UE INF203

Devoir Surveillé du 7 mars 2017

Durée : 1 heure 30 minutes
Documents autorisés : mémo bash **non annoté** - Dictionnaire papier non annoté pour les étudiants non francophones.
Ni calculette, ni aucun appareil électronique.
Le barème est donné à titre indicatif.
Les parties sont indépendantes et peuvent être traitées dans n'importe quel ordre. Il en est de même pour les questions des parties A et B.
Il est conseillé de lire attentivement l'énoncé avant de commencer.

Partie A ($\sim$ 10 points)

Dans cette partie, vous allez écrire (en C) et utiliser un programme qui s'appellera ***greppref***. Ce programme affiche à l'écran, chacun sur une ligne, les mots d'un fichier qui commencent par un certain préfixe. Le nom du fichier et le préfixe sont donnés en argument au programme.

Exemple :

exemple.txt
Si mon tonton
tond ton tonton
ton tonton sera tondu

```
./greppref exemple.txt ton
tonton
tond
ton
tonton
ton
tonton
tondu
```

A1. Préfixe

Écrivez une fonction de prototype :

```
int est_prefixe(char pref[], char mot[]);
```

qui retourne 1 si la chaîne `pref` est préfixe de la chaîne `mot`, et 0 sinon. On rappelle que la chaîne vide est préfixe de toute chaîne, et que toute chaîne est préfixe d'elle-même.
Exemple : si `pref` est la chaîne "ton"

- si `mot` est la chaîne "tondu", la fonction retourne 1,
- si `mot` est la chaîne "sera", la fonction retourne 0.
- si `mot` est la chaîne "to", la fonction retourne 0.

```
int est_prefixe(char pref[], char mot[]) {
int i =0 ;
        while (pref[i] != '\0' && mot[i] != '\0' && pref[i] == mot[i]) {
                i++ ;
        }
        return pref[i]=='\0' ;
}
```

A2. Utilisation du programme *greppref*.

Dans cette question, on suppose que le programme ***greppref*** existe.

1. Quelle est la ligne de commande permettant de créer un fichier ***resultat*** contenant tous les mots du fichier ***exemple.txt*** commençant par `"ton"` ?

```
./greppref exemple.txt ton > resultat
```

2. ***resultat*** étant le fichier créé à la question précédente, donnez la sortie à l'écran de la commande [1]
   `sed s/ton// resultat`

```
ton
d

ton

ton
du
```

3. Donnez une ligne de commande permettant de produire le même affichage sans créer le fichier ***resultat***.

```
./greppref exemple.txt ton | sed s/ton//
```

1. Au cours du TP3, vous avez affiché le premier chapitre de Candide en substituant pour chaque ligne, la première occurence de Cunegonde par Juliette en utilisant la commande : `sed s/Cunegonde/Juliette/ Candide_chapitre1.txt`

Voici le contenu (partiel ou non) des fichiers sources du programme ***greppref*** :

traitement_chaines.h

```
/* retourne 1 si pref est prefixe de mot,
           0 sinon */
int est_prefixe(char pref[], char mot[]);
```

traitement_chaines.c

```
int est_prefixe(char pref[], char mot[]){
        /* RÉPONSE À LA QUESTION A1 */
         }
```

lecture.h

```
/* f est un descripteur de fichier ouvert
   en lecture
 - s'il reste un mot à lire avant la fin du
   fichier, le stocke dans la chaine mot
   et renvoie 1
 - sinon renvoie 0 */
int mot_suivant(FILE *f, char mot[]);
```

lecture.c

```
int est_separateur(char c) {
        return (c==' ' || c == '\n') ;
}

int mot_suivant(FILE *f, char mot[]){
         /* RÉPONSE À LA QUESTION A5 */
         }
```

greppref.c

```
/* programme principal */
/* RÉPONSE À LA QUESTION A3 */
```

A3. Programme principal.
En utilisant les fonctions `mot_suivant` (qui fera l'objet de la question A5) et `est_prefixe` (écrite en A1), écrivez la fonction `main` du fichier ***greppref.c***. On supposera que les mots du fichier à lire ont au plus 20 caractères. Affichez un message d'erreur si le nombre d'arguments est incorrect, ou si le fichier passé en argument n'est pas accessible en lecture.

```
int main(int argc, char *argv[]) {
FILE *f ;
char mot[21] ;
int yamot ;

if (argc != 3) {
        printf("syntaxe : %s fichier prefixe \n", argv[0]) ;
        return 1 ;
        }

f = fopen(argv[1], "r") ;
if (f==NULL) {
        printf("ouverture de %s en lectuer impossible \n", argv[1]) ;
        return 2 ;
        }

yamot=mot_suivant(f, mot) ;
while (yamot) {
        if (est_prefixe(argv[2], mot)) {
                printf("%s\n", mot) ;
        }
        yamot=mot_suivant(f, mot) ;
        }
fclose(f) ;
return 0 ;
}
```

A4. Compilation.

