Travaux Pratiques
Initiation à la programmation avec le shell Bash
Feuille n.1

1 Utiliser les variables

Les variables sont des boîtes dans lesquelles nous pouvons stocker des **valeurs**. En bash, les valeurs sont des **chaînes de caractères**. Testez dans un terminal la séquence de commandes suivantes :

```
X="un bon TP"
echo "Je vous souhaite ${X}"
```

Transformez ces deux lignes en script (programme) en les plaçant dans un fichier, par exemple `foo.sh`, en ajoutant le shebang[1] `#!/bin/bash` comme première ligne. Vous pouvez utiliser l'éditeur de texte `emacs` pour ce faire. Rendez ensuite le fichier exécutable (`chmod +x foo.sh`) et exécutez-le dans un terminal (`./foo.sh`). Vous etes en ce moment à la fois le **développeur** de ce petit programme (celui qui l'édite) et un **utilisateur** (quelqu'un qui le lance) . Essayez à présent (en tant que développeur) de remplacer la ligne d'affectation par une ligne qui demandera à l'utilisateur (donc à vous même) de saisir une phrase :

```
#!/bin/bash
echo "Je suis un gentil petit programme. Que voulez-vous que je vous souhaite?"
read X
echo "Je vous souhaite ${X}"
```

Essayez de remplacer `${X}` par la syntaxe abrégée `$X` (sans accolades) et testez à nouveau. Rien ne devrait changer à l'exécution. En effet, cette **abréviation** est presque toujours possible, sauf quand le nom de la variable doit être isolé explicitement (avec les accolades) du reste de la chaîne de caractères. Par exemple, si on imagine d'utiliser les tirets bas (underscore) à la place des blancs (cela sera courant lors de la manipulation de noms de fichiers) :

```
echo "Je_vous_souhaite_${X}_et_tout_le_meilleur"    # marche correctement
echo "Je_vous_souhaite_$X_et_tout_le_meilleur"      # ne marche pas bien
```

La seconde forme ne marche pas bien puisque Bash s'imagine que le nom de la variable n'est pas `X` mais `X_et_tout_le_meilleur` (qui est une autre boîte, certes, mais pas celle dont on voulait parler).

2 La programmation est un jeu de Lego

La programmation est un jeu de Lego : vous assemblez les valeurs (`"Je vous souhaite"`, un caractère blanc et le contenu de `X`) pour construire de nouvelles valeurs et vous assemblez des (sous-)programmes (`echo`, `read`) pour construire de nouveaux (sous-)programmes.

Assemblage des *valeurs* en Bash. En Bash, l'assemblage de valeurs est très simple : la plupart du temps il s'agit de chaînes de caractères, qu'on assemble en les "collant" les unes aux autres (**concaténation** ou **juxtaposition**).

1. Wikipedia : Le shebang, représenté par #!, est un en-tête d'un fichier texte qui indique au système d'exploitation que ce fichier n'est pas un fichier binaire mais un script (ensemble de commandes) ; sur la même ligne est précisé l'interpréteur permettant d'exécuter ce script.

Assemblage des *sous-programmes* en Bash. Pourquoi fait on appel à des sous-programmes pour réaliser un programme ? Tout simplement parce que le service que nous devons rendre peut être rendu en faisant appel aux services offerts par des programmes préexistants. Il s'agit d'un simple principe de **sous-traitance**, comme pour les entreprises qui sous-traitent certaines tâches [2].

Par exemple, supposons de vouloir rendre le service suivant : afficher sur le terminal la date de hier, celle d'aujourd'hui et celle de demain. Puisque le programme préexistant `date` permet de faire ces trois choses (il faut juste le savoir !), nous lui sous-traitons ces trois problèmes. Éditez et testez cette solution :

```
#!/bin/bash
date -d yesterday
date -d today
date -d tomorrow
```

Que compose t'on au juste ? Dans le programme précédent nous faisons trois fois appel au programme `date`, pour que chacune de ces instances (processus) dépose une ligne sur le terminal. Il faut savoir que sous Unix, lorsqu'on fait appel à un programme, le processus correspondant hérite les canaux de communication de sortie (standard et erreur) du processus appelant (le "parent"). Donc chaque instance de `date` écrira sa sortie (standard) sur le même canal de notre programme, qui sera par défaut le terminal. Nous aurons donc composé **trois effets sur le canal de sortie standard** (le canal 1).

Remarque En faisant appels à plusieurs sous-traitants nous composons (cumulons) tous les effets produits.

