Introduction aux systèmes d'exploitation (IS1)

TP 8 – Agenda
1. Fonctions de base
CORRECTION

Voici une correction du TP8. Comme toujours, de nombreuses solutions sont possibles et les miennes ne sont pas toujours meilleures que celles que vous avez choisies. Vous trouverez en italique des remarques d'ordre général sur des sujets que vous avez assez mal compris ou mal traités durant le TP.

Exercice 1 – *Mise en place*

Pour rentrer à la main des évènements dans le fichier `.agenda`, j'utilise le format qui est indiqué dans l'énoncé :

```
$ echo 20081118=1030=TP IS1=debut agenda=20081118=1230 >> ~/.agenda
```

Mon fichier `.agenda` est bien situé à la source de mon répertoire personnel et je vérifie que ses droits sont `rw-r--r--`. Ensuite, je crée le répertoire `agenda` dans le répertoire `bin` :

```
$ mkdir -p ~/bin/agenda
```

L'option `-p` me permet de créer aussi le répertoire `bin` s'il n'existait pas (voir le manuel). Je mets ce répertoire dans la variable `PATH` pour pouvoir atteindre, de n'importe quel répertoire, les exécutables situés dans le répertoire `bin/agenda` :

```
$ PATH=$PATH:~/bin/agenda
```

Enfin, j'ajoute la ligne précédente à mon fichier `.bashrc` :

```
$ echo 'PATH=$PATH:~/bin/agenda' >> ~/.bashrc
```

Ainsi, je n'aurai pas à ajouter le répertoire `bin/agenda` dans le `PATH` à chaque nouvelle session.

1 Notions de base sur les scripts

Quelques remarques :

1. *Quand on vous donne un cahier des charges, il est important de le respecter. On vous demande par exemple d'écrire un exécutable* `verifierDate.sh` *qui prend trois arguments* `jj`, `mm` *et* `aaaa` *et renvoie comme valeur de retour 0 si* `jj/mm/aaaa` *est une date. Écrire un exécutable* `verid` *qui demande à l'utilisateur une date avec la commande* `read`*, et affiche* `bravo, c'est une date` *si vous avez rentré une date correcte, ne constitue pas une réponse acceptable à la question.*
2. *Vous remarquerez que dans ma correction, je me suis efforcé de commenter mes programmes (toutes les lignes qui suivent un #, sont considérées comme des commentaires, et ne sont donc pas exécutées par le* `bash`*). Si je dois relire le code l'an prochain, je pense avoir assez d'éléments pour ne pas perdre du temps à comprendre ce que fait chaque exécutable. Et vous ?*
3. *Souvenez-vous qu'il y a cinq moyens de communiquer avec un exécutable* `shell`*, représentés sur la Figure 1 :*

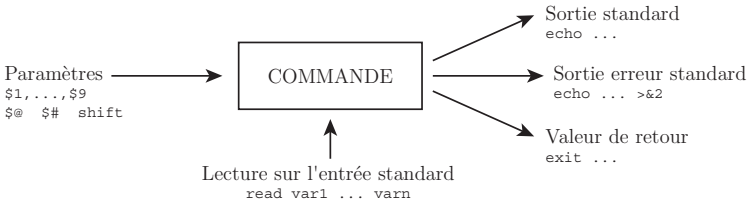

FIG. 1 – Entrées et sorties d'une commande Unix.

   (a) *les paramètres : comme toute commande Unix, vos scripts peuvent prendre un ou plusieurs paramètres. Ces paramètres sont tapés directement après le nom du script : par exemple* `monscript.sh a b c` *a trois paramètres* `a`, `b` *et* `c`*. Comme expliqué dans l'énoncé du TP8, on accède aux valeurs des 9 premiers paramètres grace à* `$1`*,...,* `$9`*. Le nombre de paramètres est stocké dans la variable* `#` *et l'ensemble des paramètres dans* `@`*. La commande* `shift` *sert à décaler tous les paramètres de* `@` *en supprimant le premier (voir le manuel).*

   (b) *la commande* `read` *: 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. Notez bien que la commande* `read` *permet de rentrer des données au programme* pendant *qu'il s'execute.*

   (c) *la sortie standard : pour écrire `phrase` sur la sortie standard, il suffit de taper la commande* `echo` *`phrase`. La sortie standard d'un programme `prg` peut être placée plus tard dans une variable `var` en tapant `var`=$(`prg`), ou encore (`var`=‘`prg`‘).*

   (d) *la sortie erreur : si on veut que la `phrase` s'affiche toujours dans le terminal, il faut la renvoyer sur la sortie erreur avec la redirection* `echo` *`phrase` >&2.*

   (e) *la valeur de retour : elle est utilisée pour les tests dans les structures de contrôle (conditonnelles, boucles). Pour renvoyer une valeur de retour `val`, il faut taper la commande* `exit` *`val`. La valeur de retour du dernier processus exécuté s'obtient avec* `echo $?`.

