files found:\n" } /\<[a|x].*\.conf$/ { print $9 }'
Files found:
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf

kelly is in /etc>
```

L'instruction **END** peut être ajoutée pour insérer du texte après que le flot en entrée ait été entièrement traité :

```
kelly is in /etc> ls -l | \
awk '/\<[a|x].*\.conf$/ { print $9 } END { print \
"Can I do anything else for you, mistress?" }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
Can I do anything else for you, mistress?

kelly is in /etc>
```

2.5. Les scripts Gawk

Au fur et à mesure que les commandes deviennent complexes, vous voudrez les mémoriser dans un script, pour être réemployées. Un script **awk** contient des instructions **awk** définissant des patrons et des actions.

Pour illustrer nous allons produire un rapport qui affiche nos partitions les plus pleines. Voir Section 2.2, « Formater les champs ».

```
kelly is in ~> cat diskrep.awk
BEGIN { print "*** WARNING WARNING WARNING ***" }
/\<[8|9][0-9]%/ { print "Partition " $6 "\t: " $5 " full!" }
END { print "*** Donnez de l'argent pour un nouveau disque VITE ! ***" }

kelly is in ~> df -h | awk -f diskrep.awk
*** WARNING WARNING WARNING ***
Partition /usr  : 97% full!
```

```
*** Donnez de l'argent pour un nouveau disque VITE ! ***
kelly is in ~>
```

awk d'abord affiche le message de début, puis formate toutes les lignes qui contiennent un 8 ou un 9 au début de chaque mot, suivi par un autre chiffre et le signe de pourcentage. Un message final est ajouté.

> **Mise en relief de la syntaxe**
>
> Awk est un langage de programmation . Sa syntaxe est reconnue par la plupart des éditeurs qui font la mise en relief de la syntaxe comme pour d'autres langages tel que C, Bash, HTML, etc.

3. Les variables Gawk

Tandis que **awk** traite le fichier en entrée, il utilise plusieurs variables. Certaines sont modifiables, d'autres sont en lecture.

3.1. Le séparateur de champs en entrée

Le *séparateur de champs*, qui est soit un simple caractère, soit une expression régulière, contrôle la façon dont **awk** découpe l'enregistrement entré en champs. L'enregistrement en entrée est examiné à la recherche de séquences de caractères qui correspondent au séparateur défini ; les champs eux-mêmes sont les textes entre chaque séparateur.

Le séparateur de champs est défini par la variable intégrée `FS`. Notez que cette variable et `IFS` utilisée par les Shell compatible POSIX sont distinctes.

La valeur de la variable de séparateur de champs peut être changée dans le programme **awk** avec l'opérateur d'assignement =. Souvent le bon moment pour faire ce changement c'est au début de l'exécution avant qu'aucune entrée n'ait été traitée, de sorte que le tout premier enregistrement est lu avec le séparateur idoine. Pour ce faire employer le patron spécial **BEGIN**.

Dans l'exemple ci-dessous, nous écrivons une commande qui affiche tous les utilisateurs de votre système avec une description :

```
kelly is in ~> awk 'BEGIN { FS=":" } { print $1 "\t" $5 }' /etc/passwd
--output omitted--
kelly   Kelly Smith
franky  Franky B.
eddy    Eddy White
willy   William Black
cathy   Catherine the Great
sandy   Sandy Li Wong

kelly is in ~>
```

Dans un script **awk**, cela ressemblerait à ça :

```
kelly is in ~> cat printnames.awk
BEGIN { FS=":" }
{ print $1 "\t" $5 }

kelly is in ~> awk -f printnames.awk /etc/passwd
--output omitted--
```

Choisir un séparateur de champs d'entrée soigneusement pour éviter des soucis. Un exemple pour illustrer ceci : disons que vous obtenez l'entrée sous la forme de lignes qui ressemble à ça :

« Sandy L. Wong, 64 Zoo St., Antwerp, 2000X »

Vous écrivez une commande ou un script, qui affiche le nom de la personne dans cet enregistrement :

```
awk 'BEGIN { FS="," } { print $1, $2, $3 }' inputfile
```

Mais une personne pourrait avoir un doctorat, et ça pourrait s'écrire comme ça :

« Sandy L. Wong, Doctorat, 64 Zoo St., Antwerp, 2000X »

Votre **awk** donnera un mauvais résultat sur cette ligne. Au besoin faire un **awk** supplémentaire ou un **sed** pour uniformiser le format des données en entrée.

Le séparateur de champs en entrée est par défaut un ou des espaces ou des tabulations.

3.2. Les séparateurs de résultat

3.2.1. Les séparateurs de champs de résultat

Les champs sont habituellement séparés par des espaces dans le résultat. Ceci est visible quand vous employez la syntaxe correcte pour la commande **print** où les paramètres sont séparés par des virgules :

```
kelly@octarine ~/test> cat test
record1         data1
record2         data2

kelly@octarine ~/test> awk '{ print $1 $2}' test
record1data1
record2data2

kelly@octarine ~/test> awk '{ print $1, $2}' test
record1 data1
record2 data2

kelly@octarine ~/test>
```

Si vous ne mettez pas de virgule, **print** considérera les éléments de résultat comme un seul argument, c'est à dire il omet l'emploi du *séparateur de résultat* par défaut, `OFS`.

N'importe quel caractère peut être employé comme séparateur de champs de résultat en définissant cette variable intégrée.

3.2.2. Le séparateur d'enregistrement de résultat

Le résultat d'une instruction **print** est appelée un *enregistrement de résultat.* Chaque commande **print** produit un enregistrement de résultat, et ajoute une chaîne appelée le *séparateur d'enregistrement de résultat*, `ORS` (NdT : output record separator). La valeur par défaut de cette variable est « \n », le caractère saut de ligne. Donc, chaque instruction **print** génère une ligne distincte.

Pour modifier la façon dont les champs et les enregistrements de résultat sont séparés, assignez une autre valeur à `OFS` et `ORS` :

```
kelly@octarine ~/test> awk 'BEGIN { OFS=";" ; ORS="\n-->\n" } \
{ print $1,$2}' test
record1;data1
-->
record2;data2
-->

kelly@octarine ~/test>
```

Si la valeur de `ORS` ne contient pas de saut de ligne, le résultat global tiendra sur une seule ligne.

3.3. Le nombre d'enregistrements

L'intégré `NR` stocke le nombre d'enregistrements qui sont traités. Il est incrémenté après la lecture d'une nouvelle ligne d'entrée. Vous pouvez l'utiliser à la fin pour compter le nombre total d'enregistrements, ou à chaque enregistrement de résultat :

```
kelly@octarine ~/test> cat processed.awk
```

```
BEGIN { OFS="-" ; ORS="\n--> done\n" }
{ print "Record number " NR ":\t" $1,$2 }
END { print "Number of records processed: " NR }

kelly@octarine ~/test> awk -f processed.awk test
Record number 1:        record1-data1
--> done
Record number 2:        record2-data2
--> done
Number of records processed: 2
--> done

kelly@octarine ~/test>
```

3.4. Les variables définies par l'utilisateur

En plus des variables intégrées, vous pouvez définir les vôtres. Quand **awk** rencontre une référence à une variable qui n'existe pas (qui n'est pas prédéfinie), la variable est créée et initialisée à une chaîne nulle. Pour toutes les références suivantes, la valeur de la variable est la dernière valeur affectée. Une variable peut être une chaîne ou une valeur numérique. Le contenu des champs en entrée peut aussi être affecté à une variable.

Une valeur peut être assignée directement par l'opérateur =, ou vous pouvez utiliser la valeur courante de la variable combinée avec d'autres opérateurs :

```
kelly@octarine ~> cat revenues
20021009        20021013        consultancy     BigComp         2500
20021015        20021020        training        EduComp         2000
20021112        20021123        appdev          SmartComp       10000
20021204        20021215        training        EduComp         5000

kelly@octarine ~> cat total.awk
{ total=total + $5 }
{ print "Send bill for " $5 " dollar to " $4 }
END { print "--------------------------------\nTotal revenue: " total }

kelly@octarine ~> awk -f total.awk test
Send bill for 2500 dollar to BigComp
Send bill for 2000 dollar to EduComp
Send bill for 10000 dollar to SmartComp
Send bill for 5000 dollar to EduComp
--------------------------------
Total revenue: 19500

kelly@octarine ~>
```

Les raccourcis de type C comme **VAR+= value** sont aussi acceptés.

3.5. Plus d'exemples

L'exemple de la Section 3.2, « Ecrire des fichiers de résultat » devient bien plus facile quand on utilise un script **awk** :

```
kelly@octarine ~/html> cat make-html-from-text.awk
BEGIN { print "<html>\n<head><title>Awk-generated HTML</title></head>\n<body bgcolor=\"#ffffff\">\n<pre
{ print $0 }
END { print "</pre>\n</body>\n</html>" }
```

Et les commandes à exécuter sont aussi bien plus directe quand on utilise **awk** plutôt que **sed** :

```
kelly@octarine ~/html> awk -f make-html-from-text.awk testfile > file.html
```

> **Exemples d'Awk sur votre système**
>
> Nous nous référons encore au répertoire qui contient les scripts d'initialisation de votre système. Entrez une commande similaire à la suivante pour voir d'autres exemples pratiques de l'usage très répandu de la commande **awk** :
>
> ```
> grep awk /etc/init.d/*
> ```

3.6. Le programme printf

Pour un contrôle du format du résultat plus précis que celui fournit par **print**, employez **printf**. La commande **printf** peut spécifier une largeur de champs pour chaque élément, de même que divers choix de formatage des nombres (tel que la base de calcul à considérer, si il faut afficher un exposant, si le signe est à afficher, et combien de chiffres à afficher après la virgule). Ceci est fait en ajoutant une chaîne appelée *chaîne de formatage*, qui contrôle comment et où afficher les autres arguments.

La syntaxe est la même que celle de l'instruction du langage C **printf** ; voir votre guide d'instruction C. Les pages info de **gawk** contiennent plein d'explications.

4. Résumé

L'utilitaire **gawk** interprète un langage de programmation à but spécial, prenant en charge les travaux de reformatage de données avec seulement quelques lignes de code. C'est une version libre de la commande UNIX **awk**.

Cet outil lit les lignes de données entrées et peut aisément repérer le colonnage du résultat. Le programme **print** est le plus courant pour filtrer et formater des champs définis.

La déclaration de variable au coup par coup est directe et permet des sommations simples, des statistiques et autres opérations sur le flot en entrée traité. Les variables et les commandes peuvent être insérées dans un script **awk** pour un traitement en tâche de fond.

Autres choses que vous devriez savoir à propos de **awk** :

- Le langage reste très répandu sur UNIX et ses clones, mais pour exécuter des tâches similaires Perl est maintenant plus souvent employé. Cependant, **awk** a une courbe d'apprentissage plus rapide (c'est à dire que vous apprenez en peu de temps). En d'autres mots, Perl est plus difficile à apprendre.
- Les deux, Perl et **awk**, se partagent la réputation d'être incompréhensible, même pour les auteurs de programmes dans ces langages. Donc annotez votre code !

5. Exercices

Il y a quelques exemples concret où **awk** peut être pratique.

1. Pour le premier exercice, l'entrée sont les lignes de la forme suivante :

   ```
   Username:Firstname:Lastname:Telephone number
   ```

   Faire un script **awk** qui convertit une telle ligne en un enregistrement LDAP à ce format :

   ```
   dn: uid=Username, dc=example, dc=com
   cn: Firstname Lastname
   sn: Lastname
   telephoneNumber: Telephone number
   ```

   Créer un fichier contenant quelques enregistrements de test et vérifiez..

2. Créer un script Bash utilisant **awk** et les commandes standard UNIX qui affiche les 3 utilisateurs les plus gros consommateurs d'espaces disques dans le répertoire `/home` (si il ne situe pas dans une partition distincte, faire le script pour la partition `/` ; celle-ci existe sur tout système UNIX). D'abord, exécutez les commandes depuis la ligne de commande. Puis mettez-les dans un script. Le script devrait produire un résultat compréhensible (lisible par le chef). Si tout semble fonctionner, faire en sorte que le script vous envoie le résultat par mail (employez par exemple **mail** `-s` ***Disk space usage*** <you@your_comp> < `result`).

   Si le démon des quotas tourne, utilisez ses informations ; si non utilisez **find**.

3. Créer un résultat de style XML à partir d'une liste séparée par des **tabulations** de la forme suivante :

```
Meaning very long line with a lot of description

meaning another long line

othermeaning    more longline

testmeaning     looooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooong line, but
```

Le résultat devrait être :

```
<row>
<entry>Meaning</entry>
<entry>
very long line
</entry>
</row>
<row>
<entry>meaning</entry>
<entry>
long line
</entry>
</row>
<row>
<entryothermeaning</entry>
<entry>
more longline
</entry>
</row>
<row>
<entrytestmeaning</entry>
<entry>
looooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooong line, but i mean really lo
</entry>
</row>
```

En plus, si vous connaissez quelque peu XML, écrivez un script BEGIN et END pour compléter la table. Ou faites le en HTML.

Chapitre 7. Les instructions de condition

Table des matières

1. Introduction de if
    1.1. Généralité
    1.2. Applications simples de if
2. L'emploi avancé de if
    2.1. les blocs if/then/else
    2.2. Les blocs if/then/elif/else
    2.3. Les instructions if imbriquées
    2.4. Opérations booléennes
    2.5. Emploi de l'instruction exit et du if
3. Utiliser les instructions case
    3.1. Les conditions simplifiées
    3.2. Exemple de script d'initialisation
4. Résumé
5. Exercices

Résumé

Dans ce chapitre nous traiterons de l'emploi de conditions dans les scripts Bash. Ceci comprend les sujets suivants :

- L'instruction **if**
- L'usage du statut d'exécution d'une commande
- Comparer et tester les entrées et des fichiers

- les blocs **if/then/else**
- Les blocs **if/then/elif/else**
- Utiliser et tester les paramètres positionnels
- Les instructions **if** imbriquées
- Les expressions booléennes
- Utiliser les instructions **case**

1. Introduction de if

1.1. Généralité

A certains moments vous pouvez vouloir donner une alternative au traitement effectué par le script, en fonction de l'échec ou la réussite d'une commande. Le bloc **if** permet de spécifier de telles conditions.

La syntaxe la plus compacte de la commande **if** est :

```
if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi
```

La liste **TEST-COMMAND** est exécutée, et si elle retourne le statut zéro, la liste **CONSEQUENT-COMMANDS** est exécutée. Le statut retourné est le statut d'exécution de la dernière commande exécutée, ou zéro si aucune condition n'est vraie.

Le **TEST-COMMAND** souvent comprend des comparaisons de numériques ou de chaînes, mais cela peut être aussi toute commande qui retourne un statut à zéro quand elle s'est bien exécutée et d'autres valeurs en cas d'échec. Une expression unaire est souvent utilisée pour examiner le statut d'un fichier. Si l'argument `FILE` d'une de ces primitives est de la forme `/dev/fd/N`, alors le descripteur de fichier « N » est contrôlé. `stdin`, `stdout` et `stderr` et leur descripteur de fichier respectif peuvent aussi être employés dans les tests.

1.1.1. Expressions employées avec if

Le tableau ci-dessous contient un aperçu de ce qu'on appelle « primitives » et qui servent aux commandes **TEST-COMMAND** ou liste de commandes. Ces primitives sont mises entre crochets pour indiquer le test d'une expression conditionnelle..

Tableau 7.1. Expressions primitives

Primitives	**sens**
[`-a FICHIER`]	Vrai si `FICHIER` existe.
[`-b FICHIER`]	Vrai si `FICHIER` existe et est un fichier de type bloc.
[`-c FICHIER`]	Vrai si `FICHIER` existe et est un fichier de type caractère.
[`-d FICHIER`]	Vrai si `FICHIER` existe et est de type répertoire.
[`-e FICHIER`]	Vrai si `FICHIER` existe.
[`-f FICHIER`]	Vrai si `FICHIER` existe et est un fichier régulier.
[`-g FICHIER`]	Vrai si `FICHIER` existe et son bit SGID est positionné.
[`-h FICHIER`]	Vrai si `FICHIER` existe et est un lien symbolique.

Primitives	**sens**
[-k FICHIER]	Vrai si FICHIER existe et son bit collant est positionné.
[-p FICHIER]	Vrai si FICHIER existe et est un tube nommé (FIFO).
[-r FICHIER]	Vrai si FICHIER existe et est lisible.
[-s FICHIER]	Vrai si FICHIER existe et a une taille supérieure à zéro.
[-t FD]	Vrai si le descripteur de fichier FD est ouvert et qu'il se réfère à un terminal.
[-u FICHIER]	Vrai si FILE existe et son bit SUID (Set User ID) est positionné.
[-w FICHIER]	Vrai si FICHIER existe et est en écriture.
[-x FICHIER]	Vrai si FICHIER existe et est exécutable.
[-O FICHIER]	Vrai si FICHIER existe et appartient à l'identifiant effectif de l'utilisateur.
[-G FICHIER]	Vrai si FICHIER existe et appartient à l'identifiant effectif du groupe.
[-L FICHIER]	Vrai si FICHIER existe et est un lien symbolique.
[-N FICHIER]	Vrai si FICHIER existe et qu'il a été modifié depuis qu'il a été lu.
[-S FICHIER]	Vrai si FICHIER existe et est un connecteur réseau (socket).
[FILE1 -nt FILE2]	Vrai si FILE1 a été modifié plus récemment que FILE2, ou si FILE1 existe et FILE2 n'existe pas.
[FILE1 -ot FILE2]	Vrai si FILE1 est plus ancien que FILE2, ou si FILE2 existe et FILE1 non.
[FILE1 -ef FILE2]	Vrai si FILE1 et FILE2 se réfère à la même entité et même numéro d'inode.
[-o OPTIONNAME]	Vrai si l'option Shell « OPTIONNAME » est activée.
[-z STRING]	Vrai si la longueur de « STRING » est zéro.
[-n STRING] or [STRING]	Vrai si la longueur de « STRING » n'est pas zéro.
[STRING1 == STRING2]	Vrai si les chaînes sont identiques. « = » peut être employé au lieu de « == » pour une stricte compatibilité POSIX.
[STRING1 != STRING2]	Vrai si les chaînes ne sont pas égales.
[STRING1 < STRING2]	Vrai si « STRING1 » précède « STRING2 » selon le lexique du paramétrage local.
[STRING1 > STRING2]	Vrai si « STRING1 » suit « STRING2 » selon le lexique du paramétrage local.
[ARG1 OP ARG2]	« OP » est -eq, -ne, -lt, -le, -gt ou l'option -ge. Ces opérateurs arithmétiques binaires renvoient vrai si « ARG1 » est égal, non égal, inférieur, supérieur ou égal, supérieur, ou supérieur ou égal à « ARG2 », respectivement. « ARG1 » et « ARG2 » sont des entiers.

Les expressions peuvent être combinées avec les opérateurs suivants dans l'ordre de leur préséance :

Tableau 7.2. Combinaison d'expressions

Opération	Effet
[! EXPR]	Vrai si **EXPR** est faux.
[(EXPR)]	Renvoie la valeur de **EXPR**. Ceci peut être utilisé pour modifier la préséance normale des opérateurs.
[EXPR1 -a EXPR2]	Vrai si **EXPR1** et **EXPR2** sont vrai..
[EXPR1 -o EXPR2]	Vrai si soit **EXPR1** ou **EXPR2** est vrai.

L'intégrée **[** (ou **test**) évalue les expressions conditionnelles en utilisant un jeux de règles basé sur le nombre d'arguments. Plus d'information sur ce sujet se trouve dans la documentation Bash. Tout comme le **if** est terminé par **fi**, le crochet ouvrant devrait être fermé après que les conditions aient été listées.

1.1.2. Les commandes qui suivent l'instruction then

La liste **CONSEQUENT-COMMANDS** qui suit l'instruction **then** peut être toute commande UNIX valide , tout programme exécutable, tout script Shell exécutable ou toute instruction Shell, à l'exception de **fi**. Il est important de se rappeler que **then** et **fi** sont considérés comme des instructions à part entière dans le Shell. De ce fait, quand elles sont fournies à la ligne de commande, elles sont séparées par un point-virgule.

Dans un script, les différentes parties de l'instruction **if** sont d'ordinaire bien séparées. Ci-dessous, quelques exemples.

1.1.3. Contrôler des fichiers

Le premier exemple contrôle l'existence d'un fichier :

```
anny ~> cat msgcheck.sh
#!/bin/bash

echo "Ce script vérifie si le fichier des messages existe."
echo "Vérification..."
if [ -f /var/log/messages ]
  then
    echo "/var/log/messages existe."
fi
echo
echo "...fait."

anny ~> ./msgcheck.sh
Ce script vérifie si le fichier des messages existe.
Vérification...
/var/log/messages existe.

