Programmation shell

(programmation de scripts en shell bash)

– Travaux Dirigés (**2**) –

Résumé

Le but de ce TD est de vous apprendre à écrire des scripts shell pour automatiser des tâches répétitives (renommage massif de fichiers, création d'une hiérarchie de répertoires, recherche complexe dans un gros ensemble de fichiers, et toute autre tâche automatisable...). Mais aussi de comprendre les limites de ce qu'il est possible de faire en shell.

Table des matières

1 L'environnement shell **2**
1.1 Les variables d'environnement . . . 2
1.2 Personnalisation du prompt . . . 2
1.3 Localisation de l'environnement . . . 3
1.4 Les alias . . . 4
1.5 Les fichiers de configuration du shell . . . 5

2 Les bases du script shell **5**
2.1 Les scripts et le "*sha-bang*" . . . 5
2.2 Les caractères spéciaux . . . 6
2.3 Expansion des variables . . . 6
2.4 Les commandes internes . . . 7
2.5 Les variables tableaux (optionnel) . . . 7

3 Programmation de scripts shell **9**
3.1 Les tests . . . 9
3.2 Les branchements . . . 10
3.3 Les boucles . . . 11
3.4 Les fonctions . . . 12

Avant propos

L'environnement shell propose un langage de programmation complet qui permet surtout d'automatiser de tâches répétitives. Vous pouvez donc disposer de fonctions avancées telles que des branchements, des boucles, des appels de procédures, des interactions avec l'utilisateur, *etc.* Nous allons voir, dans ce TD, comment programmer dans ce langage et aussi comprendre ses limitations.

De plus, il faut signaler qu'il existe quelques documents qui vous permettront d'approfondir le sujet, nous vous conseillons de lire le "*Advanced Bash-Scripting Guide*" [1] de Mendel Cooper, mais aussi le Wikibook "*Bash Shell Scripting*" [2], ou encore le "*Bash Guide*" [3] et bien d'autres encore.

1. Voir : `http://tldp.org/LDP/abs/html/`
2. Voir : `http://en.wikibooks.org/wiki/Bash_Shell_Scripting`
3. Voir : `http://mywiki.wooledge.org/BashGuide`

1 L'environnement shell

L'environnement shell peut se configurer pour répondre à vos propres besoins ou pour s'adapter au système sur lequel vous travaillez (installation locale de logiciels, ajout de commandes personnalisées, *etc*). La plupart du temps, le shell se configure en positionnant des variables. La table 1 résume les façons de connaître et modifier votre environnement.

Fonction	Commande
Accède au contenu de la variable `VAR` (notez le '`$`')	`$VAR`
Affiche l'ensemble de l'environnement	`env`
Positionne la variable `VAR` à `value`	`VAR=value`
Positionne la variable `VAR` à `value` et propage la variable à tous les processus fils	`export VAR=value`
Réinitialise la variable `VAR`	`unset VAR`

Table 1 – Configuration de l'environnement.

Questions

1. Visualisez tout votre environnement actuel.
2. Regardez ce que contient la variable `PWD`.
3. Positionnez la variable `VAR` à '`12345`'. Vérifiez, puis réinitialisez `VAR`.
4. Survolez la page de manuel de `bash`. Vous pourrez toujours y trouver des informations utiles.

1.1 Les variables d'environnement

Quelques variables du shell sont listées dans la table 2. Elles contrôlent différents aspects de votre environnement ou vous permettent de récupérer des informations sur l'état de votre shell (votre position dans le système de fichier, le type de système d'exploitation sur lequel vous êtes, *etc*).

Fonction	Variable
Chemin vers le répertoire utilisateur	`HOME`
Chemins vers les exécutables	`PATH`
Position courante dans le système de fichier	`PWD`
Position précédente dans le système de fichiers	`OLDPWD`
Contient un nombre aléatoire	`RANDOM`
Chemin du shell utilisé	`SHELL`
Profondeur de shells imbriqués	`SHLVL`
Éditeur par défaut à utiliser	`EDITOR`

Table 2 – Les principales variables d'environnement.

1.2 Personnalisation du prompt

Un application de la personnalisation de l'environnement est de se composer un prompt [4] à soi. Il existe quatre types de prompts en shell et quatre variables qui permettent de les configurer (voir table 3).

Le prompt primaire apparaît en permanence dans votre console, le prompt secondaire apparaît lorsque vous introduisez une césure dans la ligne de commande avec le caractères '\'. Nous ne traiterons pas les autres prompts (`PS3` et `PS4`) ici.

Le prompt peut être composé de caractères fixes et ne jamais changer, mais il peut aussi contenir des informations qui vous aident à savoir où vous êtes. Ainsi, il est possible de composer un prompt qui vous donne l'heure, l'endroit où vous êtes, la machine sur laquelle vous êtes, *etc*. Pour cela, il suffit d'utiliser les commandes spéciales listées dans le table 4.

4. Pour ceux qui l'ignorent, le prompt est le petit texte qui précède toujours le curseur sur votre console.

Fonction	Variable
Variable du prompt primaire	`PS1`
Variable du prompt secondaire	`PS2`
Variable du prompt de la commande '`select`'	`PS3`
Variable du prompt de debug via la commande '`set -x`'	`PS4`

Table 3 – Les variables de prompts.