2.1 Révision Unix avant de continuer.

Tout programme devient un processus, et tout processus peut produire une panoplie d'effets, par exemple des modifications du système de fichiers (création de fichiers ou répertoires), ou des communication sur Internet (download ou upload). En particulier, un processus produit durant son exécution deux chaînes de caractères [3] : la **sortie standard** (qu'il dépose sur le canal 1 ou `stdout`, par défaut connecté au terminal) et la **sortie erreur** (qu'il dépose sur le canal 2 ou `stderr`, par défaut lui aussi connecté au terminal). Lorsqu'il se termine, le processus produit aussi un **code de retour** (nombre entier), qui peut être 0 (pas d'erreur à signaler, tout s'est bien passé), ou différent de 0 (je ne me suis pas exécuté comme prévu, une erreur est survenue). Ce code renseigne donc le processus parent sur l'issue du processus lancé (le "fils").

Par exemple, si vous tapez au terminal [4] :

```
% date
mercredi 1 octobre 2014, 18:17:02 (UTC+0200)
```

La ligne affichée sur le terminal est la sortie (standard) de cette commande. Pour connaître en revanche le code de retour, vous devez aussitôt afficher le contenu d'une boîte spéciale appelée "?" (point d'interrogation) qui contient toujours le code de retour de la dernière commande exécutée :

```
% echo $?
0
```

Ici, donc, tout s'est clairement bien passé. En revanche, si vous tapez :

```
% date --option-non-reconnue
date : option non reconnue « --option-non-reconnue »
Saisissez « date --help » pour plus d'informations.
```

Les deux lignes affichée sont la sortie d'erreur qui, elle aussi, s'affiche par défaut sur le terminal (on pourrait la rediriger sur un fichier avec `2>` ou l'ignorer avec `2>/dev/null`). Si on regarde maintenant le code de sortie :

```
% echo $?
1
```

nous pouvons constater que l'exécution s'est mal passée (la sortie erreur nous explique en effet le problème et c'est bien son rôle).

2. Définition de *sous-traiter* depuis `http://www.cnrtl.fr` : céder à un sous-traitant, en totalité ou en partie, une entreprise, une affaire, une fourniture dont on conserve la maîtrise.

3. On suppose que les processus terminent, sinon nous devrions parler de flux (stream), c'est-à-dire des chaînes de caractère potentiellement infinies

4. Le début de la phrase "`% `" est le message d'accueil, ou **prompt**, que vous ne devez pas écrire

2.2 Expérience des deux techniques d'assemblage des sous-traitants

Lorsque vous apprendrez le langage C, vous verrez que le principe de composition des sous-programmes (qu'on appellera **fonctions**) est basé aussi bien sur les valeurs de retour (qui pourront être bien plus complexe qu'un simple entier) que sur les effets produits par les sous-traitants (sur l'état de la mémoire).

En Bash, le principe de composition des sous-programmes se base essentiellement sur les effets produits par les sous-traitants et marginalement sur le code de retour. Parmi les effets possibles, la sortie (standard) des sous-traitants nous intéresse particulièrement. Autrement dit, la plupart du temps, on appelle un sous-programme dans un script pour qu'il "fabrique" une valeur (une chaîne de caractère) intéressante. De façon moins fréquente, dans le cadre d'un `if-then-else`, nous sommes intéressés par le code de retour du sous-programme (test) écrit après le mot `if` : ce code de retour servira justement pour aiguiller l'exécution vers la partie `then` (en cas de succès) ou vers la partie `else` (en cas d'échec).

Composer les sorties. On fait appel à un sous-programme pour qu'il fabrique sa sortie (standard) qu'on relie à la notre, ou qu'on stocke quelque part (dans une variable ou un fichier). Dans l'exemple précédent avec les trois dates, nous avons laissé les sous-traitants écrire leur sortie sur le même canal que notre programme, c'est-à-dire le terminal. L'autre cas de figure est celui de "capturer" la sortie d'un sous-traitant et la stocker quelque part. Pour la stocker dans un fichier, on utilisera la *redirection* de Bash. Essayez cet exemple où la sortie de deux sous-traitants (`echo` et `date`) est stockée dans le même fichier `/tmp/foo` :

```
echo "La sortie que nous avons capturée est :" 1> /tmp/foo   # redirection en création
date 1>> /tmp/foo                                            # redirection en ajout
```

(Le `1` est optionnel). Pour la stocker dans une variable nous utiliserons le mécanisme de *substitution de commande* `$(COMMANDE)`. Essayez :

```
X=$(date)
echo "La sortie que nous avons capturée est : $X"
```

Vous constaterez avoir capturé la sortie standard de la commande `date` et l'avoir placée dans la boîte `X`.