...fait.
```

1.1.4. Vérifier des options Shell

A ajouter dans vos fichier de configuration Bash :

```
# Ces lignes affichent un message si l'option noclobber option est positionnée :

if [ -o noclobber ]
  then
       echo "Vos fichiers sont protégés contre une réécriture accidentelle du fait d'une redirection."
fi
```

L'environnement

L'exemple ci-dessus fonctionnera si il est soumis à la ligne de commande :

```
anny ~> if [ -o noclobber ] ; then echo ; echo "Vos fichiers sont protégés contre une r
Vos fichiers sont protégés contre une réécriture.
anny ~>
```

Cependant, si vous employez les tests de conditions qui dépendent de l'environnement, vous résultats variables alors que vous exécutez la même commande dans un script, parce que le sc Shell, dans lequel les variables et options attendues pourraient ne pas être définies automatic

1.2. Applications simples de if

1.2.1. Tester le statut d'exécution

La variable ? stocke le statut d'exécution de la commande précédemment exécutée (le processus le plus récemment achevé au premier plan).

L'exemple suivant montre un simple test :

```
anny ~> if [ $? -eq 0 ]
More input> then echo 'That was a good job!'
More input> fi
That was a good job!

anny ~>
```

L'exemple suivant démontre que **TEST-COMMANDS** pourrait être toute commande UNIX qui retourne un statut, et le **if** à son tour renvoie un statut à zéro :

```
anny ~> if ! grep $USER /etc/passwd
More input> then echo "votre compte utilisateur ne se trouve pas sur le système local"; fi
votre compte utilisateur ne se trouve pas sur le système local

anny > echo $?
0

anny >
```

Le même résultat peut être obtenu comme ceci :

```
anny > grep $USER /etc/passwd

anny > if [ $? -ne 0 ] ; then echo "pas un compte local" ; fi
pas un compte local

anny >
```

1.2.2. Comparaisons numériques

Les exemples ci-dessous emploient des comparaisons numériques :

```
anny > num=`wc -l work.txt`

anny > echo $num
201

anny > if [ "$num" -gt "150" ]
More input> then echo ; echo "vous avez assez travaillé pour aujourd'hui."
More input> echo ; fi

vous avez assez travaillé pour aujourd'hui.


anny >
```

Ce script est exécuté par cron chaque dimanche. Si le numéro de semaine est pair, cela vous rappelle

le passage des éboueurs :

```
#!/bin/bash

# Calculer le numéro de semaine à partir de la commande date :

WEEKOFFSET=$[ $(date +"%V") % 2 ]

# Tester si il y a un reste.  Si pas de reste, c'est une semaine paire donc envoyer un message.
# Sinon, ne rien faire.

if [ $WEEKOFFSET -eq "0" ]; then
  echo "Dimanche soir, les éboueurs passent." | mail -s "Les éboueurs passent" your@your_domain.org
fi
```

1.2.3. Comparaisons de chaînes

Un exemple de comparaison de chaînes avec le test de l'identifiant utilisateur :

```
if [ "$(whoami)" != 'root' ]; then
        echo "Vous ne détenez pas la permission de lancer $0 en tant que non administrateur."
        exit 1;
fi
```

Avec Bash, vous pouvez raccourcir ce type de construction. L'équivalent compact du test ci-dessus est ce qui suit :

```
[ "$(whoami)" != 'root' ] && ( echo vous êtes connectés avec un compte non administrateur; exit 1 )
```

Similaire à l'expression « && » qui indique quoi faire si le test s'avère vrai, « || » spécifie quoi faire si le test est faux.

Une expression régulière peut être employée aussi dans les comparaisons :

```
anny > genre="féminin"

anny > if [[ "$genre" == f* ]]
More input> then echo "Très honoré, Madame."; fi
Très honoré, Madame.

anny >
```

Les vrais Programmeurs

La plupart des programmeurs préféreront employer l'intégrée **test** qui est équivalent à l'emp de comparaison, comme ceci :

```
test "$(whoami)" != 'root' && (echo vous êtes connectés avec un compte non administrate
```

No exit?

Si vous appelez **exit** dans un sous-Shell, celui-ci ne passera pas de variables à son parent. Employez { and } au lieu de (and) si vous ne voulez pas que Bash en fourchant crée un sous-Shell.

Voir les pages info de Bash pour plus de détails sur la correspondance de patron avec les blocs « ((EXPRESSION)) » et « [[EXPRESSION]] ».

2. L'emploi avancé de if

2.1. les blocs if/then/else

2.1.1. Exemple simple

Voici la construction à employer pour que le cours du traitement s'oriente d'une façon si la commande **if** renvoie vrai, et d'une autre si elle renvoie faux. Un exemple :

```
freddy scripts> genre="masculin"

freddy scripts> if [[ "$genre" == "f*" ]]
More input> then echo "Très honoré, Madame."
More input> else echo "Comment se fait-il que le verre de Madame  soit vide ?"
More input> fi
Comment se fait-il que le verre de Madame  soit vide ?

freddy scripts>
```

> **[] versus [[]]**
>
> Contrairement à `[`, `[[` empêche le découpage en mots des valeurs de variables. Donc, si `VAR="var with spaces"`, vous n'avez pas besoin de mettre des guillemets à `$VAR` dans un test - même si cela reste une bonne habitude. Aussi, `[[` inhibe l'analyse des chemins, de sorte que des chaînes littérales avec des jokers ne seront pas interprétées comme des noms de fichiers. Les `[[`, == et `!=` font interpréter la chaîne à la droite comme un patron glob du Shell qui doit correspondre à la valeur à la gauche, par exemple : `[[ "value" == val* ]]`.

De même que la liste **CONSEQUENT-COMMANDS** suivant le **then**, la liste **ALTERNATE-CONSEQUENT-COMMANDS** suivant le **else** peut contenir toute commande de style UNIX qui retourne un statut d'exécution.

Un autre exemple, tiré de celui de la Section 1.2.1, « Tester le statut d'exécution » :

```
anny ~> su -
Password:
[root@elegance root]# if ! grep ^$USER /etc/passwd 1> /dev/null
> then echo "votre compte utilisateur ne se trouve pas sur le système local"
> else echo "votre compte utilisateur ne se trouve pas sur le système local"
> fi
votre compte utilisateur se trouve dans /etc/passwd file
[root@elegance root]#
```

Nous permutons vers le compte *root* pour montrer l'effet du **else** - votre *root* est d'ordinaire un compte local tandis que votre compte personnel peut être géré par un système central, tel qu'un serveur LDAP.

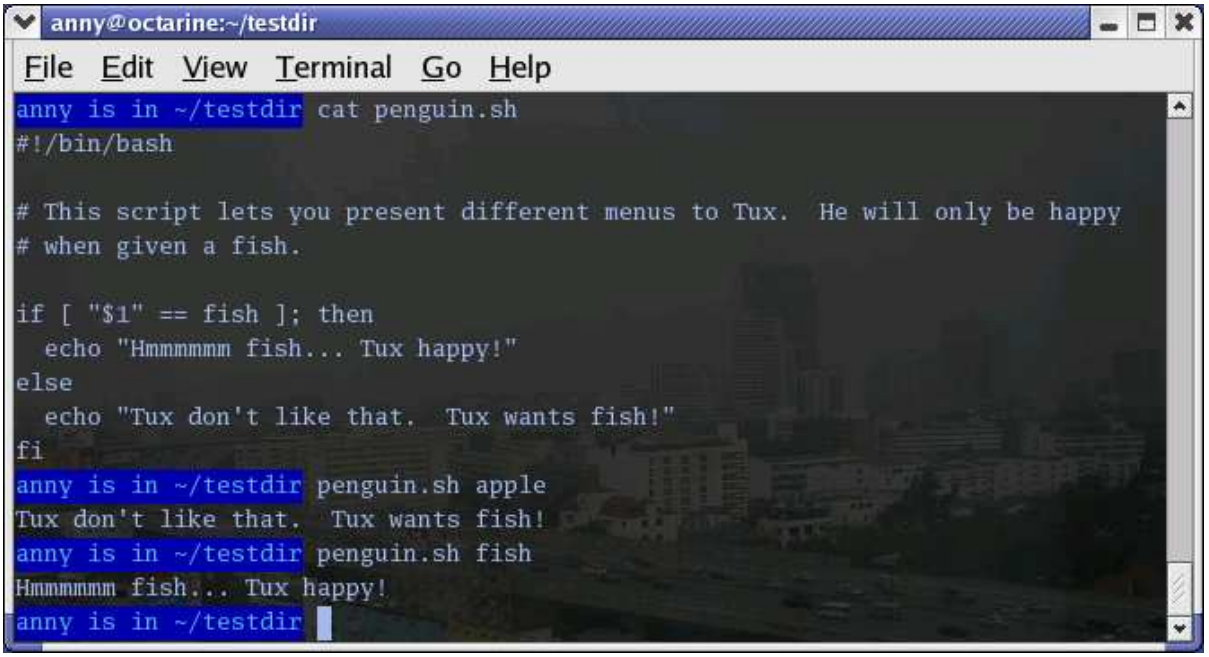
2.1.2. Contrôle des paramètres de la ligne de commande

Au lieu de déclarer une variable puis d'exécuter un script, il est fréquemment plus élégant de mettre la valeur de la variable dans la ligne de commande.

Pour ce faire nous employons les paramètres positionnels `$1`, `$2`, ..., `$N`. `$#` mémorise le nombre de paramètres de la ligne de commande. `$0` mémorise le nom du script.

Voici un exemple simple :

Figure 7.1. Test d'une ligne de commande avec if



Voici un autre exemple avec 2 paramètres :

```
anny ~> cat weight.sh
#!/bin/bash

# Ce script affiche un message au sujet de votre poids si vous donnez
# votre poids en kilos et votre taille en centimètres.

weight="$1"
height="$2"
idealweight=$[$height - 110]

if [ $weight -le $idealweight ] ; then
  echo "Vous devriez manger un peu plus gras."
else
  echo "Vous devriez manger un peu plus de fruits."
fi

anny ~> bash -x weight.sh 55 169
+ weight=55
+ height=169
+ idealweight=59
+ '[' 55 -le 59 ']'
+ echo 'Vous devriez manger un peu plus gras.'
Vous devriez manger un peu plus gras.
```

2.1.3. Tester le nombre de paramètres

L'exemple suivant montre comment changer le script précédent de sorte qu'il affiche un message si plus ou moins de 2 paramètres sont donnés :

```
anny ~> cat weight.sh
#!/bin/bash

# Ce script affiche un message au sujet de votre poids si vous donnez
# votre poids en kilos et votre taille en centimètres.

if [ ! $# == 2 ]; then
  echo "Usage: $0 poids_en_kilos taille_en_centimètres"
  exit
fi

weight="$1"
height="$2"
idealweight=$[$height - 110]

if [ $weight -le $idealweight ] ; then
  echo "Vous devriez manger un peu plus gras."
else
  echo "Vous devriez manger un peu plus de fruits."
fi

anny ~> weight.sh 70 150
Vous devriez manger un peu plus de fruits.

anny ~> weight.sh 70 150 33
Usage: ./weight.sh poids_en_kilos taille_en_centimètres
```

Le premier paramètre est référencé par `$1`, le second par `$2` et ainsi de suite. Le nombre total de paramètres est stocké dans `$#`.

Consulter la Section 2.5, « Emploi de l'instruction exit et du if » pour voir une façon plus élégante d'afficher des messages de mode d'emploi.

2.1.4. Test de l'existence d'un fichier

Ce test est fait dans beaucoup de scripts, parce que il n'y a pas d'intérêt à lancer un programme si vous savez qu'il ne va pas fonctionner :

```
#!/bin/bash

# Ce script donne des informations au sujet d'un fichier.

FILENAME="$1"

echo "Properties for $FILENAME:"

if [ -f $FILENAME ]; then
  echo "Size is $(ls -lh $FILENAME | awk '{ print $5 }')"
  echo "Type is $(file $FILENAME | cut -d":" -f2 -)"
  echo "Inode number is $(ls -i $FILENAME | cut -d" " -f1 -)"
  echo "$(df -h $FILENAME | grep -v Mounted | awk '{ print "On",$1", \
which is mounted as the",$6,"partition."}')"
else
  echo "Le fichier est non-existant."
fi
```

Notez que le fichier est référencé au moyen d'une variable ; dans ce cas c'est le premier paramètre du script. Alternativement, quand aucun paramètre n'est donné, l'emplacement du fichier est mémorisé généralement dans une variable au début du script, et son contenu est connu par l'invocation de la variable. De sorte que si vous voulez changer le nom d'un fichier dans un script, vous n'avez que à le modifier une fois.

> **Nom de fichier avec des espaces**
>
> L'exemple plus haut échouera si la valeur de `$1` peut être découpée en plusieurs mots. Dans ce cas, la commande **if** peut être figée soit en mettant des guillemets autour du nom de fichier, soit en employant *[[* au lieu de *[*.

2.2. Les blocs if/then/elif/else

2.2.1. Généralité

C'est la forme complète de l'instruction **if** :

```
if TEST-COMMANDS; then

CONSEQUENT-COMMANDS;

elif MORE-TEST-COMMANDS; then

MORE-CONSEQUENT-COMMANDS;

else ALTERNATE-CONSEQUENT-COMMANDS;

fi
```

La liste **TEST-COMMANDS** est exécutée, et si son statut d'exécution est zéro, la liste **CONSEQUENT-COMMANDS** est exécutée. Si **TEST-COMMANDS** renvoie un statut différent de zéro, chaque liste **elif** est exécutée à son tour, et si leur statut d'exécution est zéro, le **MORE-CONSEQUENT-COMMANDS** correspondant est exécuté et la commande se termine. Si

else est suivi par une liste **ALTERNATE-CONSEQUENT-COMMANDS**, et que la dernière commande dans le dernier **if** ou **elif** renvoie un statut différent de zéro, alors **ALTERNATE-CONSEQUENT-COMMANDS** est exécuté. Le statut retourné est le statut d'exécution de la dernière commande exécutée, ou zéro si aucune condition n'est vraie.

2.2.2. Exemple

Ceci est un exemple que vous pouvez mettre dans votre crontab pour une exécution quotidienne :

```
anny /etc/cron.daily> cat disktest.sh
#!/bin/bash

# Ce script fait un test très simple pour contrôler l'espace disque.

space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`
alertvalue="80"

if [ "$space" -ge "$alertvalue" ]; then
  echo "Au moins un de mes disques est bientôt plein !" | mail -s "daily diskcheck" root
else
  echo "Espace disque correct" | mail -s "daily diskcheck" root
fi
```

2.3. Les instructions if imbriquées

Comprise dans une instruction **if** on peut inclure une autre instruction **if**. Vous pouvez inclure autant de niveaux de **if** imbriqués que vous pouvez appréhender logiquement.

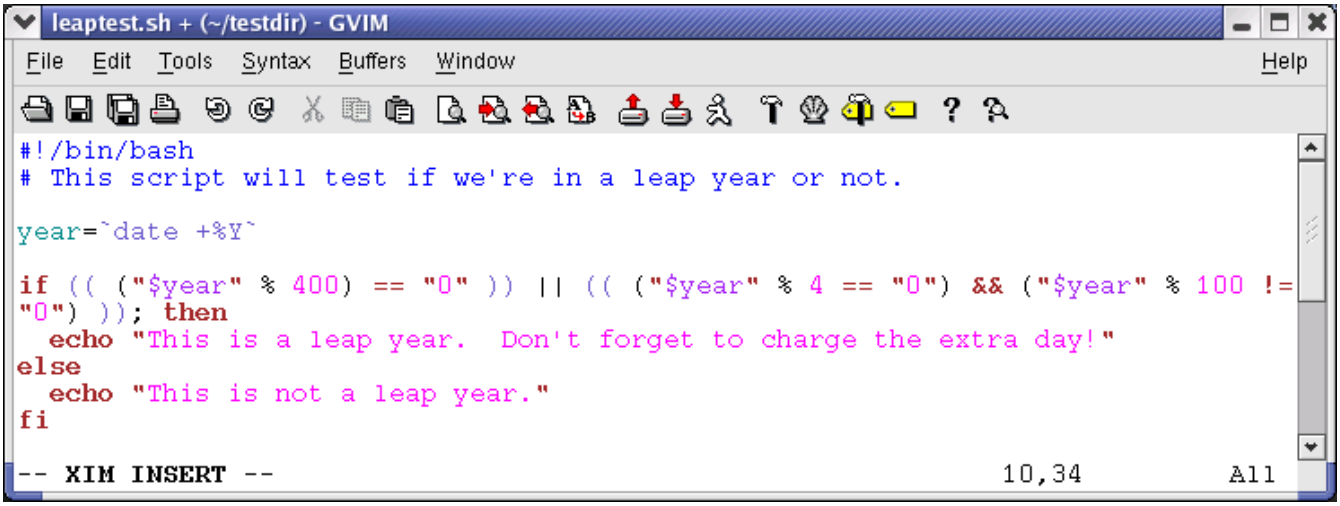
Voici un exemple testant l'année bissextile :

```
anny ~/testdir> cat testleap.sh
#!/bin/bash
# Ce script teste si nous sommes dans une année bissextile ou pas.

year=`date +%Y`

if [ $[$year % 400] -eq "0" ]; then
  echo "This is a leap year.  Février a 29 jours."
elif [ $[$year % 4] -eq 0 ]; then
        if [ $[$year % 100] -ne 0 ]; then
          echo "Année bissextile, Février a 29 jours."
        else
          echo "Année non bissextile.  Février a 28 jours."
        fi
else
  echo "Année non bissextile.  Février a 28 jours."
fi

anny ~/testdir> date
Tue Jan 14 20:37:55 CET 2003

anny ~/testdir> testleap.sh
Année non bissextile.
```

2.4. Opérations booléennes

Le script ci-dessus peut être abrégé avec les opérateurs booléens « AND » (&&) et « OR » (||).

Figure 7.2. Exemple employant les opérateurs booléens



Nous employons le double crochet pour tester les expressions arithmétiques, voir la Section 4.6, « L'expansion arithmétique ». Ceci est équivalent à l'instruction **let**. Ici, vous allez être bloqué si vous employez les crochets, si vous essayez quelque chose de la sorte **$[$year % 400]**, parce que ici, les crochets ne représentent pas une vraie commande mais eux-mêmes.

Parmi d'autres éditeurs, **gvim** est l'un de ceux qui supporte les codes de couleur selon le format de fichier ; de tel éditeurs sont pratiques pour pister les erreurs d'écriture.

2.5. Emploi de l'instruction exit et du if

Nous avons déjà rencontré l'instruction **exit** dans la Section 2.1.3, « Tester le nombre de paramètres ». Il achève l'exécution du script. Il est plus souvent utilisé si l'entrée requise de l'utilisateur est incorrecte, si une instruction a échouée ou si une autre erreur intervient.

L'instruction **exit** admet un argument optionnel. Cet argument est le code sous forme d'entier du statut d'exécution, qui est renvoyé au parent et stocké dans la variable `$?`.

Un argument à zéro signifie que le script s'est exécuté correctement. Tout autre valeur peut être employée par le programmeur pour renvoyer divers messages au parent, afin que divers traitements soient activés selon l'échec ou la réussite du processus enfant. Si aucun argument n'est donné à la commande **exit**, le Shell parent exploite la valeur courante de la variable `$?`.

Ci-dessous un exemple avec un script `penguin.sh` légèrement adapté, lequel renvoie son statut d'exécution vers son parent, `feed.sh` :

```
anny ~/testdir> cat penguin.sh
#!/bin/bash

# Ce script vous laisse présenter divers menus à Tux.  Il ne sera heureux que
# quand il aura du poisson.  Nous avons aussi ajouté un dauphin et (logiquement) un chameau.

if [ "$menu" == "poisson" ]; then
  if [ "$animal" == "pingouin" ]; then
    echo "Hmmmmmm poisson... Tux heureux !"
  elif [ "$animal" == "dauphin" ]; then
    echo "Pweetpeettreetppeterdepweet !"
  else
    echo "*prrrrrrrt*"
  fi
else
  if [ "$animal" == "pingouin" ]; then
    echo "Tux déteste ça.  Tux veut du poisson !"
    exit 1
  elif [ "$animal" == "dauphin" ]; then
    echo "Pweepwishpeeterdepweet !"
    exit 2
  else
    echo "Voulez-vous lire cette affiche ?!"
    exit 3
  fi
fi
```

Ce script est appelé depuis le suivant, qui donc exporte ses variables `menu` et `animal` :

```
anny ~/testdir> cat feed.sh
#!/bin/bash
# Ce script procède selon le statut d'exécution renvoyé par penguin.sh

export menu="$1"
export animal="$2"

feed="/nethome/anny/testdir/penguin.sh"

$feed $menu $animal

case $? in

1)
  echo "Gaffe : Vous feriez mieux de lui donner du poisson, avant qu'il ne s'énerve..."
  ;;
2)
  echo "Gaffe : C'est à cause de gens comme vous qu'il quitte la terre tout le temps..."
  ;;
3)
  echo "Gaffe : Achetez la nourriture que le zoo fournit pour les animaux, i@**@, comment pensez-vous c
  ;;
*)
  echo "Gaffe : N'oubliez pas le guide !"
  ;;
esac

anny ~/testdir> ./feed.sh apple penguin
Tux déteste ça.  Tux veut du poisson !
Gaffe : Vous feriez mieux de lui donner du poisson, avant qu'il ne s'énerve...
```

Comme vous le voyez, le statut d'exécution peut être déterminé librement. Les commandes ont souvent une série de codes définis ; voir le manuel du programmeur pour plus d'informations sur chaque commande.

3. Utiliser les instructions case

3.1. Les conditions simplifiées

Les instructions **if** imbriquées paraissent pratiques, mais dès que vous êtes confrontés à quelques variantes, cela engendre la confusion. Pour des conditions complexes, employez la syntaxe de **case** :

```
case EXPRESSION in CASE1) COMMAND-LIST;; CASE2) COMMAND-LIST;; ... CASEN) COMMAND-LIST;; esac
```

Chaque cas est une expression qui cible un patron. Les commandes **COMMAND-LIST** de la première correspondance trouvée sont exécutées. Le symbole « | » est employé pour séparer de multiples patrons, et l'opérateur «) » termine la liste des patrons. Chaque case et ses commandes associées est appelé une *clause*. Chaque clause doit se terminer par « ;; ». Chaque instruction **case** est terminée par l'instruction **esac**.

Dans l'exemple, nous montrons l'emploi de case pour envoyer un message d'avertissement plus précis avec le script `disktest.sh` :

```
anny ~/testdir> cat disktest.sh
#!/bin/bash

# Ce script fait un test très simple pour vérifier l'espace disque.

space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 | cut -d "%" -f1 -`

case $space in
[1-6]*)
  Message="Tout est bon."
  ;;
[7-8]*)
  Message="Commencer à songer à faire de la place.  Il y a une partition qui est $space % pleine."
  ;;
9[1-8])
  Message="Dépêchez-vous avec ce nouveau disque..  Une partition est $space % pleine."
  ;;
99)
  Message="Je suis en train de me noyer !  Il y a une partition à $space % !"
  ;;
*)
  Message="Il semble que je tourne avec un espace disque inexistant..."
  ;;
esac

echo $Message | mail -s "disk report `date`" anny

anny ~/testdir>
```

```
Vous avez un nouveau mail.

anny ~/testdir> tail -16 /var/spool/mail/anny
From anny@octarine Tue Jan 14 22:10:47 2003
Return-Path: <anny@octarine>
Received: from octarine (localhost [127.0.0.1])
        by octarine (8.12.5/8.12.5) with ESMTP id h0ELAlBG020414
        for <anny@octarine>; Tue, 14 Jan 2003 22:10:47 +0100
Received: (from anny@localhost)
        by octarine (8.12.5/8.12.5/Submit) id h0ELAltn020413
        for anny; Tue, 14 Jan 2003 22:10:47 +0100
Date: Tue, 14 Jan 2003 22:10:47 +0100
From: Anny <anny@octarine>
Message-Id: <200301142110.h0ELAltn020413@octarine>
To: anny@octarine
Subject: disk report Tue Jan 14 22:10:47 CET 2003

Commencer à songer à faire de la place.  Il y a une partition qui est 87 % pleine.

anny ~/testdir>
```

Bien sûr vous pourriez avoir ouvert votre programme de messagerie pour contrôler le résultat ; c'est juste pour montrer que le script envoie un mail correct avec « To: », « Subject: » and « From: » header lines.

Beaucoup plus d'exemples de l'instruction **case** peuvent être trouvés dans le répertoire des scripts d'initialisation de votre système. Le script de démarrage emploie un case **start** et **stop** pour démarrer ou arrêter les processus du système. Un exemple théorique peut être trouvé dans la section suivante.

3.2. Exemple de script d'initialisation

Les scripts d'initialisation ont souvent l'usage d'instructions **case** pour démarrer, arrêter et mettre en file d'attente les services du système. Voici un extrait du script qui démarre Anacron, un démon qui lance des commandes périodiquement avec une fréquence spécifiée en jours.

