INF600A — Langages de script et langages dynamiques
Examen intra (Automne 2018)

Durée: 13h30 – 16h30
Documentation personnelle permise, y compris *laptop* ou tablette, mais en mode «lecture de documents personnels» — pas de sites Web!

Nom: _____

Code permanent: | | | | | | | | | | | | |

1	2	3	4	5	6	Total
/9	/9	/10	/12	/6	/6	/50

- Répondez directement sur le questionnaire, en utilisant seulement le recto des pages.
- Dans quelques cas, vous devez compléter un script partiellement défini.

  Les parties à compléter sont indiquées par un commentaire `# A COMPLETER` ou par une ligne à remplir telle que la suivante : ..... .
- Quelques rappels :
  - `grep -v` *`motif`* : Émet les lignes *qui ne matchent pas* *`motif`*
  - `sort -t`S `-k`n`,`n : Trie une série de lignes sur la base du n^e champ, les champs étant spécifiés par le séparateur «S».
  - `head` retourne les n premières lignes ou n premiers caractères :

```
$ cat foo.txt
abc
def
$ head -1 foo.txt
abc
$ head -3 foo.txt
abc
def
$ head -c1 foo.txt
a$ head -c2 foo.txt
ab$
```

1. Motifs et expressions régulières (9 pts)

Pour chaque motif ci-bas, donnez la **chaine la plus courte possible** qui *matche* le motif.

- Lorsqu'un caractère doit provenir d'une classe (ensemble) spécifique de caractères, utilisez **le premier caractère** de la classe.
- Lorsque vous devez utiliser un caractère arbitraire — par ex., «`?`» pour du *file globbing*, «`.`» pour une expr. rég. simple ou étendue—, **utilisez le caractère `X`**.

Note : Dans le cas de l'expansion de noms de fichiers (*file globbing*), vous pouvez supposer que le motif est utilisé comme argument pour la commande `ls`. Notez que dans les deux cas ci-bas, il n'y a aucune erreur signalée — les motifs génèrent un nom de fichier valide!

Motif	**Expansion de noms de fichiers (*file globbing*)**
`a*b{2,34}d*e*[A-Z]+`	
`b\([!0-9][^0-9]\)?[a-z]`	

Motif	**Expression régulière simple**
`a*b{2,34}d*e*[A-Z]+`	
`b\([!0-9][^0-9]\)?[a-z]`	

Motif	**Expression régulière étendue**
`a*b{2,34}d*e*[A-Z]+`	
`b\([!0-9][^0-9]\)?[a-z]`	

2. Pipelines avec `awk`, `grep`, `sed`, `sort`, `tr` et `xargs` (9 pts)

Soit le fichier suivant :

```
$ cat inscriptions.txt
Bidon,Joe,BIDJ11059001,7316,INF1120:A+|INF3180:B|MET1105:C
Doe,Catherine,DOEC23619007,7416,INF2120:A|INF5171:C|MET1105:B+
Durocher,Nellie,DURN26539801,4702,
Trudel,Gaetan,TRUG05069001,7316,INF1130:A|INF5180:A
```

Indiquez ce qui sera émis sur `stdout` par l'exécution de chacun des pipelines ci-bas.

```
$ cat inscriptions.txt |
  awk -F, '{ print $5 }' |
  grep -v "^$" |
  sed 's/|/\n/g' |
  sed 's/:.*$//' |
  grep -E '^[A-Z]{3}[^1-3].+$' |
  sort --uniq
```

```
$ cat inscriptions.txt |
   grep -E '^([^,]+,){4}.+$' |
   awk -F, '{ print $1 }' |
   sort --reverse
```

```
$ ls Data
47X.txt  73X.txt  74X.txt

$ X=7
$ cat inscriptions.txt |
  awk -F, "\$4 ~ /^$X/ { print \$4 }" |
  sed 's/^\([0-9][0-9]\).*$/\1X/' |
  sort --uniq |
  xargs -I {} mv Data/{}.txt Data/data-{}.txt

$ ls Data
```