Composer avec le code de retour. On fait appel à un sous-programme pour aiguiller l'exécution. Essayez par exemple (les point-virgules sont équivalents aux retour-chariots) :

```
if date; then echo "OUI"; else echo "NON"; fi
```

La commande `date` affiche un résultat, certes (on pourrait l'ignorer avec `1>/dev/null`), mais ce qui nous intéresse pour la suite c'est son code de retour, qui sera 0 (succès), et ce sera donc au tour de la partie `then` d'être exécutée. Vérifiez que la chaîne `OUI` soit bien affichée. En revanche, si vous essayez :

```
if date --option-non-reconnue; then echo "OUI"; else echo "NON"; fi
```

c'est bien la chaîne `NON` qui doit être affichée, puisque la commande `date` échoue.

3 Exercices avec `zenity`

Le programme `zenity` est un outil permettant d'afficher des boîtes de dialogue depuis la ligne de commande ou à l'intérieur d'un script. La table 1 donne un aperçu de ses possibilités. Essayez les appels suivants en observant chaque fois la sortie et le code de retour (avec `echo $?`). Un échec devrait être signifié par le code de retour 1 lorsque, par exemple, on appuie sur le bouton "Annuler" :

```
zenity --calendar
zenity --file-selection
zenity --entry --title="Département" --text="Veuillez indiquer votre département"
zenity --entry --title="Couleur" --text="Veuillez indiquer la couleur" Bleu Blanc Rouge
zenity --info --text="Vous apprenez à programmer"
zenity --question --text "Voulez-vous continuer?"
```

Pour avoir de l'aide sur un type de dialogue spécifique vous pouvez taper `zenity --help-OPTION` où `OPTION` est l'une des options présentées dans la Table 1, en enlevant les deux premiers tirets de l'option. Par exemple, pour avoir l'aide sur le calendrier et la sélection d'un fichier, vous pouvez taper :

TABLE 1 – Usages possibles de `zenity` (extrait de `http://doc.ubuntu-fr.org/zenity`)

Type de fenêtre	Argument	Description
Calendrier	`--calendar`	Affiche un calendrier
Entrée	`--entry`	Permet la saisie de caractères
Erreur	`--error`	Affiche une erreur à l'écran
Navigateur de fichier	`--file-selection`	Permet la sélection d'un fichier
Info	`--info`	Affiche une information
Liste	`--list`	Affiche une liste
Notification	`--notification`	Afficher une notification dans la zone prévue à cet effet
Progress	`--progress`	Permet de suivre une progression
Question	`--question`	Affiche une question (OUI / NON)
Info texte	`--text-info`	Affiche un texte dans une fenêtre
Avertissement	`--warning`	Afficher un avertissement
Scale	`--scale`	Choisir une valeur numérique à l'aide d'un curseur
Formulaire	`--forms`	Affiche un formulaire (à partir de la version 3.2.0)
Mot de passe	`--password`	Demande un mot de passe (peut être utilisé avec 'sudo -S')

```
zenity --help-calendar
zenity --help-file-selection
```

3.1 Calendrier

Écrire et tester un script Bash qui demande à l'utilisateur une date (par le sous-traitant `zenity`) et l'affiche sur le terminal (par le sous-traitant `echo`) après la phrase

`"Vous avez choisi une date parfaite, c'est-à-dire : "`.

3.2 Entrée

Écrire et tester un script Bash qui demande à l'utilisateur d'entrer son âge (par le sous-traitant `zenity`) et l'affiche sur le terminal (par le sous-traitant `echo`) :

1. `"Que vous êtes vieux !"` si l'âge est supérieur à 13
2. `"Vous êtes trop jeune pour utiliser ce programme !"` sinon

Écrire ensuite une deuxième version du programme où le point 1. est réalisé avec une fenêtre "Info" de `zenity`, et le point 2. avec une fenêtre "Erreur" de `zenity`.

3.3 Question

Réunissez les deux programmes précédents dans un seul programme, de façon que le service du point 3.1 soit rendu en premier lieu. Pour la suite, posez la question à l'utilisateur (par le sous-traitant `zenity`) si continuer ou pas, et le cas échéant, rendez le service du point 3.2, sinon terminez tout simplement l'exécution.

3.4 Conversion d'un fichier image JPEG → PNG

Service à rendre : convertir un fichier image JPEG (avec l'extension `.jpg`) en fichier image PNG (extension `.png`) en réduisant éventuellement la taille à 100%, 50%, 30% ou 10% de l'original (au choix de l'utilisateur).

Réalisation : utilisez `zenity --file-selection` (pour que l'utilisateur choisisse le fichier `.jpg`) et `zenity --entry` (pour l'éventuelle réduction). La conversion avec réduction de l'image se fera facilement grâce au programme `convert`, dont voici un exemple d'appel :

```
convert rose.jpg -resize 50% rose.png
```

Remarque : si le contenu d'une variable `A` se termine par `.jpg`, nous pouvons construire une nouvelle chaîne où `.jpg` est remplacé par `.png` avec l'opération `${A%.jpg}.png` (explication : le `%.jpg` retire du contenu de `A` le suffixe `.jpg`).