Les Scripts avec le shell bash

version 0.8.5

Cours & travaux pratiques

```
function affiche_evenmt¶
{¶
····rm /var/tmp/extrait_journal¶
····date_cour=$1¶
····date_arret=$2¶
····while [ $date_cour <= $date_arret ] ;··do¶
····· grep $date_cour < $fich >> /var/tmp/extrait_journal \ 2>/dev/null¶
·····date_cour=$(date +%Y-%m-%d -d "$date_cour +1 day")¶
 → done¶
 → less /var/tmp/extrait_journal¶
} #·fin·de·la·fonction affiche_evenmt¶
```

Michel Scifo

Le Maître Réfleur

Attention

Ce livre numérique ne comporte ni dispositif de cryptage limitant son utilisation ni identification par un tatouage permettant d'assurer sa traçabilité, mais il est sous licence CC-BY-NC-ND (cf Annexe 5) ; à ce titre, il est librement diffusible, à condition de ne pas être vendu seul ou associé.

Nous essayons de respecter le Code de la Typographie, mais quand la tâche s'avère trop complexe, ou trop éloignée de nos préoccupations, nous passons outre !

Dépôt légal 1er trimestre 2015
ISBN : 978-9537431-5-9

Le Maître Réfleur
12 AV Général Roux
38800 Le Pont-de-Claix

Sommaire

Attention 2
Introduction 11
Qu'est-ce qu'un script 14
La Programmation 14
La Programmation par l'exemple 15
Analyse de l'énoncé 15
Premier niveau, le tout 16
Deuxième niveau : exemple d'une fonction 18
Troisième niveau ; exemple de sous-procédure 19
Quand arrête-t-on l'analyse ? 19
Un Programme bien pensé 20
Un Exemple concret : la bataille navale 23
Énoncé 23
Description de l'affichage 24
Analyse niveau 1 25
Analyse niveau 2 26
Analyse niveau 3 27
Représentation des données 27
analyse générale 29
Qu'est-ce qu'un script bash ? 32
Rappels 32
Quels outils utiliser ? 33
vim 34
Geany 35
Mise en œuvre 37
Un Exemple de script bash 39
Coloration syntaxique 40
Premier script 41
Exécution d'un script 41
Un Petit « Bonjour » 44
Le Langage de programmation 46
Les Variables & les expressions 47
Définir & utiliser une variable 47
Les Expressions 50

Les Variables du système....56
Les Commandes....57
Les Commandes séquentielles....58
Les Structures de contrôle....60
Exemple récapitulatif 1....69
Première étape : que faut-il faire ?....69
Deuxième étape : comment faire ? 1er niveau, ... .69
Troisième étape : comment faire ? 2nd niveau, ...70
Quatrième étape : le script bash....70
Exemple récapitulatif 2....72
Compléments sur la programmation en bash....75
Généralités....75
L'Initialisation....76
Shells interactifs, mais pas de login....76
Shells non-interactifs....76
Fonctionnement....77
Exécution des commandes....77
Environnement....78
Statut ou code de retour....79
Transferts de données : redirections & tubes....79
Tubes (Pipes)....80
Redirection....80
Les Expressions....87
Expressions simples, les opérateurs insolites....87
Notion de commandes....88
Commandes simples....89
Listes....89
Commandes composées....90
Développement, expansion & substitution....96
Interprétation d'une ligne de commande....96
Expansion des accolades....98
Développement du Tilde....100
Remplacement des paramètres....101
Substitution de commandes....108
Évaluation Arithmétique....109

Substitution de Processus....109
Séparation des mots....109
Développement des noms de fichiers....110
Motifs génériques....110
Suppression des protections 01021....112
Les Expressions rationnelles....113
Définitions....113
Règles....114
Informations complémentaires....116
Cahier d'exercices....118
Rappels de cours....118
Avertissement....118
Exercices simples....120
Ex. 0 : expressions....120
Énoncé 1....120
Commandes....121
Aide....121
Énoncé 2....121
Commandes....122
Aide....122
Ex. 1 : énumération....123
Énoncé....123
Commandes....123
Aide....123
Ex. 2 : salutation....125
Énoncé....125
Commandes....125
Aide....125
Ex. 3 : améliorations de salutation....126
Énoncé....126
Aide....127
Ex. 4 : énumérations variées....128
Aide....128
Ex. 5 : trouver le nombre caché....129
Aide....129

Questions supplémentaires....130
Ex. 6 : chasse aux troyens....130
Énoncé....130
Commandes....130
Aide....130
EX. 7 : choisir un jeux....131
Énoncé....131
Variante KISS....132
Variante KICK....132
Aides....132
EXERCICES COMPLEXES....133
Ex. 11 : bataille navale....133
Analyse niveau 4....133
La Traduction en bash....139
Commandes....141
Aide....141
Ex. 12 : sauvegarde....144
Énoncé....144
Commandes....144
Aide....144
Ex. 13 : occupation des partitions....147
Énoncé....147
Commandes....147
Aide....147
Ex. 14 : liste des machines connectées....148
Énoncé....148
Commandes....148
Aide....148
Ex. 15 : consultation des logs personnalisée....151
Énoncé....151
Commandes....151
Aide....151
Ex. 16 : conversion décimal-binaire....160
Énoncé....160

Aide....160
Ex. 17 : compréhension d'un script....161
Énoncé....161
Aide....161
Liste de la fonction à analyser....161
Conclusion....163
Notes....165
Annexe 1 Les Caractères spéciaux....175
Séparateurs/opérateurs....175
Caractères de contrôles....179
Les Espaces....180
L'Échappement....181
Annexe 2 Quelques commandes & fichiers utiles....183
Les Commandes externes pour tous....183
Les Commandes pour l'administration....189
Les Fichiers de configuration....192
Les Commandes internes de bash....194
Annexe 3 Les Expressions rationnelles....199
Théorie....199
Pratique....202
Syntaxe des expressions rationnelles....204
Avertissement....204
. 204
* 204
+ 205
? 205
*?, +?, ?? 205
\{n\} ou {n} 206
\{n,m\} ou {n,m} 206
[...] 206
[^ ...] 207
^ 207
$ 208
\ 208
| 208
(...) 209
Raccourcis....209
Divers....210

Quelques exemples 211
ls 211
grep 211
sed 211
awk 211
vim 211
Annexe 4 Corrigés des exercices scripts Bash 213
Avertissement 213
Corrigés des exercices simples 214
Ex. 0 214
Corrigé 214
Corrigé énoncé 2 petit-joueur 215
Ex. 1 216
Démarche 216
Corrigé 1 217
Corrigé 2 (méthode KISS) 217
Corrigé 3 (méthode KICK) 217
Ex. 2 218
Démarche 218
Corrigé 1 218
Corrigé 2 219
Ex. 4 221
Démarche de base 221
Corrigé compte_w.sh 222
Corrigé compte_u.sh 222
Corrigé compte_f.sh 223
Corrigé variante 2 224
Corrigé variante 3 224
Ex. 5 224
Démarche 224
Corrigé 225
EX. 6 226
Démarche 226
Corrigé 226
EX. 7 226

Démarches....226
Corrigé 1....227
Corrigé 2....228
Corrigés exercices complexes....229
Ex. 11 bataille navale....229
Démarche....229
Corrigé DBD....230
Corrigé MSO....235
Ex. 12....250
Démarche....250
Corrigé....251
Ex. 13....252
Corrigé 1....252
Ex. 14....256
Corrigé 1....256
Corrigé 2....259
Ex. 15....261
Corrigé 1....262
Corrigé 2....266
Annexe 5 Les Licences Creative Commons....273
Paternité, BY....273
Pas d'utilisation commerciale, NC....273
Pas de travaux dérivés, ND....274
Partage à l'identique, SA....274
Remerciements....275
Index lexical....277
A....277
B....277
C....277
D....277
E....277
F....278
I....278
L....278
M....278
N....278
O....278

P.....279
R.....279
S.....279
T.....279
U.....279
V.....279

Introduction

Ce cours/TP est une réécriture, une adaptation & une extension de deux articles de Tristan Colombo parus dans Linux Pratique Hors-Série n°20, *Premier pas en script shell* & *Scripts shell notions avancées*. En effet, l'auteur était d'une part soumis à la contrainte de l'article de revue & d'autre part, à une problématique de public différente.

Si remarquables que soient ces deux textes, ils devaient être adaptés à la formation de technicien supérieur en réseaux informatiques. *C'est-à-dire d'une part, à des personnes n'ayant jamais programmé & d'autre part, cherchant à optimiser l'exploitation de serveurs & de postes de travail & non à écrire des logiciels.* Cependant la philosophie en a été conservé.

Une autre adaptation a porté sur la distribution employée par l'auteur, différente de celle que nous employons. Nous sommes seul responsable des erreurs qu'il peut contenir, en particulier dans les exemples : ceux-ci ont été testés en mode texte, leur mise en forme dans *LibreOffice* peut avoir altéré les lignes originales malgré quelques macros & malgré tout le soin qui y a été apporté.

L'origine des scripts se trouve dans la paresse des informaticiens, certaines commandes *bash* étant difficiles à concevoir & longues à écrire, il était souhaitable de les enregistrer pour les ré-exécuter, à l'identique ou avec des valeurs différentes.

Derrière toute application, bien conçue, ayant une interface graphique, se cachent des appels à des programmes en mode console.

Prenons l'exemple de *K3b*, le logiciel de gravure de KDE : toutes les actions font appel à des programmes en ligne de commandes (`mkisofs`, `cdrecord`, etc.). En connaissant les informations à fournir à ces programmes, vous pouvez donc les utiliser directement en ligne de commandes. Et si vous voulez enchaîner des actions, vous aurez le choix entre cliquer frénétiquement des dizaines de fois sur votre souris [01001], ou écrire un script *bash*...

Lors de la rédaction de la version 0.7, nous avons découvert un site particulièrement intéressant du domaine public, dont nous avons adapté quelques scripts, particulièrement pour `sed` & `awk`, deux utilitaires que nous n'avions plus employés depuis des années. Il s'agit du site http://www. tldp.org/LDP/abs/html/index.html.

Les remarques de Dominique Billard, sur cette même version 0.7 ont générée un remaniement de fond du cahier d'exercice & la multiplication des exemples.

La mention ***cf. option : option1, option2*** signifie que les options de *Bash* mentionnées peuvent influer sur le comportement de la commande ou l'opérateur examiné. Reportez vous à `man bash` pour savoir comment, ces précisions dépassant largement le cadre d'une initiation.

Nous avons réalisé en rédigeant cette version, que l'exercice complexe n° 2 : bataille navale méritait de voir son analyse complète figurer dans le paragraphe La Programmation par l'exemple. Donc l'analyse de l'exercice 11 du chapitre Cahier d'exercices ne se trouvera pas dans ce chapitre. De plus, le corrigé proposera deux versions très différentes du script.

À faire

L'index, l'amélioration la mise en page, la rédaction d'exemples & d'exercices sur les expressions rationnelles, l'achèvement de la traduction des mémentos des commandes & des fichiers de configuration & l'ultime relecture.

Refaire l'index suite au remplacement de la table des matière par le sommaire. En attendant, ajoutez 8 aux numéros de pages.

Qu'est-ce qu'un script

Un *script* est un programme qui sera interprété. Écrire un script, en bref, c'est programmer. Il faut donc définir ce que sont la programmation en général & les scripts en particulier.

La Programmation

C'est la définition d'une tâche exécutable par l'ordinateur.

Elle comprend deux phases : la définition de la tâche, stricto-sensu, on dit l'analyse, & sa traduction en une tâche exécutable par l'ordinateur on dit la programmation (au sens strict).

Cette distinction correspondait aux deux métiers du développement informatique, analyste & programmeur, aujourd'hui fusionnés. Cette disparition à trois origines :

- l'augmentation spectaculaire des performances des matériels & des logiciels ;
- l'augmentation tout aussi spectaculaire des compétences professionnelles des informaticiens ;
- & la manie contemporaine d'agir avant de réfléchir.

Le résultat de la phase d'analyse s'appelle un algorithme (synonyme de recette).Le résultat de la phase de programmation s'appelle un programme. Dans les deux cas, il s'agit de partir d'une situation donnée pour arriver à une situation finale en un nombre limité d'étapes.

Une fois l'algorithme écrit, (en français logique), il faut le traduire en un des langages de programmation existants [01002].

Jusqu'en l'an 2000, le plus usité des langages de programmation était le COBOL. Pour éviter le bug de l'an 2000, il a été remplacé par le langage C.

Les plupart des langages à la mode aujourd'hui descendent du C : C++, C#, Java, Perl, PHP [01003], etc.

La Programmation par l'exemple

Énoncé du problème : il est 6 heures du matin, vous êtes avachi sur la table de votre cuisine dans l'espoir de boire un café réalisé avec votre merveilleux percolateur, mais vous n'avez pas le courage de le faire (situation de départ).

Cette situation désagréable se produisant tous les jours, vous décidez d'y remédier en achetant un robot que vous programmerez afin de prendre votre petit-déjeuner au lit vers 6 heures (situation d'arrivée).

Vous rangez votre robot dans un coin de la cuisine.

Analyse de l'énoncé

Exprimons le problème en français précis, comme si nous devions expliquer l'opération à un demeuré :

1 À six heures, il faut s'activer & chercher le café.
2 S'il n'y en a pas, il ne reste plus qu'à pleurer (alternatives possibles réveiller les voisins, pour leur en demander, retourner se coucher).
3 S'il y en a, il faut sortir le café du placard.
4 Si c'est du café en grain alors il faut le moudre.

5 Il faut ensuite mettre de l'eau dans le réservoir de la cafetière, un filtre dans le porte-filtre.

6 Puis il faut mettre du café dans le porte filtre & allumer la cafetière.

7 Il faut ensuite patienter, puis verser le café dans une tasse & le boire.

En examinant le texte on constate qu'il comporte plusieurs éléments :

- des mots structurant la pensée (si, alors),
- des conditions (à six heures, s'il n'y en a pas, etc.),
- des verbes d'action (chercher, pleurer, moudre, verser, etc.),
- des objets constants (le placard, la cafetière & ses composants, le filtre),
- des objets variants ou pouvant varier (l'eau en quantité, le café en quantité & en qualité, éventuellement la tasse).

Nous allons maintenant, adapter le travail à faire au robot.

Notez la colorisation, elle sert à mettre en valeur les cinq éléments précédents.

Premier niveau, le tout

1. **S'il est 6 heures** alors
   - *activer le robot*,
   - *chercher* le café
2. **S'il n'y en a pas** alors
   - *pleurer* très fort.

Sinon
- *sortir* le paquet
- **s'il n'est pas moulu** alors
  - le *moudre*
- *mettre* de l'eau & du café dans le filtre de la cafetière électrique,
- *mettre en marche* la cafetière ;
- *sortir* le plateau, une *tasse*, une *soucoupe*, une *cuiller* & *des sucres* du placard ;
- *arranger le tout sur* le plateau ;
- *attendre* que le café *ait fini de passer* ;
- *verser* le café dans la tasse,
- *porter* le tout dans la chambre.

⁕ Les mots en rouge sont définis dans la grammaire du langage.

⁕ Les *mots en italique* sont des ordres que l'on appelle aussi des procédures ou des fonctions ; procédures & fonctions étant des instructions un peu plus complexes que celles définies à la base dans le langage du robot.

⁕ Les mots en violet sont des constantes : ni votre chambre, ni la cafetière, ni le plateau ne changent. En revanche, le sucre, le café, l'eau ne sont pas les mêmes, sauf si vous êtes coincés dans une improbable boucle temporelle. Vous pouvez, également, si vous n'êtes ni maniaque ni seul, changer de tasse de soucoupe & de cuiller.

⁕ Les **mots en gras** sont des tests représentant des alternatives : si le test est vrai, la première possibilité est exécutée sinon c'est la seconde. Ils peuvent être représentés par une

variable, une constante ou une relation entre plusieurs éléments à comparer ou à évaluer.

Les mots en vert sont des variables, des zones de la mémoire ayant un nom & une valeur [01004].

Dans ce cas précis se sont des objets physiques que l'on peut caractériser comme des objets informatiques :

- ils ont des *propriétés* (ou caractéristiques : contenance, vide ou pleine, forme, couleur, fragilité, etc.) ;
- on les utilisent pour des actions (remplir, vider, laver, ranger, etc. appelées *méthodes* en informatique) ;
- ils subissent des *évènements* : remplissage, vidage, lavage, etc.

L'application de la notion d'évènement à des objets physiques n'a pas grand sens, le lien entre action & événement étant fort. Il n'en est pas de même en informatique où les évènements sont liés à la manipulations de la souris ou à la frappe de touches. En informatiques, beaucoup de variables sont simples, c'est-à-dire seulement une zone mémoire avec un nom & un contenu.

Si le robot sait exécuter ces ordres, le programme est fini. Ce sera peut-être le cas en 2048 (l'an 2K), mais ce n'est pas le cas en 2015, donc, pour chaque procédure ou fonction, il faut définir ce qu'elle est.

Deuxième niveau : exemple d'une fonction

Mettre l'eau & le café dans la cafetière électrique

1. *ôter* le couvercle du réservoir d'eau,

2. *remplir* le réservoir de la quantité d'eau voulue (ici une tasse),
3. *replacer* le couvercle du réservoir,
4. *mettre* du café dans le filtre.

Ici encore, si le robot ne connaît pas ces mots, il va falloir détailler chaque action afin d'obtenir des mots & des phrases du langage parlé par le robot, ou facilement traduisibles.

Troisième niveau ; exemple de sous-procédure ou sous-fonction

Ôter le couvercle du réservoir d'eau :

1. - *placer* le bras à hauteur du couvercle,
2. - *écarter* les doigts,
3. - *avancer* le bras & la main,
4. - *resserrer* les doigts,
5. - *lever* le bras & la main,
6. - *tourner* le bras de 60 degrés,
7. - *descendre* le bras au niveau de la table,
8. - *écarter* les doigts,
9. - *relever* le bras [01005].

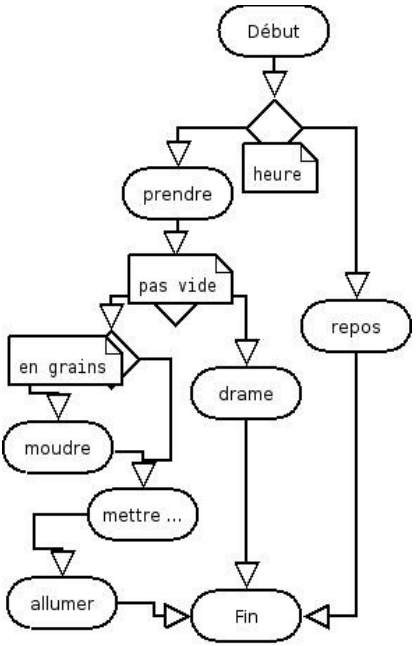
Quand arrête-t-on l'analyse ?

Quand on constate que tous les mots que l'on a écrit correspondent à des mots du langage du robot.

Selon les théoriciens de la programmation, dans un programme bien pensé, il ne devrait pas y avoir plus de quatre niveaux. Ici, le quatrième niveau consisterait à définir les mots

placer, *écarter*, etc. s'il ne font pas partie du vocabulaire du robot. En pratique, comme on peut bien penser sans être bien pensant, on fait ce qu'on peut !

Comme un langage de programmation contient peu de mots de base (moins de 300 en général) & que beaucoup de gens l'emploie, on trouve sur Internet des centaines ou des milliers de mots, ou, si vous préférez, de fonctions, supplémentaires, regroupés dans des fichiers que l'on appelle *bibliothèques* (*dll*, *lib*, etc.) au lieu de les appeler *dictionnaires* ! Ainsi Perl & PHP dépassent les dix mille mots. La difficulté & de savoir comment & quand les utiliser. À ce titre le manuel en ligne du PHP est exemplaire (www.php.net).



Un Programme bien pensé

Dans tous les cas, le programme sera contenu dans un fichier sur mémoire permanente, afin de ne pas perdre le travail effectué. Il sera composé d'un suite d'instructions, dans certains cas, ou d'une suite de fonctions, l'une d'entre elles

pouvant donner son nom au programme & constituant la procédure ou la fonction principale.

La décomposition en fonctions vise à faciliter l'écriture du programme. Elle facilitera sa lecture &, donc, sa maintenance.

À notre grande perplexité, certains trouvent le diagramme de droite plus clair que le texte ci-dessous. Ce type de diagramme, qui date de la préhistoire de la programmation, présente l'inconvénient de ne pas toujours représenter fidèlement ce que l'on écrira dans le script.

Voici une liste globale avec une présentation légèrement différente. À vous de retrouver les différentes catégories de mots.

Exemple 1 :

DÉBUT

- **fonction** aller chercher le café dans le placard … *(suit la définition de la fonction)*
- **fonction** moudre le café …
- **fonction** mettre de l'eau & du café dans la cafetière …
- **fonction** mettre l'eau dans le réservoir d'eau …
- **fonction** mettre le café dans le filtre …

….

- **fonction** drame …

Partie principale

..**RÉPÉTER**

…..**SI** heure = 6 & pas week-end **ALORS** aller chercher le café

dans le placard
…..**SI** paquet de café = pas encore vide **ALORS**
…….**SI** état = **en grains** **ALORS**
……..moudre le café
…….**FIN_SI**
…….mettre de l'eau & du café dans la cafetière
…….l'allumer
…….le porter dans la chambre
…..**FIN_ALORS**
…..**SINON**
…….drame
…..**FIN_SI**
….**FIN_SI**
..**JUSQU'À** la bonne heure
Fin programme

Notez que, malgré le **répéter**, tout ce qui suit le premier **si** ne s'exécute qu'une fois. En informatique, *attendre*, c'est répéter *ne rien faire*.

Ici, les expressions en mauve, contiennent un verbe explicitement ou implicitement, elles correspondent à des instructions ou à des fonctions. Les expressions en rouge indiquent des variables. En vert nous introduisons un nouveau concept, celui d'opérateur, symbole permettant de combiner des valeurs variables ou constantes (les opérandes) pour en obtenir une nouvelle. Les **expressions en noir** sont des mots structurants.

Les opérateurs peuvent être *unaires*, c'est-à-dire n'ayant qu'un opérande (signes + ou -, négation), *binaires* ayant deux opérandes (opérations arithmétiques, comparaisons, opération logiques –ou ou or, et ou and, pas ou non ou not), concaténation de chaînes de caractères (« bon » concaténé avec « jour » donne « bonjour »), ou même *ternaires*. Ils servent à calculer une expression, c'est-à-dire une combinaison de variables ou de constantes avec un ou plusieurs opérateurs. Nous y reviendrons.

Un Exemple concret : la bataille navale

Nous ne présenterons ici que l'analyse générale du script. L'adaptation au *bash* sera un des exercices du Cahier d'exercices.

Énoncé

Écrire un script de jeu, bataille_navale.sh. Comme nous sommes là pour travailler & non pour nous amuser (même si nous sommes convaincu que jouer pendant cinq minutes toutes les heures à des jeux de réflexions abstraits ou résoudre pendant le même laps des casse-tête, aide à améliorer la productivité intellectuelle), nous réduirons la taille du tableau à 3 lignes (notées de A à C) & 4 colonnes (notées de1 à 4) & nous n'y placerons que 2 bateaux, un d'une case & l'autre de deux (consécutives bien sûr).

Un tir sera les coordonnées de la case visée. Pour le bateau de deux cases, il faudra, donc, deux tirs pour le couler.

Lorsque les deux bateaux seront coulés, nous afficherons le message « Bravo ! Vous avez gagné ! »

Description de l'affichage

L'écran de départ ressemblera à :

Pour tirer une torpille, il vous faut indiquer les coordonnées de la case visée.
C3, par exemple, indique une case dans laquelle vous n'avez pas encore tiré,
++ un tir qui n'a rien touché,
*1 un tir qui contenait une cible.
Voici l'écran de départ.

```
-----------------
| A1 | A2 | A3 | A4 |
-----------------
| B1 | B2 | B3 | B4 |
-----------------
| C1 | C2 | C3 | C4 |
-----------------
```

Indiquez les coordonnées de votre cible :

Exemple d'affichage en fin de jeu :

```
-----------------
| A1 | ++ | ++ | ++ |
-----------------
| *2 | *2 | ++ | B4 |
-----------------
```

```
| *1 | ++ | C3 | ++ |
```

Bravo, vous avez gagné !!

Analyse niveau 1

Afficher règle
Afficher le plan d'eau de départ
Calculer la place des bateaux
Répéter
  Lire un coup & vérifier sa validité
  Afficher le résultat du tir dans la case
  Si un bateau a été touché alors
    si **le bateau coule** alors
      afficher coulé
    sinon
      afficher touché
  sinon
     afficher raté
  fin si
jusqu'à ce que tous les bateaux soient coulés
Afficher message victoire

Nous avons graissé les phrases demandant à être approfondies, que l'on peut considérer comme des fonctions potentielles. Potentielles parce que l'approfondissement de l'analyse montrera peut -être qu'il s'agit d'instructions mono-lignes

suffisamment claires (moins longues ou moins répétitives) pour qu'il ne soit pas indispensable d'en faire des fonctions.

De façon générale, dans un algorithme, les verbes & les phrases correspondent à des instructions, que celles-ci soient du langage (afficher se traduisant par une des instructions d'affichage du *bash*, dans notre cas : `echo`) ou ajoutées par nos soins ou par ceux d'autres programmeurs.

Un groupe nominal correspond à une variable (un bateau, le bateau, les bateaux → variable bateaux, un coup, du tir → tir, etc.), si sa valeur change, un nom sans article ou un adjectif, à une constante (message victoire, raté, touché, coulé, etc.) quand sa valeur ne change jamais.

Analyse niveau 2

C'est l'analyse des phrases précédemment graissées.

1. Afficher la règle c'est lister les chaînes de caractères suivantes :

   Vous devez à chaque tour indiquer les coordonnées de la case à atteindre.

   Il vous faut couler tous les bateaux pour gagner.

2. Afficher le plan d'eau de départ, c'est afficher les chaînes déjà présentées dans Description de l'affichage.

3. Calculer la place des deux bateaux de façon aléatoire, revient à tirer aléatoirement deux coordonnées (deux bateaux à placer) de A1 à C4 (douze cases dans le tableau apparent). Il faut en plus calculer une position pour la

deuxième case du second bateau. Il nous faut envisager de sauvegarder la position des bateaux.

4. Regarder si tous les bateaux sont coulés revient à compter le nombre de tirs aboutis. On sait qu'il faut autant de tir touchant que de cases de bateaux, ici trois.

5. Lire un coup ne pose pas de problème, mais il faut vérifier sa validité. D'une part, les coordonnées du tir indiqué doivent exister & d'autre part, éventuellement, le coup ne doit pas avoir été déjà joué (les torpilles & les obus coûtent chers !).

6. Afficher le dessin avec le résultat du coup dans la case, à chaque coup & redessiner le plan d'eau suivi du commentaire 'manqué', 'touché' ou 'coulé'.

7. Pour définir l'état d'un bateau il faut avoir sa position, ce que l'on a & son état ce que l'on a aussi & il faut faire ce travail pour chaque bateau.

Analyse niveau 3

C'est ici que nous allons aborder la représentation des données & définir les fonctions nécessaires.

L'usage est de définir une fonction par un nom suivi d'une liste éventuelle de paramètres & de l'ensemble des instructions permettant de la réaliser.

Représentation des données

Les données affichées, pour représenter le plan d'eau, sont les chaînes de caractères variables décrites dans l'énoncé que l'on

nommera Lig_A, Lig_B & Lig_C. Lig_Fix sera la chaîne fixe séparant les précédentes.

La manipulation de chaînes de caractères s'avérant plus complexe que celle des nombres, nous représenterons, en interne, le plan d'eau par un tableau (liste de valeurs repérables par leur rang) de nombres à deux dimensions, plan_d_eau[1,1] correspondant à 'A1' & plan_d_eau[3,4] à 'C4'.

Chaque case du tableau contiendra, donc un nombre représentant son contenu, par exemple, 0 quand elle contient les coordonnées, 1 pour le premier bateau, 2 pour le second & 3 pour un tir raté. Si une case de bateau est atteinte elle contiendra −1 pour le premier ou −2 pour le second (multiplication par −1), ou d'autres valeurs indiquant que le case a été atteinte (4, & 5, ajout de 3 ; 10 & 20, multiplication par 10 ; 732 & 813, valeurs arbitraires ; etc.), mais différentes de 0, 1, 2 & 3. La multiplication par 10 a notre préférence. Si nous devions envisager l'extension du plan d'eau, du nombre & des types de bateaux (jeu classique, dans une grille 10×10, avec 10 bateaux de 1 à 4 cases), elles nous permettrait, dans ce cas, quels que soient le nombre & la taille des différents bateaux, une condition d'arrêt unique : la somme des nombres contenus dans les cases de bateaux est un multiple de 10.

Il nous faudra une donnée constante contenant les positions des lettres de coordonnées dans les trois chaînes affichées :

positions ← [3, 8, 13, 18] ^{01104}.

Il nous faut, également, noter la liste des cases occupées par les bateaux, c'est-à-dire l'abscisse & l'ordonnée de chaque case de bateau. Cette liste bateaux contiendra [*(0,0), (0,0), (0,0)*] au départ Elle sera modifiée lors du calcul des cases occupées.

Chaque tir, saisi par le joueur est une chaîne variable de deux caractères indiquant la case visée, que nous nommerons coup.

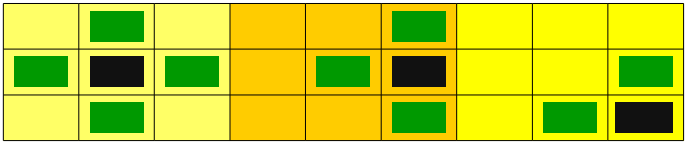
Analyse générale

Les deux premiers points *affichage de la règle* & *affichage du plan d'eau* sont une succession de commandes d'affichage.

1 Placement des bateaux

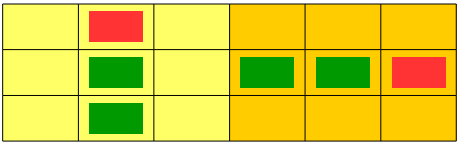
Les situations des bateaux diffèrent :

- pour le premier, il nous suffit de tirer au sort les coordonnées d'une case ;
- pour le second, il nous faut vérifier que la case tirée au sort n'est pas déjà occupée par le premier bateau & recommencer si c'est le cas, puis, que la seconde case, nécessairement orthogonale à la première, n'est pas occupée par le premier bateau & recommencer si c'est le cas.



Emplacements possibles de la seconde case

Si l'on pouvait éviter que les deux bateaux soient dans le prolongement l'un de l'autre ce serait un plus.



Placements à éviter si possible

À la réflexion il paraît plus simple de commencer à placer le bateau 2 car le placement de la seconde case est plus simple à traiter, alors que la complexité du placement du bateau 1 est identique à celle de placement de la case 1 du bateau 2 placé en second.

2 Fin de la partie

Il y a deux façons de déterminer le gain :

- compter le nombre de coup réussi, puisque nous savons qu'il y a trois cases à toucher ;
- nous placer dans une perspective d'évolution & compter plutôt le contenu des cases de bateaux, s'il est divisible par 10 alors la partie est gagnée & le jeu terminé !

La première façon est moins évolutive, mais elle fonctionne qu'elle que soit la méthode d'affectation de valeur en cas de coup au but ; le seconde ne fonctionne qu'avec la multiplication par 10, mais quel que soit le nombre & le style de bateaux.

3 Analyse d'un coup

Vérifier que les coordonnées existent sinon les redemander.

Il faut regarder dans le tableau plan d'eau ce que contient la case :

- *0* : le coup est raté, la case vaut maintenant 3
- *1* : le bateau 1 est touché & coulé la case vaut 10, il faut modifier la variable cases de bateaux,
- *2* : le bateau 2 est touché, la case vaut 20 & si l'autre case vaut 20, il est coulé ; il faut aussi modifier la variable cases de bateaux ;
- *3*, *10*, *20* : message d'erreur, car la case a déjà été visée.

4 Affichage du plan d'eau

Selon la valeur de la case du plan d'eau il faudra modifier la chaîne concernée, on remplacera les cordonnées par :

- *0*, *10*, *20* : rien ;
- *1* : '*1' ;
- *2* : '*2' ;
- *3* : '++'.

Il faut stocker les positions des coordonnées, dans notre cas :

- 3, 8, 13 & 18, si l'on numérote à partir de un ;
- 2, 7, 12 & 17 si l'on numérote à partir de zéro.

Arrivé à ces tade, pour pouvoir continuer notre analyse, il nousd faut connaître le langage de programmation *Bash*. C'est ce que nous allons aborder.

Qu'est-ce qu'un script bash ?

Rappels

Tout d'abord, ***bash*** est un un interpréteur de commandes, c'est-à-dire un programme capable de comprendre & d'exécuter un ensemble de commandes paramétrables. En d'autre termes c'est un environnement permettant de saisir des commandes qui seront exécutées & renverront un résultat. Comme il sert d'intermédiaire entre l'utilisateur & le système d'exploitation on dit que c'est un *shell*, une coquille qui entoure le *noyau* du système.

Lorsque vous ouvrez un terminal ou lorsque vous ne vous connectez pas à Linux en mode graphique, vous aboutissez sur le *shell*, probablement ***bash***, de votre système. Un message d'attente ou *prompt* apparaît alors vous signifiant que l'interpréteur attend que vous saisissiez une commande. Comme ce *prompt* est configurable, il varie d'une distribution à l'autre & vous pouvez l'adapter selon vos besoins. Il ressemble plus ou moins à :

[nom_de_connexion@nom_PC dossier]$.

En tapant une commande après ce *prompt*, si cette dernière est reconnue par le système, vous obtiendrez un résultat. Les commandes ***bash*** admettent généralement la même structure : le nom de la commande suivi parfois d'options (souvent précédées du caractère - ou de --) ou d'arguments, éventuellement des deux.

Par exemple, dans la commande ls -aild /home, ls est la commande affichant le contenu d'un dossier, -aild représentent ses options a, i, l, d, & /home son argument.

Un script shell est un ensemble de commandes contenues dans un fichier de façon à être exécutées plus ou moins, séquentiellement. En plus des commandes, le shell offre la possibilité d'utiliser des structures de contrôle qui permettront de gérer de manière précise l'exécution des commandes.

Quels outils utiliser ?

Pour écrire des scripts shell, qui sont donc des fichiers texte, sans enjolivement (police de caractère à espacement fixe, coloration syntaxique [01006] prédéfinie, cadrage à gauche des paragraphes, baptisés lignes), vous allez avoir besoin, bien sûr, d'un éditeur de texte, c'est-à-dire d'un traitement de texte simplifié, orienté programmation. Mais attention, *si vous n'avez jamais programmé*, certains éditeurs de texte sont plus pratiques que d'autres dans le cadre de l'écriture de lignes de code. Il est bien plus simple d'utiliser un éditeur possédant une coloration syntaxique des instructions du shell.

Il existe de nombreux éditeurs mettant en place cette fonctionnalité. Ce qui les différencie, c'est le nombre de langages qu'ils sont capables de reconnaître, leur capacité de personnalisation & leur difficulté d'emploi. Nous pouvons ici citer *gedit*, *kedit*, *bluefish* & *geany* qui sont très simples à employer. *A contrario*, *vim* ou *emacs* seront un peu plus difficiles à prendre en main

(mais ils offrent des possibilités bien supérieures). Tous ces éditeurs sont disponibles dans les dépôts des diverses distributions Linux & vous n'aurez aucun mal à les installer (toutes les distributions contiennent au moins un éditeur qui sera installé par défaut). Nous en employons deux : *vim* & *geany*.

VIM

Vim remplace *vi* depuis le début des années 2000. Il présente de nombreuses améliorations. Cet éditeur extrêmement performant est digne d'un *environnement de développement intégré* (*EDI*) *WHYSIWIG*, mais son ergonomie laisse à désirer. Il n'est pas nécessaire de connaître toutes ses subtilités pour rédiger des scripts courts.

Cependant, comme c'est pratiquement le seul éditeur systématiquement disponible avec tous les Unix & toutes les distributions Linux, il peut être utile de lire le tutoriel vimtutor [01007].

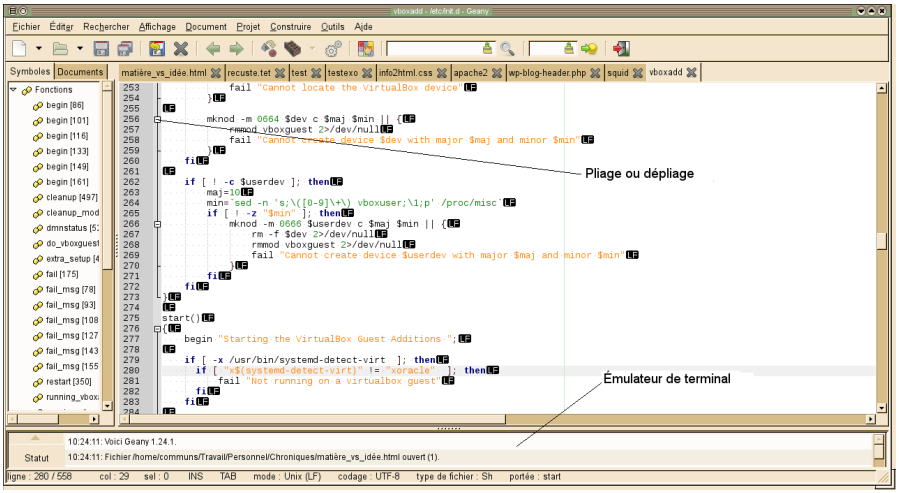
Voici un tableau résumant les commandes nécessaires à l'écriture de courts scripts

Commande	Effet
Ins	Mode insertion
Ins Ins	Mode remplacement
Esc [01007]	Retour au mode initial
Attention : les commandes qui suivent ne fonctionnent qu'en mode initial !	
← ↑ → ↓	Usage habituel

Commande	Effet
[intervalle]D	Efface la ligne ou les lignes de l'intervalle à partir de la position du curseur
[intervalle]d	Efface la ligne ou les lignes de l'intervalle
:	Mode commande
Sous-commandes du mode commande Elles doivent être précédées de la frappe de :.	
^	Début de ligne (AltGr 9)
$	Fin de ligne
/chaîne/	Recherche de chaîne
n	Reproduction de la recherche ver la fin du texte
N	Reproduction de la recherche ver la début du texte
wq ou x	Sauvagarde & sortie
w	Sauvegarde sans sortie
q!	Sortie sans sauvegarde
/modèle/	Recherche le modèle
s/anc/nouv/	Remplace anc par nouv

Pour certaines commandes comme *d* ou *D*, il est possible de spécifier un nombre de ligne, cela effacera autant de lignes que précisées à partir de la ligne courante : Si l'on est sur troisième ligne, *10d* efface les lignes 3 à 12.

Geany



Selon Wikipédia, ce logiciel d'Enrico Tröger (site http://www.geany.org/) est un éditeur de texte léger incluant les fonctions élémentaires d'un environnement de développement intégré. Il a peu de dépendances & démarre rapidement. Il est disponible pour plusieurs systèmes d'exploitation tel que Windows, Linux, Mac OS X, BSD & Solaris. Il supporte, entre autres, les langages C/C++, Java, JavaScript, PHP, HTML/CSS, Python, Perl, Ruby, Pascal & Haskell.

Geany ne vise pas la sophistication d'Eclipse. Il peut remplacer sous Windows des éditeurs tels que NoteTab ou ConTEXT.

C'est un logiciel libre sous licence GNU GPL.

Geany propose les fonctionnalités de base d'un environnement de développement intégré (EDI), afin d'être indépendant des autres logiciels :

- auto-complétion [01206],
- interface à documents multiples,
- gestion des projets,
- coloration syntaxique,
- pliage de code (permet de faire ressortir la structure du script en masquant les lignes comprises dans une itération ou dans une condition),
- liste de symboles,
- &, surtout, un émulateur de terminal intégré.

Il existe bien d'autres éditeurs. Si *bluefish* nous parait bien plus ergonomique que *geany*, en particulier pour développer des sites web, il lui manque le simulateur de terminal pour tester les scripts `bash`. En revanche, les *nano*, *gedit*, *kate* & autres, nous paraissent bien moins ergonomiques que *vim*, pour trois raisons :

- le libellé des commandes y semble aléatoire ;
- les menus confus, dans leur libellé & dans leur organisation ;
- le même travail y demande plus de saisies de commandes.

Attention : il ne s'agit que d'impressions. Cela ne veut pas dire qu'ils faut les éviter, mais qu'ils ne conviennent pas à notre façon de travailler.

Mise en œuvre

Pour exécuter un script, l'alternative consiste soit à lancer l'interpréteur de commande correspondant en lui donnant en paramètre le nom du fichier de script concerné, soit à indiquer directement dans le fichier quel est l'interpréteur de commande concerné & à affecter les droits d'exécution au fichier de script.

La première solution est la plus simple à mettre en œuvre. Une fois votre fichier tapé & enregistré, vous n'avez plus qu'à lancer la commande : **bash** premier_script ou **sh** premier_script (**sh** étant, aujourd'hui, un raccourci pour **bash**) si vous avez nommé votre script premier_script. Cette pratique a un inconvénient : on ne sait pas quel interpréteur employer *a priori*, d'où l'idée d'ajouter une extension en fin de nom pour l'indiquer, même si cela revient à taper deux fois la même information (le nom de l'interpréteur) à chaque exécution.

La seconde solution, qui évite cet inconvénient, consiste à ajouter l'extension .sh en fin de nom & à ajouter le droit d'exécution au fichier s'avère un tout petit peu plus longue à mettre en place, mais elle est plus sure !

Dans un premier temps, il faut indiquer sur la première ligne du fichier de script quel est l'interpréteur qui permettra d'exécuter ce fichier. Cette ligne possède une syntaxe particulière : les caractères #!, nommés *shabang* (contraction des noms anglais de ces deux caractères, *sharp* & *bang*, écrit parfois *shebang*), suivis du chemin absolu (depuis la racine) vers l'interpréteur. Dans le cas du ***bash***, la première ligne sera : *#!/bin/bash*. Il s'agit d'un commentaire spécial, qui, s'il n'est

pas exécutable, indique au shell quel interpréteur de script il faut utiliser, avec ce fichier.

Si vous utilisez un autre interpréteur, vous pourrez obtenir son chemin absolu en utilisant la commande which suivie de son nom.

Exemple 2 :

```
$ which perl
/usr/bin/perl
```

Dans un deuxième temps, il faudra rendre ce script exécutable avec la commande chmod appliquée au script, par exemple :

Exemple 3 :

```
$ ls -l test
-rw-r--r-- 1 mmichek users 61 Dec 30 21:27 test
$ chmod u+x test
$ ls -l test
-rwxr--r-- 1 mmichek users 61 Dec 30 21:27 test
```

Ces opérations effectuées, il ne vous reste alors plus qu'à taper le nom de votre fichier, dans un terminal pour l'exécuter.

Un Exemple de script bash

Les fichiers de configuration de votre shell sont des fichiers cachés. Ceux de *bash* se nomment .bashrc & .bash_profile, ils se trouvent dans votre répertoire personnel, ce sont des fichiers de script.

Coloration syntaxique

Afin de faciliter, la lecture des scripts, la plupart des éditeurs de textes permettent de colorer de différentes manières les mots d'un fichier de script, de façon à en faire ressortir la syntaxe. Voici par exemple le contenu du fichier .bash-profile du PC utilisé pour adapter ce texte, avec & sans coloration, dite syntaxique.

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
unset USERNAME
```

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs

PATH=$PATH:$HOME/bin

export PATH
unset USERNAME
```

Notons les lignes vides, elles ne servent qu'à faciliter la lecture ; sur un script si court, elles ne sont pas indispensables, sur des textes plus longs, elles sont absolument nécessaires.

Notons, encore, que les lignes commençant par le caractère « # » sont des commentaires : jamais exécutés, ils servent à faciliter la compréhension du script par leurs lecteurs. Ils sont indispensables, mais ils se doivent d'être judicieux ; celui de la première ligne ne l'est pas, contrairement aux deux suivants (Si vous ne connaissez pas le nom du fichier que vous êtes en train de lire, changez de métier !). L'effet du dièse est le même en milieu de ligne : tout ce qui suit est ignoré de l'interpréteur.

Nous reviendrons plus loin sur la signification des différents attributs. Les couleurs employées sont de notre cru ; *vim* l'éditeur de base en mode texte propose un autre ensemble de couleurs ; en fait chaque éditeur dispose de jeux de couleurs différents un pour chacun des langages de script qu'il connaît.

Premier script

Dans ce paragraphe, nous allons voir comment réaliser notre premier script. Personnellement, nous emploierons `geany`, ou `vim` si le premier n'est pas installé, pour saisir les différentes lignes du script, puis nous l'exécuterons.

Exécution d'un script

Nous avons déjà mentionné l'emploi du *shabang* pour indiquer le nom de l'interpréteur concerné. Cela ne suffit pas. Il faut encore donner le droit d'exécuter le fichier avec `chmod` & taper le nom du script précédé de ./ puisqu'il n'est pas dans un des dossiers définis dans la variable *PATH*.