4. *En* `shell` *comme dans tout langage de programmation, on écrit des programmes simples que l'on utilise pour construire des programmes plus complexes. Dans tout le TP, il est donc indispensable de* réutiliser les petits programmes dans les grands.

Exercice 2 – *Lire une date*

Dans le fichier `lireDate.sh` je tape le script shell suivant :

LIREDATE.SH

```
#!/usr/local/bin/bash
#argument : un message d'invitation ($@)
#read : une date au format jj mm aaaa
#sortie : une date au format aaaammjj

echo $@ >&2
read j m a
echo $a$m$j
```

La première ligne me permet de renvoyer sur la sortie erreur standard tous les arguments rentrés en paramètres (c'est-à-dire le message d'invitation). La deuxième attend que l'utilisateur rentre une date au format `jj mm aaaa`. La dernière affiche la date entrée par l'utilisateur au format `aaaammjj`. Ici, on voit bien la distinction entre

1. paramètres et lecture avec la commande `read` : lorsque je tape `lireDate.sh entrer une date`, le terminal affiche `entrer une date`, puis la commande `read` attend que l'utilisateur tape une date.
2. sortie standard et sortie erreur standard : le message qui invite à taper une date s'affichera toujours sur la sortie erreur (c'est-à-dire le `shell`), alors que la date au format `aaaammjj` s'affiche sur la sortie standard. Ainsi, si je tape dans un programme `d=$(lireDate.sh bonjour)`, alors `bonjour` s'affiche dans le terminal, alors que la date entrée par l'utilisateur est stockée dans la variable `d`.

Comme d'habitude, je rends mon fichier exécutable avec la commande `chmod u+x lireDate.sh` (je ferai ceci pour tous les exécutables de ce TP sans le repréciser à chaque fois).

La commande `lireHeure.sh` est identique.

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

Pour écrire mon exécutable `ajouter.sh`, j'utilise les briques de base que j'ai construites à la question précédente. Si j'ai un exécutable *prg*, alors la commande *var*=$(*prg*) effectue *prg* et passe ce qui sort sur sa sortie standard dans la variable *var*.

AJOUTER.SH

```
#!/usr/local/bin/bash
#demande dates et heures de début et de fin d'un évènement
#demande  nom et description de l'évènement
#ajoute une ligne correspondant à l'évènement dans le fichier .agenda
#retrie le fichier .agenda

dd=$(lireDate.sh "Date de début")
hd=$(lireHeure.sh "Heure de début")
df=$(lireDate.sh "Date de fin")
hf=$(lireHeure.sh "Heure de fin")
echo "Nom" ; read nom
echo "Description (optionnelle)" ; read desc
echo $dd=$hd=$nom=$desc=$df=$hf >> ~/.agenda
sort ~/.agenda -o ~/.agenda
```

J'ai utilisé la commande `sort` pour trier, avec l'option `-o` pour écraser l'ancien fichier `.agenda` et le remplacer par le fichier trié.

2 Conditionnelles

Un programme qui fait un test et renvoie une valeur de retour en fonction du résultat ne fait pas "rien". Ce n'est pas parce qu'un programme n'affiche rien qu'il est inutile...

Par ailleurs, pour faire un test en `shell`*, on utilise la commande* `test`*. Ne vous étonnez pas cependant de trouver aussi les deux structures suivantes :*

- `[ args ]` *est équivalent à* `test args`*. Par exemple,* `[ $x -lt 5 ]` *teste si la valeur de la variable* `x` *est strictement inférieure à* 5.
- *les doubles parenthèses* `((...))` *permettent aussi de faire des tests sur des valeurs entières. Par exemples,* `(( $x<5 ))` *est aussi équivalent à* `test $x -lt 5`*. Dans la suite, j'utilise ces doubles parenthèses pour simplifier l'écriture de mes* `scripts`*.*

Exercice 4 – *Vérifier une date et une heure*

1. Pour l'exécutable `verifierHeure.sh`, il suffit de savoir si `hh`<23 et `mm`<59 :

VERIFIERHEURE.SH

```
#!/usr/local/bin/bash
#arguments : heures hh ($1), minutes mm ($2)
#valeur de retour : 0 si hh:mm est une heure valide, 1 sinon

(( $1<24 && $2<60 ))
```

2. L'exécutable `verifierDate.sh` est un peu plus compliqué. Il y a quatre cas possibles (en fonction du mois) que je sépare par une structure `case`.