```
case "$1" in
        start)
            start
            ;;

        stop)
            stop
            ;;

        status)
            status anacron
            ;;
        restart)
            stop
            start
            ;;
        condrestart)
            if test "x`pidof anacron`" != x; then
                stop
                start
            fi
            ;;

        *)
            echo $"Usage: $0 {start|stop|restart|condrestart|status}"
            exit 1

esac
```

Les tâches à exécuter dans chaque cas, telles arrêter ou démarrer le démon, sont définies par des fonctions, dont la source est partiellement dans le fichier `/etc/rc.d/init.d/functions`. Voir le Chapitre 11, *Fonctions* pour plus d'explications.

4. Résumé

Dans ce chapitre nous avons appris comment écrire des conditions dans un script de sorte que différentes tâches puissent être menées à bien selon le succès ou l'échec d'une commande. L'action peut être déterminée par l'emploi de l'instruction **if**. Ceci permet d'effectuer des comparaisons de chaînes et arithmétique, et de tester le statut d'exécution, l'entrée et les fichiers requis par le script.

Un simple test **if/then/fi** souvent précède des commandes dans un script Shell afin d'éviter la production de résultat, de sorte que le script peut aisément être lancé en tâche de fond ou via l'outil

cron. Les conditions trop complexes sont généralement intégrées à une instruction **case**.

A la suite d'un test positif, le script peut explicitement informer le parent par le biais du statut **exit** 0. A la suite d'un échec, tout autre nombre peut être retourné. Basé sur ce code retour, le programme parent peut déterminer l'action appropriée.

5. Exercices

Voici quelques idées pour vous lancer dans l'écriture de scripts **if** :

1. Employez un bloc **if/then/elif/else** qui affiche les informations du mois courant. Le script devrait afficher le nombre de jours du mois, et donner des informations sur l'année bissextile si le mois courant est février.

2. Faire la même chose avec une instruction **case** et une variante de l'usage de la commande **date**.

3. Modifier `/etc/profile` afin d'être accueilli par un message personnalisé quand vous vous connectez au système en tant que *root*.

4. Modifier le script `leaptest.sh` dans Section 2.4, « Opérations booléennes » afin qu'il nécessite un paramètre, l'année. Tester que exactement un seul paramètre est passé.

5. Ecrire un script appelé `whichdaemon.sh` qui vérifie que les démons **httpd** et **init** sont lancés sur votre système. Si un **httpd** est lancé, le script devrait afficher un message comme « Cette machine fait tourner un serveur WEB. » Employez **ps** pour contrôler les processus.

6. Écrire un script qui fait la sauvegarde de votre répertoire racine sur une machine distante en utilisant **scp**. Le script devrait écrire son rapport dans un fichier journal, par exemple `~/log/homebackup.log`. Si vous n'avez pas de seconde machine pour y copier la sauvegarde, se servir de **scp** pour tester la copie sur la machine locale. Ceci nécessite des clés SSH entre 2 hôtes, ou sinon vous devez fournir un mot de passe. La création de clés SSH est expliquée dans le **man** ***ssh-keygen***.

7. Adaptez le script à partir du 1er exemple à la Section 3.1, « Les conditions simplifiées » pour inclure le cas de l'occupation de exactement 90% de l'espace disque, et de moins de 10% de l'espace disque.

   Le script devrait se servir de **tar** `cf` pour la création de la sauvegarde et **gzip** ou **bzip2** pour la décompression du fichier `.tar`. Mettre tous les noms de fichiers dans des variables. Mettre le nom du serveur distant et du répertoire distant dans une variable. Ce sera plus facile de réutiliser ce script ou d'y faire des modifications dans le futur.

   Le script devrait vérifier l'existence d'une archive compressée. Si elle existe, la supprimer d'abord pour éviter les messages d'erreur.

   Le script devrait aussi contrôler l'espace disque disponible. Avoir à l'esprit qu'à un moment donné vous pourrez avoir en même temps sur le disque les données dans votre répertoire racine, celles dans le fichier `.tar` et celle dans l'archive compressée. Si il n'y a pas assez d'espace, quitter avec un message d'erreur dans le fichier journal.

   Le script devrait nettoyer l'archive compressée avant de se terminer.

Chapitre 8. Ecrire des scripts interactifs

Table des matières

1. Afficher les messages utilisateurs
   1.1. Interactif ou pas ?
   1.2. Utiliser la commande intégrée echo
2. Récupérer la saisie utilisateur
   2.1. L'emploi de la commande intégrée read

2.2. Demander une entrée utilisateur
2.3. Redirection et descripteurs de fichiers
2.4. Fichier d'entrée et fichier de sortie
3. Résumé
4. Exercices

Résumé

Dans ce chapitre nous expliquerons comment interagir avec les utilisateurs de nos scripts :

- En affichant des messages et des explications conviviaux à l'intention des utilisateurs
- Récupérer la saisie utilisateur
- Demander une entrée utilisateur
- Utiliser les descripteurs de fichiers pour lire et écrire depuis et sur de multiples fichiers

1. Afficher les messages utilisateurs

1.1. Interactif ou pas ?

Certains scripts s'exécutent sans aucune interaction avec l'utilisateur. Les avantages des scripts non interactifs comprennent :

- Le script s'exécute d'une façon prédictible chaque fois.
- Le script peut s'exécuter en tâche de fond.

Beaucoup de scripts, cependant, demandent des entrées de la part de l'utilisateur, ou donnent des résultats à l'utilisateur alors que le script tourne. Les avantages des scripts interactifs sont, parmi d'autres :

- Scripts plus flexibles
- L'utilisateur peut adapter le script pendant qu'il est lancé ou plutôt faire en sorte qu'il agisse de diverses façons.
- Le script peut afficher la progression du traitement.

Quand vous écrivez un script interactif, ne jamais être avare de commentaires. Un script qui affiche des messages appropriés est bien plus convivial et peut être plus facilement débuggé. Un script peut effectuer un traitement tout à fait correct, mais vous aurez bien plus d'appels d'utilisateurs si il ne les informe pas de l'état d'avancement. Donc inclure des messages qui demandent à l'utilisateur d'attendre le résultat parce que le traitement est en cours. Si possible, essayer d'indiquer combien de temps l'utilisateur aura à attendre. Si l'attente doit régulièrement durer un bon moment à l'exécution de certaines tâches, vous pouvez considérer l'intérêt d'intégrer la situation du processus dans le résultat du script.

Quand vous sollicitez de l'utilisateur une saisie, le mieux est aussi de donner plutôt trop que pas assez d'informations au sujet du type de données attendues. Ceci s'applique au contrôle des paramètres et au message mode d'emploi l'accompagnant.

Bash a les commandes **echo** et **printf** pour fournir des commentaires aux utilisateurs, et bien que vous devriez être familiarisé maintenant au moins avec **echo**, nous verrons d'autres exemples dans les sections suivantes.

1.2. Utiliser la commande intégrée echo

La commande intégrée **echo** affiche ses arguments, séparés par des espaces, et termine par un saut de ligne. Le statut renvoyé est toujours zéro. **echo** a plusieurs options :

- `-e` : interprète les caractères protégés.
- `-n` : supprime les sauts de ligne résiduels de la fin.

Comme exemple d'ajout de commentaires, nous écrirons le `feed.sh` et le `penguin.sh` de la Section 2.1.2, « Contrôle des paramètres de la ligne de commande » de façon améliorée :

```
michel ~/test> cat penguin.sh
#!/bin/bash

# Ce script vous laisse présenter divers menus à Tux.  Il ne sera heureux que
# quand il aura du poisson.  Pour s'amuser un peu, nous ajoutons quelques animaux.

if [ "$menu" == "poisson" ]; then
  if [ "$animal" == "pingouin" ]; then
    echo -e "Hmmmmmm poisson... Tux heureux !\n"
  elif [ "$animal" == "dauphin" ]; then
    echo -e "\a\a\aPweetpeettreetppeterdepweet !\a\a\a\n"
  else
    echo -e "*prrrrrrrt*\n"
  fi
else
  if [ "$animal" == "pingouin" ]; then
    echo -e "Tux déteste ça.  Tux veut du poisson !\n"
    exit 1
  elif [ "$animal" == "dauphin" ]; then
    echo -e "\a\a\a\a\a\aPweepwishpeeterdepweet !\a\a\a"
    exit 2
  else
    echo -e "Voulez-vous lire cette affiche ?!  Ne pas nourrir les "$animal"s !\n"
    exit 3
  fi
fi

michel ~/test> cat feed.sh
#!/bin/bash
# Ce script agit en fonction du statut d'exécution renvoyé par penguin.sh

if [ "$#" != "2" ]; then
  echo -e "Utilisation du script feed:\t$0 nourriture nom-animal \n"
  exit 1
else

  export menu="$1"
  export animal="$2"

  echo -e "Nourrissage $menu to $animal...\n"

  feed="/nethome/anny/testdir/penguin.sh"

  $feed $menu $animal

result="$?"

  echo -e "Nourrissage fait.\n"

case "$result" in

  1)
    echo -e "Gaffe : \"Vous feriez mieux de lui donner du poisson, Sinon il s'énerve...\"\n"
    ;;
  2)
    echo -e "Gaffe : \"Pas étonnant qu'il fuit notre planète...\"\n"
    ;;
  3)
    echo -e "Gaffe : \"Achetez la nourriture fournie par le zoo à l'entrée, i***\"\n"
    echo -e "Gaffe : \"Vous voulez les empoisonner ?\"\n"
    ;;
  *)
    echo -e "Gaffe : \"N'oubliez pas le guide !\"\n"
    ;;
  esac

fi

echo "Fin..."
echo -e "\a\a\aMerci de votre visite. En espérant vous revoir bientôt !\n"

michel ~/test> feed.sh apple camel
Nourrir le chameau avec des pommes...

Avez-vous vu l'affiche ?!  Ne pas nourrir les chameaux !

Nourrissage fait.

Gaffe : "Achetez la nourriture que le zoo fournie à l'entrée, i ***"

Gaffe : "Vous voulez les empoisonner ?"

Fin...
Merci de votre visite. En espérant vous revoir bientôt !

michel ~/test> feed.sh apple
```

```
Utilisation du script feed :       ./feed.sh menu nom-animal
```

Plus d'informations sur les caractères d'échappement à la Section 3.2, « Le caractère Echap (escape) ». Le tableau suivant donne un aperçu des séquences reconnues par la commande **echo** :

Tableau 8.1. Séquences d'échappement reconnues par la commande echo

Séquence	**sens**
\a	Alerte (sonnerie).
\b	Retour arrière.
\c	Supprime les saut de lignes résiduel à la fin.
\e	Escape.
\f	Saut de page.
\n	Saut de ligne.
\r	Retour chariot.
\t	Tabulation horizontale.
\v	Tabulation verticale.
\\	Slash inversé.
\ONNN	Le caractère sur 8 bits dont la valeur en base octal est NNN (de 0 à 3 chiffres en octal).
\NNN	Le caractère sur 8 bits dont la valeur en base octal est NNN (de 1 à 3 chiffres en octal).
\xHH	Le caractère sur 8 bits avec la valeur en base hexadécimale (de 1 à 2 chiffres en hexadécimal).

Pour plus d'information sur la commande **printf** et la façon dont elle permet de formater les résultats, voir les pages info de Bash.

2. Récupérer la saisie utilisateur

2.1. L'emploi de la commande intégrée read

La commande intégrée **read** est la contrepartie de **echo** et **printf**. La syntaxe de la commande **read** est la suivante :

```
read [options] NAME1 NAME2 ... NAMEN
```

Une ligne est lue depuis l'entrée standard, ou depuis le fichier dont le descripteur est fourni en argument à l'option `-u` option. Le premier mot de la ligne est affecté au premier nom `NAME1`, le second mot au second nom, et ainsi de suite, avec les mots résiduels et leurs séparateurs affectés au dernier nom `NAMEN`. Si il y a moins de mots lus sur le flot d'entrée qu'il y a de noms, les noms résiduels sont valorisés à vide.

Les caractères de la valeur de la variable `IFS` sont employés pour découper l'entrée en mots ou jetons ; voir la Section 4.8, « Le découpage de mots ». Le caractère slash inversé peut être utilisé pour inhiber le sens particulier du caractère lu suivant et pour la continuation de la ligne.

Si aucun nom n'est fourni, la ligne lue est affectée à la variable `REPLY`.

La commande **read** renvoie un code à zéro, sauf si un caractère de fin de fichier est rencontré, si **read** dépasse son temps imparti ou si un descripteur de fichier invalide est fourni en argument à l'option `-u` option.

Les options suivantes sont supportées par l'intégrée Bash **read** :

Tableau 8.2. Options de l'intégrée read

Option	sens
-a ANAME	Les mots sont affectés séquentiellement aux éléments de la variable tableau ANAME, en commençant par l'index 0. Tous les éléments sont supprimés de ANAME avant l'affectation. Les autres arguments NAME sont ignorés.
-d DELIM	Le premier caractère de DELIM est utilisé pour terminer la ligne entrée, plutôt que le saut de ligne.
-e	**readline** est utilisé pour obtenir la ligne.
-n NCHARS	**read** s'arrête après avoir lu NCHARS caractères plutôt que d'attendre une ligne entrée complète.
-p PROMPT	Affiche le PROMPT, sans saut de ligne final, avant de tenter de lire tout autre entrée. L'invite est affichée seulement si l'entrée vient d'un terminal.
-r	Si cette option est donnée, le slash inversé n'agit pas comme caractère d'échappement. Le slash inversé est considéré comme faisant parti de la ligne. En particulier, un couple « saut de ligne-slash inversé » ne devrait pas être utilisé comme symbole de continuation de ligne.
-s	Mode Silencieux. Si l'entrée provient d'un terminal, les caractères n'y sont pas renvoyés.
-t TIMEOUT	Donne un temps imparti à **read** et renvoie une erreur si une ligne complète n'a pas été lue avant TIMEOUT secondes. Cette option n'a pas d'effet si **read** n'a pas son entrée depuis un terminal ou un tube.
-u FD	Obtenir les lignes entrées depuis le fichier de descripteur FD.

Ceci est un rapide exemple, améliorant le script leaptest.sh du chapitre précédent :

```
michel ~/test> cat leaptest.sh
#!/bin/bash
# Ce script teste si vous avez saisi une année bissextile ou pas.

echo "Saisissez une année que vous voulez tester  (4 chiffres), puis appuyer sur [ENTREE] :"

read year

if (( ("$year" % 400) == "0" )) || (( ("$year" % 4 == "0") && ("$year" % 100 !=
"0") )); then
  echo "$année bissextile."
else
  echo "année non bissextile."
fi

michel ~/test> leaptest.sh
Saisissez une année que vous voulez tester  (4 chiffres), puis appuyer sur [ENTREE] :
2000
2000 année bissextile.
```

2.2. Demander une entrée utilisateur

L'exemple suivant montre comment vous pouvez vous servir de l'invite pour expliquer ce que l'utilisateur devrait saisir.

```
michel ~/test> cat friends.sh
#!/bin/bash

# Ce programme garde votre carnet d'adresse à jour.

friends="/var/tmp/michel/friends"

echo "Bonjour, "$USER".  Ce script vous enregistrera dans la base de données des amis de Michel."

echo -n "Saisir votre nom et appuyer sur  [ENTREE] : "
read name
echo -n "Saisir votre sexe et appuyer sur [ENTREE] : "
read -n 1 gender
echo

grep -i "$name" "$friends"

if  [ $? == 0 ]; then
  echo "Vous êtes déjà enregistré, terminé."
  exit 1
elif [ "$gender" == "m" ]; then
  echo "Vous êtes ajouté à la liste des amis de Michel."
  exit 1
else
  echo -n "Quel âge avez-vous ? "
  read age
  if [ $age -lt 25 ]; then
    echo -n "De quelle couleur sont vos cheveux ? "
    read colour
    echo "$name $age $colour" >> "$friends"
    echo "Vous êtes ajouté à la liste des amis de Michel.  Merci beaucoup !"
  else
    echo "Vous êtes ajouté à la liste des amis de Michel."
    exit 1
  fi
fi

michel ~/test> cp friends.sh /var/tmp; cd /var/tmp

michel ~/test> touch friends; chmod a+w friends

michel ~/test> friends.sh
Bonjour, michel.  Ce script vous enregistrera dans la base de données des amis de Michel.
Saisir votre nom et appuyer sur  [ENTREE] : michel
Saisir votre sexe et appuyer sur [ENTREE] : m
Vous êtes ajouté à la liste des amis de Michel.

michel ~/test> cat friends
```

Remarquez qu'aucun affichage n'est omis ici. Le script enregistre seulement les informations sur les gens qui intéressent Michel, mais il avertira toujours que vous avez été ajouté à la liste, sauf si vous y êtes déjà.

D'autres gens peuvent maintenant lancer le script :

```
[anny@octarine tmp]$ friends.sh
Bonjour, anny.  Ce script vous enregistrera dans la base de données des amis de Michel.
Saisir votre nom et appuyer sur  [ENTREE] : anny
Saisir votre sexe et appuyer sur [ENTREE] :f
Quel âge avez-vous ? 22
De quelle couleur sont vos cheveux ? noir
Vous êtes ajouté à la liste des amis de Michel.
```

Finalement, la liste `friends` ressemble à ceci :

```
tille 24 noir
anny 22 noir
katya 22 blonde
maria 21 noir
--output omitted--
```

Bien sûr, cette situation n'est pas idéale, parce que chacun peut renseigner (mais pas se retirer) du fichier de Michel. Vous pouvez palier ce défaut en utilisant des accès privilégiés au fichier de script, voir SUID et SGID dans le guide d'introduction à Linux.

2.3. Redirection et descripteurs de fichiers

2.3.1. Généralité

Comme vous avez pu le réaliser à la suite d'une utilisation rudimentaire du Shell, l'entrée et la sortie d'une commande peuvent être redirigées avant son exécution, grâce à la notation appropriée - les

opérateurs de redirection - interprétée par le Shell. La redirection peut aussi être employée pour ouvrir et fermer des fichiers dans l'environnement d'exécution du Shell.

La redirection peut aussi se produire dans le script, de sorte qu'il puisse recevoir son entrée depuis un fichier, par exemple, ou qu'il puisse envoyer les résultats vers un fichier. Ultérieurement, l'utilisateur peut visualiser le fichier de résultat, ou il peut être exploité en entrée par un autre script.

Les fichiers d'entrée et de sortie sont désignés par des entiers indicateurs qui repèrent tout les fichiers ouverts d'un processus donné. Ces valeurs numériques sont connues sous le nom de descripteurs de fichiers. Les descripteurs les plus courant sont *stdin*, *stdout* et *stderr*, avec les numéros 0, 1 et 2, respectivement. Ces numéros et leur entité respective sont réservés. Bash peut aussi considérer des ports TCP ou UDP sur les hôtes du réseau en tant que descripteur.

L'affichage ci-dessous montre comment les descripteurs réservés pointent sur des entités concrètes :

```
michel ~> ls -l /dev/std*
lrwxrwxrwx  1 root    root     17 Oct  2 07:46 /dev/stderr -> ../proc/self/fd/2
lrwxrwxrwx  1 root    root     17 Oct  2 07:46 /dev/stdin -> ../proc/self/fd/0
lrwxrwxrwx  1 root    root     17 Oct  2 07:46 /dev/stdout -> ../proc/self/fd/1

michel ~> ls -l /proc/self/fd/[0-2]
lrwx------  1 michel  michel   64 Jan 23 12:11 /proc/self/fd/0 -> /dev/pts/6
lrwx------  1 michel  michel   64 Jan 23 12:11 /proc/self/fd/1 -> /dev/pts/6
lrwx------  1 michel  michel   64 Jan 23 12:11 /proc/self/fd/2 -> /dev/pts/6
```

Notez que chaque process a sa propre vue des fichiers sous `/proc/self`, puisque c'est un lien symbolique vers `/proc/<process_ID>`.

Vous pouvez consulter **info MAKEDEV** et **info proc** pour plus de détails sur le sous-répertoire `/proc` et la façon dont votre système gère les descripteurs standards de chaque processus lancé.

Quand vous exécutez n'importe quelle commande, les étapes suivantes sont déroulées, dans l'ordre :

- Si la sortie standard de la commande précédente a été dirigée sur l'entrée standard de la commande en cours, alors `/proc/<current_process_ID>/fd/0` est modifié pour cibler le même tube anonyme que `/proc/<previous_process_ID/fd/1`.
- Si la sortie standard de la commande en cours a été dirigée sur l'entrée standard de la commande à suivre, alors `/proc/<current_process_ID>/fd/1` est modifié pour cibler un autre tube anonyme.
- La redirection pour la commande en cours est traitée de gauche à droite.
- La redirection de « N>&M » ou « N<&M » après une commande a pour effet de créer ou modifier le lien symbolique `/proc/self/fd/N` avec la même cible que le lien symbolique `/proc/self/fd/M`.
- Les redirections de « N> file » et « N< file » ont pour effet de créer ou modifier le lien symbolique `/proc/self/fd/N` avec le fichier cible.
- La fermeture du descripteur de fichier « N>&- » a pour effet de supprimer le lien symbolique `/proc/self/fd/N`.
- Seulement maintenant la commande courante est exécutée.

Quand vous lancez un script depuis la ligne de commande, rien ne change tellement parce que le processus Shell enfant utilisera les mêmes descripteurs que son parent. Quand le parent n'existe pas, par exemple quand vous lancez un script par l'outil *cron* les descripteurs standards sont des tubes et autres fichiers (temporaires), à moins qu'un autre moyen de redirection soit employé. Ceci est démontré dans l'exemple ci-dessous, lequel produit le résultat avec un simple script **at** :

```
michel ~> date
Fri Jan 24 11:05:50 CET 2003

michel ~> at 1107
avertissement : les commandes seront exécutées avec (par ordre)
a) $SHELL b) login shell c)/bin/sh
at> ls -l /proc/self/fd/ > /var/tmp/fdtest.at
```

```
at> <EOT>
job 10 at 2003-01-24 11:07

michel ~> cat /var/tmp/fdtest.at
total 0
lr-x------    1 michel michel  64 Jan 24 11:07 0 -> /var/spool/at/!0000c010959eb (deleted)
l-wx------    1 michel michel  64 Jan 24 11:07 1 -> /var/tmp/fdtest.at
l-wx------    1 michel michel  64 Jan 24 11:07 2 -> /var/spool/at/spool/a0000c010959eb
lr-x------    1 michel michel  64 Jan 24 11:07 3 -> /proc/21949/fd
```

Et un avec **cron** :

```
michel ~> crontab -l
# NE PAS EDITER CE FICHIER - éditer le modèle et le réinstaller.
# (/tmp/crontab.21968 installed on Fri Jan 24 11:30:41 2003)
# (Cron version -- $Id: chap8.xml,v 1.9 2006/09/28 09:42:45 tille Exp $)
32 11 * * * ls -l /proc/self/fd/ > /var/tmp/fdtest.cron

michel ~> cat /var/tmp/fdtest.cron
total 0
lr-x------    1 michel michel  64 Jan 24 11:32 0 -> pipe:[124440]
l-wx------    1 michel michel  64 Jan 24 11:32 1 -> /var/tmp/fdtest.cron
l-wx------    1 michel michel  64 Jan 24 11:32 2 -> pipe:[124441]
lr-x------    1 michel michel  64 Jan 24 11:32 3 -> /proc/21974/fd
```

2.3.2. Redirection des erreurs

Dans l'exemple précédent il apparaît clairement que vous pouvez fournir les fichiers d'entrée et de sortie à un script (voir Section 2.4, « Fichier d'entrée et fichier de sortie » pour plus de détails), mais certains oublient de rediriger les erreurs - un affichage dont peut dépendre la suite. Aussi, si vous êtes chanceux, les erreurs vous seront adressées par mail et d'éventuels dysfonctionnements pourront vous apparaître. Si vous n'êtes pas chanceux, les erreurs feront planter votre script et ne seront ni capturées ni adressées nulle part, par conséquent vous ne pourrez débugger.

Quand vous redirigez les erreurs, faites attention à l'ordre de préséance. Par exemple, cette commande lancée dans `/var/spool`

```
ls -l * 2> /var/tmp/unaccessible-in-spool
```

va rediriger la sortie standard de la commande **ls** vers le fichier `unaccessible-in-spool` dans `/var/tmp`. La commande

```
ls -l * > /var/tmp/spoollist 2>&1
```

redirigera et le standard d'entrée et le standard d'erreur vers le fichier `spoollist`. La commande

```
ls -l * 2 >& 1 > /var/tmp/spoollist
```

dirige seulement le standard de sortie vers le fichier de destination, parce que le standard d'erreurs est copié vers le standard de sortie avant que le standard de sortie soit redirigé.

Par commodité, les erreurs sont souvent redirigées vers `/dev/null`, si il est sûr qu'elles n'ont pas d'intérêt. Des centaines d'exemples peuvent être trouvés dans les scripts de lancement de votre système.

Bash autorise à la fois le standard de sortie et le standard d'erreurs à être redirigés vers le fichier dont le nom est le résultat de l'expansion de `FILE` avec cette forme :

```
&> FILE
```

C'est l'équivalent de **> FILE 2>&1**, la forme employée dans les exemples précédents. C'est aussi combiné souvent avec la redirection vers `/dev/null`, par exemple quand vous voulez juste qu'une commande s'exécute, quelque soit le résultat ou le statut qu'elle donne.

2.4. Fichier d'entrée et fichier de sortie

2.4.1. Avec /dev/fd

Le répertoire `/dev/fd` contient des entrées nommées `0`, `1`, `2`, etc. Ouvrir le fichier `/dev/fd/N` est équivalent à dupliquer le descripteur *N*. Si votre système possède `/dev/stdin`, `/dev/stdout` et `/dev/stderr`, vous verrez qu'ils sont équivalents à `/dev/fd/0`, `/dev/fd/1` et `/dev/fd/2`, respectivement.

La principale utilisation des fichiers `/dev/fd` est faites par le Shell. Ce mécanisme permet aux programmes qui utilisent des chemins en paramètre de voir le standard d'entrée et le standard de sortie de la même façon que d'autres chemins. Si `/dev/fd` n'est pas disponible sur votre système, vous devrez trouver un moyen pour résoudre ce problème. Ce qui peut être fait, par exemple, avec un tiret (-) qui indique que le programme doit lire depuis un tube. Un exemple :

```
michel ~> filter body.txt.gz | cat header.txt - footer.txt
Ce texte est affiché au début de chaque travail d'affichage et merci à l'administrateur d'avoir mis en

Texte à filtrer.