3. Script pour traiter un *log* de `git` (10 pts)

Un fichier texte nommé `gitlog.txt` a été produit à partir de la commande `git` suivante, exécutée à partir d'un dépôt `git` :

```
$ git log --pretty=format:"%h,%ad,%an,%ae,%s" --date=short >gitlog.txt
```

Un **extrait** du fichier résultant est présenté à la figure 1. Chaque ligne du fichier décrit un *commit* avec les informations suivantes — fichier CSV (*comma-separated values*) :

- *Hash* (forme abrégée)
- Date
- Nom de l'auteur
- Courriel de l'auteur
- Description (brève) — message de format arbitraire, sans «,»

Figure 1 Extrait du fichier `gitlog.txt`.

```
$ cat gitlog.txt
023e98d,2017-11-30,Maurizio Drocco,drocco@di.unito.it,cleaned from 'we's
ceccc9f,2017-11-30,Maurizio Drocco,drocco@di.unito.it,Framework API
05e02b6,2017-11-30,Marco Aldinucci,marco.aldinucci@unito.it,cleaning
63c6a23,2017-11-28,Maurizio Drocco,drocco@di.unito.it,removed natbib
18ed43e,2017-11-28,Maurizio Drocco,drocco@di.unito.it,removed spark-streaming cite
ebf0df6,2017-11-28,Maurizio Drocco,drocco@di.unito.it,trimmed a lot
1cd4c46,2017-11-28,Marco Aldinucci,aldinuc@gmail.com,minor cut
f0b6446,2017-11-27,Maurizio Drocco,drocco@di.unito.it,typo
ff5043b,2017-11-27,Marco Aldinucci,aldinuc@gmail.com,fixes
   .
   .
   .
0be58c3,2016-04-27,Guy Tremblay,tremblay.guy@uqam.ca,Various modifications to Section 8
6502e0a,2016-04-27,Guy Tremblay,tremblay.guy@uqam.ca,Macro for consistent style of operation name
fdea2b4,2016-04-27,Maurizio Drocco,drocco@di.unito.it,enriched makefile, added .gitignore
89faf02,2016-04-26,Guy Tremblay,tremblay.guy@uqam.ca,Varions minor modifications and remaks
648ccd0,2016-04-26,Guy Tremblay,tremblay.guy@uqam.ca,Revision to abstract
5cec576,2016-04-26,Guy Tremblay,tremblay.guy@uqam.ca,Macro GT so I can indicate some specific remarks
108f6a8,2016-04-26,Guy Tremblay,tremblay.guy@uqam.ca,Demand-driven references + Use of makefile to easi
cee2941,2016-04-26,Claudia Misale,misale@di.unito.it,first commit
8bf99a9,2016-04-26,Claudia Misale,misale@di.unito.it,Initial commit
```

On veut définir un script `process-gitlog.sh` qui va analyser un tel fichier et qui va produire diverses informations. La figure 2 présente des exemples d'utilisation de ce script.

Figure 2 Exemple de sorties produites en utilisant le script `process-gitlog.sh` sur le fichier `gitlog.txt` de la Figure 1. Notez que l'argument pour `nb_commits` peut être une expression régulière **étendue**.

```
$ ./process-gitlog.sh
usage:
  ./process-gitlog.sh fichier cmd [arguments...]


$ ./process-gitlog.sh gitlog.txt nb_commits tremblay
191

$ ./process-gitlog.sh gitlog.txt nb_commits "trz?.+"
191

$ ./process-gitlog.sh gitlog.txt nb_commits "Drocc"
248

$ ./process-gitlog.sh gitlog.txt max_commits
Maurizio Drocco
```