VERIFIERDATE.SH

```
#!/usr/local/bin/bash
#arguments : jour ($1), mois ($2), année ($3)
#valeur de retour : 0 si jour/mois/année est une date, 1 sinon

case $2 in
    01|03|05|07|08|10|12) (( $1<=31 )) ;;
    04|06|09|11) (( $1<=30 )) ;;
    02) (( $1<=28 )) || ((( $1==29 )) && bissextile.sh $3) ;;
    *) exit 1 ;;
esac
```

J'ai isolé la difficulté de l'année bissextile dans un petit exécutable `bissextile.sh`. Une année est bissextile si elle est divisible par 4 mais pas par 100, ou alors si elle est divisible par 400 :

BISSEXTILE.SH

```
#!/usr/local/bin/bash
#argument : une année a ($1)
#valeur de retour : 0 si l'année a est bissextile, 1 sinon

(( ($1%4==0 && $1%100!=0) || $1%400==0 ))
```

3. Enfin, l'exécutable `verifierDateHeure.sh` est un jeu de conditionnelles imbriquées :

VERIFIERDATEHEURE.SH

```
#!/usr/local/bin/bash
#paramètres : une option -d ou -h et 2 ou 3 arguments
#comportement : voir l'énoncé du TP8

if (( $#==0 )) ; then echo "erreur" >&2 ; exit 1 ; fi
case $1 in
    -d) case $# in
            1) echo "entrer une date" ; read j m a
               if !(verifierDate.sh $j $m $a) ; then echo "erreur" >&2 ; exit 1 ; fi ;;
            4) verifierDate.sh $2 $3 $4 ;;
            *) echo "erreur" >&2 ; exit 1 ;;
        esac ;;
    -h) case $# in
```

```
            1) echo "entrer une heure" ; read h m
               if !(verifierHeure.sh $h $m); then echo "erreur" >&2 ; exit 1 ; fi ;;
            3) verifierHeure.sh $2 $3 ;;
            *) echo "erreur" >&2 ; exit 1 ;;
        esac ;;
    *) echo "erreur" >&2 ; exit 1 ;;
esac
```

Observez que le nombre de paramètres dans le cas d'une date est soit 1, soit 4 : il y a l'option `-d` plus 0 ou 3 arguments. Même remarque dans le cas d'une heure.

Exercice 5 – *Ajouter (bis)*

Je modifie l'exécutable `lireDate.sh` (resp. `lireHeure.sh`) pour qu'il n'affiche son résultat que si la date (resp. heure) est valide :

LIREDATE.SH

```
#!/usr/local/bin/bash
#argument : un message d'invitation ($@)
#read : une date au format jj mm aaaa
#sortie : une date valide au format aaaammjj

echo $@ >&2
read j m a
if $(verifierDate.sh $j $m $a) ; then echo $a$m$j ; fi
```

Ensuite, je ne rajoute une ligne à l'agenda que si les dates et heures de début et de fin sont valides, et si la fin est postérieure au début :

AJOUTER.SH

```
#!/usr/local/bin/bash
#demande dates et heures de début et de fin d'un évènement
#demande  nom et description de l'évènement
#vérifie la validité des dates et heures
#ajoute une ligne correspondant à l'évènement dans le fichier .agenda
#retrie le fichier .agenda

dd=$(lireDate.sh "Date de début")
hd=$(lireHeure.sh "Heure de début")
df=$(lireDate.sh "Date de fin")
hf=$(lireHeure.sh "Heure de fin")
echo "Nom" ; read nom
echo "Description (optionnelle)" ; read desc
if ( test $dd && test $hd && test $df && test $hf && test $dd$hd < $df$hf )
then echo $dd=$hd=$nom=$desc=$df=$hf >> ~/.agenda ; sort ~/.agenda -o ~/.agenda
else echo "erreur" >&2 ; exit 1
fi
```

La commande `test`, suivie juste d'une chaîne de caractères teste si cette chaîne est vide ou pas. Souvenez-vous aussi qu'écrire les dates à l'envers fait correspondre l'ordre alphabétique et l'ordre chronologique, et donc que `test $dd$hd < $df$hf` teste si la date de début est bien antérieure à la date de fin.

3 Sélection de caractères dans une chaîne

Exercice 6 – *Afficher un événement*

1. On utilise les outils de manipulation de chaînes de caractères présentés dans l'énoncé :

AFFICHERDATE.SH

```
#!/usr/local/bin/bash
#argument : date au format aaaammjj ($1)
#sortie : date au format jj/mm/aaaa

echo ${1:6}/${1:4:2}/${1:0:4}
```

On fait de même pour `afficherHeure.sh`

2. Si on a un mot `m` dont on ne sait pas la longueur, on peut récupérer ses 13 dernières lettres avec `${m:$((${#m}-13))}`, ou même plus simplement `${m:${#m}-13}`.