Par exemple, ce prompt donne votre nom de login, le nom de la machine sur laquelle vous travaillez et le répertoire dans lequel vous êtes : `PS1='[\u@\h \W]\$ '`

Fonction	Commande
Date (jour, mois)	`\d`
Date en suivant le format donné	`\D{format}`
Nom de la machine	`\h`
Nom complet de la machine (avec le réseau)	`\H`
Nombre de processus en cours d'exécution	`\j`
Retour à la ligne	`\n`
Retour en début de ligne	`\r`
Nom du shell utilisé	`\s`
Heure courante HH :MM :SS	`\t` (24h) ou `\T` (12h)
Heure courante (24h) HH :MM	`\A`
Nom de l'utilisateur courant	`\u`
Version du shell	`\v`
Chemin vers le répertoire courant	`\w`
Nom du répertoire courant	`\W`
Numéro de la commande dans l'historique	`\!`
Numéro de la commande depuis le démarrage de la console	`\#`
Empêche le caractère '`c`' d'être interprété	`\c`

Table 4 – Les principales variables d'environnement.

Enfin, il est aussi possible de colorer certaines parties de son prompt pour les faire ressortir et rendre le tout plus lisible. Colorer une zone de texte se fait en l'encadrant par le préfixe '`\[x;ym`' (`x` et `y` permettent de définir la couleur voulue, voir le table 5) et de clore la zone avec le postfixe '`\e[m`'. Par exemple, en reprenant l'exemple du prompt donné précédemment, si vous désirez mettre en rouge le nom du répertoire dans lequel vous êtes, il faudra exécuter : `export PS1='[\u@\h \e[0;31m\W\e[m]\$ '`
Si l'on veut qu'en plus le rouge soit plus appuyé, il suffit de le mettre en gras avec la commande suivante : `export PS1='[\u@\h \e[1;31m\W\e[m]\$ '`

Questions

1. Affichez le contenu de votre variable `PS1`. Puis, positionnez `PS1` à '`'[\u@\h \W]\$ '`'.
2. Affichez l'heure en bleu (et en gras) au sein de votre prompt.
3. Provoquez l'apparition du prompt secondaire, puis modifiez le.

1.3 Localisation de l'environnement

Une des première chose que vous pouvez personnaliser est la langue dans laquelle le système s'adresse à vous. Les pages de manuel, les messages d'erreur, *etc*, peuvent être traduits dans la langue que vous préférez. On appelle cela "*localiser l'environnement*". Pour spécifier au système ce que vous voulez, il suffit de positionner correctement les variables indiquées dans la table 6. En pratique, seuls les variables `LANG` et `LC_ALL` sont vraiment utilisée. Mais, si vous voulez positionner plus finement votre environnement, les autres variables peuvent s'avérer aussi utiles.

Décoration	Code
Normal	00
Gras	01
Sombre	02
Clair	03
Souligné	04
Inversé	07
Transparent	08
Barré	09

Couleur	Code	
	Avant-plan	Arrière-plan
Transparent	00	00
Noir	30	40
Rouge	31	41
Vert	32	42
Jaune	33	43
Bleu	34	44
Violet	35	45
Cyan	36	46
Blanc	37	47

Table 5 – Les codes couleurs en shell.

Fonction	Variable
Positionne la langue pour tout ce qui ne fait pas appel aux variables LC_	LANG
Écrase LANG et toutes les autres variables LC_	LC_ALL
Façon de trier les listes (caractères spéciaux)	LC_COLLATE
Façon de traiter l'expansion	LC_CTYPE
Traduction des messages d'erreurs	LC_MESSAGES
Type de format numérique utilisé	LC_NUMERIC

Table 6 – Les variables d'environnement responsables de la localisation.

Le contenu d'une variable de localisation contient trois informations, le code de la langue (`fr`, `en`, `it`, ...), le code du pays (`FR`, `UK`, `US`, ...) et le type d'encodage (`utf8`, `iso8859-1`, `big5`, ...). Le contenu de la variable s'écrit ainsi : `<codelangue>_<CODEPAYS>.<encodage>`

Par exemple : `en_US.utf8`, `fr_BE.iso8859-1`, ...

Questions

1. Regardez le contenu des différentes variables d'environnement de votre session.
2. Exécutez la commande '`locale`' dans votre console, puis la commande '`locale --all-locales`'.
3. Positionnez la variable `LC_ALL` à `en_US.utf8`. Exécutez à nouveau la commande '`locale`', puis regardez le contenu d'une page de manuel, provoquez quelques erreurs sur votre shell (par exemple en appellant une commande qui n'existe pas).
4. Remettez votre environnement à son état de départ.

1.4 Les alias

Parfois, vous auriez envie de pouvoir avoir des raccourcis pour une commande ou un ensemble de commandes que vous tapez souvent. C'est à ça que servent les *alias*. Les alias se définissent via la commande `alias` suivi du nom que l'on veut donner à la commande et d'une définition de ce qu'elle fait.