Complétez le squelette de script `process-gitlog.sh` présenté aux pages suivantes pour réaliser les fonctionalités indiquées.

```
#!/bin/bash -
set -o nounset
set -o errexit

usage()  { ... }
erreur() { ... }

# Nombre de commits effectues par un usager.
# Arguments:
#   $1: nom de fichier (log au format indique ci-haut)
#   $2: nom de l'usager (expr. reg. etendue)
nb_commits()  { ... }  # Voir page suivante

# Liste de tous les noms d'usagers du log ou chaque nom
#    n'apparait au plus **qu'une seule fois**
# Arguments:
#   $1: nom de fichier (log)
noms_usager() { ... }  # Voir page suivante

# Nom de l'usager ayant fait le plus grand nombre de commits.
# Arguments:
#   $1: nom de fichier (log)
max_commits() { ... }  # Voir page suivante

main() {
    local log="$1"

    .....
    local cmd="$1"

    .....

    case "$cmd" in
        nb_commits)
            [[ $# == 0 ]] && usage

            .....
        max_commits)

            .....
        *)
            erreur "cmd inconnue: '$cmd'"
    esac
}

[[ $# -lt 2 ]] && usage
main "$@"
```

```
nb_commits() {
    local log="$1"; shift
    local nom="$1"; shift
    # A COMPLETER









}
```

```
noms_usagers() {
    local log="$1"; shift
    # A COMPLETER









}
```

```
max_commits() {
    local log="$1"; shift
    local OIFS="$IFS"; IFS=$'\n'
    for nom in                                ; do
        echo $(nb_commits "$log" "$nom"),"$nom"
    done | sort -n -r | awk -F, 'NR == 1 {                }'
    IFS=$OIFS
}
```

4. Script `gv.sh` : Le/les vins les mieux notés (12 pts)

Avec le script `gv.sh` du Devoir #1, on veut pouvoir identifier **le ou les vins qui sont les mieux notés** — c'est-à-dire le ou les vins qui ont été bus et qui ont reçu les notes les plus élevées. Plus précisément, on veut pouvoir faire cela de deux façons différentes :

a) Avec un pipeline de commandes au niveau du *shell*.

b) Avec une (nouvelle) commande, `mieux_notes`, ajoutée au script `gv.sh`

La figure 3 présente des exemples d'exécution — dont une version «**bonus**» de la commande `mieux_notes` : voir plus loin pour une description plus détaillée du comportement attendu.

Figure 3 Exemples d'exécution pour identifier les vins les mieux notés.

```
$ cat .vins.txt
1:10/06/18:rouge:Barolo:2012:Fontanafreda:30.35:4:Tres bon.
2:10/06/18:rouge:Chianti:2014:Volpaia:32.65:4:Tres bon.
4:03/07/18:blanc:Alsace:2016:Pfaff:16.50:3:Aromatique.
6:03/07/18:rose:Languedoc:2016:Clavel:18.00::
7:22/09/18:blanc:Aligote:2017::12.50:2:Frais.

$ ./gv.sh selectionner --bus | ./gv.sh - lister
1 [rouge - 30.35$]: Barolo 2012, Fontanafreda (10/06/18) => 4 {Tres bon.}
2 [rouge - 32.65$]: Chianti 2014, Volpaia (10/06/18) => 4 {Tres bon.}
4 [blanc - 16.50$]: Alsace 2016, Pfaff (03/07/18) => 3 {Aromatique.}
7 [blanc - 12.50$]: Aligote 2017, (22/09/18) => 2 {Frais.}

$ cmd1 | cmd2 | ... | cmdk   # Pipeline de commandes pour un vin parmi les mieux notes.
Volpaia => 4

$ ./gv.sh mieux_notes | ./gv.sh - lister
2 [rouge - 32.65$]: Chianti 2014, Volpaia (10/06/18) => 4 {Tres bon.}

$ ./gv.sh mieux_notes -3 | ./gv.sh - lister
2 [rouge - 32.65$]: Chianti 2014, Volpaia (10/06/18) => 4 {Tres bon.}
1 [rouge - 30.35$]: Barolo 2012, Fontanafreda (10/06/18) => 4 {Tres bon.}
4 [blanc - 16.50$]: Alsace 2016, Pfaff (03/07/18) => 3 {Aromatique.}

# *** VERSION BONUS: Voir plus loin. ***
$ ./gv.sh mieux_notes_bonus -3 | ./gv.sh - lister
1 [rouge - 30.35$]: Barolo 2012, Fontanafreda (10/06/18) => 4 {Tres bon.}
2 [rouge - 32.65$]: Chianti 2014, Volpaia (10/06/18) => 4 {Tres bon.}
```