Ce texte est à afficher à la fin de chaque travail d'affichage.
```

La commande **cat** d'abord lit le fichier `header.txt`, puis son standard d'entrée lequel est le standard de sortie de la commande **filter**, et finalement le fichier `footer.txt`. Le sens spécial du tiret en tant que paramètre de ligne de commande pour se référer au standard d'entrée ou de sortie est une confusion qui a perduré dans beaucoup de programmes. Il peut aussi y avoir des problèmes quand vous spécifiez le tiret en tant que premier paramètre, puisqu'il peut être interprété comme une option de la commande précédente. L'usage de `/dev/fd` permet l'uniformité et évite la confusion :

```
michel ~> filter body.txt | cat header.txt /dev/fd/0 footer.txt | lp
```

Dans cet exemple propre, toutes les sorties sont cumulées dans l'entonnoir **lp** pour les envoyer vers l'imprimante par défaut.

2.4.2. Read et exec

2.4.2.1. Assigner des descripteurs de fichiers

Une autre façon de considérer les descripteurs de fichiers est d'y penser comme un indicateur numérique assigné à un fichier. Au lieu d'employer le nom de fichier, vous pouvez employer le numéro de descripteur. L'intégrée **exec** peut être utilisée pour remplacer le Shell du process actif ou pour réassigner des descripteurs de fichiers à ce Shell. Par exemple, elle peut être employée pour assigner un descripteur de fichier à un fichier. Employez

```
exec fdN> file
```

pour assigner le descripteur N au fichier `file` en sortie, et

```
exec fdN< file
```

pour assigne le descripteur N au fichier `file` en entrée. Après qu'un descripteur ait été assigné à un fichier, il peut être employé avec les opérateurs Shell de redirection, comme le montre l'exemple suivant :

```
michel ~> exec 4> result.txt

michel ~> filter body.txt | cat header.txt /dev/fd/0 footer.txt >& 4

michel ~> cat result.txt
Ce texte est affiché au début de chaque travail d'impression et remercie l'administrateur d'avoir mis e

Texte à filtrer.

Ce texte est à afficher à la fin de chaque travail d'affichage.
```

Le descripteur 5

L'emploi de ce descripteur de fichier peut être cause de soucis, voir the Advanced Bash-Scripting Guide, chapitre 16. Il vous est sérieusement recommandé de ne pas l'employer.

2.4.2.2. Read dans un script

Ce qui suit est un exemple qui montre comment vous pouvez permuter l'entrée depuis un fichier sur l'entrée de la ligne de commande et vice-versa :

```
michel ~/testdir> cat sysnotes.sh
#!/bin/bash

# Ce script fait un index des fichiers de configuration importants, les sauvegarde tous dans
# un fichier et autorise l'ajout de commentaires à chaque fichier.

CONFIG=/var/tmp/sysconfig.out
rm "$CONFIG" 2>/dev/null

echo "Le résultat sera mémorisé dans $CONFIG."

# Crée fd 7 avec la cible de fd 0 (save stdin "value")
exec 7<&0

# update fd 0 to target file /etc/passwd
exec < /etc/passwd

# Read the first line of /etc/passwd
read rootpasswd

echo "Sauvegarde des info de root..."
echo "Les infos du compte root :" >> "$CONFIG"
echo $rootpasswd >> "$CONFIG"

# Modifie fd 0 pour cibler fd 7  (old fd 0 target); supprime fd 7
exec 0<&7 7<&-

echo -n "Entrez un commentaire ou [ENTER] sans commentaire : "
read comment; echo $comment >> "$CONFIG"

echo "Mémorise les infos de l'hôte..."

# D'abord préparer un fichier hôte sans commentaires
TEMP="/var/tmp/hosts.tmp"
cat /etc/hosts | grep -v "^#" > "$TEMP"

exec 7<&0
exec < "$TEMP"

read ip1 name1 alias1
read ip2 name2 alias2

echo "La configuration de l'hôte local :" >> "$CONFIG"

echo "$ip1 $name1 $alias1" >> "$CONFIG"
echo "$ip2 $name2 $alias2" >> "$CONFIG"

exec 0<&7 7<&-

echo -n "Entrez un commentaire ou [ENTER] sans commentaire : "
read comment; echo $comment >> "$CONFIG"
rm "$TEMP"

michel ~/testdir> sysnotes.sh
Le résultat sera mémorisé dans /var/tmp/sysconfig.out.
Sauvegarde des info de root...
Entrez des commentaires [ENTER] for no comment : pense-bête pour mot de passe : vacance
Sauvegarde des informations système...
Entrez des commentaires [ENTER] for no comment : dans le DNS central

michel ~/testdir> cat /var/tmp/sysconfig.out
Les infos du compte root :
root:x:0:0:root:/root:/bin/bash
pense-bête pour mot de passe : vacance
Votre configuration sur la machine locale :
127.0.0.1 localhost.localdomain localhost
192.168.42.1 tintagel.kingarthur.com tintagel
dans le DNS central
```

2.4.3. Fermer les descripteurs de fichiers

Parce que les processus enfants héritent des descripteurs ouverts, c'est une bonne pratique que de

fermer les descripteurs quand on en a plus besoin. Ceci est fait avec la syntaxe suivante

```
exec fd<&-
```

syntax. Dans l'exemple ci-dessus, le descripteur 7, qui a été assigné au standard d'entrée, est fermé chaque fois que l'utilisateur a besoin d'utiliser le périphérique d'entrée standard, habituellement le clavier.

Suit un exemple simple de redirection du standard d'erreurs sur un tube :

```
michel ~> cat listdirs.sh
#!/bin/bash

# Ce script garde le standard de sortie, tandis qu'il redirige le standard d'erreurs
# afin d'être traité par awk.

INPUTDIR="$1"

# fd 6 targets fd 1 target (console out) in current shell
exec 6>&1

# fd 1 targets pipe, fd 2 targets fd 1 target (pipe),
# fd 1 targets fd 6 target (console out), fd 6 closed, execute ls
ls "$INPUTDIR"/* 2>&1 >&6 6>&- \
                                # Closes fd 6 for awk, but not for ls.

| awk 'BEGIN { FS=":" } { print "YOU HAVE NO ACCESS TO" $2 }' 6>&-

# fd 6 closed for current shell
exec 6>&-
```

2.4.4. Les documents *intégrés* (NdT : here documents, que l'on appele aussi 'document lié')

Fréquemment votre script peut avoir besoin d'appeler un autre programme ou script qui nécessite une entrée. Le document *intégré* fournit un moyen d'enjoindre au Shell de lire l'entrée de la source actuelle jusqu'à une ligne contenant seulement la chaîne ad hoc (pas de blancs résiduels). Toutes les lignes lues jusqu'à celle-là sont envoyées comme entrée standard de la commande.

Le résultat est que vous n'avez pas besoin de faire appel à différents fichiers ; vous pouvez utiliser les caractères spéciaux du Shell, et c'est plus lisible qu'un flot d'**echo** :

```
michel ~> cat startsurf.sh
#!/bin/bash

# Ce script fournit aux usagers un moyen facile de choisir entre plusieurs navigateurs.

echo "Voici les navigateurs WEB de ce système :"

# Début du document 'intégré'
cat << BROWSERS
mozilla
links
lynx
konqueror
opera
netscape
BROWSERS
# Fin du document 'intégré'

echo -n "Lequel préférez-vous ? "
read browser

echo "Démarrage de $browser, Merci de patienter..."
$browser &

michel ~> startsurf.sh
Voici les navigateurs WEB de ce système :
mozilla
links
lynx
konqueror
opera
netscape
Lequel préférez-vous ? opera
Démarrage de opera, Merci de patienter...
```

Bien que nous parlions de *document intégré*, il doit être un bloc dans le script. Voici un exemple qui installe un paquetage automatiquement, même si vous devriez normalement confirmer :

```
#!/bin/bash

# Ce script installe un paquetage automatiquement avec yum.

if [ $# -lt 1 ]; then
        echo "Utilisation : $0 package."
        exit 1
fi

yum install $1 << CONFIRM
y
CONFIRM
```

Et voici comment le script tourne. Quand la question « Is this ok [y/N] » apparaît, le script répond « y » automatiquement :

```
[root@picon bin]# ./install.sh tuxracer
Gathering header information file(s) from server(s)
Server: Fedora Linux 2 - i386 - core
Server: Fedora Linux 2 - i386 - freshrpms
Server: JPackage 1.5 for Fedora Core 2
Server: JPackage 1.5, generic
Server: Fedora Linux 2 - i386 - updates
Finding updated packages
Downloading needed headers
Resolving dependencies
Dependencies resolved
I will do the following:
[install: tuxracer 0.61-26.i386]
Is this ok [y/N]: EnterDownloading Packages
Running test transaction:
Test transaction complete, Success!
tuxracer 100 % done 1/1
Installed:  tuxracer 0.61-26.i386
Transaction(s) Complete
```

3. Résumé

Dans ce chapitre nous avons appris comment afficher des commentaires et comment solliciter une saisie de la part de l'utilisateur. Ce qui est effectué habituellement par la combinaison de **echo**/**read**. Nous avons aussi abordé comment les fichiers peuvent être employés en entrée et en sortie par le biais des descripteurs de fichiers et la redirection, et comment cela peut être combiné avec la saisie utilisateur.

Nous avons insisté sur l'importance des messages à destination des utilisateurs dans vos scripts. Comme d'habitude quand d'autres se servent de vos scripts, mieux faut donner trop d'informations que pas assez. Le document *intégré* est un type de construction Shell qui permet la création de liste, contenant des choix pour les utilisateurs. Cette construction peut aussi être employée pour exécuter d'autres sortes de tâches interactives en tâche de fond, sans intervention.

4. Exercices

Ces exercices sont des applications pratiques des constructions abordées dans ce chapitre. Quand vous écrivez un script, il est préférable de tester dans un répertoire qui ne contienne pas trop de données. Ecrire chaque étape, puis tester cette portion de code, plutôt que d'écrire tout d'un seul coup.

1. Ecrire un script qui demande l'âge de l'usager. Si il est égal ou supérieur à 16, afficher un message indiquant que l'usager est autorisé à boire de l'alcool. Si l'usager est en dessous de 16 ans, afficher un message disant combien d'années l'usager devra attendre avant d'être autorisé légalement à boire de l'alcool.

   En plus, calculer combien de bière un usager de plus de 18 ans a bu statistiquement (100 litres/an) et donner cette information à l'usager.

2. Ecrire un script qui prenne un fichier en paramètre. Servez-vous d'un document *intégré* qui présente à l'usager différents choix pour compresser ce fichier. Les choix possibles peuvent être **gzip**, **bzip2**, **compress** et **zip**.

3. Ecrire un script appelé `homebackup` qui automatise **tar** afin que la personne qui lance le script ait

toujours les options désirées (`cvp`) et sauvegarder le répertoire de destination (`/var/backups`) afin de faire une sauvegarde du répertoire racine de l'usager. Ajouter les fonctionnalités suivantes :

- Tester le nombre de paramètres. Le script ne devrait pas en avoir. Si des paramètres sont présents, sortir après avoir affiché le message d'utilisation du script.
- Déterminer si le répertoire `backups` a assez d'espace libre pour contenir la sauvegarde.
- Demander à l'usager si il veut une sauvegarde complète ou incrémentale. Si l'usager n'a pas eu encore de sauvegarde complète, afficher un message disant qu'une complète sera faite. Dans le cas d'une sauvegarde incrémentale, ne la faire que si la complète n'est pas plus vieille que d'une semaine.
- Comprimer la sauvegarde avec un outil de compression quelconque. Informer l'usager que le script va compresser, parce que ça peut prendre du temps, et l'usager pourrait s'inquiéter si aucun résultat n'apparaît à l'écran.
- Afficher un message indiquant à l'usager la taille de la sauvegarde compressée.

Voir **info tar** ou Introduction à Linux, chapitre 9 : « Preparing your data » for background information.

4. Ecrire un script appelé `simple-useradd.sh` qui ajoute un utilisateur au système local. Le script devrait :
   - Ne prendre qu'un seul paramètre, ou sinon sortir avec un message d'utilisation.
   - Contrôler `/etc/passwd` et sélectionner le premier identifiant non affecté. Afficher un message contenant l'identifiant.
   - Créer un groupe privé pour cet utilisateur, en contrôlant le fichier `/etc/group`. Afficher un message contenant l'identifiant du groupe.
   - Rassembler des informations sur l'utilisateur : un commentaire décrivant cet utilisateur, choix dans une liste de Shell (tester sa validité, si non sortir avec un message), la date d'expiration du compte, les autres groupes auxquels ce nouvel utilisateur peut appartenir.
   - Avec les informations obtenues, ajouter une ligne à `/etc/passwd`, `/etc/group` et `/etc/shadow` ; créer le répertoire racine de l'utilisateur (avec les autorisations correctes ! Ajouter l'utilisateur aux groupes secondaires désirés.
   - Définir le mot de passe de cet utilisateur à une chaîne connue.
5. Réécrire le script de la Section 2.1.4, « Test de l'existence d'un fichier » afin qu'il lise son entrée depuis la saisie utilisateur plutôt que du premier paramètre.

Chapitre 9. Tâches répétitives

Table des matières

1. La boucle loop
   - 1.1. Comment ça marche ?
   - 1.2. Exemples
2. La boucle while
   - 2.1. Qu'est-ce que c'est ?
   - 2.2. Exemples
3. La boucle until
   - 3.1. Qu'est-ce que c'est ?
   - 3.2. Exemple
4. Redirection d'entrée/sortie et boucles
   - 4.1. Redirection des entrées

4.2. Redirection des sorties
5. Break et continue
5.1. L'intégrée break
5.2. L'intégrée continue
5.3. Exemples
6. Faire des menus avec l'intégrée select
6.1. Généralité
6.2. Sous-menus
7. L'intégrée shift
7.1. Qu'est-ce qu'elle fait ?
7.2. Exemples
8. Résumé
9. Exercices

Résumé

À la fin de ce chapitre, vous serez capable de

- Ecrire des boucles **for**, **while** and **until** , et décider quelle boucle convient à quel besoin.
- Utiliser les intégrées Bash **break** et **continue**
- Ecrire des scripts avec l'instruction **select**.
- Ecrire des scripts qui admettent un nombre variable de paramètres.

1. La boucle loop

1.1. Comment ça marche ?

La boucle **for** est la première des 3 structures de boucles du Shell. Cette boucle autorise la spécification d'une liste de valeurs. Une liste de commandes est exécutée pour chaque valeur de la liste.

La syntaxe de cette boucle est :

```
for NOM [in LIST ]; do COMMANDES; done
```

Si **[in LIST]** est absent, il est remplacé par **in** `$@` et **for** exécute les **COMMANDES** une fois pour chaque paramètre positionnel déclaré (voir Section 2.5, « Paramètres spéciaux » et Section 2.1.2, « Contrôle des paramètres de la ligne de commande »).

Le statut retourné est le statut d'exécution de la dernière commande exécutée. Si aucune commande n'est exécutée parce que `LIST` ne résulte en aucun élément, le code retour est zéro.

`NOM` peut être tout nom de variable, même si `i` est employé très souvent. `LIST` peut être toute liste de mots, chaînes ou nombres qui peuvent être des littéraux ou générés par toute commande. Les **COMMANDES** à exécuter peuvent être aussi toute commande système, script, programme ou instruction Shell. Au premier passage dans la boucle, `NOM` est valorisé à la valeur du premier élément dans `LIST`. Au deuxième passage, sa valeur est donnée par le second élément dans la liste, et ainsi de suite. La boucle termine quand `NOM` a pris une fois la valeur de chaque élément de `LIST` et qu'il ne reste plus d'éléments dans `LIST`.

1.2. Exemples

1.2.1. Utiliser la substitution de commande pour spécifier les éléments de LIST

Le premier exemple est une ligne de commande qui montre l'emploi d'une boucle **for** pour effectuer une copie de sauvegarde de chaque fichier `.xml`. Une fois cette commande exécutée, il est plus sûr de travailler sur les sources :

```
[carol@octarine ~/articles] ls *.xml
file1.xml  file2.xml  file3.xml

[carol@octarine ~/articles] ls *.xml > list

[carol@octarine ~/articles] for i in `cat list`; do cp "$i" "$i".bak ; done

[carol@octarine ~/articles] ls *.xml*
file1.xml  file1.xml.bak  file2.xml  file2.xml.bak  file3.xml  file3.xml.bak
```

Celui-ci liste les fichiers de `/sbin` qui sont des purs fichiers texte, et donc peut-être des scripts :

```
for i in `ls /sbin`; do file /sbin/$i | grep ASCII; done
```

1.2.2. Utiliser la valeur d'une variable pour spécifier les éléments de LIST

Ce qui suit est un script spécifique pour convertir des fichiers HTML respectant un certain schéma en fichiers PHP. La conversion est faite en extrayant les 25 premières lignes et les 21 dernières, les remplaçant par 2 étiquettes PHP qui correspondent aux lignes d'entête et d'empied :

```
[carol@octarine ~/html] cat html2php.sh
#!/bin/bash
# specific conversion script for my html files to php
LIST="$(ls *.html)"
for i in "$LIST"; do
     NEWNAME=$(ls "$i" | sed -e 's/html/php/')
     cat beginfile > "$NEWNAME"
     cat "$i" | sed -e '1,25d' | tac | sed -e '1,21d'| tac >> "$NEWNAME"
     cat endfile >> "$NEWNAME"
done
```

Comme nous ne faisons pas un comptage des lignes, il n'y a pas moyen de savoir le numéro de ligne à laquelle commencer la suppression avant d'avoir atteint la fin. Cette difficulté est surmontée en utilisant **tac**, lequel inverse l'ordre des lignes dans un fichier.

La commande basename

Plutôt que d'employer **sed** pour remplacer le suffixe `html` par `php`, il serait plus propre d'employer la commande **basename**. Voir les pages man pour plus de détails.

Caractères spéciaux

Vous aurez des soucis si la liste est transformée en noms de fichiers contenant des espaces et autres caractères irréguliers. Une construction plus appropriée pour obtenir la liste serait d'utiliser la fonction globale du Shell, comme ceci :

```
for i in $PATHNAME/*; do
        commands
done
```

2. La boucle while

2.1. Qu'est-ce que c'est ?

La structure **while** permet une exécution répétitive d'une liste de commandes tant que la commande qui contrôle le **while** s'exécute avec succès (code retour égal à zéro). La syntaxe est :

```
while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done
```

CONTROL-COMMAND peut être toute(s) commande(s) qui peut s'achever avec un statut de succès ou d'échec. Le **CONSEQUENT-COMMANDS** peut être tout programme, script ou bloc

Shell.

Dès que **CONTROL-COMMAND** échoue, la boucle est arrêtée. Dans un script, la commande suivant l'instruction **done** est exécutée.

Le statut retourné est le statut d'exécution de la dernière commande de **CONSEQUENT-COMMANDS** ou zéro si aucune n'est exécutée

2.2. Exemples

2.2.1. Exemple simple d'utilisation de while

Voici un exemple pour les impatients :

```
#!/bin/bash

# Le script ouvre 4 fenêtres de terminal.

i="0"

while [ $i -lt 4 ]
do
xterm &
i=$[$i+1]
done
```

2.2.2. Des boucles while imbriquées

L'exemple ci-dessous a été écrit pour copier des images qui sont prises par une WEBcam vers un répertoire WEB. Toute les 5 minutes une image est prise. Toute les heures, un nouveau répertoire est créé, pour contenir les images de cette heure. Chaque jour, un nouveau répertoire est créé contenant 24 sous-répertoires. Le script s'exécute en tâche de fond.

```
#!/bin/bash

# Ce script copie les fichiers de mon répertoire racine dans le répertoire du serveur WEB.
# (Utilisez des clés scp et SSH pour un répertoire distant)
# Un nouveau répertoire est créé à chaque heure.