Attention, il est possible de réécrire par dessus une commande existante, et dans ce cas lorsque vous taperez le nom de la commande c'est l'alias qui sera exécuté. Pour désactiver l'aliasing temporairement, il suffit de préfixer le nom de la commande par '`\`'. Si l'on veut enlever l'alias dans une console, il suffit d'utiliser la commande '`unalias`'.

Questions

1. Regardez la page de manuel de la commande `bash` pour lire la section sur les commandes '`alias`' et '`unalias`'.
2. Tapez la commande '`alias`' (cela affichera la liste des alias déjà défini).
3. Définissez un alias pour '`ls`' qui exécutera la commande '`ls -alh`'. Vérifiez que l'alias est présent en faisant '`alias`'. Essayez la commande '`ls`', puis la commande '`\ls`'. Finalement, faites : `unalias ls`

1.5 Les fichiers de configuration du shell

Il existe deux façons de modifier votre environnement shell. La première consiste à exécuter directement la commande dans une console, ce qui modifiera seulement le shell de votre console et la modification disparaîtra lorsque vous fermerez la console.

L'autre manière consiste à rajouter la modification dans l'un des fichiers de configuration de votre shell. Il existe plusieurs fichiers, chacun est exécuté à des moments précis de vos sessions (ouverture de la session, ouverture d'une console, fermeture de la session, *etc*). La table 7 liste les principaux fichiers de configuration du shell et leurs fonctions.

Notez que pour éviter d'avoir à fermer et relancer votre console, vous pouvez toujours exécuter le contenu d'un fichier de configuration pour bénéficier des modifications que vous y avez fait en faisant : `source <fichier>`

Fonction	Fichier
Exécuté à chaque démarrage d'une console	`.bashrc`
Exécuté à chaque début de session (login)	`.bash_profile`
Exécuté à chaque sortie de console et sortie de session	`.bash_logout`
Historique des dernières commandes	`.bash_history`

Table 7 – Les principaux fichiers de configuration de `bash`.

Questions

1. Lisez tous vos fichiers de configuration shell (`.bashrc`, `.bash_profile`, `.bash_logout`).
2. Ajoutez un alias dans votre `.bashrc` et vérifiez qu'il a bien été pris en compte.
3. Ajoutez un message de bienvenue '`echo 'Bonjour!'`' à la fin de votre `.bashrc`. Vérifiez que le message apparaît à chaque ouverture d'une console. Ajoutez un message aussi dans les autres fichiers de configuration et trouvez à quel moment le message apparaîtra.

2 Les bases du script shell

2.1 Les scripts et le "*sha-bang*"

Même s'il est possible d'écrire ses programmes shell sur une ligne de commande, il est souvent plus pratique de les écrire dans des fichiers pour les exécuter sans avoir à tout réécrire à chaque fois. On appelle ce genre de fichiers des *scripts shell*.

De manière plus générale, un *script* est avant tout un fichier texte, lisible par un humain, qui contient des commandes qui seront exécutées par un intepréteur [5]. Il existe de nombreux langages de scripts comme Perl, Python, Ruby, *etc*. Le shell fait aussi partie de la famille des langages de scripts.

Exécuter un script shell peut se faire de deux manières différentes. La première, la plus simple, consiste à invoquer l'interpréteur de script en lui passant en paramètre le fichier contenant le script. Par exemple : '`/bin/bash myscript.sh`', où '`/bin/bash`' est l'interpréteur et '`myscript.sh`' est le script.

La deuxième méthode est de rendre le script exécutable ('`chmod +x`') et d'indiquer sur la première ligne du script l'interpréteur à utiliser pour l'exécuter. Par convention, on débute la ligne qui pointe l'interpréteur avec ce que l'on appelle communément un "*sha-bang*" [6] ('`#!`').

Par exemple, si l'on écrit un script Python, il faut commencer le fichier par : '`#!/usr/bin/python`'. S'il s'agit d'un script en shell POSIX standard, ce sera : `#!/bin/sh`'. Et, enfin, dans notre cas, si c'est un script avec des commandes spécifiques au `bash`, alors on écrira : `#!/bin/bash`'. Une fois ceci fait, le fichier est capable d'être exécuté automatiquement et il suffit de faire : '`./myscript.sh`' [7].

5. Un *interpréteur* est un programme destiné à lire un fichier contenant des commandes (en général lisibles par un être humain, mais pas forcément) et à exécuter ces commandes sur le système.
6. Ce terme est une contraction de *sharp* ('#', en anglais) et *bang* ('!', en anglais) qui donne *sha-bang* ('`#!`').
7. Notez le "`./`" devant le nom du script qui donne le chemin relatif jusqu'au fichier.

Questions

1. Créez un script shell qui affiche le texte "`Hello World!`".
2. Faites un script qui exécute les commandes suivantes : '`cd ; echo $PWD`'. Que notez-vous de particulier ? Expliquez pourquoi le répertoire courant ne change pas à la fin du script sur votre console.
3. Faites un script qui commence par '`#!/bin/more`' à la place de l'interpréteur habituel. Exécutez le, que se passe-t-il ?
4. Même question avec '`#!/bin/rm`', que se passe-t-il à l'exécution du script ?

