Introduction aux systèmes d'exploitation (IS1)

TP 8 – Agenda
1. Fonctions de base

Dans ce TP et les suivants, vous allez développer un petit agenda électronique.

Les executables nécessaires au fonctionnement de l'agenda seront stockés dans votre répertoire `~/bin/agenda/` que l'on prendra soin de mettre dans le `PATH`. Les données de votre agenda seront stockées dans un fichier `.agenda` situé dans votre répertoire personnel. Ce fichier contiendra une liste d'événements : chaque événement est représenté par une ligne dans le fichier `.agenda` de la forme `dd=hd=nom=description=df=hf`, où

- `dd` est la date de début de l'événement, au format `aaaammjj` (par exemple `20081118`). *Remarque : on a choisi de mettre en premier les années, puis les mois et les jours pour que l'ordre lexicographique corresponde à l'ordre chronologique*
- `hd` est l'heure de début de l'événement, au format `hhmm` (par exemple `1030`)
- `df` et `hf` sont la date et l'heure de fin de l'événement
- `nom` et `description` sont le nom et une description (optionnelle) de l'événement

On dira que `dd, hd, nom, description, df` et `hf` sont les champs de l'agenda. Les lignes du fichier `.agenda` seront toujours triées par ordre alphabétique.

Exercice 1 – *Mise en place*

Créez votre fichier `.agenda` contenant deux événements correspondants aux deux TPs de cette semaine. Créez le répertoire `~/bin/agenda/` et mettez-le dans votre `PATH`.

1 Notions de base sur les scripts

Pour éviter d'avoir à ré-écrire des commandes complexes que l'on utilise souvent, il est préférable de les enregistrer dans des fichiers que l'on pourra exécuter quand il nous plaira. On appelle ces fichiers des `scripts`.

Quelques règles avant de commencer.

1. Un script contient des séquences de commandes telles que l'on pourrait les taper dans un terminal. Les commandes successives peuvent être séparées par des retours à la ligne (qui sont interprétés comme des points virgules).
2. Tout fichier de script commence par une ligne permettant d'identifier le programme qui doit être utilisé pour l'exécuter. Dans le cas d'un script `bash`, la première ligne du fichier doit contenir : `#!/usr/local/bin/bash` (vérifiez le chemin de votre `bash` avec la commande `which bash`)
3. Une ligne commençant par le caractère `#` est considérée comme un commentaire et n'est pas exécutée par le shell. Comme toujours, il est très important de bien commenter son script pour qu'il soit compréhensible pour le reste du monde.
4. Par convention, l'extension d'un script est : `.sh`.
5. Pour qu'un utilisateur puisse exécuter un script, il doit posséder les droits en **exécution** et en **lecture** sur ce script.

Variables. Dans un script, on peut utiliser des variables comme dans un terminal (voir TP6). On peut afficher la valeur de la variable *var* avec la commande `echo $var`. On peut assigner une valeur à une variable *var*

- soit avec la commande `var=val` (la variable *var* a alors pour valeur *val*)
- soit avec la commande `read var`, qui attend que l'utilisateur rentre une valeur pour la variable *var*

En fait, la commande `read var1 var2 ... varn` lit une ligne de texte sur l'entrée standard (le terminal en général) et assigne le premier mot à la variable *var1*, le deuxième à la variable *var2*, etc. La dernière variable récupère tous les mots restants.

Paramètres. Comme vous l'avez remarqué, la plupart des commandes Unix peuvent être suivies d'un ou plusieurs paramètres, qui peuvent être des options, des noms de fichiers ou de répertoires, etc.

Il est aussi possible de passer des paramètres à vos scripts. Pour les manipuler, il existe plusieurs variables spéciales. En particulier :

- la variable `#` contient le nombre de paramètres passés au script
- pour chaque entier `i` entre 1 et `$#`, la variable `$i` contient le i-ème paramètre
- la variable `$@` contient la liste de tous les paramètres séparés par des espaces
- la variable `$0` contient le nom du programme en cours d'exécution

Par exemple, si l'on tape la commande `monscript a "b c" d` dans un shell, alors `$#` vaudra 3, `$1` aura pour valeur `a`, `$2` aura pour valeur `b c`, et `$3` aura pour valeur `d`.