3. L'exécutable `afficherEvenement.sh` s'écrit alors facilement en utilisant les scripts `afficherDate.sh` et `afficherHeure.sh` :

AFFICHEREVENEMENT.SH

```
#!/usr/local/bin/bash
#argument : une ligne de l'agenda ($1)
#sortie : affichage de l'évènement au format de l'énoncé

debut=${1:0:13} ; milieu=${1:14:${#1}-28} ; fin=${1:${#1}-13}
echo "début : " $(afficherDate.sh ${debut:0:8}) $(afficherHeure.sh ${debut:9:4})
echo "fin : " $(afficherDate.sh ${fin:0:8}) $(afficherHeure.sh ${fin:9:4})
echo $milieu
```

Exercice 7 – *Afficher (bis)*

J'utilise la commande `cut` pour séparer les deux champs `nom` et `description`. Pour cela, il me suffit de définir le délimiteur avec l'option `-d=` et de choisir mes champs avec l'option `-f`. Remarquez l'utilisation du tube.

Pour la seconde question, j'insère juste une conditionnelle.

AFFICHEREVENEMENT.SH

```
#!/usr/local/bin/bash
#argument : une ligne de l'agenda ($1)
#sortie : affichage de l'évènement au format de l'énoncé

echo "évènement : " $(echo $1 | cut -d= -f3)
echo "début : " $(afficherDate.sh ${1:0:8}) $(afficherHeure.sh ${1:9:4})
echo "fin : " $(afficherDate.sh ${1:${#1}-13:8}) $(afficherHeure.sh ${1:${#1}-4})
desc=$(echo $1 | cut -d= -f4)
if test $desc ; then echo "description : " $desc ; fi
```

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

Beaucoup ont remarqué que la commande `grep` *ne faisait pas exactement ce qui est indiqué dans l'énoncé, en particulier pour* `+` *et* `?`*. On pouvait s'en sortir en échappant ces caractères (i.e., en les remplaçant par* `\+` *et* `\?`*), ou bien en utilisant la commande* `egrep`*.*

Exercice 8 – *Expressions régulières*

1. Parmi les chaînes de caractères `aabbbb`, `accbbac`, `bbabbab`, `abbdaac`, `abbdbca`,
   - l'expression régulière `[ac]*b+` sélectionne toutes les chaînes. En effet, pour qu'une chaîne soit sélectionnée par `[ac]*b+`, il suffit qu'elle contienne au moins un `b`
   - l'expression régulière `[^c]*ba?.$` sélectionne les chaînes qui finissent par un `b`, suivi éventuellement d'un `a` puis d'une autre lettre, c'est-à-dire les chaînes `aabbbb`, `accbbac` et `bbabbab`
   - l'expression régulière `^a[bd]\{1,3\}[ac]*$` sélectionne les chaînes qui commencent par un `a`, puis ont entre 1 et 3 lettre(s) `b` ou `d`, puis ont un certain nombre de `a` ou de `c`, c'est-à-dire la chaîne `abbdaac`
2. Voici les expressions régulières qu'il fallait écrire :
   - `^a.*[^a]$` : chaînes qui commencent par `a` et finissent par autre chose que `a`
   - `bb.*bb` : chaînes qui contiennent deux fois `bb` (mais pas forcément `bbbb`)
   - `^\(.\).*\1$` : chaînes qui commencent et finissent par la même lettre

Dans une expression régulière, la présence de parenthèses `\( \)` *sert à se souvenir d'un motif qu'on a rencontré. Par exemple, l'expression régulière* `\([ab]\)\1` *sélectionne les chaînes de caractères qui contiennent* `aa` *ou* `bb` *(à ne pas confondre avec l'expression* `[ab]\{2\}` *qui sélectionne les chaînes de caractères qui contiennent* `aa`*,* `ab`*,* `bb` *ou* `ba`*).*