2.2 Les caractères spéciaux

Nous avons déjà vu un grand nombre de caractères spéciaux du shell dans le TD précédent, la table 8 présente les principaux caractères spéciaux spécifiques à la programmation shell mais ne se veut pas exhaustif. Le plus important reste le caractère '`#`' qui permet de mettre des commentaires dans le script shell (notez au passage qu'il a un rôle différent du *sha-bang* de la section précédente). Nous verrons par la suite comment utiliser ces caractères spéciaux plus en détails au fur et à mesure des autres sections.

Usage	Caractère(s)
Commentaire (le reste de la ligne est ignorée)	`#`
Pas d'opération (nop)	`:`
Négation d'une expression	`!`
Sépare deux commandes (toujours suivi d'une espace)	`cmd;␣cmd`

Table 8 – Les caractères spéciaux.

2.3 Expansion des variables

Comme le langage shell est essentiellement conçu pour faire de la manipulation de fichiers, les deux principaux types de données sont : les **chaînes de caractères** et, dans une moindre mesure, les **entiers**. Mais, la majeure partie des variables sont considérées comme étant des chaînes de caractères (ou sont implicitement convertibles vers ce type de données). Ce qui fait que la plupart des utilisateurs du shell ne savent pas qu'il peut exister d'autres types de données (que nous verrons plus tard dans ce TD).

Quoiqu'il en soit, la manipulation de ces chaînes de caractères est primordiale, et un certain nombre d'opérateurs existent pour les manipuler au sein des variables qui les contiennent. On parle d'*expansion d'une variable* lorsque l'on évalue son contenu, éventuellement au travers de l'un de ces opérateurs.

La table 9 présente la majeure partie de ces opérateurs d'expansion. À novueau, vous noterez que peu sont consacrés aux entiers et la majeure partie sont plus focalisés sur les chaînes de caractères.

Questions

1. Parmi les commandes suivantes, cochez celles qui utilisent le caractère '`#`' comme un commentaire :
   - ☐ `echo "The # here begins a comment."`
   - ☐ `echo 'The # here begins a comment.'`
   - ☐ `echo The \# here begins a comment.`
   - ☐ `echo The # here begins a comment.`
2. Affichez toutes les lettres minuscules de '`a`' à '`z`' puis tous les nombres de -1 à 100, et enfin toutes les lettres majuscules de '`A`' à '`Z`' suivie des lettres minuscules de '`a`' à '`z`'.
3. Affichez le résultat du calcul suivant : $2 * \texttt{a} + 2$ avec `a` une variable d'environnement qui contient 3.
4. Affichez le caractère unicode 2126.
5. Affichez la chaîne de caractères suivante : `$PATH`
6. Affichez la longueur de la chaîne contenue dans votre variable `PATH`.
7. Affichez les caractères 3 à 7 de votre variable `PATH`.
8. Testez les constructions '`${STR=default}`', '`${STR+default}`' et '`${STR?err_msg}`' en positionnant (ou non) une variable '`var`'.

Type d'expansion	Commande
Expansion d'une liste	`{a..z}`
Expansion arithmétique d'une expression entière	`$((a+2))`
Expansion d'une commande shell	`` `cmd` ``
Expansion d'une commande shell (forme plus moderne de la précédente)	`$(cmd)`
Expansion du code d'un caractère unicode (ici '∅')	`$'\u2205'`
Quotes partielles (évalue les variables quotées)	`"$STR"`
Quotes complètes (n'évalue pas les variables quotées)	`'$STR'`
Expansion de la variable	`${STR}`
Affiche la longueur de la chaîne `STR`	`${#STR}`
Affiche la chaîne `STR` en partant du caractère 3	`${STR:3}`
Affiche 2 caractères de la chaîne `STR` en partant du caractère 3	`${STR:3:2}`
Expansion, mais vaut '`default`' si `STR` n'est pas définie	`${STR=default}`
Expansion, mais vaut '`default`' si `STR` est déjà définie	`${STR+default}`
Expansion, mais s'arrête en affichant '`err_msg`' si `STR` n'est pas définie	`${STR?err_msg}`
Supprime la plus petite expression régulière '`regexp`' du début de `STR`	`${STR#regexp}`
Supprime la plus grande expression régulière '`regexp`' du début de `STR`	`${STR##regexp}`
Supprime la plus petite expression régulière '`regexp`' de la fin de `STR`	`${STR%regexp}`
Supprime la plus grande expression régulière '`regexp`' de la fin de `STR`	`${STR%%regexp}`
Substitue la première occurrence de '`orig`' en '`new`'	`${STR/orig/new}`
Substitue toutes les occurrences de '`orig`' en '`new`'	`${STR//orig/new}`

Table 9 – Les différentes expansions du `bash`.

9. Positionnez une variable `var` à '`fffooo`', que donnent les commandes suivantes :
   a. `echo ${var#f*}`
   b. `echo ${var##f*}`
   c. `echo ${var%o*}`
   d. `echo ${var%%o*}`
10. Affichez votre variable `PATH` en remplaçant le premier '`/`' par un '`%`'. Puis, remplacez les tous.

