- Introduction à l'algorithmique et au langage C -

Scripts Bash

TD1$^+$

1re année ESIEA - Semestre 1

L. Beaudoin & R. Erra & A. Gademer & L. Avanthey

2015 - 2016

Avant propos

Dans le TD « Initiation au Shell », nous avons vu qu'il existe une multitude de lignes de commandes pour effectuer des actions systèmes : manipuler des fichiers, gérer les permissions, lancer des applications, etc..

Nous allons maintenant mesurer l'énorme potentiel de cet outil : plutôt que d'écrire les lignes de commandes une par une, pourquoi ne pas les mettre à la suite pour réaliser des tâches plus complexes et les automatiser ? C'est ce que nous allons faire dans ce TD. Ce sera vos premiers ***programmes*** *: nous les appelons des* ***scripts Bash****.*

1 Outils de travail

Pour écrire des programmes (qui ne sont ni plus ni moins que des fichiers textes) nous allons avoir besoin d'un **éditeur de texte**.

Nous vous proposons deux outils au choix :

- L'éditeur de texte `Vim` dans le `Terminal`.
- L'éditeur de texte `Gedit` sous `Gnome`.

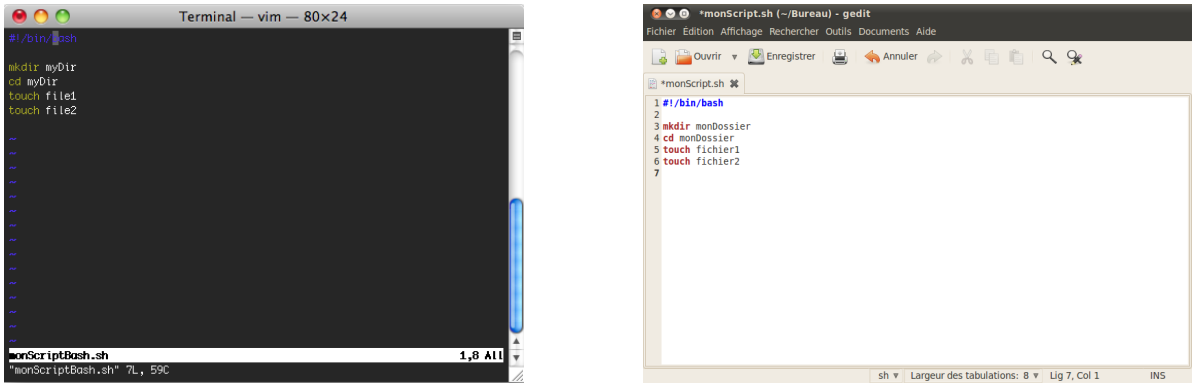

Figure 1 – ***Les logiciels `Vim` et `Gedit` sont des éditeurs de texte avancés avec une coloration syntaxique.***

Attention !
Dans les années supérieures, vous découvrirez ce que nous nommons des Environnements de Développement Intégré (IDE pour Integrated Development Environment) comme **Netbeans**, **Éclipse**, **CodeBlocks** ou **Xcode**.

Ces environnements de travail permettent de travailler sur de très gros projets, de manière collaborative et avec des outils très puissants de manipulation de code.

Leur inconvénient majeur est qu'ils nécessitent une pléthore de fichiers auxiliaires (fichiers de projet), qui sont totalement superflus pour le moment et risque de vous embrouiller. Par ailleurs, en masquant les étapes de compilation derrière un clic bouton, **ils vous empêchent d'apprendre correctement votre cours**.

Pour bien travailler
`Microsoft Windows` n'est pas un environnement de travail propice pour apprendre la programmation. En effet, il n'intègre pas par défaut les outils de compilation et nécessite l'usage des environnements de développements pré-cités.

