Enoncé du TP 7 Système

C. Pain-Barre

INFO - IUT Aix-en-Provence

version du 4/12/2012

ⓘ Démarrer les PC sous Linux. Les exercices du début sont à faire via une connexion SSH (normale) sur allegro.

1 Substitution de commandes et d'expressions arithmétiques

Exercice 1

Étude des substitutions de commandes et d'expressions arithmétiques

1. Se placer dans `tpunix`
2. Exécuter **ls** pour afficher le contenu du répertoire de travail
3. Exécuter la commande :

   ```
   liste="$(ls)"
   ```

   qui affecte à la variable **liste** la chaîne constituée de ce qu'écrit **ls** (les fichiers du répertoire de travail, séparés par un retour à la ligne)

   ⓘ Il n'est pas nécessaire d'encadrer **$(ls)** de guillemets dans la commande ci-dessus. En effet, l'instruction d'affectation/création de variables (*identificateur*=*mot*) est un cas particulier car les substitutions de paramètres, commandes ou expressions arithmétiques présentes dans *mot* ne sont pas sujettes à la décomposition en mots.

4. Afficher le contenu de la variable **liste** en l'encadrant par des guillemets, qui empêchent sa décomposition en mots. Les fichiers doivent ainsi être écrits un par ligne, car les retours à la ligne qui les séparent sont préservés.
5. Afficher le contenu de la variable **liste** sans l'encadrer par des guillemets. La décomposition en mots est alors opérée et **echo** affichera les mots qui en résultent (ses arguments) en les séparant par un espace.
6. Exécuter :

   ```
   shopt -os xtrace
   ```

   ⓘ Cette commande active l'option *xtrace* de bash lui demandant de "tracer" les exécutions de commandes, en les affichant avant de les exécuter. L'affichage est précédé du contenu de **PS4** (par défaut "+␣"). Les commandes imbriquées sont affichées en marquant le niveau d'imbrication avec le premier caractère de **PS4**. Cette option peut permettre de mieux comprendre ce qui se passe.

7. Exécuter à nouveau :
   ```
   liste="$(ls)"
   ```
   et observer l'effet de cette option.
8. Désactiver l'option *xtrace* avec :
   ```
   shopt -ou xtrace
   ```

   > ⓘ Ne pas hésiter à l'activer quand elle peut aider à comprendre certains mécanismes.

9. Créer une variable **xyz** contenant la valeur 100
10. En utilisant **echo** et la substitution d'expressions arithmétiques (mais pas le point-virgule), afficher le message "**Résultat :** " suivi du résultat de la multiplication de **xyz** et de 7
11. Créer un répertoire `essai` et s'y placer
12. Exécuter la commande :
    ```
    touch f1 f2 'avec␣espace'
    ```
    qui crée les 3 fichiers vides `f1`, `f2` et `avec␣espace`
13. Exécuter **ls** pour afficher le contenu du répertoire de travail
14. Exécuter la commande :
    ```
    tls1=($(ls))
    ```
    qui diffère de la commande tapée en 3. Celle-ci crée le tableau **tls1** qui contient un élément par suite de caractères non blancs (mot) écrite par **ls** (à cause de la décomposition en mots) :
    a. afficher le nombre d'éléments de **tls1**. Cela devrait être 4 car **ls** a affiché 4 mots (pour 3 fichiers)
    b. afficher le contenu du 1er élément (indice 0) de **tls1** qui devrait être "`avec`"
15. Exécuter la commande :
    ```
    tls2=(*)
    ```
    En apparence, on devrait obtenir le même résultat qu'en 14 mais en réalité, les noms de fichiers résultant de la substitution des motifs ne sont pas sujets à la décomposition en mots :
    a. afficher le nombre d'éléments du tableau **tls2**. Cela devrait être 3
    b. afficher le contenu (exact) du 1er élément (indice 0) de **tls2**

Exercice 2

Étude de la décomposition en mots

1. Saisir (ou copier/coller) la définition de la fonction suivante :
   ```
   function ckmots {
       echo "$# argument(s) :"
       local t=("${@/#/-->}")
       t=("${t[@]/%/<--\n}")
       echo -ne " ${t[@]}"
   }
   ```
   dont le rôle est d'afficher le nombre de ses arguments, puis ses arguments, à raison d'un par ligne et encadré entre **-->** et **<--**. Elle nous permettra de vérifier le résultat de la décomposition en mots.