2.4 Les commandes internes

Une commande interne (ou "*built-in*") du shell est une commande qui est programmée à l'intérieur du shell lui-même. Ceci par opposition aux commandes externes, c'est à dire qui font appel à un programme extérieur au shell. Par exemple, nous avons précédemment vu la commande '`alias`' qui permet de créer des commandes personnalisées. Si vous faites un '`which alias`' ou encore un '`type alias`', vous verrez que cette commande ne correspond à aucun programme sur le système. '`alias`' est donc une commande dites "*interne*" (ou encore "*built-in*") du shell au contraire de commandes telles que '`ls`' ou '`cp`', qui correspondent à de vrais programmes, donc sont des commandes dites "*externes*". La table 10 liste la plupart des commandes internes et leur usage.

Attention: Certaines commandes internes ont le même nom que des commandes externes. La commande '`echo`', par exemple, est à la fois un programme mais aussi une commande interne. Même si les deux commandes sont quasiment identiques, il existe des différences. Notamment, il y a une différence entre les commandes '`/bin/echo --version`' et '`echo --version`'. Dans le premier cas, vous avez exécuté la **commande externe** '`/bin/echo`' alors que, dans le deuxième cas, vous avez exécuté la **commande interne** '`echo`'. Cette ambiguïté est parfois source de bogues dans les programmes.

2.5 Les variables tableaux (optionnel)

Il existe en fait deux types de tableaux en `bash`, le type '`array`' qui associe un index entier à une chaîne de caractères, et le type '`associative array`' qui associe une chaîne de caractère (la *clé*) à une autre chaîne de caractères (la *valeur*).

Usage	Commande
Définis, liste ou efface des alias de commandes	`alias/unalias`
Change de répertoire (*Change Directory*)	`cd`
Définis la valeur ou les attributs d'une variable	`declare`
Affiche la chaîne de caractères passée en argument	`echo`*
Exécute les arguments comme une commande shell	`eval`
Remplace le shell par la commande donnée en argument	`exec`
Force la sortie du shell	`exit`
Positionne l'attribut '`export`' sur les variables shell	`export`
Parse les options et arguments passés en ligne de commande	`getopts`
Fourni une aide en ligne à propos de commandes internes	`help`
Évalue une expression arithmétique	`let`
Affiche ses arguments sous un format donné (voir : '`man strftime`')	`printf`*
Donne le nom et le chemin absolu vers le répertoire courant	`pwd`*
Lis une ligne de `stdin` et la séquence en champs distincts	`read`
Rend une variable shell non-inscriptible (constante)	`readonly`
Définis ou efface une variable shell	`set/unset`
Exécute les commandes contenues dans un fichier	`source`
Test la valeur de vérité d'une expression	`test`*
Retourne toujours vrai/faux	`true`*/`false`*
Donne le type d'une commande	`type`

Table 10 – Les commandes internes du shell ('*' : commandes à la fois internes et externes).

Le type '`array`' sert essentiellement à implémenter les tableaux uni-dimensionnels (vecteur), alors que le type '`associative array`' sert non seulement de table de stockage (dictionnaire) mais aussi de tableau multi-dimensionnel (matrice) en utilisant une clé du type '`i,j,k,...`' qui est en fait une chaîne de caractères. La syntaxe complète pour utiliser pour ces deux types est précisée dans la table 11.

Attention: Les variables de type '`associative array`' ne sont disponibles que dans des versions récentes de `bash` (version 4.0 et au-delà). Il est donc possible que vous ayez à travailler sur des systèmes qui ne disposeront pas de cette option.

Usage	Commande
Déclaration d'un vecteur et de ses éléments	`array=( 1 2 3 a foo )`
Déclaration d'un vecteur avec indices	`array=( [0]=1 [2]=2 [1]=a )`
Déclaration de la cellule d'un vecteur	`array[5]=bar`
Accès à la première cellule d'un vecteur	`${array}` ou `${array[0]}`
Accès à une cellule quelconque d'un vecteur	`${array[2]}`
Déclaration d'un dictionnaire	`declare -A array`
Déclaration d'un dictionnaire avec ses clés	`declare -A array=( [0,0]=1 [aa]=foo )`
Déclaration d'une entrée dans le dictionnaire	`array[aa]=bar`
Accès à une cellule quelconque d'un dictionnaire	`${array[aa]}`
Accès à l'ensemble du vecteur/dictionnaire	`${array[@]}`
Nombre d'éléments du vecteur/dictionnaire	`${#array[@]}`

Table 11 – Les principales commandes sur les tableaux.

Attention: Les cellules vides d'un tableau n'apparaissent pas lors de l'affichage via `${array[@]}`. Cela peut parfois générer une certaine confusion lors du déboguage. Faites donc attention à ne pas laisser de cellules vides en plein milieu de votre tableau.