PICSDIR=/home/carol/pics
WEBDIR=/var/www/carol/webcam

while true; do
        DATE=`date +%Y%m%d`
        HOUR=`date +%H`
        mkdir $WEBDIR/"$DATE"

        while [ $HOUR -ne "00" ]; do
                DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
                mkdir "$DESTDIR"
                mv $PICDIR/*.jpg "$DESTDIR"/
                sleep 3600
                HOUR=`date +%H`
        done
done
```

Notez l'emploi de l'instruction **true**. Il signifie : continuer l'exécution jusqu'à une interruption forcée (avec **kill** ou **Ctrl**+**C**).

Ce petit script peut être utilisé pour des tests de simulation ; il génère des fichiers :

```
#!/bin/bash

# Ce script génère un fichier toute les 5 minutes

while true; do
touch pic-`date +%s`.jpg
sleep 300
done
```

Notez l'emploi de la commande **date** pour générer toute sorte de nom de fichier et de répertoire. Voir les pages man pour plus de détails.

> **Mettez à profit le système**
>
> L'exemple précédent existe pour le besoin de la démonstration. Des contrôles réguliers peuvent être facilement faits avec l'outil système *cron*. Ne pas oublier de rediriger les sorties et les erreurs quand un script est utilisé par crontab!

2.2.3. Contrôle d'une boucle while avec des saisies au clavier

Ce script peut être interrompu par l'usager quand une séquence **Ctrl**+**C** est frappée :

```
#!/bin/bash

# Ce script vous apporte sagesse

FORTUNE=/usr/games/fortune

while true; do
echo "Sur quel sujet voulez-vous un conseil ?"
cat << topics
politique
startrek
noyau
sports
excusesbidon
magie
amour
littérature
drogues
éducation
topics

echo
echo -n "Faites votre choix : "
read topic
echo
echo "Conseil gratuit sur le sujet $topic : "
echo
$FORTUNE $topic
echo

done
```

Un document *intégré* est utilisé pour présenter à l'usager les choix possibles. Et de nouveau le test **true** répète les commandes de la liste **CONSEQUENT-COMMANDS** encore et encore.

2.2.4. Calcul d'une moyenne

Ce script calcule la moyenne à partir de la saisie utilisateur, qui est testée avant d'être traitée : Si la saisie n'est pas dans un intervalle, un message est affiché. Si **q** est frappée la boucle est abandonnée :

```
#!/bin/bash

# Calcul de la moyenne d'une série de nombres.

SCORE="0"
AVERAGE="0"
SUM="0"
NUM="0"

while true; do

  echo -n "Entrez votre score [0-100%] ('q' pour quitter) : "; read SCORE;

  if (("$SCORE" < "0"))  || (("$SCORE" > "100")); then
    echo "Soyez sérieux.  Banal, essayer encore : "
  elif [ "$SCORE" == "q" ]; then
    echo "Evaluation moyenne : $AVERAGE%."
    break
  else
    SUM=$[$SUM + $SCORE]
    NUM=$[$NUM + 1]
    AVERAGE=$[$SUM / $NUM]
  fi

done

echo "Je quitte."
```

Remarquez que les variables sur les dernières lignes ne sont pas protégées pour pouvoir en faire des calculs.

3. La boucle until

3.1. Qu'est-ce que c'est ?

La boucle **until** est très similaire à **while**, à part qu'elle s'exécute jusqu'à ce que **TEST-COMMAND** s'exécute avec succès. Tant que cette commande échoue, la boucle se poursuit. La syntaxe est la même que pour la boucle **while** :

```
until TEST-COMMAND; do CONSEQUENT-COMMANDS; done
```

Le statut retourné est le statut d'exécution de la dernière commande exécutée dans la liste **CONSEQUENT-COMMANDS** ou zéro si aucune n'est exécutée. **TEST-COMMAND** peut être toute commande qui peut s'achever avec un statut de succès ou d'échec, et **CONSEQUENT-COMMANDS** peut être toute commande UNIX, script ou bloc Shell.

Comme nous l'avons déjà expliqué auparavant ; le « ; » peut être remplacé par un ou plusieurs sauts de lignes quelque soit l'endroit où il apparaît.

3.2. Exemple

Script amélioré de `picturesort.sh` (voir Section 2.2.2, « Des boucles while imbriquées »), qui teste l'espace disque disponible. Si pas assez d'espace disque disponible, supprimer les images des mois précédents :

```
#!/bin/bash

# Ce script copie les fichiers de mon répertoire racine dans le répertoire du serveur WEB.
# Un nouveau répertoire est créé à chaque heure.
# Si les images prennent trop de place, les plus anciennes sont supprimées.

while true; do
        DISKFUL=$(df -h $WEBDIR | grep -v File | awk '{print $5 }' | cut -d "%" -f1 -)

        until [ $DISKFUL -ge "90" ]; do

                DATE=`date +%Y%m%d`
                HOUR=`date +%H`
                mkdir $WEBDIR/"$DATE"

                while [ $HOUR -ne "00" ]; do
                        DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
                        mkdir "$DESTDIR"
                        mv $PICDIR/*.jpg "$DESTDIR"/
                        sleep 3600
                        HOUR=`date +%H`
                done

        DISKFULL=$(df -h $WEBDIR | grep -v File | awk '{ print $5 }' | cut -d "%" -f1 -)
        done

        TOREMOVE=$(find $WEBDIR -type d -a -mtime +30)
        for i in $TOREMOVE; do
                rm -rf "$i";
        done

done
```

Notez l'initialisation des variables `HOUR` et `DISKFULL` et l'emploi d'options avec **ls** et **date** afin d'obtenir une liste correcte pour `TOREMOVE`.

4. Redirection d'entrée/sortie et boucles

4.1. Redirection des entrées

Plutôt que de contrôler une boucle en testant le résultat d'une commande ou la saisie utilisateur, vous pouvez spécifier un fichier depuis lequel l'entrée est lue pour contrôler la boucle. Dans un tel cas, **read** est souvent la commande de contrôle. Tant que des lignes sont entrées dans la boucle, les

commandes de la boucle sont exécutées. Dès que toutes les lignes ont été lues la boucle s'arrête.

Parce que la structure de boucle est considérée comme étant une structure de commande (tel que **while TEST-COMMAND; do CONSEQUENT-COMMANDS; done**), la redirection doit apparaître après l'instruction **done** afin de respecter la syntaxe.

```
command < file
```

Ce genre de redirection convient aux autres types de boucles.

4.2. Redirection des sorties

Dans l'exemple suivant, le résultat de la commande **find** est utilisé comme entrée de la commande **read** afin de contrôler une boucle **while** :

```
[carol@octarine ~/testdir] cat archiveoldstuff.sh
#!/bin/bash

# Ce script crée un sous-répertoire dans le répertoire courant où sont gardés les
# fichiers supprimés.
# Cela pourrait être adapté à cron (avec modifications) pour être exécuté
# chaque semaine ou mois.

ARCHIVENR=`date +%Y%m%d`
DESTDIR="$PWD/archive-$ARCHIVENR"

mkdir "$DESTDIR"

# avec guillemets pour récupérer les noms de fichiers ayant des espaces, avec read -d pour la suite
# fool-proof usage:
find "$PWD" -type f -a -mtime +5 | while read -d $'\000' file

do
gzip "$file"; mv "$file".gz "$DESTDIR"
echo "$file archived"
done
```

Les fichiers sont compressés avant d'être déplacés dans le répertoire d'archive.

5. Break et continue

5.1. L'intégrée break

L'instruction **break** est employée pour quitter la boucle en cours avant sa fin normale. Ceci peut être nécessaire quand vous ne savez pas à l'avance combien de fois la boucle devra s'exécuter, par exemple parce qu'elle est dépendante de la saisie utilisateur.

Cet exemple montre une boucle **while** qui peut être interrompue. Ceci est une version légèrement améliorée du script `wisdom.sh` de la Section 2.2.3, « Contrôle d'une boucle while avec des saisies au clavier ».

```
#!/bin/bash

# Ce script vous apporte sagesse
# Vous pouvez maintenant quitter d'une façon décente.

FORTUNE=/usr/games/fortune

while true; do
echo "Sur quel sujet voulez-vous un conseil?"
echo "1.  politique"
echo "2.  startrek"
echo "3.  noyau"
echo "4.  sports"
echo "5.  excusesbidon"
echo "6.  magie"
echo "7.  amour"
echo "8.  littérature"
echo "9.  drogues"
echo "10. éducation"
echo

echo -n "Entrez votre choix, ou 0 pour quitter : "
read choice
echo

case $choice in
```

```
      1)
      $FORTUNE politique
      ;;
      2)
      $FORTUNE startrek
      ;;
      3)
      $FORTUNE noyau
      ;;
      4)
      echo "Le sport est une perte d'argent, d'énergie et de temps."
      echo "Retournez à votre clavier."
      echo -e "\t\t\t\t -- \"Unhealthy is my middle name\" Soggie."
      ;;
      5)
      $FORTUNE excusesbidon
      ;;
      6)
      $FORTUNE magie
      ;;
      7)
      $FORTUNE amour
      ;;
      8)
      $FORTUNE littérature
      ;;
      9)
      $FORTUNE drogues
      ;;
      10)
      $FORTUNE éducation
      ;;
      0)
      echo "OK, au revoir!"
      break
      ;;
      *)
      echo "Ceci n'est pas un choix valide, taper un chiffre entre 0 et 10."
      ;;
esac
done
```

Mémorisez que **break** quitte la boucle, pas le script. Ceci se voit en ajoutant une commande **echo** à la fin du script. Cet **echo** sera aussi exécuté à la saisie qui provoque l'exécution du **break** (quand l'usager frappe « o »).

Dans les boucles imbriquées, **break** autorise la spécification de la boucle dont il faut sortir. Voir les pages Bash **info** pour plus de détails.

5.2. L'intégrée continue

L'instruction **continue** repart à l'itération d'une boucle **for**, **while**, **until** ou **select**.

Quand elle est utilisée dans une boucle **for** la variable de contrôle prend la valeur de l'élément suivant de la liste. Quand elle est utilisée dans une structure **while** ou **until** à contrario, l'exécution repart avec la première commande de **TEST-COMMAND** en haut de la boucle.

5.3. Exemples

Dans les exemples suivants, les noms de fichiers sont convertis en minuscule. Si la conversion n'est pas nécessaire, une instruction **continue** recommence l'exécution de la boucle. Ces commandes ne consomment pas trop de ressources système, et la plupart du temps, un effet similaire peut être obtenu avec **sed** et **awk**. Cependant, il est utile de connaître ces structures pour l'exécution de travaux coûteux, cela ne serait sans doute pas nécessaire si les tests étaient positionnés aux bons endroits dans le script, en partageant les ressources système.

```
[carol@octarine ~/test] cat tolower.sh
#!/bin/bash

# Ce script convertit tous les noms de fichiers contenant des majuscules en nom de fichier contenant qu

LIST="$(ls)"

for name in "$LIST"; do

if [[ "$name" != *[[:upper:]]* ]]; then
continue
fi

ORIG="$name"
```

```
NEW=`echo $name | tr 'A-Z' 'a-z'`

mv "$ORIG" "$NEW"
echo "nouveau nom pour $ORIG est $NEW"
done
```

Ce script a au moins un inconvénient : il écrase les fichiers existants. L'option `noclobber` de Bash est seulement utile quand intervient des redirections. L'option `-b` de la commande **mv** offre plus de sécurité, mais seulement dans le cas de réécriture accidentelle, comme le montre ce test :

```
[carol@octarine ~/test] rm *

[carol@octarine ~/test] touch test Test TEST

[carol@octarine ~/test] bash -x tolower.sh
++ ls
+ LIST=test
Test
TEST
+ [[ test != *[[:upper:]]* ]]
+ continue
+ [[ Test != *[[:upper:]]* ]]
+ ORIG=Test
++ echo Test
++ tr A-Z a-z
+ NEW=test
+ mv -b Test test
+ echo 'nouveau nom pour Test est test'
new name for Test is test
+ [[ TEST != *[[:upper:]]* ]]
+ ORIG=TEST
++ echo TEST
++ tr A-Z a-z
+ NEW=test
+ mv -b TEST test
+ echo 'nouveau nom pour TEST est test'
nouveau nom pour TEST est test

[carol@octarine ~/test] ls -a
./  ../  test  test~
```

tr fait parti du paquet *textutils*, il peut effectuer toute sorte de transformation de caractère.

6. Faire des menus avec l'intégrée select

6.1. Généralité

6.1.1. Emploi de select

La structure **select** permet la génération facile de menus. La syntaxe est assez similaire à celle de la boucle **for** :

```
select WORD [in LIST]; do RESPECTIVE-COMMANDS; done
```

`LIST` est interprété, ce qui génère une liste d'éléments. L'expansion est affichée sur le standard d'erreurs ; chaque élément est précédé d'un numéro. Si **in `LIST`** est absent, les paramètres positionnels sont affichés, comme si **in `$@`** avait été spécifié. `LIST` est affiché une fois seulement.

A l'affichage des tous les éléments, l'invite `PS3` est affichée et une ligne du standard d'entrée est lue. Si la ligne consiste en un nombre qui correspond à un des éléments, la valeur de `WORD` est définie avec le nom de cet élément. Si la ligne est vide, les éléments et l'invite `PS3` sont réaffichés. Si un caractère *EOF* (End Of File) est lu, la boucle se termine. Parce que la plupart des usagers n'ont pas idée de la combinaison de touches pour EOF, il est plus convivial d'ajouter une commande **break** comme l'un des éléments. Tout autre valeur issue de la ligne lue définira `WORD` comme une chaîne nulle.

La ligne lue est mémorisée dans la variable `REPLY`.

Les **RESPECTIVE-COMMANDS** sont exécutées après chaque choix valide jusqu'à ce que le nombre représentant le **break** soit lu. Ce qui fait quitter la boucle.

6.1.2. Exemples

Voici un exemple très simple, mais comme vous le constatez, il n'est pas très convivial :

```
[carol@octarine testdir] cat private.sh
#!/bin/bash

echo "Ce script peut mettre un accès privé à tout fichier de ce répertoire."
echo "Entrez le numéro du fichier que vous voulez protéger :"

select FILENAME in *;
do
     echo "Vous avez sélectionné $FILENAME ($REPLY), il est maintenant accessible que par vous."
     chmod go-rwx "$FILENAME"
done

[carol@octarine testdir] ./private.sh
Ce script peut mettre un accès privé à tout fichier de ce répertoire.
Entrez le numéro du fichier que vous voulez protéger :
1) archive-20030129
2) bash
3) private.sh
#? 1
Vous avez sélectionné archive-20030129 (1)
#?
```

Déclarer l'invite `PS3` et ajouter la possibilité de quitter l'améliore :

```
#!/bin/bash

echo "Ce script peut mettre un accès privé à tout fichier de ce répertoire."
echo "Entrez le numéro du fichier que vous voulez protéger"

PS3="Votre choix : "
QUIT="QUITTER CE PROGRAMME - Je me sens plus en confiance là."
touch "$QUIT"

select FILENAME in *;
do
  case $FILENAME in
        "$QUIT")
          echo "Fin."
          break
          ;;
        *)
          echo "Vous avez sélectionné $FILENAME ($REPLY)"
          chmod go-rwx "$FILENAME"
          ;;
  esac
done
rm "$QUIT"
```

6.2. Sous-menus

Toute instruction dans une structure **select** peut être un autre **select** ce qui autorise un(des) sous-menu(s) dans un menu.

Par défaut la variable `PS3` n'est pas changée dans une boucle **select** imbriquée. Si vous voulez une invite différente dans le sous-menu, assurez-vous de la définir aux bons moments.

7. L'intégrée shift

7.1. Qu'est-ce qu'elle fait ?

La commande **shift** est l'une des intégrées Bourne Shell qui est fournie par Bash. Cette commande prend un paramètre, un nombre. Les paramètres positionnels sont décalés sur la gauche le nombre de fois *N*. Les paramètres positionnels de `N+1` à `$#` sont renommés avec les noms de variable `$1` à `$# - N+1`.

Disons que nous avons une commande qui a `10` paramètres, et N vaut 4, alors `$4` devient `$1`, `$5` devient `$2` et ainsi de suite. `$10` devient `$7` et les anciens`$1`, `$2` et `$3` sont éliminés.

Si N est zéro ou supérieur à `$#` les paramètres positionnels ne sont pas changés (le nombre total de paramètres, voir Section 2.1.2, « Contrôle des paramètres de la ligne de commande ») et la commande n'a pas d'effet. Si N est absent, il est considéré valant `1`. Le code renvoyé est zéro à moins que N soit supérieur à `$#` ou inférieur à zéro, sinon il est différent de zéro.

7.2. Exemples

Une instruction shift est typiquement employée quand le nombre de paramètres d'une commande n'est pas connu par avance, par exemple quand l'usager peut donner autant de paramètres qu'il le souhaite. Dans de tels cas, les paramètres sont généralement traités dans une boucle **while** avec un test sur **(($#))**. Ce test est vrai tant que le nombre de paramètres est supérieur à zéro. La variable `$1` et l'instruction **shift** traite chaque paramètre. Le nombre de paramètres est diminué chaque fois que **shift** est exécutée et finalement devient zéro, sur quoi la boucle **while** s'arrête.

L'exemple suivant, `cleanup.sh`, emploie l'instruction **shift** pour traiter chaque fichier d'une liste générée par **find** :

```
#!/bin/bash

# Ce script peut éliminer des fichiers qui n'ont pas été accédés depuis plus de 365 jours.

USAGE="Utilisation : $0 dir1 dir2 dir3 ... dirN"

if [ "$#" == "0" ]; then
        echo "$USAGE"
        exit 1
fi

while (( "$#" )); do

if [[ $(ls "$1") == "" ]]; then
        echo "Répertoire vide, rien à faire."
  else
        find "$1" -type f -a -atime +365 -exec rm -i {} \;
fi

shift

done
```

-exec versus xargs

La commande **find** ci-dessus peut être remplacée par ce qui suit :

```
find options | xargs [commandes_a_executer_sur_fichiers_trouves]
```

La commande **xargs** construit et exécute des lignes de commandes depuis l'entrée standard. Ceci présente l'avantage que la ligne de commande est renseignée jusqu'à ce que la limite du système soit atteinte. Seulement à ce moment la commande à exécuter sera lancée, dans l'exemple ci-dessus ce serait **rm**. Si il y a plus de paramètres, une nouvelle ligne de commande sera utilisée, jusqu'à ce qu'elle soit elle aussi pleine ou jusqu'à ce qu'il n'y ait plus de paramètres. La même chose avec **find -exec** appelle la commande à exécuter sur chaque fichier trouvé. Donc, l'usage de **xargs** accélère grandement l'exécution des scripts et améliore les performances de votre machine.

Dans l'exemple suivant, nous avons modifié le script de la Section 2.4.4, « Les documents *intégrés* (NdT : here documents, que l'on appele aussi 'document lié') » afin qu'il accepte de multiples paquets à installer d'un coup :

```
#!/bin/bash
if [ $# -lt 1 ]; then
        echo "Utilisation : $0 package(s)"
        exit 1
fi
while (($#)); do
        yum install "$1" << CONFIRM
y
CONFIRM
shift
done
```

8. Résumé

Dans ce chapitre, nous avons vu comment les commandes répétitives peuvent être incorporées dans des structures de boucles. La plupart des boucles sont construites avec **for**, **while** ou **until** ou une combinaison de ces commandes. La boucle **for** exécute une tâche un nombre défini de fois. Si vous ne savez pas combien de fois une commande devrait s'exécuter, employez plutôt **until** ou **while** pour spécifier quand la boucle devrait s'arrêter.

Les boucles peuvent être interrompues ou réitérées avec **break** et **continue**.

Un fichier peut être utilisé comme entrée pour une boucle qui a un opérateur de redirection, les boucles peuvent aussi lire les résultats de commandes qui sont fournis à la boucle par un tube.

La structure **select** est employée pour afficher des menus dans les scripts interactifs. Le décalage des paramètres d'une ligne de commande peut être fait avec **shift**.

9. Exercices

Rappelez-vous : quand vous écrivez des scripts, travaillez par étapes et les tester avant de les incorporer à votre script.

1. Créer un script qui fera une copie (récursive) des fichiers dans `/etc` de sorte qu'un administrateur système débutant puisse les éditer sans crainte.

2. Ecrire un script qui prendra exactement un paramètre, le nom d'un répertoire. Si le nombre de paramètres est supérieur ou inférieur à 1, afficher le message d'utilisation. Si l'argument n'est pas un répertoire, afficher un autre message. Pour le répertoire donné, afficher les 5 fichiers les plus gros et les 5 fichiers le plus récemment modifiés.

3. Pouvez-vous expliquer pourquoi il est si important de mettre les variables entre guillemets dans l'exemple de la Section 4.2, « Redirection des sorties » ?

4. Ecrivez un script similaire à celui de la Section 5.1, « L'intégrée break », mais pensez à un moyen de quitter après que l'usager ait effectué 3 boucles.

5. Penser à une meilleur solution que **move `-b`** pour le script de la Section 5.3, « Exemples » pour éviter d'écraser les fichiers existants. Par exemple, tester si un fichier existe ou pas. Ne faites pas de travail inutile!

6. Réécrire le script `whichdaemon.sh` de la Section 2.4, « Opérations booléennes », de sorte qu'il :

    - Affiche une liste des serveurs à contrôler, tel que Apache, le serveur SSH, le démon NTP, un démon nom, un démon d'administration, etc.

    - Pour chaque choix l'utilisateur peut afficher des informations importantes, telles que le nom du serveur WEB, les informations de trace NTP, etc.

    - Optionnellement, prévoir une possibilité pour les utilisateurs de contrôler d'autres serveurs que ceux listés. Dans de tels cas, vérifiez que au moins le processus en question tourne.

    - Revoir les scripts de la Section 2.2.4, « Calcul d'une moyenne ». Remarquez comment les caractères entrés autre que **q** sont traités. Réécrire ce script afin qu'il affiche un message si des caractères sont saisis.

Chapitre 10. Un peu plus sur les variables

Table des matières

1. Types de variables
    1.1. Affectation générale de valeur.
    1.2. Utiliser l'intégrée declare
    1.3. Constantes

2. Variables tableau
    2.1. Créer des tableaux
    2.2. Invoquer les variables d'un tableau
    2.3. Supprimer des variables tableau
    2.4. Exemples de tableaux
3. Opérations sur les variables
    3.1. Arithmétique sur les variables
    3.2. Longueur de variable
    3.3. Transformation de variables
4. Résumé
5. Exercices

Résumé

Dans ce chapitre, nous aborderons l'emploi plus poussé des variables et paramètres. Une fois achevé, vous serez capable de :

- Déclarer et utiliser un tableau de variables
- Spécifier le type de variable que vous voulez utiliser
- Rendre les variables en lecture seule
- Employer **set** pour affecter une valeur à une variable

1. Types de variables

1.1. Affectation générale de valeur.

Comme nous l'avons déjà vu, Bash comprend plusieurs types de variables ou paramètres. Jusqu'à maintenant, nous ne nous sommes pas inquiété du type de variable affectée, de sorte que nos variables pouvaient stocker toute sorte de valeur que nous leur affections. Une simple ligne de commande illustre ceci :

```
[bob in ~] VARIABLE=12

[bob in ~] echo $VARIABLE
12

[bob in ~] VARIABLE=string

[bob in ~] echo $VARIABLE
string
```

Il y a des cas où vous voulez éviter ce genre de comportement, par exemple quand vous manipulez des numéros de téléphone et autres codifications. A part les entiers et les variables, vous pourriez aussi vouloir spécifier une variable avec une valeur constante. Ceci est souvent fait au début du script, quand la valeur de la constante est définie. Ensuite, il est seulement fait référence au nom de la variable stockant la constante, de sorte que quand la logique veut que la constante soit changée, cela n'est fait qu'à un endroit. Une variable peut aussi être une série de variables de tout type, c'est à dire un *tableau* de variables (`VAR0VAR1`, `VAR2`, ... `VARN`).

1.2. Utiliser l'intégrée declare

Avec l'instruction **declare** nous pouvons encadrer l'affectation de valeur à une variable.

La syntaxe de **declare** est la suivante :

```
declare OPTION(s) VARIABLE=value
```

Les options suivantes sont employées pour déterminer le type de donnée de la variable et pour lui assigner des attributs :

Tableau 10.1. Options de l'intégrée declare

Option	sens
`-a`	Variable tableau.
`-f`	Utilise uniquement les noms de fonction
`-i`	La variable doit être considérée en tant qu'entier, une évaluation arithmétique est effectuée quand une valeur est assignée à la variable (voir Section 4.6, « L'expansion arithmétique »).
`-p`	Affiche les attributs et la valeur de chaque variable. Quand `-p` est employé, les options supplémentaires sont ignorées.
`-r`	fait que la variable est en lecture seule. A cette variable ne peut alors lui être affecté une autre valeur par une instruction ultérieure, de même qu'elle ne peut être supprimée.
`-t`	Donne à chaque variable l'attribut *trace*.
`-x`	Marque chaque variable comme exportée pour les commandes suivantes via l'environnement.

L'emploi de + au lieu de - inhibe les attributs. Quand c'est employé dans une fonction, **declare** créé des variables locales.

L'exemple suivant montre comment l'assignation du type de variable influence la valeur.