1. — dans quel(s) fichier(s) doit-on trouver la ligne ci-dessous ?
   `#include "traitement_chaines.h"`

   ```
   traitement_chaines.c et greppref.c
   ```

   — dans quel(s) fichier(s) doit-on trouver la ligne ci-dessous ?
   `#include "lecture.h"`

   ```
   lecture.c et greppref.c
   ```

2. Dessinez le graphe de dépendance entre les fichiers lors de la compilation séparée des différents fichiers permettant de construire l'exécutable ***greppref***.
3. Écrivez un fichier ***Makefile*** pour créer l'exécutable ***greppref*** en utilisant la compilation séparée.

```
CC=gcc
CFLAGS=-Wall -Werror

greppref : greppref.o lecture.o traitement_chaines.o
	$(CC) -o greppref greppref.o lecture.o traitement_chaines.o

greppref.o : greppref.c traitement_chaines.h lecture.h
	$(CC) $(CFLAGS) -c greppref.c

traitement_chaines.o : traitement_chaines.c traitement_chaines.h
	$(CC) $(CFLAGS) -c traitement_chaines.c

lecture.o : lecture.c lecture.h
	$(CC) $(CFLAGS) -c lecture.c
```

A5. Lecture d'un mot dans un fichier.

Le fichier à lire est constitué de mots séparés par des espaces ou des retours à la ligne (comme le fichier ***exemple.txt***). La fonction `mot_suivant` est chargée de lire le mot suivant dans le fichier (à partir de la position courante), ou de détecter qu'il ne reste plus de mots à lire.

— s'il reste un mot à lire, celui-ci est lu et stocké dans la chaîne `mot`, et la fonction renvoie 1,
— sinon, la fonction renvoie 0.

Écrivez la fonction `mot_suivant` (vous pouvez utiliser la fonction `est_separateur`). *Indications.* Le mot à lire est éventuellement précédé d'un ou plusieurs séparateurs. La méthode consiste à lire les caractères jusqu'à ce qu'on rencontre soit la fin du fichier, soit un non-séparateur. Dans ce dernier cas, on lit alors les caractères en les mémorisant dans la chaîne `mot` jusqu'à ce qu'on rencontre un séparateur ou la fin du fichier. N'oubliez pas d'ajouter le caractère de fin de chaîne dans `mot`.

```
int mot_suivant(FILE *f, char mot[]) {
char c ;
int i=0 ;
        fscanf(f, "%c", &c) ;
        while (! feof(f) && est_separateur(c)) {
                fscanf(f, "%c", &c) ;
                }
        while (! feof(f) && ! est_separateur(c)) {
                mot[i]=c ;
                fscanf(f, "%c", &c) ;
                i++ ;
        }
        mot[i]='\0' ;
        return i!=0 ;
}
```

Partie B (∼ 5 points)

On rappelle que pour écrire des entiers en base `b`, on utilise `b` symboles différents. Lorsque `b` est inférieur ou égal à 10, les symboles utilisés sont les chiffres `'0'`, `'1'`, ... Par exemple

- en base 2, on utilise `'0'` et `'1'`,
- en base 5 on utilise `'0'`, `'1'`, `'2'`, `'3'` et `'4'`,
- en base 10 on utilise `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, `'6'`, `'7'`, `'8'` et `'9'`.

B1. Écrivez une fonction C de profil

```
int verif(int b, char chiffres[]);
```

qui vérifie que `b` est compris entre 2 et 10 inclus, et que la chaîne de caractères `chiffres` est l'écriture d'un nombre en base `b` : dans ce cas, la fonction `verif` retourne 1, sinon, elle retourne 0;

```
int verif(int b, char chiffres[]) {
int i ;
int res ;
        res = ((2 <= b) && (b <= 10)) ;
        if (res) {
                i = 0 ;
                while ((chiffres[i] != '\0') &&
                       ('0' <= chiffres[i]) &&
                       (chiffres[i] < '0' + b)) {
                        i++ ;
                }
                res = res && (chiffres[i] == '\0') ;
        }
        return res ;
}
```