Questions

1. Déclarez une variable tableau nommée '`array`' qui contient déjà les éléments : '`1`', '`2`', '`a`' et '`foo`'. Affichez combien il y a d'éléments dans ce tableau, affichez l'ensemble du tableau, changez le '`a`' en '`aaa`' puis ajoutez un cinquième élément qui contiendra '`bbb`', finalement affichez à nouveau le nombre d'éléments et le tableau dans son ensemble.
2. En reprenant '`array`' tel quel, affichez le cinquième élément à partir du deuxième caractère.
3. Exécutez la commande suivante et expliquez pourquoi, alors que l'affichage complet du tableau montre bien deux éléments, la variable `array[1]` est une variable vide :
`array=( [0]=foo [3]=bar ); echo ${array[@]}; echo "array[1]=${array[1]}"`
4. Déclarez le tableau suivant '`array1=( )`' puis faites afficher son nombre d'éléments et l'ensemble du tableau. Quelle différence remarquez-vous avec le tableau '`array2=( " )`' ?
5. Déclarez un dictionnaire '`matrix`' de $3 \times 3 \times 3$ qui contiendra la matrice unité.
6. Déclarez un dictionnaire qui traduit les jours de la semaine en français en leur équivalent anglais.

3 Programmation de scripts shell

Les structures de contrôle d'un langage sont toutes les commandes qui permettent de choisir un chemin d'exécution par rapport à un autre, de répéter une action jusqu'à ce qu'un test soit faux, *etc.* Évidemment, il ne peut y avoir de programmation sans les tests, les branchements et les boucles. C'est pour cela que les choses sérieuses ne font que commencer à partir de maintenant.

3.1 Les tests

En programmation, on appelle un *test* quelque chose qui va évaluer la valeur de vérité d'une formule et renvoyer le booléen qui correspond. Le shell `bash` dispose de plusieurs commandes qui permettent d'évaluer la valeur de vérité d'une formule, elles sont toutes listées dans la table 12.

Usage	Commande
Retourne la valeur de vérité de `EXPR`	`test EXPR`
Identique à '`test`' mais avec une syntaxe plus lisible	`[ EXPR ]`
Étends '==' et '!=' sur les chaînes de caractères et les expressions régulières	`[[ EXPR ]]`
Évaluation arithmétique, retourne le résultat de `EXPR==1`	`(( EXPR ))`

Table 12 – Les différentes commandes de test.

Les tests sont très souvent source d'erreurs dans les programmes. Leur syntaxe est extrêmement fragile et nécessite de prêter une attention toute particulière lorsque vous en écrivez. Vous verrez que malgré cet avertissement vous aurez à vous confronter à des erreurs qui viendront de là.

Notez aussi que, de façon surprenante, la commande '`[`' est non seulement une commande interne mais aussi un programme. Il vous suffit de faire un '`which [`' pour vous en rendre compte.

Enfin, voici une petite astuce qui vous permettra de faire les exercices plus facilement, vous pouvez récupérer la valeur d'un test effectué précédemment en faisant '`echo $?`'.

Questions

1. Lisez l'aide ('`help`') des commandes internes suivantes : '`test`', '`[`', '`[[`', '`((`'.
2. Faites un test qui vérifie la présence ou non d'un répertoire '`bar`', puis '`.`' dans le répertoire courant.
3. Lesquelles de ces conditions seront évaluées à '`true`' ?

☐ `[[ 0 ]]`	☐ `[[ xyz ]]`	☐ `(( 0 ))`
☐ `[[ 1 ]]`	☐ `[[ $xyz ]]` (`$xyz` non-définie)	☐ `(( 1 ))`
☐ `[[ -1 ]]`	☐ `[[ -n $xyz ]]` (`$xyz` non-définie)	☐ `(( 5 > 4 ))`
☐ `[[ " ]]`	☐ `! [[ -n $xyz ]]` (`$xyz` non-définie)	☐ `(( 5 - 5 ))`

4. Écrire un test qui vérifie que la variable '`var`' contient bien la chaîne de caractères "`foobar`".

5. Si 'var=17', quels sont les tests qui sont évalués à **vrai** dans la liste suivante (et pourquoi) :

☐ `[[ $var -eq 17 ]]`	☐ `[[ $var -ne 17 ]]`	☐ `[[ $var == 17 ]]`
☐ `[[ $var != 17 ]]`	☐ `[[ $var -gt 16 ]]`	☐ `[[ ! $var -gt 16 ]]`
☐ `[[ $var -ge 17 ]]`	☐ `(( $var > 16 ))`	☐ `(( $var - 17 ))`
☐ `(( $var - 18 ))`	☐ `(( $var >= 17 ))`	☐ `(( $var <= 19 ))`

6. Si 'var=furball', quels sont les tests qui sont évalués à **vrai** dans la liste suivante (et pourquoi) :

☐ `[[ $var == fur* ]]`	☐ `[[ $var == "fur*" ]]`	☐ `[[ $var == 'fur*' ]]`
☐ `[[ $var = "fuzball" ]]`	☐ `[[ $var="fuzball" ]]`	☐ `[[ $var = fuzball ]]`
☐ `[[ $var = furball ]]`	☐ `[[ $var != furball ]]`	☐ `[[ $var != fuzball ]]`
☐ `[[ $var < furball ]]`	☐ `[[ $var < fuzball ]]`	☐ `[ $var < fuzball ]`

3.2 Les branchements

Il existe deux types de branchements, l'un simple, de type `if-then-else` et l'autre multiple de type `switch-case`. Nous présentons des exemples typiques d'utilisation de ces deux constructions pour donner une idée de la syntaxe.