De plus, le comportement de vos programmes peut différer selon le système d'exploitation et le compilateur installé, ce qui pose des problèmes de compatibilité lors des rendus. Si `Mac OS X`, en tant que système UNIX pose moins de problèmes que `Microsoft Windows`, il n'est pas non plus totalement compatible avec `GNU/Linux`. **Vous ferez donc toujours attention à tester vos programmes sur des machines de l'École avant de les soumettre à vos professeurs !**

Pour travailler chez vous « comme à l'école »
Nous avons mis à votre disposition une machine virtuelle Ubuntu qui reprend les mêmes paramètres que les machine de l'école. Cela vous permet de « faire tourner » un OS `GNU/Linux` dans une fenêtre, que vous soyez sous `Microsoft Windows`, `Mac OS-X` ou `GNU/Linux`. Pour savoir l'installer et l'utiliser, consultez la fiche annexe que nous mettons à votre disposition.

1.1 Vim

C'est un éditeur de texte très pratique qui se lance dans le terminal. Nous l'utilisons très souvent pour éditer et modifier des fichiers ou lorsque nous n'avons pas de serveur graphique.

Vous pouvez le lancer dans le terminal en lui indiquant directement le nom du ficher à éditer et si ce dernier n'existe pas, il sera automatiquement créé :

```
vim monFichier
```

EXERCICE 1

Lancer la commande ci-dessus.

L'interface de `Vim` est assez inhabituelle mais redoutablement efficace une fois prise en main. **Toutes les opérations s'effectuent au clavier.** En particulier, **oubliez les menus** « Fichier », « Édition », etc., qui appartiennent au `Terminal` et non à `Vim`.

Un éditeur modal

`Vim` possède de nombreux modes mais trois d'entre eux sont essentiels :
- le mode **Normal** (la ligne du bas est vide)
- le mode **Insertion** (la ligne du bas indique `-- INSERTION --`)
- le mode **Ligne de commande** (la ligne du bas commence par `":"`)

Le mode Normal : `Vim` démarre dans ce mode. Il est alors possible, entre autres, de manipuler le contenu du fichier grâce à des raccourcis (copier, couper, coller, suppression de ligne, annulation, indentation automatique, etc.). Ce mode, appelé aussi *mode Commande*, est le mode central : il permet d'accéder à tous les autres modes.

Le mode Insertion : C'est le mode le plus intuitif pour vous car il se comporte de la même manière que les éditeurs de texte que vous connaissez. Nous activons ce mode depuis le mode normal avec la touche `i` (pour « Insertion ») ou `INSERT` et nous en ressortons avec la touche `Echap`. **C'est dans ce mode que se fait la saisie du texte.**

Astuce

En mode Insertion, si vous êtes dans le `Terminal` de `Gnome`, le copier/coller à la souris fonctionne. Ce qui est bien pratique pour importer ou exporter du texte depuis ou vers une autre application.

Le mode Ligne de commande : Nous entrons dans ce mode depuis le mode normal en tapant deux-points « : » suivi de la commande que nous voulons exécuter. Vim revient automatiquement en mode normal après l'exécution. Deux commandes importantes : `:w` (sauvegarder), `:q` (quitter).

Vim *HowTo*

Retrouvez les différentes commandes fondamentales et les raccourcis de VIM dans la fiche récapitulative que nous mettons à votre disposition.

1.2 Gedit

Vous le trouverez dans le `Menu Applications > Accessoires > Éditeur de texte Gedit`.

C'est un éditeur de texte avancé comme vous en avez l'habitude, avec une coloration syntaxique (dès que vous aurez nommé votre fichier) et d'autres fonctionnalités comme la numérotation des lignes, la gestion de l'indentation ou le terminal intégré. N'hésitez pas à aller voir dans le `Menu Édition > Préférence > Onglet Greffons`.

Quel éditeur ?

Pour écrire les scripts `Bash` de ce TP, nous utiliserons de préférence `Vim`. Vous pourrez utiliser `Gedit` plus tard.

2 Écrire des scripts

Sur un ordinateur, nous devons souvent réaliser des tâches récurrentes, par exemple :
- renommer un groupe de photos, les réduire, faire une archive et l'envoyer par mail,
- supprimer tous les fichiers temporaires d'un dossier et des sous-dossier pour faire de la place,
- se connecter à un parc de machine, les mettre à jour, puis produire un rapport d'activité,
- récupérer des images satellites, leur appliquer un traitement, puis générer une vidéos de la séquence,
- etc..