Exercice 9 – *Comprendre* `grep`

1. En regardant la signification des options dans la page du manuel de grep, on constate que :
   - `grep -n "h.h." fic` sélectionne les lignes du fichier `fic` qui contienne un mot du type `h.h.` (par exemple `haha` ou `hohe`) et les affiche en les faisant précéder de leur numéro de ligne dans le fichier `fic`.
   - `grep -o "a.*c[^c]\{3,5\}c" fic` cherche dans le fichier `fic` le motif `a.*c[^c]\{3,5\}c` (c'est-à-dire quelque chose qui commence par `a`, suivi d'un certain nombre de caractères quelconques, puis d'un `c`, puis de 3 à 5 caractères autre que `c`, puis d'un dernier `c`). Seules les portions de lignes correspondantes au motif sont affichées.
2. Voici les commandes à taper :
   - `grep -c "^[A-Z].*\."$ fic` pour faire afficher le nombre de lignes du fichier `fic` qui commencent par une majuscule et finissent par un point. L'option `-c` permet de compter les lignes au lieu de les afficher. On pouvait aussi utiliser un tube vers la commande `wc -l`. Remarquez l'échappement `\.` avant le point en fin de ligne (un point tout seul signifie n'importe quel caractère).
   - `grep -v "e.*e.*e" fic` pour faire afficher les lignes du fichier `fic` qui ne contiennent pas trois lettres `e`. L'option `-v` permet d'inverser la sélection. Remarquez que `e.*e.*e` est différent de `eee` qui ne sélectionne que les lignes ayant trois `e` consécutifs.
   - `grep bon fic | grep -v bonjour` pour faire afficher les lignes du fichier `fic` qui contiennent `bon` mais pas `bonjour`.

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

Pour chercher une entrée dans l'agenda à partir d'un mot clef, on utilise évidemment ce qu'on vient d'apprendre sur `grep` :

CHERCHER.SH

```
#!/usr/local/bin/bash
#read : un mot clef
#sortie : affichage de(s) (l')évènement(s) contenant le mot clef

echo "entrer un mot clef" ; read clef
case $(grep -c "$clef" ~/.agenda) in
    0) echo "pas d\'évènement correspondant" ;;
    1) afficherEvenement.sh "$(grep "$clef" ~/.agenda)" ;;
    *) grep "$clef" ~/.agenda ;;
esac
```

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

Pour supprimer une entrée dans l'agenda, on utilise l'option `-v` de `grep` vue à l'exercice 9 :

SUPPRIMER.SH

```
#!/usr/local/bin/bash
#read : un mot clef
#supprime du fichier .agenda le(s) évènement(s) contenant le mot clef

echo "entrer un mot clef" ; read clef
echo "Supprimer les lignes suivantes ?" ; grep "$clef" ~/.agenda
echo -n "(oui/non) " ; read rep
if test $rep = oui ; then grep -v "$clef" ~/.agenda > pr ; mv -f pr ~/.agenda ; fi
```

J'utilise un fichier provisoire `pr` pour ne pas effacer tout le fichier `.agenda` (c'est le même problème qu'avec la commande `sort`). Je le déplace ensuite sur `~/.agenda` avec la commande `mv`.

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

Pour vérifier le format des arguments, je rajoute au début de mon fichier `verifierDate.sh` une ligne qui teste si j'ai bien 3 arguments qui sont tels que `$1 $2 $3` corresponde bien au motif `[0-9]\{2\} [0-9]\{2\} [0-9]\{4\}$` (*i.e.*, "deux chiffres - deux chiffres - quatre chiffres") :

VERIFIERDATE.SH

```
#!/usr/local/bin/bash
#arguments : jour ($1), mois ($2), année ($3)
#valeur de retour : 0 si jour/mois/année est une date, 1 sinon

if (( $#!=3 )) || ! expr "$1 $2 $3" : "[0-9]\{2\} [0-9]\{2\} [0-9]\{4\}$" > /dev/null
then echo "erreur" >&2 ; exit 1 ; fi
case $2 in
    01|03|05|07|08|10|12) (( $1<=31 )) ;;
    04|06|09|11) (( $1<=30 )) ;;
    02) (( $1<=28 )) || ((( $1==29 )) && bissextile.sh $3) ;;
    *) exit 1 ;;
esac
```

Je renvoie le message de la commande `expr` dans la poubelle universelle `/dev/null`.
Je fais de même pour `verifierHeure.sh`.