Les exemples présentés dans les listings 1, 2, 3 et 4 montrent à peu près toutes les variations que l'on peut rencontrer autour du `if-then-else`. Ainsi, le listing 1 présente un test classique sans `else`. Le listing 2 présente une imbrication de tests, qu'il vaut mieux remplacer par une unique expression lorsque c'est possible. Enfin, les listings 3 et 4 présentent les syntaxes de tests complets avec le `elif` et le `else`.

```
if [[ $var == 0 ]]; then
  echo "var is equal to zero."
fi
```

Listing 1 – Branchement `if-then`.

```
if [[ $var -gt 0 ]]; then
  if [[ $var -lt 5 ]]; then
    echo "var is between 0 and 5."
  fi
fi
```

Listing 2 – Branchements `if-then` imbriqués.

```
if [[ $var -gt 0 ]] &&
   [[ $var -lt 5 ]]; then
  echo "var is between 0 and 5."
else
  echo "var is outside 0 and 5."
fi
```

Listing 3 – Branchements `if-then-else`.

```
if [[ $var -gt 0 ]] &&
   [[ $var -lt 5 ]]; then
  echo "var is between 0 and 5."
elif [[ $var -lt 0 ]]; then
  echo "var is less than 0."
else
  echo "var is greater than 5."
fi
```

Listing 4 – Branchements `if-then-elif-else`.

Pour finir avec les branchements, `switch-case` est présenté dans le listing 5. Plusieurs choses sont à détailler sur cet extrait de code. Tout d'abord, on notera que chaque liste d'actions liées à un cas doit être terminée soit par '`;;`' (sort du `switch-case`), soit par '`;&`' (exécute les commandes du cas qui suit (*fallthrough*)). Ensuite, il faut remarquer que les wildcards sont utilisables pour spécifier les cas. Cela veut dire que l'on peut grouper des cas, par exemple en écrivant '`2|3`'. Et, cela veut aussi dire que le cas par défaut qui capture tout ce qui n'a pu être matché auparavant est '`*`'.

```
read var # Ask the user through stdin.
case $var in
   1) echo "var is equal to 1";;
 2|3) echo "var is equal to 2 or 3";&
 foo) echo "var is equal to foo";;
   *) echo "var is not equal to 1, 2, 3 or foo";;
esac
```

Listing 5 – Branchements `switch-case`.

Questions

1. Faites un script qui dit si l'argument donné par l'utilisateur (la variable '`$1`') est entre 0 et 5.
2. Faites un script qui demande le prénom de l'utilisateur, puis qui affiche : si le prénom est `Alice`, `Bob` ou `Charles` : `Bonjour!` et sinon : `Vous n'êtes pas Alice, Bob ou Charles.`

3.3 Les boucles

Il existe différents types de boucles en `bash` : les boucles '`foreach`', les boucles '`for`', les boucles '`while`', les boucles '`until`' et, enfin, les boucles '`select`'. Nous allons détailler tout ces types de boucles, expliciter leurs différences et donner des exemples pour les illustrer.

3.3.1 Les boucles 'foreach' et 'for'

La boucle la plus utilisée en shell est, en fait, une boucle '`foreach`'[8], c'est à dire qu'il s'agit d'un itérateur sur une liste d'éléments qui applique des commandes à chacun des éléments de la liste qu'il itère. La syntaxe d'une boucle '`foreach`' en shell est donnée dans les listings 6 et 7.

```
for i in 1 2 3 4 5; do
  echo "Welcome $i times."
done
```

Listing 6 – Boucle `foreach`.

```
for i in 1 2 3 4 5; do
  for j in a b c d ee; do
    echo "Welcome $i$j times."
  done
done
```

Listing 7 – Boucle `foreach` imbriquées.

Les boucles '`for`' classiques existent tout de même, mais ont une syntaxe légérement différente des boucles '`foreach`'. On peut y mettre des conditions d'arrêt quelconques sans itérer sur une liste d'éléments. Les listings 8 et 9 en présentent un exemple d'utilisation.

```
for (( i=1; i <= 5; i++ )); do
    echo -n "$i "
done
```

Listing 8 – Boucle `for`.

```
for (( i=1; i <= 5; i++ )); do
  for (( j=1; j <= 5; j++ )); do
    echo -n "$i,$j "
  done
  echo
done
```

Listing 9 – Boucle `for` imbriquées.

3.3.2 Les boucles 'while' et 'until'

Les boucles '`while`' et '`until`' sont quasiment identiques l'une et l'autre. Cependant, la boucle '`while`' continue à boucler tant que sa condition est vraie alors que la boucle '`until`' s'arrêtera lorsque sa condition sera vraie. Ces deux constructions permettent plus de clarté dans le code en évitant des doubles négations ou des expressions parfois compliquées à évaluer à vue d'œil.

```
i=1 # initialize i
while [[ $i -le 5 ]]; do
  echo "Welcome $n times."
  i=$(( i+1 ))
done
```

Listing 10 – Boucle `while`.

```
i=1  # initialize i
until [[ $i -gt 6 ]]; do
  echo "Welcome $i times."
  i=$(( i+1 ))
done
```

Listing 11 – Boucle `until`.