Exercice 2 – *Lire une date*

1. Écrivez un exécutable `lireDate.sh` qui lit une date au format `jj mm aaaa` et l'écrit sur la sortie standard au format `aaaammjj`.
2. Modifier votre programme pour qu'il prenne en argument une phrase et l'affiche avant de lire la date. Ainsi, si on veut lire une date de début d'un événement, on utilisera la commande `lireDate.sh "date de début de l'événement ?"` (et de même pour la date de fin).
3. Faites un exécutable `lireHeure.sh` similaire pour lire une heure.

Exercice 3 – *Ajouter un événement dans l'agenda*

1. Écrivez un executable `ajouter.sh` qui demande à l'utilisateur des dates et heures de début et de fin d'un événement, puis lui demande le nom et la description de l'événement, et ajoute cet événement à l'agenda.
2. Que doit-on faire pour qu'en plus, après avoir ajouté le nouvel événement au fichier `.agenda`, les lignes soient encore triées par ordre alphabétique ?

2 Conditionnelles

Valeur de retour. Quand une commande termine, elle renvoie à son environnement (et en particulier à son processus parent) une valeur de retour. En général, cette valeur de retour est `0` si le programme a terminé correctement, et `1` sinon. La valeur de retour du dernier processus exécuté est contenue dans la variable `?`. La commande `exit val` permet de terminer un exécutable en renvoyant la valeur de retour *val*.

Tests. La commande `test` permet, via toutes ses options, d'effectuer de nombreux tests sur des fichiers, des chaînes de caractères, des entiers, etc... La page du manuel de test présente toutes les options possibles. La valeur de retour de `test prop` est `0` si la propriété *prop* est vraie et `1` sinon.

La structure `if then else fi`**.** La commande `if` *cmds* `; then` *cmds1* `; else` *cmds2* `; fi` décide de ce qu'elle fait en fonction de la valeur de retour de la séquence de commandes *cmds* :
- si la valeur de retour de *cmds* est 0, alors, la séquence de commandes *cmds1* est exécutée
- sinon, la séquence de commandes *cmds2* est exécutée

Si on veut une condition alternative, on peut utiliser le `elif` (contraction de `else if`) : la commande `if` *cmds1* `; then` *cmds2* `; elif` *cmds3* `; then` *cmds4* `; else` *cmds5* `; fi` decide de son comportement en fonction des valeurs de retour de *cmds1* et *cmds3* :
- si la valeur de retour de *cmds1* est 0, alors *cmds2* est exécutée
- sinon,
  - si la valeur de retour de *cmds3* est 0, alors *cmds4* est exécutée
  - sinon, *cmds5* est exécutée

La structure `case`**.** On utilise la commande `case` quand le comportement d'un programme dépend de la valeur que prend une certaine variable :

```
case $var in
           v1 ) cmds1 ;;
       v2 | v3 ) cmds2 ;;
             *) cmds3 ;;
esac
```

Si la valeur de la variable *var* est *v1* alors la séquence de commandes *cmds1* est exécutée. Si la valeur de *var* est *v2* ou *v3* alors la séquence de commandes *cmds2* est exécutée. Si *var* a une autre valeur alors la séquence de commandes *cmds3* est exécutée.

Notez le double point-virgule pour terminer chaque liste d'instructions.