2. Tester la fonction **ckmots** en exécutant :

   ```
   ckmots aa bb cc
   ```

   qui devrait afficher :

   ```
   3 argument(s) :
    -->aa<--
    -->bb<--
    -->cc<--
   ```

3. Dans le répertoire `essai` comparer le résultat des commandes suivantes :

   ```
   ckmots *
   ckmots $(ls)
   ckmots "$(ls)"
   ```

   et remarquer que le découpage en mots n'a lieu qu'avec la deuxième commande.

4. Le découpage en mots se base sur **IFS** qui contient par défaut l'espace, la tabulation et le retour à la ligne. Cette variable peut être modifiée pour rendre cette décomposition personnalisable. Taper :

   ```
   OIFS="$IFS"
   ```

   pour sauver la valeur de **IFS** dans la variable **OIFS** puis taper :

   ```
   IFS=$'\t\n'
   ```

   pour recréer **IFS** mais sans l'espace.

5. Taper à nouveau :

   ```
   ckmots $(ls)
   ```

   et observer que l'espace ne sert plus de séparateur dans la décomposition.

6. Taper :

   ```
   IFS="e$IFS"
   ```

   pour ajouter le **e** dans **IFS**, puis à nouveau :

   ```
   ckmots $(ls)
   ```

   et observer le découpage réalisé et notamment que les **e** ont disparu car ont été traités comme séparateurs de mots.

7. Taper :

   ```
   IFS="$OIFS"        ou      IFS=$'␣\t\n'
   ```

   pour remettre **IFS** à sa valeur initiale puis taper :

   ```
   ckmots $PATH
   ```

   qui ne devrait faire apparaître qu'un seul argument

8. Modifier **IFS** pour ne contenir que le deux-points (**:**) puis exécuter à nouveau :

   ```
   ckmots $PATH
   ```

   et observer le découpage réalisé.

9. Créer un tableau **tabpath** où chaque élément contient un chemin du **PATH**. Vérifier en faisant afficher son élément d'indice 1 (normalement `/usr/bin`)

10. Taper :

    ```
    IFS="$OIFS"        ou      IFS=$'␣\t\n'
    ```

    pour remettre **IFS** à sa valeur initiale puis taper :

    ```
    ckmots $((11 * 12))
    ```

    qui ne devrait faire apparaître qu'un seul argument

11. Taper :

    ```
    IFS=3
    ```

    puis à nouveau :

    ```
    ckmots $((11 * 12))
    ```

    afin d'observer l'effet de cette modification.
12. Remettre **IFS** à sa valeur par défaut en tapant :