3.3.3 Les boucles 'select'

Les boucles '`select`' permettent de proposer un menu interactif à l'utilisateur et de lui demander de choisir une option dans ce menu. Combiné avec une utilisation de l'opérateur de branchement '`case`', cela offre la possibilité de gérer des interactions complexes avec l'utilisateur. Le listing 12 présente le code d'un de ces menu, et le listing 13 montre le résultat du programme. Notez aussi que la variable `PS3`,

8. Voir : `http://en.wikipedia.org/wiki/Foreach_loop`

qui est une des variables de prompt, sert à personnaliser le prompt qui apparaît à chaque attente d'une réponse.

```
PS3="Select a day (1-4): "

select i in mon tue wed exit; do
  case $i in
    mon) echo "Monday";;
    tue) echo "Tuesday";;
    wed) echo "Wednesday";;
    exit) exit;;
  esac
done
```

Listing 12 – Boucle `select`.

```
prompt#> ./myselect.sh
1) mon
2) tue
3) wed
4) exit
Select a day (1-4): 1
Monday
Select a day (1-4): 4
```

Listing 13 – Résultat du `select`.

Questions

1. Faites un programme qui crée 20 répertoires appelés `01`, `02`, ..., `20`. Puis qui les peuples avec, chacun, 52 fichiers appelés `aa.tst`, `ab.tst`, ...
2. Renommez tous les fichiers `*.tst` en `*.txt`.

3.3.4 Opérateurs de contrôle des boucles

Certains opérateurs permettent de contrôler la continuation ou la sortie inopinée des boucles. Principalement, il n'y en a que deux types : `break` et `continue`. La table 13 liste leurs usages et syntaxes.

Usage	Commande
Sort de la boucle	`break`
Sort de `n` boucles imbriquées	`break n`
Passe à l'itération suivante	`continue`
Passe à l'itération suivante pour `n` boucles imbriquées	`continue n`

Table 13 – Les différents opérateurs de contrôle sur les boucles.

Questions

1. Écrivez un script qui affiche tous les caractères unicodes de 1000 à 5000.
2. Écrivez un script qui écrit un message sur la console et qui attende que la touche <`q`> soit pressée pour quitter. Si jamais c'est une autre touche qui est appuyée, faites afficher un message d'erreur et attendez à nouveau que la touche <`Enter`> soit pressée.

3.4 Les fonctions

En programmation, les fonctions permettent de décomposer un logiciel en blocs d'instructions plus élémentaires. On utilise les fonctions à la fois pour éviter les répétitions de code, mais aussi pour rendre le programme plus compréhensible et plus modulaire.

En shell, les fonctions sont vraiment basiques, et se rapprochent plus de blocs de code que l'on nomme que de vraies fonctions au sens où l'on l'entends habituellement. En fait, lors de l'exécution, tout se passe comme si le nom de la fonction était remplacé par le bloc de code qui lui correspond. Le listing14 montre comment déclarer et utiliser une fonction sans argument.

```
do_this () {
  echo "This is a function";
}

do_this # call function
```

Listing 14 – Déclaration d'une fonction.

```
do_this () {
  echo "The first argument is $1";
}

do_this "arg1" "arg2" # call function
```

Listing 15 – Fonction avec arguments.

Il faut aussi noter qu'il n'est pas nécessaire de nommer explicitement les arguments de la fonction lors de sa déclaration. Il suffit d'y faire directement référence dans le bloc de programme via les variables spéciales `$1`, `$2`, *etc* qui correspondent respectivement au premier argument, second argument, *etc*. De plus, la variable spéciale `$@` correspond à la liste de l'ensemble des arguments de la fonction et `$0` correspond au nom du fichier contenant le script. Le listing 15 montre comment déclarer et utiliser une fonction avec arguments.

Questions

1. Écrivez une fonction `foo` qui imprime le nom du script, l'ensemble de ses arguments ainsi que ses deux premiers arguments séparément l'un de l'autre. Puis, depuis le corps du script, appelez cette fonction avec zéro, un (`"arg1"`), deux (`"arg1" 0`) et trois arguments (`"arg1" 0 "arg3"`).
2. Écrivez une fonction récursive qui calcule la factorielle de l'entier qui est donné. Renvoyez une erreur si aucun argument n'est donné.
3. En reprenant le script de la fonction factorielle, ajoutez-y une fonction `usage()` qui affiche le message suivant en cas d'erreur :

```
Usage: factorial.sh VALUE
   Compute the factorial of VALUE and display it.
```

4. Écrivez un script avec une fonction qui affiche un triangle de Pascal [9] de hauteur donnée en argument. Voici un exemple de fonctionnement du programme :

```
$> ./triangle.sh 9
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

5. Écrivez un petit jeu de pendu en shell qui interagit avec l'utilisateur. Essayez de le découper en fonctions de base qui permette une certaine modularité.

9. Voir : `http://en.wikipedia.org/wiki/Pascal's_triangle`