En résumé, il y a trois façons de procéder

Exemple 4 :

```
chmod 755 monscript.sh
./monscript.sh
```

C'est le meilleur moyen puisque dans ce cas le script possédera son propre processus. L'exécution démarrera une session shell non interactive.

Remarque : *notez que monscript.sh est écrit de deux façons différentes,* la première *indique que c'est un paramètre de la commande chmod,* la seconde *que c'est devenu une commande externe. Cette chaîne de caractère pourrait être écrite sans ce document aussi bien* monscript.sh*, pour indiquer que c'est aussi un fichier sur le disque, ou, encore* monscript.sh *pour indiquer que dans le contexte il s'agit d'une chaîne de caractères.*

Exemple 5 :

```
sh monscript.sh
```

Dans ce cas on peut remplacer *sh* par un autre interpréteur en fonction du contenu du script, il faudrait, alors, aussi changer l'extension. Si l'on emploie `sh`, on exécute une session non interactive de *bash* ayant le script comme paramètre. Il n'y a pas de processus du script.

Bien que l'on puisse écrire `bash` à la place de `sh`, à la fois comme nom d'interpréteur & comme extension, l'usage & la paresse le déconseille !

Exemple 6 :

```
. monscript.sh
```

Celle-ci est réservée aux scripts *bash*, car . est une commande interne de ce logiciel. C'est un autre nom de la commande interne **source**, qui liste & exécute le contenu d'un script *bash*, ce qui explique qu'on l'emploie pour intégrer un fichier de fonctions ou pour exécuter une fonction , dans un script. Le script est exécuté dans la session shell en cours.

Un Petit « Bonjour »

Pour ce premier exemple, nous allons utiliser une des commandes les plus simples du shell : la commande d'affichage `echo` [01008] Cette commande affiche une chaîne de caractères sur le périphérique de sortie standard (l'écran par défaut).

Pour utiliser cette même commande dans un script, il faut ouvrir un fichier texte (que nous appellerons ici bonjour.sh) & y taper :

Exemple 7 :

```
1  #! /bin/bash
2
3  echo "Bonjour"
```

Nous insistons, mais, si l'on excepte le *shabang*, un commentaire ne sert pas forcément à un autre qu'à vous-même : nous oublions rapidement la signification d'une ligne de script absconse, les commentaires permettent d'une part de se remémorer la signification des lignes suivantes & de vérifier que les lignes de code correspondent à ce qu'on voulait obtenir, en cas de bogues !

Un commentaire peut finir une ligne au lieu de la commencer ; au lieu des deux dernières lignes nous aurions pu écrire :

Exemple 8 :

```
1  #!/bin/bash
2  echo "Bonjour" # Nous avons affiché un message !
```

Enregistrez le fichier sous le nom de bonjour.sh, changez ses droits pour le rendre exécutable & tapez son nom après le

message d'attente. En principe, un message d'erreur doit s'afficher, car votre dossier de travail n'est pas dans la variable PATH.

Pour réussir à l'exécuter il faut saisir :

```
./bonjour.sh
```

C'est votre première commande externe !

Pour réaliser des scripts utiles, il nous faut d'abord comprendre le langage de programmation de *bash* & comprendre comment trouver la méthode permettant d'arriver à écrire des scripts pas nécessairement optimisé, mais facile à écrire & surtout à maintenir.

Le Langage de programmation

Évidemment, si l'on ne pouvait écrire que des scripts de ce genre, cela ne présenterait aucun intérêt, mais pour en produire de plus complexes, il faut introduire deux notions importantes : celle de *données* & celle de *structure de contrôle*.

Comme leur nom l'indique, les données sont des informations données par le contexte &, de plus, nécessaires à la réalisation de la tâche objet du script. Ces données sont souvent constantes, comme le placard ou la cafetière dans le programme précédent : vous ne vous amusez ni à changer l'emplacement du paquet de café à chaque fois ni à changer de cafetière parce que vous n'en avez qu'une. D'autres informations ont des valeurs changeante : l'eau & le café ne sont jamais les mêmes puisque les quantités utilisées ont été ingérées & qu'elles changent avec le nombre de tasses souhaité. Ce sont ces données pouvant recevoir différentes valeurs que l'on nomme *variables*.

Quand nous cuisinons, même si nous sommes multi-tâches, nous essayons d'effectuer chaque instruction l'une derrière l'autre, en séquence. Pourtant, certaines actions entraînent une rupture de séquence : vous ne pouvez pas passer à la ligne suivante tant que vous n'avez pas terminé l'action en cours. Ainsi quand vous découpez une carotte en rondelle, vous effectuez une itération consistant à découper une rondelle de la carotte, jusqu'à ce qu'il ne reste plus de carotte à découper.

Les structures de contrôles permettent d'organiser l'exécution des instructions soit en choisissant certaines & pas d'autres soit en répétant d'autres ou en combinant ces deux éléments (choix & répétition) ;

Les Variables & les expressions

Une variable pourrait être représentée par une boîte avec une étiquette. Lorsque nous parlons du nom inscrit sur l'étiquette, nous faisons référence au contenu de ma boîte. Si nous plaçons une olive dans notre boîte & que nous appelons la boîte « *zimboum* », quand nous dirons « *Tiens, peux-tu me donner le zimboum ?* », tout le monde traduira instantanément par « *Tiens, peux-tu me donner l'olive de la boîte ?* Et si dans notre boîte nous plaçons deux olives, alors lorsque nous dirons « *Donne moi le zimboum !*», la traduction sera « *Donne moi les deux olives !*» : nous ne ferons plus référence à une olive, mais à deux, car notre boîte en contient maintenant deux.

Pour revenir dans un cadre un peu plus formel, une variable est définie par un identifiant (son nom, zimboum) & une valeur (son contenu, les olives). Comme le shell est sensible à la casse (différence majuscule/minuscule), il est d'usage de n'utiliser que des caractères minuscules pour nommer ses propres variables. Les noms en majuscules sont réservés aux variables d'environnement.

La principale opération relative aux variables consiste à leur donner une valeur. On parle d'*assignation* ou d'*affectation*.

Définir & utiliser une variable

La définition se fait de manière très naturelle, presque, comme en mathématiques [01009] :

```
nom_de_la_variable=valeur_de_la_variable
```

Le nom de la variable doit être le plus significatif possible : quantité_de_café est plus compréhensible que qdc, lui même un peu plus compréhensible que c ou q.

Faites attention à ne pas mettre d'espace entre la fin du nom de la variable & le signe égal.

La valeur pourra être un nombre entier, une lettre, une chaîne de caractères encadrée par des guillemets informatiques ", des apostrophes informatiques ' ou des accents graves ` (AltGr 7), etc. L'effet de ces symboles différé grandement, nous y reviendrons.

Pour utiliser la valeur contenue dans une variable, il faudra la faire précéder du caractère $. Ainsi, la variable an contenant 12, pour accéder à la valeur 12 de an, nous devrons écrire $an.

Reprenons notre exemple précédent où le texte à afficher sera cette fois placé dans une variable.

Exemple 9 :

```
1  #!/bin/bash
2
3  # Nous allons afficher un message !
4  msg = "Bonjour"
5  echo msg # erreur de frappe
6  echo $msg
```

Test

```
$ sh bonjour.sh
```

Résultats

```
$ msg
$ Bonjour
```

Ma première ligne n'est pas le résultat attendu !

Soit le texte placé après le premier echo suivi du même résultat que précédemment car, lors de l'exécution, $msg est remplacé par la valeur contenue dans la variable msg.

Il est ensuite possible d'exécuter des opérations avec les variables. Si les variables sont des entiers, on pourra utiliser les opérateurs arithmétiques classiques, plus % pour le reste de la division entière ^{01010} & ** pour la puissance. La syntaxe est alors un peu particulière & il y a, en première approximation, deux manières de faire :

- soit on encadre l'opération par $((...)) ce qui donne par exemple : ax=$((3+4)) ; les doubles parenthèses indiquent qu'il s'agit d'un expression arithmétique & non d'une chaîne de caractères ;
- soit on utilise la commande let "..." : let "ax=3+4".

Dans le cas des chaînes de caractères, pour réaliser une concaténation (coller des chaînes bout à bout), il suffit de mettre les variables côte à côte. Par exemple, si va="Linux" & vb=" Pratique",

alors, on peut écrire msg=vavb & la variable msg contiendra la valeur Linux Pratique.

Voici un tableau récapitulatif indiquant la valeur d'une variable après des opérations d'exemple.

Commande	Valeur de la variable var
var=3+4	3+4
let "var=3+4"	7
var=$((3+4))	7
va="Linux"	
var=$va" Pratique"	Linux Pratique

Tableau 1

Notez l'espace dans les guillemets !

Les Tableaux simples

Un tableau simple est une table à une colonne dans laquelle les différents éléments sont accessibles par leur rang : mois[3] désigne le quatrième élément de la variable tableau mois qui en contient 12 numérotés de 0 à 11. Pour accéder à la valeur de cet élément 'avril' il faut employer l'opérateur **${...}** utilisé pour les paramètres positionnels ayant un numéro plus grand que 9. En d'autres termes, la commande **echo** **${**mois[3]**}** affichera 'avril'.

L'initialisation pourra se faire élément par élément ou en une fois avec la syntaxe

mois=(janvier février mars avril mai juin juillet … décembre)

Notez qu'en *bash* une chaîne de caractère ne contenant pas de séparateur de mots n'a pas besoin d'être entre guillemets.

Notez aussi que c'est l'espace qui sépare les valeurs dans un tableau.

Pour obtenir la liste de tous les éléments d'un tableau on emploie la formule **${**tableau[*]**}** ou **${**tableau[@]**}** comme pour les paramètres positionnels.

Exemple 10 :

```
echo "Le septième mois de l'année est "${mois[6]}"."
```

Résultat

```
Le septième mois de l'année est juillet.
```

Les Tableaux associatifs

On appelle ainsi des tableaux dans lesquels les éléments ne sont pas repérés par un nombre à partir de 0, mais par une chaîne de caractères. Ils sont très employés en PHP, moins en *bash*, excepté dans un cas particulier : les tables de hachage. Ces dernières sont des tables dans lesquelles la valeur d'un élément est calculée à partir de la clé permettant de le repérer à l'aide d'utilitaires comme `md5sum`, `hsasum` ou `cksum`.

Exemple II :

```
$ hashcode["titi.txt"]=`shasum titi.txt`
$ echo ${hashcode["titi.txt"]}
```

Résultat

```
ec085b39941bb43076e41fc916d8382874809f1a
```

Cette valeur n'est pas une chaîne de caractères déterminée aléatoirement, comme pour un mot de passe, mais un nombre hexadécimal, exprimant la somme de contrôle (un nombre calculé de telle façon qu'aucun autre fichier puisse aboutir au même résultat) du fichier titi.txt !

Les Expressions

Les suites de caractère 3+4, $((3+4)), $va" Pratique" constituent des expressions. Une expression peut être numérique (sa valeur est un nombre), alphanumérique (sa valeur est un texte) ou logique (elle est vraie ou fausse). Les symboles comme « + » sont des opérateurs.

Les opérateurs sont :

TYPE	LISTE
numérique	+, -, *, /, %, **
assignation numérique	+=, -=, *=, /=, %=
opérateurs bit à bit	<<, <<=, >>, >>=, &, &=, |, |=, ~, !, ^

TYPE	LISTE
logique	&& (ET), || (OU), ! (NON)
paramètre	shift

Il n'existe pas en *bash* d'opérateurs de comparaison. En pratique, ce sont les options de la commande test qui les remplace. Cette commande peut s'exécuter de deux façons :

- soit en utilisant le mot test suivi de l'expression de comparaison ;
- soit en encadrant la comparaison par des crochets : […].

Les options de la commande test sont qualifiées d'opérateurs quand on les emploie entre les crochets carrés (dans ce dernier cas, il faut faire suivre le crochet ouvrant d'une espace & précéder le crochet fermant d'une autre.)

La valeur booléenne (vrai=1, faux=0) d'une expression logique ou de comparaison, calculée avec les commandes test ou expr, s'obtient par $((test expression)) ou par `expr expression`. Dans les commandes if, while, until, etc. c'est la valeur de retour (vrai=0, faux=1) de la commande, contenue dans le paramètre spécial $? qui est employée. Si vous utilisez l'*algèbre booléenne* cela est déterminant !

Les Options/opérateurs de la commande test

Ils sont de quatre types.

1. Tests sur les objets du système de fichiers

[-e $FICHIER]	vrai si l'objet désigné par $FICHIER existe dans le répertoire courant,
[-s $FICHIER]	vrai si l'objet désigné par $FICHIER existe dans le répertoire courant & si sa taille est supérieure à zéro,
[-f $FICHIER]	vrai si l'objet désigné par $FICHIER est un fichier dans le répertoire courant,
[-r $FICHIER]	vrai si l'objet désigné par $FICHIER est un fichier lisible dans le répertoire courant,
[-w $FICHIER]	vrai si l'objet désigné par $FICHIER est un fichier inscriptible dans le répertoire courant,

[-x $FICHIER]	vrai si l'objet désigné par $FICHIER est un fichier exécutable dans le répertoire courant,
[-d $FICHIER]	vrai si l'objet désigné par $FICHIER est un répertoire dans le répertoire courant.

Exemple 12 :

```
$ fichier="titi.txt"; if [ -f $fichier ]; then echo $fichier; fi
```

2. Tests sur les chaînes de caractères

[c1 != c2]	vrai si c1 & c2 sont différents,
[c1 = c2]	vrai si c1 & c2 sont égaux,
[-z c]	vrai si c est la chaîne vide (Zero),
[-n c]	vrai si c n'est pas la chaîne vide (Non zero).

Exemple 13 :

```
$ c1="a"; c2="b"; if [ $c1 = $c2 ]; then echo $c1; else echo $c2; fi
```

3. Tests sur les nombres

[n1 -eq n2]	vrai si n1 & n2 sont égaux (EQual),
[n1 -ne n2]	vrai si n1 & n2 sont différents (Not Equal),
[n1 -lt n2]	vrai si n1 est strictement inférieur à n2 (Less Than),
[n1 -le n2]	vrai si n1 est inférieur ou égal à n2 (Less or Equal),
[n1 -gt n2]	vrai si n1 est strictement supérieur à n2 (Greater Than),
[n1 -ge n2]	vrai si n1 est supérieur ou égal à n2 (Greater or Equal).

Exemple 14 :

```
$ let n1=1; let n2=2; if [ $n1 -eq $n2 ]; then echo $n1; else echo $n2; fi
```

4. Tests logiques

[! a]	vrai si a est faux. ! est la négation.
[a1 -a a2]	vrai si a1 & a2 sont vrais. C'est le et logique (and).
[a1 -o a2]	vrai si a1 ou a2 est vrai. C'est le ou logique (or).

Exemple 15 :

```
$ if [ $nb2 -gt 10 -a $nb2 -lt 100 ]; then echo vrai ; else echo faux ;
```

Les Opérateurs alphanumériques de bash

La commande `expr` fournit toute une batterie d'opérateurs sur les chaînes de caractères, mais il en existe quelques-uns forts pratiques, inclus dans le *bash* directement, en voici la liste.

${#chaine}	Donne la longueur de la variable chaine.
${chaine:position}	Extrait une sous-chaîne de chaine à partir de la position position
${chaine:position:longueur}	Extrait longueur caractères d'une sous-chaîne de chaine à la position position
${chaine#souschaine}	Supprime la correspondance de la plus petite souschaine à partir du début de chaine [0110].
${chaine##souschaine}	Supprime la correspondance de la plus grande souschaine à partir du début de chaine.
${chaine%souschaine}	Supprime la plus petite correspondance souschaine à partir de la fin de chaine.
${chaine%%souschaine}	Supprime la plus grande correspondance souschaine à partir de la fin de chaine.
${chaine/souschaine/remplacement}	Remplace la première correspondance de souschaine par remplacement.
${chaine//souschaine/remplacement}	Remplace toutes les correspondances de souschaine avec remplacement.

Si remplacement est vide, souschaine est supprimée.

Exemple 16 :

Si chaine contient a1234567890ABCDEFazertyCDEBILE813-666666Cz.

```
$ echo ${#chaine}41
$ echo ${chaine:1:16}   1234567890ABCDEF
```

Si chaine contient 123456789ABCDEFazertyCDEBILE813-666666C.

```
$ echo ${chaine#12345678}     9ABCDEFazertyCDEBILE813-
666666C
$ echo ${chaine##1*8}          13-666666C
```

Si chaine contient a123456789ABCDEFazertyCDEBILE813-666666.

```
$ echo ${chaine%1*6}   a123456789ABCDEFazertyCDEBILE8
$ echo ${chaine%%1*6}  a
```

Si chaine contient ABCDEFazertyCDEBILE.

```
$ echo ${chaine/DE/HA}      ABCHAFazertyCDEBILE8
$ echo ${chaine//DE/HA}     ABCHAFazertyCHABILE8
$ echo ${chaine/DE/}        ABCFazertyCBILE8
```

Les Opérateurs de la commande expr

Ceux de la commande `expr` sont relativement normalisés, mais un peu plus lourds à mettre en œuvre.

Opérateurs	Signification
Opérateurs arithmétiques	
nb1 **+** *nb2*	addition
nb1 **-** *nb2*	soustraction
nb1 ***** *nb2*	multiplication
nb1 **/** *nb2*	division
nb1 **%** *nb2*	modulo (reste de la division entière)
Opérateurs de comparaison	
val1 **>** *val2*	vrai si *val1* est strictement supérieur à *val2*
val1 **>=** *val2*	vrai si *val1* est supérieur ou égal à *val2*
val1 **<** *val2*	vrai si *val1* est strictement inférieur à *val2*
val1 **<=** *val2*	vrai si *val1* est inférieur ou égal à *val2*
val1 **=** *val2*	vrai si *val1* est égal à *val2*
val1 **!=** *val2*	vrai si *val1* est différent de *val2*
Opérateurs logiques	
chaine1 **&** *chaine2*	vrai si les 2 chaînes sont vraies
chaine1 **|** *chaine2*	vrai si l'une des 2 chaînes est vraie
Opérateurs alphanumériques	
chaine **:** *expr_rat* **match** *chaîne expr_rat*	cherche une correspondance du modèle *expr_rat* dans chaine

Opérateurs	Signification
substr *chaine pos long*	sous-chaîne de *chaine* débutant à la position *pos* (comptée à partir de 1) de longueur *long*
index *chaine car*	valeur de la position du premier caractère car trouvé dans *chaine*, sinon 0
length *chaine*	longueur de *chaine*
+ *mot*	interpréter le *mot* comme une chaîne, même si c'est un opérateur comme match ou /
Opérateurs divers	
-*nb1*	Opposé de nb1
(*expression*)	Regroupement

Les opérandes de l'expression à évaluer doivent toujours être séparés par au moins une espace ou une tabulation.

Exemple 17 :

```
`expr 11 % 3`          2
`expr 11 \> 3`         0 (booléen) ou 1 (statut)
`expr substr AZERTY 3 2` ER
```

Enfin les commandes awk (données structurées en colonnes) & sed (données structurées en lignes) permettent d'effectuer des traitements sophistiqués sur les chaînes de caractères.

Exemple 18 : sed

```
for fichier in * ; do
  nom_incorrect=`echo "$fichier"|sed -n /[\+\{\:\"\\\=\?~\(\)\<\>\&\*\|\$]/p`
  echo $nom_incorrect
done
```

Résultat

```
ti*ti
```

Exemple 19 : awk

```
$ nb_lignes=`wc testexo | awk '{ print $1 }'`   36
```

Si vous le souhaitez, plongez vous dans les manuels de ces commandes pour plus de précisions. À titre d'information le dossier

/etc/init.d de notre PC contient cinquante-quatre scripts de démarrages dont treize emploient **sed** & deux, **awk**.

Les Variables du système

Elles sont de deux sortes, les *variables spéciales* & celles d'*environnement*.

* Les variables d'environnement comportent des variables crées par l'interpréteur de commande (*SHELL*, *USER*, *PATH*, *HOME*, *PSI*, etc.) & d'autres par certains logiciels installés (*QTDIR*, *GTK_MODULES*, etc.). Il faut réfléchir à deux fois avant d'en modifier la valeur.
* La valeur des variables spéciales est calculée par l'interpréteur de commande ou fournie par l'utilisateur, pour celles dites paramètres positionnels. Ce sont les suivantes.

Variable	Description
$0	Le nom du script (dans l'exemple précédent, sa valeur est ./bonjour si le script est appelé depuis son répertoire de stockage).
$1, ... $9, ${10}, ...	Les arguments passés au script : $1 est le premier argument, etc. On les nomme paramètres positionnels.
$*	La liste de tous les arguments passés au script (donc, à partir de $1), séparés par un espace. Cette liste est une chaîne unique !
$@	La liste de tous les arguments passés au script, séparés par un espace, comme précédemment. La nuance est que si l'on place cette variable entre guillemets, les paramètres sont considérés comme des mots séparés, exactement comme les paramètres de la ligne de commande.
$#	Le nombre d'arguments passés au script.
$?	Le code de retour de la dernière commande exécutée. Toutes les commandes shell renvoient une valeur : 0 lorsque la commande s'est exécutée correctement & une valeur d'erreur sinon. Par exemple, après un appel à **ls**, $? contiendra 0. En revanche, s'il n'y a pas de dossier /zimboum, après **ls** /zimboum, $? contiendra la valeur 2, valeur fournie par la commande ls.
$!	Le numéro de processus de la dernière commande lancée en tâche de fond.
$$	Le numéro de processus du script lui-même.

Tableau 2

Outre ces variables prédéfinies, vous pouvez créer les variables dont vous avez besoin. Cela vous permettra d'écrire des scripts plus intéressants puisque l'utilisateur pourra transmettre des données au script sans pour autant avoir à le modifier, puisque le script lui-même pourra modifier ces valeurs. Voici un exemple simple d'utilisation de quelques-unes de ces variables. Les deux premières lignes seront désormais omises, mais il ne vous faudra pas les oublier.

Exemple 20 :

```
1  echo "Le script s'appelle " $0
2  echo "Il y a" $# " arguments."
3  echo "La liste de ces arguments est : "
4  echo $*
5  echo "Le premier argument est : " $1
6  echo "Le deuxième argument est : " $2
```

Nommez le script variables.sh, changez en les droits, pour le rendre exécutable & passez lui comme arguments les chaînes indiquées en bleu ; vous obtiendrez, alors, l'affichage suivant :

```
$ ./variables.sh "toot" '3+4' azerty
Le script s'appelle ./variables.sh
Il y a 3 arguments.
La liste de ces arguments est :
toot 3+4 azerty
Le premier argument est : toot
Le deuxième argument est : 3+4
```

Si vous tapez `sh` au lieu de `./`, la première ligne deviendra `Le script s'appelle variables.sh`.

Vous pouvez, également, faire l'exercice Ex. 0 p. 120 à l'exception du § Expression logique.

Ici, l'utilisateur transmet des données au lancement du script. Mais il est possible qu'une information soit requise par le script au milieu de son traitement. Il faut alors utiliser une commande pour permettre à l'utilisateur de saisir les données requises. C'est le rôle de la commande **read**.

Elles sont de deux sortes, celles internes sous divisées en :

- commandes séquentielles, s'exécutant les unes derrière les autres & en structures de contrôle
- & celles externes que l'on peut regrouper en
  * commandes d'informations (**man**, **info**, **which**, **ping**, etc.),
  * commandes d'action sur le système (services & commandes systèmes),
  * commandes de manipulation de flux de données (**grep**, **cut**, etc.) & de données (**expr**, éditeurs, interpréteurs, etc.),

qu'elles soient en mode texte ou en mode graphique.

Les Commandes séquentielles

Elles sont nombreuses, mais, pour l'instant, seules les commandes **echo**, **read** & **test** nous intéresserons.

echo

Deux options nous concernent :

- **-n** qui évite le retour à la ligne après l'affichage (cf exercice 1) ;
- **-e** qui permet de gérer les caractères d'échappement, ce qui est pratique pour afficher des caractères non disponibles au clavier.

read

Elle permet de lire des données au clavier & de les stocker dans une variable dont le nom est spécifié à la suite de la commande : read nom_variable. La variable contenant la saisie de l'utilisateur sera alors employée comme n'importe quelle variable en la préfixant par le caractère $ pour accéder à sa valeur.

Voici un exemple de script bonjour utilisant la commande **read** :

Exemple 21 :

```
1  echo "Comment vous appelez-vous ? "
2  read nom
3  echo "Bonjour " $nom
```

En ligne 2, l'exécution du script sera suspendue jusqu'à ce que l'utilisateur ait saisi un texte validé par la touche ↵. Le texte sera alors stocké dans la variable nom & l'exécution du script reprendra en ligne 3 pour afficher le message.

Deux options de `read` nous intéresseront à ce stade : `-n` & `-p`.

La première permet de définir le nombre de caractères à saisir. Quand il est atteint la saisie s'achève sans avoir à taper ↵.

La seconde permet d'afficher un message avant d'attendre la saisie sur la même ligne.

Exemple 22 :

```
read -n 1 -p "Aimez-vous le chocolat (oui/non) ?" reponse
echo $reponse
```

La commande `read` permet également de lire plusieurs variables : il suffit de spécifier plusieurs noms de variables à la suite de l'appel à la commande &, lors de la saisie de l'utilisateur, le caractère ␣ (espace) sera utilisé pour délimiter les valeurs. En modifiant l'exemple précédent, on obtiendrait :

Exemple 23 :

```
1 echo "Comment vous appelez-vous (prénom nom) ?"
2 read prenom nom
3 echo "Votre prénom : " $prenom
4 echo "Votre nom : " $nom
```

Lors de l'exécution, si vous saisissez deux mots, le premier sera stocké dans `prenom` & le second dans `nom`.

Attention : si vous saisissez plus de deux mots, le premier sera bien stocké dans `prenom`, mais tous les autres seront stockés dans `nom`.

Exemple 24 :

```
$ ./nom.sh
$Comment vous appelez-vous (prenom nom)? Charles attend le train
$Votre prénom : Charles
$Votre nom : attend le train
```

Enfin, la commande `read` peut être utilisée pour réaliser une pause dans un programme en demandant à l'utilisateur d'appuyer sur ↵ pour continuer. Il s'agit en fait de l'usage classique de la commande, mais comme on ne souhaite pas récupérer la saisie dans une variable, on ne précise pas de nom de variable.

Exemple 25 :

```
echo
read -p "Appuyer sur [Entrée] pour continuer..."
```

test ou [...]

Cette commande déjà abordée dans le paragraphe Variables & expressions, remplace l'absence d'opérateurs de comparaisons [01210]. Ces options peuvent être, éventuellement, combinées avec les opérateurs logiques **&&**, **||** ou **!**. Ils s'emploient aussi bien avec des instructions conditionnelles qu'avec des instructions itératives.

Les Structures de contrôle

Les instructions dans un script sont exécutées une ligne après l'autre. Il est, parfois nécessaire de ne pas exécuter certaines lignes dans certaines situations ou d'en exécuter d'autres seulement dans un contexte précis. Il s'avère, également, souvent indispensable de ré-exécuter des instructions, puisque la raison d'être de la programmation s'avère de faire exécuter les tâches répétitives par l'ordinateur !

La détermination du contexte se fait au moyen d'un test, c'est-à-dire, généralement, de la comparaison de la valeur d'une variable & d'une expression.

Enfin, outre les variables (noms communs), il est parfois nécessaire d'ajouter des verbes au langage. Ces nouveaux mots sont nommés *fonctions*, ils ne sont inclus dans le lexique de la langue que durant l'existence du script, un peu comme les mots des jargons qui n'existent que pour les groupes les pratiquant.

Ces structures de contrôle ne sont pas spécifiques au *bash*, elles existent sous des formes voisines en *PHP*, en *Perl*, en *Python*, etc.

Dans tous les exemples qui suivent les espaces en début de lignes ne servent qu'à faciliter la lecture du script. Cette indentation est chaudement recommandée, même si elle n'est pas obligatoire. En effet, tous ces scripts pourraient s'écrire sur une seule ligne illisible.

***Rappel** : Afin de gagner de la place les deux premières lignes seront systématiquement sautées, mais il vous faudra les réintroduire, dans votre fichier. Les voici centrées & encadrées :*

#!/bin/bash

Les instructions conditionnelles

Nous avons pu voir précédemment comment construire des scripts rudimentaires & interagir avec le shell. Nous pouvons manipuler des variables, mais nous ne pouvons pas encore décider que faire en fonction de la valeur d'une variable. Par exemple, si nous posons une simple question « *Voulez-vous continuer (O/N) ?* », comment faire comprendre au script qu'il doit effectuer une action si l'utilisateur a saisi « O » & une autre s'il a saisi « N » ? Nous avons besoin des structures conditionnelles :

si telle condition est vraie (ou vérifiée) alors
  exécutez telle action
sinon éventuellement
  exécutez telle autre action.

Cela est rendu possible grâce à l'instruction if.

1 if

L'instruction if qui se traduit par « *si* » s'écrit de la manière suivante :

if condition_1; **then**
  instructions vrai1…
[elif condition_2; **then**
  instructions vrai2…**]**
  [else
  instructions faux… etc.**]**
fi

Les mots *condition_i* doivent être remplacés par les tests nécessaires.

Les crochets carrés indiquent que leur contenu est facultatif. La fin de l'instruction if est signalée par le mot *fi*. Le ; après la condition indique sa fin. On peut imbriquer les structures if. Grâce au mot *elif*, contraction des mots *else* & *if* consécutifs.

Exemple 26 :

```
1  echo "Répondez par oui ou par non à la question suivante !"
2  echo "Connaissez-vous la réponse à cette question ? "
3  read reponse
4
5  if [ $reponse = "oui" ] 01310; then
6    echo "Bravo !"
7  elif [ $reponse = "non" ]; then
8      echo "Ignare !"
9    else
10     echo "Répondez par oui ou par non, SVP !"
11 fi
```

Vous pouvez dès maintenant faire les exercices Ex. 0-Expression logique p. 120 & Ex. 2 p. 125 du Cahier d'exercices.

2 case

Cette instruction remplace, plus efficacement, des if imbriqués qui testeraient différentes valeurs d'une variable.

Sa structure est :

```
case contenu d'une variable in # selon … dans
  val1) bloc d'instruction;
  …
  [valn) bloc d'instructions;]
  [*) bloc d'instruction]
esac
```

Un bloc d'instruction est une liste d'instruction terminée par un ; par conséquent, la dernière instruction du bloc est suivie de 2 points-virgules.

L'option « * » désigne les autres cas possibles, elle sert, souvent, à traiter les erreurs de saisie. Il pourrait n'y avoir qu'une seule valeur traitée dans le case, mais cela n'apporterait rien par rapport à l'instruction if.

Exemple 27 :

```
1  case $choix in
2    1) echo "un";;
3    2) echo "deux";;
```

```
4     3) echo "trois";;
5     4) echo "soleil";;
6     *) echo "ERREUR !"
7   esac
```

En le modifiant légèrement, on peut employer ce script pour tester la parité, comme deux nombres sont impairs & deux autres pairs, il faudrait écrire deux fois le même message ; on peut éviter cela en employant l'opérateur ou noté |.

Remarque : si bash était rigoureux, il faudrait un ; à la fin du dernier bloc d'instruction.

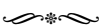
Exemple 28 :

```
1   case $choix in
2     1 | 3) echo "impair";;
3     2 | 4) echo "pair";;
4     *) echo "ERREUR !"
5   esac
```


Les instructions itératives

Dans les deux exemples précédents, vous avez probablement trouvé agaçant de devoir relancer le script pour tester chaque possibilité. Ces instructions permettent d'éviter cet inconvénient, mais pas seulement.

Le mot *itération* est compliqué, c'est pourquoi on parle plus volontiers de *boucle*. Le *bash* propose quatre types de structure itérative, deux générales, avec un nombre de boucles inconnu, s'exécutant l'une, tant que la condition de démarrage est vraie & l'autre, tant que la condition d'arrêt est fausse, une pour le cas où le nombre de boucles est connu ou déterminable & une dernière réservée aux menus en mode texte.



1 while … do … done

Elle commence par le mot while (*tant que*) suivi d'une condition terminée par un « ; ». Les instructions à répéter sont comprises entre les mots do (*faire*) & done (*fait*)(il en sera de même pour les autres instructions d'itération).

Exemple 29 :

```
reponse="o"
while [ $reponse = "o" ]; do # tant que … répéter
  echo "Bravo !"
  echo "Faut-il continuer (o/n)? "
  read reponse
done # fait
```

2 until …do … done

Cette instruction ressemble beaucoup à la précédente comme le montre l'exemple suivant, pourtant le fonctionnement diffère.

```
reponse="o"
until [ $reponse != "o" ]; do # jusqu'à ce que … répéter
  echo "Bravo !"
  echo "Faut-il continuer (o/n)? "
  read reponse
done # fait
```

Elle s'avère utile quand on ne peut pas modifier la valeur initiale ou quand on a besoin d'une condition négative.

Attention : contrairement à ce qui ce fait dans la plupart des autres langages, la condition suivant **until** *est testée avant l'itération. De ce fait, le résultat est le même avec une condition & sa négation.*

Exemple 30 :

```
let limite=2
let compteur=3
while [ $compteur -le $limite ]; do
  echo "boucle while compteur="$compteur
  let compteur+=1
done
let compteur=3
echo 'négation de $compteur -le $limite'
until [ $compteur -gt $limite ]; do
  echo "boucle until compteur="$compteur
```

```
    let compteur+=1
done
```
Résultat

négation de $compteur -le $limite

Aucune des deux boucles n'a été exécutée ! En C l'itération **until** *aurait été exécutée une fois puisque le test s'y fait après l'itération.*

3 for ...do ... done

Les cas, où le nombre de boucles est calculable, sont le parcours d'une liste de valeurs fixes (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche) d'une liste de longueur variable (liste des fichiers contenus dans un dossier) ou l'atteinte d'un borne numérique. Il existe donc deux formes d'instruction **for** l'une pour le parcours de liste, l'autre pour l'atteinte d'une limite. En fait, il en existe une troisième forme qui s'applique dans de moins en moins de cas.

Le parcours d'une liste est simple, il suffit d'indiquer la liste après le mot **in**. La liste peut être présentée extensivement comme dans "lundi" "mardi" "mercredi" "jeudi" "vendredi" ou compréhensivement comme dans « les jours ouvrés » ou dans `ls` qui fournit la liste des fichiers du répertoire dans lequel s'exécute le script.

L'atteinte d'une limite de fait au moyen de l'opérateur d'intervalle **..**. Ainsi, 2..1000..2 atteindre 1000 en démarrant à 2 & en comptant de 2 en 2 & 2..1000 fera le même chose de 1 en 1.

La troisième forme peut, le plus souvent être remplacée par la seconde, elle s'inspire de la syntaxe du lange C & ressemble beaucoup à une boucle **while**, car la condition s'écrit

((valeur_initiale;valeur finale;incrément))

soit concrètement ((cpt=2;cpt<=1000;cpt+=2)). La logique du *bash* voudrait que l'on écrive ((cpt=2;$cpt -le 1000;cpt+=2)). C'est probablement pour normaliser le langage que la seconde forme a été introduite dans la version 3 de *bash*, afin de remplacer la troisième.

```
1  # première forme extensive
2  for jour in "samedi" "dimanche"; do
3    echo "Bravo !"
```

```
done
# première forme compréhensive
# dans ce for, il n'y a pas de « ; » car le mot do est à la ligne suivante
for fichier in `ls ~` # Attention il s'agit des accents graves (AltGr 7). ~ est un raccourci pour le nom du dossier de connexion.
do
  echo $fichier
done
# deuxième forme
for nb in {1..10..2}; do
  echo $nb
done
# troisième forme à proscrire
for (( nb=1;nb<=10;nb+=2 )) do
  echo $nb
done
```

Toutes ces instructions permettent de définir des boucles infinies avec une condition toujours vraie ou toujours fausse.

4 select ...do ... done

Elle affiche un menu, avec une option par ligne. Il faudra inclure entre do & done, en général au moyen d'une instruction case, les traitements à effectuer selon le choix. Il faut taper le numéro affiché en début de ligne pour obtenir la valeur correspondante de la variable.

```
select choix in "Entrée 1" "Entrée 2" ; do
  echo "Vous avez choisi" $choix
done
```

Vous avez constaté que vous ne pouviez arrêter votre script. Il faut pour cela employer une instruction de rupture.

Les Instructions de rupture

Elles sont deux, une pour interrompre complètement la boucle, l'autre pourra atteindre directement la fin de l'itération en cours.

1 break

Elle sort de l'itération, permettant ainsi d'éviter les boucles infinies avec une instruction select par exemple.

```
1  select choix in "Bonjour " "Salut" "Fin" ; do
2    case ${choix :0 :1} in
3     "B") echo "Vous avez choisi" $choix;;
4     "S") echo "Vous avez choisi" $choix;;
5     "F") echo "Vous avez choisi" $choix;;
6          break ;;
7    esac
8  done
```

2 continue

Elle permet d'empêcher l'exécution d'une suite d'instruction.

```
1  # Affiche les nombres de 1 à 20 (mais pas 3 et 11).
2  nb=0
3  while [ $nb -le 20 ]; do
4    let nb+=1
5    if [ "$nb" -eq 3 ] || [ "$nb" -eq 11 ];. then
6      continue  # Continue avec une nouvelle itération de la boucle.
7    fi
8    echo -n "$nb " # Ceci ne s'exécutera pas pour 3 et 11.
9  done
```

Les Fonctions

On nomme fonction un ensemble d'instructions nommé, remplissant une fonction précise (calculer la valeur d'une variable, effectuer un traitement particulier sur des données). Le remplacement de plusieurs lignes par une seule vise à faciliter l'écriture & la lecture du script, en morcelant sa complexité. Tout comme les fonctions mathématiques, elles peuvent avoir un ou plusieurs arguments.

Pour définir une fonction, on emploie la syntaxe suivante :

```
function nom_de_fonction
```
ou
```
nom_de_fonction()
```
suivi de
```
{
  instructions
}
```

Contrairement à ce qui existe dans certains langages, les paramètres ne sont pas définis entre les parenthèses suivant le nom : ils apparaissent dans les instructions & sont nommés $1, $2, etc. C'est une source de confusion, lors de la mise au point, puisqu'il ne s'agit plus des paramètres du script, mais de ceux de la fonction. Les variables spéciales ont le même nom que pour le script.

Exemple 31 :

```
1  function hello
2  {
3    # Fonction affichant un message de bienvenue
4    # $1 : Prénom de la personne à saluer
5    # $2 : Nom de la personne à saluer
6    if [ $# -ne 2 ]; then
7      echo "Usage : hello prénom nom !"
8      exit 1 ; ## la commande exit fournit le code d'erreur du script, une valeur comprise entre 0 & 255, 0 signifiant la réussite du script. Cette valeur est stockée dans la variable « $? ».
9    else
10     echo "Bonjour "$1" "$2 # Nous aurions pu écrire "Bonjour $1 $2"
11   fi
12 }
13 hello "Michel" "Scifo"
14 echo "Numéro d'erreur : "$?
```

Il s'agit d'une fonction effectuant un traitement. Quand la fonction calcule une valeur, on peut l'utiliser dans une expression. Une

fonction bash ne peut calculer qu'une valeur & elle la transmet par une commande echo.

Exemple 32 :

```
1  double()
2  {
3    echo $(($1 * 2)); ## $1 est le paramètre de la fonction
4  }
5
6  resultat=$(double $1) ; ## $1 est le paramètre du script
7
8  echo "2 * $1 = "$resultat
```

Il existe une commande return qui affecte une valeur à la variable spéciale $?, elle sert, prioritairement à indiquer un numéro d'erreur (exit est préférable) &, éventuellement & abusivement, à donner une valeur réutilisable à la fonction.

Exemple récapitulatif 1

Énoncé

Copier les six plus gros fichiers cachés dans le dossier travail en les *décachant*.

Première étape : que faut-il faire ?

Il faut :

lister les fichiers cachés ;
les trier par taille décroissante ;
récupérer les noms des six plus gros ;
si le dossier travail n'existe pas le créer ;
les copier dans le dossier travail, en enlevant le point qui les commence.

Deuxième étape : comment faire ? 1er niveau, compréhension de l'énoncé

* La liste des fichiers cachés triée par taille décroissante s'obtient avec la commande « ls -aS ».
* Récupérer les six plus gros consiste à parcourir la liste pour copier chaque nom & en s'arrêtant après le sixième.

⁕ Enlever le point qui les commence nécessite de copier la sous-chaîne commençant au deuxième caractère dans une variable qui sera le nouveau nom du fichier.
⁕ Il faudra tester si le répertoire existe & s'il n'existe pas le créer avec mkdir.

Troisième étape : comment faire ? 2nd niveau, en français abrégé

```
si travail n'existe pas alors [1]
  créer travail [2]
fin si
nb ← 0 [5]
pour chaque fichier_caché dans le dossier répéter [3]
  copier fichier_caché vers travail/sous-chaîne(fichier_caché,2) [4]
  nb ← nb+1 [5]
  si nb = 6 alors [6]
    arrêter l'itération
  fin si
fin pour
```

Quatrième étape : le script bash

Travail préparatoire

1. Vérification de l'existence du dossier avec la commande `test -e`.
2. Pour créer le fichier nous pouvons soit employer une instruction conditionnelle `if`, soit combiner les commandes avec l'opérateur `&&`. Nous utiliserons cette seconde méthode.
3. Constitution de la liste de fichiers cachés : il faut employer la commande `ls -aS` ou `ls -laS` si nous voulons obtenir la taille des fichiers. Mais cette commande affiche également les fichiers non cachés. Pour ne sélectionner que ceux-là, il faut soit dans la boucle vérifier que le premier caractère du nom de fichier est un « . », soit utiliser la commande `grep` pour ne sélectionner que les noms commençant par un point `grep -E '^[\.]'`.

En examinant la liste des fichiers cachés obtenue par `ls -aS`, nous constatons la présence des fichiers nommés « `.` » & « `..` ». Pour les enlever, soit nous les traitons en exception en testant le nom du fichier, soit, après avoir lu le manuel de `ls`, nous recourons à la com-

mande `ls -AS` (vous saisissez l'utilité de lire les pages de manuel [01410] !).

Si vous avez employé la commande `ls -FAS` vous avez constaté que certains des plus gros fichiers sont des dossiers, comme nous ne voulons copier que des fichiers il faudra les sauter.

4. La copie se fait sans problème.
5. Comme il y a plus de six fichiers cachés, il nous faudra compter les fichiers copiés.
6. Il faudra interrompre la boucle quand nous atteindrons ce nombre. Nous emploierons la combinaison des commandes pour commander l'exécution de l'instruction `break`.

Liste

```
1  [ ! -e travail ] && mkdir travail
2  nb=0
3  for fic_cach in `ls -AS | grep -E '^[\.]'`; do
4    if [ ! -d $fic_cach ]; then
5      cp $fic_cach travail/${fic_cach:1}
6      let nb+=1
7      [ $nb -eq 6 ] && break
8    fi
9  done
```

La ligne 7 est équivalente à `if [ $nb -eq 6 ]; then break; fi`.

Une des ambiguïtés du *bash* provient de l'évaluation des expressions logiques. Celles-ci sont des expressions pouvant être vraies ou fausses. Il y a deux sortes d'opérateurs qui s'y rapportent :

- les opérateurs logiques (`||`, `&&` & `!`),
- les opérateurs de comparaison (`=`, `!=`, `<`, `<=`, `>` & `>=`). Ces opérateurs doivent être remplacés par leur équivalent littéraire : `-eq`, `-ne`, `-gt`, `-lt`, `-ge`, `-le`) pour les comparaisons numériques.

De plus, les expressions doivent être entourées, selon les cas de crochets carrés, de doubles crochets carrés ou de doubles parenthèses. Nous y reviendrons.

Exemple récapitulatif 2

Il s'agit de faire un programme qui, lorsqu'on lui donne un nom de dossier, affiche une sous liste ou la liste complète des fichiers qu'il contient & propose d'effacer celui sélectionné avec une confirmation différente selon qu'il s'agit d'un fichier ou d'un dossier que l'on a ou pas les droits pour le faire.

Nous allons employer des fonctions & vous proposer deux façons d'écrire le programme. Ce sera à vous d'écrire les fonctions, en appliquant, pour chacune, la démarche indiquée, en quatre étapes (que faut-il faire –énoncé ou 0e niveau– ? comment le faire –1er niveau– ! comment le faire –2e niveau– ! & écriture en bash –3e & dernier niveau) ! pour le premier exemple. Il vous faudra expliquer comment vous avez trouvé votre solution.

Voici l'algorithme

```
trouver les fichiers & les mettre dans la liste
tantque l'utilisateur ne veut pas arrêter répéter
  effacer la console
  afficher la liste
  choisir le fichier à effacer
  si l'utilisateur ne s'arrête pas alors
    demander confirmation de l'effacement
    si confirmé alors
      effacer le fichier
    sinon
      signaler l'absence d'effacement
    finsi
  finsi
fintantque
```

Le tableau ci-dessous contient deux versions du programme principal. Les mots en italique sont les noms des fonctions à écrire.