    ```
    IFS=$' \t\n'
    ```

13. Revenir à `tpunix` et supprimer `essai`

Exercice 3

Utilisation des substitutions de commandes

1. En utilisant la substitution de commandes et la commande **tty**, faire afficher les informations détaillées sur votre terminal
2. En utilisant la substitution de commandes et la commande **tty**, ajouter le droit d'écriture aux membres du groupe et aux autres sur votre terminal
3. Vérifier que les droits ont bien changé
4. Enlever ces droits sur le terminal puis vérifier.
5. En utilisant la commande **type -P**, faire afficher les informations détaillées sur le fichier exécutable correspondant à la commande **passwd**
6. Avec les substitutions de commandes, créer une variable **luid** contenant l'UID de l'utilisateur `cpb`, en l'extrayant du fichier `/etc/passwd`. Vérifier que cette variable contient bien `1002`.
   Aide : l'extraction consiste à filtrer le fichier pour ne retenir que la ligne de l'utilisateur recherché, puis à ne garder que le champ voulu.
7. Avec les substitutions de commandes, créer un **tableau kiwi** qui contient les informations (nom et prix) sur le fruit `kiwi`, extraites du fichier `fruit.price`. Vérifier que ce tableau a bien été créé avec les deux éléments `kiwi` et `3.89`
8. Avec les substitutions de commandes et **sans** utiliser le tableau **kiwi**, créer une variable **kiwiprix** qui contient le prix du fruit `kiwi`, en l'extrayant du fichier `fruit.price`. Vérifier que cette variable contient bien `3.89`.
9. Créer une variable **f** contenant **trucmuche**
10. À l'aide de **mkdir**, de la substitution de commandes, de **echo**, de la variable **f** et de **sed** (pour mettre en majuscules), créer le répertoire `TRUCMUCHE`
11. En une seule commande et en utilisant **find** et **cat**, créer un fichier `TOUT.TXT` contenant la concaténation de tous les fichiers ordinaires d'extension `.txt`, contenus dans l'arborescence de votre répertoire `tp`

2 Valeur de retour, test de condition et suite de commandes

Exercice 4

Utilisation des tests de conditions et de la variable **$?**

1. En utilisant **test**, tester si le répertoire `/tmp` existe. Faire afficher la valeur de **$?** avec **echo** pour voir le résultat du test. Elle devrait valoir 0 (vrai pour le code de retour de **test**)
2. Créer une variable **VAR** contenant la valeur 50.
3. Utiliser le test des expressions arithmétiques (et non pas la commande **test**) pour tester si **VAR** est inférieure à 7. Cette fois, **$?** devrait contenir 1 car c'est faux. Faire afficher sa valeur avec **echo**
4. Faire à nouveau afficher la valeur de **$?** avec **echo**. Cette fois, elle vaut 0. Pourquoi ?
5. Tester à nouveau si **VAR** est inférieure à 7, mais avec la commande **test**. Vous devriez obtenir à nouveau 1, sinon vous avez probablement utilisé une comparaison de chaînes...
6. Utiliser les expressions arithmétiques pour tester si **VAR** est comprise entre 30 et 100 puis faire afficher la valeur de **$?**
7. Tester si la variable **VAR** est inférieure à 40 ou multiple de 10.

Exercice 5

Utilisation des opérateurs **&&** *et* **||** *dans les suites de commandes*

1. Reprendre le dernier test et utiliser l'opérateur **&&** pour afficher un message si le test est vrai
2. Tester si le fichier ordinaire `/tmp` existe et utiliser l'opérateur **||** pour afficher un message si ce n'est pas le cas
3. En combinant le test d'expressions arithmétiques et **test**, tester si **VAR** est multiple de 10 et si vous avez le droit d'exécution sur `/tmp` et afficher un message si c'est le cas.
4. De même, tester si **VAR** est supérieur à 100 ou si vous avez le droit d'exécution sur `/tmp` et afficher un message si c'est le cas.

✍ Il est rarement pertinent de combiner plusieurs commandes avec ces opérateurs. On restreint généralement leur utilisation à la combinaison de deux commandes. Au delà, la suite de commandes devient difficilement lisible, surtout si on mélange les opérateurs, et il vaudrait mieux dans ce cas utiliser des **if**.

3 Premiers scripts

💣 **La première ligne de tous les scripts créés doit être :**

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

De plus, il faut avoir le droit d'exécution sur un script avant de pouvoir l'exécuter.

Exercice 6

Script affichant un message dépendant du nombre de ses arguments

Créer le fichier **`argtest.bash`** qui contient un script **`bash`** qui écrit un message indiquant si son nombre d'arguments est égal à 1, strictement supérieur à 1 ou nul, en utilisant des instructions **if**.
Rappel : le nombre d'arguments (effectifs) d'un script est contenu dans la variable **`$#`**.

Exercice 7

Scripts comparant ses arguments

> ✍ L'option **-q** de **grep** lui demande de ne pas afficher les lignes qui correspondent mais de se terminer avec une valeur de retour à 0 dès qu'il en a trouvé une qui correspond. Si aucune ne correspond, se termine avec une valeur de retour à 1.