Or ces répétitions manuelles sont pénibles et fastidieuses... mais surtout très faciles à éviter !

La description même du problème décrit **un algorithme** : nous allons donc créer des programmes de commandes `Shell` appelés scripts `Bash`.

2.1 Shell is born again !

Depuis les années 70 et la création du **Bourne Shell** par Steve Bourne, de nombreux langages de commandes `Shell` ont été inventés : `csh` (C SHell), `ksh` (Korn SHell) et surtout `bash` (Bourne-Again SHell). Développé depuis la fin des années 80 par la *Free Software Foundation (FSF)*, `Bash` est l'**interpréteur de commandes** le plus répandu sur les machines UNIX (dont `GNU/Linux` et `Mac OS-X`).

Le saviez-vous ?
La *Free Software Foundation* a été fondée par Richard Stallman [1] en 1985 et a pour mission la promotion du logiciel libre.

2.2 Let's Bash something

Les scripts `Bash` sont des **fichiers textes** qui commencent toujours par la ligne :

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

Remarque
Il est d'usage d'utiliser l'extension « `.sh` » pour les scripts `Shell`, `Bash`, etc., mais ce n'est pas obligatoire.

EXERCICE 2

Ouvrez un terminal et déplacez vous sur le Bureau. Créez un répertoire « Scripts » et déplacez vous dedans. Lancez maintenant l'éditeur `Vim` (`vim myScript.sh`) et recopiez-y la première ligne (touche `i` pour passer en mode insertion, `echap` pour revenir en mode normal). N'oubliez pas de sauvegarder (`:w`).

Astuce
Le package `nautilus-open-terminal` est installé sur vos machine. Il vous permet d'ouvrir un `Terminal` directement dans le répertoire que vous visualiser dans `Nautilus` avec `Clic droit > Ouvrir un Terminal`.

1. Vous ne connaissez pas Richard Stallman ? Allez voir sa biographie !

Les scripts regroupent plusieurs commandes `Shell` qui seront exécutées les unes à la suite des autres. Prenons un exemple simple avec les commandes de base que nous avons apprises dans le TD « *Initiation au Shell* ».

```
#!/bin/bash
# This is my first Bash script!
mkdir myDirectory
cd myDirectory
touch file1
touch file2
```

Les commentaires

Le caractère `#` marque les lignes de commentaires. Tout ce qui est écrit après `#` ne sera pas considéré comme une commande, ce qui nous permet d'écrire des commentaires explicatifs !
Attention, la première ligne, elle, n'est pas pour autant facultative !

EXERCICE 3

Recopiez ces autres lignes puis sauvegardez et quittez l'éditeur (`:wq`).

Bravo ! Vous venez d'écrire votre premier script `Bash`. Il nous reste maintenant à le lancer pour que toutes les commandes que nous avons mis dedans s'exécutent. Pour cela il faut écrire la ligne suivante dans le terminal :

```
bash myScript.sh
```

EXERCICE 4

Exécutez votre script Bash grâce à la ligne ci-dessus. Vérifiez ensuite que les commandes ont bien été exécutées en affichant les contenus des répertoires.

QUESTION 1

Que se passe-t-il si vous relancez le script ? Pourquoi ?

Tout cela manque un peu d'interaction, voyons quelques nouvelles commandes ! Les commandes marquées d'une étoile * sont **à retenir absolument**).

TABLE 1: Commandes `Shell` d'interaction.

Nom	Commande	Effet
ECHO	`echo "TEXTE"*`	Renvoie TEXTE sur le flux de sortie.
	`echo "TEXTE" >2`	Renvoie TEXTE sur le flux d'erreur.
SUBST	`$( CMD )`	Exécute la commande CMD et substitue l'expression par le résultat de la commande.

EXERCICE 5

 Créez un nouveau fichier (`vim hello.sh`). Écrivez-y un script `Bash` qui affiche « *Hello* » lorsque nous l'exécutons. Testez-le (`bash hello.sh`).

EXERCICE 6

 Créez un nouveau fichier (`vim smarthello.sh`). Écrivez-y un script `Bash` qui affiche « *Hello, it's HHhMM.* » où HHhMM est remplacé par l'heure actuelle. Testez-le (`bash smarthello.sh`).