Solution 1	Solution 2
#!/bin/bash # Ce script peut avoir 2 paramètres ; le premier est obligatoire, c'est le nom du	#!/bin/bash # Ce script peut avoir 2 paramètres ; le premier est obligatoire, c'est le nom du

Solution 1	Solution 2

dossier, le second facultatif est un motif inclus dans le nom des fichiers à sélectionner.
 	
liste_fic=$(*remplis_liste* "$1" "$2")
while true; do
  clear
  echo "Liste des fichiers"
  echo "------------------------"
  choix=$(*afficher_menu* "$liste_fic")
  if ["$choix" = "Quitter"]; then
  break # ou exit 0
  else
  confirmer $choix
  if [$? -eq 1]; then
  effacer_le_fichier $choix
  else
  echo "Annulation de la suppression"
  fi
  pause
  liste_fic=$(*remplis_liste* "$1" "$2")
  fi
done
	dossier, le second facultatif est un motif inclus dans le nom des fichiers à sélectionner.
	
	liste_fic=$(*remplis_liste* "$1" "$2")
	choix=" "
	until ["$choix" -eq "Quitter"]; do
	  clear
	  echo "Liste des fichiers"
	  echo "------------------------"
	  choix=$(*afficher_menu* "$liste_fic")
	  if ["$choix" = "Quitter"]; then
	  continue
	  fi
	  confirmer $choix
	  if [$? -eq 1]; then
	  effacer_le_fichier $choix
	  fi
	  else
	  echo "Annulation de la suppression"
	  fi
	  pause
	  liste_fic=$(*remplis_liste* "$1" "$2")
	done

La fonction *remplis_liste* repose sur la commande `ls`, *effacer_le_fichier* sur la commande `rm`, *confirmer_choix* sur la commande `read` & *afficher_menu* sur `select`, mais à quoi sert la commande `true` ? que devrait faire la fonction `pause` ?

Que manque-t-il en dehors de commentaires, pour que ce script soit parfait ?

Que se passerait-il si vous enleviez les guillemets lors des passages de paramètres aux fonctions ?

Compléments sur la programmation en bash

Si l'on se limite à la grammaire du langage telle qu'elle est définie dans la section précédente, ce langage n'est pas très performant.

En fait, sa puissance vient de quatre facteurs :

- les *expansions*,
- les *protections*,
- les *expressions rationnelles*,
- &, & ce n'est le moindre, *la possibilité d'employer & de combiner les commandes sophistiquées de manipulation de données* intégrées dans unix : `grep`, `find`, `awk`, `sed`, `cut`, `sort`, etc.

Nous allons consacrer une section à chacun de ses facteurs.

Auparavant il faut définir une expression qui va être réutilisée, celle de *mot réservé*.

Un *mot réservé* est un mot de base lexicale du langage dont le sens ne peut être modifié (on ne peut les employer pour nommer une variables [01510]) ; en voici la liste : **if**, **then**, **elif**, **else**, **fi**, **case**, **in**, **esac**, **select**, **for**, **until**, **while**, **do**, **done**, **!**, **[[**, **]]**, **{**, **}**, **function**, **time** [01610].

Il faut aussi définir un certain nombre de généralités qui faciliteront l'écriture & la lecture des scripts, puisque l'objectif premier de ce cours est d'atteindre une compréhension claire des scripts contenus dans init.d & celle de certains scripts d'installation de logiciels.

Généralités

Avant d'entrer dans le détail, il faut bien comprendre ce qu'est *bash*, car *bash* n'est pas un simple interpréteur de script comme le *Perl*, le *PHP*, etc. C'est un interpréteur de commande. Cela a quelques conséquences sur son exécution. De fait, on distingue les

exécutions, on dit les *shells* ou les *sessions*, selon qu'elle sont de *login* (de connexion) ou pas, selon qu'elle sont *interactives* ou pas.

Un shell est dit de login si le premier caractère de son argument numéro zéro est un – (autrement dit pour l'exécuter vous tapez `-bash` au lieu de `bash`), ou s'il est invoqué avec l'option `-login`. Vous pouvez le vérifier : en tapant `echo $0` (`$0` donnant le nom de la commande en cours d'exécution), vous obtiendrez `-bash`.

Un shell est dit interactif si son entrée standard & sa sortie standard sont toutes deux connectées à un terminal, ou s'il est invoqué avec l'option `-i`. La variable d'environnement PS1 (le message d'attente du système n'est pas vide) est positionné, & le paramètre `$-` (liste des options de la commande en cours) contient la lettre `i` si *bash* est interactif, ce qui permet à un script ou à un fichier de démarrage de vérifier l'état du shell.

Le paragraphe suivant décrit comment *bash* exécute ses fichiers d'initialisation. Si l'un de ces fichiers existe, mais n'est pas accessible en lecture, *bash* signale une erreur. Les tildes sont remplacées par des noms de fichiers.

Cela a une conséquence sur l'initialisation de *bash*.

L'INITIALISATION

Avertissement : *certaines distributions, s'éloignant des standards unixiens de fait, emploient des fichiers d'initialisation avec des noms différents ou ne proposent pas tous les fichiers décrits ci-dessous.*

SHELLS INTERACTIFS, MAIS PAS DE LOGIN

Au démarrage, si ~/.bashrc existe, il est exécuté. ***Cf option : -norc , -rcfile.***

SHELLS NON-INTERACTIFS

Au démarrage, si la variable d'environnement *BASH_ENV* est non-nulle, elle est développée (elle est remplacée par son contenu), & le fichier dont elle contient le nom est exécuté, comme si l'on appliquait la commande **if** ["*$BASH_ENV*"]; **then** . *$BASH_ENV*; **fi**, mais on n'utilise pas *PATH* pour rechercher le chemin d'accès.

cf option : Invocation.

Bash tente de déterminer s'il est exécuté par le démon exécutant des shells à distance (généralement appelé `rshd`). Si *bash* se rend compte qu'il est exécuté par `rshd` , il lit & exécute les commandes de ~/.bashrc , si ce fichier existe & est accessible en lecture. Il ne fera pas cela comme le ferait `sh` . ***cf option : --norc, --rcfile.***

cf option : -p.

Fonctionnement

Ces trois paragraphes vous permettrons de mieux comprendre ce qui se passe lors de l'exécution d'un script.

Exécution des commandes

Après le découpage de la ligne de commande en mots, si le résultat est une commande simple suivie d'une éventuelle liste d'arguments, les actions suivantes sont effectuées.

Si le nom de la commande ne contient pas de slash, c'est qu'il ne s'agit pas d'un chemin relatif ou absolu, donc, le shell tente de la trouver dans les dossiers listés dans *PATH*, mais auparavant il regarde s'il existe une fonction shell de ce nom ; si oui, elle est appelée, sinon il continue la recherche dans la liste des fonctions internes (commandes internes & alias) ; sinon *bash* continue en cherchant dans chacun des dossiers membres de *PATH* un fichier exécutable du nom désiré. Si la recherche réussit l'interpréteur exé-

cute la commande sinon il affiche un message d'erreur & renvoie un code de retour non nul.

Si la recherche réussit, ou si l'argument 0 est rempli avec le nom fourni, les autres arguments seront éventuellement remplis avec le reste de la ligne de commande.

Si l'exécution échoue parce que le programme n'est pas un exécutable & si le fichier n'est pas un répertoire, on le considère alors comme un script shell (un fichier contenant une série de commandes). Un sous-shell (une session shell non interactive) est alors créé pour exécuter ce script. Ce sous-shell se réinitialisera lui-même, comme si un nouveau shell avait été invoqué pour exécuter le script, mais il continuera à mémoriser l'emplacement des commandes connues de son parent.

Si le programme est un fichier commençant par #!, le reste de la première ligne indique un interpréteur pour ce programme. Le shell activera l'exécution de cet interpréteur (Si l'interpréteur est **/bin/bash**, il démarrera une session non interactive. Dans ce cas la ligne de commande ressemblera à

bash [argbash1 [argbash2 ...]] nomscript [argscript1 [argscript2 ...]]).

En d'autres termes vous pourriez, par exemple, employer la ligne de *shabang* pour lancer un shell de connexion en même temps que votre script au lieu de la session non interactive usuelle.

ENVIRONNEMENT

Quand un programme est invoqué, il reçoit un tableau de chaînes que l'on appelle environnement. Il s'agit d'une liste de paires nom-valeur, de la forme nom=valeur. Le *bash* permet de manipuler l'environnement de plusieurs façons. Au démarrage, il analyse son propre environnement, & crée un paramètre pour chaque nom trouvé, en le marquant comme exportable vers les processus fils. Les commandes exécutées héritent de cet environnement. Les commandes **export** & **declare -x** [01012] permettent d'ajouter ou de supprimer des

paramètres ou des fonctions de l'environnement. Si la valeur d'un paramètre de l'environnement est modifiée, la nouvelle valeur devient une partie de l'environnement, & elle remplace l'ancienne. L'environnement hérité par les commandes exécutées est l'environnement initial (dont les valeurs peuvent être modifiées), moins les éléments supprimés par la commande `unset`, plus les éléments ajoutés par les commandes `export` & `declare -x`.

L'environnement d'une commande simple ou d'une fonction peut être augmenté temporairement en la faisant précéder d'une affectation de paramètre. Ces affectations ne concernent que l'environnement vu par cette commande ou fonction.

Cf option : -k, commande set

Quand *bash* invoque une commande externe, la variable spéciale $_ contient le chemin d'accès complet à cette commande, & elle est transmise dans l'environnement.

Statut ou code de retour

Ces deux expressions sont synonymes. Au niveau du shell, une commande qui se termine avec un code de retour nul est considérée comme réussie, le zéro indique le succès ; un statut non-nul indique un échec. Quand une commande se termine à cause d'un signal fatal, *bash* utilise la valeur 128+signal comme code de retour.

Si une commande n'est pas trouvée, le processus fils créé pour l'exécuter renvoie la valeur 127. Si la commande est trouvée mais pas exécutable, son statut vaudra 126.

bash lui-même renvoie le code de retour de la dernière commande exécutée, à moins qu'une erreur de syntaxe ne se produise, auquel cas il renvoie une valeur non-nulle.

La commande **echo *$?*** affiche ce code de retour !

Transferts de données : redirections & tubes

Il ne s'agit pas ici de l'assignation d'une valeur à une variable, mais des transferts de données entre commandes (tubes) ou entre périphériques (redirections).

TUBES (PIPES)

Un tube est une séquence d'au moins deux commandes séparées par le caractère |. Le format d'un tube est :

[time][!] commande_1 | commande_2 [| ...].

La sortie standard de la commande_1 est connectée à l'entrée standard de la commande_2. Cette connexion est établie avant toute redirection indiquée dans une commande elle-même.

Les crochets carrés indiquent, ici, que leur contenu est facultatif, c'est, clairement, le cas du mot réservé !. Dans le second groupe de crochets carrés, le chaînage avec des commandes supplémentaires est facultatif.

Si le mot réservé ! précède un tube, la valeur de sortie de celui-ci sera la négation de la valeur de retour de la dernière commande (Elle vaudra zéro si la valeur était différente de zéro & un si elle était nulle !). Sinon, le statut d'un tube sera celui de la dernière commande. L'interpréteur attend la fin de toutes les commandes du tube avant de renvoyer une valeur.

Si le mot réservé time [01013] précède le tube, les temps écoulés, consommés par le programme & par le système pour le programme sont indiqués quand le tube se termine. L'option -p change le format de sortie pour celui spécifié par POSIX. La variable *TIMEFORMAT* peut être affectée avec une chaîne de format indiquant comment les informations doivent être affichées.

Chaque commande du tube est exécutée comme un processus indépendant (c'est-à-dire dans un sous-shell).

REDIRECTION

Avant qu'une commande ne soit exécutée, il est possible de rediriger son entrée & sa sortie en utilisant une notation spéciale interprétée par le shell. Les redirections peuvent également servir à ouvrir ou fermer des fichiers dans l'environnement actuel du shell.

Elle consiste à modifier la source pour un flux de données entrant (saisies) ou la destination pour un flux de données sortant (résultats ou erreurs).

Généralement un shell a trois fichiers ouverts, correspondants à trois *descripteurs* (parfois appelés *canaux*, ce sont, en fait, des *variables fichiers*, ce qui explique que l'on puisse en changer la valeur) :

- le descripteur 0, entrée standard (unité physique le clavier, /proc/self/fd/0),
- le descripteur 1, sortie standard (unité physique l'écran, /proc/self/fd/1),
- le descripteur 2, sortie d'erreur (unité physique l'écran, /proc/self/fd/5).

Un tube est une redirection de la sortie standard vers un fichier temporaire en mémoire qui va servir d'entrée pour la commande suivante. C'est parce que les données transmises sont temporaires & qu'elles le sont de commande en commande que l'on parle de *flux* (*stream*).

Les opérateurs de redirection décrits ci-dessous peuvent apparaître avant, ou au sein d'une commande simple ou suivre une commande. Les redirections sont traitées dans l'ordre d'apparition de gauche à droite.

Dans les descriptions suivantes, si le numéro de descripteur de fichier [01018] est omis, & si le premier caractère de l'opérateur de redirection est **<**, celui-ci correspondra à l'entrée standard (descripteur de fichier 0). Si le premier caractère de l'opérateur est **>**, la redirection s'appliquera à la sortie standard (descripteur de fichier 1).

Le mot qui suit l'opérateur de redirection dans les descriptions suivantes est soumis à l'expansion des accolades, du tilde, des paramètres, à la substitution de commandes, à l'évaluation arithmétique, à la suppression des protections, & au développement des noms de fichiers. S'il se modifie pour donner plusieurs mots, *bash* détectera une erreur.

⁎ Remarquez que l'ordre des redirections est important. Par exemple, la commande,

```
1  ls > liste_répertoires 2>&1
```

redirige à la fois la sortie standard & la sortie d'erreur vers le fichier liste_répertoires, alors que la commande

```
1  ls 2>&1 > liste_répertoires
```

ne redirige que la sortie standard vers le fichier liste_répertoires, car la sortie d'erreur a été renvoyée vers la sortie standard avant que celle-ci ne soit redirigée vers liste_répertoires.

Bash gère plusieurs noms de fichiers de manière particulière, lorsqu'ils sont utilisés dans des redirections, comme décrit dans la table suivante :

/dev/fd/n	Si n est un entier valide (*entre 0 & 7, par exemple sur notre système actuel*), le descripteur de fichier n est dupliqué.
/dev/stdin	Le descripteur de fichier 0 est dédoublé.
/dev/stdout	Le descripteur de fichier 1 est dédoublé.
/dev/stderr	Le descripteur de fichier 2 est dédoublé.
/dev/tcp/host/port	Si host est une adresse Internet ou un nom d'hôte valide, & si port est un numéro de port entier ou un nom de service, *bash* tentera d'ouvrir une connexion TCP sur le socket correspondant.

Une erreur d'ouverture ou de création de fichier peut déclencher un échec.

Les redirections qui utilisent des descripteurs de fichiers supérieurs à 9 doivent être utilisées avec précaution, car il peut y avoir des conflits avec les descripteurs de fichiers que le shell utilise en interne.

1 Redirection d'entrée

Lorsque l'on applique une redirection d'entrée, le fichier dont le nom résulte du développement du mot sera ouvert en lecture avec le descripteur de fichier numéro *n*, ou en tant qu'entrée standard (descripteur de fichier 0) si *n* n'est pas mentionné.

Le format général des redirections d'entrée est le suivant :

```
[n]<mot
```

2 Redirection de sortie

Lors d'une redirection de sortie, le fichier dont le nom résulte du développement du mot est ouvert en écriture, avec le descripteur de fichier *n*, ou en tant que sortie standard (descripteur de fichier 1) si *n* n'est pas mentionné. Si le fichier n'existe pas, il est créé. S'il existait déjà, sa taille est ramenée à 0.

Le format général des redirections de sortie est le suivant :

```
[n]>mot
```

cf option : noclobber de set.

3 Redirection pour ajout en sortie

Lorsqu'on redirige ainsi la sortie, le fichier dont le nom résulte du développement du mot est ouvert pour ajout en fin de fichier, avec le descripteur *n*, ou en tant que sortie standard (descripteur 1) si n n'est pas mentionné. Si le fichier n'existe pas, il est créé.

Le format général pour la redirection de sortie avec ajout est :

```
[n]>>mot
```

4 Redirection de la sortie standard & de la sortie d'erreur

Bash permet la redirection simultanée de la sortie standard (descripteur 1) & de la sortie d'erreur (descripteur 2), dans un fichier dont le nom est le résultat du développement du mot avec cette construction.

Il y a deux formes pour effectuer cette double redirection :

```
1  &>mot
```

&

```
1  >&mot
```

On préfère généralement la première. Elle est sémantiquement équivalente à

```
1  >mot 2>&1
```

5 Document en ligne

Avec ce type de redirection, le shell va lire son entrée standard jusqu'à ce qu'il atteigne une ligne contenant uniquement le mot prévu (sans espaces à la suite), nommé étiquette. Une fois cette étiquette atteinte, il exécutera la commande demandée en lui fournissant en entrée le texte lu avant l'étiquette, que l'on appelle document en ligne.

Le format des documents en ligne est le suivant :

```
1  <<[-]étiquette
2          document en ligne
3  étiquette
```

Il n'y a ni remplacement de paramètre, ni substitution de commande, ni développement de chemin d'accès, ni évaluation arithmétique sur le mot. Si l'un des caractères du mot *étiquette* est protégé, *l'étiquette* est obtenue après suppression des protections dans le mot, & les lignes du document ne sont pas développées. Sinon, toutes les lignes du texte saisi sont soumises au remplacement des paramètres, à la substitution de commandes, & à l'évaluation arithmétique. Cet opérateur, inspiré du PHP, nous semble rarement employé.

Si l'opérateur de redirection est <<-, alors les tabulations en tête de chaque ligne sont supprimées, y compris dans la ligne contenant étiquette. Ceci permet d'indenter de manière naturelle les documents en ligne au sein des scripts, car elles sont remplacées par des espaces.

6 Chaînes en ligne

Une variante aux documents en ligne, le format est :

```
1 <<<mot
```

Le mot est développé & fourni à la commande sur son entrée standard. Cet opérateur, aussi inspiré du PHP, nous semble rarement employé.

7 Dédoublement de descripteur de fichier

* L'opérateur de redirection [n]<&mot permet de dupliquer les descripteurs de fichiers en entrée.

Si mot se transforme en un ou plusieurs chiffres, le descripteur de fichier *n* devient une copie de ce descripteur.

Si les chiffres de mot ne correspondent pas à un descripteur en lecture, une erreur se produit.

Si mot prend la forme -, le descripteur *n* est fermé.

Si *n* n'est pas mentionné, on utilise l'entrée standard (descripteur 0).

* L'opérateur de redirection [n]>&mot est utilisé de manière similaire pour dupliquer les descripteurs de sortie.

Si *n* n'est pas précisé, on considère la sortie standard (descripteur 1).

Si les chiffres de mot ne correspondent pas à un descripteur en écriture, une erreur se produit.

Un cas particulier se produit si *n* est omis, & si mot ne se développe pas sous forme de chiffres. Alors, les sorties standard & d'erreurs sont toutes deux redirigées comme précédemment.

8 DÉPLACEMENT DE DESCRIPTEURS DE FICHIERS

L'opérateur de redirection [n]<&chiffre- déplace le descripteur de fichier chiffre vers le descripteur de fichier n, ou sur l'entrée standard (descripteur de fichier 0) si n n'est pas spécifié. chiffre est fermé après avoir été dédoublé en n.

De la même manière, l'opérateur de redirection [n]>&chiffre- déplace le descripteur de fichier chiffre vers le descripteur de fichier n sur sur la sortie standard (descripteur de fichier 1) si n n'est pas spécifié.

9 OUVERTURE EN LECTURE/ÉCRITURE D'UN DESCRIPTEUR DE FICHIER

L'opérateur de redirection [n]<>mot ouvre le fichier dont le nom résulte du développement du mot, à la fois en lecture & en écriture & lui affecte le descripteur de fichier n, ou bien le descripteur 0 si n n'est pas mentionné. Si le fichier n'existe pas, il est créé.

Les Expressions

Le *bash* est un langage contenant une grande variété d'expressions. Nous avons déjà abordées les expressions simples. Nous allons voir les opérateurs *insolites*.

Expressions simples, les opérateurs insolites

Le tableau suivant montre le résultat d'opérations inhabituelles.

Opération	Résultat
Valeurs initiales	
valeura	65 soit en base 2 01000001
valeurb	15 soit en base 2 00001111
Opérateurs arithmétiques	
valeura *modulo* valeurb, **%** Le modulo est le reste de la division des nombres entiers.	5
Opérateurs bits à bits	
décalage à droite de 1 bit de valeura ⟺ 65/2, **>>**	32
décalage à gauche de 2 bits de valeura ⟺ 65*4, **<<**	260
valeura *&* valeurb bits à bits, **&**	1 soit en base 2 00000001
valeura *ou* valeurb bits à bits, **|**	79 soit en base 2 01001111
Opérateur d'assignation	
assignation numérique additionnelle de valeura +=2, **+=**	67
assignation numérique multiplicative de valeura *=3, ***=**	201

Écrivez le script affichant ce tableau, sans bordure ni valeurs en base 2, & en séparant les colonnes par le caractère [:]. Comme il ne comprend que des commandes d'assignation & des commandes `echo`, il ne nécessite pas d'autres corrections que la comparaison de

votre résultat avec ce tableau (Rappel : pour obtenir le résultat d'une expression arithmétique, il faut l'entourer de $((& de)).)

Notez que le décalage à droite est équivalent à la division des nombres entiers pour les puissances de 2 (1 bit étant la division par 2, 2 bits par 4, etc.)

Notez également que le ET & le OU logique (notés respectivement && & ||) renverraientt tous les deux la valeur 1, soit VRAI (TRUE), 0 représentant la valeur FAUX (FALSE).

Attention : Pour la valeur de retour des commandes 0 signifie VRAI & ≠0 signifie FAUX. Il ne s'agit pas d'une inconséquence :

- *les valeurs logiques doivent satisfaire à l'algèbre de Boole dans laquelle VRAI vaut 1 & FAUX 0 ;*
- *la valeur de retour d'une commande répond au besoin de ses programmeurs ceux-ci veulent signaler les erreurs par un nombre. La valeur 0 est une valeur unique pour indiquer la réussite, les autres nombres signalent précisément le type d'erreur.*

Exemple 33 :

```
$> ls titi
ls: impossible d'accéder à titi: Aucun fichier ou dossier de ce type
$ echo $?
2
```

Ni les premières ni les secondes ne sont conçues pour autre chose. C'est par paresse que certains programmeurs les utilisent hors contexte, avec le risque d'effets de bord !

NOTION DE COMMANDES

En dehors des commandes internes & externes déjà invoquées, il existe trois notions à comprendre : celle de commande simple, celle de liste de commandes & celle de commandes combinées.

Commandes simples

Une commande simple est une séquence d'affectations de variables, facultative, suivie de mots séparés par des blancs & des redirections, & terminée par un opérateur de contrôle [01014]. La syntaxe est la suivante :

[Variable=Valeur ...] [Nom_commande] [Arguments] [Redirection ...].

Exemple 34 :

```
1  $ LANG=fr_FR.UTF-8 man 1 man >toto 2>/dev/null
2  $ man man
```

La première commande redirige la page en français de la commande man du chapitre 1 du manuel Linux, dans le fichier toto du dossier de travail & envoie les messages d'erreur dans le trou noir /dev/null.

La seconde affiche la page en anglais sur l'écran après vous avoir demandé la page du chapitre que vous voulez consulter, car on trouve une page man dans chacun des chapitres1, 7, 1p du manuel.

Le statut, ou la valeur de retour, d'une commande simple est son code de sortie, ou *128+n* si le processus de la commande a été interrompu par le signal *n* [01015].

Listes

Une liste est une séquence d'une ou plusieurs commandes séparées par l'un des opérateurs **;**, **&**, **&&**, ou **||**, & terminée par ; &, ou ↵ (retour-chariot ou **\n**). Dans cette liste d'opérateurs, **&&** & **||** ont une précédence identique (*précédence* est le synonyme *jargonesque* de *priorité*), suivis par **;** & **&**, qui ont également une même *précédence*.

```
1  commande_1 ; commande_2 [ ; commande_n ...]
1  commande_1 & commande_2 [ & commande_n ...]
```

Si une commande se termine par l'opérateur de contrôle **&**, l'interpréteur l'exécute en arrière-plan, dans un sous-shell. L'interpréteur n'attend pas que la commande se termine & retourne un code 0.

Les commandes séparées par un **;** sont exécutées successivement, l'interpréteur attend que chaque commande se termine avant de lancer la suivante. Le statut est celui de la dernière commande exécutée.

Les opérateurs de contrôle **&&** & **||** indiquent respectivement une liste liée par un ET logique, & une liste liée par un OU logique.

Une liste ET a la forme :

```
1  commande_1 && commande_2 [ && commande_n ...]
```

La énième commande n'est exécutée que si toutes les précédentes ont eu un code de retour nul.

Une liste OU a la forme :

```
1  commande_1 || commande_2 [ || commande_n ...]
```

La énième commande n'est exécutée que si toutes les précédentes ont eu un statut non nul.

La valeur de retour des listes ET & OU est celle de la dernière commande exécutée dans la liste.

COMMANDES COMPOSÉES

Une commande composée est l'une des constructions suivantes :

* **(** liste **)**

  liste est exécutée dans un sous-shell. Les affectations de variables, & les commandes internes qui affectent l'environnement de l'interpréteur n'ont pas d'effet une fois que la commande se termine. Le code de retour est celui de la liste.

* **{** liste**[;|\n]** **}**

liste est simplement exécutée avec l'environnement du shell en cours. liste doit se terminer par un caractère \n ou ;. Cette construction est connue sous le nom de *commandes groupées*, elle s'emploie particulièrement pour la définition des fonctions. Le code de retour est celui de la liste. Veuillez noter que, contrairement aux méta-caractères, (&), { & } sont des mots réservés qui ne doivent apparaître que là où un mot réservé peut être reconnu. *Puisqu'ils ne provoqueront pas un coupage de mot, ils doivent être séparés de la liste par une espace !*

* ((expression))

L'expression est évaluée comme une expression arithmétique entière. Si la valeur arithmétique de l'expression est non-nulle, le code renvoyé est 0 ; sinon 1 est renvoyé. Cela est strictement identique à let "expression".

Exemple 35 :

```
$ (( 0 && 1 )) # 'ET' logique
$ echo $?
1
$ (( 1 && 1 )) # 'ET' logique
$ echo $?
0
```

* [[expression]]

Cet opérateur est dit d'expression logique étendue, car il évalue des expressions que la commande test n'arrive pas à évaluer (comme, par exemple, les opérateurs &&, ||, < & > ou l'évaluation des expressions octales & hexadécimales). Il renvoie 1 ou 0 (VRAI ou FAUX) selon la valeur de la condition expression.

Exemple 36 :

```
$ decimal=15
$ octal=017   # = 15 (decimal)
$ if [[ "$decimal" -eq "$octal" ]]; then echo égal; else echo
```

inégal; fi
égal

Les expressions sont composées d'éléments primaires. Le coupage des mots & l'expansion des chemins ne sont pas réalisés sur les portions entre [[&]]; l'expansion des tildes, des paramètres, des variable, des expressions arithmétiques, la substitution des commandes & des processus, ainsi que la disparition des apostrophes sont réalisés. Les opérateurs conditionnels (ce sont les options de la commande test) tels que -f ne doivent pas être *guillemetés* afin d'être reconnus comme primaires.

Opérateurs supplémentaires

Quand les opérateurs == & != 01016 sont utilisés, la chaîne placée à droite de l'opérateur est considérée comme étant un motif & est recherchée selon les règles de recherche des motifs. ***Cf option : nocasematch***. La valeur renvoyée est 0 si les chaînes correspondent (==) (ou respectivement ne correspondent pas –!=), & 1 sinon. Toute partie du motif peut être protégée avec des apostrophes pour forcer sa comparaison en tant que chaîne (sans développement).

Un opérateur binaire supplémentaire, =~, est disponible, avec la même priorité que == & !=. Lorsqu'il est utilisé, la chaîne à droite de l'opérateur est considérée comme une expression rationnelle étendue & est mise en correspondance en conséquence. La valeur renvoyée est *0* si la chaîne correspond au motif, & *1* si elle ne correspond pas. Si l'expression rationnelle n'est pas syntaxiquement correcte, la valeur de retour de l'expression conditionnelle est *2* 01017.

Cf option : nocasematch.

Les sous-chaînes mise en correspondance avec des sous-expressions entre parenthèses dans l'expression rationnelle sont enregistrées dans la variable tableau *BASH_REMATCH*. L'élément d'index 0 de *BASH_REMATCH* est la partie de la chaîne correspondant à l'expression

rationnelle complète. L'élément d'index n de *BASH_REMATCH* est la partie de la chaîne correspondant à la énième sous-expression entre parenthèses.

RAPPELS SUR LES EXPRESSIONS

Les expressions peuvent être combinées en utilisant les opérateurs suivants, par ordre décroissant de priorité :

(expression)	Retourne la valeur de l'expression. Cela peut être utilisé pour outrepasser la priorité normale des opérateurs.
! expression	Vraie si expression est fausse.
expression1 && expression2	Vraie si expression1 &t expression2 sont toutes les deux vraies.
expression1 || expression2	Vraie si expression1 ou expression2 est vraie.

Les opérateurs && & || n'évaluent pas *expression2* si la valeur de *expression1* suffit à déterminer le code de retour de l'expression conditionnelle entière.

COMMANDES COMPOSÉES STRUCTURANTES

Toutes les instructions structurantes du langage sont des commandes composées. Nous les rappelons ici, en insistant sur l'aspect **mots réservés** de ces commandes internes.

* **for** nom [**in** mot] ; **do** liste ; **done**

La liste de mots à la suite de **in** est développée, créant une liste d'éléments. La variable nom prend successivement la valeur de chacun des éléments, & liste est exécutée à chaque fois.

Si **in** mot est omis, la commande **for** exécute la liste une fois pour chacun des paramètres positionnels (Rappel : un paramètre

position est un paramètre désigné par sa position sur la ligne, comme *$0*, *$1*, etc.) ayant une valeur. Le code de retour est celui de la dernière commande exécutée.

Si le développement de ce qui suit **in** est une liste vide, aucune commande n'est exécutée & 0 est renvoyé.

* **select** nom [**in** mot] ; **do** liste ; **done**

La liste de mots à la suite de **in** est développée, créant une liste d'éléments. L'ensemble des mots développés est imprimé sur la sortie d'erreur standard, chacun précédé par un nombre.

Si **in** mot est omis, la liste est celle des paramètres positionnels. Le symbole d'accueil PS3 est affiché, & une ligne est lue depuis l'entrée standard.

Si la ligne est constituée d'un nombre correspondant à l'un des mots affichés, la variable nom est remplie avec ce mot.

Si la ligne est vide, les mots et le symbole d'accueil sont affichés à nouveau. Si une fin de fichier (EOF, Ctrl D) est lue, la commande se termine. Pour toutes les autres valeurs, la variable nom est vidée. La ligne lue est stockée dans la variable *REPLY*. La liste est exécutée après chaque sélection, jusqu'à ce qu'une commande **break** soit atteinte.

Le code de retour de **select** est celui de la dernière commande exécutée dans la liste, ou zéro si aucune commande n'est exécutée.

* **case** mot **in** [motif [| motif]

Une commande **case** commence d'abord par développer le mot, puis essaye de le mettre en correspondance successivement avec chacun des motifs en utilisant les mêmes règles que pour les noms de fichiers. Le mot est développé en utilisant le développement du tilde, le développement des paramètres & des variables, la substitution arithmétique, la substitution de commande, la substitution de processus & la suppression d'apostrophes.

Chaque motif examiné est développé en utilisant le développement du tilde, le développement des paramètres & des variables, la substitution arithmétique, la substitution de commande & la substitution de processus.

Cf option : nocasematch. Quand une correspondance est trouvée, la liste associée est exécutée. Dès qu'un motif correct a été trouvé, il n'y a plus d'autre essais.

Le statut vaut zéro si aucun motif ne correspond, sinon il s'agit du code de la dernière commande exécutée dans la liste.

* **if** liste ; **then** liste ; [**elif** liste ; **then** liste ;] ... [**else** liste ;] **fi**

La liste du **if** est exécutée. Si son code de retour est nul, la liste du **then** est exécutée. Sinon, chacune des listes des **elif** est exécutée successivement, & si un code de retour est nul, la liste du **then** associé est exécutée, & la commande se termine. En dernier ressort, la liste du **else** est exécutée. Le code de retour est celui de la dernière commande exécutée, ou zéro si aucune condition n'a été vérifiée.

Attention : dans les expressions logiques des instructions 0 vaut VRAI & une autre valeur FAUX, alors que pour les opérateurs logiques, qui peuvent être employés dans ces mêmes expressions, 1 vaut VRAI & 0 vaut FAUX. Cette ambiguïté peut être une source d'erreur !

* **while** liste ; **do** liste ; **done**
  until liste ; **do** liste ; **done**

La commande **while** répète la liste du **do** tant que la dernière commande de la liste du **while** renvoie un statut nul. La commande **until** agit de même manière, sauf que le test est négatif, & la liste du **do**, exécutée tant que la liste du **until** renvoie un code non-nul. Le statut des commandes **while** & **until** est celui de la

dernière commande exécutée dans la liste **do**, ou zéro si aucune commande n'a été exécutée.

* [**function**] nom **()** commande-composée [redirection]

Ceci définit une fonction possédant appelée nom. Le mot réservé **function** est optionnel. *S'il est fourni, les parenthèses sont optionnelles.* Le corps de la fonction est, traditionnellement, la commande-composée entre **{** & **}**, mais peut être toute commande décrite dans le paragraphe Commandes composées p. 90.

La commande-composée est exécutée chaque fois que nom est spécifié comme le nom d'une commande normale. Toutes les redirections (cf Reditrection p. 80) spécifiées lorsqu'une fonction est définie sont effectuées lorsque la fonction est exécutée.

Le code de retour d'une définition de fonction est zéro à moins qu'il y ait une erreur de syntaxe ou qu'une fonction en lecture seule, de même nom, existe déjà.

Lorsque la fonction est exécutée, le code de retour est celui de la dernière commande exécutée dans le corps de la fonction.

DÉVELOPPEMENT, EXPANSION & SUBSTITUTION

Il faut comprendre le fonctionnement de l'interpréteur de comprendre pour saisir l'importance des mécanismes d'expansion & de substitution.

INTERPRÉTATION D'UNE LIGNE DE COMMANDE

Lors de l'exécution d'une commande simple (syntaxe : [variable ...] commande [option] [arguments ...] [redirection ...]), le shell effectue les développements (traitement des options) affectations & redirections de gauche à droite, suivants.

1. Les mots que l'analyseur a repéré comme affectation de variables (ceux qui précèdent le nom de la commande) & les

redirections sont mémorisés pour une mise en place ultérieure.

2. Les autres mots sont développés. S'il reste des mots après le développement, le premier est considéré comme le nom d'une commande & les suivants comme ses arguments.
3. Les redirections sont mises en place.

Le texte suivant le = dans chaque affectation est soumis au développement du tilde, des paramètres, à la substitution de commande, à l'évaluation arithmétique & à la suppression des protection avant de remplir la variable.

Si aucun nom de commande ne résulte des précédentes opérations, l'assignation de variable modifie l'environnement en cours. Sinon elle est ajoutée à celui de la commande exécutée & ne modifient pas l'environnement du shell. Si l'une des tentatives d'affectation concerne une variable en lecture seule, une erreur se produit, & la commande se termine sur un code non-nul.

Si aucun nom de commande n'est obtenu, les redirections sont réalisées mais ne modifient pas l'environnement du shell en cours. Une erreur de redirection renvoie un code de retour non-nul.

S'il reste un nom de commande après l'expansion, l'exécution a lieu. Sinon la commande se termine.

Si l'un des développements contient une substitution de commande, le code de retour est celui de la dernière substitution de commande réalisée. S'il n'y en a pas, la commande se termine avec un code de retour nul.

Le point 2 montre l'importance des *expansions* & des *substitutions*. Ces mots sont synonymes de celui de *développement*. Il s'agit à notre sens, dans les trois cas, de mauvaises traductions : il existe en français deux mots représentants ces mécanismes l'*extension* & la *périphrase*. Dans la première, on remplace une expression abstraite par sa signification concrète : par exemple on remplace *les*

jours de la semaine par *lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche*. Dans la seconde une expression abstraite par une valeur concrète, par exemple *un jour* & *un lundi*.

Les expansions sont appliquées à la ligne de commande après qu'elle ait été divisée en mots. Il existe sept types de développements : *expansion des accolades*, *développement du tilde*, *remplacement des paramètres & des variables*, *substitution de commandes*, *évaluation arithmétique*, *découpage des mots*, & *développement des noms de fichiers*.

Leur ordre est :

- expansion :
  - des accolades,
  - du tilde,
  - des paramètres,
  - des variables,
  - des commandes ;
- évaluation arithmétique (selon la méthode de-gauche-à-droite) ;
- découpage des mots ;
- développement des noms de fichiers.

Sur les systèmes qui le supportent, un développement supplémentaire a lieu : la substitution de processus.

Seuls l'expansion des accolades, le découpage des mots, & le développement des noms de fichiers peuvent modifier le nombre de mots (extensions ou expansions). Les autres développement transforment un mot unique en un autre mot unique (périphrases ou substitutions).

EXPANSION DES ACCOLADES

L'expansion des accolades est un mécanisme permettant la création de chaînes quelconques. Il est similaire au développement des noms

de fichiers, mais les noms de fichiers créés n'existent pas nécessairement. Les motifs qui seront développés prennent la forme d'un *préambule* facultatif, suivi par soit une série de chaînes séparées par des virgules, soit une expression de type séquence encadrée par des accolades. Un *postambule* peut éventuellement suivre la série de chaînes. Le *préambule* est inséré devant chacune des chaînes contenues entre les accolades, & le *postambule* est ajouté à la fin de chacune des chaînes résultantes, le développement se faisant de gauche à droite.

Plusieurs développements d'accolades peuvent être imbriqués. Les résultats de chaque développement ne sont pas triés, l'ordre de gauche à droite est conservé.

Exemple 37 :

a{d,c,b}e se développe en ade ace abe où a est le préambule & e le postambule.

Une *expression de type séquence* prend la forme {x..y}, où x & y sont soit des entiers, soit des caractères seuls. Lorsqu'il s'agit d'entiers, l'expression est remplacée par la liste des nombres entre x & y, x & y compris. S'il s'agit de caractères, l'expression est remplacée par l'ensemble des caractères situés entre x & y d'un point de vue lexicographique. Notez que x & y doivent être du même type.

L'expansion des accolades est effectuée en premier, & tous les caractères ayant une signification spéciale pour les autres développement sont conservés dans le résultat. Il s'agit d'une modification purement littérale. *Bash* n'effectue aucune interprétation syntaxique du texte entre les accolades.

Une formule correcte pour le développement doit contenir des accolades ouvrantes & fermantes non protégées, & au moins une virgule non protégée ou une expression séquence valide. Toute formule incorrecte n'est pas développée & reste inchangée. Une { ou une ,

peuvent être protégées par une barre oblique inverse pour éviter d'être considérés comme partie d'une expression entre accolades.

Cette construction est généralement utilisée comme raccourci lorsque le préfixe commun aux différentes chaînes est relativement long :

Exemple 38 : `mkdir -p /usr/local/src/bash/{old,new,dist,bugs}`

qui permet de créer, en une ligne, tous les sous-dossiers de même niveau

ou

Exemple 39 : `chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}`

qui permet de changer, en une ligne, le propriétaire de fichiers se trouvant dans des dossiers différents.

Pour info, le développement des accolades induit une légère incompatibilité avec les versions traditionnelles de l'interpréteur BOURNE *sh*. *sh* n'effectue aucun traitement sur les accolades ouvrantes & fermantes lorsqu'elles apparaissent dans un mot, & les laisse inchangées. *Bash* supprime les accolades dans les mots, après développement.

Exemple 40 : En *sh*, le mot `fichier{1,2}` reste inchangé

Exemple 41 : En *bash*, le mot `fichier{1,2}` devient `fichier1 fichier2`

DÉVELOPPEMENT DU TILDE

Si un mot commence avec le caractère tilde (~), tous les caractères précédant le premier slash non protégé (voire tous les caractères s'il n'y a pas de slash), sont considérés comme un préfixe tilde. Si aucun caractère du préfixe tilde n'est protégé, les caractères suivant le tilde sont traités comme un *nom de connexion* possible. Si ce *nom de login* est une chaîne nulle, le tilde est remplacé par la valeur du paramètre *HOME*. Si *HOME* n'existe pas, le tilde est remplacé par le répertoire de connexion de l'utilisateur exécutant le shell.

Si le préfixe tilde est +, la valeur du paramètre shell *PWD* le remplace. Si le préfixe tilde est -, la valeur du paramètre shell *OLDPWD* lui est substitué. Si les caractères à la suite du tilde dans le préfixe

tilde représentent un nombre *N* préfixé éventuellement d'un + ou d'un - le préfixe tilde est remplacé par l'élément correspondant de la pile de répertoires telle qu'il serait affiché par la commande interne dirs invoquée avec le préfixe tilde en argument. Si les caractères à la suite du tilde dans le préfixe tilde représentent un nombre sans signe, on suppose qu'il s'agit de +.

Exemple 42 :

Si vous êtes connecté en root & si vous travaillez dans le dossier /etc,

`cd ~toto`

vous place dans le dossier /root/toto, s'il existe

ou

`cd ~-`

vous ramène dans le dossier où vous trouviez précédemment.

Si le nom est invalide, ou si le développement du tilde échoue, le mot est inchangé.

Chaque affectation de variable est soumis au développement du tilde s'il suit immédiatement un : ou le premier =. On peut donc utiliser des chemins d'accès avec un tilde pour remplir les variables *PATH*, *MAILPATH* & *CDPATH*, le shell fournira la valeur correcte.

REMPLACEMENT DES PARAMÈTRES

Attention *: le mot paramètre désigne trois types d'objets différents :*

- ***les données apparaissant sur la ligne de commande*** *(paramètres de positionnement,*
- **les variables (ou paramètres) spéciales**, *renseignant sur la ligne de commande,*
- ***les variables*** *(paramètres ayant un nom non défini, dans bash).*

Le caractère $ permet d'introduire le remplacement des paramètres, la substitution de commandes, ou l'expansion arithmétique.

Le nom du paramètre ou du symbole à développer peut être encadré par des accolades, afin d'éviter que les caractères suivants ne soient considérés comme appartenant au nom de la variable.

Lorsque les accolades sont utilisées, l'accolade finale est le premier caractère } non protégé par une barre oblique inverse ni inclus dans une chaîne protégée ni dans une expression arithmétique, une substitution de commande ou un développement de paramètre.

ACCÈS À LA VALEUR

${paramètre**}**

Cette expression est remplacé par la valeur du paramètre. Les accolades sont nécessaires quand le paramètre est un paramètre positionnel ayant plusieurs chiffres, ou s'il est suivi de caractères n'appartenant pas à son nom. C'est le mécanisme permettant d'accéder à la valeur d'une variable.

Si le premier caractère du paramètre est un point d'exclamation, un niveau d'indirection de variable [01019] est introduit. *Bash* utilise la valeur de la variable formée par le reste du paramètre comme un nom de variable. Cette variable est alors développée & la valeur utilisée pour le reste de la substitution plutôt que la valeur du paramètre lui-même. On appelle ce mécanisme le développement indirect.

Les exceptions à celui-ci sont les développements de **${!**prefix***}** & de **${!**nom**[@]}** décrits plus loin. Le point d'exclamation doit immédiatement suivre l'accolade ouvrante afin d'introduire l'indirection.

Dans chacun des exemples suivants, le mot est soumis au développement du tilde, au remplacement des paramètres, à la substitution de commandes & à l'évaluation arithmétique. *Bash* vérifie si un paramètre existe & s'il n'est pas nul. L'omission du double point ne fournit qu'un test d'existence.

UTILISATION D'UNE VALEUR PAR DÉFAUT

${paramètre**:-**mot**}**

Si le paramètre est inexistant ou nul, on substitue le développement du mot. Sinon, c'est la valeur du paramètre qui est fournie.

Assignation d'une valeur par défaut

${paramètre**:=**mot**}**

Si le paramètre est inexistant ou nul, le développement du mot lui est affecté. La valeur du paramètre est alors renvoyée.

Les paramètres positionnels & spéciaux ne peuvent pas être affectés de cette façon.

Remarque : Ces deux extensions sont employées dans les scripts que l'on veut voir exécuté même si aucun paramètre attendu n'est fourni comme par exemple ***ls*** *qui utilise le dossier courant quand on l'emploie sans nom de fichier.*

Aucune de ces deux extensions ne devrait être employée pour des paramètres fonctionnels n'ayant pas de valeur ou une valeur nulle alors qu'ils devraient en avoir une, car dans ce cas, c'est qu'il y a un problème de fond ! Cela nous semble similaire à une ligne de programme de prévisions économiques, lue un jour, disant que si le Chiffre d'Affaire prévisionnel du mois était négatif, il fallait le mettre à 1 ou encore cette autre proposant de mettre le PNB à zéro si la prévision s'avérait négative, aucune de ces deux valeurs ne pouvant être négative !