Écrire un script **`in.bash`** qui doit prendre 3 arguments numériques et qui renvoie 0 si le deuxième argument est compris entre le premier et le troisième, et 1 sinon. Pour cela, on vous demande de définir et d'utiliser dans le script une fonction **isnum** qui ne prend qu'un seul argument et qui teste s'il est numérique (est composé éventuellement d'un signe suivi de chiffres uniquement) à l'aide de la commande **grep**. Le script doit vérifier qu'il a bien 3 arguments numériques, sinon renvoyer une valeur de retour à 2.

> ⓘ Avant de créer effectivement le script, il peut être plus pratique de tester le fonctionnement de **grep** et/ou de la fonction demandée dans un shell...

4 Personnalisation de l'environnement bash

> ⓘ Fermer la connexion SSH sur allegro. Les exercices qui suivent sont à faire sur le PC.

Les alias et les fonctions écrits au cours du TP précédent ne devraient plus exister. Ils n'ont existé que dans le shell dans lequel ils ont été définis. Ce shell étant terminé, ils ont disparu avec lui. Pour rendre des alias/fonctions **persistants**, il faut placer leur définition dans l'un des fichiers de configuration de bash.

Nous considérerons par la suite que nous utilisons des login-shell dans les terminaux. En démarrant, ces shells exécutent les fichiers `/etc/profile` et `~/.bash_profile`. Notre `~/.bash_profile` contiendra une instruction faisant aussi exécuter `~/.bashrc`.

Le fichier `/etc/profile` n'est pas modifiable par les utilisateurs. Dans les exercices qui suivent, nous personnaliserons l'environnement en modifiant les fichiers `~/.bash_profile` et `~/.bashrc` :

- le fichier **`~/.bash_profile`** servira à personnaliser :
  - ◇ la variable **`PATH`**
  - ◇ le masque de création de fichiers (**umask**)
  - ◇ éventuellement l'autorisation d'écriture sur le terminal (**mesg**)

- le fichier **~/.bashrc** servira à définir :
  - ◇ des **alias** que l'on placera à part dans le fichier `~/.bash_aliases`
  - ◇ des **fonctions** que l'on placera à part dans le fichier `~/.bash_functions`

Exercice 8

Copie des fichiers de personnalisation.

Sur les PC, le répertoire d'accueil des utilisateurs est éphémère. Nous avons dû adapter ces chemins pour exploiter des fichiers stockés dans `net-home`, via l'utilisation de liens symboliques.

1. Sur un terminal du PC, exécuter la commande :

   ```
   ls -la ~
   ```

   qui devrait faire apparaître les liens symboliques (raccourcis) que nous avons pré-créés dans votre répertoire d'accueil :

   - `.bash_aliases` est un lien vers `net-home/.bash_aliases`
   - `.bash_functions` est un lien vers `net-home/.bash_functions`
   - `.bash_logout` est un lien vers `net-home/.bash_logout`
   - `.bash_profile` est un lien vers `net-home/.bash_profile`
   - `.bashrc` est un lien vers `net-home/.bashrc`
2. Tous ces liens sont pour le moment orphelins car aucun de ces fichiers ne devrait encore exister dans votre `net-home`. Pour le vérifier, taper :

   ```
   ls -laL ~
   ```

   où l'option **-L** demande de déréférencer (suivre) les liens symboliques. Ceux orphelins (ou inaccessibles) provoquent l'affichage de messages d'erreur et apparaissent avec des points d'interrogation en guise d'information.
3. Copier dans `net-home`, les 3 fichiers (cachés) `.bash_logout`, `.bash_profile` et `.bashrc` présents dans `/commun/pain-barre/bashskel`. Les liens correspondants deviennent alors valides.