Exercice 13 – *Comprendre* `sed`

1. Voici la signification des commandes :
   - `sed -e '/^[^a-zA-Z0-9]$/d' fic` affiche le fichier `fic` en supprimant toutes les lignes qui contiennent exactement un caractère qui n'est ni une lettre, ni un chiffre.
   - `sed -e '3,5s/\([a-z]\)[a-z]*\([a-z]\)/\1*\2/g' fic` affiche le fichier `fic`, en remplaçant dans les lignes 3, 4 et 5 tous les mots ne contenant que des lettres minuscules par leur première et leur dernière lettre, séparées par une `*`. Remarquez l'utilisation des marqueurs `\( \)` et de `\1` et `\2` qui permettent de se souvenir d'une chaîne rencontrée.
   - Pour la commande `sed -e '/d.\{3\}t/,/f.n/s/a/b/' fic`, c'est un peu plus compliqué. Soit `n` le numéro de la première ligne contenant le motif `d.\{3\}t` (par exemple le mot `début`), et `m` le numéro de la première ligne après `n` contenant le motif `f.n` (par exemple le mot `fin`). Alors `sed -e /d.\{3\}t/,/f.n/s/a/b/ fic` affiche le fichier `fic` en remplaçant dans toutes les lignes entre `n` et `m` le premier `a` par un `b`.
2. Voici les commandes à taper :
   - pour remplacer par `*****` tous les mots de cinq lettres commençant par `m` et terminant par `e`, on commence par mettre au début et à la fin de chaque mot une marque, par exemple le caractère `§` : `sed -e 's/ /§ §/g' -e 's/^/§/' -e 's/$/§/' fic`. Ensuite, il suffit de remplacer tous les motifs `§m[^§]\{3\}e§` par `*****` et de supprimer les derniers `§`. On obtient donc la commande (à écrire sur une seule ligne bien sûr) :
     ```
     sed -e 's/ /§ §/g' -e 's/^/§/' -e 's/$/§/'
         -e 's/§m[^§]\{3\}e§/*****/g' -e 's/§//g' fic
     ```
   - pour remplacer tous les mots qui commencent par une majuscule par cette majuscule suivie d'un point, on cherche des motifs `\([A-Z]\)[a-zA-Z]*` qui suivent soit un espace, soit le début de la ligne, et on les remplace par `\1.` :
     ```
     sed -e 's/ \([A-Z]\)[a-zA-Z]*/ \1./g' -e 's/^\([A-Z]\)[a-zA-Z]*/\1./g' fic
     ```
     L'utilisation des marqueurs `\( \)` est ici indispensable.

Exercice 14 – *Afficher (ter)*

Comme toujours, il y a plusieurs méthodes pour cet exercice. J'en propose ici deux qui me paraissent intéressantes pour des raisons différentes.

La première consiste à utiliser exclusivement `sed` : étant donnée une (ou plusieurs) ligne(s) de l'agenda, je remplace d'abord le motif

`\([0-9]\{4\}\)\([0-9]\{2\}\)\([0-9]\{2\}\)=\([0-9]\{2\}\)\([0-9]\{2\}\)`

par `\3\/\2\/\1 \4:\5` (*i.e.*, je mets les dates et heures dans un format lisible). Ensuite, je remplace le motif `\([^=]\{16\}\)=\([^=]*\)=\([^=]*\)=\([^=]\{16\}\)` par

`evenement : \2\\ndebut : \1\\nfin : \4\\ndescription : \3\\n.`

Ensuite, il ne reste plus qu'à afficher : la commande `echo -e` permet d'afficher le résultat en sautant des lignes à chaque `\n`.

AFFICHEREVENEMENTS.SH

```
#!/usr/local/bin/bash
#argument : une ou plusieurs lignes de l'agenda
#sortie : affichage de(s) (l')évènement(s) au format de l'énoncé

echo -e $(echo $1 | sed -e 's/\([0-9]\{4\}\)\([0-9]\{2\}\)\([0-9]\{2\}\)=\([0-9]\
{2\}\)\([0-9]\{2\}\)/\3\/\2\/\1 \4:\5/g' -e 's/\([^=]\{16\}\)=\([^=]*\)=\([^=]*\)
=\([^=]\{16\}\)/evenement : \2\\ndebut : \1\\nfin : \4\\ndescription : \3\\n/g')
```

Pour ne pas afficher la ligne description quand elle est vide, il faut utiliser un motif de plus.

Une autre solution est de faire écrire par notre programme un sous-exécutable `commande.sh` qui fait ce qui nous intéresse et de le lui faire exécuter. Ainsi, mon programme crée un fichier `commande.sh`, puis lui donne le droit d'exécution `x`. Ensuite, pour chaque ligne de l'agenda de la forme `A=B=C=D=E=F`, mon programme écrit dans ce fichier `commande.sh` les lignes suivantes :

```
echo "évènement : " C ;
echo "début : " $(afficherDate.sh A) $(afficherHeure.sh B) ;
echo "fin : " $(afficherDate.sh E) $(afficherHeure.sh F) ;
if test D ; then echo "description : " D ; fi ;
echo ;
```

Enfin, mon programme exécute la commande `commande.sh` qu'il a construite, puis l'efface. Voici ce que ça donne en `shell` :

AFFICHEREVENEMENTS.SH

```
#!/usr/local/bin/bash
#argument : une ou plusieurs lignes de l'agenda
#sortie : affichage de(s) (l')évènement(s) au format de l'énoncé

echo '#!/usr/local/bin/bash' > commande.sh
echo echo >> commande.sh
chmod u+x commande.sh

echo $1 | sed -e s/\([0-9]*\)=\([0-9]*\)=\([^=]*\)=\([^=]*\)=\([0-9]*\)=\([0-9]*\
)/echo "évènement : " \3 ; echo "début : " $(afficherDate.sh \1) $(afficherHeure.
sh \2) ; echo "fin : " $(afficherDate.sh \5) $(afficherHeure.sh \6) ; if test "\4
" ; then echo "description : " \4 ; fi ; echo ;/g >> commande.sh

./commande.sh
rm -f commande.sh
```