Affichage d'une erreur si inexistant ou nul

${paramètre**:?**mot**}**

Si paramètre est inexistant, ou nul, le développement de mot (ou un message approprié si aucun mot n'est fourni) est affiché sur la sortie d'erreur standard, & l'interpréteur s'arrête, s'il n'est pas interactif. Autrement, la valeur du paramètre est utilisée.

Utilisation d'une valeur différente

${paramètre**:+**mot**}**

Si le paramètre est nul, ou inexistant, rien n'est substitué. Sinon le développement du mot est renvoyé.

EXTRACTION DE SOUS-CHAÎNE

https://www.opensuse.org/fr/${paramètre**:**début**}**

${paramètre**:**début**:**longueur**}**

Se développe pour fournir la sous-chaîne du nombre de caractère indiqué par longueur, commençant à début. Si longueur est omis, le résultat est la sous-chaîne commençant au caractère début & s'étendant jusqu'à la fin du paramètre. longueur & début sont des expressions arithmétiques. longueur doit être positive ou nulle. *Si début est négatif, sa valeur est considérée à partir de la fin du contenu du paramètre.*

Si le paramètre est @, le résultat correspond aux longueur paramètres positionnels commençant au paramètre numéro début.

Si le paramètre est un nom de tableau indexé par @ ou *, le résultat rassemble longueur membres du tableau commençant au début-moins-unième (Les tableaux étant indicés à partir de 0).

Une valeur négative de début est prise relativement à la valeur maximum de l'index du tableau considéré, augmentée de un. Notez qu'une valeur négative de début doit être séparée du deux-points par au moins une espace pour éviter toute confusion avec le développement de « **:-** » (cf point 1). L'indexation des caractères débute à zéro, sauf dans le cas des paramètres positionnels qui sont indexés à partir de 1 [01020].

Exemple 43 :

```
#!/bin/bash
echo 'Extraction du 1er paramètre avec ${*:1:1} :' ${*:1:1}
echo 'Extraction du 2e caractère de $1 avec ${1:1:1} :' ${1:1:1}
```

```
echo 'Extraction de l'avant dernier de $2 avec ${2: -2:1} : '${2: -2:1}
```

Résultat

```
$ sh test azertyuiopqsdf 147852369
Extraction du 1er paramètre avec ${*:1:1} : azertyuiopqsdf
Extraction du 2e caractère de $1 avec ${1:1:1} : z
Extraction de l'avant dernier de 147852369 avec ${2: -2:1} : 6
```

Noms commençant par une chaîne

2 `${#préfixe*}`
`${!préfixe@}`

Se développe en les noms des variables dont les noms commencent par préfixe, séparés par le premier caractère de la variable d'environnement IFS. Avec l'opérateur @ si l'expansion est entre guillemets (""), chaque nom de variable sera un mot séparé.

Liste des cases d'un tableau

3 `${!nom[@]}`
`${!nom[*]}`

Si nom est une variable de type tableau, elle se développe en la liste des indices du tableau affecté à nom. Si nom n'est pas un tableau, se développe en 0 s'il existe & en rien autrement. Si @ est utilisé & que le développement apparaît entre guillemets, chaque indice se développe en un mot séparé.

Longueur d'un paramètre

`${#parameter}`

Fournit le nombre de caractère du paramètre. Si le paramètre est * ou @, la valeur est le nombre de paramètres positionnels. Si le paramètre est un nom de tableau indexé par * ou @, la valeur est le nombre d'éléments dans le tableau.

SUPPRESSION D'UN PRÉFIXE

4 **${**paramètre**#**motif**}**
${paramètre**##**motif**}**

Le motif est développé pour fournir un modèle, comme dans l'expansion des noms de fichiers. S'il correspond au début de la valeur du paramètre, alors le développement prend la valeur du paramètre après suppression du plus petit motif commun (cas « # »), ou du plus long motif (cas « ## »).

Si le paramètre est @ ou *, la suppression de motif est appliquée à chaque paramètre positionnel successivement & le développement donne la liste finale. Si le paramètre est une variable tableau indexée par @ ou *, la suppression de motif est appliquée à chaque membre du tableau successivement & le développement donne la liste finale.

Cela implique que le motif soit variable, comme le *bash* ne connaît pas les expressions rationnelles, cela implique la présence du caractère * dans le motif. 'ABC' est un motif fixe ne correspondant qu'à 'ABC', 'A*C' correspond aussi bien à 'ABC' qu'à 'A123vc,zC'.

Exemple 44 :

```
# suppression de la plus petite sous-chaîne de $1 finissant par un a
# attention le motif est *a & non a
echo ${1#*a}
# suppression de la plus longue sous-chaîne de $1 finissant par un a
echo ${1##*a}
```

Résultat

```
$ sh test aazaaarat
azaaarat        # ne supprime que la premier a
t               # supprime jusqu'au dernier a
```

Si au lieu de *a nous avions saisi z*r comme motif, rien ne se serait passé, car le motif ne commençait pas la chaîne.

SUPPRESSION D'UN SUFFIXE

${paramètre**%**motif**}**
${paramètre**%%**motif**}**

Même traitement que dans le paragraphe précédent, mais à partir de la fin du paramètre.

REMPLACEMENT D'UNE SOUS-CHAÎNE

${paramètre**/**motif**/**chaîne**}**

Le motif est développé comme dans le traitement des noms de fichiers. Le paramètre est développé & la plus longue portion correspondant au motif est remplacée par la chaîne.

Si le motif commence par /, toutes les correspondances de motif sont remplacés par chaîne. Normalement, seule la première correspondance est remplacée.

Si le motif commence par #, il doit correspondre au début de la valeur développée du paramètre.

Si le motif commence par %, il doit correspondre à la fin du développement du paramètre.

Si la chaîne est nulle, les portions correspondant au motif sont supprimées & le / après le motif peut être omis.

Si le paramètre est @ ou *, l'opération de substitution est appliquée à chacun des paramètres positionnels successivement & le résultat est la liste finale.

Si le paramètre est une variable tableau indexée par @ ou *, l'opération de substitution s'applique à chaque membre du tableau successivement & le résultat est la liste finale.

Exemple 45 :

```
echo ${1/aaa/bbb}"|"${2/aaa/bbb}
echo ${1/%aaa/bbb}"|"${2/%aaa/bbb}
echo ${1/#aaa/bbb}"|"${2/#aaa/bbb}
echo ${1//aaa/}"|"${2//aaa/}
```

```
echo ${1//aaa/bbb}"|"${2//aaa/bbb}
```

Résultat

```
$ sh test zzaaavfaaads aaazzaaadfaaafaaa
zzbbbvfaaads|bbbzzaaadfaaafaaa      # remplacement en début
zzaaavfaaads|aaazzaaadfaaafbbb      # remplacement en toute fin
zzaaavfaaads|bbbzzaaadfaaafaaa      # remplacement en tout début
zzvfds|zzdff                        # suppressions
zzbbbvfbbbds|bbbzzbbbdfbbbfbbb      # remplacements partout
```

SUBSTITUTION DE COMMANDES

La substitution de commandes permet de remplacer le nom d'une commande par son résultat. Il en existe deux formes :

```
$(commande)
```

ou

```
`commande`
```

Bash effectue la substitution en exécutant la commande & en la remplaçant par sa sortie standard, dont les derniers sauts de lignes sont supprimés. Les sauts de lignes internes ne sont pas supprimés mais peuvent disparaître lors du découpage en mots. La substitution de commande `$(cat fichier)` peut être remplacée par l'équivalent plus rapide `$(< fichier)`.

Quand l'ancienne forme de substitution avec les accents graves (*backquotes*) « ` » est utilisée, le caractère antislash garde sa signification littérale, sauf s'il est suivi de `$`, `` ` ``, ou `\`. Le premier *backquote* non protégée par un antislash termine la substitution de commande. Quand on utilise la forme `$(commande)`, tous les caractères entre parenthèses gardent leurs valeurs littérales. Aucun n'est traité spécialement.

Les substitutions de commandes peuvent être imbriquées. Avec l'ancienne forme, il faut protéger les accents graves internes avec une barre oblique inverse.

Si la substitution apparaît entre guillemets, le découpage des mots, & l'expansion des noms de fichiers ne sont pas effectués.

Évaluation Arithmétique

L'évaluation arithmétique permet de remplacer une expression par le résultat de son évaluation. Son format est :

`$((expression))`

L'expression est manipulée de la même manière que si elle se trouvait entre guillemets, mais un guillemet se trouvant entre les parenthèses n'est pas traité spécifiquement. Tous les mots de l'expression subissent le développement des paramètres, la substitution des commandes & la suppression des apostrophes & des guillemets. Les développements arithmétiques peuvent être imbriquées.

L'évaluation est effectuée en suivant les règles mentionnées dans le paragraphe CALCUL ARITHMÉTIQUE du manuel bash (cf. fascicule 2 Annexe 3 p. 132). Si l'expression est invalide, *bash* affiche un message indiquant l'erreur & aucune substitution n'a lieu.

Substitution de Processus

La substitution de processus n'est disponible que sur les systèmes acceptant le mécanisme des *tubes nommés* (FIFO) ou celui des *descripteurs de noms de fichiers* (pseudos fichiers /dev/fdi ou i est un chiffre de 0 à 8). Elle prend la forme `<(liste)` ou `>(liste)`. cf. fascicule 2 du cours Linux, Annexe 3 pour plus de détails.

Séparation des mots

Les résultats du remplacement des paramètres, de la substitution de commandes & de l'évaluation arithmétique, qui ne se trouvent pas entre guillemets sont analysés par le shell afin d'appliquer le découpage des mots.

L'interpréteur considère chaque caractère du paramètre IFS comme un délimiteur & redécoupe le résultat des transformations précédentes en fonction de ceux-ci. Si la valeur de IFS est exactement ␣⇥↵, (\ \t \n, la valeur par défaut), alors toute la séquence contenue dans IFS sert à délimiter les mots. cf. fascicule 2 Annexe 3 pour plus de précisions.

Les arguments nuls explicites (chaîne vide notée "" ou '') sont conservés. Les arguments nuls implicites, résultant du développement des paramètres n'ayant pas de valeurs, sont éliminés. Si un paramètre sans valeur est développé entre guillemets, le résultat est un argument nul qui est conservé.

Notez que si aucun développement n'a lieu, le découpage des mots n'est pas effectué.

DÉVELOPPEMENT DES NOMS DE FICHIERS

Après le découpage des mots & si l'option -f n'est pas indiquée, *bash* recherche dans chaque mot les caractères *, ?, (, & [. Si l'un d'eux apparaît, le mot est considéré comme un motif & est remplacé par une liste, classée par ordre alphabétique, des noms de fichiers correspondant à ce motif. cf. fascicule 2 Annexe 3 pour plus de précisions.

MOTIFS GÉNÉRIQUES

Tout caractère apparaissant dans un motif, hormis les caractères spéciaux décrits ci-après correspond à lui-même. Le caractère NUL (code ASCII 0) ne peut pas se trouver dans un motif. Un *backslash* protège le caractère suivant ; la barre oblique de protection est abandonné si elle correspond. Les caractères spéciaux doivent être protégés s'ils doivent se correspondre littéralement.

Les caractères spéciaux ont les significations suivantes :

- * correspond à n'importe quelle chaîne, y compris la chaîne vide ;

- ? correspond à n'importe quel caractère ;
- [...] correspond à l'un des caractères entre crochets.

* Une paire de caractères séparés par un trait d'union indique une expression intervalle; tout caractère qui correspond à n'importe quel caractère situé entre les deux bornes incluses, en utilisant les paramètres régionaux courant & le jeu de caractères.
  - Si le premier caractère suivant le [est un ! ou un ^ alors la correspondance se fait sur les caractères non-inclus.
  - L'ordre de tri des caractères dans les expressions intervalle est déterminé par les paramètres régionaux courants & par la valeur de la variable shell LC_COLLATE si elle existe.
  - Un - peut être mis en correspondance en l'incluant en premier ou dernier caractère de l'ensemble.
  - Un] peut être mis en correspondance en l'incluant en premier caractère de l'ensemble.
* Entre [&], on peut indiquer une classe de caractère en utilisant la syntaxe **[:**classe**:]**, où classe est l'une des classes suivantes, définies dans la norme POSIX : alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit .

  Une classe correspond à un caractère quelconque qui s'y trouve. La classe de caractères word correspond aux lettres, aux chiffres & au caractère souligné « _ ».
* Entre [&], on peut indiquer une classe d'équivalence en utilisant la syntaxe [=c=], qui correspond à n'importe quel caractère ayant le même ordre (comme indiqué dans la localisation en cours) que le caractère c.
* Entre [&], la syntaxe **[.**symbole**.]** correspond au symbole de classement symbole. Autrement dit, si vous avez défini une classe d'équivalence [=c=] qui désigne les caractères CcçÇ, [.ç.] ne désigne que ç.

Cf option : extglob, shopt. plusieurs opérateurs de correspondance étendue sont reconnus. Dans la description suivante, une

liste-motif est une liste d'un ou plusieurs motifs séparés par des |. Les motifs composés sont formés en utilisant un ou plusieurs sous-motifs comme suit :

- ?(liste-motif) correspond à zéro ou une occurrence des motifs indiqués ;
- *(liste-motif) correspond à zéro ou plusieurs occurrences des motifs indiqués ;
- +(liste-motif) correspond à une ou plusieurs occurrences des motifs indiqués ;
- @(liste-motif) correspond à une occurrence exactement des motifs indiqués ;
- !(liste-motif) correspond à tout sauf les motifs indiqués.

SUPPRESSION DES PROTECTIONS 01021

Après les développements précédents, toutes les occurrences non-protégées des caractères \, `, & " qui ne résultent pas d'un développement sont supprimées.

Les Expressions rationnelles

Remarque : le saut de cette section, en première lecture, ne devrait pas vous empêcher de faire la quasi-totalité des exercices proposés.

Une expression rationnelle est une chaîne de caractères, dite aussi *motif*, qui décrit un ensemble de chaînes de caractères possibles selon une syntaxe précise. Cette description est appelée *correspondance*.

Le mot *illégal* est souvent employé pour désigner les infractions aux règles de la grammaire des expressions rationnelles, mais ces infractions ne sont pas susceptibles de poursuites judiciaires !

Ces expressions sont employées par *bash*, par `vi`, par `emacs`,par `grep`, par `sed`, par `awk` & par tous les langages de scripts comme *PHP* ou *Perl* & par les logiciels bureautiques comme *writer* & *calc*, respectivement traitement de texte & tableur, des suites *LibreOffice/OpenOffice.org*.

Nous allons nous intéresser, essentiellement, à leur emploi dans les scripts *Bash* & par voie de conséquence par `grep`, `awk` & `sed`.

Les expressions rationnelles (ER ou exp-rat par la suite), définies par POSIX.2 existent sous deux formes : les ER modernes (en gros celles de `egrep` ou `awk`, que POSIX.2 appelle expressions rationnelles étendues), & les ER anciennes (en gros celles de `sed`, les ER basiques pour POSIX.2). Il faut essayer de n'employer que les modernes, mais si l'on veut profiter de la puissance de `sed` ce n'est pas possible !

Les colonisés parlent d'expressions régulières (traduction mot à mot de l'anglais *regular expression*).

Définitions

* Un *atome* est :
  - un ensemble vide () (correspond à une chaîne nulle),
  - une expression entre crochets (voir § suivant),

- un point . (correspondant à n'importe quel caractère),
- un signe ^ (chaîne vide en début de ligne),
- un signe $ (chaîne vide en fin de ligne),
- un \ suivi d'un des caractères dits spéciaux (^, ., [, $, (,), |, *, +, ?, {, & \ ; ces caractères sont en fait des opérateurs, le \ étant le caractère d'échappement qui les fait correspondre à leur valeur littérale sans signification particulière),
- un \ suivi de n'importe quel autre caractère (correspondant au caractère pris sous forme littérale, comme si le \ était absent),
- un caractère ordinaire sans signification particulière (correspondant à ce caractère),
- une { suivie d'un caractère autre qu'un chiffre est considérée sous sa forme littérale, elle constitue un atome pas un encadrement !

* Un *encadrement* est une { suivie d'un entier décimal non signé, suivi éventuellement d'une virgule, suivie éventuellement d'un entier décimal non signé, toujours suivi d'une }. Les entiers doivent être compris entre 0 & RE_DUP_MAX (255), & s'il y en a deux, le second doit être supérieur ou égal au premier.
* Une sous-chaîne est composée d'un ou de plusieurs atomes.
* Une sous-expression est composée d'une ou de plusieurs sous-chaîne entouré par des parenthèses.

L'encadrement indique le nombre de répétition de l'atome, de la sous chaîne ou de la sous-expression qui le précède.

Règles

* Un atome suivi de * correspond à une séquence de 0 ou plusieurs correspondances pour l'atome.
* Un atome suivi d'un + correspond à une séquence de 1 ou plusieurs correspondances pour l'atome.
* Un atome suivi d'un ? correspond à une séquence de zéro ou une correspondance pour l'atome.

* Un atome suivi d'un encadrement contenant un entier i & pas de virgule, correspond à une séquence de i correspondances pour l'atome exactement, exemple a{3} pour trouver aaa.
* Un atome suivi d'un encadrement contenant un entier i & une virgule correspond à une séquence d'au moins i correspondances pour l'atome, exemple a{3,} trouvera aaa & aaaa.
* Un atome suivi d'un encadrement contenant deux entiers i & j correspond à une séquence de i à j (compris) correspondances pour l'atome, exemple a{2,3} trouvera aa & aaa mais ni a ni aaaa.
* Il est illégal de terminer une exp-rat avec un \ seul.

Dans le cas où une ER peut correspondre à plusieurs sous-chaînes d'une chaîne donnée, elle correspond à celle qui commence le plus tôt dans la chaîne.

Si l'*exp-rat* peut correspondre à plusieurs sous-chaînes débutant au même point, elle correspond à la plus longue sous-chaîne.

Les sous-expressions correspondent aussi à la plus longue sous-chaîne possible, à condition que la correspondance complète soit la plus longue possible, les sous-expressions débutant le plus tôt dans l'ER ayant priorité sur celles débutant plus loin. Notez que les sous-expressions de haut-niveau ont donc priorité sur les sous-expressions de bas-niveau les composant.

La longueur des correspondances est mesurée en caractères, pas en éléments fusionnés. Une chaîne vide est considérée comme plus longue qu'aucune correspondance. Par exemple :

- bb* correspond au trois caractères du milieu de abbbc ;
- (vers|vert)(atile|igineux) correspond aux caractères de versatile, de vertigineux, mais aussi de versigineux & de vertatile ;
- si (.*).* est mis en correspondance avec abc, la sous-expression entre parenthèses correspond aux trois caractères ;
- si (a*)* est mis en correspondance avec bc l'exp-rat entière & la sous-ER entre parenthèses correspondent toutes les deux avec une chaîne nulle.

Si une correspondance sans distinction de casse est demandée (***cf option : nocasematch de bash, -i de grep***) toutes les différences entre majuscules & minuscules disparaissent de l'alphabet. Un symbole alphabétique apparaissant hors d'une expression entre crochets est remplacé par une expression contenant les deux casses (par exemple x désigne aussi bien x que X). Lorsqu'il apparaît dans une expression entre crochets, tous ses équivalents sont ajoutés ([x] devient [xX] & [^x] devient [^xX]).

Aucune limite particulière n'est imposée sur la longueur d'une exp-rat, mais les programmes destinés à être portables devrait limiter les leurs à 256 octets, car une implantation compatible POSIX peut refuser les expressions plus longues.

INFORMATIONS COMPLÉMENTAIRES

- Les expressions rationnelles obsolètes (basiques) diffèrent sur plusieurs points.
  - |, +, & ? y sont des caractères normaux sans équivalents.
  - Les délimiteurs d'encadrements sont \{ & \}, car { & } sont des caractères ordinaires.
  - Les parenthèses pour les sous-expressions sont \(& \), car (&) sont des caractères ordinaires.
  - ^ est un caractère ordinaire sauf au début d'une ER ou au début d'une sous-expression entre parenthèses [01022].
  - $ est un caractère ordinaire sauf à la fin d'une exp-rat ou à la fin d'une sous-expression entre parenthèses.
  - * est un caractère ordinaire s'il apparaît au début d'une ER ou au début d'une sous-expression entre parenthèses (après un éventuel ^).
- De plus, il existe un nouveau type d'atome, la *référence arrière* : \ suivi d'un chiffre décimal non nul n correspond à la même séquence de caractères que ceux mis en correspondance avec la énième sous-expression entre parenthèses. (les sous-expressions sont numéro-

tées par leurs parenthèses ouvrantes, de gauche à droite), ainsi dans l'expression <([ib])>(.*?)</\1> permet d'extraire le contenu des balises & dans une page HTML. Le \1 recopie le contenu de la première expression ce qui permet d'écrire les balises de fermeture & .

⁕ Enfin, il ne faut pas confondre expression rationnelle & facilité du langage ainsi la commande

```
mkdir -p ./{rep1,rep2/rep21,rep3}
```

qui permet de créer quatre sous-dossiers ne contient pas d'expression rationnelle, elle ne contient que le mécanisme dit d'expansion des accolades de *bash*.

L'annexe 2 reprend, avec des exemples testés avec différents logiciels, ces informations.

Ces explications vous permettent de saisir toute la puissance de ce langage de script.

C'est maintenant à vous de jouer !

Cahier d'exercices

Rappels de cours

Les commandes internes, notées comme `ceci` (aspect commande interne) ou comme **ceci** (aspect structure du script) ou encore comme **ceci** (aspect instruction non structurante) dans les scripts, sont définies dans le *shell*, leur aide est accessible avec le commande interne `help` ou avec une option `-h` ou `--help`.

Les commandes externes, notées comme `ceci`, sont stockées dans les dossiers /bin, /usr/bin, pour les commandes accessibles à tous & dans /sbin & /usr/sbin pour celles réservés à l'administrateur. La commande `whereis` vous permet de trouver l'emplacement des commandes externes. Leur aide se trouve dans les pages `man`. Vous les trouverez traduites sur Internet ! Certaines pages traduites sont accessibles par la commande LANG=fr_FR[.UTF-8] `man` nom_commande. Des guides sont présents dans les sous-dossiers de /usr/share.

Les fascicules PDTU contiennent, également, des exemples d'utilisation des différentes commandes.

Avertissement

N'oubliez pas l'extension .sh à la fin du nom de vos scripts.

EXERCICES SIMPLES

Vous ne les trouverez peut-être pas si simples, mais ils le sont pour deux raisons :

- ils nécessitent peu de lignes de programmes,
- ils sont accompagnés d'une aide détaillée.

EX. 0 : EXPRESSIONS

ÉNONCÉ 1

Écrire le script expressions.sh qui, calcule des expressions à partir des valeurs suivantes :

- nb1 vaut 100129901099,
- nb2 vaut 97,
- borne_inf vaut 10,
- borne_sup vaut 100,
- pos vaut 11,
- long vaut 5,
- ch1 contient a1234567890ABCDEFazertyCDEBILE813-666666.

Dans tous les cas, il faut afficher l'expression avant son résultat, par exemple, si nb1 vaut 3 & nb2 vaut 4, pour donner le résultat de leur multiplication, il vous faudra afficher 3*4 = 12.

EXPRESSIONS ARITHMÉTIQUES

* Calculer & afficher :
  - nb1 multiplié par nb2
  - nb1 modulo nb2

EXPRESSION LOGIQUE

Afficher vrai si nb2 est compris entre borne_inf & borne_sup, faux sinon.

Extraction d'une sous-chaîne de caractères

Afin de constater la différence entre l'emploi de l'opérateur **${ }** & l'opérateur d'extraction de sous-chaîne de la commande `expr` afficher les long caractères de ch1 commençant au caractère pos, précédés du texte de l'expression, exemple ${'toto':2:2} vaut 'ot'.

Compte du nombre de caractères dans une chaîne ou une sous-chaîne

Ces opérateurs ne savent pas gérer un motif qui ne commence pas au début de la chaîne dont on veut connaître l'effectif. Il faut donc sauter les caractères non concernés.

Afficher une expression permettant de calculer le nombre de chiffres au début de ch1 & son résultat d'abord avec la commande `expr`, puis avec **${ }**, sans sauter le premier caractère & en le sautant (ce qui revient à extraire la sous-chaîne commençant au caractère suivant).

Affichage d'une sous-chaîne

Refaire l'exercice précédant, en affichant les chiffres au lieu de les compter.

Commandes

`echo`, `expr` & `test`, plus pour les vérifications : `if then else fi`.

Aide

Relisez attentivement le paragraphe Les Expressions p. 50 & suivantes.

Énoncé 2

Écrire un script nb_jours.sh, qui calcule le nombre de jour entre deux dates. Les dates étant passées en paramètres date_de_début puis date_de_fin au format jj/mm/aa.

COMMANDES

echo, **date** & assignation

AIDE

Nous ne nous préoccuperons pas du cas singulier (nombre de jour = 1) !

Il existe deux façons de procéder la façon grand joueur & la façon petit joueur.

* La première consiste à compter le nombre de jour écoulé depuis une date le 1er janvier 1600, date retenue comme celle de l'adoption généralisée du calendrier grégorien dans le monde (Cette adoption c'est, en fait, échelonnée entre 1582 –Italie, France, etc.– & 1949 –Chine– , mais le mathématicien suisse CARL FRIEDRICH GAUSS –1777-1855–, qui inventa l'algorithme de calcul, s'en moquait ! Donc, elle ne donne pas le nombre de jours correct pour les dates antérieures à la date d'adoption ; pour ces dates il faut enlever de 10 à 13 jours, selon les pays, mais elle est efficace pour les dates contemporaines !) Elle nécessite un calcul complexe, puisqu'il faut tenir compte des années multiples de 4 qui sont bissextiles, des années multiples de 100 qui ne le sont pas & de celles multiples de 400 qui le sont, sans parler des longueurs des mois, pour les dates quelconques (autres que le 01/01).
* La seconde emploie un format de la commande **date**. Celle-ci peut donner la date du jour sous la forme du nombre de secondes écoulées depuis le 1er janvier 1970, date théorique de la création d'Unix. En faisant la différence entre cette présentation de deux dates & en divisant le résultat par le nombre de secondes d'une journée (24×3 600=86 400), on obtient le nombre de jours entre les deux.

Nous allons petit-jouer ! Cela nous permettra de manipuler des expressions dates, alphanumériques & numériques !

Le format naturel des dates pour les calculs est celui dit rationnel : aaaammjj. Il faut donc transformer les dates au format français abrégé

(jj/mm/aa). En pratique, comme ce script nous est destiné & que nous ne faisons pas n'importe quoi, nous ne testerons pas les dates & n'importe quel séparateur (-, / ou même a, sauf [ou <) fera l'affaire. Comme nous imposons une année à deux chiffres, nous concaténerons '20' devant l'année pour obtenir une année du XXIe siècle.

Exemple 46 :

```
$ . nb_jours.sh 01/01/15 03/01/15
Il y a 2 jours entre le 01/01/2015 & le 03/01/2015.
```

Ex. 1 : énumération

Énoncé

Écrire un script enumnum.sh qui lorsqu'on lui passe le nombre n comme argument affiche la liste des nombres de 1 à n (le signe $ en début de ligne représente le prompt traditionnel – c'est le signe > avec l'OpenSuse) :

```
1 2 3 .... n
$
```

Commandes

Cet exercice peut être réalisé avec les seules commandes `echo` & `for`, mais d'autres commandes comme `seq` peuvent servir.

Aide

Le fait que *n* soit passé en argument implique *primo*, que vous tapiez un nombre après le nom du script & *secundo*, que vous utilisiez sa valeur avec la variable prédéfinie $1.

Comment sait-on que la commande `seq` existe ?

Quand on doit écrire un script, il faut toujours se demander si le travail n'a pas déjà été fait.

Dans le cas présent, il faut se poser la question *Existe-t-il des commandes affichant ou manipulant des nombres utilisables par n'importe qui* ? En *bash* cela s'énonce `apropos number | grep "(1)"`. Si cela ne donne rien, il faut consulter internet, mais pas avant ! Qui plus est, dans le cadre de ce TP, il faut éviter de récupérer des scripts tout fait, car le seul moyen de comprendre un langage c'est de le pratiquer, pas de recourir à des scripts fait par d'autres, comme les assistés le pensent. Vous n'apprendrez jamais l'anglais en utilisant les services d'un interprète !

Ce premier exercice n'était qu'une application du cours qui précède. Voici maintenant un exercice qui nécessite une démarche plus rigoureuse. L'idée étant primo, de toujours scinder un problème complexe, en sous-problèmes moins complexes, secundo, quand on ne peut plus simplifier avec un raisonnement traitant tous les aspects du problème, chercher comment le traduire.

La plus grosse difficulté provient de la paresse d'esprit qui nous incite à sauter des étapes.

Ex. 2 : salutation

Énoncé

Écrire un script nommé salut.sh, qui, quand vous lui aurez donné votre prénom en paramètre, vous demandera « Allez-vous bien ou mal ? » prénom & qui affichera le message : « Bravo ! » prénom « ! Nous allons passer une bonne journée ! » dans le premier cas, le message : prénom « , ne désespérez pas ça va s'améliorer ! » dans le second & le message : « La journée s'annonce rude, » prénom« , car vous n'avez rien compris à la question ! » dans le cas d'une réponse autre. Bien sûr, vous pouvez écrire n'importe quel message vous convenant mieux !

Commandes

Il vous faut employer les commandes `read`, `echo`, `case`, `=`.

Aide

Nous avons besoin de deux variables. La première contiendra le prénom, la seconde la réponse à la question. Nous avons trois résultats possibles correspondant à deux alternatives imbriquées. Cela permet d'employer une instruction du type `selon` réponse `dans`...

L'instruction `read` permettra d'afficher la question & la réponse (cf. page 58).

Nous avons trois cas de figure : bien, mal ou autre chose, la commande `case` (cf. p. 62) semble plus appropriée que l'imbrication de commande `if` ... `elif` ... `else` ... `fi`, puisque l'existence de plus de deux cas similaires est la raison d'être de cette instruction.

La commande `echo` permettra d'afficher les trois messages lié à la réponse.

Attention : les langages de programmation étant d'origine anglo-saxonne, ils n'acceptent, généralement, aucun caractère diacritique (avec

accents, cédilles, tildes, etc.) dans les noms de variables & de fonctions. En règle générale, sauf en PHP à notre connaissance, ne sont autorisées que les lettres (A..Z, a..z), les chiffres (0..9) & le souligné(_).

Exemple 47 :

IDENTIFIANT	*VALIDITÉ*
prenom	*OK*
prénom	*KO, caractère accentué*
nom_de_famille	*OK*
NomEtPrenom	*OK*
nom&prenom	*KO esperluette*
Nom-prenom	*KO tiret non autorisé, alors qu'il l'est dans les noms de fichiers.*
tva106	*OK*

EX. 3 : AMÉLIORATIONS DE SALUTATION

ÉNONCÉ

1. Il s'agit d'améliorer le script précédent, car ne rien répondre (frappe de la seule touche ↵) n'est pas traité comme une erreur. Il faudrait obliger l'utilisateur à saisir une donnée & ne passer au traitement de la réponse qu'une fois assuré de la validité de la réponse. En vous limitant à la vérification de l'existence d'au moins un caractère dans la réponse (la réponse n'est pas vide), comment procéderiez-vous pour y arriver ?

2. Afin d'ajouter ce script à la fin de .bashrc, au lieu de passer le prénom en paramètre, nous allons utiliser le nom de connexion contenu dans la variable USER.

Que se passe-t-il quand on ajoute l'exécution du script à la fin de .bashrc ?

Que se passe-t-il si vous lancez une nouvelle session en ligne de commande (avec un terminal ou une console virtuelle) ?

3. Modifiez ce programme pour l'adapter à chacun des membres de votre famille. Comment procédez-vous ?

Aide

Attendre une réponse correcte demande de la comparer avec un résultat correct, ici un mot au sens de *bash*, c'est-à-dire, une suite de caractères quelconques, non vide, mais pouvant ne contenir qu'un caractère.

Vérifier qu'il s'agit d'un mot sémantiquement correct est beaucoup trop complexe !

Pour étendre le script afin de saluer chaque membre de la famille qui se connecte en mode texte (une famille de linuxiens

fous, Brrr !), vous pouvez écrire un petit script pour chacun des membres de la famille & en rendre l'exécution automatique à la connexion.

EX. 4 : ÉNUMÉRATIONS VARIÉES

Il s'agit d'une reprise de l'exercice 1 auquel nous allons ajouter trois variantes.

Écrivez trois scripts compte_w.sh, compte_u.sh & compte_f.sh qui afficheront sur l'écran les nombres de 1 à limite, limite étant saisie par l'utilisateur. Le programme compte_w.sh effectuera le calcul avec une commande **while**, le script compte_u.sh exécutera le travail avec une commande **until**, le programme compte_f.sh opérera avec une commande **for ... in**.

VARIANTE 1 : Vous demanderez à l'utilisateur de ressaisir limite tant que la valeur n'est pas comprise entre 5 & 10 exclus.

VARIANTE 2 : Modifiez vos programmes pour écrire le compte à rebours de la navette spatiale (énumération décroissante des nombres à raison d'un par seconde) & terminez-les par le message Feu.

Que constatez vous ?

Quelle commande vous aidera à résoudre ce problème

VARIANTE 3 : Modifier un de ces scripts pour que limite soit passée en argument du programme.

AIDE

Bien que l'énoncé de la variante 1 suggère l'emploi d'une commande while, le script peut s'écrire avec une boucle until.

Dans la variante 2, pour trouver la commande, employez apropos avec un mot anglais lié au problème.

Ex. 5 : trouver le nombre caché

Écrire un script le_bon_nombre.sh qui vous demande de trouver un nombre, compris entre 1 & 100, choisi au hasard par le script, en un minimum de coup. Le programme affichera « Trop grand ! » ou « Trop petit ! » en fonction de votre proposition ; il affichera un message de félicitation quand vous trouverez le nombre !

Aide

Dans Linux, il n'existe pas de commandes générant des nombres au hasard. Il faut donc en créer une. Comme c'est extrêmement complexe, nous emploierons un biais simple, donnant l'illusion d'un nombre aléatoire & évitant de programmer. Cela consiste à prendre le nombre de minutes écoulé depuis le début de l'heure, `` `date +%M` ``, à le multiplier par 60, puis à y ajouter le nombre de seconde `` `date +%S` ``, à prendre ensuite le reste de la division par 100 (opérateur modulo – `%` –, un nombre compris entre 0 & 99) & à y ajouter 1. Vous pouvez écrire l'opération en une ligne, mais, si vous la trouvez trop complexe, vous pouvez l'éclater sur quatre ou cinq lignes, les deux premières étant :

```
let nombre_a_trouver=`date +%M`
let nombre_a_trouver=$nombre_a_trouver * 60
```

En fait, s'il n'existe pas de commande, il existe une variable prédéfinie nommée *RANDOM*, dont la valeur, qui change à chaque référence, est un nombre aléatoire.

Nous allons utiliser une fonction pour simplifier l'écriture du script, pour diminuer sa complexité de lecture. Cette fonction vous l'avez, quasiment, écrite, dans un des exercices précédents, puisqu'elle sert à vérifier que le nombre saisi est bien compris entre 1 & 100.

Vous avez déjà écrit dans l'exercice précédent tout ce qui est nécessaire pour ce petit jeu.

QUESTIONS SUPPLÉMENTAIRES

En jouant intelligemment, même en étant malchanceux, il faut au plus 7 coups pour trouver le résultat ; pouvez-vous expliquez pourquoi ? Combien de coups faudrait-il pour trouver un nombre compris entre 1 & 1 000 ?

EX. 6 : CHASSE AUX TROYENS

ÉNONCÉ

Écrire le script no_trojan.sh qui, à l'aide de la commande iptables, interdira les ports de chevaux de Troie bien connus : 1234, 2222, 3333, 4321, 5555, 6666, 9999 & 12 345 en entrée sur votre machine.

COMMANDES

for ... in, iptables

AIDE

Ce script est assez représentatif d'un script d'administration : il ne nécessite pas une grande compétence en programmation, mais une bonne connaissance de la (ou des) commande utile. Sa seule diffi-

culté est de lire la documentation d'iptables (man iptables), la commande qui sert de base aux pare-feu Linux.

Afin de vous épargner ce supplice, voici les informations utiles :

- iptables fonctionne avec des règles ;
- celles pour interdire une entrée sont de la forme :
  * INPUT -p protocole --dport numéro_de_port -j DROP ;
- il faut une ligne par protocole ;
- il y a deux protocoles pour lesquels il faut bloquer les ports, udp & tcp ;
- les règles sont stockées dans un fichier auquel on peut les ajouter en ligne de commande avec l'option -A d'iptables ;
- iptables se trouve dans le dossier /sbin ; seul root peut l'exécuter.

La commande sudo iptables -L -n -v vous permettra de vérifier que ça a fonctionné.

Ex. 7 : choisir un jeux

Énoncé

Écrire le script liste_jeu.sh qui vous permet de choisir un jeu dans un menu.

La liste des jeux est : Trax, Othello 10×10, Pente, Puissance 4×4, Mauvaise Paye, Guerre nucléaire globale [1].

Si la réponse est mauvaise le script répondra « Erreur, jeu inconnu !! »

---

1 Ce n'est pas une incitation au terrorisme, mais une référence au film culte Wargames de John Badham, en 1985, dans lequel un ado, qui cherche à se procurer, en avant-première, la prochaine version de ses jeux vidéo préférés, se connecte à l'ordinateur du DoD, gérant les réponses à des attaques nucléaires soviétiques ; celui-ci propose différents jeux de stratégie allant des échecs à la guerre nucléaire globale, en passant par la guerre bactériologique. C'est bien sûr la guerre nucléaire que l'ado choisit !

VARIANTE KISS

KISS symbolise ici la philosophie Unix : « *Keep It Simple, Stupid!* »

La réponse sera « Vous avez choisi de jouer [à|au|à la] nom_du_jeu_choisi. »

Si le jeu choisi est Guerre nucléaire globale vous ajouterez à la fin « Est-ce bien raisonnable ?? », sinon vous ajouterez « C'est un bon choix pour aujourd'hui ! »

VARIANTE KICK

KICK symbolise notre philosophie personnelle : « *Keep It Complex, Kid!* », en français : « *Pourquoi faire simple quand on peut faire compliqué !* »

Le jeu commencera par la phrase : Choisissez un jeu SVP !

Les réponses seront :

1. Trax est un jeu qui ne laisse pas de traxe !
2. Ne restez pas sec après avoir Othello !
3. Pente est un bon jeu, mais ne suivez pas une mauvaise pente !
4. Même puissant, un 4x4 est nocif !
5. Ce n'est pas un jeu, c'est la réalité !
6. Ce n'est pas une bonne idée ! J'espère que vous plaisantez !

Les jeux un à quatre seront suivi du message : Excellent choix, mais il va vous falloir programmer nom_du_jeu_choisi pour y jouer :-))

AIDES

L'instruction d'affichage de menu est `select ... do ... done`. L'instruction de choix est `case ... in ... esac`.

Dans l'option *KISS*, vous pouvez employer une variable par jeu & une fois le choix effectué & le jeux traité, regarder s'il s'agit du 6e.

Pour traiter l'option *KICK* nous allons avoir besoin de trois notions que nous n'avons pas encore abordées dans ce cahier : le remplacement d'une partie d'une chaîne de caractères, les tableaux simples & les tableaux associatifs.

Remplacement d'une partie de chaîne

Ce sujet est abordé en détail aux paragraphes Les Opérateurs alpha-numériques de bash p. 53 & Les Opérateurs de la commande expr p. 54. Ils sont repris dans les sous paragraphes du paragraphe Remplacements des paramètres p. 104 à 107. Ainsi que dans l'exercice Ex.0 p. 121.

Tableaux simples

L'idée est de stocker les noms des jeux dans un tableau & de les repérer par un numéro allant de 0 à 5 puisque la première case d'un tableau a toujours le numéro 0.

Ce tableau va nous servir pour l'emploi du tableau associatif.

Tableaux associatifs

Un tableau associatif est, donc, un tableau dans lequel les cases sont repérées par un texte & non par un nombre. Ici, le tableau contiendra les messages propres à chaque jeu & il sera repéré par le nom du jeu stocké dans le tableau précédent.

Il nous faudra donc employer l'opérateur **${ }** comme indiqué dans les paragraphes Les Tableaux simples p. 49 & Les Tableaux associatifs p. 53.

Exercices complexes

Ex. 11 : bataille navale

Analyse niveau 4

L'énoncé & le début de l'analyse sont dans la section Un exemple concret de l'Introduction p. 23 & suivantes.

Nous n'abordons pas encore l'écriture en *bash*, mais nous allons commencer à nous approcher du langage.

Les fonctions 1 (*Afficher la règle*) & 2 (*Afficher le plan d'eau*) seront traduites en *bash*, par l'emploi intensif de la commande `echo`.

Avant de nous lancer dans les fonctions, il faut penser à initialiser le tableau plan_d_eau à 0.

Pour cela, il faut préciser ce que va être plan_d_eau. Pour l'instant, nous avons raisonné, indépendamment du langage avec un tableau à deux dimensions (lignes & colonnes), afin de bien comprendre les traitements effectuer. Comme *bash* ne connaît que les tableaux à une dimension nous allons simuler un tableau à deux dimensions avec un tableau n'en ayant qu'une. Avec des lignes numérotées de 1 à 3 & des colonnes de 1 à 4, sachant que notre tableau aura douze cases numérotées de 0 à 11, & que la ligne 1 commence à la case 0, la 2, à la case 4 & la trois, à la case 8, la formule de conversion sera : num_case ← (num_lig-1)×4)+(num_col-1). Inversement, on aura : num_lig ← (num_case/4)+1 & num_col ← (num_case mod 4)+1.

Pour ce qui est de l'initialisation, il suffit de parcourir le tableau en assignant 0 à chaque case.

Pour nous simplifier la vie, nous pourrions tricher & employer un tableau de 13 cases (0 à 12) dont nous n'utiliserions pas la case 0. Les formules devenant (num_lig)×4)+(num_col), (num_case/4) & (num_case mod 4).

3. *placer_les_bateaux(*bat, nbcas*)*

Le paramètre bat est ici un bateau & nbcas, son nombre de cases.

La variable bateaux contient au départ [0 ,0 ,0] ; c'est ici qu'on lui assigne ses valeurs.

Tous les langages de programmation intègrent un outil de génération de nombres aléatoires. Nous utiliserons celui de *bash* qui

n'est pas une fonction mais une variable prédéfinie, nommée *RANDOM* dont la valeur, comprise entre 0 & 32 767, change aléatoirement chaque fois que l'on s'y réfère. Pour la fonction *hasard*, nous emploierons le reste de la division entière, par 12 ou 4, de *RANDOM*.

La fonction *tirage_correct* doit vérifier que la case obtenue n'appartient pas à un autre bateau & est orthogonale (cf. Seconde_Case & Prolonge_Bat).

- **répéter**
  - case1 ← *hasard*(12)
- **jusqu'à** plan_d_eau[case1]=0
- **si** nbcas>1 **alors**
  - bateaux[1] ← case1
  - **répéter**
    - case2 ← *hasard*(4)
  - **jusqu'à** *tirage_correct*(case1, case2)
  - bateaux[2] ← case2
- **sinon**
  - bateaux[0] ← case1
- **finsi**

0	1–0	2	3
4–1	5	6–2	7
8	9–3	10	11

Dans ce tableau les chiffres de 0 à 11 représentent les indices des cases, les seconds chiffres indiquent la case correspondant au tirage aléatoire de la seconde case si la première est la 5.

Les secondes cases possibles du deuxième bateau sont résumées dans le tableau suivant :

0	1 & 4	4	0, 5 & 8	8	4 & 9
1	0, 2 & 5	5	1, 4, 6 & 9	9	5, 8 & 10

2	1, 3 & 6	6	2, 5, 7 & 10	10	6, 9 & 11
3	2 & 7	7	3, 6 & 11	11	7 & 10

De fait, le placement des bateaux s'avère la partie la plus complexe du script, alors qu'il n'y en a que deux !

La fonction *tirage_correct* ressemblera à :

- **selon** case1 **choisir**
  - 0 :
    - **selon** case2 **choisir**
      - 2: case2 ← 1
      - 3 : case2 ← 4
      - *: case2 ← case1
    - **finchoix** *0*
  - 1 :
    - **selon** case2 **choisir**
      - 1 : case2 ← 0
      - 2 : case2 ← 2
      - 3 : case2 ← 5
      - * : case2 ← case1
    - **finchoix** *1*
  - …
- **finchoix** *case1*
- *tirage_correct* ← plan_d_eau[case2]=0

4. *gagne*

Deux possibilités : totalisation des coups au but & calcul du reste de la somme des valeurs divisée par 10.