Exercice 9

Étude des fichiers `/etc/profile` *et* `~/.bash_profile`*, puis personnalisation de* `~/.bash_profile`

1. Afficher le fichier `/etc/profile` et observer son contenu. Il contient les premières instructions qu'un login-shell bash exécute en démarrant. Principalement, il définit la variable **PATH** (différente pour `root` et les autres utilisateurs), puis définit la variable **PS1** spécifiant la forme de l'invite de commandes (*prompt*), et définit une valeur par défaut pour **umask**.
2. Afficher le fichier **~/.bash_profile** et observer son contenu. Il contient les instructions qu'un login shell bash exécute **après** celles de `/etc/profile` :
   - les lignes :

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

font exécuter le fichier **~/.bashrc** s'il existe.

> ⓘ Il s'agit d'une instruction conditionnelle **if ... then ... fi** qui exécute l'instruction **. ~/.bashrc** si la condition **[-f ~/.bashrc]** est vraie, où :
>
> ◇ l'instruction **.** *référence* fait exécuter le fichier *référence* par le shell courant. C'est assimilable à une inclusion du fichier *référence* car tout se passe comme si les instructions qu'il contient se trouvaient à la place de cette instruction. Elle s'écrit aussi **source** *référence* car les commandes internes **.** (*point*) et **source** sont synonymes ;
>
> ◇ **[-f** *référence* **]** est une condition vraie si le **fichier ordinaire** *référence* existe.

- les lignes :

```
if [ -d ~/net-home/bin ] ; then
    PATH=~/net-home/bin:"${PATH}"
fi
```

placent en début de la variable **PATH** le chemin **~/net-home/bin** s'il existe (où la condition **[-d** *référence* **]** est vraie si le **répertoire** *référence* existe).

> ⓘ Cela veut dire qu'un utilisateur peut créer un répertoire `bin` dans son répertoire `net-home` et y placer des fichiers exécutables. Leur emplacement sera ajouté au **PATH** : taper leur nom suffira pour les exécuter, comme les autres utilitaires du système.
>
> On notera que le placement de `~/net-home/bin` **en début** du **PATH** rend les utilitaires "privés" de l'utilisateur prioritaires sur ceux du système.

3. Créer le répertoire `~/net-home/bin` puis y copier le fichier `/commun/pain-barre/unix/sorties`
4. Depuis votre répertoire d'accueil, taper :

   ```
   sorties
   ```

   Un message d'erreur devrait s'afficher car la commande **sorties** n'est pas trouvée.
5. L'erreur précédente est normale car le fichier `~/.bash_profile` a été exécuté avant que le répertoire `~/bin` ne soit créé. Il n'a donc pas été intégré au **PATH**. Pour le vérifier, taper la commande :

   ```
   echo $PATH
   ```

   qui ne devrait pas faire apparaître le chemin `~/net-home/bin`.
6. Ouvrir un nouveau terminal sur le PC. Depuis votre répertoire d'accueil, taper à nouveau :

   ```
   sorties
   ```

   qui devrait suffire à l'exécuter. Fermer ce terminal.
7. Enlever `sorties` de `~/net-home/bin` pour le replacer dans `net-home`
8. Se placer dans `net-home`, et taper la commande :

   ```
   sorties
   ```

   qui devrait produire une erreur car `sorties` ne se trouve plus dans un chemin du **PATH**.
9. Pour que la recherche des commandes externes se fasse aussi dans le répertoire de travail, on peut l'ajouter dans le **PATH**, de préférence à la fin. Modifier le fichier `~/.bash_profile` pour ajouter la ligne suivante à la fin du fichier :

   ```
   PATH="$PATH":.
   ```

qui ajoute à la fin du **PATH** le répertoire de travail (répertoire **.**).

10. Ouvrir un nouveau terminal sur le PC. Se placer dans `net-home` puis taper à nouveau :

    sorties

    qui devrait suffire à l'exécuter. Fermer ce terminal.

11. Procéder à des modifications éventuelles de `~/.bash_profile` si vous souhaitez personnaliser le masque (**umask**) et/ou l'autorisation d'écriture sur le terminal [1] (**mesg**).