On peut alors intégrer tout ça dans les exécutables `chercher.sh` et `supprimer.sh` :

CHERCHER.SH

```
#!/usr/local/bin/bash
#read : un mot clef
#sortie : affichage de(s) (l')évènement(s) contenant le mot clef

echo "entrer un mot clef" ; read clef
case $(grep -c "$clef "~/.agenda) in
    0) echo "pas d\'évènement correspondant" ;;
    *) afficherEvenements.sh "$(grep "$clef" ~/.agenda)" ;;
esac
```

SUPPRIMER.SH

```
#!/usr/local/bin/bash
#read : un mot clef
#supprime du fichier .agenda le(s) évènement(s) contenant le mot clef

echo "entrer un mot clef" ; read clef
echo "Supprimer les évènements suivants ?"
afficherEvenements.sh "$(grep "$clef" ~/.agenda)"
echo -n "(oui/non) " ; read rep
if test $rep = oui ; then grep -v "$clef" ~/.agenda > pr ; mv -f pr ~/.agenda ; fi
```

5 Quelques problèmes pour aller plus loin

Exercice 15 – *Afficher plusieurs événements*

1. Pour que les événements soient en plus numérotés, on peut par exemple reprendre ma deuxième solution à l'exercice précédent et ajouter un compteur. On modifie notre programme `afficherEvenements.sh` de sorte que pour la ième ligne de l'agenda (de la forme `A=B=C=D=E=F`), il écrive dans le fichier `commande.sh`

```
echo "évènement" $i : C ;
echo "début : " $(afficherDate.sh A) $(afficherHeure.sh B) ;
echo "fin : " $(afficherDate.sh E) $(afficherHeure.sh F) ;
if test D ; then echo "description : " D ; fi ;
echo ; i=$(($i+1))
```

Voici ce que ça donne en `shell` :

AFFICHEREVENEMENTS.SH

```
#!/usr/local/bin/bash
#argument : une ou plusieurs lignes de l'agenda
#sortie : affichage de(s) (l')évènement(s) au format de l'énoncé

echo '#!/usr/local/bin/bash' > commande.sh
echo i=1 >> commande.sh
echo echo >> commande.sh
chmod u+x commande.sh

echo $1 | sed -e s/\([0-9]*\)=\([0-9]*\)=\([^=]*\)=\([^=]*\)=\([0-9]*\)=\([0-9]*\
)/echo "evenement" $i : \3 ; echo "debut : " $(afficherDate.sh \1) $(afficherHeur
e.sh \2) ; echo "fin : " $(afficherDate.sh \5) $(afficherHeure.sh \6) ; if test "
\4" ; then echo "description : " \4 ; fi ; echo ;  i=$(($i+1)) ; /g >> commande.sh

./commande.sh
rm -f commande.sh
```

2. Pour afficher tous les événements à une date donnée en les numérotant, on utilise un `grep` associé avec l'exécutable `afficherEvenements.sh` :

AFFICHERJOURNEE.SH

```
#!/usr/local/bin/bash
#lit une date d avec lireDate.sh
#sortie : les évènements à la date d numérotés

d=$(lireDate.sh "entrer une date")
if test $d ; then afficherEvenements.sh "$(grep $d ~/.agenda)" ; fi
```

3. Lorsqu'il y a plusieurs éléments qui contiennent le mot clef, le programme suivant les affiche en les numérotant, puis demande lesquels l'utilisateur souhaite supprimer. L'utilisateur peut alors rentrer quelque chose du type `2,5-7,9` (qui signifie les évènements numérotés $2, 5, 6, 7$ et 9), que je stocke dans la variable `rep`. Je récupère alors dans la variable `l` les numéros de lignes à supprimer avec la commande

```
l=$(echo $(grep -n "$clef" ~/.agenda | cut -d: -f1) | cut -d" " -f"$rep").
```


Ensuite, je supprime toutes ces lignes avec la commande

```
sed $(echo $l | sed -e s/\([0-9]*\)/-e \1d/g) ~/.agenda.
```

Je dirige le résultat sur un fichier provisoire `pr` que je substitue finalement à `.agenda`.