Pour la première, là encore, il y a deux façons de procéder : lors de l'analyse des tirs ou après. Dans le premier cas il faudra initialiser le nombre de coups au but avant la saisie du premier tir, puis

l'augmenter jusqu'au total voulu. Dans le second, on vérifie après l'affichage du plan d'eau le nombre de coups au but.

Pour la seconde, le calcul après l'affichage semble plus rationnel.

1 FAÇON 1

Le nombre de case total est de 3

- nb_gagné ← 0 *à l'initialisation du script*
  - nb_gagné ← nb_gagné+1 *lors de l'analyse du coup si la case contient 1 ou 2*
  - gagne ← nb_gagné=3

Avec les multiples de dix les deux dernières lignes deviennent :

nb_gagné ← nb_gagné+plan_d_eau[case] lors de l'analyse du coup si la case contient 10 ou 20

gagne ← nb_gagne **mod** 10=0

2 FAÇON 2

Avant de passer au coup suivant on cumule la valeur des cases de bateaux.

Pour chaque case **répéter**

nb_gagné ← nb_gagné+plan_d_eau[bateaux[case]]

finpour

gagne ← nb_gagné **mod** 10=0

5. *saisie_du_tir(*c_coup*)*

Les constantes clig & ccol pourraient être initialisées après l'affichage de la règle.

La variable tir_ok sert à contrôler la validité des coordonnées saisies.

Le paramètre c_coup contiendra le numéro de la case concernée.

- tir_ok ← faux
- clig ← 'ABC'

- ccol ← '1234'
- **répéter**
  - *afficher* msg_dem_tir
  - *lire les deux caractères du tir* (premier, second)
  - **si** le premier **est dans** clig **&** le second **dans** ccol **alors**
    - tir_ok ← vrai
  - **finsi**
- **jusqu'à** tir_ok=vrai
- c_coup ← (*position de* premier *dans* clig-1)*4+*position de* second *dans* ccol-1

6. *affichage_résultat_coup* (coup)

Le paramètre de cette fonction pourrait être, lors de son appel, la résultat de la fonction précédente.

- **selon** plan_d_eau[coup] **choisir**
  - 0 : plan_d_eau[coup] ← 3
  - 1 :
    - plan_d_eau[coup] ← 10
    - nb_gagné ← nb_gagné+10
    - *afficher* msg_coulé
  - 2 :
    - plan_d_eau[coup] ← 20
    - nb_gagné ← nb_gagné+20
    - **si** nb_gagné ≤ 30 **alors**
      - *afficher* msg_touché
    - **sinon**
      - *afficher* message_coulé
    - **finsi**
  - * : *afficher* msg_erreur_coup_déjà_joué
- **finchoix**

7. devenu sans objet

La Traduction en bash

Il y a deux points à définir :

- les données nécessaires,
- les traitements à effectuer.

Les données

Reprenons les données apparentes

```
01234567890123456789O
-----------------------
| A1 | A2 | A3 | A4 |
-----------------------
| B1 | B2 | B3 | B4 |
-----------------------
| C1 | C2 | C3 | C4 |
-----------------------
```

Bash commence l'indexation des caractères d'une chaîne à zéro, la première ligne indique le rang des caractères, elle ne sert qu'à la programmation & n'apparaîtra pas dans le jeu.

Le première ligne significative est reproduite quatre fois. Elle est constante. La seconde comporte la lettre A aux positions 2, 7, 12 & 17 suivie d'un chiffre indiquant le numéro de colonne. Les deux dernières reproduisent ce schéma avec les lettres B & C.

En pratique, les éléments d'une chaîne sont accessibles grâce à l'opérateur **${**chaîne:position_dans_la_chaîne:1**}** & l'on compte les caractères à partir de zéro !

En première approche, nous pouvons représenter ces données soit par un tableau à deux dimensions de chaînes de caractères, soit par un tableaux associatifs de chaînes de caractères ; soit par des chaînes de caractères indépendantes.

* La première option est impossible en *bash*, mais nous pouvons simuler un tel tableau par un tableau comptant, ici, douze cases contenant des données accessibles par

  $plan_do_ch[index, variant de 0 à 11]

  & valant de '| A1 ' à '| C4 |'.
* La seconde option nécessite des données accessibles par

  ${lignes['index']:position:longueur**}**,

  autrement, dit par

  ${**lignes['A']:0:5**}** ou **${lignes['C']:15:6**}**,

  pour les mêmes valeurs ou cela semble préférable par

  ${**lignes['A']:2:2**}** ou **${lignes['C']:17:2**}**.
* La troisième option nécessite trois chaînes, A, B & C, représentant chacune une des lignes du tableau. L'accès aux données s'y fera par **${**ligne:position:2**}**, autrement dit de **${**A:2:2**}** ou **${**C:17:2**}**.

  Les valeurs iront de 'A1' à 'C4' en passant par '++' & '*i' où i est le numéro du bateau.

Notre préférence va à la troisième solution, pas parce qu'elle est meilleure, mais parce qu'elle permet une gestion simple de l'affichage.

Il nous faut établir une correspondance entre le tableau plan_d_eau & les chaînes A, B & C.

* De quelles données avons nous besoin ?
  - de la ligne constante ('————————————————————') ;
  - des coordonnées des tirs possibles ('A1' à 'C4') ;
  - des chaînes de coups raté ('++'), touché_bateau_1 ('*1'), touché_ bateau_2 ('*2') ;
  - d'une constante indiquant la condition de fin de partie (par exemple '*1*2*2') ou d'une condition logique (somme des cases multiple de 10) ;

- des messages ; ‘Quel est votre tir ? ’ ; ‘Raté !’, ‘Touché !’, ‘Coulé !’, ‘Bravo, vous avez gagné, le droit de rejouer si vous le souhaitez !’ ;
- du tableau de douze nombres plan_do (plus court que plan_d_eau) que nous remplirons de zéros au départ ;
- des emplacements des bateaux (par exemple ‘A1B3C3’, [‘A1’, ‘B3’, ‘C3’] ou [‘A1’, ‘B3C3’]), *[Pour ces deux types de données un tableau de nombre peut s’avérer judicieux puisque les opérateurs ${*tableau} & ${@tableau} permettent de transformer le tableau en une ou plusieurs chaînes de caractères si besoin est !]* ; si l’on choisi la représentation en chaînes de caractères, il faudra un tableau des positions des emplacements dans la chaîne (‘024’ ou [0, 2, 4]) ;
- du coup joué (2 caractères) ;
- des lignes, A, B & C, initialisées avec les valeurs de l’affichage de départ ;
- éventuellement d’une variable fini (initialisée avec les emplacements) que la comparera à la constante fin ;
- de variables auxiliaires servant à faciliter l’écriture du programme.

Commandes

Commandes internes =, test, function, until, if, for, case, test, echo, read ; commande externe clear.

Aide

Ce programme présente une difficulté : le travail avec des chaînes de caractère, dans un langage qui propose peu d’outils pour cela (avec, en outre, une syntaxe absconse) [01023].

La Manipulation des chaînes de caractères

Le paragraphe La Traduction en bash p. 139 montre comment nous avons établi ce qui suit.

Les outils dont nous disposons :

- la connaissance des positions des lettres dans les chaînes A, B & C (tableau poslet [2, 7, 12, 17] ou chaîne poslet ' 2 71217') ;
- la connaissance de la position des cases des bateaux dans la chaîne emplacement (tableau posbat [0, 2, 4] ou chaîne posbat '024') ;
- l'opérateur **${**chaine**:**$pos**:**$long**}** qui permet d'extraire une sous-chaîne ;
- l'opérateur d'accès aux cases d'un tableau **${**poslet[0]**}** ;
- la conversion des chaînes en nombres : l'opérateur **$(( ))**, qui permet d'évaluer une expression arithmétique, transformera la chaîne ' 3' en nombre *3*, mais pas la chaîne '03'.
- la conversion des coordonnées en cases du plan d'eau, les coordonnées de A1 à A4 occupant les cases 0 à 3, celles de B1 à B4, les éléments 4 à 7 & celles de C1 à C4 les cases 8 à 11.

Que le coup 'A2' soit raté ou réussi nous devons reconstituer la chaîne A en assemblant le début de A, avant A2 (les caractères 0 à 6, dans ce cas) le résultat du coup (les caractères '++', '*1' ou '*2' en positions 7 & 8, dans ce cas) & la fin de A, après A2 (les caractères de 9 à 20, dans ce cas).

Nous avons déterminé la position de 'A2' dans A en regardant les lignes ci-dessous.

```
00000000011111111112
01234567890123456789
| A1 | A2 | A3 | A4 |
```

Comme le programme ne voit rien, il va falloir lui traduire la formule que nous avons présentée p. 134.

Ces trois lignes nous permettent de dire que 2, 7,12 & 17 correspondent au rang de début des sous-chaînes 'A1' à 'A4'.

Nous rangeons ces rangs dans le tableau postir qui va contenir (2 7 12 17) afin que tous les nombres aient deux caractères. Le rang de

début de 'A2' est donc ' 7'. C'est la deuxième sous-chaîne de postir, mais son rang de début dans postir n'est pas 3 (2 (deuxième lettre du coup A2)*2 (nombre de lettres d'un coup)-1 (première lettre du coup)) mais 2 – (2*2-2), car le rang du premier caractère d'une chaîne bash est 0 (c'est vrai dans tous les langages s'inspirant plus ou moins du C, comme PHP, Perl, etc.)

La deuxième lettre de coup est obtenue par ${coup:1}.

```
let index=$((${coup:1}*2-2))
let longdeb=$((${postir:$index:2}-1))
let debfin=$((${postir:$index:2}+2))
A=${A:0:$longdeb}$resultcoup${A:$debfin}
```

En clair, on juxtapose le début de la ligne du coup, avec le résultat du coup & la fin de la ligne du coup.

Ce calcul fera l'objet d'une fonction. Le tir ayant déjà été analysé, puisqu'on connaît son résultat, cette fonction aura comme paramètres la ligne & le numéro de case à traiter (On remplacera A par 1 & ${coup:1:1} par $2).

La fonction d'affichage du plan d'eau est très simple : elle commence par un effacement d'écran puis affiche les chaînes de caractères en alternant la chaîne constante & les chaînes variables.

La fonction de saisie & de validation d'un tir commence par la saisie des deux caractères du coup, suivie d'une itération vérifiant que le premier caractère est compris entre dans 'ABC' & le second dans '1234'. Si ce n'est pas le cas elle redemande un tir.

La fonction d'effet du coup n'est pas très complexe.

Si le coup correspond à la première case du tableau emplacements, on inscrits deux *1 à la place, s'il correspond à une des deux autres cases on inscrit des *2 à la place. Dans le premier cas le bateau est coulé, dans le second il est touché si une des deux cases

contient des coordonnées, coulé sinon. Si toutes les cases sont touchées la partie est terminée !

La fonction de tirage au sort des emplacements est la plus complexe.

Sa première partie génère un nombre entre 0 & 11, au hasard, & le converti en une lettre & un chiffre.

Sa seconde partie vérifie que les coordonnées du bateau 2 sont adjacentes entre elles mais pas avec celles du bateau 1.

Il lui faut également vérifier que les deux bateaux ne se touche ni ne se recouvre.

En d'autres termes si le bateau 1 est en B2, le bateau 2 ne peut pas avoir de cases en B2, A2, C2, B1 & B3. C'est le point délicat.

Ex. 12 : sauvegarde

Énoncé

Écrire un script qui assure la sauvegarde dans le dossier /var/tmp/sauve des fichiers de votre dossier de connexion non encore sauvegardés & qui affiche selon le travail réalisé :

- Le fichier NomDuFichier est déjà sauvegardé !,
- ou Sauvegarde du fichier NomDuFichier en cours !

Commandes

Cet exercice nécessite d'employer les commandes test, for, if, in, echo, cp, mkdir & find. Le même résultat peut être obtenu de façon plus complexe en employant ls -R au lieu de find.

Aide

Analyse de l'énoncé

Il doit exister un dossier de sauvegarde. Il ne peut pas se trouver dans votre dossier de connexion, sinon l'opération de sauvegarde le

sauvegarderai. Dans une entreprise, il se trouverai sur un périphérique destiné à ce besoin (disque USB amovible, SAN , NAS, dévédérom, etc.), dans le cadre du TP nous emploierons un dossier sauve dans /var/tmp ou un dossier sauve dans /home.

Il faut parcourir la liste des fichiers en testant pour chacun s'il a déjà été sauvegardé. Si oui afficher un message d'information le disant, sinon afficher un autre message disant le contraire & copier le fichier.

Dans un premier temps on considérera qu'un fichier a déjà été sauvegardé, s'il est présent dans le dossier.

Quand le script fonctionnera, on considérera que la sauvegarde a été faite, si sa date de dernière mise à jour est au moins aussi récente que celle du fichier à sauvegarder.

Comment détermine-t-on les commandes nécessaires

En adaptant l'analyse précédente à l'interpréteur concerné *bash*.

* C'est `echo` qui permet d'afficher les messages (PDTU 1, fascicule 2 Annexe 3, ce document).
* Faire une sauvegarde, c'est copier des fichiers, ce que fait la commande `cp` (présentée fascicule PDTU 1 & TP1).
* Ces fichiers sont dans des sous-dossiers de votre dossier de connexion. Il faut conserver votre organisation en sous-dossiers, la commande `find` (cf. PDTU 1 & TP2) liste tous les fichiers voulus avec leur chemin absolu. La commande `ls -R` permet, également de lister tous les fichiers. Nous lui préférerons `find`, car, comme elle ne fournit pas le chemin absolu d'accès aux fichiers, il faudrait effectuer un traitement plus complexe.
* La commande `mkdir` permet de créer les sous-dossiers qui n'existent pas encore dans le dossier de sauvegarde.
* Il vous faut tester si le ficher a déjà été sauvegardé ou non (commandes `if` & `test` – cf. fascicule 2 Annexe 3 & début de ce document).

* Il faudra répéter la copie pour chaque fichier, c'est le rôle de la commande for.
* La commande in introduit une liste de données : la liste des noms de fichiers résultant de l'exécution de la commande find.

Ex. 13 : occupation des partitions

Énoncé

Écrire un script permettant de suivre l'occupation du disque correspondants aux répertoires de connexion des utilisateurs (dans /home donc).

Ce script pourra s'exécuter périodiquement (gestion par `crond`). Exécuté interactivement, il demandera une valeur limite pour chaque dossier de connexion que vous stockerez dans le fichier /var/tmp/admin/quotas. Créez le dossier /var/tmp/admin, en ligne de commande, s'il n'existe pas.

Dans les deux cas, il émettra une alarme si un seuil est dépassé, sous forme de mail à l'administrateur.

Commandes

Les commandes `test`, `clear`, `rm`, `cat`, `du`, `ls`, `if`, `read`, `grep`, `cut`, `echo`, `while`, `for`, `exit`, `mail`, les tubes (pipes) & la redirection seront nécessaires pour réaliser ce script.

Aide

Dans un premier temps, il faut demander pour chaque utilisateur son quota. Afin de simplifier, nous allons supposer que le dossier /home ne contient que des dossiers de connexions & qu'ils portent tous le nom de connexion d'un utilisateur. Nous emploierons la liste de ces dossiers pour demander les seuils. Sinon il nous faudrait extraire les noms d'utilisateurs & les répertoires de connexion du fichier /etc/passwd.

Ensuite nous utiliserons la commande `du` pour comparer les tailles réelles & le seuil en mégaoctets.

EX. 14 : LISTE DES MACHINES CONNECTÉES

ÉNONCÉ

Écrire un script permettant de vérifier la connexion, avec la commande ping, d'un ensemble d'adresses IP.

Le script devra :

- permettre la saisie des adresses IP devant être vérifiées au niveau connexion ;
- donner l'état connecté ou non de chaque adresse ;
- donner l'adresse MAC des machines connectées ;
- donner le nom (DNS) des machines connectées lorsque celui-ci est défini.

COMMANDES

Utilisation des commandes echo, read, test, for, cut, arp, while, case, tr, grep, return, sed, less, more, ping, nslookup & d'expressions rationnelles & de mécanismes de redirection & de flux.

AIDE

Pour accepter la saisie d'une adresse IP, il nous faut nous assurer de sa validité. Cela est impossible, sans avoir travaillé les expressions rationnelles. Si vous avez sauté le paragraphe, c'est le moment de le lire. Ensuite, il faut pouvoir les ajouter ou les supprimer de la liste. La vérification se fera lors de l'ajout & lors de la suppression.

Il faudra ensuite pouvoir lister toutes les adresses traitées seules ou avec les informations demandées (état de la connexion, nom adresse MAC & nom DNS éventuel).

Compte tenu de la plus grande difficulté de cet exercice, voici un embryon d'algorithme.

```
si la liste des adresses IP n'existe pas alors
    la constituer en demandant des adresses & en les stockant dans
```

```
    un fichier.
sinon
    ajouter ou supprimer des adresses dans le fichier
fin si
valider la liste
pour chaque adresse répéter
    chercher les informations demandées
    les afficher
fin pour
```

Quand on veut contrôler une saisie on emploie une expression rationnelle. Pour écrire l'expression rationnelle décrivant une adresse IP, il faut définir précisément ce qu'est une adresse IPV4.

Il ne suffit pas de dire que c'est une suite de quatre entiers dont la valeur est comprise entre 0 & 255. Il faut analyser plus finement la représentation de ces nombres.

* Entre 0 & 99, on peut dire qu'il s'agit d'un ou deux chiffres de 0 à 9.
* Entre 100 & 199 il s'agit d'un 1 suivi de deux chiffres allant de 0 à 9.
* Entre 200 & 249 on peut dire qu'il s'agit des nombres entre 20 & 4 suivi d'un chiffre allant de 0 à 9.
* Entre 250 & 255 du nombre 25 suivi d'un chiffre entre 0 & 5.

On répète 4 fois cette séquence en plaçant un point pour séparer les nombres.

C'est ce que décrit la séquence suivante :

```
# Elle délimite un nombre suivi d'un point (cas des 3 premiers
octets). Elle délimite les écritures possibles d'un nombres compris
entre 0 & 255.
[0-9][0-9]? # Décrit les nombres compris entre 0 & 99. En toute
rigueur il faudrait les faire précéder d'un [0] pour le cas où l'on
voudrait faire précéder le nombre d'un 0 non significatif. Le « ? »
```

indique qu'il peut y avoir 1 ou 2 chiffres.
| *# ou*
1\d\d *# Décrit les nombres compris entre 100 & 199. Le « \d » remplace [0-9] dans les expressions rationnelles étendues. Pour être employée avec grep, il faudra ajouter l'option -E.*
| *# ou*
2[0-4]\d *# Décrit les nombres entre 240 & 249.*
| *# ou*
25[0-5] *# décrit les nombres entre 250 & 255.*
) *# Regroupe toutes les descriptions de nombres.*
\. *# Ajoute un « . » à la fin de la description.*
) *# Regroupe la description de nombre & le point.*
{3} *# Répète trois fois la séquence précedente.*
(\d\d?|[0-1]\d\d|2[0-4]\d|25[0-5]) *# reprise de la description de nombre pour le dernier octet, car il ne se termine pas par un point.*
 # On fera précéder la première séquence par « ^ » pour indiquer qu'elle se trouve en début de chaîne & on fera suivre la seconde d'un « $ » pour indiquer qu'elle se trouve en fin de chaîne.

Au « \d » près c'est ce que LAURENT à écrit & c'est correct. Cependant, quand on regarde d'un peu plus prêt, on réalise que l'on peut traiter la première séquence comme la seconde car le traitement des nombres de 0 à 99 est le même que celui des nombres de 100 à 199. La séquence s'écrira donc

(([0-1]**)**\d\d**?|**2**(**[0-4]**)**\d**|**25**(**[0-5]**))**\.**){**3**}(**[0-1]**)**\d\d**?|**2**(**[0-4]**)**\d**|**25**(**[0-5]**))** ou en employant la numérotation des sous expression

(([0-1]**)**\d\d**?|**2**(**[0-4]**)**\d**|**25**(**[0-5]**))**\.**){**3**}(**\2**)**.

En faisant la même analyse de la présentation traditionnelle d'un numéro de téléphone, à la fois similaire & plus simple, vous devriez aboutir à une expression comme celle-ci : **(**0\d**)((**\.\d\d**){**4**})**.

Ex. 15 : consultation des logs personnalisée

Énoncé

Écrire un ou plusieurs scripts permettant de consulter le fichier log (/var/log/messages) de manière simple.

Les scripts devront :

- permettre la saisie de mots clés dont on veut les lignes correspondantes (nom du démon concerné par exemple) ;
- permettre de limiter par la date les informations extraites du fichier ;
- permettre de visualiser les seules informations nouvelles depuis la dernière consultation ;
- offrir un mécanisme d'effacement du fichier (effacement jusqu'à telle date).

Commandes

Utilisation des commandes `echo`, `sort`, `cut`, `read`, `if`, `case`, `while`, `cp`, `for`, `cat`, `tail`, `head`, `mv`, `grep`, `stat`, `sed`, `rm`, de la redirection, des pipes & des fonctions.

Aide

Le plus simple est de commencer par écrire quatre scripts, chacun dédié à une des requêtes. C'est l'esprit ***KISS*** (*Keep It Simple Stupid !*) d'Unix ! Bien que l'énoncé précise le fichier /var/log/messages, il serait judicieux de passer le nom du fichier en paramètre de façon à pouvoir employer le script avec n'importe quel journal. Dans ce cas, le script ne fonctionnera qu'avec des fichiers ayant une organisation semblable à celle de /var/log/messages, c'est dire débutant par la date du message.

⁕ Le cœur du premier script est la commande `grep -E`. Elle sera entourée d'instructions de traitements des paramètres essentiellement des noms de démons. Comme il sera possible d'en saisir plu-

sieurs par ligne, il faudra composer l'expression rationnelle permettant de les afficher toutes (liste mélangée) ou employer plusieurs instructions `grep` (listes distinctes).

* Le second script demandera une ou deux dates & affichera les informations postérieures à la date s'il n'y en a qu'une & comprise entre les dates s'il y en a deux.
* Le troisième isolera la date du résultat de la commande `stat` appliquée au fichier passé en paramètre & affichera les logs ultérieurs.
* Le quatrième effacera toutes les lignes antérieures à une date donnée.
* Les quatre premiers scripts ne concerneront que le fichier /var/log/messages.
* Un cinquième script combinera les quatre précédents & les étendra à tous les fichiers de log, afin d'illustrer le passage des paramètres.

Si *systemd* est installé le fichier /var/log/messages n'existe plus & les log sont consultables à l'aide de la commande `journalctl`. I faut donc rediriger le résultat de cette cmomande dans un fichier pour employer ces scripts.

SCRIPT 1

```
tant qu'il y a des noms de démons répéter
  afficher le nom du démon
  extraire les lignes contenant ce nom
  afficher une ligne de séparation
fin tantque
```

SCRIPT 2

1 ALGORITHME

```
s'il y a des paramètres alors
  vérifier l'existence du fichier
```

```
  vérifier que ce sont des dates
  s'il y a deux paramètres alors
    date_deb ← première date
    date_fin ← deuxième date
  sinon
    s'il n'y en  a qu'une
      date_deb ← première date
      date_fin ← première date
  finif
  date_cour ← date_deb
  tant que date_cour<=date_fin répéter
    extraire les lignes contenant date_cour
    augmenter date_cour d'un jour
  fin tantque
sinon
  afficher usage
finsi
```

2 Aides à la traduction

Remarque 1 *: L'écriture de l'algorithme montre trois types de traitement, la traduction pourrait donc employer une instruction de choix multiple au lieu d'alternatives imbriquées.*

Remarque 2 : Les lignes commençant par un verbe (vérifier, extraire, augmenter, afficher) pourrait faire l'objet d'une définition de fonction, mais ce n'est pas une obligation, les fonctions servant à simplifier la lecture & donc la mise au point du script.

La principale difficulté de la traduction est d'augmenter la valeur de la date. En effet, le contenu des variables date_deb, date_fin & date_cour est un texte de la forme aaaa-mm-jj. Il faut donc le

convertir en nombre pour l'augmenter d'un jour & le reconvertir en date. Cette conversion n'est pas tout à fait intuitive !

En fait, en informatique les dates sont des nombres. Dans les systèmes *Unix*, le premier jour est le 1er janvier 1970, date officielle de la naissance d'*Unix*. La commande `date` permet d'afficher une date, avec un format précis. Sans précision, elle affiche la date du système, avec l'option `-d` elle affiche la date dont la valeur suit. Jusque là, il suffit de lire la page `man` consacrée à cette commande. Là où cela se complique c'est que cette page n'indique pas comment augmenter la valeur de la date d'un jour, mais elle signale que le manuel complet de la commande est accessible avec la commande `info`. En lisant les pages la concernant dans `info`, & en réfléchissant un peu, vous découvrirez que l'on peut préciser un décalage en jours, en mois ou en année en ajoutant un nombre suivi des mots day, month ou year. La commande pour augmenter la date d'un jour sera donc :

```
date +%Y-%m-%d -d "$date_cour +1 day",
```

qu'il faudra intégrer dans le script bash en tenant compte des contraintes du langage.

SCRIPT 3

1 ALGORITHME

```
vérifier le fichier
date_deb ← date de dernière consultation du fichier
date_fin ← date du jour
date_cour ← date_deb
tant que date_cour<=date_fin répéter
  extraire les lignes contenant date_cour
  augmenter date_cour d'un jour
fin tantque
```

2 AIDES À LA TRADUCTION

Les fichiers Unix ont plusieurs dates. La commande stat permet de les afficher.

```
$ stat sans\ titre
  Fichier : « sans titre »
   Taille : 31824       Blocs : 64         Blocs d'E/S : 4096   fichier
Périphérique : 806h/2054d      Inœud : 1604       Liens : 1
Accès : (0755/-rwxr-xr-x)  UID : ( 1002/ mmichek)   GID : (  100/
users)
 Accès : 2014-12-30 07:36:13.513750488 +0100
Modif. : 2014-03-23 17:40:22.000000000 +0100
Changt : 2014-11-22 15:30:54.920858580 +0100
  Créé : -
```

Attention : si vous travaillez en utilisateur root*, contrairement à ce qu'il faudrait faire, le résultat sera en anglais.*

La commande qui permet d'obtenir la date de dernière consultation est la suivante :

```
stat sans\ titre |grep Accès|grep -v UID|cut -d\  -f3
```

Le surlignement fait apparaître les espaces indispensables dans ce cas. Si le script est exécuté par root, il faudra remplacer Accès par Access, ou pour simplifier par Acc valable dans les deux langues.

La commande nécessaire pour obtenir la date du jour au format nous intéressant à été employée dans l'exercice précédent.

Remarque : fondamentalement ce script ne diffère du précédent que par l'initialisation des variables date_deb *&* date_fin *qui est calculée au lieu d'être fournie par les paramètres* $1 *&* $2*.*

Script 4

Attention il faut éviter d'effacer des données dans les fichiers journaux. L'utilitaire logrotate est là pour ça ! Il permet d'archiver pério-

diquement les journaux. Nous ferons donc une copie du fichier dans le dossier /var/temp, avant d'effacer quoique ce soit.

1 ALGORITHME 1

```
fichier ← premier paramètre
vérifier l'existence de fichier
copier le fichier fichier dans /var/tmp
récupérer la date de début & la mettre dans date_cour
date_deb ← deuxième_paramètre
tant que date_cour <= date_fin répéter
  supprimer la ligne
  augmenter date_cour d'un jourdeb
fin tantque
```

2 AIDES À LA TRADUCTION

La date de début commence la première ligne, la commande `head -n1 $fich|cut -c1-10`, la récupérera.

La difficulté, ici s'avère la suppression de la ligne. C'est la commande `sed` qui le permettra avec une syntaxe proche de celle-ci :

```
sed -n -i "/^$date_cour/d" $fich
```

En tapant `info sed` sur une console ou en cherchant sur Internet vous trouverez la signification des deux options (`n` & `i`) & de la sous-commande (`d`).

3 ALGORITHME 2

Toutefois quand on réfléchit un peu, beaucoup, énormément, on se dit que supprimer les lignes de début de la copie ou extraire les lignes de fin de l'original c'est la même chose. L'algorithme pourrait donc s'écrire :

```
vérifier le fichier
date_deb ← deuxième_paramètre
```

```
date_fin ← date_du_ jour
date_cour ← date_deb
tant que date_cour<=date_fin répéter
  extraire les lignes contenant date_cour
  augmenter date_cour d'un jour
fin tantque
```

Ce qui revient à appeler le script 2 en passant la date de début & en calculant la date de fin, comme indiquée dans le script 3.

C'est une des utilités de réfléchir avant d'agir que de permettre une économie substantielle d'énergie.

Script 5

Ce script va admettre quatre options :

- -d qui affiche les messages relatifs aux démons dont le nom suit ;
- -p qui affiche les messages relatifs à une période donnée ;
- -c qui affiche tous les messages depuis la dernière consultation ;
- -e qui efface les messages antérieurs à une date donnée.

Pour simplifier, nous limiterons l'emploi à une commande à la fois. La logique voudrait que l'on puisse combiner l'option -d avec une des autres.

De plus comme nous avons constaté des actions similaires dans les différents scripts, nous écrirons des fonctions qui seront communes aux cinq scripts. Il faudra bien sûr modifier les scripts pour en enlever les fonctions & introduire éventuellement leur appel.

Le corps du script consistera dans l'appel des quatre premiers scripts après avoir décodé la ligne de commande.

En pratique comme les scripts trois & quatre sont des appels du deux on pourrait se contenter d'écrite les scripts 1, 2 & 5 & le fichier contenant les fonctions.

1 Algorithme

```
# Récupération des fonctions
```

```
  Insérer le fichier les contenant
# Analyse des paramètres
  si le premier paramètre n'est pas une des options alors
  afficher un message d'usage
sinon
  selon sa valeur
    -d) :
          analyse des paramètres suivants
          script_1
        fin -d
 … etc.
  fin selon
fin si
```

2 AIDES À LA TRADUCTION

Les fonctions vous sembleront évidentes quand vous aurez écrit les quatre premiers scripts.

Il vous faudra peut-être les modifier pour les faire fonctionner dans tous les scripts.

Pour inclure le fichier de fonctions dans le ou les scripts il suffit d'ajouter après le *shabang*, la ligne

```
. chemin_absolu_du_fichier
```

Une autre difficulté apparaît celle du transfert des paramètres. Si leur nombre était le même pour les quatre scripts, il n'y aurait pas de problème, mais ceux de script1 peuvent varier de 1 à, au plus, 5, (Au-delà, il serait bon que vous preniez des vacances !) Comme nous ignorons leur nombre précis, il nous faudrait employer une itération pour traiter chacun des paramètres comme le fait LAURENT, avec la syntaxe **${!**arg_index**}**, mais cela ferait double emploi avec le traitement des paramètres déjà fait dans les scripts. En consultant la liste des paramètres spéciaux on note l'existence du paramètre

$@, qui placé entre guillemets récupère la liste des paramètres tout en conservant les paramètres séparés. Nous sommes sauvés.

EX. 16 : CONVERSION DÉCIMAL-BINAIRE

ÉNONCÉ

Voici un exercice relativement facile : il s'agit d'écrire un script qui lorsqu'on lui passe en premier paramètre un des mots (et, ou, non) & en second & troisième paramètres deux nombres compris entre 0 & 255 effectue l'opération bit à bit correspondante & afficher le résultat en base 2.

Il vous faudra tester les paramètres.

AIDE

Pour vous faciliter le travail, le script suivant, convdecbin.sh, à transformer en fonction, effectue la conversion décimal binaire. Il est optimisable.

```
#!/bin/bash
let nb=$1
resultat=""
let reste=0
let puis2=1
let exp=0
if [ $1 -lt 2 ]; then
  echo $1" en base 10 s'écrit "$1" en base 2."
  exit
fi
while [ $1 -gt $puis2 ]; do
  reste=$(($nb % 2))
  nb=$(($nb / 2))
  let exp+=1
  puis2=$(($puis2 * 2))
  if [ $puis2 -eq $1 ]; then
```

```
    resultat="1"$reste$resultat
  else
    resultat=$reste$resultat
  fi
done
echo $1" en base 10 s'écrit "$resultat" en base 2."
```


Ex. 17 : compréhension d'un script

Énoncé

Une fois que cela sera fait, écrivez l'algorithme de la fonction start du script /etc/init.d/vboxadd (Cela suppose que *VirtualBox* soit installé sur votre système. Si ce n'est pas le cas, installez-le !),en consultant la liste complète de ce fichier pour obtenir les informations manquantes & les pages concernées de man ou d'info ou, si vous avez du temps à perdre, des sites web !

Notez les éventuelles bizarreries.

Aide

Pour arriver à faire ce travail, il faut appliquer quelques conseils.

* Cherchez le sens des variables d'environnement.
* Lisez le script d'origine : /etc/init.d/vboxadd.
* Faites des suppositions sensées sur les mots dont vous ignorez le sens comme begin ou fail par exemple.
* Testez, quand c'est possible, les commandes dans un terminal.
* Cherchez l'action des options de commande, ne vous contentez pas de regarder leur résultat.
* Relisez les explications relatives à l'instruction bloc.

Liste de la fonction à analyser

```
1  function start ()
```

```
2   {
3       begin "Starting the VirtualBox Guest Additions ";
4
5       if [ -x /usr/bin/systemd-detect-virt  ]; then
6         if [ "x$(systemd-detect-virt)" != "xoracle"  ]; then
7             fail "Not running on a virtualbox guest"
8         fi
9       fi
10
11      uname -r | grep -q -E '^2\.6|^3' 2>/dev/null && ps -A -o
  comm | grep -q '/*udevd$' 2>/dev/null || no_udev=1
12      running_vboxguest || {
13          rm -f $dev || {
14              fail "Cannot remove $dev"
15          }
16
17          rm -f $userdev || {
18              fail "Cannot remove $userdev"
19          }
20
21          $MODPROBE vboxguest >/dev/null 2>&1 || {
22              fail "modprobe vboxguest failed"
23          }
24          case "$no_udev" in 1)
25              sleep .5;;
26          esac
27      }
28      case "$no_udev" in 1)
29          do_vboxguest_non_udev;;
30      esac
```

```
31
32      running_vboxsf || {
33          $MODPROBE vboxsf > /dev/null 2>&1 || {
34              if dmesg | grep "vboxConnect failed" > /dev/null 2>&1;
  then
35                  fail_msg
36                  echo "Unable to start shared folders support.  Make
  sure that your VirtualBox build"
37                  echo "supports this feature."
38                  exit 1
39              fi
40              fail "modprobe vboxsf failed"
41          }
42      }
43
44      # This is needed as X.Org Server 1.13 does not auto-load the
  module.
45      running_vboxvideo || $MODPROBE vboxvideo > /dev/null 2>&1
46  # 5 lignes de commentaires supprimées
47      succ_msg
48      return 0
49  }
```

Conclusion

Si vous avez réussi vous êtes prêt à à aborder un langage de script très différent, moins complexe sur certains plans & beaucoup plus sur d'autres, puisqu'il gère des objets : *PHP*.

Notes

01001

M^r^ Colombo exagère un tantinet !

01002

Un langage de programmation est toujours composé d'un programme traduisant les mots & phrases selon, le vocabulaire & la syntaxe du langage en langage machine ou de pseudo-machine (processeur virtuel comme le processeur Java).

Ils sont de deux sortes :

* les compilés qui comme un traducteur de livre, traduisent une fois le programme écrit complètement
* les interprétés qui travaillant comme un interprète traduisent une ligne après l'autre.

Aujourd'hui, les interpréteurs sont si rapides, que le travail du programmeur n'est pas très différent, dans les deux cas ; seule la vitesse d'exécution diffère, les programmes compilés s'exécutant plus rapidement.

Avant l'arrivée du Turbo Pascal de la défunte société Borland, les compilations de gros programmes pouvait durer plusieurs jours ; actuellement les compilations ne dépassent qu'exceptionnellement l'heure.

Une pseudo machine ou un pseudo-processeur est un processeur virtuel qui permet d'exécuter un programme sur des ordinateurs ayant des processeurs différent, sans qu'il soit nécessaire de le retraduire, s'il est compilé. C'est le cas du langage Java & c'est la raison de la nécessité d'installer une machine virtuelle Java (un programme traduisant le langage du pseudo-processeur Java en langage machine de l'ordinateur), afin d'exécuter des programmes ou des scripts java.

01003

Les langages, Perl, Python & PHP sont des logiciels sous licence GPL ; le langage bash, est inclus dans l'interpréteur de commande du même nom qui est partie intégrante des distributions Unix & Linux ; Javascript est une propriété de la société Novell, Jscript, PowerShell, C#, VBA (Visual Basic for Application) & VBScript (Visual Basic Script) de Microsoft & Java de la société Oracle.

01004

Une valeur est :

- un nombre entier ou décimal, signé ou non-signé ;
- une chaîne de caractère (succession de caractères n'ayant pas forcément un sens, exemple "qg,:!45Q", "Arthur") ;
- une valeur logique (vrai ou faux), ces valeurs tendent à être remplacées par des valeurs numérique, 0 valant vrai ou faux, 1 ou une autre valeur que 0, le contraire ;
- une valeur prise dans une énumération ("vrai", "faux" ; "est", "sud", "ouest", "nord") ;
- un ensemble structuré d'information (tableau, fiche, image, son, fichier, lien vers une URL, etc.)

Dans les langages de scripts il s'agit soit d'un nombre, soit d'une chaîne de caractère ou d'un tableau.

01104

La notation des tableaux que nous avons retenue dans nos algorithmes n'a aucun rapport avec celles du *bash* ou du *PHP*.

01005

Nous avons supposé que vous aviez les moyens d'acquérir un robot ayant deux bras, deux mains & plus de deux doigts par main.

01006

La coloration syntaxique consiste à afficher les différents *mots* de votre programme de différentes couleurs en fonction de leur catégorie (mot-clé, valeur numérique, opérateur arithmétique, etc.)

Nous avons notre système de coloration syntaxique, qui s'avère suffisamment détaillé pour éviter la plupart des ambiguïtés. Cependant, certains mots ayant plusieurs fonctions, il n'est pas toujours possible de les colorer uniformément. Ainsi le mot if est un **mot réservé**, c'est également une commande interne & une **instruction de structuration**. Il en est de même pour des caractères isolés comme les crochets carrés , [&] qui peuvent être considérés comme :

* un **mot réservé** ;
* une commande interne quand ils remplacent la commande test ;
* un opérateur ou un élément d'expression régulière ;
* un **opérateur** quand ils sont doublés ;
* un élément de variable quand ils repèrent un élément dans un tableau ;
* un méta-symbole quand ils indiquent le caractère facultatif d'un paramètre ou d'une option.

Comme l'erreur est humaine, malgré l'emploi de macro-commandes initiales, il peut y avoir des changements de couleurs intempestifs, suite à des corrections ou à des modifications !

01206

L'auto-complétion consiste à compléter automatiquement la saisie que vous êtes en train de réaliser. Il ne faut pas la confondre avec les suggestion de saisie fournies par les navigateurs ou les *intelliphones*. Dans ces dernières, le logiciel vous propose plusieurs choix basés sur vos dernières saisies ou sur des listes de mots fréquemment saisies. Dans la première, la liste se trouve dans des listes prédéfinies fixes : liste des fichiers d'un

dossier ou liste des mots d'un langage & rien n'est suggéré ; s'il y a ambiguïté, le logiciel s'arrête avant l'ambiguïté dans l'attente d'une frappe.

01007

La commande vimtutor vous le propose en anglais & vimtutor fr en français. La réalisation de tous les exercices demande trois heures au maximum. Il ne faut pas hésiter à les refaire pour mieux assimiler !

01008

Il existe une autre commande d'affichage printf dont l'emploi serait préférable pour deux raisons :

* elle permet des formatages précis & sophistiqués des données ;
* elle est compatible avec la norme POSIX.

Cependant echo reste plus employée en raison de sa simplicité & de l'absence de besoin de présentation pour la majorité des commandes en mode texte.

01009

Pour deux raisons :

* en mathématique, il n'y a pas de dissociation entre le nom de la variable & sa valeur. Le signe = signifie la valeur de la variable vaut. En informatique il y a dissociation, l'assignation modifie le contenu de la variable, l'égalité non ;
* en mathématique les seules valeurs sont numériques ; en informatique, elles peuvent être numériques, alphanumériques, ou binaires (images, sons, etc.)

01010

S'il emploie l'opérateur modulo, c'est parce qu'il ne travaille que sur des entiers : on peut changer les bits dans un octet, mais on ajoute, on supprime ou on copie des octets entiers. De fait, echo $((5/2)) affiche 2 & non 2,5.

01110

Remarque concernant ces quatre opérateurs (**#**, **##**, **%**, **%%**) Pour que cela fonctionne, il faut deux conditions :

* que la chaîne débute (**#** & **##**) par le caractère de la sous-chaîne ou qu'elle se termine par le caractère de la sous-chaîne (**%** & **%%**) ;
* que la chaîne contienne un élément variable (*) pour les opérateurs **##** & **%%**.

Ces opérateurs ne semblent présenter d'intérêt que pour modifier le début ou la fin des noms de fichiers.

01210

Les opérateurs de comparaisons sont *égal*, *différent*, *inférieur*, *inférieur ou égal*, *supérieur* & *supérieur ou égal*. Le langage de programmation de *bash* n'inclut aucun d'entre eux. De fait, celle-ci sont réalisées grâce aux commandes **test** & **expr**.

01310

Il est légitime de considérer les signes **[** & **]** après **if**, comme des commandes internes de *bash*, puisqu'ils remplacent la commande interne **test**.

01410

S'il vous arrive de poser sur un forum une question idiote comme : *Quand je fais* **ls -as**, *je vois les fichiers . & .., alors que je ne voudrais pas les voir. Comment faut-il faire pour ne plus les voir ?*

Vous risquez de vous attirez une réponse en quatre lettre : *RTFM* ou *Read The Fucking Manual*, initialement *Read The Fine Manual*, un sigle anglais devenu une expression d'argot Internet, qu'on peut traduire par *Regarde Ton Fichu Manuel.*

01510

Bash est extrêmement laxiste, il ne vous signalera pas nécessairement une erreur (mais il fera peut-être n'importe quoi !) ou, s'il vous en signale une,

elle vous semblera n'avoir aucun sens dans le contexte. Les deux cas les plus fréquents sont les mots *test* (commande interne du bash) & *case* (mot réservé) !

01610

Cette liste est tirée de la page ***bash*** du manuel unix. Elle nous laisse perplexe : aucun nom de variables ou de fonctions peut être !, [[,]], {, }, donc on ne peut les redéfinir ! Il en est de même pour (, [,)),], &, |, &&, ||, ;, .. & $. Pourquoi ces signes ne seraient-ils pas des mots réservés ?

01011

Une famille de norme de systèmes ouverts basée sur Unix™. Le shell ***bash*** est concerné par la norme POSIX 1003.2, relative au shell & aux outils standards.

C'est un peu le standard officiel qui définit les interfaces communes à tous les systèmes de type Unix (Les quatre premières lettres forment l'acronyme de *Portable Operating System Interface* –interface portable de système d'exploitation– & le X exprime l'héritage UNIX.)

Mais il a deux gros défauts : sa documentation coûte très cher & la certification encore plus ! De fait, aucune distribution libre, qu'elle soit Linux ou Unix BSD n'est certifiée ; seuls les Unix propriétaires comme Solaris, HP-UX ou Mac Os X le sont.

Si votre programme doit fonctionner sur d'autres Unix alors vous coderez en respectant les interfaces POSIX, indiquées dans les chapitres de ***man*** dont le numéro est suivi d'un p.

Bien entendu c'est une contrainte puisqu'on doit se limiter au plus petit dénominateur commun & qu'on ne peut plus utiliser les spécificités techniques de chaque plate-forme. Ou alors on fait des chemins spécifiques dans le code pour chacun des SE mais on le paye en temps de développement, facilité de relecture, nombre de bugs, etc.

01012

La commande interne **declare** sert à déclarer des variables, en général & à leur donner une valeur initiale certaine. Comme son usage est facultatif vous pouvez l'employer pour cataloguer celles que vous emploierez !

01013

Le mot réservé **time** a pour effet de mesurer le temps nécessaire à l'exécution de la commande qui suit. Il n'est pas spécifique aux tubes, par exemple :

```
time ls -FRAC /etc >toto 2>/dev/null

real    0m0.020s
user    0m0.007s
sys     0m0.012s
```

où la première valeur indique le temps écoulé entre l'appui sur ↵ & la fin de l'affichage, la seconde, le temps CPU imputable à l'utilisateur & la troisième, celui imputable au système d'exploitation. L'écart est dû aux arrondis d'affichage.

01014

Les caractères de contrôles sont : &, &&, (), {}, ;, |, ||, ↵

01015

Rappel : un signal est une information envoyé à un processus pour le terminer, l'arrêter ou le continuer, s'il a été arrêté. Ce sont des nombres compris entre *1* & *32*. Ils sont représentés symboliquement par des mots, par exemple SIGKILL pour *9*. Il existe également dans les noyaux Linux temps réel des signaux temps réel dont la valeur varie entre *33* & *64*.