12. Faire ré-exécuter ce fichier par le shell du terminal courant en tapant :

    source ~/.bash_profile

> ✍ De façon générale, pour que le shell courant prenne en compte les modifications apportées à l'un de ses fichiers de configuration après qu'il ait été démarré, il faut utiliser la commande interne **source** (ou son équivalent **.** (*point*)) avec pour argument la référence du fichier modifié.

13. Se placer ensuite dans `net-home` et taper :

    sorties

    qui devrait l'exécuter.

Exercice 10

Étude du fichier `~/.bashrc` *et création d'alias persistants dans* `~/.bash_aliases`

Nous avons vu précédemment que le fichier `~/.bash_profile` contient une instruction faisant aussi exécuter par les login-shell le fichier `~/.bashrc` (s'il existe). Nous allons étudier puis personnaliser ce fichier. Toujours sur un terminal du PC :

1. Afficher le fichier **~/.bashrc** et observer son contenu. Il contient de nombreuses instructions configurant les bash interactifs, et facilitant leur utilisation. Cela comprend :
   - l'utilisation de l'historique (pour rappeler des commandes déjà tapées) ;
   - une meilleure gestion de la fenêtre terminal avec un prompt éventuellement coloré et une gestion du titre de la fenêtre ;
   - l'utilisation éventuelle de couleurs par la commande **ls** (alors remplacée par un alias **ls** qui active l'option de coloration)
   - des définitions d'alias recommandés (**ll**, **cp**, etc.)
   - des définitions d'alias en commentaires (à décommenter si on le souhaite en supprimant le **#** du début de ligne)
   - l'inclusion (exécution) du fichier **~/.bash_aliases** s'il existe
   - l'inclusion (exécution) du **~/.bash_completion** s'il existe, qui particularise l'usage de la tabulation pour compléter les noms de fichiers en fonction de la commande utilisée.
2. En principe, personne (à part éventuellement les redoublants) ne possède encore de fichier `~/.bash_aliases`. Ce fichier va contenir la définition de vos alias personnels. Taper :

   vi ~/.bash_aliases

   pour le créer [2] (et/ou l'éditer), puis :

1. Ce qui n'a pas beaucoup d'intérêt sur les PC...
2. En réalité, **vi** va suivre le lien symbolique de `~/.bash_aliases` et créer ce fichier dans `net-home`.

(a) ajouter la définition de l'alias **clean** suivant :

```
alias clean='rm -fv *~ .*~ .*.sw?'
```

qui supprime du répertoire de travail les éventuels fichiers de sauvegarde (cachés ou non) de **vi** (se terminant par **~**) ainsi que les fichiers temporaires de **vi** (fichiers cachés se terminant par **.sw** suivi d'une lettre, normalement **p**).

(b) ajouter la définition de l'alias **sl** suivant :

```
alias sl='ls'
```

que certains peuvent apprécier...

(c) ajouter la définition de l'alias **up** suivant :

```
alias up='cd ..'
```

qui fait remonter au répertoire parent.

(d) ajouter la définition d'un alias **ssha** qui ouvre une connexion SSH (normale) sur allegro

(e) sauver le fichier en quittant **vi**

3. Ces alias ne seront définis que dans les prochains shell exécutés sur le PC. Ils ne le sont pas dans le shell courant car il n'a pas exécuté le fichier `~/.bash_aliases` qui n'existait pas encore quand il a démarré. Taper **alias** pour afficher la liste des alias existants (et constater qu'ils n'existent pas).
4. Utiliser **source** ou **.** (*point*) pour que le shell du terminal courant exécute `~/.bash_aliases`
5. Taper à nouveau **alias** pour vérifier que les nouveaux alias ont bien été pris en compte
6. Les tester. Attention : si l'alias **clean** a mal été défini, vous pouvez supprimer d'autres fichiers que ceux attendus !