```
[bob in ~] declare -i VARIABLE=12

[bob in ~] VARIABLE=string

[bob in ~] echo $VARIABLE
0

[bob in ~] declare -p VARIABLE
declare -i VARIABLE="0"
```

Notez que Bash a une option pour déclarer une valeur numérique, mais aucune pour une valeur chaîne. Ceci s'explique puisque, par défaut, si aucune spécification n'est indiquée, une variable peut stocker tout type de donnée :

```
[bob in ~] OTHERVAR=blah

[bob in ~] declare -p OTHERVAR
declare -- OTHERVAR="blah"
```

Dès que vous restreignez l'affectation de valeurs à une variable, elle ne peut que contenir ce type de donnée. Les restrictions possibles sont soit entier, constante, ou tableau.

Voir les pages info de Bash pour une aide sur le statut renvoyé.

1.3. Constantes

En Bash, les constantes sont créées en mettant en lecture seule une variable. L'intégré **readonly** marque chaque variable spécifiée comme non modifiable.. La syntaxe est :

```
readonly OPTION VARIABLE(s)
```

La valeur de ces variables ne peut plus être changée par une instruction ultérieure. Si l'option `-f` est donnée, chaque variable réfère à une fonction Shell ; voir Chapitre 11, *Fonctions*. If `-a` est spécifié, chaque variable réfère à un tableau de variables. Si aucun argument n'est donné, ou si `-p` est indiqué, une liste de toutes les variables en lecture est affichée. Avec l'option `-p` le résultat peut être réutilisé

comme entrée.

Le statut d'exécution est zéro, à moins qu'une option invalide ait été spécifiée, qu'une des variables ou fonctions n'existe pas, ou que `-f` ait été fourni en tant que nom de variable au lieu d'un nom de fonction.

```
[bob in ~] readonly TUX=penguinpower

[bob in ~] TUX=Mickeysoft
bash: TUX: readonly variable
```

2. Variables tableau

2.1. Créer des tableaux

Un tableau est une variable contenant plusieurs valeurs. Toute variable peut être utilisée comme étant un tableau. Il n'y a pas de limite maximum à la taille d'un tableau, ni de besoin que les éléments soient indexés ou assignés de façon contiguë. Les tableaux démarrent à zéro : le premier élément est donc adressé avec le numéro 0.

Une déclaration indirecte peut se faire avec la syntaxe suivante de déclaration de variable :

```
ARRAY[INDEXNR]=value
```

Le *INDEXNR* est traité comme une expression arithmétique qui doit être évalué comme nombre positif.

Une déclaration explicite d'un tableau est faite avec l'intégrée **declare** :

```
declare -a ARRAYNAME
```

Une déclaration avec un numéro d'index sera aussi acceptée, mais le numéro d'index sera ignoré. Des attributs du tableau peuvent être spécifiés en employant les intégrées **declare** et **readonly**. Les attributs s'appliquent à toutes les variables du tableau ; vous ne pouvez avoir des tableaux mitigés.

Les variables de tableau peuvent aussi être créées avec une affectation composée selon ce format :

```
ARRAY=(value1 value2 ... valueN)
```

Chaque valeur est alors de la forme *[indexnumber=]string*. Le numéro d'index est optionnel. Si il est fourni, l'index prend la valeur du numéro ; sinon l'index de l'élément affecté est le numéro du dernier index assigné, plus un. Le format est accepté par **declare** également. Si aucun numéro d'index n'est fourni, l'indexation commence à zéro.

Ajouter un élément manquant ou suplémentaire à un tableau se fait avec la syntaxe :

```
ARRAYNAME[indexnumber]=value
```

Se rappeler que l'intégrée **read** possède l'option `-a` qui autorise la lecture et l'affectation de valeurs des éléments d'un tableau.

2.2. Invoquer les variables d'un tableau

Afin de se référer au contenu d'un élément du tableau, employer le symbole accolade. C'est nécessaire, comme vous le voyez dans l'exemple suivant, pour échapper à l'interprétation du Shell des opérateurs d'expansion. Si le numéro d'index est @ ou *, tous les éléments du tableau sont considérés.

```
[bob in ~] ARRAY=(one two three)

[bob in ~] echo ${ARRAY[*]}
one two three

[bob in ~] echo $ARRAY[*]
```

```
one[*]

[bob in ~] echo ${ARRAY[2]}
three

[bob in ~] ARRAY[3]=four

[bob in ~] echo ${ARRAY[*]}
one two three four
```

Se référer au contenu d'un élément de tableau sans indiquer le numéro d'index est équivalent à se référer au contenu du premier élément, celui d'index zéro.

2.3. Supprimer des variables tableau

L'intégrée **unset** est employée pour détruire un tableau ou des variables éléments du tableau :

```
[bob in ~] unset ARRAY[1]

[bob in ~] echo ${ARRAY[*]}
one three four

[bob in ~] unset ARRAY

[bob in ~] echo ${ARRAY[*]}
<--no output-->
```

2.4. Exemples de tableaux

Exemples pratiques de la manipulation de tableaux sont difficiles à trouver. Vous trouverez plein de scripts qui ne font pas autre chose sur votre système que de calculer des séries mathématiques avec des tableaux, par exemple. Et ça devrait être l'un des exemples les plus intéressants...la plupart des scripts ne font que montrer d'une façon hyper simplifiée et théorique ce que vous pouvez faire avec les tableaux.

La raison de cette fadeur tient en ce que les tableaux sont des structures plutôt complexes. Vous verrez que les exemples les plus pratiques pour lesquels les tableaux peuvent être utilisés sont déjà mis en oeuvre sur votre système, mais à plus bas niveau, en langage C dans lequel la plupart des commandes UNIX sont écrites. Un bon exemple est la commande intégrée Bash **history**. Les lecteurs intéressés peuvent voir le répertoire `built-ins` dans l'arbre des sources Bash et jeter un coup d'oeil à `fc.def`, qui est exécutée à la compilation des intégrées.

Une autre raison pour laquelle de bons exemples sont difficiles à trouver est que tous les Shell ne reconnaissent pas les tableaux, ce qui gêne la compatibilité.

Après des jours de recherche, j'ai finalement trouvé cet exemple qui s'exécute chez un fournisseur INTERNET. Il distribue des fichiers de configuration de serveur WEB Apache sur des hôtes dans une ferme WEB :

```
#!/bin/bash

if [ $(whoami) != 'root' ]; then
        echo "Must be root to run $0"
        exit 1;
fi
if [ -z $1 ]; then
        echo "Utilisation : $0 </path/to/httpd.conf>"
        exit 1
fi

httpd_conf_new=$1
httpd_conf_path="/usr/local/apache/conf"
login=htuser

farm_hosts=(web03 web04 web05 web06 web07)

for i in ${farm_hosts[@]}; do
        su $login -c "scp $httpd_conf_new ${i}:${httpd_conf_path}"
        su $login -c "ssh $i sudo /usr/local/apache/bin/apachectl graceful"

done
exit 0
```

D'abord 2 tests sont effectués pour contrôler que l'utilisateur idoïne fait s'exécuter ce script avec les paramètres ad hoc. Les noms des hôtes qui doivent être configurés sont listés dans le tableau

`farm_hosts`. Puis tous ces hôtes sont chargés avec le fichier de configuration Apache, après quoi le démon est redémarré. Notez l'emploi de commandes de la suite Secure Shell, qui encryptent les connections aux hôtes distants.

Merci, Eugène et ses collègues, pour cette contribution.

Dan Richter a fourni l'exemple suivant. Voici le problème auquel il était confronté :

« ...Dans mon entreprise, nous avons des démonstrations sur notre site WEB, et chaque semaine quelqu'un est chargé de les tester toutes. Donc j'ai un travail cron qui remplit un tableau avec les candidats possibles, qui utilise **date +%W** pour déterminer la semaine dans l'année, et fait une opération modulo pour trouver le bon index. La personne gâtée reçoit un courriel. »

Et voici la solution :

```
#!/bin/bash
# Ceci est le script : get-tester-address.sh
#
# D'abord nous testons si Bash admet les tableaux..
# (Les tableaux ont été ajoutés récemment.)
#
whotest[0]='test' || (echo 'Echec : les tableaux ne sont pas admis dans cette version de Bash.' && exit

#
# Our list of candidates. (Vous êtes libre d'ajouter
# ou d'enlever des candidats.)
#
wholist=(
      'Bob Smith <bob@example.com>'
      'Jane L. Williams <jane@example.com>'
      'Eric S. Raymond <esr@example.com>'
      'Larry Wall <wall@example.com>'
      'Linus Torvalds <linus@example.com>'
   )
#
# Compte le nombre de testeurs candidats.
# (Boucle jusqu'à trouver une chaîne vide.)
#
count=0
while [ "x${wholist[count]}" != "x" ]
do
   count=$(( $count + 1 ))
done

#
# Maintenant nous calculons à qui c'est le tour.
#
week=`date '+%W'`       # La semaine dans l'année (0..53).
week=${week#0}          # Elimine de possible zéro au début.

let "index = $week % $count"   # week modulo count = la personne gâtée

email=${wholist[index]}     # Récupérer l'adresse email de cette personne.

echo $email             # Affiche l'adresse email.
```

Ce script est alors appelé dans d'autres scripts, tel que celui-ci, qui utilise un document *intégré* :

```
email=`get-tester-address.sh`   # Trouver à qui envoyer le courriel.
hostname=`hostname`             # Le nom de la machine locale.

#
# Envoyer le courriel à la bonne personne.
#
mail $email -s '[Demo Testing]' <<EOF
La personne gâtée de la semaine est : $email

Rappel : la liste de démonstrations est ici :
    http://web.example.com:8080/DemoSites

(Ce courriel a été généré par $0 depuis ${hostname}.)
EOF
```

3. Opérations sur les variables

3.1. Arithmétique sur les variables

Nous avons déjà abordé la question à la Section 4.6, « L'expansion arithmétique ».

3.2. Longueur de variable

La syntaxe **${#VAR}** calcul le nombre de caractères d'une variable. Si VAR vaut « * » ou « @ », cette valeur est remplacée par le nombre de paramètres positionnels ou le nombre d'éléments dans le tableau en général. En voici une démonstration ci-dessous :

```
[bob in ~] echo $SHELL
/bin/bash

[bob in ~] echo ${#SHELL}
9

[bob in ~] ARRAY=(one two three)

[bob in ~] echo ${#ARRAY}
3
```

3.3. Transformation de variables

3.3.1. Substitution

```
${VAR:-WORD}
```

Si VAR n'est pas défini ou est nul, l'expansion de *WORD* est employée ; sinon la valeur de VAR est remplacée :

```
[bob in ~] echo ${TEST:-test}
test

[bob in ~] echo $TEST


[bob in ~] export TEST=a_string

[bob in ~] echo ${TEST:-test}
a_string

[bob in ~] echo ${TEST2:-$TEST}
a_string
```

Cette forme est souvent employée dans les tests conditionnels, par exemple dans celui-ci :

```
[ -z "${COLUMNS:-}" ] && COLUMNS=80
```

C'est une notation plus courte pour

```
if [ -z "${COLUMNS:-}" ]; then
        COLUMNS=80
fi
```

Voir la Section 1.2.3, « Comparaisons de chaînes » pour plus de détails au sujet de ce type de test de condition.

Si le tiret (-) est remplacé par le signe égal (=), la valeur est affectée au paramètre si il n'existe pas :

```
[bob in ~] echo $TEST2


[bob in ~] echo ${TEST2:=$TEST}
a_string

[bob in ~] echo $TEST2
a_string
```

La syntaxe suivante teste l'existence d'une variable. Si elle n'est pas déclarée, l'expansion de *WORD* est affichée sur le standard de résultat et un Shell non-interactif se termine. Une démonstration :

```
[bob in ~] cat vartest.sh
#!/bin/bash

# Ce script teste si une variable est déclarée.  Si non,
# Il quitte en affichant un message.

echo ${TESTVAR:?"Il y a tellement encore que je voudrais faire..."}
```

```
echo "TESTVAR est déclarée, nous pouvons traiter."

[bob in testdir] ./vartest.sh
./vartest.sh: line 6: TESTVAR: Il y a tellement encore que je voudrais faire...

[bob in testdir] export TESTVAR=present

[bob in testdir] ./vartest.sh
present
TESTVAR est déclarée, nous pouvons traiter.
```

Avec « + » au lieu du point d'exclamation la variable prend la valeur de l'expansion de *WORD* ; si elle n'existe pas, rien ne se produit.

3.3.2. Suppression de sous-chaînes

Pour éliminer d'une variable un nombre de caractères égal à *OFFSET*, la syntaxe à employer est :

```
${VAR:OFFSET:LENGTH}
```

Le paramètre *LENGTH* définit combien de caractères garder, à partir du premier caractère après le décalage. Si *LENGTH* est omis, le reste du contenu de la variable est conservé :

```
[bob in ~] export STRING="thisisaverylongname"

[bob in ~] echo ${STRING:4}
isaverylongname

[bob in ~] echo ${STRING:6:5}
avery
```

```
${VAR#WORD}
```

et

```
${VAR##WORD}
```

Cette syntaxe est employée pour éliminer les correspondances du patron donné par l'expansion de *WORD* de VAR. *WORD* est interprété pour donner un patron tout comme dans l'expansion de nom de fichier. Si le patron correspond au début du résultat d'expansion de VAR, alors le résultat est la valeur de VAR réduit au plus court patron correspondant (« # ») ou le plus long (quand employé avec « ## »).

Si VAR est * ou @, l'opération de suppression du patron est effectuée sur chaque paramètre positionnel, et l'expansion est la liste résultante.

Si VAR est une variable tableau indexée par « * » ou « @ », l'opération de substitution de patron est effectuée pour chaque élément du tableau l'un après l'autre, et l'expansion est la liste résultante. Ceci est montré dans l'exemple ci-dessous :

```
[bob in ~] echo ${ARRAY[*]}
one two one three one four

[bob in ~] echo ${ARRAY[*]#one}
two three four

[bob in ~] echo ${ARRAY[*]#t}
one wo one hree one four

[bob in ~] echo ${ARRAY[*]#t*}
one wo one hree one four

[bob in ~] echo ${ARRAY[*]##t*}
one one one four
```

L'effet opposé est obtenu avec « % » et « %% », comme dans l'exemple suivant. *WORD* devrait correspondre à une portion en fin de chaîne :

```
[bob in ~] echo $STRING
thisisaverylongname

[bob in ~] echo ${STRING%name}
thisisaverylong
```

3.3.3. Remplacer des parties de noms de variables

Ceci est fait avec la syntaxe suivante

`${VAR/PATRON/CHAINE}`

ou l'option

`${VAR//PATRON/CHAINE}`

syntax. La première forme remplace seulement la première correspondance, la seconde remplace toutes les occurrences de *PATRON* par *CHAINE* :

```
[bob in ~] echo ${STRING/name/string}
thisisaverylongstring
```

Vous trouverez plus de détails dans les pages info de Bash.

4. Résumé

Normalement, une variable peut stocker tout type de donnée, à moins qu'elles soient déclarées explicitement. Les variables constantes sont déclarées avec la commande intégrée **readonly**.

Un tableau stocke un ensemble de variables. Si un type de donnée est déclaré, alors tous les éléments du tableau seront considérés comme contenant seulement ce type de donnée.

Les fonctionnalités du Bash permettent la substitution et la transformation de variables « au vol ». Les opérations standards incluent le calcul de la longueur de la variable, l'arithmétique de variables, la substitution du contenu - ou d'une partie - d'une variable.

5. Exercices

Voici quelques casse-têtes :

1. Ecrire un script qui fait ce qui suit :
    - Affiche le nom du script qui s'exécute.
    - Afficher le premier, le troisième et le dixième paramètre donné au script.
    - Afficher le nombre total de paramètres du script.
    - Si il y a plus de 3 paramètres positionnels, employez **shift** pour décaler toutes les valeurs de 3 places vers la gauche.
    - Afficher toutes les valeurs des paramètres restants.
    - Affiche le nombre de paramètres.

   Tester avec zéro, un, trois et plus de dix paramètres.
2. Ecrire un script qui installe un navigateur WEB simple (en mode texte), avec **wget** et **links `-dump`** pour afficher les pages HTML à l'intention de l'usager. L'usager a 3 choix : entrer une URL,entrer **b** pour boucler, **q** pour quitter. Les 10 dernières URL entrées par l'usager sont stockées dans un tableau, duquel l'usager peut récupérer une URL avec la fonctionnalité *boucler*.

Chapitre 11. Fonctions

Table des matières

1. Introduction

1.1. Qu'est-ce qu'une fonction ?
1.2. La syntaxe des fonctions
1.3. Les paramètres positionnels dans les fonctions
1.4. Afficher une fonction
2. Exemples de fonctions dans des scripts
2.1. Recyclage
2.2. Définir le chemin
2.3. Sauvegarde à distance
3. Résumé
4. Exercices

Résumé

Dans ce chapitre nous aborderons :

- Qu'est-ce qu'une fonction
- Création et affichage de fonctions depuis la ligne de commande
- Fonctions dans les scripts
- Passer des arguments à une fonction
- Quand utiliser une fonction

1. Introduction

1.1. Qu'est-ce qu'une fonction ?

La fonction Shell est le moyen de grouper des commandes en vue d'une exécution ultérieure, par l'appel d'un nom donné à ce groupe, ou *routine*. Le nom de la routine doit être unique dans le Shell ou le script. Toutes les commandes qui constituent une fonction sont exécutées comme des commandes régulières. Quand une fonction est appelée comme le nom d'une simple commande, la liste des commandes associées à ce nom de fonction est traitée. Une fonction est exécutée à l'intérieur du Shell dans lequel elle a été déclarée : aucun processus nouveau n'est créé pour interpréter les commandes.

Les commandes intégrées spéciales sont détectées avant les fonctions Shell pendant l'analyse des commandes. Les intégrées spéciales sont : **break**, **:**, **.**, **continue**, **eval**, **exec**, **exit**, **export**, **readonly**, **return**, **set**, **shift**, **trap** et **unset**.

1.2. La syntaxe des fonctions

Les fonctions emploient plutôt la syntaxe

```
function FONCTION { COMMANDES; }
```

ou l'option

```
FONCTION () { COMMANDES; }
```

Chacune définit une fonction Shell **FONCTION**. L'emploi de la commande intégrée **function** est optionnel ; cependant, si elle n'est pas employée, les parenthèses sont nécessaires.

Les commandes listées entre les accolades forment le corps de la fonction. Ces commandes sont exécutées du moment que **FONCTION** est spécifié en tant que nom de commande. Le statut d'exécution est celui de la dernière commande exécutée dans le corps de la fonction.

Erreurs communes

Les accolades doivent être séparées du corps de la fonction par un espace, sinon elles sont interprétées d'une mauvaise façon.

Le corps d'une fonction doit se terminer par un point-virgule ou un saut de ligne.

1.3. Les paramètres positionnels dans les fonctions

Les fonctions sont comme des mini-scripts : elles peuvent accepter des paramètres, elles peuvent utiliser des variables connues seulement dans la fonction (avec l'intégrée Shell **local**) et elles peuvent renvoyer des valeurs au Shell appelant.

Une fonction a aussi un système pour interpréter des paramètres positionnels. Cependant, les paramètres positionnels passés à une fonction n'ont pas nécessairement les mêmes valeurs que ceux passés à une commande ou un script.

Quand une fonction est exécutée, les arguments de la fonction deviennent les paramètres positionnels le temps de l'exécution. Le paramètre spécial `#` qui est remplacé par le nombre de paramètres positionnels est modifié en conséquence. Le paramètre positionnel `0` est inchangé. La variable Bash `FUNCNAME` est valorisé avec le nom de la fonction, tandis qu'elle s'exécute.

Si l'intégrée **return** est exécutée dans une fonction, la fonction s'interrompt et l'exécution reprend avec la commande qui suit la fonction appelée. Quand une fonction s'achève, les valeurs des paramètres positionnels et le paramètre spécial `#` sont restaurés à la valeur qu'ils avaient avant l'exécution de la fonction. Si un argument numérique est donné à **return**, c'est ce statut qui est retourné. Un exemple simple :

```
[lydia@cointreau ~/test] cat showparams.sh
#!/bin/bash

echo "Ce script montre l'emploi d'arguments de fonction."
echo

echo "Le paramètre positionnel 1 pour le script est $1."
echo

test ()
{
echo "Le paramètre positionnel 1 pour la fonction est $1."
RETURN_VALUE=$?
echo "Le code retour de cette fonction est $RETURN_VALUE."
}

test other_param

[lydia@cointreau ~/test] ./showparams.sh parameter1
Ce script montre l'emploi d'arguments de fonction.

Le paramètre positionnel 1 pour le script est 1.

Le paramètre positionnel 1 pour la fonction est other_param.
Le code retour de cette fonction est 0.

[lydia@cointreau ~/test]
```

Notez que la valeur retournée ou code retour de la fonction est souvent stockée dans une variable, afin qu'elle puisse être testée ultérieurement. Les scripts d'initialisation de votre système souvent emploient la technique de tester la variable `RETVAL`, comme ceci :

```
if [ $RETVAL -eq 0 ]; then
        <lancer le démon>
```

Ou comme cet exemple tiré du script `/etc/init.d/amd`, où l'optimisation de Bash est mis en oeuvre.