01016

Attention : dans la plupart des langages dérivant du C, ces opérateurs sont les opérateurs de comparaisons *égalité* & *différence*.

01017

Attention : ces trois opérateurs sur les expressions rationnelles renvoie 0 quand ils réussissent, alors que l'égalité (=) & la différence (!=) renvoient 1. Ce n'est pas gênant dans un test, car cela est géré, mais évitez d'employer ces résultats afin de faire des calculs. Notez, également l'ambiguïté de ce dernier opérateur qui indique la différence entre chaînes en tant qu'option de la commande **test** & la non correspondance d'un motif dans une expression rationnelle.

01018

Vous le savez, dans Unix, tout est fichier & tous les fichiers sont repérés par un numéro unique, le *i-node*. Les descripteurs de fichiers sont en quelques sortes des variables que l'on peut associer à des fichiers. Trois de ces *variables* sont standardisées, celles nommées 0 (entrée standard), 1 (sortie standard) & 2 (sortie des messages d'erreur). Quand vous tapez une commande, l'entrée standard est le clavier (/proc/self/fd/0) & la sortie standard, l'écran (/proc/self/fd/1). Dans un tube, sauf pour la première & la dernière, l'entrée standard est la commande précédente & la sortie standard, la commande suivante.

Vous pouvez définir des descripteurs de fichiers supplémentaires, selon vos besoins, mais cela dépasse le cadre de cette initiation.

01019

Juste pour information : l'indirection de variables sert, dans les cas rarissimes, où vous voulez créez une variable dont le nom est constitué, par exemple, d'une combinaison comportant un ou plusieurs autres variables.

01020

Autrement dit, le premier paramètre positionnel a le numéro un, mais toutes les sous-chaînes sont indexées à partir de zéro.

Notez-le : le caractère numéro 2 est le troisième ! En partant de la fin on extrait le 2e puis le 3e !

01021

La protection est celle des caractère spéciaux, elle est obtenue en les faisant précéder d'un \. Si les expansions & les substitutions ont réussi, il ne devrait rester aucun de ces caractères.

01022

Dans les documentations en anglais ou en français, on vous explique qu'il *correspond à la chaîne vide, mais seulement en début d'une ligne dans le texte à faire correspondre*. Il nous arrive, par tradition, de conserver cette formulation, mais elle est complètement idiote : une chaîne vide est vide ; si un caractère lui est concaténé, elle n'est plus vide. Il serait plus juste de dire que ^ indique de commencer la recherche en début de chaîne &, inversement, que $ demande de la commencer par la fin.

01023

Lors des version antérieures à la 0.8, nous avions écrit qu'il n'existait pas d'outil pour la génération de nombres aléatoires en *bash*. En fait, s'il n'existe aucune commande, il existe *une variable prédéfinie* un peu particulière, dont nous avions oublié l'existence : RANDOM. Sa particularité est qu'on ne lui assigne pas de valeur, c'est *bash* qui le fait chaque fois qu'on s'y réfère & sa valeur est comprise entre 0 & 32767.

Annexe 1
Les Caractères spéciaux

Une des difficultés de Linux s'avère la complexité de ses commandes & en particulier celle de *Bash*. Celle-ci provient en grande partie de la polysémie des signes employés. Plus particulièrement ce sont certains caractères ou groupes de caractères que l'on dit spéciaux, parce que, par défaut, ils ne sont pas traités comme des textes, mais comme des commandes (elles exécutent des actions), des opérateurs (ils combinent des entités – variables, constantes, fonctions, commandes) ou des séparateurs (ils isolent des entités).

L'échappement permet de rendre littéraux ces caractères, il est réalisé pour un caractère, par l'emploi du signe « \ » & pour plusieurs par l'usage des guillemets simples « ' » (apostrophe dactylographique) & doubles « " ».

Dans les tableaux qui suivent, la première colonne indique le signe, la seconde, les différents rôles qu'il tient en fonction du contexte.

Séparateurs/opérateurs

#	Commentaires, substitution de paramètres, conversion de base, filtrage de motif.
;	Séparateur de commande. Permet de placer deux commandes ou plus sur la même ligne.
;;	Fin de ligne du traitement d'un cas dans alternative case.
.	Commande interne source de Bash. Composant d'un nom de fichier

	Filtrage d'un caractère dans les expressions rationnelles
"	Citation partielle. "CHAÎNE" empêche l'interprétation de la plupart des caractères spéciaux présents dans la CHAÎNE. Délimiteur d'expression rationnelle dans bash.
'	Citation totale. 'CHAÎNE' empêche l'interprétation de tous les caractères spéciaux présents dans la CHAÎNE. Délimiteur d'expression rationnelle dans bash.
,	Opérateur virgule. L'opérateur virgule relie une suite d'opérations arithmétiques. Toutes sont évaluées, mais seul le résultat de la dernière est renvoyé.
`	Groupe de commandes, lance un sous–shell. Employer () de préférence.
\	Échappement [anti–slash]. Le \ peut être utilisé pour écrire " & ' pour pouvoir les écrire sous forme littérale ou les mettre entre guillemets.
/	Séparateur du chemin d'un fichier [barre oblique, slash]. Sépare les composants d'un nom de fichier C'est aussi l'opérateur arithmétique de division C'est le délimiteur standard d'expression rationnelle
`	Substitution de commandes [accent grave]. La construction `commande` rend la sortie de commande disponible pour initialiser une variable. Connu sous le nom de guillemets inversés.
:	Commande nulle [deux–points]. Commande interne, équivaut à true. Sert de bouche–trou Évalue une suite de variables en utilisant la substitution de paramètres En combinaison avec l'opérateur de redirection >, tronque un fichier à la taille zéro, sans changer ses permissions. Équivalent à cat /dev/null >fichier. Séparateur de champ, dans /etc/passwd & dans la variable $PATH.
!	Inverse le sens d'un test ou d'un état de sortie. Mot–clé Bash. Références indirectes de variable. À partir de la ligne de commande, appelle le mécanisme d'historique de bash.
*	Joker pour l'expansion des noms de fichiers Utilisé seul, il correspond à tous les noms de fichiers d'un répertoire donné. Représente un caractère répété plusieurs fois (ou zéro) dans une expression régulière. Opérateur arithmétique + est une multiplication.
**	Opérateur d'exponentiation.

?	Opérateur de test. À l'intérieur de certaine expressions, il indique un test pour une condition. Peut servir d'opérateur à trois arguments Teste si une variable a été initialisée. Sert de joker pour un seul caractère pour l'expansion d'un nom de fichier dans un remplacement. Représente un caractère dans une expression régulière étendue.
$	Dans une expression régulière, un $ signifie la fin d'une ligne de texte. Sert en général combiné avec (, { ou [.
${}	Substitution de paramètres. $+, $@ Paramètres spéciaux. $? Résultat d'exécution de la dernière commande. $$ Variable contenant l'identifiant du processus. $1 à ${99} Paramètres de position
()	Groupe de commandes, lance un sous–shell. Initialisation de tableaux.
{}	Expansion d'accolades. *Bloc de code* [accolades]. Aussi connu sous le nom de *groupe en ligne*, cette construction crée une fonction anonyme. Néanmoins, contrairement à une fonction, les variables d'un bloc de code restent visibles par le reste du script. Contrairement à un groupe de commandes entre parenthèses, comme ci–dessus, un bloc de code entouré par des accolades ne sera pas lancé dans un sous–shell.
{} \;	Chemin. Principalement utilisé dans les constructions find. Ce n'est pas une commande intégrée du shell.
[ ]	Teste l'expression entre [&]. Notez que [fait partie de la commande intégrée test (& en est un synonyme), ce n'est pas un lien vers la commande externe /usr/bin/test (employée par *ksh*).
[[ ]]	Teste l'expression entre [[&]] (mot–clé du shell).
[ ]	Élément d'un tableau.
[ ]	Suite de caractères. devant servir de motif.
(( ))	Expansion d'entiers. Développe & évalue une expression entière entre ((&)).
> &>	Redirection.

>& >> <	Ce signe indique également lé décalage à droite d'un bit.
>, <	Opérateurs de comparaison de chaînes de caractères. Opérateurs de comparaison d'entiers
<<<	Redirection utilisée dans une chaîne en ligne.
<<	Redirection utilisée dans un document en ligne. Décalage à gauche d'un bit
\<, \>	Délimitation d'un mot dans une expression régulière.
|	Tube ou pipe. Passe la sortie de la commande précédente à l'entrée de la suivante. Cette méthode permet de chaîner les commandes. Opérateur *OU* bit à bit Opérateur *OU* dans les expressions rationnelles
>|	Force une redirection (même si l' option **noclobber** est mise en place). Ceci va forcer l'écrasement d'un fichier déjà existant.
||	Opérateur *OU* (*OR*) logique. Dans une structure de test , l'opérateur || a comme valeur de retour 0 (succès) si l'un des deux est vrai.
&	Faire tourner la tâche en arrière–plan. Une commande suivie par un & fonctionnera en tâche de fond. Opérateur *ET* bit à bit
&&	Opérateur *ET* (*AND*) logique. Dans une structure de test, l'opérateur && renvoie 0 (succès) si & seulement si les deux conditions sont vraies.
-	Préfixe d'option. Introduit les options pour les commandes ou les filtres. Préfixe pour les opérateurs de comparaison numérique de **test**. Redirection à partir de ou vers stdin ou stdout [tiret] : **(cd** /source/répertoire **&& tar cf** – . **)	(cd** /dest/répertoire **&& tar xpvf** –**)** *# Déplace l'ensemble des fichiers d'un répertoire vers un autre* Notez que dans ce contexte le signe – n'est pas en lui–même un opérateur *Bash*, mais plutôt une option reconnue par certains utilitaires UNIX qui écrivent dans stdout ou lisent dans stdin, tels que **tar**, **cat**, etc. Répertoire courant précédent. **cd** – revient au répertoire précédent, en utilisant la variable d'environnement *$OLDPWD*. Moins. Le signe moins est une opération arithmétique. Séparateur des bornes d'un ensemble de caractères dans les expression rationnelles

=	Égal. Opérateur d'affectation. opérateur de comparaison de chaînes de caractères.
+	Opérateur arithmétique d'addition. Opérateur d'expression régulière. Option pour une commande ou un filtre. Certaines commandes, intégrées ou non, utilisent le + pour activer certaines options & le – pour les désactiver.
%	Modulo. Opérateur arithmétique modulo (reste d'une division entière). opérateur de reconnaissance de modèles.
~	Répertoire de l'utilisateur [tilde]. Le ~ équivaut à *$HOME*.
~+	Répertoire courant. Correspond à la variable interne *$PWD*.
~-	Répertoire courant précédent. Correspond à la variable interne *$OLDPWD*.
^	Début de ligne. Dans une expression régulière, un ^ correspond au début d'une ligne de texte.

Caractères de contrôles

Ce sont ceux dont le code ASCII est inférieur à 32, ils modifient le comportement d'un terminal ou de l'affichage d'un texte & s'obtiennent par une combinaison de la touche CONTROL & d'une lettre.

Touches	Abr.	Rôle
Ctrl B	\b	Retour en arrière (backspace) non destructif.
Ctrl C	\cc	Termine un job en avant–plan.
Ctrl D	\cd	Se déconnecte du shell (similaire à un exit). C'est le caractère << EOF >> (End Of File, fin de fichier), qui termine aussi l'entrée de stdin. En saisissant du texte sur la console ou dans une fenêtre xterm, Ctl–D efface le caractère sous le curseur. Quand aucun caractère n'est présent, Ctl–D vous déconnecte de la session comme de normal.
Ctrl G	\a	CLOCHE (bip).
Ctrl H	\ch	Supprime le caractère précédent (Backspace).
Ctrl I	\t	Tabulation horizontale.
Ctrl J	\r	Retour chariot (line feed).

Touches	Abr.	Rôle
Ctrl K	\v	Tabulation verticale. En saisissant du texte sur la console ou dans une fenêtre xterm, Ctl–K efface les caractères en commençant à partir du curseur & en finissant à la fin de la ligne.
Ctrl L	\f	Formfeed (efface l'écran du terminal), a le même effet que la commande clear.
Ctrl Q	\cq	Sort du mode pause du terminal (XON). Ceci réactive le stdin du terminal après un gel.
Ctrl S	\cs	Pause du terminal (XOFF). Ceci gèle le stdin du terminal (utilisez Ctrl–Q pour en sortir).
Ctrl U	\cu	Efface une ligne de l'entrée du début de la ligne au curseur. Avec certains paramétrages, il efface la ligne d'entrée entière, quelque soit la position du curseur.
Ctrl V	\cv	Lors d'une saisie de texte, Ctl–V permet l'insertion de caractères de contrôle. Par exemple, les deux lignes suivantes sont équivalentes : echo –e '\x0a' & echo <Ctl–V><Ctl–J>
Ctrl W	\cw	Efface les caractères compris entre le courant & le prochain espace, y compris le courant. Ou efface en arrière jusqu'au premier caractère non alphanumérique.
Ctrl Z	\cz	Met en pause un job en avant–plan.

Les Espaces

Les espaces (caractère espace, tabulation, saut de ligne, saut de page) fonctionnent comme un séparateur, séparant les commandes ou les variables. Les espaces blanches sont constitués d'espaces, de tabulations, de sauts de lignes ou d'une combinaison de ceux–ci. Dans certains contextes, tels que les affectations de variables, les espaces blanches ne sont pas permises, & sont considérées comme une erreur de syntaxe.

Les saut de lignes ou lignes blanches n'ont aucun effet sur l'action d'un script : ils servent à en séparer visuellement les différentes parties.

La variable IFS est une variable spéciale définissant pour certaines commandes le séparateur des champs en entrée. Elle a pour valeur par défaut *espace blanche*, *tabulation* & *passage à la ligne*.

L'Échappement

L'échappement se fait au moyen du caractère \, certains emplois figurent dans le tableau précédent. Il en est quelques autres.

À la fin d'une ligne, un anti-slash indique que la commande continue à la ligne suivante. Cette fonction est particulièrement utile pour les grandes commandes afin de les rendre plus facilement lisibles.

Autre cas, les méta-caractères, notamment * (astérisque), ne sont pas interprétés par *Bash* en tant que littéraux, ce qui est gênant dans certains cas. Les commandes **find**, **sed** exemplifient ce point délicat. Si à l'aide de la commande **find** on souhaitait chercher dans le répertoire courant, représenté par ., & ses sous-répertoires, tous les fichiers dont le nom commence par my, on serait tenté d'écrire la ligne de commande suivante **find** . **-name** my*. Mais la commande renverra Find: Les chemins doivent précéder l'expression.

En effet, *Bash* va substituer à la chaîne my* la liste des fichiers contenus dans le répertoire courant, ce que **find** considère comme étant une liste de chemins, qui doivent être spécifiés en premier lieu, & non comme le nom des fichiers à rechercher.

Une des solutions consiste à utiliser un anti-slash avant le caractère * pour l'*échapper* & forcer *Bash* à l'interpréter comme un littéral. Ce qui donne **find** . **-name** my*.

Une autre solution serait d'utiliser les guillemets. On pourrait par exemple écrire **find** . **-name** "my*"

Enfin, extension des informations du tableau précédent, il est possible d'échapper d'autres caractères, comme le montre le tableau suivant.

Échappement par antislash Transformation par Bash

\e	Échappement caractère de code ascii 27 correspondant à la touche [Échap]
\\	Anti-slash, barre de fraction inverse
\'	Une apostrophe (le nom anglais de ce caractère est *quote*)
\nnn	Le caractère 8 bits dont la valeur en octal est *nnn*
\xHH	Le caractère 8 bits dont la valeur en hexadécimal est *HH*
\cx	Le caractère Ctrl X

Ce texte est une simplification & une mise en forme du chapitre 3 du Guide avancé d'écriture des scripts Bash de Guillaume Evain (http://www.evain.info/script/getFile.php?idf=85), traduction de Advanced Bash-Scripting Guide de Mendel Cooper (http://tldp.org/LDP/abs/html/).

Annexe 2
Quelques commandes & fichiers utiles

Ce ne sont ni toutes les commandes ni tous les fichiers de configurations. Ces listes représentent une partie des commandes. & des fichiers de configuration. Elles donnent une idée de la richesse fonctionnelle d'un système Linux & elles montrent la nécessité d'employer les commandes de recherche d'information : man, whatis, apropos ,info, etc. Les commandes graphiques & de programmation ont été systématiquement éliminées de ces listes, entre autres.

Elles ont été obtenue sur une distribution LinuxMint14 par la commande : LANG=fr_FR whatis -s n -r . | sort >/home/.../Scripts/com_ext. Avec n=1 pour obtenir la liste des commandes du premier chapitre du manuel Commandes utilisateur, n=8, pour les Commandes administrateurs, n=5, pour les Fichiers de configuration.

La variable LANG explique la présence de phrases en français pour les commandes les plus usitées. Nous avons traduites celles des commandes moins usitées.

Les Commandes externes pour tous

Nom	Fonction
7z	Créatoin d'archives compressées
abs2rel	Conversion d'un chemin absolu en relatif
apropos	Chercher le nom & la description des pages de manuel
at	Exécution différée de commandes
awk	Analyse de texte organisée en colonnes.
basename	Suppression du dossier & de l'extension d'un nom de fichier.
bash	GNU Bourne Again SHell

NOM	FONCTION
bluefish	Editeur pour programmeur expérimenté & concepteur web
bunzip2	Décompression fichier bzip2
bzip2	Compression de ficher d'archive
cabextract	Extraction de fichiers d'archives .cab
cancel	Abandon d'un job
cat	Concaténation de fichiers & affichage à l'écran
chacl	Changement de l'acl d'un fichier ou d'un dossier
chattr	Changement des attributs de fichiers
check-regexp	Test d'expressions rationnelles
chgrp	Changement du groupe
chmod	Changement des attributs (rwx) d'un fichier
chown	Changement du propriétaire
chroot	Changement du dossier racine
clear	Efface l'écran du terminal
cp	Copie de fichiers & de dossiers
crontab	Modification du fichier crontab d'un utilisateur
cut	Extrait des portion de chaque ligne d'un fichier
date	Affiche ou modifie l'heure & la date du système
df	Affiche l'occupation des partitions
dig	Utilitaire examen DNS
dmesg	Affichage les messages kernel
dnsdomainname	Affiche le nom de domaine du système
domainname	Affiche ou définit le nom d'hôte du système
dos2unix	Convertit les fichiers textes du format DOS/Mac vers Unix

Nom	Fonction
	et inversement
du	Estimation de l'espace disque employé par un dossier
egrep	Grep avec expressions rationnelles étendues
env	Exécute un programme dans un environnement modifié
file	Détermine le type d'un fichier
free	Affcihe le montant de mémoire disponible & utilisée
fuser	Identification des processus employant un fichier
getfacl	Fournit les ACL d'un fichier
gpg	Outil *OpenPGP* d'encodage & de signature
groups	Affiche les groupes auxquel appartient un utilisateur
gunzip	Décompression d'archives
gzip	Compression d'archives
hexdump	Affiche un fichier en hexadécimal, décimal, octal, ou ASCII
host	Outil DNS
hostname	Affiche ou définit le nom d'hôte du système
info	Lit les aides info
init	Gestionnaire de service de *systemd*
intro	Introduction aux commandes utilisateur
ipptool	Utilitaire de requêtes IP
journalctl	Interrogation du journal de *systemd*
kill	Termine un processus
less	Meilleur outil d'affichage du contenu d'un fichier texte
locate	Recherche de fichiers par leur nom
login	Ouverture de session

NOM	FONCTION
lp	Impression de fichiers
ls	Liste le contenu d'un dossier
machinectl	Contrôle de gestionnaire de machine de *systemd*
mail	Émission & réception de courriels
man	Interface de consultation du manuels Linux
md5sum	Calcule les somme de contrôle *md5*
mkdir	Crée un dossier
more	Affiche un fichier texte par page-écran
mv	Déplace ou renomme un ou des fichiers
nano	NAno un NOuvel éditeur, un clone libre & amélioré de *Pico*
passwd	Modifie le mot de passe d'un utilisateur
ps	Affiche un instantané des processus en cours
pstree	Affiche l'arborescence des processus
pwd	Affiche le nom du dossier de travail courant
rlogin	Ouverture de session distante
rm	Suppression de fichiers ou de dossiers
rmdir	Suppression de dossiers vides
rsh	Shell distant
rsync	Outil rapide de copie de fichiers distants ou locaux
scp	Copie sécurisée de fichiers distzants
sed	Éditeur de flux permettant de modifier des fichiers textes
seq	Affiche une séquence de nombres
setfacl	Modification des attributs acl d'un fichier
shasum	Calcul la somme de contrôle *sha*

Nom	Fonction
sleep	Attente durant le temps spécifié
sort	Tri de lignes de texte
spamassassin	Filtre de courriel pour éliminer les spams
split	Éclate un fichier en plusieurs
stat	Affiche le statut d'un fichier ou d'une partition
su	Exécute une comande avec subtituion d'utilisateur & de groupe
syslinux	Installe le chargeur de démarrage sur une partition FAT
systemctl	Administration de *systemd*
systemd	Gestionnaire de services entre autres choses
systemd-journalctl	Journaux système de *systemd*
tail	Affihce la fin d'un fichier
tar	Utilitaire d'archivage
tee	Lecture de données de l'entrée standard afin de les écrire sur la sortie standard ou sur un fichier
test	Commande test externe pour ksh
tidy	Valide, corrige, & imprime proprement les fichiers html
top	Affiche les processus en cours (gestionnaire de tâche en ligne de commande)
tree	Affiche récursivement le contenu d'un dossier sous forme d'arborescence
true	Ne fait rien correctement
uname	Affiche des informations sur le système
unix2mac	Convertit les fichiers textes du format dos/mac vers unix & inversement

NOM	FONCTION
unrar	*Extraction, test & affichage d'archives rar*
unzip	Extraction, test & affichage d'archives zip
users	Liste des utilisateur connecté à l'hôte courant
uuidgen	Créé un nouvel UUID pour une partition
vdir	Équivalent à *ls -l*
vim	*Vi IMproved*, éditeur de texte pour programmeurs
vimtutor	Tutoriel *vim*
w	Affiche les utilisateurs connectés & ce qu'ils font
wall	Envoi u message à tous les utilisateurs
wc	Affiche les nombres de lignes, de mots & d'octets pour chaque fichier
wget	Téléchargeur réseau en ligne de commande
whatis	Afficher une ligne de description des pages de manuel
whereis	Affiche les exécutables, les sources & les pages de manuel d'une commande
which	Affiche le chermin d'accès complet d'une commande
who	Indique qui est connecté
whoami	Affiche le nom d'utilisateur réel
write	Envoi d'un message à un autre utilisateur
xargs	Permet d'employer dans un tube des commande qui ne sont pas prévus pour y fonctionner
zip	Archive & compresse des fichiers
zsh	L'interpréteur de commande *zsh*

Les Commandes pour l'administration

Nom	Fonction
acpid	Service de gestion de l'alimentation
addgroup	Ajout d'un groupe (debian & consorts)
adduser	Ajout d'un utilisateur (debian & consorts)
anacron	Exécution périodique d ecommande
arp	Gestion du cache arp
arpd	Service arp
atd	Service d'exécutions ultérieure de jobs
badblocks	Recherche des blocs en erreur d'un dispositif
btrfs	Gestion du SGF btrfs
chroot	Déplace la racine de l'arborescence
cron	Service d'exécution de commandes programées
delgroup	Suppression d'un groupe (debian & consorts)
deluser	Suppression d'un utilisateur (debian & consorts)
dhclient	Client DHCP
dkms	Gestion dynamiques des modules du noyau
e4defrag	Défragmenteur pour le SGF ext4
findmnt	Affiche l'arborescence des SGF
fsck	Teste & répare un SGF
gparted	Gestionnaire de partitions
groupadd	Crée un nouveau groupe (toutes distributions)
groupdel	Efface un groupe (toutes distributions)
groupmod	Modifie un groupe (toutes distributions)
Halt	Ces trois commandes peuvent arrêter, ou redémarrer la

NOM	FONCTION
poweroff reboot	machine selon l'option présente ou absente
hdparm	Gestionnaire de disques durs
ifconfig	Configure une interface réseau ou les affiche toutes
ifdown	Désactive une interface réseau
ifup	Active une interface réseau
init	Service de démarrage d'Upstart (Ubuntu)
insmod	Insère un module dans le noyau Linux
ip	Affiche ou manipule les routes, les adresses, etc.
iptables	Manipulation des règles de *NetFilter*, le firewall de Linux
iw	Idem ip pour les réseaux sans fil
iwconfig	Idem if config pour les réseaux sans fil
logrotate	Rotation & compressions des journaux *rsyslog*
lpadmin	Configuration des imprimntes & des classes de *cups*
lsblk	Liste les périphéques en mode bloc (disques divers)
lsmod	Affiche les modules du noyau
lsof	Liste les fichiers ouverts
lspci	Affiche les périphériques PCI
lsusb	Affiche les périphériques USB
mkfs	Formate une partition
modprobe	Ajoute ou enlève des modules du noyau
mount	Monte un SGF
net	Administration de samba & des serveurs cifs distants
netstat	Affiche toutes les information relatives au réseau

Nom	Fonction
NetworkManager	Service d'administration du réseau
nmbd	Serveur de nom Netbios au dessus d'IP
nologin	Refuse poliment une connexion
nstat	Outil de statistiques réseau
ntpdate	Modifiaction de la date & de l'heure par *ntp*
pam_...	Utilitaires *pam*
pidof	Affiche le numéro de processus d'un programme exécuté
ping	Envoi d'une requête ICMP à un hôte du réseau
rarp	Gestion de la table rarp
route	Affiche ou gère la table de routage IP
rsyslogd	Gestionnaire de journaux locaux ou distants
runlevel	Affiche le niveau de démarrage
service	Exécute un script de gestion de service
shutdown	Arrêt du système paramétrable
smbd	Serveur smb/cifs
smbpasswd	Gestion des mots de passe des utilisateurs samba
sudo	Exécute une commande avec un utilisateur différent
sudo_root	Exécte les comandes d'administration
sysctl	Configure les paramètres du noyau pendant l'exécution
tcpdump	Vidage dans un fichier du trafic du réseau
traceroute	Affiche le chemin permettant d'atteindre un hôte du réseau
udevadm	Gestionnaire de *udev*
umount	Demontage d'un SGF
updatedb	Mise-à-jour de la base de données pour *locate*

NOM	FONCTION
User[add|del|mod)	Utilitaires unix standards de gestion des utilisateurs
visudo	Éditeur du fichier sudoers

LES FICHIERS DE CONFIGURATION

NOM	FONCTION
ldap.conf	Fichier de configuration d'*OpenLDAP*
anacrontab	Fichier de configuration d'*anacron*
modules	Modules dunoyau à chager lors du démrrage
interfaces	Fichier de configuration pour *ifup* & *ifdown*
acl	Liste de contrôle d'accès
charmap	Table d'encodage des caractères
core	Fichier de vidage mémoire générés lors d'un plantage
crontab	Liste des tâches programmées
dhclient.conf	Fichier de configuration du client DHCP
dhclient.leases	Base de données des baux des clients DHCP
dir_colors	Fichier de configuration de dircolors
filesystems	Liste des SGF suppportés par Linux dont : ext3, ext4, reiserfs, xfs, jfs, msdos, vfat, ntfs, proc, nfs, iso9660, hpfs, sysv, smb
fonts-conf	Fichier de configuration des polices de caractères
fstab	Liste des SGF à monter au démarrage
group	Fichier des groupes d'utilisateurs
host.conf	Fichier de configuration de *Resolver*
hosts	Liste statique des noms d'hôte
init	Script de démarrage *upstart*

Nom	Fonction
inittab	Fichier de configuration du démon Init
interface-order	Fichier de configuration de *Resolvconf*
keymaps	Carte des caractères accessibles au clavier
locales.conf ou locale	Définition des spécificités de langues régionales (fr_Fr, etc.)
mlocate.db	Base de données pour *locate*
NetworkManager.-conf	Fichier de configuration de *Networkmanager*
networks	Liste des réseaux accessibles
pam.conf	Fichier de configuration de *pam*
pam.d	Dossier de configuration de *pam*
passwd	Définition des utilisateurs
proc	Pseudo-SGF d'information sur les processus
protocols	Définition des protocoles
resolv.conf	Fichier de configuration de *resolver*
resolver	Fichier de configuration de *resolver*
rsyslog.conf	Fichier de configuration de *rsyslogd*
securetty	Liste des terminaux accessibles à root
services	Liste des services réseau
shadow	Fichier protégé des mots de passe des utilisateurs
shells	Chemins d'accès aux shells valides
smb.conf	Fichier de configuration de *samba*
smbpasswd	Mots de passe codés de *samba*
snmp.conf	Fichier de configuration des applications *net-snmp*
ssh_config	Fichier de configuration du client *Openssh*

NOM	FONCTION
sudoers	Définition des politiques de sécurité de
sysctl.conf	Fichier de configuration de *sysctl*

LES COMMANDES INTERNES DE BASH

Dans ce tableau, commande désigne aussi bien une commande interne une commande externe ou une commande composée, arg un ou plusieurs arguments, c'est-à-dire des informations complémentaires que vous donnez, option une liste d'option ou une option, expression désigne soit une valeur numérique ou alphanumérique, soit une expression numérique, logique ou rationnelle.

Ces crochets [] indiquent un élément facultatif, ceux-ci [] un opérateur.

COMMANDES
AIDE
help [-dms] [modèle ...]
history [-c] [-d *offset*] [*n*] history -anrw [nomfichier] history -ps arg [arg ...]
STRUCTURATION
[[expression]]
[[expression]] # utilise l'ordre lexicographique é avant f & non ASCII comme test (é après z)
{ commande; }
break [*n*]
case mot in [modèle [| modèle]...) commande ;;]... esac

COMMANDES
continue [*n*]
declare [-aAfFgilrtux] [-p] [nom[=valeur] ...]
exit [*n*]
for ((exp1; exp2; exp3)); do commande; done
for nom [in mots ...] ; do commande; done
function nom { commande ; } () { commande ; }
if commande; then commande; [elif commande; then commande;]... [else commande;] fi
let arg [arg ...] #
local [option] nom[=valeur] ...
return [*n*]
select nom [in mots ... ;] do commande; done
test [expression] [expression]
time [-p] pipeline
times commande [option] [arg]
trap [-lp] [[arg] specsignal ...]
true
until commande; do commande; done
Variables # Nom & sens des variables de bash
while commande; do commande; done

COMMANDES
AUTRES
: # ne fait rien
alias [-p] [nom[=valeur] ...]
bg [job_spec ...]
builtin [comm_interne [arg ...]]
cd [-L|[-P [-e]]] [dossier]
command [-pVv] commande [arg ...]
commande [&]
dirs [-clpv] [+N] [-N]
echo [-neE] [arg ...]
enable [-a] [-dnps] [-f nomfichier] [nom ...]
eval [arg ...]
exec [-cl] [-a nom] [commande [arguments ...]] [redirection ...]
export [-fn] [nom[=valeur] ...] export -p
false
fg [spectâche]
hash [-lr] [-p chemin] [-dt] [nom ...]
jobs [-lnprs] [tâche ...] # tâche est le numéro de la tâche jobs -x commande [arguments]
kill [-s nomsignal | -n *numsignal* | -specsignal] *numprocessus* | *numtâche* ... kill -l [specsignal]

COMMANDES
logout [*n*]
popd [-n] [+N | -N]
printf [-v var] format [arguments]
pushd [-n] [+N | -N | dir]
pwd [-LP]
read [-ers] [-a tableau] [-d délimiteur] [-i texte] [-n *nchars*] [-N *nchars*] [-p prompt] [-t *timeout*] [-u fd] [nom ...]
readarray [-n compte] [-O origine] [-s compte] [-t] [-u fd] [-C callback] [-c quantum] [tableau]
set [-abefhkmnptuvxBCHP] [-o option-nom] [--] [arg ...]
shift [*n*]
. nomfichier [arg] source nomfichier [arg]
suspend [-f]
type [-afptP] nom [nom ...]
typeset [-aAfFgilrtux] [-p] nom[=valeur] ...
ulimit [-SHacdefilmnpqrstuvx] [*limite*]
umask [-p] [-S] [*mode*]
unalias [-a] nom [nom ...]
unset [-f] [-v] [nom ...]
wait [*id*]

Annexe 3
Les Expressions rationnelles

Théorie

La définition formelle insiste sur l'aspect complexe d'une expression rationnelles. Elle la divise en morceaux appelés branches qui ne peuvent pas êtres vides & qui sont séparés par |. Chaque branche est elle-même divisée en morceaux appelés pièces & chaque pièce, en morceau appelé atomes, c'est-à-dire en un élément indécomposable. Un atome peut être suivi d'un encadrement qui indique combien de fois il faut le multiplier.

- Un *atome* est :
  - un ensemble vide () (correspond à une chaîne nulle),
  - une expression entre crochets (voir § suivant),
  - un point . (correspondant à n'importe quel caractère),
  - un signe ^ (chaîne vide en début de ligne),
  - un signe $ (chaîne vide en fin de ligne),
  - un \ suivi d'un des caractères dits spéciaux (^, ., [, $, (,), |, *, +, ?, {, & \ ; ces caractères sont en fait des opérateurs, le \ étant le caractère d'échappement qui les fait correspondre à leur valeur littérale sans signification particulière),
  - un \ suivi de n'importe quel autre caractère (correspondant au caractère pris sous forme littérale, comme si le \ était absent),
  - un caractère ordinaire sans signification particulière (correspondant à ce caractère),
  - une { suivie d'un caractère autre qu'un chiffre est considérée sous sa forme littérale, elle constitue un atome pas un encadrement !
- Un *encadrement* est une { suivie d'un entier décimal non signé, suivi éventuellement d'une virgule, suivie éventuellement d'un

entier décimal non signé, toujours suivi d'une }. Les entiers doivent être compris entre 0 & RE_DUP_MAX (255), & s'il y en a deux, le second doit être supérieur ou égal au premier.

* Une *pièce* est un atome suivi éventuellement d'un unique caractère *, +, ?, ou d'un encadrement.
* Une *branche* est une ou plusieurs pièces concaténées. Elle correspond à ce qui correspond à la première pièce, suivi de ce qui correspond à la seconde & ainsi de suite.
* Une *expression entre crochets* est une liste d'atomes ordinaires encadrés par [&]. Elle correspond normalement à n'importe quel caractère de la liste.
  - Si la liste débute par ^, elle correspond à n'importe quel caractère sauf ceux de la liste.
  - Si deux caractères de la liste sont séparés par un -, ils représentent tout l'intervalle de caractères entre-eux (eux compris). Par exemple, [0-9] représente n'importe quel chiffre décimal. Il est *illégal* d'utiliser la même limite dans deux intervalles, comme « a-i-u ».

Exemple 48 :

```
$ ls M[a-u]*
Modèles:
sites

Musique:
$ ls M[a-i-u]*
$
```

Les intervalles dépendent beaucoup de l'ordre de classement des caractères & les programmes portables devraient éviter de les utiliser.

  - Une *fusion* est une séquence de caractères qui se comporte comme un seul, encadrée par [. & .] correspond à la séquence des caractères de la fusion. Une séquence est un élément unique de l'expression entre crochets. Ainsi, un expression entre cro-

chets contenant une fusion multicaractères peut correspondre à plus d'un caractère. Par exemple, si la séquence inclut la fusion ch, alors l'ER [[.ch.]]*c correspond aux cinq premiers caractères de chchcc.

- Une *classe d'équivalence* est une séquence encadrée par [= & =], elle correspond à la séquence de caractères de tous les éléments équivalents à celui-ci (exemple [=e=] contient {e,é,è,ê,ë}, c'est-à-dire, tous les caractères qui seront classés alphabétiquement comme e), y compris lui-même. (S'il n'y a pas d'autres éléments équivalents, le fonctionnement est le même que si l'encadrement était [. & .]). Par exemple, si o & ô sont membres d'une classe équivalence, alors [[=o=]], [[=ô=]], & [o] sont tous synonymes. Une classe d'équivalence ne doit pas être une borne d'intervalle.
- Une *classe de caractères*, dans une expression entre crochets, est encadrée par [: & :]. Elle correspond à la liste de tous les caractères de la classe. Les classes standards sont :

CLASSE	DESCRIPTION
[[:alpha:]]	n'importe quelle lettre
[[:digit:]]	n'importe quel chiffre
[[:xdigit:]]	caractères hexadécimaux
[[:alnum:]]	n'importe quelle lettre ou chiffre
[[:space:]]	n'importe quelle espace
[[:punct:]]	n'importe quel signe de ponctuation
[[:lower:]]	n'importe quelle lettre bas de casse
[[:upper:]]	n'importe quelle lettre haut de casse
[[:blank:]]	espace ou tabulation
[[:graph:]]	caractères affichables & imprimables
[[:cntrl:]]	caractères dits de contrôle (code ASCII<32)

[[:print:]]	caractères imprimables exceptés ceux de contrôle

- Une classe de caractères ne doit pas être utilisée comme borne d'intervalle.

* Deux opérations peuvent être réalisées avec une expression rationnelle : la correspondance & la substitution. *Bash* & *grep* ne permettent que la correspondance, alors que *sed* & *awk* permettent également la substitution.
* On vient de voir que certains caractères ont un impact sur la correspondance. La liste de ces caractères spéciaux est : $, ^, ., *, +, ?, [,] & \. Un caractère spécial doit être précédé du caractère \ pour être considéré comme un caractère ordinaire.

De façon générale une expression rationnelle est entourée par le signe /. (Il est possible de choisir un autre délimiteur, par exemple #, si dans une expression plusieurs / apparaissent & aucun #, cela évite de les échapper.) Précédée de m (m/[ab]*/), il s'agit d'une correspondance ; précédée de s, d'une substitution (s/qwerty/azerty/). Comme la correspondance est beaucoup plus fréquente que la substitution le m peut- être omis. En *bash*, on n'utilise pas le m puisque seule la correspondance est autorisée & on remplace le / par le " ("[ab]*") ou le ' ('[ab]*').

PRATIQUE

Comme il est inutile de réinventer la roue, il n'est pas utile de repartir de zéro pour un écrit comme celui-ci.

Si certains des exemples & des exercices sont de notre cru, d'autres proviennent de pages web qui ont été sauvegardées au format texte simple, il y a quelques années, sans informations sur le site d'origine. Les sites www.linux-france.org, openclassrooms.com, www.commentcamarche.com, perldoc.perl.org ont été, également, mis à contribution. Dans tous les cas, nous avons modifiés les énoncés de façon à les adapter à notre propos. Si vous reconnaissez des sites non

mentionnés, nous vous prions de nous en informer afin que nous puissions réparer cet oubli.

SYNTAXE DES EXPRESSIONS RATIONNELLES

AVERTISSEMENT

Une expression rationnelle peut être une simple chaîne de caractère : toto.

Le texte qui suit reprend en les développant les information de la section précédente. Il intègre des opérateurs disponibles uniquement avec certains logiciels & des deux versions d'expressions rationnelles. Il est donc chaudement recommandés de lire la documentation des différents logiciels avec lesquels vous souhaitez employer des expressions rationnelles, ou, à défaut, si vous êtes patient de tester chaque opérateur.

.

Ce caractère spécial correspond à tous les caractères excepté celui de passage à la ligne (*newline*, noté aussi /n en ***bash***). En utilisant la concaténation, vous pouvez créer des expressions rationnelles comme a.b, qui correspond à toutes chaînes de trois caractères commençant par a & finissant par b.

*

C'est un opérateur suffixe qui indique de répéter la correspondre précédente autant de fois que possible. Ainsi, o* correspond à un quelconque nombre de o (éventuellement aucun).

Il s'applique toujours à l'expression précédente la plus petite possible. Ainsi, dans fo* seul le o est répétitif, elle correspond à f, fo, foo, etc.

La recherche de correspondance d'une construction * fait correspondre, immédiatement, autant de répétitions pouvant être trouvées. Elle continue alors avec la suite du motif. Si la suite échoue, un retour en arrière est utilisé, supprimant certaines des correspondances de la

construction * modifiée, au cas où il serait possible de faire correspondre la suite du motif.

Par exemple, en faisant correspondre ca*ar sur la chaîne caaar, le a* essaie d'abord de correspondre aux aaa ; mais la suite du motif est ar & il ne reste plus que r à faire correspondre ; cet essai échoue alors. La prochaine alternative est que a* corresponde à aa exactement. Avec ce choix, l'exp-rat correspond parfaitement.

+

C'est un opérateur suffixe, similaire à * excepté qu'il doit faire correspondre l'expression précédente au moins une fois. Ainsi, par exemple, ca+r correspond aux chaînes car & caaaar mais non à la chaîne cr, alors que ca*r correspond à ces trois chaînes.

?

C'est un opérateur suffixe, similaire à * excepté qu'il fait correspondre l'expression précédente soit une fois, soit pas du tout. Par exemple, ca?r correspond à car ou cr & à rien d'autre.

*?, +?, ??

Ce sont les variantes dites non gourmandes des opérateurs précédents. Les opérateurs *, + & ? sont gourmands car ils cherchent la correspondance la plus longue possible, dans la chaîne cible. Avec un ? à la suite, ils cherchent la correspondance la plus courte possible.

Exemple 49 :

Texte	Abbbbzzzz abbsdsqdaabbbbssq
ab*	zzzz sdsqdssq
ab*?	bbbbzzzz bbsdsqdbbbbssq

\{n\} ou {n}

C'est un opérateur suffixe qui spécifie une répétition n fois de l'expression rationnelle précédente.

Exemple 50 :

x{4} correspond à la chaîne xxxx & rien d'autre.

\{n,m\} ou {n,m}

C'est un opérateur suffixe qui spécifie une répétition entre n & m fois--ainsi, l'expression rationnelle précédente doit correspondre au moins n fois, mais pas plus de m fois. Si m est omis, il n'y a alors pas de limite supérieure, mais l'expression rationnelle précédente doit correspondre au moins n fois.

{0,1} est équivalent à **?**.

{0,} est équivalent à *****.

{1,} est équivalent à **+**.

Exemple 51 :

x{2,4}	xx, xxx & xxx.
x{2,}	xx, xxx, xxxx, xxxx, etc.

[...]

Les caractères entre les crochets définissent un ensemble de caractères qui commence après [& se termine avant]. Dans sa forme la plus simple, les caractères entre les deux crochets sont ce à quoi ce jeu peut correspondre.

Exemple 52 :

[ad]	(), a, d, ad, da
c[ad]*r	cr, car, cdr, caddaar, etc.

Attention [a-z] correspond à n'importe quelle lettre ASCII minuscule& non pas à un nombre quelconque de lettres minuscules.

Des jeux de caractères peuvent être mélangés librement avec des caractères individuels, comme dans [a-z$%.], qui correspond à n'importe quelle lettre ASCII minuscule ou $, % un point.

Notez que les caractères habituellement spéciaux pour une exprat ne le sont pas dans un jeu de caractères. Un jeu complètement différent de caractères spéciaux existe dans les jeux de caractères :], -, & ^.

Pour inclure un] dans un jeu de caractères, il doit être le premier des caractères. Par exemple, []a] correspond à] ou a. Pour inclure -, placez - comme premier ou dernier caractère du jeu, ou placez-le après un autre caractère spécial. Ainsi, []-] correspond à] & -.

Pour inclure ^ dans un jeu, placez-le n'importe où sauf au début du jeu.

[^ ...]

Cet opérateur définit un jeu de caractères complémentaire, comprenant tous les caractères absent dans les crochets.

Ainsi, [^a-z0-9A-Z] correspond à tous les caractères sauf aux lettres & aux chiffres.

^ n'est pas un caractère spécial dans un jeu de caractères à moins qu'il ne soit le premier caractère. Le caractère suivant le ^ est traité comme étant premier (en d'autres mots, - &] ne sont pas spéciaux s'ils suivent ^).

^

C'est un caractère spécial qui correspond à la chaîne vide, mais seulement en début d'une ligne dans le texte à faire correspondre . Autrement, il ne correspond à rien. Ainsi, ^foo correspond à foo

placé en début de ligne. Il serait plus juste de dire qu'il indique de commencer la recherche en début de ligne.

$

Ce caractère à la même fonction que ^, mais avec la fin de ligne. Ainsi, oof$ correspond à une chaîne oof en fin de ligne.

\

Ce caractère a deux fonctions :

- il cite les caractères spéciaux (\ inclus) ;
- il introduit des constructions spéciales supplémentaires.

Puisque \ cite les caractères spéciaux, \$ est une expression rationnelle qui correspond seulement à $, & \[est une expression rationnelle qui correspond seulement à [, & ainsi de suite.