Indice

Utilisez SUBST et une commande vue durant le TD « *Initiation au Shell* ».

EXERCICE 7 (*Défis*[2])

 Écrivez un nouveau script `Bash` qui affiche le jour de dernière modification du fichier le plus récent du répertoire, sous la forme : « *The most recent file has been modified on 2011-09-03.* »

Indices

Indice 1 : certaines options de `ls` permettent de trier l'ordre d'apparition des fichiers.
Indice 2 : il faut chaîner de nombreuses commandes de manipulation de flux pour récupérer la précieuse information.

3 Retrouvons l'algorithmique !

Souvenez-vous

Comme nous l'avons déjà vu lors du TD « *Déroulement d'un algorithme* », les algorithmes sont basés sur quatre piliers fondamentaux : la séquentialité, la mémoire, les embranchements et les boucles.

3.1 Séquentialité

Chaque étape est exécutée l'une après l'autre. Nous avons pu le constater en exécutant nos scripts `Bash`. **Ce sera vrai pour tous les programmes informatiques**, bien qu'à un niveau avancé nous pourrons faire semblant de travailler simultanément.

En conséquence de quoi, l'ordre d'écriture du programme est essentiel !

3.2 Mémoire

Dans le TD « *Déroulement d'un algorithme* », nous avions vu que nous avions parfois besoin de sauvegarder des valeurs. Nous utilisions pour cela l'instruction spéciale SAUVER. En Bash, pour stocker de l'information et la récupérer ultérieurement, nous utilisons l'opérateur « = ».

2. Les questions Défis sont des questions optionnelles que vous pouvez faire à la maison pour approfondir le TP.

```
#!/bin/bash
myName="Bryan"
age=4
place="kitchen"
echo "Hello my name is $myName."
echo "I'm $age years old."
echo "And please, I'm not in the $place."
```

Syntaxe

Attention, ne mettez pas d'espace de part et d'autre du signe « = ».

QUESTION 2

Recopiez et exécutez le code précédent. Que s'affiche-t-il ? Qu'en déduisez-vous ?

Voyons une autre commande utile :

TABLE 2: Commandes `Shell` d'interaction.

Nom	Commande	Effet
READ*	`read VAR`	Récupère la prochaine chaîne de caractères du flux d'entrée (clavier par défaut) et la sauve dans VAR.

EXERCICE 8

Écrivez un script `Bash` qui demande à l'utilisateur son nom et le récupère depuis l'entrée clavier (avec `read`). Puis affichez « *Hello NAME !* » où `NAME` est le nom entré par l'utilisateur.

EXERCICE 9

Écrivez un script `Bash` qui indique à l'utilisateur le chemin absolu du dossier où il se trouve (avec `pwd` et `SUBST`) : « *Hello ! You are here : PATH* » où `PATH` est le chemin absolu du répertoire courant.

3.3 Embranchements

Les embranchements, comme nous l'avons vu dans le TD « *Déroulement d'un algorithme* » permettent de faire des choix. En `Bash`, l'embranchement SI-ALORS s'écrit sous la forme :

```
if TEST; then
   # Actions if true
fi
```

Et l'embranchement SI-SINON :

```
if TEST; then
   # Actions if true
else
   # Actions if false
fi
```

3.3.1 Tests numériques

TEST devra être remplacé par une expression booléenne, c'est-à-dire qui peut être soit vraie, soit fausse. Voici une série d'opérateurs pour effectuer des tests sur des expressions arithmétiques. Ceux marqués d'une étoile * sont **à retenir absolument**.

Pour information

Nous retrouverons les opérateurs de test marqués d'une étoile * dans beaucoup de langages, dont le langage C !

TABLE 3: Opérateur de test arithmétiques.