[4] a) **Un pipeline pour identifier un vin parmi les mieux notés**

Écrivez un pipeline de commandes — donc au niveau du *shell* — qui permet d'identifier **un vin** parmi ceux ayant été bus et avec la meilleure note. Comme illustré dans la figure 3, la sortie produite indique le **nom** du vin et sa **note**.

Notez qu'il peut y avoir plusieurs vins ayant la même meilleure note, et votre pipeline peut retourner **n'importe lequel** d'entre eux. Donc, dans l'exemple ci-haut, la sortie produite aurait aussi pu être la suivante :

```
Fontanefreda => 4
```

Contrainte à respecter : Pour cet exercice, vous devez utiliser **le plus possible** des appels — pipelinés — **au script** `gv.sh`. Vous pouvez utiliser d'autres commandes Unix lorsque le script `gv.sh` *ne permet pas l'opération* dont vous avez besoin.

```
$
```

[8] b) **Une commande `mieux_notes` pour identifier les k vins les mieux notés**
Complétez le code de la fonction `mieux_notes` dont le squelette est fourni à la p. 11.

- Plus précisément, cette nouvelle commande du script `gv.sh` permet de trouver les k vins avec les meilleures notes. Lorsque l'option «`-k`» n'est pas spécifiée, alors $k = 1$.
- Comme pour `selectionner` et `trier`, la sortie produite (sur `stdout`) est une série d'enregistrements. Pour `mieux_notes`, ces enregistrements sont triés en ordre **décroissant** des notes — des meilleures notes au moins bonnes notes — et dans un ordre arbitraire (à votre choix) pour les vins avec la même note.
- Les cas d'erreurs suivants doivent aussi être traités et signalés :
  - Nombre invalide pour `k`, par exemple, `-0` ou `-2ab`.
  - Argument en trop, par exemple, «`./gv.sh mieux_notes -3 foo`».
- Vous devez définir `gv.sh`, donc vous ne devez pas appeler `./gv.sh` pour faire une sous-tâche ; mais vous pouvez évidemment appeler une fonction définie dans le script.

***** BONUS *****

Un bonus sera accordé pour une version plus «sophistiquée», nommée `mieux_notes_bonus`. Comme l'illustre la figure 3 (dernier appel), cette version a le comportement suivant :

- Seuls les vins ayant **exactement la meilleure note** sont indiqués. Donc, dans l'exemple, seuls ceux avec une note de 4 — la note maximale — sont indiqués. Il est donc possible qu'il y ait **moins** que `k` vins sélectionnés.
- Les vins avec la meilleure note sont présentés **en ordre croissant de prix** — donc du moins cher au plus cher.

Hypothèses et contraintes à respecter pour `mieux_notes` (ou `mieux_notes_bonus`) :

- Comme dans le devoir #1, le séparateur utilisé dans le dépôt (fichier de vins) est donné par la variable `$SEP` (ou `$SEPARATEUR`) — «`:`».
- La fonction `mieux_notes` est appelée du script principal (`gv.sh`) comme toutes les autres fonctions, donc avec la liste de tous les arguments et options fournis sur la ligne de commande, **à l'exception de l'option pour spécifier le dépôt** («`--depot=...`» ou «`-`»). C'est la variable globale `le_depot` qui indique le dépôt à utiliser.