*Note : pour une compatibilité historique, les caractères spéciaux sont traités comme ordinaires s'ils sont dans un contexte où leur "caractère spécial" n'a pas de sens. Par exemple, *foo traite * comme ordinaire car il n'y a pas d'expression précédente sur laquelle * puisse agir. C'est une pratique pauvre que de dépendre de ce comportement ; il est préférable de toujours citer les caractères spéciaux, indépendamment de l'endroit où ils apparaissent.*

Pour la plus grande part, \ suivi d'un caractère correspond seulement à ce caractère. Il y a toutefois quelques exceptions. Le second caractère de la séquence est toujours un caractère ordinaire lorsqu'il est utilisé seul.

|

C'est l'opérateur OU. Deux expressions rationnelles a & b séparées par | forment une expression qui correspond à un texte si a correspond à ce texte ou si b y correspond. Il fonctionne en essayant

de faire correspondre a, & en cas d'échec, essaie de faire correspondre b.

Ainsi, foo|bar correspond soit à foo soit à bar, & rien d'autre.

| s'applique aux expressions qui l'entoure les plus longues possibles. Seul un regroupement (...) peut limiter le pouvoir de regroupement de |.

(...)

Ces parenthèses créent une sous-expression. Ces sous expressions sont numérotées sur la parenthèse ouvrante à partir de la gauche. En clair, la première parenthèse ouvrante correspond à la première sous-expression & ainsi de suite.

Après la fin la parenthèse fermante d'une sous-expression, la recherche de correspondance stocke les positions de début & de fin du texte lui correspondant. Alors, plus loin dans l'expression rationnelle, vous pouvez utiliser \ suivi du chiffre n pour dire répéter le texte correspondant à la énième construction sous-expression. C'est ce qu'on appelle une référence arrière !

Exemple 53 :

(/d{2})(.\1){4} pour tester un numéro de téléphone (2 chiffres, puis 4 fois 1 point suivi de 2 chiffres).

Raccourcis

On dit aussi classes abrégées. Bien qu'ils soient normalisés, ils ne facilitent pas la lecture des exp-rat : \d est moins explicite que [0-9]. Ce sont les suivants.

Raccourci	Signification
\d	Indique un chiffre. Équivalent à [0-9]
\D	Indique ce qui n'est pas un chiffre. Équivalent [^0-9]

RACCOURCI	SIGNIFICATION
\w	Indique un caractère alphanumérique ou un tiret de soulignement. Équivalent [a-zA-Z0-9_]. À condition de ne pas employer de caractères diacritiques !
\W	Indique ce qui n'est pas un mot. Équivalent [^a-zA-Z0-9_]. À condition de ne pas employer de caractères diacritiques !
\t	Indique une tabulation
\n	Indique une nouvelle ligne
\r	Indique un retour chariot
\s	Indique une espace blanche (\t, ,\n, \r)
\S	Indique ce qui n'est pas une espace blanche
.	Indique n'importe quel caractère. Il autorise donc tout !

DIVERS

Les éditeurs (*emacs*, *vim*) & les traitements de texte comme le *writer* des suites LibreOffice & OpenOffice.org intègrent une gestion très sophistiquées des expressions rationnelles.

Quelques exemples, regroupés par logiciel, avant les exercices.

QUELQUES EXEMPLES

LS

Exemple 54 : Liste de tous les fichiers commençant par m & o

```
ls [m,o]*
```

GREP

Exemple 55 : Liste des UID desutilisateurs humains

```
grep ":10[0-9][0-9]:" </etc/passwd
```

SED

Exemple 56 : Récupération du nom d'un script ayant l'extension .start

```
sed -ne "s/\(.*\).start/\1/p"
```

AWK

Exemple 57 : Affichage du type de SGF de la partition /

```
awk '$2 == "/" && $1 != "rootfs" {print $3}' /proc/mounts
```

Exemple 58 : Liste des partitions NTFS avec Opensuse 13.2

```
awk '$3 ~ /^fuse*/ {print $2}' /proc/mounts | grep var
```

$1, $2 & $3 indiquent les colonnes du fichier concerné, le séparateur de colonne par défaut est un des caractères de *IFS* ([ ], [⇥])

VIM

Deux exemples fonctionnant également, en mode graphique dans *geany* ou *writer*, à l'écriture du caractère espace près (au lieu de \s)

Exemple 59 : Suppression des lignes vides

```
:g/^$/ d
```

Exemple 60 : suppression des espaces à la fin d'une ligne

```
s:\s*$::
```

Annexe 4 Corrigés des exercices scripts Bash

Avertissement

Pour réussir ces scripts, il faut d'abord définir ce que l'on veut faire, puis chercher avec les commandes apropos & man les commandes externes ou internes, & leurs options, nécessaires à l'obtention du résultat.

Les scripts simples sont des entraînements : ils introduisent toutes les notions nécessaires pour réaliser les scripts complexes. Ceux-ci, à l'exception de celui de bataille navale, ont été réalisés par un de nos brillants stagiaires Laurent Villette. Les corrigés proposés qu'ils soient de Laurent Villette, de Dominique Billard ou de nous-mêmes ne sont ni les plus rapides ni les plus courts, cependant ils sont dignes d'un utilisateur unix expérimenté & ils fonctionnent ! Pour les nôtres, nous avons essayé de leur garder un aspect pédagogique. C'est à vous de dire si cet objectif a été atteint.

Dans les scripts de Laurent, seuls les noms des variables ont été changés afin de les rendre plus clairs, car il emploie des noms anglicisés, alors que nous préférons utiliser des noms français afin de distinguer les éléments faisant partie du langage ou de la distro, de nos ajouts.

Laurent les a écrits, dans les deux jours impartis pour le TP. Les commentaires critiques ou explicatifs & les corrigés alternatifs, que nous y avons ajouté, sont le fruit de plusieurs jours de travail répartis sur plusieurs mois. Ses commentaires sont présentés dans cette écriture :

```
# Laurent Villette,
les nôtres sont dans celle-ci
# Michel Scifo
```

Nos corrigés alternatifs ne sont pas nécessairement, plus performants ou plus courts, leur rédaction visant à monter le lien avec la réflexion précédant leur rédaction.

De même que pour les énoncés, les corrigés sont en deux groupes : simples & complexes.

CORRIGÉS DES EXERCICES SIMPLES

EX. 0

CORRIGÉ

```
1  #!/bin/bash
2  # expressions.sh - Michel Scifo janvier 2015
3  # Attention nous supprimons les lignes vides dans les scripts puir
   des raisons d'économie de papier, mais nous en mettons
   sytématiquement dans tous les scripts que nous écrivons
4  # Les expressions en bash
5  #Initialisation
6  let nb1=100129901099
7  let nb2=97
8  let pos=11
9  let long=5
10 let borne_inf=10
11 let borne_sup=100
12 ch1=aI234567890ABCDEFazertyCDEBILE813-666666
13 # Expressions arithmétiques
14 let nb3=$nb1*$nb2
15 echo "$nb1 * $nb2 = $nb3"
16 let nb3=`expr $nb1 % $nb2`
17 echo "$nb1 % $nb2 = $nb3"
18 # Expressions chaînes
```

```
19    echo 'Avec ${ } le premier caractère à la rang 0 !'
20    sousch=${ch1:$pos:$long}
21    echo "Souschaine de $ch1 de $long car. début en $pos =
   $sousch."
22    echo "Avec expr le premier caractère à la rang 1 !"
23    sousch=`expr substr $ch1 $pos $long`
24    echo "Souschaine de $ch1 de $long car. début en $pos =
   $sousch"
25    echo "Avec expr sans saut du premier caractère"
26    echo "Nombre de chiffre au début de $ch1 est `expr "$ch1" : '[0-
   9]*'`."
27    echo "Avec expr saut du premier caractère"
28    echo "Chiffres au début de $ch1 sont `expr "${ch1:1}" : '\([0-
   9]*\)'`."
29    echo "Sans expr"
30    echo "Nombre de chiffres au début de $ch1 est `expr "$ch1" : '[0-9]*'`."
31    echo "Chiffres au début de $ch1 sont `expr "${ch1:1}" : '\([0-9]*\)'`."
32    # Expressions logiques
33    if [ $nb2 -gt $borne_inf -a $nb2 -lt $borne_sup ]; then echo
   "c'est vrai $nb2 est compris entre 10 & 100 !" ; else echo 'faux' ; fi
34    echo 'Suppression des caractères entre 1 & 6: en fin '${ch1%1*6}
```

CORRIGÉ ÉNONCÉ 2 PETIT-JOUEUR

```
#!/bin/bash
# nb_jours.sh - Michel Scifo – février 2015
   # passage de 'jj/mm/aa' à '20aammjj'
   a="20"${1:6:2}${1:3:2}${1:0:2}
   b="20"${2:6:6}${2:3:2}${2:0:2}
   # %s format de présentation de la date en secondes, -d indique que
la chaîne suivant est une date.
```

```
let c=$(date -d $a +%s)
let d=$(date -d $b +%s)
let nb=$d-$c
let nb=$(($nb/(24*3600)))
echo "Il y a $nb jours entre le `date -d $a +%d/%m/%Y` & le `date -d $b +%d/%m/%Y`."
```

Ex. 1

Démarche

Il faut répéter l'affichage des nombres de 1 à n. Il faut donc employer une instruction d'itération. Comme l'on sait qu'il y aura n exécutions, l'instruction **for** paraît indiquée. Sa forme canonique, dans ce cas, ressemblera à **for** var **in** {1**..**$n} **do** ...

Laurent a préféré employer la forme issue du *langage C* qu'il connaît mieux que le *bash*.

Cependant avant de se lancer dans l'écriture d'un script, il est recommandé de regarder s'il n'existe pas une commande faisant déjà tout ou partie du travail.

La combinaison des commandes **apropos** & **grep** donne le résultat suivant.

```
$ apropos numbers | grep '(1)'
addr2line (1)      - convert addresses into file names and line numbers.
factor (1)         - factor numbers
numfmt (1)         - Convert numbers from/to human-readable strings
perlnumber (1)     - semantics of numbers and numeric operations in Perl
seq (1)            - print a sequence of numbers
```

Une séquence étant une suite de nombre, notre problème devrait se simplifier. L'emploi de la commande **man** 1 seq fournira les renseignements indispensables au corrigé 2. Dans certain cas, il nous faudra consulter en outre l'aide liée à la commande **info**.

Ici le résultat est parfait, le plus souvent, il fournit des indications partielles, des pistes de réflexion.

Remarque : Il faut penser à fournir des mots anglais à la commande **apropos** *bien que le nom de la commande paraisse français, car apropos est un mot anglais calqué sur à propos !*

CORRIGÉ 1

```
1  #!/bin/bash
2  # par Laurent VILLETTE
   # En mettant le do sur une ligne différente on évite d'écrire le ; après la condition.
3  for ((var=1; $var<=$1; var++))
4  do
5    echo -n "$var "
6  done
   # Cette commande assure le passage à la ligne suivante pour l'affichage du prompt du système.
7  echo
```

CORRIGÉ 2 (MÉTHODE KISS)

En utilisant **seq** !

```
1  #!/bin/bash
2  # enumnum.sh V2- Michel Scifo - novembre 2015
3  seq -s " " 1 .. $1
```

CORRIGÉ 3 (MÉTHODE KICK)

Mais la lecture attentive du manuel de *bash* (dans PDTU 2, par exemple) permettait de noter qu'il existe des expressions séquences notées entre accolades, comme {1..20}. Ces expressions ont un défaut, elle ne savent pas évaluer le contenu d'un variable. En

d'autres termes, `echo {1..$1}` affiche {1..$1} [1]. Il faut utiliser la commande interne `eval` pour y arriver.

```
1  #!/bin/bash
2  # enumenum V3 – Michel Scifo – septembre 2015
3  $( eval echo {1..$1} )
```

La commande `eval` devra être utilisée après la commande `in` liée au `for` : `for var in $( eval echo {1..$1} )` ...

EX. 2

DÉMARCHE

Il y aura deux variables :

- le prénom passé en paramètre, par défaut, il sera noté $1, comme ce n'est pas très clair, nous affecterons $1 à une variable prenom ;
- la réponse à la question, nous l'appellerons reponse.

Les messages de DOMINIQUE BILLARD ne sont pas ceux que nous avons proposés, mais ils ne changent rien à la structure du script !

CORRIGÉ 1

```
1  #! /bin/bash
2  # salut.sh - Dominique Billard & Michel Scifo– janvier 2015
3  prenom=$1
4  # Notez que l'on peut afficher le contenu d'une variable aussi bien
   dans les guillemets qu'en dehors !
5  read -p "Bonjour $prenom ! comment allez vous ce matin ?
   (bien/mal) " reponse
6  case $reponse in
7    bien) echo "Belle journée pour travailler n'est ce pas
```

1 Bien que le signe dollar comporte parfois une barre ($) & parfois deux ($), cela ne change ni sa valeur ni sa fonction !

```
"$prenom" !";;
    mal) echo $prenom", vous devriez aller boire un petit café avant de travailler !";;
    *) echo "Mauvaise réponse "$prenom" mais bonne journée quand même !"
esac
```

Corrigé 2

Voici une autre façon de répondre à l'énoncé sans employer la commande `case`.

```
#!/bin/bash
# salut.sh – Dominique Billard – janvier 2015
prenom=$1
read -p "Bonjour "$prenom" ! Comment allez vous ce matin,  bien ou mal ? " reponse
if [ $reponse = "bien" ]; then
    echo "Belle journée pour programmer n'est ce pas "$prenom" !"
elif [ $reponse = "mal" ]; then
    echo $prenom", Vous devriez aller boire un petit café avant de programmer !"
else
    echo "Mauvaise réponse "$prenom" mais bonne journée quand même !"
fi
```

Elle est plus facile à écrire car on ne pense à la commande case qu'après avoir vu l'imbrication des instructions **if**, mais elle présente deux inconvénients, elle est un peu plus lente &, si l'on ne saisit rien, elle affiche deux messages d'erreur.

```
./test: ligne 5 : [: = : opérateur unaire attendu
./test: ligne 7 : [: = : opérateur unaire attendu
```

Mauvaise réponse Jo mais bonne journée quand même !

Comment pourriez-vous les faire disparaître sans recourir à la commande **case** ?

Il n'y a pas de corrigé de l'exercice 3 !

EX. 4

DÉMARCHE DE BASE

Dans ces trois scripts, l'essentiel du travail porte sur les conditions, puisque le traitement dans l'itération ne change pas.

En effet, il ne faut pas réfléchir beaucoup pour comprendre que pour tester la validité de limite il faut employer une itération.

Pour cela, il faut bien comprendre qu'une condition peut être une expression logique ou une liste.

Ces tableaux devraient vous permettre de mieux comprendre.

CONDITION	NÉGATION	TEST	EXPR
Égal			
nb1=nb2	nb1<>nb2	$nb1 -eq $nb2	$nb1 [==|=] $nb2
		$nb1 -ne $nb2	$nb1 != $nb2
Inférieur			
nb1<nb2	nb1>=nb2	$nb1 -lt $nb2	$nb1 < $nb2
		$nb1 -ge $nb2	$nb1 >= $nb2
Supérieur			
nb1>nb2	nb1<=nb2	$nb1 -gt $nb2	$nb1 > $nb2
		$nb1 -le $nb2	$nb1 <= $nb2
Compris dans un intervalle bornes exclues			
nb1>nb2 et nb1<nb3	nb1<=nb2 ou nb1=>nb3	$nb1 -gt $nb2 -a $nb1 -lt $nb3	$nb1 > $nb2 & $nb1 < $nb3
		$nb1 -le $nb2 -o $nb1 -ge $nb3	$nb1 <= $nb2 | $nb1 >= $nb3
Compris dans un intervalle bornes inclues			
nb1=>nb2 et nb1<=nb3	nb1<nb2 ou nb1>nb3	$nb1 -ge $nb2 -a $nb1 -le $nb3$	$nb1 >= $nb2 & $nb1 <= $nb3
		nb1 -lt $nb2 -o $nb1 -gt $nb3	$nb1 < $nb2 | $nb1 > $nb3
Listes			

TYPES DE LISTE	OPÉRATEURS	EXEMPLES
Exécution d'une commande	Accents graves	`` `ls -ai` `` `` `cat fichier.txt` ``
Contenu d'un fichier	Redirection d'entrée	< fichier.txt
Liste des paramètres positionnels	Paramètres spéciaux	$* $@
Séquence de nombres	Expression séquence	{1..7} eval echo {$deb..$fin} `` `seq 1 $fin` ``

CORRIGÉ COMPTE_W.SH

```
1  #! /bin/bash
2  # compte_w.sh Dominique Billard & Michel Scifo – janvier 2015
3  read -p "Donnez un nombre entre 6 & 9 : " limite
4  if  [ $limite -le 5 -o $limite -ge 10 ]; then
5    nombre=1
6    while [ $nombre -le $limite ]; do
7      echo -n $nombre" "
8      let nombre+=1
9    done
10   echo
11 else
12   echo "Nombre incorrect !"
13 fi
```

Résultat

```
Donnez un nombre entre 6 & 9 : 7
1 2 3 4 5 6 7
```

CORRIGÉ COMPTE_U.SH

Ces deux corrigés intègrent la Variante 1

```
1  #! /bin/bash
2  # compte_w.sh Dominique Billard & Michel Scifo – janvier 2015
3  let limite=0
4  until [ $limite -gt 5 -a $limite -lt 10 ]; do
5    read -p "Donnez un nombre entre 6 & 9 : " limite
6  done
7  nombre=1
8  until [ $nombre -gt $limite ]; do
9    echo -n $nombre" "
10   let nombre+=1
11 done
12 echo
```

Résultat

```
Donnez un nombre entre 6 & 9 : 7
1 2 3 4 5 6 7
```

Corrigé compte_f.sh

```
1  #! /bin/bash
2  # compte_w.sh Michel Scifo – janvier 2015
3  let limite=0
4  # Ici, il n'est pas possible d'employer deux commandes for, car
   nous ignorons le nombre de répétitions possibles pour la première
   boucle.
5  while [ $limite -le 5 -o $limite -ge 10 ]; do
6    read -p "Donnez un nombre entre 6 & 9 : " limite
7  done
8  for nombre in $(eval echo {1..$limite}); do
9    echo -n $nombre" "
10 done
11 echo
```

Résultat

```
Donnez un nombre entre 6 & 9 : 7
1 2 3 4 5 6 7
```

CORRIGÉ VARIANTE 2

```
1  #! /bin/bash
2  # 43210.sh – Dominique Bilard & Michel Scifo – janvier 2015
3  for nombre in {10..0}; do
4    if [ $nombre -eq 0 ]; then
5      echo "Feu !"
6    else
7      echo $nombre
8      sleep 1
9    fi
10 done
```

CORRIGÉ VARIANTE 3

```
1  #! /bin/bash
2  # compte_u.sh v3 – Dominique Billard – janvier 2015
3  let nombre=1
4  until [ $nombre -gt $1 ]; do
5    echo -n $nombre" "
6    let nombre+=1
7  done
8  echo
```

EX. 5

DÉMARCHE

Rappel : Une fonction peut être définie n'importe ou dans le script, mais avant sa première utilisation. Nous placerons celles que nous écrivons en début de script.

Corrigé

```
1   # !/bin/bah
2   # lejustenombre.sh – Dominique Billard & Michel Scifo – janvier 2015
3   function obtient_essai
4   {
5     let Essai=0 # Essai vient de recevoir une valeur il faut donc la
  réinitialiser à 0 pour entrer dans l'itération
6     while [ $Essai -lt 1 -o $Essai -gt 100 ]; do
7       read -p "Entrez un nombre entre 1 et 100 : " Essai
8     done
9   }
10
11    let A_trouver=$(($(($(($(($(`date +%M` * 60)) + `date +%S`)) % 100))
  + 1))
12    let Essai=0 # assignation nécessaire pour entrer dans la boucle
  until
13    until [ $Essai -eq $A_trouver ]; do
14      obtient_essai
15      if [ $Essai -lt $A_trouver ]; then
16        echo "Trop petit !"
17      elif [ $Essai -gt $A_trouver ]; then
18    echo "Trop grand !"
19      fi
20    done
21    echo "****BRAVO**** vous avez gagné !"
```

EX. 6

DÉMARCHE

Le moyen le plus simple pour constituer une liste avec des nombres discontinus & sans liens logiques & de les juxtaposer dans une chaîne de caractères séparés par une espace.

Avec certaines distributions, les utilitaires se trouvent dans /sbin plutôt que dans /usr/sbin. C'est la raison pour laquelle, comme iptables se trouve dans un dossier de la variable PATH, vous pouvez vous dispenser d'écrire son chemin absolu. C'est ce qui explique la présence des [&] dans la liste ; il vous faudra les enlever, si vous faites un copier/coller.

Il faut passer en administrateur pour exécuter ce script.

La commande grep permet de n'afficher que les règles concernant les blocages de port. Si vous l'enlevez vous verrez l'ensemble des règles d'iptables sur votre système.

CORRIGÉ

```
1  #!/bin/bash
2  # no_trojan.sh – Dominique Billard – janvier 2015
3  port_interdit="1234 2222 3333 4321 5555 6666 9999 12345"
4  for port in $port_interdit; do
5    [/usr/sbin/]iptables -A INPUT -p udp --dport $port -j DROP
6    [/usr/sbin/]iptables -A INPUT -p tcp --dport $port -j DROP
7  done
8  [/usr/sbin/]iptables -L -n -v | grep DROP
```

EX. 7

DÉMARCHES

L'usage des tableaux en *bash* suscite deux problèmes :

- l'usage d'une syntaxe lourde pour accéder aux données **${**tab[0]**}**;

- la nécessité de multiplier les variables, car pour obtenir, par exemple le premier caractère de la chaîne 'Trax' on ne peut écrire
  `echo ${tab[0]:0:1}`
  il faut écrire
  `aux=${tab[0]}`
  `echo ${aux:0:1}`

Pour cette raison, dans le corrigé 1, nous emploierons six variables jeu1 à jeu6 pour recevoir les noms des jeux.

Comme il s'agit d'un menu nous utiliserons la commande **select**.

Le corrigé 2, montre la syntaxe avec tableaux, en ajoutant un tableau associatif.

CORRIGÉ 1

```
1  #!/bin/bash
2  # liste_jeux.sh – Dominique Billard – janvier 2015 – aux jeux &
   aux messages près
3  jeu1=Trax
4  jeu2="Othello_10x10"
5  jeu3=Pente
6  jeu4="Puissance 4x4"
7  jeu5="Mauvaise Paye"
8  jeu6="Guerre nucléaire"
9  message="Vous avez choisi de jouer "
10 echo "Bonjour, à quoi voulez-vous jouer ? "
11 select choix_jeu in "$jeu1" "$jeu2" "$jeu3" "$jeu4" "$jeu5" "$jeu6";
12 do
13     case "$choix" in
14         "$jeu1"|"$jeu3") echo $message"au "$choix;
15         break;;
16         "$jeu2"|"$jeu4") echo $message"à "$choix;
```

```
        break;;
        "$jeu5"|"$jeu6") echo $message"à la "$choix;
        break;;
        *) echo "Erreur, jeu inconnu !!"
    esac
done
if [ "$choix" = "$jeu6" ]; then
    echo "Est-ce bien raisonnable ??"
else
    echo "C'est un bon choix pour aujourd'hui !!"
fi
```

CORRIGÉ 2

```
#!/bin/bash
# liste_jeux.sh -Michel Scifo – janvier 2015
declare -a jeux
echo
jeux=(Trax Othello_10x10 Pente Puissance_4x4 Mauvaise_Paye Guerre_nucléaire)
echo
declare -A messages
messages=(
    [${jeux[0]}]=" est un jeu qui ne laisse pas de traxe !"
    [${jeux[1]}]="Ne restez pas sec après avoir Othello !"
    [${jeux[2]}]=" est un bon jeu, mais ne suivez pas une mauvaise pente !"
    [${jeux[3]}]="Même puissant, un 4x4 est nocif !"
    [${jeux[4]}]="Ce n'est pas un jeu, c'est la réalité !"
    [${jeux[5]}]="Ce n'est pas une bonne idée ! J'espère que vous plaisantez !"
```

```
15  )
16
17  echo "Choisissez un jeu SVP !"
18  select choix in ${jeux[*]}; do
19    clear
20    case $choix in
21      ${jeux[0]}|${jeux[2]})
22        echo $choix ${messages[$choix]}
23        break;;
24      ${jeux[1]})
25        echo ${messages[$choix]/Othello/${choix/_/\ }}
26        break;;
27      ${jeux[3]})
28        echo ${messages[$choix]}
29        break;;
30      ${jeux[4]}|${jeux[5]})
31        echo ${messages[$choix]}
32        exit;;
33      *) echo "Jeux inconnu !"
34    esac
35  done
36  echo "Excellent choix, mais il va vous falloir programmer "${
    choix/_/\ }" pour y jouer :-))"
```

Corrigés exercices complexes

Ex. II bataille navale

Démarche

Nous allons présenter deux approches du script. Celle de Dominique Billard & la notre. Celle de Dominique, emploie un minimum de com-

mandes & d'opérateurs, mais elle nécessite, au départ, un effort de conceptualisation, peut-être moins usuel. L'importante différence de longueur entre les deux versions provient pour l'essentiel, d'une simplification du problème du placement, car d'une part, il autorise deux bateaux à se toucher & d'autre part, la deuxième case y est toujours placé horizontalement (Le traitement de ces deux points nécessite plus de 80 lignes de script). Nous avons procédé de même, avant d'analyser le problème complexe du placement. Certains la trouveront plus simple à mettre en œuvre que la nôtre, qui emploie plus de commandes & d'opérateurs, mais qui suit notre analyse de près.

CORRIGÉ DBD

```
1  #!/bin/bash
2  # © Dominique Billard janvier 2015
3  function affiche_regle
4  {
5    echo "                    Bataille Navale"
6    echo "Nous allons cacher deux bateaux un d'une case & l'autre de 2, mais"
7    echo " consécutives."
8    echo "Il faudra 2 tirs au but pour couler le deuxième bateau,"
9    echo " un, pour le premier."
10   echo "Vous taperez les coordonnées de la case visée & le programme vous"
11   echo "donnera le résultat de votre tir en affichant '*1' ou '*2',"
12   echo "si le coup a atteint un bateau, '++' dans le cas contraire,"
13   echo "à la place des coordonnées."
14   echo "Le programme affichera également 'touché' ou 'coulé', selon le cas."
15   echo "Lorsque les deux bateaux sont coulés affichage de"
16   echo "Vous avez réussi la mission, game over !!"
```

```
17      echo
18      echo "Affichage au début du jeu"
19      echo "------------------------ "
20      echo " | A1  | A2  | A3  | A4 | "
21      echo "------------------------ "
22      echo " | B1  | B2  | B3  | B4 | "
23      echo "------------------------ "
24      echo " | C1  | C2  | C3  | C4 | "
25      echo "------------------------ "
26      echo "entrez la case choisie "
27      echo
28      echo "Exemple d'affichage en fin de jeu"
29      echo
30      echo "------------------------ "
31      echo " | A1  | ++  | ++  | ++ | "
32      echo "------------------------ "
33      echo " | *2  | *2  | ++  | B4 | "
34      echo "------------------------ "
35      echo " | *1  | ++  | C3  | ++ | "
36      echo "------------------------ "
37      echo "vous avez réussi la mission game over !!"
38      echo "                  ***"
39  }
```

Dominique a choisi de présenter le jeu comme une combinaison de caractères plutôt que comme l'affichage de cinq chaînes. Cela lui permet d'écrire une fonction d'affichage plus simple.

Notez l'usage des variables i, j, k : ce sont des compteurs : des variables auxiliaires désignés par une lettre. Pour des raisons historiques, il est d'usage de nommer le premier indice i, le second j, le troisième k, etc.

```
40  function affiche_jeu
```

```
{
  clear
  # affichage du tableau de jeu sous forme des réponses possibles
  # ou de ++ si la case a déja été tirée
  # de *1 ou *2 si coup au but sur bato1 ou bato2
  k=1
  for i in {A..C};do
    for j in {1..25};do
      echo -n "-"
    done
    echo
    for j in {1..4};do
      case $((case_jeu[$k])) in
        0|1|2) echo -n " | $i$j ";;
            3) echo -n " | *1 ";;
            4) echo -n " | *2 ";;
            5) echo -n " | ++ "
      esac
      let k+=1
    done
    echo "|"
  done
  for j in {1..25}; do
    echo -n "-"
  done
echo
}
function cherchresultat
{
    if [ $((case_jeu[$1])) -eq 1 ]; then
```

```
71          echo "bateau 1 coulé !!"
72          let touche_bato1+=1
73          case_jeu[$1]=3
74      elif [ $((case_jeu[$1])) -eq 2 ]; then
75          let touche_bato2+=1
76          if [ $touche_bato2 -eq 2 ]; then
77              echo "bateau 2 coulé !!"
78              case_jeu[$1]=4
79          else
80              echo "bateau 2 touché !!"
81              case_jeu[$1]=4
82          fi
83      else
84          echo "raté !!"
85          case_jeu[$1]=5
86      fi
87      if [ $((touche_bato1+$touche_bato2)) -eq 3 ]; then
88          affiche_jeu
89          echo "Vous avez réussi la mission game over !!"
90          gagne=vrai
91      else
92          gagne=faux
93      fi
94  }
95  # la case avec le bateau 1 aura la valeur 1
96  # les cases avec le bateau 2 auront la valeur 2
97  clear     # on efface la console
98  affiche_regle
99  hasard=$RANDOM
100 let bato2=($hasard%12)+1 #tirage aléatoire de la place du 2e
```

```
    bateau
101 # initialisation des cases elles valent 0 si rien 1 ou 2 si bateau
102 # création d'un pseudo tableau de 4*3 cases
```

En fait ce tableau a 13 cases, mais, sa case 0 n'est jamais employée. DOMINIQUE, profitant de la permissivité de *bash*, évite ainsi d'enlever 1 systématiquement à l'indice.

```
103 for i in {1..12}; do
104   let case_jeu[$i]=0
105 done
106 # positionnement des bateaux
107 let case_jeu[$bato2]=2
108 # Le bato2 ne peut pas être coupé en deux par exemple en B4C1
109 case $bato2 in
110     4|8|12) let case_jeu[$((bato2-1))]=2;;
111           *) let case_jeu[$((bato2+1))]=2
112   esac
113 hasard=$RANDOM
114 let bato1=($hasard%12)+1
115 while [ case_jeu[$bato1] = 2 ]; do  # les 2 bateaux ne doivent pas
    être à la même place
116   hasard=$RANDOM
117   let bato1=($hasard%12)+1
118 done
119 let case_jeu[$bato1]=1
120 gagne="faux"
121 touche_bato1=0
122 touche_bato2=0
123 repind=0
124 # Boucle principale du jeu, on en sort lorsque le
125 # joueur a gagné
```

```
126  while [ $gagne = faux ]; do
127    affiche_jeu
128    read -p "entrez la case choisie " reponse
129    # On transforme la réponse en index pour notre tableau
130    case ${reponse:0:1} in
131      A) let repind=${reponse:1:1};;
132      B) let repind=${reponse:1:1}+4;;
133      C) let repind=${reponse:1:1}+8;;
134      *) echo "erreur de saisie"
135    esac
136    # gagne=`cherchresultat $repind`
137    cherchresultat $repind
138    sleep 2
139  done
```

CORRIGÉ MSO

INITIALISATION

```
1   #!/bin/bash
2   # Michel Scifo © mars 2015
3   # Programme de bataille navale dans un petit plan d'eau
4   # Ce script est sous licence Creative Uncommons CA-FÉ-IN-ÉS
5
6   # INITIALISATION
7   lig_A='| A1 | A2 | A3 | A4 |'
8   lig_B='| B1 | B2 | B3 | B4 |'
9   lig_C='| C1 | C2 | C3 | C4 |'
10  declare plan_do=(0 0 0 0 0 0 0 0 0 0 0 0 0)
11  declare pos_coord=(2 7 12 17)
```

AFFICHER_PLAN_DO

```
12  function dessine_plan_d_o
13  {
14    # calcul de la ligne modifiée
15    lig=${1:0:1}
16    case $lig in
17      A)aux=$lig_A;;
18      B)aux=$lig_B;;
19      C)aux=$lig_C;;
20    esac
21    let index=$((${1:1}-1)) # on enlève 1 car les tableaux sont
    indexés à partir de 0.
22    let longdeb=${pos_coord[$index]}
23    let debfin=${pos_coord[$index]}+2
24    if [ $2 != '--' ]; then
25      aux=${aux:0:$longdeb}$2${aux:debfin}
26    fi
27    case $lig in
28      A)lig_A=$aux;;
29      B)lig_B=$aux;;
30      C)lig_C=$aux
31    esac
32    # affichage du plan d'eau
33    local lig_const="---------------------"
34    clear
35    echo $lig_const
36    echo $lig_A
37    echo $lig_const
38    echo $lig_B
39    echo $lig_const
40    echo $lig_C
```

```
41      echo $lig_const
42      # affichage du commentaire
43      case $2 in
44       '++') echo 'Manqué !';;
45       'x1') echo 'Bateau 1 coulé !';;
46       'x2') echo 'Bateau 2 touché ou coulé !';;
47       '--') echo 'Déjà tiré !'
48      esac
49      # demande du coup suivant
50      echo "Case visée ? "
51    } # fin dessine_plan_d_o
```

CALCUL_LIGNE

Le sort de cette fonction est un classique de la programmation : bien qu'elle soit une fonction indépendante, fonctionnant testée isolément, il ne nous a pas été possible de la faire fonctionner indépendamment de **dessine_pla_do**. Du coup nous l'avons intégrée dans cette dernière (lignes 14 à 31).

Cette fonction place le résultat du tir dans la ligne adaptée. Notez l'utilisation du contenu d'une case de tableau.

```
function calcul_ligne
{
# $l est la ligne à modifier
declare index longdeb debfin
let index=$((${coup:l}-1)) # on enlève 1 car les tableaux sont  indicés à partir de 0
let longdeb=${pos_coord[$index]}
let debfin=${pos_coord[$index]}+2
echo ${1:0:$longdeb}$resultcoup${1:$debfin}
}
```

Elle sera appelée, avec une ligne comme
lig_A=$(calcul_ligne $"$lig_A")

* Les variables auxiliaires index, longdeb, & debfin n'ont pas d'utilité en dehors de la fonction. Les déclarer dans la fonction empêche qu'elles soient connues en dehors. Cela peut éviter des effets de bord ! Un effet de bord se produit quand, par exemple, vous employez une autre variable auxiliaire de même nom, ailleurs dans le script, & que cette dernière se trouve avoir la valeur assignée dans la fonction, au lieu de celle attendue !
* Afin de récupérer le résultat d'une fonction, il faut la faire évaluer comme une expression avec l'opérateur $(...). Le résultat récupéré l'est grâce à un affichage. Contrairement à ce qui existe dans d'autres langages, la commande return ne peut fournir le résultat de la fonction que si celui-ci est un nombre compris entre 0 & 255. Raison pour laquelle on l'emploie, surtout, comme exit pour les valeurs d'erreurs.
* Pour passer la valeur d'une variable à une fonction il faut employer l'opérateur $"...".

OBTIENT_TIR_OK & CONVERT_TIR

La première demande les coordonnées de la case visée & vérifie qu'elles sont correctes, la seconde les convertir en numéro de case de plan_do.

```
52  function obtient_tir_ok
53  {
54    until `false`; do
55      read -n 2  tir
56      tir=${tir^} # transforme la miniscule éventuelle en majuscule
57      lig=${tir:0:1}
58      case $lig in
59        A|B|C)
```

```
60            col=${tir:1:1}
61            kase=
62            case $col in
63              1|2|3|4) echo ${tir}
64               break;;
65              *)continue
66            esac;;
67            *) continue
68          esac
69        done
70      } # fin obtient_tir_ok
71
72      function convert_tir
73      # obtention de la case de plan_do
74      {
75        aux=${tir:0:1}
76        xau=${tir:1:1}
77
78        case $aux in
79          A) kasox=$(($xau-1));; # on enlève 1 car le tableau est
    numéroté à partir de 0
80          B) kasox=$(($xau+3));;#          "
81          C) kasox=$(($xau+7));; #          "
82        esac
83        echo $kasox
84      } # fin convert_tir
```

Cette fonction illustre toute la difficulté de la programmation en *bash*. En effet, ce langage prévoit d'extraire aisément une sous-chaîne à partir de sa position dans la chaîne, mais, uniquement pour vérifier sa présence, il faut recourir aux expressions ration-

nelles, car le mot réservé **in** ne fonctionne pas avec la commande **if** (**if** lettre **in** {A..C} ; ... ne fait rien).

Soit on emploie une expression rationnelle avec une syntaxe ressemblant à `` `expr "ABC": '${tri:0:1}'` ``, soit on emploie la commande externe `sed`. Si l'on est allergique aux expressions rationnelles, on n'a que deux solutions :

- employer une commande **until** comparant successivement la sous-chaîne recherchée aux sous-chaînes successives de même longueur de la chaîne de référence ;
- utiliser une commande **case** pour remplacer le **if**.

Cette dernière solution semblant assez souvent employée, nous la retenons ici.

La commande externe `false` ne fait rien, mais elle le fait mal, elle renvoie toujours la valeur faux ; comme son inverse, la commande `true`, elle permet de créer des boucles infinies dont on sort au moyen de la commande **break**. Elle évite de trop réfléchir à la condition d'arrêt quand celle-ci s'avère complexe. Ce n'est pas le cas ici, mais une condition d'arrêt demandant, dans ce cas, l'emploi d'une variable auxiliaire, le recours à `false` peut se justifier.

ANA_RESULT

Cette fonction calcule le résultat du tir. Afin d'économiser quelques lignes nous l'avons complétée d'une fonction auxiliaire gérant l'état touché ou coulé du deuxième bateau.

Presque systématiquement, quand il y des réponses multiples à un test & que certaines réponses sont similaires, nous employons des réponses par défaut à ces tests, afin d'économiser du temps d'exécution & des lignes de programme pas indispensables.

```
85  function ana_result
86    # $1 numéro de la case visée
87  {
```

```
88    let cont_case=${plan_do[$kase]}
89    case $cont_case in
90      0) # raté
91        let cont_case=3;;
92      1) # bateau1
93        let cont_case=10;;
94      2) #bateau 2
95        let cont_case=20;;
96      *) # déjà tiré
97        let cont_case=5
98    esac
99    echo $cont_case
100 } # ana resultat
101
102 function chaine_result
103 # $1 résultat numérique
104 {
105   case $1 in
106     3) aux='++';;
107    10) aux='xl';;
108    20) aux='x2';;
109     5) aux='--'
110   esac
111   echo $aux
112 } # chaine resultat
113
114 function gagnai
115 {
116   aux=0
117   for i in {0..2}; do
```

```
118       indx=${places_bat[$i]}
119       aux=$(($aux+plan_do[$indx]))
120     done
121     aux=$(($aux%10))
122     echo $aux
123   } # fin_gagnai
```

PLACEMENT DES BATEAUX

Il n'y a pas de problème pour le premier bateau. En revanche, il y a des cases dans lesquelles on ne peut placer des cases du second bateau & d'autres cases qui sont imposées pour la seconde case du second bateau. Faisons les deux listes.

B1	B2 KO					B2-C1	B1-C2 OK			
0	0	1	4			0	1	4		
1	1	0	2	5		1	0	2	5	
2	2	1	3	6		2	1	3	6	
3	3	2	7			3	2	7		
4	4	0	5	8		4	0	5	9	
5	5	1	4	6	9	5	1	4	6	9
6	6	2	5	7	10	6	2	5	7	10
7	7	3	6	11		7	3	6	11	
8	8	4	9			8	4	9		
9	9	5	8	10		9	5	8	10	
10	10	6	9	11		10	6	9	11	
11	11	7	10			11	7	10		

On le voit le traitement s'avère complexe, pour la première case, il faut vérifier qu'elle n'est pas dans une des cases interdites & pour la seconde qu'elle est dans une case autorisée qui n'est pas inter-

dite. Il existe,cependant un moyen de simplifier le traitement : considérer que le premier bateau est celui de deux cases. Dans ce cas plus de problème pour la première case & la seconde doit être une de celles autorisées. Quant au second bateau il suffit qu'il ne soit pas dans une des cases interdites.

En regardant les valeurs, on voit que sur une même ligne les cases adjacentes ont un rang diminué ou augmenté de 1 & sur des lignes différentes de 4.

1 place_case2

La deuxième case du bateau est perpendiculaire à la première, elles se trouve à l'est, au nord, à l'ouest ou au sud de la première il y a deux cas problématiques :

- la première case est sur un bord & le nb tiré indique une case inexistante ;
- le nb tiré indique une case occupé par l'autre bateau.

Il faut donc recommencer le tirage jusqu'à l'obtention d'une case jouable.

```
inutile ← vrai
Tant que inutile = vrai répéter
  tirer un nb entre 0 & 3 (modulo 4)
  calculer les coordonnées en fonction du nombre
    0 : lettre constante, chiffre - 1
    1 :  lettre précédente, chiffre constant
    2 : lettre constante, chiffre + 1
    3 : lettre suivante, chiffre constant
  fin choix
  inutile ← lettre pas dans A..C ou chiffre pas dans 1..4
  inutile ← inutile et coord=place_bat[0]
fin tantque
place_bat[2] ← coord
```

De fait, afin d'éviter des calculs complexes, il faudrait employer une chaîne '@ABCD', pour les lettres, & un intervalle *0..5*, pour les chiffres, avec une suite d'instructions qui pourrait ressembler à :

```
inutile ← 1
case $coordy in
  '@'|'D') inutile ← 0;;
  *)
    case $coordx in
      0|5) inutile ← 0
    esac
esac
```

Il reste à trouver l'indice de la lettre dans la chaîne avec la commande expr index $chaine $lettre.

De fait, il semble plus simple de placer d'abord le second bateau & ensuite le premier. Cela ne change rien aux considérations précédentes sur les cases disponibles. Nous allons y revenir !

CŒUR DU SCRIPT

```
124  # début du programme
125  affiche_regle
126  # placement des bateaux
127  for i in 1 2 3; do
128    case $i in
129      3) kase=$(`bateau1`);; # on place le bateau 1 en dernier
130      1) let kase=$(($RANDOM%12));;
131      2) let kase=$(`case2_2 $kase`)
132    esac
133    if [ $i -eq 3 ]; then
134      places_bat[0]=$kase
135      plan_do[$kase]=1
```

```
136     else
137       places_bat[$i]=$kase
138       plan_do[$kase]=2
139     fi
140   done
141   # boucle de jeu gain vaut 0 & gagnai vaut 3
142   until [ `gagnai` -eq 0 ]; do
143     tir=`obtient_tir_ok`
144     let kase=`convert_tir $tir`
145     let plan_do[$kase]=`ana_result $kase`
146     echo "gain=$gain"
147     res=`chaine_result ${plan_do[$kase]}`
148     dessine_plan_d_o $tir $res
149   done
150   echo "Bravo ! vous avez gagné !"
```

Fonctions de placement

De ces deux fonctions, c'est la première qui s'avère la plus complexe. Sans réduire la complexité, on pourrait, afin de réduire le nombre de ligne du script, considérer un tableau ayant une ligne de plus en haut & bas & une colonne de plus en début & en fin. Nous n'en ferons rien. En revanche, nous vous montrons comment nous avons déterminé les cases à problème à l'aide d'un petit tableau réalisé avec *LibreOffice Calc*.