Nom	Opérateur	Effet
INFERIEUR STRICT *	`<`	Teste si une valeur numérique est strictement inférieure à une autre
INFERIEUR OU EGAL *	`<=`	Teste si une valeur numérique est inférieure ou égale à une autre
SUPERIEUR STRICT *	`>`	Teste si une valeur numérique est strictement supérieure à une autre
SUPERIEUR OU EGAL *	`>=`	Tests de comparaison numérique
EGALITE *	`A == B`	Test si deux valeurs numérique sont égales
DIFFERENCE *	`A != B`	Test si deux valeurs numérique sont différentes
EVAL	`(( EXPRESSION ))`	Évalue EXPRESSION numérique ou booléenne.

Pour effectuer un test sur deux valeurs numériques nous utiliserons donc la syntaxe suivante :

```
number1=3
number2=4
if (( $number1 < $number2 )); then
   echo "Success"
else
   echo "Failure"
fi
```

Syntaxe

Attention, mes espaces autour des parenthèses du TEST doivent être scrupuleusement respectées !

EXERCICE 10

Écrivez un script dans lequel vous sauvegardez un chiffre secret (par exemple 5). Demandez à l'utilisateur un nombre. Si ce nombre est égal à votre chiffre secret, affichez : « *Congratulation ! You won, the secret number was NUMBER.* », où NUMBER est remplacé par le nombre secret. Si en revanche, le nombre entré par l'utilisateur est plus petit, affichez : « *You loose, it was greater... Try again !* ». Et s'il est plus grand : « *You loose, it was lower... Try again !* ».

3.3.2 Autres tests

Et voici une autre série d'opérateurs qui pourront nous être utiles pour des tests sur des données non numériques (chaînes de caractères en l'occurence). Les opérateurs marqués d'une étoile * sont **à retenir absolument**.

TABLE 4: Opérateur de test sur chaînes de caractères.

Nom	Opérateur	Effet
EGALITE	`NAME == NAME`*	Test l'égalité de deux chaînes de caractères
DIFFERENCE	`NAME != NAME`*	Test la différence de deux chaînes de caractères
EXISTENCE	`-e NAME`	Est-ce que NAME existe ?
DROIT LECTURE	`-r NAME`	Est-ce que NAME est accessible en lecture ?
DROIT ECRITURE	`-w NAME`	Est-ce que NAME est accessible en écriture ?
DROIT EXECUTION	`-x NAME`	Est-ce que NAME accessible à l'exécution ?
EXISTANCE FICHIER	`-f NAME`	Est-ce que NAME est un fichier ?
EXISTANCE REPERTOIRE	`-d NAME`	Est-ce que NAME est un répertoire ?
EVAL	`[[ EXPRESSION ]]`	Évalue EXPRESSION booléenne.

Pour effectuer un test sur deux chaînes de caractères, nous utiliserons donc la syntaxe :

```
word1="toto"
word2="tata"

if [[ $word1 == $word2 ]]; then
   echo "Success"
else
   echo "Failure"
fi

if [[ -d $word1 ]]; then
   echo "$word1 is a directory"
else
   echo "$word2 doesn't exist or isn't a directory"
fi
```

Syntaxe

Attention, les espaces autour des crochets du TEST doivent être scrupuleusement respectées !

EXERCICE 11

Écrivez un script qui demande à l'utilisateur d'entrer son nom au clavier et stocke la réponse dans `name`. Si le nom est « Bryan » alors le script affiche « *Hi Bryan, how are you today ?* ». Sinon il affiche « *Welcome NAME, I'm Terminal. It's a pleasure to meet you !* » où NAME est le nom entré par l'utilisateur.

EXERCICE 12

Écrivez un script qui :
- demande un nom de répertoire à l'utilisateur et stocke la réponse dans `dirName`,
- teste l'existence du répertoire en question,
- si ce répertoire existe, il écrit « `The directory DIRNAME allready exists!` » où DIRNAME est le nom donné par l'utilisateur, sinon il le crée.

3.3.3 Une touche de logique !

Pour combiner les tests, nous pouvons utiliser d'autres opérateurs appelés opérateurs de logique.

TABLE 5: Opérateur de logiques.