Les constructeurs `&&` **et** `||`**.** Les constructeurs `&&` et `||` permettent de faire dépendre le comportement d'une suite de commandes de leurs valeurs de retour successives :
- lorsque l'on tape la commande *cmds1* `&&` *cmds2*, la séquence de commandes *cmds1* est exécutée. Si sa valeur de retour est nulle, la séquence de commandes *cmds2* est exécutée. Sinon, elle ne l'est pas et la valeur de retour reste non nulle.
- lorsque l'on tape la commande *cmds1* `||` *cmds2*, la séquence de commandes *cmds1* est exécutée. Si sa valeur de retour est non nulle, la séquence de commandes *cmds2* est exécutée. Sinon, elle ne l'est pas et la valeur de retour reste nulle.

Exercice 4 – *Vérifier une date et une heure*

1. Écrivez un exécutable `verifierHeure.sh` qui reçoit 2 arguments au format `hh` et `mm` et renvoie comme valeur de retour 0 si `hh:mm` est une heure et 1 sinon.
2. Écrivez un exécutable `verifierDate.sh` qui reçoit 3 arguments au format `jj`, `mm` et `aaaa` et renvoie comme valeur de retour 0 si `jj/mm/aaaa` est une date[1] et 1 sinon.
3. Écrivez un exécutable `verifierDateHeure.sh` qui reçoit 1 option (soit `-d`, soit `-h`), et 0, 2 ou 3 arguments et qui :
   - s'il reçoit l'option `-d` et 0 arguments, demande à l'utilisateur de lui rentrer une date et si ça n'en est pas une, affiche une erreur sur la sortie erreur
   - s'il reçoit l'option `-d` et 3 arguments, vérifie si ces arguments forment une date et renvoie le résultat comme valeur de retour (sans rien afficher sur la sortie erreur)
   - s'il reçoit l'option `-h` et 0 arguments, demande à l'utilisateur de lui rentrer une heure et si ça n'en est pas une, affiche une erreur sur la sortie erreur

[1] On rappelle qu'une année est bissextile si elle est divisible par 4, mais pas par 100, ou bien si elle est divisible par 400. Le reste de la division euclidienne de `a` par `b` s'obtient par `$((a%b))`.

   - s'il reçoit l'option `-h` et 2 arguments, vérifie si ces arguments forment une heure et renvoie le résultat comme valeur de retour (sans rien afficher sur la sortie erreur)
   - dans tous les autres cas, affiche un message d'erreur sur la sortie erreur

Exercice 5 – *Ajouter (bis)*

Modifier votre exécutable `ajouter.sh` pour qu'avant de rajouter un événement dans le fichier `.agenda`, il vérifie que les dates et heures de début et de fin sont valides, et que la fin est postérieure au début.

3 Sélection de caractères dans une chaîne

Manipulation d'une chaîne de caractères Le shell bash propose un certain nombre d'outils simples pour manipuler une chaîne de caractères. Ainsi, si une variable `m` a pour valeur `abcdefghijkl`, alors on peut récupérer certaines de ses sous-chaînes avec les commandes suivantes :
- `echo ${#m}` affiche le nombre de caractères de la chaîne `m`, donc ici 12
- `echo ${m:4}` supprime les 4 premières lettres de `m` : on obtient donc `efghijklm`
- `echo ${m:4:3}` supprime les 4 premières lettres de `m` et ne garde que les 3 suivantes : on obtient donc `efg`
- `echo ${m#abc}` renvoie la chaîne `m`, à laquelle on a retiré au début la chaîne `abc`
- `echo ${m%ijkl}` renvoie la chaîne `m`, à laquelle on a retiré à la fin la chaîne `ijkl`

Exercice 6 – *Afficher un événement*

1. Écrivez un exécutable `afficherDate.sh` qui prend en argument une date au format `aaaammjj` et l'affiche au format `jj/mm/aaaa` (faites un exécutable similaire pour les heures).
2. Étant donné un mot `m` dont on ne sait a priori pas la longueur, comment peut on récupérer ses 13 dernières lettres ?
3. Écrivez un exécutable `afficherEvenement.sh` qui prend en argument une ligne de l'agenda et l'affiche correctement. Par exemple, l'événement

   `20081118=1030=TP_IS1=agenda1=20081118=1230`

   doit être affiché de la façon suivante :