  De plus, dans la fonction, on est certain que le dépôt à utiliser existe, car son existence a été vérifiée au préalable dans le programme principal (fonction `verifier_depot_existe`).
- Pour signaler une erreur, utilisez la fonction suivante (qui émet sa sortie sur `stderr`) :

```
erreur() { local msg="$1"; ...; exit 1; }
```

- Si approprié, utilisez les constantes et expression régulière suivantes :

```
readonly NOTE_MIN=0
readonly NOTE_MAX=5
readonly NOTE_VALIDE="^[${NOTE_MIN}-${NOTE_MAX}]\$"
```

```
#===========
# Commande mieux_notes
#
# Selectionne les k vins qui sont les mieux notes
# (k = 1 si l'option -k n'est pas specifiee)
#
# Arguments: [-k]
#
# Erreurs:
# - k pas un nombre valide (doit etre un entier >= 1)
# - argument(s) en trop
#===========

mieux_notes() {
    # A COMPLETER




}
```

5. Sujets divers (6 pts)

[2] a) Vous venez tout juste de cloner le dépôt `git` que vous allez partager avec votre coéquipier pour le devoir #2 dans le cours INF600Z. Voici l'état de la copie clonée **immédiatement après l'opération de clonage** :

```
$ ls Devoir2
foo.c       foo.c.bak       foo.out       makefile       makefile.bak

$ cd Devoir2; git status
/Users/tremblay/Devoir2
On branch master
nothing to commit, working directory clean
```

Le fichier `foo.out` est l'exécutable généré par la compilation (`gcc`) de `foo.c` ; les fichiers `*.bak` sont des copies de sauvegarde créées par l'éditeur de texte utilisé par votre coéquipier.

Est-ce correct et acceptable comme dépôt? Quel est le problème? Que devriez suggérer à votre coéquipier?

[2] b) On dit des langages de scripts qu'ils sont des «*glue languages*». **Qu'est-ce que cela signifie? Donnez un exemple simple en `bash`.**

[2] c) Dans les tests du devoir #1, les tests sont définis à l'aide de `MiniTest`, un cadre de tests unitaires pour Ruby. Chaque cas de test, spécifié par un appel à la méthode `it_`, vérifie le comportement (normal ou avec erreur) d'une commande spécifique. **Pourquoi peut-on quand même dire qu'il s'agit de «tests d'acceptation» (ou «tests systèmes») et non de «tests unitaires»?**

6. Appels à une fonction `mystere` (6 pts)

Soit la fonction `mystere` à la page 14 — que vous pouvez détacher pour faciliter la consultation. (C'est une fonction «artificielle», i.e., pas une fonction utilisable pour une tâche réelle!)

Soit le fichier `foo.txt`, le seul fichier avec une extension présent dans le répertoire courant :

```
$ ls *.*
foo.txt
```

```
$ cat foo.txt
echo
//ca
//t
```

Indiquez ce qui sera émis sur `stdout` par chacun des appels ci-bas.

```
$ mystere -ec -h -o ec hi -ho abc
```

```
$ mystere $(cat foo.txt | head -1) foo.txt
```

```
$ mystere $(cat foo.txt | grep -v "^[^/]" | sed 's|//|-|') foo.txt
```

```
$ cat foo.txt | mystere -c $(echo -at) foo.txt - <foo.txt
```

```
set -o nounset
set -o errexit

usage() { echo "erreur: Pas d'argument!"; exit -1; }

mystere() {
    [[ $# == 0 ]] && usage

    local c=""
    if [[ $1 =~ ^- ]]; then
        while [[ $1 =~ ^- ]]; do
            c="$c${1#-}"
            shift
        done
    else
        c="$1"
        shift
    fi

    $c "$@"
}
```