```
[ $RETVAL = 0 ] && touch /var/lock/subsys/amd
```

Les commandes après **&&** ne sont exécutées que si le test rend vrai ; c'est une façon plus rapide de

représenter une structure **if/then/fi**.

Le code retour de la fonction est souvent utilisé comme statut d'exécution de tout le script. Vous verrez beaucoup de scripts d'initialisation finissant avec quelque chose comme ça **exit `$RETVAL`**.

1.4. Afficher une fonction

Toutes les fonctions connues du Shell courant peuvent être affichées avec l'intégrée **set** sans options. Une fonction est conservée après avoir été appelée, à moins qu'elle soit **unset** après son exécution. La commande **which** affiche aussi les fonctions :

```
[lydia@cointreau ~] which zless
zless is a function
zless ()
{
    zcat "$@" | "$PAGER"
}

[lydia@cointreau ~] echo $PAGER
less
```

Ceci est le type de fonctions qui sont typiquement configurées dans un fichier de configuration des ressources Shell de l'utilisateur. Les fonctions sont plus flexibles que les alias et fournissent un moyen simple et facile d'adapter l'environnement utilisateur.

En voici un pour les utilisateurs DOS :

```
dir ()
{
    ls -F --color=auto -lF --color=always "$@" | less -r
}
```

2. Exemples de fonctions dans des scripts

2.1. Recyclage

Il y a plein de scripts sur votre système qui utilisent des fonctions comme un moyen structuré de passer une série de commandes. Sur certains systèmes Linux, par exemple, vous trouverez le fichier de définition `/etc/rc.d/init.d/functions`, qui est invoqué dans tous les scripts d'initialisation. Avec cette méthode, les tâches communes comme contrôler qu'un processus s'exécute, démarrer ou arrêter un démon etc., n'ont qu'à être écrites qu'une seule fois, d'une manière générique. Si la même tâche est nécessaire de nouveau, le code est recyclé.

Vous pourriez faire votre propre fichier `/etc/functions` qui contiendrait toutes les fonctions que vous utilisez régulièrement, dans différents scripts. Entrez simplement la ligne

```
. /etc/functions
```

quelque part au début du script et vous pouvez recycler les fonctions.

2.2. Définir le chemin

Le code ci-dessous peut être trouvé dans le fichier `/etc/profile`. La fonction **pathmunge** est définie, puis utilisée pour définir les chemins de *root* et des autres utilisateurs :

```
pathmunge () {
        if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
           if [ "$2" = "after" ] ; then
              PATH=$PATH:$1
           else
              PATH=$1:$PATH
           fi
        fi
}

# Path manipulation
if [ `id -u` = 0 ]; then
        pathmunge /sbin
```

```
        pathmunge /usr/sbin
        pathmunge /usr/local/sbin
fi

pathmunge /usr/X11R6/bin after

unset pathmunge
```

La fonction considère son premier argument comme étant un nom de chemin. Si ce nom de chemin n'est pas encore dans le chemin courant, il y est ajouté. Le second argument de cette fonction définit si le chemin sera ajouté au début ou à la fin de la définition actuelle du PATH.

L'utilisateur standard se voit ajouter seulement `/usr/X11R6/bin` dans leurs chemins, alors que *root* se voit adjoindre quelques répertoires supplémentaires contenant les commandes systèmes. Après avoir été utilisée, la fonction est supprimée par unset.

2.3. Sauvegarde à distance

L'exemple suivant est l'un de ceux que j'utilise pour faire mes sauvegardes des fichiers de mes livres. Il emploie des clés SSH pour effectuer la connection à distance. Deux fonctions sont définies, **buplinux** et **bupbash**, qui produisent chacune un fichier `.tar`, qui est alors compressé et envoyé vers le serveur distant. Ensuite, la copie locale est supprimée.

Le dimanche, seul **bupbash** est exécuté.

```
#/bin/bash

LOGFILE="/nethome/tille/log/backupscript.log"
echo "Starting backups for `date`" >> "$LOGFILE"

buplinux()
{
DIR="/nethome/tille/xml/db/linux-basics/"
TAR="Linux.tar"
BZIP="$TAR.bz2"
SERVER="rincewind"
RDIR="/var/www/intra/tille/html/training/"

cd "$DIR"
tar cf "$TAR" src/*.xml src/images/*.png src/images/*.eps
echo "Compressing $TAR..." >> "$LOGFILE"
bzip2 "$TAR"
echo "...done." >> "$LOGFILE"
echo "Copying to $SERVER..." >> "$LOGFILE"
scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
echo "...done." >> "$LOGFILE"
echo -e "Done backing up Linux course:\nSource files, PNG and EPS images.\nRubbish removed." >> "$LOGFI
rm "$BZIP"
}

bupbash()
{
DIR="/nethome/tille/xml/db/"
TAR="Bash.tar"
BZIP="$TAR.bz2"
FILES="bash-programming/"
SERVER="rincewind"
RDIR="/var/www/intra/tille/html/training/"

cd "$DIR"
tar cf "$TAR" "$FILES"
echo "Compressing $TAR..." >> "$LOGFILE"
bzip2 "$TAR"
echo "...done." >> "$LOGFILE"
echo "Copying to $SERVER..." >> "$LOGFILE"
scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
echo "...done." >> "$LOGFILE"

echo -e "Done backing up Bash course:\n$FILES\nRubbish removed." >> "$LOGFILE"
rm "$BZIP"
}

DAY=`date +%w`

if [ "$DAY" -lt "2" ]; then
  echo "It is `date +%A`, only backing up Bash course." >> "$LOGFILE"
  bupbash
else
  buplinux
  bupbash
fi


echo -e "Remote backup `date` SUCCESS\n----------" >> "$LOGFILE"
```

Ce script est lancé par cron, c'est à dire sans intervention de l'utilisateur, c'est pour ça que le standard d'erreurs de la commande **scp** est redirigé sur `/dev/null`.

Il pourrait être observé que toutes les étapes peuvent être combinées en une commande telle que

```
tar c dir_to_backup/ | bzip2 | ssh server "cat > backup.tar.bz2"
```

Cependant, si vous êtes intéressé par les résultats intermédiaires, qui pourrait être récupérés en cas d'échec du script, ce n'est pas ce qu'il faut écrire.

L'expression

```
command &> file
```

est équivalent à

```
command > file 2>&1
```

3. Résumé

Les fonctions fournissent un moyen facile de grouper des commandes que vous avez besoin d'exécuter régulièrement. Quand une fonction tourne, les paramètres positionnels sont ceux de la fonction. Quand elles s'arrêtent, on retrouve ceux du programme appelant. Les fonctions sont comme des mini-scripts, et comme un script, elles génèrent un code retour.

Bien que ce chapitre soit court, il contient des connaissances importantes nécessaires pour atteindre le stade suprême de la paresse, ce qui est le but recherché de tout administrateur système.

4. Exercices

Voici quelques tâches utiles que vous pouvez réalisez avec des fonctions.

1. Ajoutez une fonction à votre fichier de configuration `~/.bashrc` qui automatise l'impression des pages man. L'effet devrait être que quand vous taper **printman <commande>** les pages man de la commande sortent de l'imprimante. Contrôler le fonctionnement avec une pseudo imprimante.

   En plus, imaginez la possibilité pour l'usager de demander un numéro de section des pages man.

2. Créer un sous-répertoire dans votre répertoire racine dans lequel vous pouvez stocker des définitions de fonctions. Y mettre quelques fonctions. Des fonctions utiles peuvent être, parmi d'autres, celles qui permettent les mêmes commandes que DOS ou un UNIX commercial, ou vice versa. Ces fonctions devraient être importées dans votre environnment Shell quand `~/.bashrc` est lu.

Chapitre 12. Trapper les signaux

Table des matières

1. Signaux
   - 1.1. Introduction
   - 1.2. Utilisation de signaux avec kill
2. Piéger les signaux
   - 2.1. Généralité
   - 2.2. Comment Bash interprète trap
   - 2.3. Plus d'exemples
3. Résumé
4. Exercices

Résumé

Dans ce chapitre nous traiterons les sujets suivants :

- Signaux disponibles
- Intérêt des signaux
- Emploi de l'instruction **trap**
- Comment éviter que les usagers interrompent votre programme

1. Signaux

1.1. Introduction

1.1.1. Trouver la page man de signal

Votre système contient une page man qui liste tous les signaux disponibles, mais selon votre système, elle peut être affichée de diverses façons. Sur la plupart des Linux, ce sera avec **man 7 signal**. Dans le doute, localisez la page man exacte ainsi que la section avec une commande comme

```
man -k signal | grep list
```

ou l'option

```
apropos signal | grep list
```

Les noms de signaux peuvent être trouvés avec **kill -l**.

1.1.2. Les signaux en direction de votre Shell Bash

En l'absence de toute trappe, un Shell Bash interactif ignore *SIGTERM* et *SIGQUIT*. *SIGINT* est récupéré et considéré, et si le contrôle de travaux est actif, *SIGTTIN*, *SIGTTOU* et *SIGTSTP* sont aussi ignorés. Les commandes qui sont lancées en tant que résultat de substitution de commandes ignorent aussi ces signaux quand le clavier les a générés.

SIGHUP par défaut fait quitter le Shell. Un Shell interactif enverra un *SIGHUP* à tous les travaux, en exécution ou pas ; voir la documentation sur l'intégrée **disown** si vous souhaitez désactiver ce comportement pour un processus particulier. Utilisez l'option `huponexit` pour tuer tous les travaux à la réception du signal *SIGHUP* avec l'intégrée **shopt**.

1.1.3. Envoyer des signaux avec le Shell

Les signaux suivants peuvent être envoyés en utilisant le Shell Bash :

Tableau 12.1. Les signaux de contrôle dans Bash

Combinaison standard de touches	sens
Ctrl+C	Le signal d'interruption, envoie SIGINT à tous les travaux s'exécutant dans le Shell courant.
Ctrl+Y	Le caractère *delayed suspend*. Provoque la suspension d'un processus actif quand il tente de lire son entrée depuis le terminal. Le contrôle est rendu au Shell, l'utilisateur peut renvoyer en tâche de fond, réactiver ou tuer le processus. 'Delayed suspend' n'est pas une fonctionnalité connue de tous les systèmes.

Combinaison standard de touches	sens
Ctrl+**Z**	Le signal *suspend* envoie *SIGTSTP* à un programme en train de s'exécuter, donc il s'arrête et redonne le contrôle au Shell.

> **Paramètres du terminal**
>
> Vérifiez les paramètres **stty**. La suspension et la reprise d'affichage sont généralement désactivées si vous utilisez des émulations « modernes » de terminaux. Le **xterm** standard reconnaît **Ctrl**+**S** et **Ctrl**+**Q** par défaut.

1.2. Utilisation de signaux avec kill

La plupart des Shell récents, Bash inclus, ont une fonction intégrée **kill**. Dans Bash, à la fois les noms de signaux et leur numéro sont acceptés en tant qu'option, et les arguments peuvent être des identifiants de travaux ou de processus. Un statut d'exécution peut être renvoyé avec l'option `-l` : zéro si au moins un signal a été envoyé correctement, différent de zéro si une erreur s'est produite.

Avec la commande **kill** de `/usr/bin`, votre système peut activer des options supplémentaires, telles que la capacité de tuer des processus provenant d'autres identifiants utilisateurs que le votre, et celle de spécifier les processus par leur nom, comme avec **pgrep** et **pkill**.

Les 2 commandes **kill** envoient le signal *TERM* si aucun n'est donné.

Voici une liste des principaux signaux :

Tableau 12.2. Signaux courants de kill

Nom du signal	Valeur du signal	Effet
SIGHUP	1	Suspend
SIGINT	2	Interruption depuis le clavier
SIGKILL	9	signal kill
SIGTERM	15	signal d'arrêt d'exécution
SIGSTOP	17,19,23	Stoppe le processus

> **SIGKILL et SIGSTOP**
>
> *SIGKILL* et *SIGSTOP* ne peuvent pas être trappés, bloqués ou ignorés.

Pour tuer un processus ou une série de processus, il est de bon sens de commencer par essayer avec le signal le moins dangereux, *SIGTERM*. De cette façon, les programmes qui se soucient d'un arrêt correct ont une chance de suivre les procédures qui leur ont été demandé d'exécuter à la réception du signal *SIGTERM*, tel que purger et fermer les fichiers ouverts. Si vous envoyez un *SIGKILL* à un processus, vous retirez toute chance au processus d'effectuer un arrêt soigné, ce qui peut avoir des conséquences néfastes.

Mais si l'arrêt soigné ne fonctionne pas, le signal *INT* ou *KILL* peut être le seul moyen. Par exemple,quand un processus ne meurt pas avec **Ctrl**+**C**, c'est mieux d'utiliser **kill -9** sur cet ID de processus :

```
maud: ~> ps -ef | grep stuck_process
maud    5607   2214  0 20:05 pts/5    00:00:02 stuck_process

maud: ~> kill -9 5607

maud: ~> ps -ef | grep stuck_process
maud    5614    2214 0 20:15 pts/5    00:00:00 grep stuck_process
[1]+ Killed              stuck_process
```

Quand un processus démarre plusieurs instances, **killall** peut être plus facile. Elle prend la même option que la commande **kill**, mais s'applique à toutes les instances d'un processus donné. Tester cette commande avant de l'employer dans un environnement de production, parce qu'elle pourrait ne pas fonctionner comme attendu sur certains UNIX commerciaux.

2. Piéger les signaux

2.1. Généralité

Il peut y avoir des situations ou vous ne souhaitez pas que les usagers de vos scripts quittent abruptement par une séquence de touches du clavier, par exemple parce qu'une entrée est en attente ou une purge est à faire. L'instruction **trap** trappe ces séquences et peut être programmée pour exécuter une liste de commandes à la récupération de ces signaux.

La syntaxe de l'instruction **trap** est directe :

```
trap [COMMANDS] [SIGNALS]
```

Ceci indique à la commande **trap** de récupérer les *SIGNAUX* listés, qui peuvent être des noms de signaux avec ou sans le préfixe *SIG*, ou des numéros de signaux. Si un signal est *0* ou *EXIT*, les **COMMANDES** sont exécutées quand le Shell se finit. Si l'un des signaux est *DEBUG*, la liste des **COMMANDES** est exécutée après chaque commande simple. Un signal peut être aussi spécifié pour *ERR* ; dans ce cas **COMMANDES** sont exécutées chaque fois qu'une commande simple s'achève avec un statut différent de zéro. Notez que ces commandes ne seront pas exécutées quand le statut d'exécution différent de zéro vient d'une instruction **if**, ou d'une boucle **while** ou **until**. Aucune ne sera exécutée si un *AND* (&&) ou un *OR* (||) logique donne un statut d'exécution différent de zéro, ou quand le code retour d'une commande est inversé par l'opérateur *!*.

Le statut renvoyé par la commande **trap** elle-même est zéro à moins qu'un signal invalide ait été spécifié. La commande **trap** admet des options qui sont documentées dans les pages info de Bash.

Voici un exemple très simple, récupérant **Ctrl**+**C** frappé par l'usager, ce qui déclenche l'affichage d'un message. Quand vous essayez de tuer ce programme sans spécifier le signal *KILL*, rien ne se produit :

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

trap "echo Booh!" SIGINT SIGTERM
echo "pid is $$"

while :                 # Ceci est équivalent à « while true ».
do
        sleep 60        # Ce script ne fait pas vraiment quelque chose.
done
```

2.2. Comment Bash interprète trap

Quand Bash reçoit un signal pour lequel un piège a été défini durant l'exécution d'une commande, le piège ne sera mis en oeuvre que une fois que la commande aura terminé. Quand Bash attend après une commande asynchrone via l'intégrée **wait**, la réception d'un signal pour lequel un piège a été défini fera que l'intégrée **wait** redonnera la main immédiatement avec un statut d'exécution supérieur à 128, immédiatement après quoi le piège est exécuté.

2.3. Plus d'exemples

2.3.1. Détecter quand une variable est utilisée

Quand vous débuggez de longs scripts, vous pouvez vouloir donner à une variable l'attribut *trace* et trapper les messages *DEBUG* pour cette variable. Normalement vous déclarez juste une variable avec une affectation du genre **`VARIABLE=valeur`**. En remplaçant la déclaration de la variable avec les lignes suivantes vous pouvez obtenir des informations intéressantes sur ce que fait votre script :

```
declare -t VARIABLE=valeur
trap "echo VARIABLE est utilisée ici." DEBUG
# suite du script
```

2.3.2. Purger les déchets de traitements avant de quitter

La commande **whatis** repose sur une base de données qui est régulièrement reconstruite avec le script `makewhatis.cron` lancé par cron :

```
#!/bin/bash
LOCKFILE=/var/lock/makewhatis.lock
# Le makewhatis précédent devrait s'être exécuté avec succès :
[ -f $LOCKFILE ] && exit 0
# Avant de quitter, éliminer les fichiers verrous.
trap "{ rm -f $LOCKFILE ; exit 255; }" EXIT
touch $LOCKFILE
makewhatis -u -w
exit 0
```

3. Résumé

Des signaux peuvent être envoyés à votre programme avec la commande **kill** ou des raccourcis clavier. Ces signaux peuvent être récupérés, sur quoi une action peut être effectuée, avec l'instruction **trap**.

Certains programmes ignorent les signaux. Le seul signal qu'aucun programme ne peut ignorer est *KILL*.

4. Exercices

Quelques exemples :

1. Ecrire un script qui écrit une image de démarrage système sur une disquette avec l'outil **dd**. Si l'utilisateur essaye d'interrompre avec **Ctrl**+**C**, afficher un message disant que cette action rendra la disquette inutilisable.

2. Ecrire un script qui automatise l'installation d'un paquetage tiers de votre choix. Le paquetage doit être téléchargé par INTERNET. Il doit être décompressé, désarchivé et compilé si ces actions sont appropriées. Seule l'opération d'installation du paquetage ne devrait pas être interruptible.

Annexe A. Possibilités du Shell

Table des matières

1. Fonctionnalités courantes
2. Fonctionnalités spécifiques

Résumé

Ce document donne une vue globale des possibilités courantes d'un Shell et des spécificités possibles.

1. Fonctionnalités courantes

Les fonctionnalités suivantes sont standard dans tout Shell. Notez que les commandes stop, suspend, jobs, bg et fg sont disponibles seulement sur les systèmes qui permettent le contrôle des travaux (job control).

Tableau A.1. Fonctionnalités courantes du Shell

Commande	**sens**
>	Redirige la sortie
>>	Ajoute en fin de fichier
<	Redirige l'entrée
<<	Document en ligne (redirige l'entrée)
|	Sortie par un tube
&	Place le travail en tâche de fond.
;	Sépare des commandes sur une même ligne
*	Correspond à n'importe quel(s) caractère(s) dans un nom de fichier
?	Correspond à n'importe quel caractère unique dans un nom de fichier
[ ]	Correspond à n'importe quels caractères inclus
( )	S'exécute dans un sous-Shell
` `	Substitue le contenu par le résultat de la commande incluse
" "	Guillemets ou citation partielle (permet l'expansion de variables et de commandes)
' '	Apostrophes ou citation totale (pas d'expansion)
\	Citation du caractère qui suit (NdT : emploi du caractère dans son sens littéral->échappement)
$var	Emploie la valeur de la variable
$$	Identifiant du processus
$o	Nom de la Commande
$n	Nième argument (N entre 0 et 9)
#	Commence un commentaire
bg	Exécution en tâche de fond
break	Sortir d'une instruction de boucle
cd	Change de répertoire
continue	Passe à l'itération suivante d'une boucle dans un programme
echo	Affiche le résultat
eval	Evalue les arguments
exec	Exécute un nouveau processus Shell
fg	Exécution dans la session en cours
jobs	Affiche les travaux en cours
kill	Termine un travail en cours

Commande	sens
newgrp	Change de groupe l'utilisateur
shift	Décale les paramètres positionnels
stop	Suspend un travail en tâche de fond
suspend	Suspend un travail en cours dans la session
time	Chronomètre une commande
umask	Donne ou liste les permissions sur les fichiers
unset	Supprime une variable ou une fonction
wait	Attend qu'une tâche de fond se termine

2. Fonctionnalités spécifiques

La table suivante montre les principales différences des Shell courants (**sh**), Bourne Again SHell (**bash**), Korn shell (**ksh**) et le C shell (**csh**).

> **Compatibilité des Shell**
>
> Parce que Bash est un sur-ensemble de **sh**, toutes les commandes **sh** fonctionnent en **Bash** - mais pas vice versa. **Bash** a bien plus de possibilités qui lui sont propres, et, comme le montre la table suivante, beaucoup de possibilités venant d'autres Shell.
>
> Parce que le Turbo C Shell est un sur-ensemble de **csh**, toutes les commandes **csh** fonctionnent en **tcsh**, mais l'inverse n'est pas vrai.

Tableau A.2. Différences de fonctionnalités des Shell

sh	Bash	ksh	csh	Signification/Action
$	$	$	%	L'invite utilisateur par défaut
	>|	>|	**>!**	Force la redirection
> `fichier` 2>&1	**&> `fichier`** ou **> `fichier` 2>&1**	**> `fichier` 2>&1**	**>& `fichier`**	Redirige stdout et stderr sur `fichier`
	{ }		**{ }**	Expansion des éléments de la liste
`commande`	**`commande`** ou **$(commande)**	**$(commande)**	**`commande`**	Remplace par le résultat de la **commande** incluse
`$HOME`	`$HOME`	`$HOME`	`$home`	Répertoire utilisateur
	~	~	~	Symbole équivalent au répertoire utilisateur
	~+, ~-, **dirs**	~+, ~-	=-, =N	Accède à la pile des répertoires
var=value	**VAR=value**	**var=value**	**set var=value**	Affectation de variable
export var	**export VAR=value**	**export var=val**	**setenv var *val***	Publie une variable d'environnement

sh	Bash	ksh	csh	Signification/Action
	`${nnnn}`	`${nn}`		Les paramètres peuvent être référencés au delà des 9 premiers
`"$@"`	`"$@"`	`"$@"`		Chaque argument est connu comme une valeur indépendante
`$#`	`$#`	`$#`	`$#argv`	Le nombre d'arguments
`$?`	`$?`	`$?`	`$status`	Statut d'exécution de la commande la plus récente
`$!`	`$!`	`$!`		PID de la tâche de fond la plus récente
`$-`	`$-`	`$-`		Options classiques
`. fichier`	**source `fichier`** ou **`. fichier`**	**`. fichier`**	**source `fichier`**	Lecture de commandes depuis un fichier
	alias x='y'	**alias x=y**	**alias x y**	Nom **x** est équivalent à la commande **y**
case	**case**	**case**	**switch** ou **case**	Décline différentes éventualités
done	**done**	**done**	**end**	Fini une instruction de boucle
esac	**esac**	**esac**	**endsw**	Marque la fin du **case** ou du **switch**
exit ***n***	**exit** ***n***	**exit** ***n***	**exit** ***(expr)***	Quitte avec un statut d'exécution
for/do	**for/do**	**for/do**	**foreach**	Boucles sur plusieurs variables
	set `-f`, **set `-o nullglob|dotglob|nocaseglob|noglob`**		**noglob**	Ignore la substitution de caractères dans la génération de fichier
hash	**hash**	**alias `-t`**	**hashstat**	Affiche les commandes 'hash' (trace les alias)
hash ***cmds***	**hash** ***cmds***	**alias `-t`** ***cmds***	**rehash**	Mémorise où se situe la commande
hash `-r`	**hash `-r`**		**unhash**	Annule la mémorisation
	history	**history**	**history**	Liste les commandes passées
	ArrowUp+**Enter** ou **!!**	**r**	**!!**	Relance la commande précédente
	!*str*	**r** ***str***	**!*str***	Relance la commande la plus récemment passée qui commence par « str »
	!*cmd*:s/*x*/*y*/	**r** ***x=y cmd***	**!*cmd*:s/*x*/*y*/**	Remplace « x » par « y » dans la commande la plus récemment passée

sh	Bash	ksh	csh	Signification/Action
				commençant par « cmd », puis exécute.
if [$i -eq 5]	**if [$i -eq 5]**	**if ((i==5))**	**if ($i==5)**	Echantillon de tests de conditions
fi	**fi**	**fi**	**endif**	Marque la fin de l'instruction **if**
ulimit	**ulimit**	**ulimit**	**limit**	Déclare une limite de ressource
pwd	**pwd**	**pwd**	**dirs**	Affiche le répertoire courant
read	**read**	**read**	**$<**	Lecture depuis l'entrée
trap *2*	**trap *2***	**trap *2***	**onintr**	A pour effet d'ignorer les interruptions
	unalias	**unalias**	**unalias**	Détruit les alias
until	**until**	**until**		Begin **until** loop
while/do	**while/do**	**while/do**	**while**	Begin **while** loop

Bourne Again SHell a bien d'autres possibilités non évoquées ici. Ce tableau donne un aperçu de comment ce Shell intègre toutes les bonnes idées des autres Shell : il n'y a pas de blanc dans la colonne **bash**. Plus d'informations sur les possibilités propres à Bash peuvent être trouvées dans les pages d'info Bash, dans la section « Bash Features ».