   ```
   début : 18/11/2008 10:30
   fin : 18/11/2008 12:30
   TP_IS1=agenda1
   ```

Les commandes `cut` **et** `paste`**.** La commande `cut` permet d'extraire une portion de chaque ligne d'un fichier. Cette portion peut être
- soit un certain nombre de caractères donnés par leur position depuis le début de la ligne : par exemple la commande `cut -c1,4-7,8 fic` va retourner, pour chaque ligne de `fic` les caractères situés en positions 1,4,5,6,7 et 8 sur cette ligne.
- soit des champs situés entre des délimiteurs. L'option `-d` permet de définir les délimiteurs, et l'option `-f` permet de désigner les champs : par exemple, la commande `cut -d/ -f2 fic` va retourner, pour chaque ligne de `fic` les caractères situés entre le premier et le deuxième caractère / sur cette ligne.

La commande `cut` est en fait un filtre : les commandes `cut -c1 fic` et `cat fic | cut -c1` sont équivalentes.

La commande `paste`, au contraire, permet de recoller des fichiers ligne par ligne. Par exemple, si on tape `paste fic1 fic2 > fic3`, la première ligne de `fic3` est constituée de la première ligne de `fic1` et de la première ligne de `fic2`, séparées par un espace, et de même pour les autres lignes.

Exercice 7 – *Afficher (bis)*

1. Améliorez votre exécutable `afficherEvenement.sh` pour qu'il affiche l'événement
   `20081118=1030=TP_IS1=agenda1=20081118=1230`
   de la façon suivante :
   ```
   événement : TP1_IS1
   début : 18/11/2008 10:30
   fin : 18/11/2008 12:30
   description : agenda1
   ```
2. On a spécifié dans l'introduction que le champ `description` est optionnel. Faites en sorte que si un événement n'a pas de description, la ligne `description :` ne soit pas affichée.

4 Expressions régulières, `grep`, `expr` et `sed`

Rechercher une chaîne de caractères dans un texte : la commande `grep`. La commande `grep` recherche des chaînes de caractères dans du texte, par exemple la chaîne "bonjour" ou bien encore la chaîne "abcd". Lorsque la chaîne de caractères en question est trouvée, par défaut, `grep` affiche la ligne correspondante.
Exemples :
- `grep abc fic` recherche la chaîne `abc` dans toutes les lignes du fichier `fic`. Les lignes où cette chaîne est trouvée sont envoyées sur la sortie standard
- `grep " " fic` recherche les lignes de `fic` qui contiennent au moins un espace

Expressions régulières. En fait, `grep` ne permet pas seulement de rechercher une chaîne de caractères fixée, mais aussi des chaînes de caractères remplissant certaines conditions. Ces conditions sont exprimées par des *expressions régulières*. Une expression régulière est construite comme une expression arithmétique : on forme une expression complexe en combinant des expressions simples à l'aide d'opérateurs.
Voici quelques opérateurs utilisables dans les expressions régulières fournies à `grep` (il y en a d'autres ! voir dans les pages de manuel pour plus de détails) :

`.`	un seul caractère quelconque
`\?`	l'élément précédent est facultatif `ab\?c` représente l'ensemble `{ac,abc}`
`*`	l'élément précédent est répété 0 ou plusieurs fois `ab*` représente l'ensemble `{a,ab,abb,abbb,...}`
`\+`	l'élément précédent est répété au moins 1 fois `ab\+` représente l'ensemble `{ab,abb,abbb,...}`
`\{n\}`	l'élément précédent est répété exactement n fois `ab\{3\}c` représente `abbbc`
`\{n,\}`	l'élément précédent est répété au moins n fois `ab\{3,\}c` représente l'ensemble `{abbbc,abbbbc,abbbbbc,...}`
`\{n,m\}`	l'élément précédent est répété entre n et m fois `ab\{3,5\}c` représente l'ensemble `{abbbc,abbbbc,abbbbbc}`
`^`	début de ligne `^a` représente toutes les lignes qui commencent par `a`
`$`	fin de ligne `a$` représente toutes les lignes qui finissent par `a`
`[...]`	liste ou intervalle de caractères recherchés `[abcd]` (ou `[a-d]`) représente l'ensemble `{a,b,c,d}`
`[^ab]`	listes des caractères interdits