SUPPRIMER.SH

```
#!/usr/local/bin/bash
#read : un mot clef
#supprime du fichier .agenda le(s) évènement(s) contenant le mot clef

echo "entrer un mot clef" ; read clef
case $(grep -c "$clef" ~/.agenda) in
    0) echo "pas d\'évènement correspondant" ; exit 1 ;;
    1) echo "Supprimer l\'évènement suivant ?"
       afficherEvenement.sh "$(grep "$clef" ~/.agenda)"
       echo -n "(oui/non) " ; read rep
       if test $rep = oui ; then grep -v "$clef" ~/.agenda > pr
       mv -f pr ~/.agenda ; fi ;;
    *) echo "Plusieurs évènements correspondent à la recherche"
       afficherEvenements.sh "$(grep "$clef" ~/.agenda)"
       echo -n "Évènements à supprimer ?" ; read rep
       l=$(echo $(grep -n "$clef" ~/.agenda | cut -d: -f1) | cut -d" " -f"$rep")
       sed $(echo $l | sed -e s/\([0-9]*\)/-e \1d/g) ~/.agenda > pr
       mv -f pr ~/.agenda ;;
esac
```

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

Comme d'habitude, pour effectuer une commande à l'ouverture de chaque terminal, il faut mettre cette commande dans le fichier `.bashrc`. Je rajoute donc les deux lignes suivantes :

```
echo $(grep -c "^$(date +%Y%m%d)" ~/.agenda) "évènements aujourd\'hui"
afficherEvenement.sh $(grep "^$(date +%Y%m%d)" ~/.agenda | head -n1)
```

La commande `date`, utilisée avec `+%Y%m%d` (voir le manuel) permet d'avoir la date au format `aaaammjj`. Ensuite, un `grep` sur cette date, avec l'option `-c` renvoie le nombre d'évènements de la journée. La commande `head -n1 fic` permet de prendre la première ligne du fichier `fic`.

Exercice 17 – *Interface*

Pour revenir toujours au menu de l'interface après l'exécution du programme correspondant à l'un des 3 premiers choix du menu, il y a deux possibilités :

1. soit ajouter à la fin de chacun des programmes `chercher.sh`, `ajouter.sh` et `supprimer.sh` la ligne `read ; agenda.sh`. Ainsi, à la fin de ces programmes, on attend que l'utilisateur appuie sur `enter` et on relance le programme `agenda.sh`.
2. utiliser une boucle `while` (voir TP9) sans fin. C'est cette deuxième solution que j'ai choisie dans le programme suivant

AGENDA.SH

```
#!/usr/local/bin/bash
#affiche le menu et effectue la tache demandée par l'utilisateur

while(true)
do
    efface.sh
    echo "1. Rechercher un évènement"
    echo "2. Ajouter un évènement"
    echo "3. Supprimer un évènement"
    echo "4. Quitter"
    echo ; echo "Entrer le numero de la commande" ; read c
    case $c in
        1) efface.sh ; chercher.sh ; read ;;
        2) efface.sh ; ajouter.sh ; read ;;
        3) efface.sh ; supprimer.sh ; read ;;
        4) clear; exit 0 ;;
        *) echo \007 ;;
        esac
done
```

Dans le `script` précédent,

1. la commande `read` sert à faire attendre le terminal après l'exécution d'un des trois premiers choix du menu, pour que l'utilisateur ait le temps de voir le résultat ;
2. la commande `echo \007` emet un bib sonore ;
3. la commande `efface.sh` sert juste à effacer le terminal et à afficher date et heure :

EFFACE.SH

```
#!/usr/local/bin/bash
#efface le terminal et affiche date et heure

clear
echo "AGENDA IS1"
date "+date : %d %h %y%nheure : %H:%M"
echo
```

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

1. Vérifier qu'une chaîne de caractères `mot` ne contient pas deux lettres qui alternent 3 fois :

```
expr $mot : ".*\(.\).*\(.\).*\1.*\2.*\1.*\2.*$"
```

2. Dans une phrase, échanger `m1 m2 m1` par `m2 m1 m2` :

```
sed -e 's/$/ /' -e 's/\([a-z]*\) \([a-z]*\) \1 /\2 \1 \2 /g' fic
```

Avec `salut bonjour salut bonjour`, on obtient `bonjour salut bonjour bonjour`.

3. Remplacer chaque mot par son initiale suivie d'autant d'étoiles que son nombre de lettres : on met une marque `@` au début de chaque mot et une marque `#` après chaque lettre ; on remplace alors `@\(.\)#` par `\1` (*i.e.*, on recopie l'initiale de chaque mot), puis `.#` par `*` (*i.e.*, on remplace les lettres qui ne sont pas des initiales par une étoile) :

```
sed -e s/^/@/ -e s/ / @/g -e s/\([a-z]\)/\1#/g -e s/@\(.\)#/\1/g -e s/\(.\)#/*/g fic
```