Opérateur	Description
`EXPR1 && EXPR2`*	ET Logique
`EXPR2 || EXPR3`*	OU Logique
`! EXPR`*	NON Logique

Pour rappel, voici les tables de logique (n'oubliez pas, 1 veut dire vrai et 0 veut dire faux) :

Table ET

a	b	a ET b
0	0	0
0	1	0
1	0	0
1	1	1

Table OU

a	b	a OU b
0	0	0
0	1	1
1	0	1
1	1	1

Table NON

a	NON a
0	1
1	0

Exemple :

```
word1="toto"
word2="tata"
word3="titi"

if [[ $word1 == $word2 && $word1 == $word3 ]]; then
   echo "$word1 = $word2 = $word3"
else
   echo "The three words aren't equal."
fi
```

3.4 Boucles

Passons à la deuxième structure de contrôles que nous connaissons : les boucles.

3.4.1 Boucles événementielles

La boucle **TANT-QUE** est de type événementielle, c'est-à-dire qu'elle s'arrête quand un événement survient. Nous ne savons donc pas combien de fois nous allons boucler. Elle s'écrit ainsi :

```
while TEST; do
   # Actions to repeat
done
```

Les actions seront répétées tant que TEST sera vrai. Nous utiliserons les mêmes syntaxes pour le test que celles que nous avons vues pour les embranchements.

EXERCICE 13

 Reprenons l'exercice avec le nombre secret, et modifiez-le pour redemander un nombre à l'utilisateur tant que celui-ci n'a pas trouvé le bon nombre. Laissez-lui les messages d'indications qui lui permettent de savoir si le nombre est plus grand ou plus petit.

Boucle infinie !

Un programme bouclera à l'infini si le résultat du test de sa boucle est toujours vrai (1). Pour arrêtez le programme, utilisez la combinaison de touche `CTRL+C`.

Exemple de boucle infinie :

```
#!/bin/bash
while (( 1 )); do
   echo "I'll not stop!"
done
```

EXERCICE 14

 Testez ce script.

Pour mieux se rendre compte, nous pouvons ajouter un compteur qui nous montrera que nous continuons à boucler :

```
#!/bin/bash

cmp=0
while (( 1 )); do
    echo "$cmp I'll not stop!"
    cmp=$(( cmp + 1 ))
done
```

EXERCICE 15

 Testez ce script.

3.4.2 Boucles itératives

Une deuxième catégorie concerne les boucles itératives : celles où nous connaissons le nombre de fois où nous bouclons et dans lesquelles nous utilisons un compteur. Nous avions vu que la boucle POUR appartenait à ce type. Voici comme elle s'écrit :

```
for VAR in LISTE; do
   # Actions to repeat
done
```

VAR prendra successivement chacune des valeurs de la liste.

Quant à LISTE, elle peut soit être écrite à la main, soit provenir de l'exécution d'une commande (avec SUBST). Voici quelques exemples :

```
for cmpt in 1 6 0 3 4; do
   echo "$cmpt"
done

for name in "tata" "toto" "titi"; do
   echo "$cmpt"
done

for file in $( ls ); do
   echo "$cmpt"
done
```

Nous pouvons aussi générer une longue série de nombres séparés par un pas constant. Par exemple pour les nombres de 1 à 100, nous pouvons écrire la liste ainsi : {1..100}. Si nous voulons la même série mais avec les nombres de 2 en 2 : {1..100..2}. Et pour aller dans l'autre sens, il suffit d'inverser les bornes : {100..1} ou {100..1..2}.

Voici un exemple :

```
#!/bin/bash

for cmpt in {100..0}; do
   echo "$cmpt"
done
echo "Boum!"
```

EXERCICE 16

 Testez ce script.

EXERCICE 17

 Écrivez un script qui affiche pour chaque élément du répertoire courant : « *elementName is a file.* » si l'élément est un fichier ou « `elementName is a directory.` » si c'est un répertoire.

4 Conclusion

Bravo ! Vous avez désormais toutes les clefs pour écrire vos propres scripts. Si la somme de connaissances acquise vous semble colossale, ne vous inquiétez pas, vous pourrez toujours venir vous référer à ce TP le jour où vous en aurez besoin.

Mémo

« Les scripts Bash c'est *des commandes Shell dans un fichier.* »
« À la main c'est fastidieux, alors faites un script pour automatiser ! »
« Un script complexe peut se décomposer en petits scripts simples chaînés. »