CASE	CASE -4	CASE -1	CASE+1	CASE+4
0	-4	-1	1	4
1	-3	0	2	5
2	-2	1	3	6
3	-1	2	4	7
4	0	3	5	8
5	1	4	6	9
6	2	5	7	10
7	3	6	8	11
8	4	7	9	12
9	5	8	10	13
10	6	9	11	14
11	7	10	12	15

Les cases à problème sont celles dont l'indice deviendrait hors borne & celles consécutives, dans le tableau à une dimension, & situées sur des lignes différentes, dans celui à deux.

```
1  function bateau1
2  {
3    let refaire=0
4    while [ $refaire -eq 0 ]; do # on refait tant que toutes les cases
   alentour ne sont pas vides
5      let aux=$(($RANDOM%12))
6      auxt=${plan_do[$aux]}
7      if [ $auxt -eq 0 ]; then
8        # numéros des cases entourant la case du premier bateau
9        if [ $aux -lt 11 ]; then let c=$(($aux+1)); fi
10       if [ $aux -gt 0 ]; then let a=$(($aux-1)); fi
11       if [ $aux -gt 3 ]; then let b=$(($aux-4)); fi
12       if [ $aux -lt 8 ]; then let d=$(($aux+4)); fi
13       case $aux in # Tests du contenu des cases alentour (elles
```

doivent toutes être vides) en fonction de la valeur possible pour la deuxième case.

```
14            0)  pdoa=${plan_do[$a]}
15                pdod=${plan_do[$d]}
16                if [ $pdoa -eq 0 ] && [ $pdod -eq 0 ]; then
17                  refaire=1
18                fi;;
19            1,2)pdoa=${plan_do[$a]}
20                pdoc=${plan_do[$c]}
21                pdod=${plan_do[$d]}
22                if [ $pdoa -eq 0 ] && [ $pdoc -eq 0 ] && [ $pdod -eq 0 ];
    then
23                  refaire=1
24                fi;;
25              3)pdoa=${plan_do[$a]}
26                pdod=${plan_do[$d]}
27                if [ $pdoa -eq 0 ] && [ $pdod -eq 0 ]; then
28                  refaire=1
29                fi;;
30            4)  pdob=${plan_do[$b]}
31                pdoc=${plan_do[$c]}
32                pdod=${plan_do[$d]}
33                if [ $pdob -eq 0 ] && [ $pdoc -eq 0 ] && [ $pdod -eq 0 ];
    then
34                  refaire=1
35                fi;;
36            5,6)pdoa=${plan_do[$a]}
37                pdob=${plan_do[$b]}
38                pdoc=${plan_do[$c]}
39                pdod=${plan_do[$d]}
```

```
40              if [ $pdoa -eq 0 ] && [ $pdob -eq 0 ] && [ $pdoc -eq 0 ]
  && [ $pdod -eq 0 ]; then
41                refaire=1
42              fi;;
43          7)  pdoa=${plan_do[$a]}
44              pdob=${plan_do[$b]}
45              pdod=${plan_do[$d]}
46              if [ $pdoa -eq 0 ] && [ $pdob -eq 0 ] && [ $pdod -eq 0 ];
  then
47                refaire=1
48              fi;;
49          8)  pdob=${plan_do[$b]}
50              pdoc=${plan_do[$c]}
51              if [ $pdob -eq 0 ] && [ $pdoc -eq 0 ]; then
52                refaire=1
53              fi;;
54          9,10)pdoa=${plan_do[$a]}
55               pdob=${plan_do[$b]}
56               pdoc=${plan_do[$c]}
57               if [ $pdoa -eq 0 ] && [ $pdob -eq 0 ] && [ $pdoc -eq 0 ];
  then
58                refaire=1
59               fi;;
60          11)  pdoa=${plan_do[$a]}
61               pdob=${plan_do[$b]}
62               if [ $pdoa -eq 0 ] && [ $pdob -eq 0 ]; then
63                 refaire=1
64               fi
65        esac
66      fi
```

```
67      done
68      echo $aux
69    } # placer le bateau 1
70
71    function case2_2
72    {
73      let refaire=0
74      while [ $refaire -eq 0 ]; do
75        nb_has=$(($RANDOM%4))
76        aux=${places_bat[1]}
77        ox=$(($aux%4))
78        kres=12 # contiendra le n°de la 2e case
79        case $nb_has in
80          0) if [ $aux -gt 0 ] && [ $ox -gt 0 ];  then
81               kres=$(($aux-1))
82               let refaire=1
83             fi
84             break;;
85          2) if [ $aux -lt 11 ] && [ $ox -lt 3 ];  then
86               kres=$(($aux+1))
87               let refaire=1
88             fi
89             break;;
90          1) if [ $aux -gt 4 ];  then
91               kres=$(($aux-4))
92               let refaire=1
93             fi
94             break;;
95          4) if [ $aux -lt 8 ];  then
96               kres=$(($aux+4))
```

```
97              let refaire=1
98            fi
99            break
100         esac
101       done
102       echo $kres
103   }
```

EX. 12

Écrire un script qui sauvegarde dans le dossier /var/tmp/sauve les fichiers de votre dossier de connexion qui n'y sont pas encore & qui affiche selon le travail réalisé Le fichier NomDuFichier est déjà sauvegardé ! ou Sauvegarde du fichier NomDuFichier en cours !

DÉMARCHE

Comme la commande **cp** ne sait pas créer un dossier inexistant pour y copier des fichiers, il nous d'abord nous assurer de l'existence du dossier de sauvegarde & le créer si nécessaire.

Nous avons besoin de la liste des fichiers avec le chemin permettant d'y accéder. Dans l'exemple, nous avons choisi le chemin absolu du dossier de départ. Afin de la récupérer, nous emploierons la commande **find**, plutôt que la commande **ls**.

```
$ find /home/m -maxdepth 3 -type d -name "Mu*" | grep -v '\/\.'
/home/m/Musique
/home/m/Musique/Purcell Henry/Music for Queen Mary (Choir of ...)
$ ls -Rl /home/m/Mu*
/home/m/Musique:
Albinoni\ Tomaso\ Giovanni
[...]
```

Outre le fait que **ls** n'indique pas le chemin d'accès aux contenu des sous-dossiers, il liste tous les fichiers pas seulement ceux correspondant au filtre.

Ce que nous avons dit du dossier de départ est valable pour tous les sous-dossiers, il faudra les créer s'ils n'existent pas dans celui d'arrivée.

Nous considérerons que la sauvegarde est à faire si le fichier d'arrivée est inexistant ou plus ancien que le fichier de départ.

Corrigé

```
1  #!/bin/bash
2  # par Laurent VILLETTE
   # Le OU logique (||) étant, ici, exclusif, la commande makedir n'est exécutée que si la valeur testée n'indique pas un dossier !
3  [ -d /var/tmp/sauve$HOME ] || mkdir -p /var/tmp/sauve$HOME
   # La syntaxe de find n'est pas tout à fait correcte. La syntaxe correcte est find ~ -name "*". En effet la syntaxe indiquée ajoute aux fichiers en chemin absolu (/home/utilisateur/nomfichier) ceux en chemin relatif (nomfichier). De plus, si le find n'est pas exécuté depuis la racine, le '/' manquant entre sauve & $fichier crée quelques petites surprises.
4  for fichier in `find ~ *`
5  do
6    if [ -d $fichier ] # traitement d'un dossier
7    then
8      if [ ! -d /var/tmp/sauve$fichier ] # le dossier n'existe pas
9      then
10       echo "----- Création du répertoire /var/tmp/sauve$fichier"
11       mkdir /var/tmp/sauve$fichier
12     fi
```

```
13      else # $fichier est n'est pas un dossier
    # En toute rigueur il faudrait tester d'abord si le fichier existe avant
    de le copier ; s'il n'existe pas & s'il existe faire le test suivant qui
    vérifiera son ancienneté. Mais comme l'opérateur -nt considère un
    fichier inexistant comme plus ancien ça fonctionne.
14        if [ $fichier -nt /var/tmp/sauve$fichier ]
15        then
16          echo "[o] $fichier en cours de sauvegarde..."
17          cp -f $fichier /var/tmp/sauve$fichier
18        else
19          echo "[x] $fichier déjà sauvegardé"
20        fi
21      fi
22    done
```

Ex. 13

Écrire un script permettant de suivre l'occupation du disque correspondants aux répertoires de connexion des utilisateurs (dans /home donc).

Ce script pourra s'exécuter périodiquement (gestion par **crond**). Exécuté interactivement, il demandera une valeur limite pour chaque dossier de connexion que vous stockerez dans le fichier /var/tmp/admin/quotas. Créez le dossier /var/tmp/admin, en ligne de commande, s'il n'existe pas.

Dans les deux cas, il émettra une alarme si un seuil est dépassé, sous forme de mail à l'administrateur.

À vous de retrouver la démarche !

Corrigé 1

```
1   #!/bin/bash
2   # par Laurent VILLETTE
```

```
# La variable $# contient 0 si aucun argument n'a été passé au script. $0 est le premier mot de la ligne : le nom du script.
if [ $# = 0 ]; then
  echo "Syntaxe : $0 <commande>"
  echo "  Commandes :"
  echo "    modif"
  echo "          Lance $0 en mode paramétrage"
  echo "    verif"
  echo "          Lance $0 en mode vérification"
# - Modification ------------------------------------------------
elif [ $1 = modif ]; then
  let status=0
  until [ $status = 1 ]; do
    clear # Il faudrait préciser que les deux premiers choix sont exclusifs l'un de l'autre. Il aurait été préférable d'employer deux fichiers différents, car il s'agit de deux problèmes différents quota global & quotas individuels.
    echo -e "\n[1] Définir une taille limite globale pour le répertoire /home"
    echo "[2] Définir une taille limite individuelle pour chaque répertoire utilisateur"
    echo "[3] Quitter"
# read -n 1 ne lit qu'un seul caractère.
    read -n 1 choix
    case "$choix" in
    1)
      choix="" # Vérification de la saisie d'une valeur. Ce test est insuffisant, car il ne vérifie pas si la valeur est un nombre. Pour y arriver il vous faut employer une expression rationnelle ou évaluer une expression numérique. Si cela vous rebute vous pouvez dans
```

l'aide préciser qu'il n'y a pas de contrôle du nombre & qu'une chaîne de caractères vaudra 0.

```
22          clear
23          echo -e "\nVeuillez saisir une taille maximum en octets :"
24          read quota
25        done
26        echo "/home:$quota" > /var/tmp/admin/quota
27        status=!;;
28      2)
29        rm -f /var/tmp/admin/quota
30    # L'expression n'est pas appropriée, même si elle est correcte : `ls` est plus simple & `cat /etc/passwd | grep "home" | cut -f6 -d:` ou `grep "home" </etc/passwd | cut -f6 -d:` , plus justes, car elles évitent de récupérer des dossiers de /home qui ne sont pas affectés à un utilisateur. En effet, sur de nombreux serveurs des dossiers communs à un groupe d'utilisateurs sont stockés dans /home, par exemple un dossier compta pour les employés de la compta, un etudes pour ceux du bureau d'études, etc.
31        for rep_connex in `du -sb /home/* | cut -f 2`; do

32    quota=""
33          while [ -z $quota ]; do
34           clear
35           echo -e "\nVeuillez saisir une taille maximum en octets pour le répertoire $rep_connex"
36           read quota
37          done
38
39          echo "$rep_connex:$quota" >> /var/tmp/admin/quota
40        done
```

```
41          status=1;;
42        3)
43          clear
44          status=1
45          while [ -z $quota ]; do
46        esac
47      done
48    # - Vérification ------------------------------------------------
49    elif [ $1 = verif ]; then
50      if [ ! -f /var/tmp/admin/quota ]; then
51        echo "Le fichier de configuration n'a pas été trouvé; veuillez
  lancer la commande avec le paramètre 'modif' pour le générer."
52        exit 1
53      else # traitement des quotas indifférenciés
  # En stockant la liste des utilisateurs dans un fichier, il était possible
  d'éviter d'activer deux pipes quasiment identiques.
54        for rep_connex in `cat /var/tmp/admin/quota | grep -o ".*:" |
  cut -d : -f 1`; do
55          taille_max=`cat /var/tmp/admin/quota | grep "$rep_connex" |
  cut -d : -f 2`
  # Il serait plus judicieux d'utiliser l'option -BMB de du & d'enlever
  ensuite le M final.
56          taille=`du -sb $rep_connex | cut -f 1`
57          if [ $taille_max < $taille ]; then
58            echo "La taille du répertoire $rep_connex a dépassé la limite
  autorisée ( $taille > $taille_max )" | mail -s "Dépassement de quota" root
59          fi # $taille_max>$taille
60        done
61      fi # fin traitement des quotas
62    else
```

```
63      echo "Commande $1 inconnue. Veuillez taper $0 sans
    arguments pour afficher la syntaxe."
64      exit 1
65    fi
66    exit 0
```

Pour information, Linux intègre une excellente gestion des quotas, que l'on active en ajoutant le mot quota dans la ligne de /etc/fstab de la partition concernée & que l'on gère avec la commande quotactl.

Ex. 14

Écrire un script permettant de vérifier la connexion avec la commande ping d'un ensemble d'adresses IP.

Le script devra :

- permettre la saisie des adresses IP devant être vérifiées au niveau connexion ;
- donner l'état connecté ou non de chaque adresse ;
- donner l'adresse MAC des machines connectées ;
- donner le nom (DNS) des machines connectées lorsque celui-ci est défini.

Corrigé 1

```
1   #!/bin/bash
2   # par Laurent VILLETTE
    # Laurent a imbriqué dans son script des if-elif pour traiter les différentes commandes (-a, -s, -l, -v) & leur absence. L'emploi de l'instruction case aurait été préférable, c'est la raison d'être du corrigé 2
3   if [ $# = 0 ]; then # absence de commande
4     echo "Syntaxe : $0 <commande>"
```

```
 5      echo "     -a <ip1[,ip2,...]> : ajoute une ou plusieurs adresses IP
  à la liste"
 6      echo "     -s <ip1[,ip2,...]> : supprime une ou plusieurs adresses
  IP de la liste"
 7      echo "     -l                 : affiche la liste des IP surveillées"
 8      echo "     -v                 : lance une vérification des IP
  renseignées"
 9    elif [ $1 = -a ]; then # ajout d'une adresse
10      for adr_ip in `tr "," " " <<< $2`; do
    # L'explication de l'expression rationnelle suivante se trouve en pages 148 &
    suivantes. L'opérateur <<< a pour effet d'attribuer comme donnée d'entrée
    de la commande qui précède le contenu de l'expression qui suit.
11        grep -q -E '^(([0-9][0-9]?|[0-1][0-9][0-9]|2[0-4][0-9]|25[0-
  5])\.){3}([0-9][0-9]?|[0-1][0-9][0-9]|2[0-4][0-9]|25[0-5])$' <<<
  $adr_ip
12        if [ $? != 0 ]; then
13          echo "$adr_ip n'est pas une adresse IP valide !"
14        else
15          echo $adr_ip >> /var/tmp/liste_ip
16        fi
17      done
18    # --- SUPPRESSION ----------------------------
19    elif [ $1 = -r ]; then
20      for adr_ip in `tr "," " " <<< $2`; do
    # Comme cette ligne est identique à la ligne 11, il aurait était possible
    afin de faciliter la lecture & d'économiser nos forces de créer une
    fonction test_adr_ip contenant cette ligne.
21        grep -q -E '^(([0-9][0-9]?|[0-1][0-9][0-9]|2[0-4][0-9]|25[0-
  5])\.){3}([0-9][0-9]?|[0-1][0-9][0-9]|2[0-4][0-9]|25[0-5])$' <<<
  $adr_ip
```

```
22        if [ $? != 0 ]; then
23          echo "$adr_ip n'est pas une IP valide"
24        else
25          sed -i "/$adr_ip$/d" /var/tmp/liste_ip
26        fi
27      done
28    # --- LISTE ----------------------------------
29    elif [ $1 = -l ]; then
30      echo "Liste des adresses IP surveillées :"
31      less /var/tmp/liste_ip
32
33    # --- Test -----------------------------------
34    elif [ $1 = -c ]; then
35    # Cette façon d'utiliser for n'est pas très lisible, mais elle fait
      partie des techniques d'optimisation. Cependant son emploi avec cat
      n'est pas optimum. La combinaison cat fic | while read vari; do …
      done est moins rapide que la combinaison for vari in `<fic`; do …
      done. Vous pouvez le vérifier avec la commande time.
36      cat /var/tmp/exercice4.cfg | while read adr_ip; do
37        echo "test de $adr_ip..."
38        ping -c1 $adr_ip >> /dev/null
39        if [ $? = 0 ]; then
40          # petite erreur /sbin/arp est préférable à arp, car le dossier
          sbin n'est pas dans la variable PATH d'un utilisateur normal.
41          arp -a $adr_ip | cut -d " " -f 4
42          nslookup $adr_ip | grep -Eo 'name =.*' | cut -d " " -f 3
43        else
44          echo "injoignable"
45        fi
46        echo "-------------------------"
```

```
47      done
48    else
49      echo "Commande $1 inconnue. Veuillez taper $0 sans
   arguments pour afficher la syntaxe."
50    fi
```

CORRIGÉ 2

Cette liste est une autre façon de traiter le problème, en principe plus rapide, car l'instruction **case** est censée s'exécuter plus rapidement que l'imbrication d'instructions **if**, mais cela reste à vérifier.

```
1   #!/bin/bash
2   # par Laurent VILLETTE & Michel SCIFO
3   function test_adr_ip()
4   {
5       grep -q -E '^(([0-1]\d\d?|2[0-4]\d|25[0-5])\.){3@}([0-1]\d\d?|
  2[0-4]\d|25[0-5])$' <<< $1
6       return $?
7   }
8   if [ $# = 0 ]; then
9
10  case $1 in
11    -a) # ajout d'une adresse IP'
12      for adr_ip in `tr "," " " <<< $2`; do
13
14        if `test_adr_ip $adr_ip`; then
15          echo $adr_ip >> /var/tmp/liste_adr_ip
16        else
17          echo "$adr_ip n'est pas une IP valide"
18        fi
```

```
19        done
20      ;;
21      -s) # suppression d'une adresse IP'
22        for adr_ip in `tr "," " " <<< $2`; do
    # Tester qu'une valeur est vrai revient à tester sa valeur : ici, il n'est
    pas utile d'écrire « = 0 » derrière l'appel de la fonction.
23          if `test_adr_ip $adr_ip`; then
24            sed -i "/$adr_ip$/d" /var/tmp/liste_adr_ip
25          else
26            echo "$adr_ip n'est pas une IP valide"
27          fi
28        done
29      ;;
30      -l) # Liste des adresses saisies
31        echo "Liste des adresses IP surveillées :"
32        less /var/tmp/liste_adr_ip
33      ;;
34      -v) # vérification & affichage des données connues
    # Notez l'alimentation de la boucle while par le pipe. Elle est
    équivalente à l'instruction
    for adr_ip in `cat /var/tmp/liste_adr_ip` ou
    for adr_ip in </var/tmp/liste_adr_ip ou encore
    while read adr_ip </var/tmp/liste_adr_ip.
35        cat /var/tmp/liste_adr_ip | while read adr_ip
36        do
37          echo "test de $adr_ip..."
38          ping -cl $adr_ip >> /dev/null
39          if [ $? = 0 ]; then
40            /sbin/arp -a $adr_ip | cut -d " " -f 4
41            nslookup $adr_ip | grep -Eo 'name =.*' | cut -d " " -f 3
```

```
42          else
43            echo "injoignable"
44          fi
45          echo "-------------------------"
46        done
47      ;;
48      *)
49      echo 'Usage : $0 $1 [$2 ...]'
50      echo ' où $1 vaut -a, -s, -l, -v & $2 ... une ou plusieurs adresses
  IP.'
51      echo "   -a <ip1[,ip2,...]> : ajoute une ou plusieurs adresses IP à la
  liste"
52      echo "   -s <ip1[,ip2,...]> : supprime une ou plusieurs adresses
  IP de la liste"
53      echo "   -l                 : affiche la liste des IP surveillées"
54      echo "   -v                 : lance une vérification des IP renseignées"
    # Le symbole « ;; » n'est pas utile avant esac.
55    esac
```

Ex. 15

Écrire un ou plusieurs scripts permettant de consulter le fichier log (/var/log/messages) de manière simple.

Les scripts devront :

- permettre la saisie de mots clés dont on veut les lignes correspondantes (nom du démon concerné par exemple) ;
- permettre de limiter par la date les informations extraites du fichiers ;
- permettre de visualiser les seuls informations nouvelles depuis la dernière consultation ;
- offrir un mécanisme d'effacement du fichier (effacement jusqu'à telle date).

CORRIGÉ 1

Le script de LAURENT traite en un bloc, les quatre sujets. Quatre scripts séparés, réunis dans un cinquième aurait permis une écriture simplifiée de l'ensemble, mais ce n'était pas demandé, dans l'énoncé qu'il a lu !

```
1  #!/bin/bash
2  # par Laurent VILLETTE
3
4  if [ $# = 0 ]; then
5    echo "Syntaxe : $0 [options]"
6    echo "   -f   <argument(s) de recherche>"
7    echo "        Effectue une recherche sur un ou plusieurs mots
   clés dans le journal."
8    echo "   -t   <filtre>"
9    echo "        Effectue un filtrage du journal par date."
10   echo "        Le filtrage par date peut être au format :"
11   echo "        yyyy-mm-dd (date précise)"
12   echo "        yyyy-mm-dd/yyyy-mm-dd (fourchette)"
13   echo "        /yyyy-mm-dd (jusqu'à une date)"
14   echo "        yyyy-mm-dd/ (depuis une date)"
15   echo "   -d   <yyyy-mm-dd>"
16   echo "        Efface le journal jusqu'à la date spécifiée."
17   echo "   -n   Affiche tous les nouveaux messages depuis votre
   dernière consultation effectuée avec cette même option."
18 else
19   cp /var/log/messages /var/tmp/messages.tmp
20 # Donner des noms en français aux variables & aux fonctions que
   l'on définit permet de les distinguer de celles provenant du système.
21   display_output=1; parse_mode=0
22
```

```
23    # ----- traitement des arguments de la commande -----
24      for ((arg_index=1; $arg_index<=$#; arg_index++)); do
25        case ${!arg_index} in
26          -f)
27            parse_mode=1;;
28          -t)
29            parse_mode=2;;
30          -d)
31            parse_mode=3;;
32          -n)
33            if [ ! -f /var/tmp/exercice5.index ]; then
34              tail -n 1 /var/log/messages | head -c 32 >
  /var/tmp/exercice5.index
35              cat /var/log/messages
36            else
37              timestamp=`cat /var/tmp/exercice5.index`
38              current_line=`grep -n $timestamp /var/log/messages | cut
  -d : -f 1`
39              let current_line+=1
40              cat /var/log/messages | sed -n
  "$current_line,9999999999p"
41              tail -n 1 /var/log/messages | head -c 32 >
  /var/tmp/exercice5.index
42              display_output=0
43            fi;;
44          *)
45    # Il est préférable d'écrite les fonctions en début de script, mais si
  c'est moins clair, ce n'est pas faux !
46            function make_dates_list {
47              date1=$1; dates_list=$date1
48              while [ $date1 != $2 ]; do
```

```
49              date1=$(date +%Y-%m-%d -d "$date1 +1 day")
50              dates_list="$dates_list $date1"
51            done
52          }
53          function scan_dates {
54    # Employer comme nom de variable, le nom d'une commande interne ou externe comme 'date', 'case' ou 'test' est une très mauvaise idée, susceptible de provoquer des effets de bord.
55            for date in $date_list; do
56              cat /var/tmp/messages.tmp | grep "^$date" >> /var/tmp/messages.tmp2
57            done
58            mv -f /var/tmp/messages.tmp2 /var/tmp/messages.tmp
59          }
60          case $parse_mode in
61    # ----- recherche de texte -------------------------
62            1)
63              cat /var/tmp/messages.tmp | grep -F ${!arg_index} > /var/tmp/messages.tmp2
64              mv -f /var/tmp/messages.tmp2 /var/tmp/messages.tmp;;
65    # ----- filtrage par date --------------------------
66            2)
67              echo ${!arg_index} | grep -qE '^[0-9]{4}-[0-1][0-9]-[0-3][0-9]$' && date_mode=1
68              echo ${!arg_index} | grep -qE '^[0-9]{4}-[0-1][0-9]-[0-3][0-9]/[0-9]{4}-[0-1][0-9]-[0-3][0-9]$' && date_mode=2
69              echo ${!arg_index} | grep -qE '^[0-9]{4}-[0-1][0-9]-[0-3][0-9]/$' && date_mode=3
70              echo ${!arg_index} | grep -qE '^/[0-9]{4}-[0-1][0-9]-[0-3][0-9]$' && date_mode=4
71              case $date_mode in
```

```
72                # ---- yyyy-mm-dd
73                l)
74                  cat /var/tmp/messages.tmp | grep "^${!arg_index}" >
  /var/tmp/messages.tmp2
75                  mv -f /var/tmp/messages.tmp2
  /var/tmp/messages.tmp;;
76                # ---- yyyy-mm-dd/yyyy-mm-dd
77                2)
78                  make_dates_list `echo ${!arg_index} | grep -oE
  '^[^/]*'` `echo ${!arg_index} | grep -oE '[0-9\-]+$'`
79                  scan_dates $dates_list;;
80                # ---- yyyy-mm-dd/
81                3)
82                  make_dates_list `echo ${!arg_index} | grep -oE
  '^[^/]*'` `date +%Y-%m-%d`
83                  scan_dates $dates_list;;
84                # ---- /yyyy-mm-dd
85                4)
86                  make_dates_list `head -c 10 /var/log/messages`
  `echo ${!arg_index} | grep -oE '[0-9\-]+$'`
87                  scan_dates $dates_list;;
88                *)
89                  echo "Erreur : format de date invalide"
90                  display_output=0
91              esac;;
92    # ----- purge -------------------------------------
93              3)
94                echo ${!arg_index} | grep -qE '^[0-9]{4}-[0-1][0-9]-[0-3]
  [0-9]$'
95                if [ $? = 0 ]; then
```

```
96                make_dates_list `head -c 10 /var/log/messages` ${!arg_index}
97                for date in $dates_list; do
98                  sed -i "/^$date/d" /var/log/messages
99                done
100               echo "Les entrées sélectionnées ont été supprimées du journal."
101             else
102               echo "Erreur : format de date invalide."
103             fi
104             display_output=0;;
105           *)
106             echo "Erreur de syntaxe : option attendue devant l'argument ${!arg_index}"
107             display_output=0
108         esac
109       esac
110     done
111     if [ $display_output = 1 ]; then
112       cat /var/tmp/messages.tmp
113     fi
114   fi
115   rm /var/tmp/messages.tmp2 2> /dev/null
116   rm /var/tmp/messages.tmp 2> /dev/null
```

CORRIGÉ 2

Dans tous ces scripts le premier paramètre est le chemin absolu du fichier dans lequel on recherche des informations.

Aucun effort d'optimisation n'a été effectué : la suppression des emplois de **cat** & l'emploi d'expressions rationnelles idoines avec

grep, **awk** ou **sed** permettrait d'améliorer la rapidité d'exécution, mais nous n'en sommes pas encore là !

Script 1

La lecture du paragraphe commandes internes du mode d'emploi de *bash*, traduit en français & complété d'exemples, nous apprend l'existence d'une commande **shift** qui décale vers la gauche la liste des paramètres $1 à $n. Autrement dit, elle supprime $1 & renomme $2, $1, etc. Sans cette commande on est obligé d'effectuer une itération comme celle indiquée en ligne 23 du script de Laurent.

```
1  # !/bin/bash
2  declare fichier = "" # l'intruction declare permet de donner une valeur initiale sure aux variables.
3  declare date_cour = ""
4  declare date_deb = ""
5  declare date_fin = ""
6  if [ $# = 0 -o $# = 1 ]; then
7     echo "Usage : nom_script nom_fichier nom_démon [...]"
8  else
9     fichier=$1
10    shift
11    while [ ! $1 = "" ]; do
12      echo $1
13      grep -E $1 $fichier
14      echo "----------"
15      # évite l'affichage du message d'erreur après le dernier shift. Il existe des moyens plus élégants d'éviter l'apparition de ce message. À vous de les trouver !
16      shift 2>/dev/null
17    done
18 fi
```

SCRIPT 2

```
#! /bin/bash

function test_date
{
    return `echo $1 | grep -qE '^20[0-9]-(0[1-9]|1[0-2])-(0[1-9]|[1-2][0-9]|3[0-1])$'`
} # fin de la fonction test_date

function affiche_evenmt
{
    rm /var/tmp/extrait_journal
    date_cour=$1
    date_arret=$2
    while [ $date_cour <= $date_arret]; do
      cat $fich | grep $date_cour >> /var/tmp/extrait_journal 2>/dev/null
      date_cour=$(date +%Y-%m-%d -d "$date_cour +1 day")
    done
    less /var/tmp/extrait_journal
} # fin de la fonction affiche_evenmt

fich=$1
# Si tous les fichiers à examiner sont toujours dans /var /log, il est possible de ne demander que la saisie du chemin relatif à partir de ce dossier & de concaténer le chemin relatif avec « /var/log/ » pour obtenir le chemin absolu : $fich="/var/log/"$1.
if ! [ -e $fich ]; then
  echo "Fichier inexistant !"
  exit 2
```

```
25  fi
26  shift # enlève  le nom du fichier de la liste des paramètres
27  case $# in
28   1)
29     if test_date $1 ; then
30       date_deb=$1
31       date_fin=$1
32       affiche_evenmt $date_deb $date_fin
33     else
34        echo "Date incorrecte !"
35        exit 3
36     fi
37    ;;
38   2)
39     if test_date $1 ; then
40       date_deb=$1
41     else
42       echo "Date de début incorrecte !"
43       exit 4
44     fi
45     if test_date $2 then
46      date_fin=$2
47     else
48       echo "Date de fin incorrecte !"
49       exit 5
50     fi
51     affiche_evenmt $date_deb $date_fin
52    ;;
53   *)
54     echo "Usage : $0 nom_fichier date_de_début [date_de_fin]"
```

```
55      exit 1
56    esac
```

SCRIPT 3

```
#! /bin/bash
# Nous supposons que le nom de fichier est correct, cette éventualité est traitée dans le script2.
date_deb=`stat $1 | grep Accès | grep -v UID | cut -d \  -f 3`
date_fin=`date +%Y-%m-%d`
./script2 $1 $date_deb $date_fin
```

SCRIPT 4

```
#! /bin/bash
# Même remarque que pour le script 3
date_deb=$2
date_fin=`date +%Y-%m-%d`
./script2 $1 $date_deb $date_fin
```

FONCTIONS

fonctions.sh contiendra les fonctions **test_date** & **affiche_evenmt**.

En contrepartie, elles seront ôtées du script 2 & remplacées par une ligne . ~/TP/fonctions.sh, ou TP est le dossier dans lequel se trouve les autres scripts & fonctions.sh le nom du script les contenant.

SCRIPT 5

```
# !/bin/bash
. ~/TP/fonctions
option=$1
shift
```

```
case $option in
-d) ./script1 "$@" ;;
-p) ./script2 "$@" ;;
-c) ./script3 "$@" ;;
-e) ./script4 "$@" ;;
*)
    echo "Usage : $0 option paramètres"
    echo "Option vaut :"
    echo "    -d pour afficher les messages relatifs aux démons dont le nom suit ;"
    echo "    -p pour afficher les messages relatifs à une période donnée ;"
    echo "    -c pour afficher tous les messages depuis la dernière consultation ;"
    echo "    -e pour afficher les messages postérieurs à une date donnée."
    echo "Le premier paramètre suivant l'option est le nom du fichier à traiter"
esac
```

Nous pouvons maintenant regrouper ces six fichiers en un seul en transformant chacun des quatre premiers scripts en fonction. En dissociant un problème complexe en problèmes plus simples, nous avons facilité l'écriture du script final.

Annexe 5 Les Licences Creative Commons

D'après http://fr.wikipedia.org/w/index.php?title=Licence_Creative_Commons&redirect=no#CC_Plus, le contenu sous licence Creative Commons peut être utilisé par des tiers sous certaines conditions définies par l'auteur. Toutes les licences comportent la condition Attribution (ou paternité). Trois autres conditions de base peuvent être combinées à celle-ci pour obtenir un total de six licences homologuées par l'organisation Creative Commons.

Une des particularités de ces licences est qu'elles peuvent être représentées par des signes visuels aisément compréhensibles. Il est ainsi possible de savoir exactement ce que permet ou interdit la licence d'un simple coup d'œil.

Paternité, BY

* Nom officiel (anglais) : *Attribution* [*BY*]
* Version courante : 4.0
* (fr) *L'œuvre peut être librement utilisée, à la condition de l'attribuer à l'auteur en citant son nom.*
* (en) *The licensor permits others to copy, distribute, display, and perform the work. In return, licenses must give the original author credit.*
* Toutes les licences Creative Commons comportent cette condition, puisque, dans le cas contraire, il n'y aurait plus d'ayant droit.

Pas d'utilisation commerciale, NC

* NC nc2
* Nom officiel : NonCommercial [NC]
* (fr) *Le titulaire de droits peut autoriser tous les types d'utilisation ou au contraire restreindre aux utilisations non commerciales (les utilisations commerciales restant soumises à son autorisation).*

* (en) *The licensor permits others to copy, distribute, display, and perform the work. In return, licenses may not use the work for commercial purposes -- unless they get the licensor's permission.*

PAS DE TRAVAUX DÉRIVÉS, ND

* Nom officiel : *No Derivative Works* [*ND*]
* (fr) *Le titulaire de droits peut continuer à réserver la faculté de réaliser des œuvres de type dérivées ou au contraire autoriser à l'avance les modifications, traductions...*
* (en) *The licensor permits others to copy, distribute, display and perform only unaltered copies of the work -- not derivative works based on it.*
* Cette licence exclut la licence *Partage à l'identique*

PARTAGE À L'IDENTIQUE, SA

* Nom officiel : *Share Alike 3.0* [*SA*]
* (fr) *Le titulaire a la possibilité d'autoriser à l'avance les modifications ; peut se superposer l'obligation pour les œuvres dites dérivées d'être proposées au public avec les mêmes libertés (sous les mêmes options Creative Commons) que l'œuvre originale.*
* (en) *The licensor permits others to distribute derivative works only under a license identical to the one that governs the licensor's work.*
* Cette licence exclut la licence *Pas de travaux dérivés.*

Remerciements

Ce livre n'aurait pas vu le jour sans les critiques de Guillaume Mièvre sur notre support de cours initial qu'il jugeait trop abscons, sans les scripts écrits par Laurent Villette, dont la correction nous a permis de mieux saisir les difficultés de compréhension de certains points du langage *Bash*.

Il n'aurait pas sa forme actuelle, sans les critiques constructives de notre ami & collègue Dominique Billard, à qui l'on doit la progressivité dans les exercices & en particulier le script de bataille navale. Ses critiques sont à l'origine de la restructuration de l'ouvrage.

Nous tenons à remercier, également notre éditeur, Le Maître Réfleur, pour le soutien sans faille qu'il nous apporte.

Index lexical

A

administrateur.....105, 132, 212, 238
adresse IP.....133 sv, 243, 245 sv
alarme.....132, 238
algorithme5, 14, 60, 108, 133, 137 sv, 141, 145
analyse 3 sv, 9, 12, 14 sv, 17, 66, 100, 119, 122 sv, 131, 135, 142, 196, 216, 231, 268

B

bug.....5

C

café.....6 sv, 11, 31 sv, 205, 268
case. .12 sv, 49 sv, 54, 63, 82, 111, 118 sv, 123, 127, 129, 133, 136, 146, 154, 159, 180, 204 sv, 213, 215 sv, 232 sv, 235, 239, 242, 245, 249 sv, 255, 257, 269, 279
chaîne de caractères.28 sv, 32, 34 sv, 100, 107, 119, 212, 240, 273
chemin.25, 65, 67, 72, 131, 143, 160, 169, 182, 212, 236 sv, 252, 254
colonne.....34, 125, 159, 231
colorisation.....7
commande...3sv, 17, 20, 22sv, 24sv, 28sv, 34, 36sv, 41sv, 44, 56sv, 58sv, 61, 63sv, 65sv, 70, 72sv, 75sv, 81sv, 83, 85, 88sv, 95sv, 100, 104sv, 105, 107sv, 109, 111, 113sv, 119sv, 127, 130sv, 136sv, 145, 150sv, 159sv, 165sv, 169, 171, 180sv, 196, 199, 202sv, 208, 209, 212sv, 215sv, 224, 226, 230sv, 236sv, 241sv, 248, 250, 253, 269, 271sv, 272, 274, 275, 277
conditions.....6, 153, 162, 207, 259
connexion. 20, 53, 64, 66, 68, 70, 88, 113 sv, 130 sv, 236, 238, 242
constante.....7 sv, 14, 17, 125 sv, 129, 229
corrigés.....199 sv, 208

D

date..10, 107 sv, 115, 131, 136 sv, 170, 201 sv, 211, 247 sv
distro.....199
données15 sv, 31, 41, 43 sv, 55, 63, 67 sv, 89, 124 sv, 140, 152, 212, 246, 268, 271, 275
dossier. 20, 27, 29, 41 sv, 52 sv, 57 sv, 60 sv, 76 sv, 88, 90, 117, 130 sv, 140, 152, 169 sv, 182, 212, 236 sv, 240, 244, 254, 256
  sous- .....87, 104 sv
  -s .....65, 87, 104 sv

E

éditeur de texte.....21, 23, 174
effets de bord.....76, 224, 250
énoncés.....188, 200
ergonomique.....24
erreurs.....3, 49, 69, 73, 76
erreurs.....224
événement.....8
expression12, 34, 36, 41, 47, 56, 79 sv, 85 sv, 89, 96, 98, 102 sv, 106 sv, 128, 134 sv, 151, 153, 160 sv, 166, 180 sv, 185 sv, 190 sv, 207, 224, 239 sv
  - rationnelle....80, 100, 104, 134, 137, 156, 160, 188, 190, 192, 194 sv, 239, 243
    - -s 63, 93, 103, 133, 135, 162, 170 sv, 225 sv, 252, 277

-s séquences.....203

F

fichier.10, 20, 24 sv, 36 sv, 41, 47, 53, 57 sv, 60 sv, 64 sv, 69 sv, 76 sv, 81, 87, 90, 95, 117, 130 sv, 134, 136 sv, 139 sv, 145, 150, 156, 159 sv, 164, 169 sv, 208, 236 sv, 241, 247, 252 sv, 255 sv, 272
fonction. 8 sv, 16, 28, 47, 55 sv, 61, 65, 67, 83, 97, 115 sv, 120 sv, 124, 129, 138, 144 sv, 151, 159, 161, 165, 194, 211, 217, 223 sv, 229, 233, 243, 246, 254, 257, 268, 276

I

imbrication.....111, 205, 245
indice.....92, 220, 230, 232
instruction...7, 10, 12, 14, 16, 21, 31, 46 sv, 53, 55, 81 sv, 136 sv, 205, 230, 245 sv, 269
interpréteur20, 24 sv, 27 sv, 42, 63, 65 sv, 68, 77 sv, 83, 87, 91, 97, 131, 150
itération 8, 24, 31, 50 sv, 54 sv, 58 sv, 115, 129, 202, 207, 209, 211, 226, 231, 246, 253

L

langage..5, 7, 9 sv, 14, 29, 47, 53, 63, 75, 81, 104, 110, 120, 127, 139, 147, 149 sv, 152 sv, 199, 202, 225, 261
LibreOffice.....3, 100, 196, 231
licence.....2, 23, 150, 221, 259 sv
ligne...3, 10, 14, 22 sv, 25, 27, 29, 31, 33, 42 sv, 53 sv, 65 sv, 72 sv, 81, 83, 85, 87, 89 sv, 100 sv, 109, 113, 115, 117, 120, 125 sv, 129, 132, 137, 141 sv, 149, 157, 159 sv, 170, 174, 185, 190, 193 sv, 196, 203, 205, 222 sv, 229, 231, 238 sv, 242 sv, 253, 256, 272, 279
limite...51 sv, 63, 103, 114, 132, 183, 186, 192, 207 sv, 238 sv, 241
Linux....3, 20, 22 sv, 34, 77, 96, 115, 117, 150, 154 sv, 159, 169, 242, 281
liste10, 16 sv, 24, 28, 35 sv, 38, 42 sv, 49, 52, 57 sv, 60 sv, 63 sv, 70, 76 sv, 81 sv, 86, 92 sv, 96 sv, 99, 109, 117, 130 sv, 137, 143, 145, 151 sv, 154, 166, 169, 180, 186 sv, 207 sv, 212 sv, 236 sv, 241, 243 sv, 253, 255, 275

log.....136 sv, 247 sv, 251 sv, 254

M

macro.....3, 100, 196, 231
menus.....24, 50
message d'erreur.....18, 29, 66, 253
messages d'erreur.....77, 156, 205
mot réservé...63, 68, 79, 83, 151, 154 sv, 226

N

niveau7 sv, 13, 15, 57, 60, 67, 87, 89, 102, 119, 133, 242, 268, 271, 275

O

objets.....7 sv, 37, 88, 147
opérateur 4, 12, 34, 36, 38, 41, 46, 50, 52, 58, 69 sv, 73 sv, 77 sv, 79 sv, 82, 92, 99, 107, 115, 119, 125, 127 sv, 151 sv, 155 sv, 160 sv, 180, 190 sv, 205, 216, ,224, 238, 243, 272, 276
option. .4, 49, 54, 64 sv, 67 sv, 71, 79 sv, 82, 84, 97, 99, 103, 105, 117 sv, 125 sv, 135, 138, 142, 151, 156, 162 sv, 180 sv, 183, 241, 248, 252, 256 sv

P

paramètre 16, 24, 28, 36 sv, 56, 64, 66 sv, 72, 88 sv, 97, 111, 113, 120, 123 sv, 136 sv, 140 sv, 151, 156, 161, 204, 208, 241, 252, 257, 273
PHP...5, 10, 23, 35, 47, 63, 100, 112, 128, 147, 150, 278
problématique.....229
problème...6, 15, 59, 90, 110, 114 sv, 143, 202, 216, 228 sv, 231 sv, 245, 257, 281

R

recette.....5
redirection. .68, 71, 77, 162 sv, 208, 271 sv
résultat 5, 14 sv, 20, 33, 36, 51, 65, 72, 75, 86, 91, 94 sv, 106 sv, 113, 116, 124, 128 sv, 137, 140, 144 sv, 160, 199, 202 sv, 216, 223 sv, 226 sv
robot.....6 sv, 150

S

script. 3 sv, 10, 12, 21 sv, 24 sv, 28 sv, 31, 42 sv, 47, 50, 52, 54 sv, 58, 61 sv, 75, 90, 100, 104, 105 sv, 110 sv, 122 sv, 130 sv, 136 sv, 147, 161, 165, 167, 202, 204, 211 sv, 215 sv, 221, 224, 230 sv, 236, 238 sv, 242, 248 sv, 253, 256 sv, 261, 268 sv, 271, 276, 279 ; 281
séquence 31, 68, 77, 86 sv, 97, 101 sv, 134 sv, 186 sv, 194, 202, 208
structures de contrôle.....20, 44, 47
syntaxe. .25 sv, 33, 35, 53, 55, 67, 77, 83 sv, 98 sv, 127, 141, 143, 149, 165, 212 sv, 226, 237, 242, 245, 252

T

tableau 12, 15 sv, 18, 22, 34 sv, 60, 66, 75, 80, 91 sv, 119 sv, 125 sv, 150 sv, 162, 165 sv, 180, 183, 212 sv, 218, 220 sv, 222 sv, 225, 231 sv, 272
tâche.....2, 5, 31, 43, 162, 182
temps 25, 66, 68, 131 sv, 145, 154 sv, 226, 281

U

usage...16, 28, 32, 46, 138, 142, 155, 159, 212

V

valeur...7 sv, 14, 18 sv, 32 sv, 40, 42, 45, 47, 49, 51, 53 sv, 66 sv, 76 sv, 85, 88 sv, 93 sv, 97 sv, 101, 109, 114 sv, 120, 123, 132, 134, 138 sv, 142, 150 sv, 155, 157, 162, 165 sv, 180 sv, 185, 211, 219, 224, 226, 233, 237 sv, 246, 253, 272
variable..7 sv, 14, 17 sv, 27, 29, 32 sv, 38, 42, 45 sv, 49, 52, 54 sv, 64 sv, 67 sv, 79 sv, 84, 88 sv, 92 sv, 98, 109, 113, 115, 118, 120, 123, 127, 151 sv, 156 sv, 160 sv, 163, 165, 169, 203 sv, 212, 224, 226, 239, 244, 250, 269
variables 8, 12, 16, 31 sv, 42 sv, 45, 47, 55, 63, 69, 77 sv, 82, 84 sv, 88 sv, 92, 111 sv, 127, 129, 138, 140, 145, 154 sv, 159 sv, 165, 181, 199, 204, 213, 224, 248, 253, 269

Michel SCIFO est né à Arles, il y a plus d'un demi-siècle, mais moins de deux-tiers de siècle ; depuis 1999, il vit, la plupart du temps, en région grenobloise. Petit-fils d'un colporteur italien naturalisé, d'un cantonnier provençal & de leurs épouses respectives, humoriste patenté, économiste de formation, ayant, également étudié les bases du droit, de la sociologie, de la psychologie sociale & de la psychologie du travail, ayant une expérience d'ingénieur-conseil en informatique & en gestion, syndicaliste solidaire & adhérent sporadique d'associations telles Amnesty International ou Slow Food, entre autres, il forme des techniciens supérieurs en réseaux informatiques à l'Association de Formation Professionnelle des Adultes.

Grand lecteur, il se met à écrire en 2001 : d'une part, car, ayant un problème de cordes vocales, rendant pénible les discussions interminables avec ses amis & ses relations, il souhaitait rédiger ses idées pour éviter les redites ; d'autre part, car, ayant un esprit de contradiction exacerbé, défendant, selon les interlocuteurs, tantôt des idées de droite, tantôt des idées de gauche, il éprouvait le besoin de faire le point sur les siennes. De là naquirent deux livres destinés à ses proches, disponibles sur son site Internet **www.scifo.fr**, & une accoutumance à l'écriture. Président de l'ARDEUR (Association pour la Réhabilitation De l'Esperluette Uniformément Répartie), il emploie ce symbole, noté « & » ou « & », pour remplacer la conjonction « et ».

Ce livret est destiné à servir de support complémentaire à des apprentis administrateurs réseaux Linux, devant se colleter avec des scripts d'administration système.

À l'origine, deux articles de Linux Pratique, la réduction de temps consacré à Linux dans la formation Technicien Supérieur en Réseaux Informatiques & de Télécommunication de l'AFPA, celle du temps consacré à la pratique des scripts, de plus en plus demandée en entreprise & la demande de stagiaires d'un support clair & précis.

Toutes les personnes ayant pratiqué Linux pendant plus de 100 heures devraient pouvoir faire les exercices proposés, en y consacrant entre 16 & 40 heures de travail.