Exercice 11

Modification de `~/.bashrc` *et création de fonctions persistantes dans* `~/.bash_functions`

1. Éditer à nouveau le fichier `~/.bashrc` avec **vi**
2. Ajouter les instructions incluant le fichier **~/.bash_functions** s'il existe. Pour cela, s'inspirer des instructions incluant le fichier `~/.bash_aliases`
3. Sauver le fichier en quittant **vi**
4. En principe, personne (à part éventuellement les redoublants) ne possède encore de fichier `~/.bash_functions`. Ce fichier va contenir la définition de vos fonctions personnelles. Taper :

```
vi ~/.bash_functions
```

pour le créer (et/ou l'éditer), puis :

(a) ajouter la définition de la fonction **mk** que vous avez définie au TP précédent :

```
function mk {
    g++ -Wall -o "$1" "$1".cxx
}
```

(b) sauver le fichier en quittant **vi**

5. Utiliser **source** ou **.** pour que le shell du terminal courant exécute le fichier `~/.bash_functions`
6. Copier dans `net-home` le fichier `/commun/pain-barre/unix/sorties.cxx`, puis supprimer de `net-home` votre fichier `sorties`

7. Se placer dans `net-home` et utiliser **mk** pour produire l'exécutable `sorties` à partir de `sorties.cxx`
8. Exécuter `sorties`, ce qui devrait afficher les messages habituels

Exercice 12 (en local sur le PC)

Ajout facultatif de divers alias et fonctions

> ⓘ Cet exercice introduit des fonctions et alias supplémentaires utiles mais facultatifs.

1. Inclure dans `~/.bash_functions`, la définition d'une fonction **xvi** qui prend en paramètre la référence d'un fichier ordinaire. **xvi** exécute **en tâche de fond** un **gnome-terminal** (fenêtre terminal), sans menu en haut. Ce **gnome-terminal** ne doit pas exécuter un login-shell (le défaut), mais doit exécuter à la place **vi** (ou **vim**) et éditer le fichier en argument. Ainsi, quand on quittera **vi**, la fenêtre se fermera. La fenêtre doit aussi avoir pour titre le nom du fichier. Pour cela, consulter le manuel de **gnome-terminal** et en étudier les options **--hide-menubar**, **-t** et **-x**.
2. Inclure dans `~/.bash_functions`, la définition de la fonction **titre** suivante :

```
function titre {
    PS1='\[\e]0;'"$1"'\a\]\u@\h:\w\$ '
}
```

   qui prend en paramètre un seul mot et qui modifie le prompt afin de fixer le mot en argument comme titre de la fenêtre terminal courante (qui s'affiche sur la barre du haut).

   > ⓘ En effet, les terminaux admettent des séquences de caractères d'échappement qui permettent de réaliser plusieurs opérations de ce genre, comme changer la position du curseur, effacer l'écran, écrire du texte en couleur ou avec un attribut vidéo (gras, souligné, inversé, etc.). Ces séquences sont généralement communiquées en utilisant la commande **echo**.
   >
   > **Exemples :**
   >
   > `$ echo -ne '\033]0;ceci est un terminal\007'`
   >
   > ➪ fixe le titre de la fenêtre à « *ceci est un terminal* »
   >
   > `$ echo -ne '\033[10;30HYoupi'`
   >
   > ➪ écrit, sans aller à la ligne, le texte « *Youpi* » à la ligne 10 et en colonne 30.
   >
   > `$ echo -e '\033[31mCe texte est en rouge\033[0m'`
   >
   > ➪ écrit le texte « *Ce texte est en rouge* » en rouge puis va à la ligne.

   Dans la fonction **titre**, nous avons exploité le fait que ces séquences puissent être générées par le prompt en utilisant des caractères particuliers. Le titre de la fenêtre est mis à jour chaque fois que le prompt est affiché.
3. Définir l'alias **aliases** qui exécute deux commandes (séparées par un point virgule) :
   - la première lance l'édition du fichier `~/.bash_aliases` avec **vi** ;
   - la seconde, exécutée après la fin de la première, fait prendre en compte les modifications de ce fichier par le shell courant.

Remarquer que le fonctionnement attendu empêche d'utiliser l'alias **xvi**...

4. Définir l'alias **functions** qui a un rôle identique pour `~/.bash_functions`