Exemples :
- l'expression régulière `a*b+c?[^c]` sélectionne les chaînes de caractères `bbbca` et `aabaa` mais pas la chaîne `bcc`
- l'expression régulière `a.*b` sélectionne exactement les chaînes de caractères qui commencent par un `a` et finissent par un `b`

Exercice 8 – *Expressions régulières*

1. Parmi les chaînes de caractères `aabbbb`, `accbbac`, `bbabbab`, `abbdaac`, `abbdbca`, quelles sont celles qui sont sélectionnées par les expressions régulières suivantes :
   - `[ac]*b+`
   - `[^c]*ba?.$`
   - `^a[bd]\{1,3\}[ac]*$`
2. Écrivez une expression régulière (aussi courte que possible) qui sélectionne, parmi les chaînes de caractères `aabbbb`, `accbbac`, `bbabbab`, `abbdaac`, `abbdbca`, les chaînes suivantes :
   - `aabbbb`, `accbbac` et `abbdaac` (*i.e.*, celles qui commencent par `a` et finissent par autre chose)
   - `aabbbb` et `bbabbab` (*i.e.*, celles qui contiennent deux fois `bb`)
   - *(Pour les gens qui s'ennuient)* `bbabbab` et `abbdbca` (*i.e.*, celles qui commencent et finissent par la même lettre)

Vérifiez toutes vos affirmations avec la commande `grep`.

Exercice 9 – *Comprendre* `grep`

1. Que font les commandes suivantes :
   - `grep -n "h.h." fic`
   - `grep -o "a.*c[^c]\{3,5\}c" fic`
2. Comment feriez vous pour :
   - faire afficher le nombre de lignes du fichier `fic` qui commencent par une majuscule et finissent par un point
   - faire afficher les lignes du fichier `fic` qui ne contiennent pas trois lettres `e`
   - faire afficher les lignes du fichier `fic` qui contiennent `bon` mais pas `bonjour`

Exercice 10 – *Chercher une entrée dans l'agenda*

Écrire un exécutable `chercher.sh` qui demande à l'utilisateur un mot clef (ou une expression régulière), recherche dans le fichier `.agenda` les événements dont le nom contient ce mot clef, et affiche les lignes de l'agenda correspondantes. S'il n'y a qu'un seul événement correspondant, vous pouvez même le faire afficher avec votre exécutable `afficher.sh`.

Exercice 11 – *Supprimer une entrée dans l'agenda*

1. Écrire un exécutable `supprimer.sh` qui demande à l'utilisateur un mot clef (ou une expression régulière), et supprime dans le fichier `.agenda` les événements dont le nom contient ce mot clef.
2. Pour plus de sécurité, faites en sorte que cet exécutable affiche la liste des événements qu'il va supprimer et demande confirmation avant de le faire.

Comparer deux chaînes de caractères : la commande `expr` La commande `expr` *ch1* `:` *ch2* compare les chaînes de caractères *ch1* et *ch2*. Si *ch2* est un préfixe de *ch1*, la valeur de retour de `expr` *ch1* `:` *ch2* vaut 0, sinon elle vaut 1. En fait, on peut remplacer *ch2* par n'importe quelle expression régulière, et tester ainsi si un préfixe de *ch1* satisfait les conditions fixées par *ch2*.

Exercice 12 – *Vérifier le format d'une date*

Améliorez vos exécutables `verifierDate.sh` et `verifierHeure.sh` pour qu'ils vérifient le format (*i.e.*, qu'on leur a bien envoyé une date au format `jj mm aaaa`).