Plus d'informations :

Vous devriez au moins lire un manuel, même si c'est celui de votre Shell. Le choix pourrait être **info bash**, **bash** étant le Shell GNU et le plus facile pour le débutant. Imprimez-le et emportez-le à la maison, l'étudier dès que vous avez 5 minutes.

Annexe B. GNU Free Documentation License

Table des matières

1. Preamble
2. Applicability and definitions
3. Verbatim copying
4. Copying in quantity
5. Modifications
6. Combining documents
7. Collections of documents
8. Aggregation with independent works
9. Translation
10. Termination
11. Future revisions of this license
12. How to use this License for your documents

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

1. Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. Applicability and definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

3. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4. Copying in quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

5. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. Combining documents

You may combine the Document with other documents released under this License, under the terms

defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

7. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

9. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

10. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long

as such parties remain in full compliance.

11. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

12. How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Glossaire

Résumé

Cette section contient un aperçu des principales commandes UNIX par ordre alphabétique. Les détails d'utilisation peuvent être trouvés dans les pages info et man.

A

a2ps

> Formate des fichiers qui puissent être imprimés sur une imprimante PostScript.

acroread

> Un visionneur de PDF.

adduser

> Créer un nouvel utilisateur ou modifier les informations par défaut de nouveaux utilisateurs.

alias

> Créer un alias Shell d'une commande.

anacron

Exécute des commandes périodiquement, ne suppose pas que l'ordinateur fonctionne en continu.

apropos

Cherche des chaînes dans la base de donnée whatis

apt-get

Un outil de gestion de paquets APT.

aspell

Contrôleur d'orthographe

at, atq, atrm

Place en file d'attente, examine ou supprime des travaux en attente d'exécution.

aumix

Ajuste le mixer audio

(g)awk

Recherche de patrons et langage de traitement du texte.

B

Bash

Bourne Again SHell.

batch

Place en file d'attente, examine ou supprime des travaux en attente d'exécution.

bg

Lance un travail en tâche de fond.

bitmap

Editeur Bitmap et outils de conversion pour le système X window.

bzip2

Un compresseur de fichier qui trie sur les blocs.

C

cat

Concatène des fichiers et affiche sur la sortie standard.

cd

Change de répertoire

cdp/cdplay

Un programme interactif en mode texte pour contrôler et écouter des CD Rom audio sous

Linux.

cdparanoia

Un outil de lecture de CD audio qui inclut des fonctionnalités supplémentaires de vérification de données.

cdrecord

Enregistre un CD-R

chattr

Modifie les attributs de fichiers.

chgrp

Modifie le groupe propriétaire.

chkconfig

Modifie ou affiche les informations de niveau d'exécution (NdT : run level) des services systèmes.

chmod

Change les permissions d'accès aux fichiers.

chown

Modifie le propriétaire et le groupe d'un fichier.

compress

Comprime des fichiers.

cp

Copie des fichiers et des répertoires.

crontab

Gestion des fichiers de crontab.

csh

Ouvre un Shell C.

cut

Elimine des sections de chaque ligne de fichiers.

D

date

Affiche ou définit la date et l'heure système.

dd

Convertit et copie un fichier (vidage/affichage du contenu du disque).

df

Affiche le pourcentage d'utilisation des disques du système de fichiers.

dhcpcd

Démon client de DHCP

diff

Trouve les différences entre 2 fichiers.

dig

Envoie des paquets de requête de nom de domaines aux serveurs de noms.

dmesg

Affiche et contrôle le tampon en anneau du noyau.

du

Estime l'utilisation de l'espace des fichiers.

E

echo

Affiche une ligne de texte.

ediff

le Diff du traducteur anglais.

egrep

grep étendu.

eject

Démonte et éjecte les médias amovibles.

emacs

Lance l'éditeur Emacs.

exec

Invoque des sous-processus.

exit

Quitte le Shell courant

export

Ajoute des fonctions à l'environnement du Shell.

F

fax2ps

Converti un facsimile TIFF au format PostScript.

fdformat

Formate une disquette.

fdisk

Manipulation de la table des partitions sous Linux.

fetchmail

Récupère le courrier depuis un serveur POP, IMAP, ETRN ou compatible ODMR.

fg

Ramène un travail en exécution sur le premier plan.

file

Détermine le type de fichier.

find

Cherche des fichiers.

formail

(re)formate les courriels.

fortune

Affiche au hasard un adage supposé rendre l'espoir.

ftp

Service de transfert de fichiers (peu sûr sauf si un compte anonyme est utilisé !).

G

galeon

Navigateur graphique.

gdm

Le gestionnaire d'affichage Gnome.

(min/a)getty

Contrôle les périphériques console.

gimp

Programme de manipulation d'image.

grep

Affiche les lignes trouvées selon un certain patron.

grub

Le Shell grub

gv

Un visionneur de PDF et PostScript.

gzip

Comprime ou extrait des fichiers.

H

halt

Arrête le système

head

Affiche le début d'un fichier.

help

Affiche de l'aide sur une commande Shell intégrée.

host

Un outil d'exploration DNS.

httpd

Serveur au protocole de transfert d'hypertexte Apache.

I

id

Affiche les véritables UIDs et GIDs.

ifconfig

Configure l'interface réseau ou affiche la configuration.

info

Lit les documents info.

init

Initialisation du contrôle de processus.

iostat

Affiche des statistiques I/O(NdT : Entrée/Sortie).

ip

Affiche/modifie le statut de l'interface réseau.

ipchains

Gestion du pare-feu IP.

iptables

Gestion du filtre des paquets IP.

J

jar

Outil d'archivage java.

jobs

Affiche les tâches de fond.

K

kdm

Le gestionnaire du bureau KDE.

kill(all)

Fait se terminer le(s) processus.

ksh

Ouvre un Shell Korn.

L

ldapmodify

Modifie une entrée LDAP.

ldapsearch

Outil de recherche LDAP.

less

more avec fonctionnalités.

lilo

Le chargeur d'amorçage Linux.

links

Navigateur WWW en mode texte.

ln

Crée des liens entre fichiers.

loadkeys

Charge les tables de correspondance clavier.

localiser

Cherche des fichiers.

logout

Quitte le Shell courant.

lp

Envoie des requête au service d'impression LP.

lpc

Programme de contrôle de la ligne d'impression.

lpq

Programme de contrôle du fichier contenant la file d'impression.

lpr

Impression différée.

lprm

Supprime une requête d'impression.

ls

Affiche le contenu d'un répertoire.

lynx

Navigateur WWW en mode texte.

M

mail

Envoie et reçoit le courrier.

man

Lit les pages man.

mcopy

Copie des fichiers MS-DOS depuis/vers Unix.

mdir

Affiche un répertoire MS-DOS.

memusage

Affiche l'état d'utilisation de la mémoire.

memusagestat

Affiche des statistiques d'utilisation de la mémoire.

mesg

Contrôler l'accès en écriture à votre terminal.

mformat

Ajoute un système de fichiers MS-DOS à une disquette formatée en faible densité.

mkbootdisk

Crée une disquette de démarrage autonome du système local.

mkdir

Créé un répertoire.

mkisofs

Créé un système de fichiers ISO9660 hybride.

more

Filtre qui permet d'afficher écran par écran.

mount

Monte un système de fichiers ou affiche les informations sur les systèmes de fichiers montés.

mozilla

Navigateur WEB.

mt

Contrôle les opérations des périphériques à bande magnétique.

mtr

Outil de diagnostic réseau.

mv

Renomme des fichiers.

N

named

Serveur de nom de domaines INTERNET.

ncftp

Programme de navigation dans les services ftp (peu sûr !).

netstat

Affiche les connections réseaux, les tables de routage, des statistiques d'interface, les connections masquées.

nfsstat

Affiche des statistiques sur les systèmes de fichiers réseau.

nice

Modifie en file d'attente la priorité d'exécution d'un programme.

nmap

Outil d'exploration du réseau et scanner du point de vue sécurité.

ntsysv

Interface simple pour configurer les niveaux d'exécution.

P

passwd

Modifie le mot de passe.

pdf2ps

Transpose du PDF Ghostscript en PostScript.

perl

Practical Extraction and Report Language (NdT : langage de programmation).

pg

Page par l'entremise d'une sortie texte.

ping

Envoie une requête d'écho à un hôte.

pr

Convertit des fichiers textes en vu d'impression.

printenv

Affiche tout ou partie de l'environnement.

procmail

Processeur autonome de mails.

ps

Affiche l'état des processus.

pstree

Affiche un arbre des processus.

pwd

Affiche le répertoire courant.

Q

quota

Affiche l'état d'utilisation des disques et leur limite.

R

rcp

Copie à distance (pas sûr !).

rdesktop

Client du protocole de bureau à distance.

reboot

Arrête et redémarre le système.

renice

Modifie la priorité d'un processus qui s'exécute.

rlogin

Connection à distance - Interface utilisateur du protocole TELNET (peu sûr !).

rm

Supprime un fichier.

rmdir

Supprime un répertoire.

rpm

Le gestionnaire de paquets RPM.

rsh

Shell à distance (peu sûr !).

S

scp

Copie à distance sécurisée.

screen

Un gestionnaire d'écran à émulation VT100.

set

Affiche, déclare ou définit une variable.

setterm

Détermine les attributs du terminal.

sftp

ftp sécurisé (cryptage).

sh

Ouvre un Shell standard.

shutdown

Arrête le système.

sleep

Attend un temps donné.

slocate

Version sécurisée du Locate GNU.

slrnn

Client Usenet en mode texte.

snort

Outil de détection d'intrusion de réseau.

sort

Tri les lignes d'un fichier texte.

ssh

Shell sécurisé.

ssh-keygen

Génération de clé d'authentification.

stty

Modifie et affiche le paramétrage des lignes de terminal.

su

Change d'utilisateur.

T

tac

Concatène et affiche des fichiers de la fin vers le début.

tail

Affiche la dernière partie d'un fichier.

talk

Envoie un message à un usager.

tar

Outil d'archivage.

tcsh

Ouvre un Shell Turbo C.

telnet

Interface utilisateur au protocole TELNET (peu sûr !).

tex

Formatage de texte et configuration de type.

time

Chronomètre une commande simple ou affiche l'utilisation des ressources.

tin

Programme de lecture de nouvelles.

top

Affiche les processus les plus gourmands en CPU.

touch

Modifie les propriétés d'heure et de date d'un fichier.

traceroute

Affiche la route que les paquets prennent jusqu'à un hôte réseau.

tripwire

Un contrôleur d'intégrité de fichiers sur les systèmes UNIX.

twm

Gestionnaire de tabulations du système de gestion de fenêtre X.

U

ulimit

Contrôle les ressources.

umask

Détermine le masque pour la création de fichier par l'utilisateur.

umount

Démonte un système de fichiers.

uncompress

Décompresse des fichiers.

uniq

Elimine les lignes en double dans un fichier trié.

update

Démon du noyau qui libère les tampons orphelins sur disques.

uptime

Affiche la durée depuis laquelle le système est en route et sa charge moyenne.

userdel

Supprime un compte utilisateur et ses fichiers associés.

V

vi(m)

Lance l'éditeur vi (amélioré).

vimtutor

Le tutoriel Vim.

vmstat

Affiche des statistiques d'utilisation de la mémoire virtuelle.

W

w

Affiche qui est connecté et ce qu'ils font.

wall

Envoie un message à tous les terminaux.

wc

Affiche le nombre d'octets, mots et lignes d'un fichier.

which

Indique le chemin complet des commandes Shell.

who

Affiche qui est connecté.

who am i

Affiche l'identifiant utilisateur réel.

whois

Interroge une base de donnée des utilisateurs avec surnom ou pas.

write

Envoie un message à un autre utilisateur.

X

xauth

Outil de gestion des fichiers X.

xcdroast

Interface graphique de cdrecord.

xclock

Horloge analogique/numérique pour X.

xconsole

Gère les messages de la console système avec X.

xdm

Gestionnaire d'affichage X qui admet XDMCP, le sélectionneur d'hôte.

xdvi

Visionneur de DVI.

xfs

Serveur de police X.

xhost

Programme de contrôle d'accès au serveur pour X.

xinetd

Le démon de services INTERNET étendu.

xload

Affichage de la charge moyenne du système pour X.

xlsfonts

Afficheur de la liste des polices du système pour X.

xmms

Diffuseur audio pour X.

xpdf

Un visionneur de PDF.

xterm

Emulateur de terminal pour X.

Z

zcat

Comprime ou extrait des fichiers.

zgrep

Cherche dans d'éventuels fichiers compressés une expression régulière.

zmore

Filtre pour afficher du texte compressé.

Index

Symboles

/etc/bashrc
example, /etc/bashrc
/etc/profile
example, /etc/profile
.bash_login
example, ~/.bash_login
.bash_logout
example, ~/.bash_logout
.bash_profile
example, ~/.bash_profile
.bashrc
example, ~/.bashrc

A

alias, Que sont les alias ?
aliases
aliases in functions, Créer et supprimer des alias
creation, Créer et supprimer des alias
definition, Que sont les alias ?
delete an alias, Que sont les alias ?
examples, /etc/bashrc, ~/.bashrc, Que sont les alias ?
expand_aliases, Que sont les alias ?
misspelled commands, /etc/bashrc

usage, Que sont les alias ?
ANSI-C
quoting, Codage ANSI-C
arguments
definition, L'exécution de commandes
example existence, Test de l'existence d'un fichier
examples, Contrôle des paramètres de la ligne de commande
example test number, Tester le nombre de paramètres
exit status, Emploi de l'instruction exit et du if
function example, Les paramètres positionnels dans les fonctions
functions, Les paramètres positionnels dans les fonctions
null arguments, Le découpage de mots
number of arguments, Contrôle des paramètres de la ligne de commande
positional parameters, Paramètres spéciaux, Contrôle des paramètres de la ligne de commande
test example, Exemples de tableaux
testing, Contrôle des paramètres de la ligne de commande
to a command, Les commandes Shell
arithmetic evaluation
declare built-in, Utiliser l'intégrée declare
arithmetic expansion
operators, L'expansion arithmétique, Expressions employées avec if
syntax, L'expansion arithmétique
arithmetic expression
testing, Opérations booléennes
arrays
adding members, Créer des tableaux
attributes, Créer des tableaux
declaration, Utiliser l'intégrée declare, Créer des tableaux
dereferencing, Invoquer les variables d'un tableau
examples, Exemples de tableaux
number of elements, Longueur de variable
remove patterns, Suppression de sous-chaînes
unset, Supprimer des variables tableau
variables, Affectation générale de valeur.
awk
BEGIN, Patrons particuliers
definition, Qu'est-ce que gawk ?
example, Plus d'exemples
example fields, Afficher les champs sélectionnés
field formatting, Formater les champs
formatting characters, Formater les champs
formatting example, Formater les champs
input field separator, Le séparateur de champs en entrée
input interpretation, Afficher les champs sélectionnés
number of records, Le nombre d'enregistrements
output field separator, Les séparateurs de champs de résultat
output record separator, Le séparateur d'enregistrement de résultat
printf program, Le programme printf
print program, Afficher les champs sélectionnés
program on the command line, Commandes Gawk
program script, Commandes Gawk
regexp example, La commande print et les expressions régulières
regular expressions, La commande print et les expressions régulières
script example, Les scripts Gawk
scripts, Les scripts Gawk
user defined variables, Les variables définies par l'utilisateur
variables, Les variables Gawk

B

Bash
advantages, Bash est le Shell GNU
features, Invocation
startup files, Fichiers de démarrage de Bash
brace expansion
examples, L'expansion d'accolades
built-in
popd, Pile de répertoires
built-ins
Bash built-ins, Les commandes intégrées du Shell
Bourne Shell built-ins, Les commandes intégrées du Shell
declare, Les paramètres Shell
enable, Le Shell restreint
export, Exporter les variables
hash, Le Shell restreint
pushd, Pile de répertoires
source, Le Shell restreint, Exécuter le script
special built-ins, Les commandes intégrées du Shell

C

commands
built-in commands, Les commandes intégrées du Shell
env, Les variables Globales
execution, Généralité
fork-and-exec, Généralité
printenv, Les variables Globales
script execution, Exécuter un programme dans un script.
search for commands, L'exécution de commandes
sh-utils package, Les variables Globales
command substitution
syntax, La substitution de commande
comments
usage, Ajout de commentaires
configuration files
/etc/bashrc, /etc/bashrc
/etc/inputrc, /etc/profile
/etc/profile, /etc/profile
/etc.profile.d, /etc/profile
.bash_login, ~/.bash_login
.bash_logout, ~/.bash_logout
.bash_profile, ~/.bash_profile
.bashrc, ~/.bashrc
.profile, ~/.profile
change shell configuration, Modification des fichiers de configuration du Shell
prompt, Modification des fichiers de configuration du Shell

D

debugging, Débugger le script globalement
echo statements, Débugger qu'une partie du script
on entire script, Débugger le script globalement
options, Débugger qu'une partie du script
partial, Débugger qu'une partie du script

E

exit status
arguments, Emploi de l'instruction exit et du if

expansion
arithmetic expansion, L'expansion arithmétique
brace expansion, L'expansion d'accolades
command substitution, La substitution de commande
file name expansion, Expansion de noms de fichier
indirect expansion, Paramètre Shell et expansion de variable
process substitution, La substitution de processus
tilde expansion, L'expansion du tilde
variable expansion, Paramètre Shell et expansion de variable
word splitting, Le découpage de mots

F

features
aliases, Alias
arrays, Tableaux
conditionals, Les conditions
directory stack, Pile de répertoires
interactive shells, Qu'est-ce qu'un Shell interactif
invocation, Invocation
prompt, L'invite
restricted shell, Le Shell restreint
scripts, Les scripts Shell
shell arithmetic, L'arithmétique avec Shell
startup files, Fichiers de démarrage de Bash
fichier de configuration
/etc/passwd, Types de Shell
/etc/shells, Types de Shell
file name expansion
characters, Expansion de noms de fichier
function
arguments, Les paramètres positionnels dans les fonctions
example arguments, Les paramètres positionnels dans les fonctions
functions
execution, La fonction Shell
restricted shell, Le Shell restreint

G

gawk
awk, Qu'est-ce que gawk ?

I

input
analysis, La syntaxe Shell
input field separator
word splitting, Le découpage de mots
interactive shell
behavior, Le comportement d'un Shell interactif
invocation
interactive login shell, Invoqué pour être le Shell d'interaction, ou avec l'option `--login'
interactive non-login shell, Invoqué comme Shell interactif sans étape de connexion
invoked as sh, Invoqué avec la commande sh
non-interactive, Invoqué non interactivement
POSIX mode, Mode POSIX
remote invocation, Invoqué à distance
UID <> EUID, Invoqué alors que UID est différent de EUID

O

options
changing options, Changer les options
display all options, Afficher les options
ignoreeof, Le comportement d'un Shell interactif
nocaseglob, Expansion de noms de fichier
noclobber, Changer les options
nullglob, Expansion de noms de fichier
restricted, Le Shell restreint

P

positional parameters
example, Paramètres spéciaux
printenv
example, Les variables Globales
process substitution
syntax, La substitution de processus

Q

quoting characters
ANSI-C quoting, Codage ANSI-C
double quotes, Les guillemets
escape characters, Le caractère Echap (escape)
locale, Particularités
single quotes, Les apostrophes

R

restricted shell
behavior, Le Shell restreint

S

scripts
comments, Ajout de commentaires
considerations, Caractéristiques d'un bon script
creation, Écrire et nommer
debugging, Débugger le script globalement
example, Un exemple Bash script : mysystem.sh, script1.sh
executing shell, Quel Shell exécutera le script ?
execution, Exécuter le script
init script example, Exemple : init script (NdT d'initialisation)
init scripts, Exemple : init script (NdT d'initialisation)
logic, Un mot sur l'ordre et la logique
logic flow example, Un mot sur l'ordre et la logique
naming, Écrire et nommer
permissions, Exécuter le script
structure, Structure
terminology, Terminologie
shell
expansion, Le processus d'expansion de Shell
general functions, Les fonctions du Shell en général
interactive, Qu'est-ce qu'un Shell interactif
parameters, Les paramètres Shell
permuter entre Shells, Types de Shell

redirections, Redirections
syntax, La syntaxe Shell
types, Types de Shell
special parameters
examples, Paramètres spéciaux

T

tilde expansion
syntax, L'expansion du tilde

V

variable and parameter expansion
example, Paramètre Shell et expansion de variable
variables
advantages, Script à finalités multiples grâce aux variables
BASH_ENV, Invoqué non interactivement, Le Shell restreint
Bash reserved, Les variables réservées de Bash
Bourne shell reserved, Variables réservées du Bourne Shell
content types, Variables typées selon leur contenu
creation, Créer des variables
definition, Les paramètres Shell
detect content assignment, Changer les options
DIRSTACK, Pile de répertoires
ENV, Invoqué avec la commande sh, Le Shell restreint
environment variables, Les variables Globales
exporting, Exporter les variables
global variables, Les variables Globales
GLOBIGNORE, Expansion de noms de fichier
HISTFILE, Le comportement d'un Shell interactif
HISTSIZE, /etc/profile
HOSTNAME, /etc/profile
IFS, Le découpage de mots
MAIL, /etc/profile
PATH, Le Shell restreint, L'exécution de commandes, script1.sh, /etc/profile
positional parameters, Paramètres spéciaux
POSIXLY_CORRECT, Mode POSIX
PS2, Le comportement d'un Shell interactif
SHELL, Le Shell restreint
SHELLOPTS, Le Shell restreint
special parameters, Paramètres spéciaux
subshells, Exporter les variables
types, Types de variables
USER, /etc/profile
varibles
local variables, Variables locales

W

word splitting
input field separator, Le découpage de mots