Remplacer des motifs : la commande `sed`. Cette commande sert à remplacer automatiquement des motifs par d'autres dans un texte :

`sed -e` *cmd1* `-e` *cmd2* `...-e` *cmdk* `fic`

effectue les k commandes spécifiées, dans l'ordre, pour chacune des lignes du fichier `fic`. Une commande s'écrit

adresse(s) fonction arguments

où *adresse(s)* est une liste de 0, 1 ou 2 adresses et *fonction* est le nom d'une fonctionnalité offerte par `sed`. Une commande n'est appliquée à la ligne en cours de traitement que si l'adresse de cette ligne concorde avec le champ *adresse(s)* de la commande. Consultez le manuel pour découvrir les types d'adresses utilisées par `sed` ainsi que les fonctionnalités qu'il offre. Lorsqu'on utilise une séquence d'expressions régulières dans une commande, elles doivent être délimitées par le caractère *slash* : /*expr1*/.../*exprk*/.

Exercice 13 – *Comprendre* `sed`

1. Que font les commandes suivantes :
   - `sed -e '/^[^a-zA-Z0-9]$/d' fic`
   - `sed -e '3,5s/\([a-z]\)[a-z]*\([a-z]\)/\1*\2/g' fic`
   - `sed -e '/d.\{3\}t/,/f.n/s/a/b/' fic`
2. Comment feriez vous pour :
   - remplacer par ***** tous les mots de cinq lettres commençant par `m` et terminant par `e`
   - remplacer tous les mots qui commencent par une majuscule par cette majuscule suivie d'un point

Exercice 14 – *Afficher (ter)*

1. En utilisant uniquement `sed`, proposez une autre solution pour l'exercice 7.
2. Appliquez cette solution à l'affichage simultané de plusieurs lignes de l'agenda. Profitez-en pour intégrer cette solution à vos exécutables `chercher.sh` et `supprimer.sh`.

5 Quelques problèmes pour aller plus loin

Exercice 15 – *Afficher plusieurs événements*

1. Comment peut-on faire pour que lorsqu'on affiche n événements (par exemple lors d'une recherche), ces événements soient affichés comme à l'exercice 7, et en plus numérotés de 1 à n ?
2. Écrivez un exécutable qui demande une date et affiche tous les événements à cette date, en les numérotant.
3. Améliorez votre exécutable `supprimer.sh` de sorte que si plusieurs événements correspondent à votre mot clef, il vous affiche ces événements en les numérotant, puis vous demande lesquels vous voulez supprimer.

Exercice 16 – *Agenda à l'ouverture d'un nouveau terminal*

Faites en sorte que lorsque vous ouvrez un nouveau terminal, il vous affiche le nombre d'événements de la journée et le premier d'entre eux.

Exercice 17 – *Interface*

1. Développez une petite interface `agenda.sh` qui affiche le menu

   1. Rechercher un événement
   2. Ajouter un événement
   3. Supprimer un événement
   4. Quitter

   puis vous demande ce que vous souhaitez faire et s'exécute.
2. Comment peut-on faire pour que l'on revienne toujours au menu de l'interface après l'exécution du programme correspondant à l'un des 3 premiers choix ?

Exercice 18 – *Pour aller plus loin sur les expressions regulières*

Comment feriez-vous pour

1. vérifier qu'une chaîne de caractères ne contient pas deux lettres a et b qui alternent 3 fois : ...a...b...a...b...a...b... Par exemple, la chaîne `aaabcdbcadcbd` est interdite puisque les lettres `c` et `d` alternent 3 fois.
2. dans un fichier `fic` chaque fois qu'un mot `m1` est répété deux fois, les deux occurences étant seulement séparées par un autre mot `m2` (comme par exemple : `salut bonjour salut`), remplacer `m1 m2 m1` par `m2 m1 m2` ? Que se passe-t-il pour `salut bonjour salut bonjour` ?
3. remplacer chaque mot d'un fichier `fic` par son initiale suivie d'autant d'étoiles que son nombre de lettres ?