B2. Schéma de Horner.
Écrivez une fonction C de profil

```
int horner(int b, char chiffres[]);
```

qui calcule, en utilisant le schéma de Horner, la valeur de l'entier écrit en base `b` dans la chaîne **de caractères** `chiffres`, et retourne le résultat obtenu; on supposera que la base `b` est comprise entre 2 et 10 (inclus), et que la chaîne `chiffres` est l'écriture d'un nombre en base `b`, c'est-à-dire uniquement composée de caractères représentant des chiffres en base `b`.
Exemple de résultat attendu :

- si b=2 et chiffres="101010", le résultat retourné sera 42
- si b=8 et chiffres="313", le résultat retourné sera 203

```
int horner(int b, char chiffres[]) {
int i ;
int res ;

        res = 0 ;
        i = 0 ;
        while (chiffres[i] != '\0') {
                res = b * res + chiffres[i] - '0' ;
                i++ ;
                }
        return res ;
}
```

Voici le programme ***convertit.c*** qui utilise les fonctions `verif` et `horner` des questions précédentes :

convertit.c

```
int verif(int b, char chiffres[]) { /* RÉPONSE À LA QUESTION B1 */ }

int horner(int b, char chiffres[]) { /* RÉPONSE À LA QUESTION B2 */ }

int main(int argc, char *argv[]) {
int base ;
int nombre ;

    if (argc != 3) {
            printf("Syntaxe : %s base nombre \n", argv[0]) ;
            return 1 ;
            }
    base = horner(10, argv[1]) ;
    if (verif(base, argv[2])) {
            nombre = horner(base, argv[2]) ;
            printf(" --> %d\n", nombre) ;
            }
    else {
            printf("%s n'est pas une base admise ou %s n'est pas un nombre
            ecrit en base %s\n", argv[1], argv[2], argv[1]) ;
            }
    return 0 ;
}
```

B3. L'exécutable produit à partir de ce programme s'appelle ***convertit***. Pour chacune des trois instructions `printf` de ce code, proposez des arguments (pour le programme ***convertit***) permettant de provoquer l'exécution de ce `printf`. Sur la ligne suivante, écrivez ce qui s'affiche exactement à l'écran.

```
./convertit
./convertit base nombre

./convertit 2 101010
 --> 42

./convertit 2 313
2 n'est pas une base admise ou 313 n'est pas un nombre ecrit en base 2
```

Partie C (~ 5 points)

On cherche à comparer les sorties produites par deux programmes sur un ensemble de fichiers de données, à l'aide d'un script shell ***compare.sh***.

Les deux programmes lisent l'entrée standard (le clavier), et écrivent sur la sortie standard (l'écran).
Le script prend en arguments les noms des deux programmes, le répertoire contenant les fichiers de données, ainsi que l'extension commune à tous les fichiers de données.
Par exemple,

```
./compare.sh ./mon_prog ./prog_reference Donnees .in
```

aura pour effet de comparer les sorties produites par les programmes ***mon_prog*** et ***prog_reference*** pour chacun des fichiers de données du répertoire ***Donnees*** suffixés par **.in**.

C1. Vérification des arguments.
Écrivez les lignes du script ***compare.sh*** permettant de vérifier que ses arguments sont ceux attendus :

- il y a 4 arguments
- le premier et le deuxième sont des fichiers exécutables
- le troisième est un répertoire.

Un message d'erreur explicite sera affiché dans le cas où une des conditions ci-dessus ne serait pas remplie.

```
if [ $# -ne 4 ]
then
        echo $0 prog1 prog2 repertoire_donnees suffixe_donnees
elif [ ! -f $1 -o ! -f $2  -o ! -x $1 -o ! -x $2 ]
then
        echo $1 ou $2 n\'est pas un pgm executable
elif [ ! -d $3 ]
then
        echo $3 n\'est pas un repertoire
[...]
fi
```

C2. Extraction des noms des fichiers de données sans leur extension.
En supposant que les arguments sont corrects, écrivez les lignes du script ***compare.sh*** permettant d'extraire (et d'afficher) les noms des fichiers du répertoire (troisième argument) ayant pour extension le quatrième argument, sans cette extension.
Par exemple, si le répertoire contient les fichiers :

```
exemple.in  haha.txt  mes_donnees.in  ploum.in  sorties.out  sorties.txt
```

et si l'extension est `.in`, alors le script affichera :

```
exemple
mes_donnees
ploum
```

```
for f in $3/*$4
do
        pref=$(basename $f $4)
[...]
done
```

C3. Exécution et comparaison.
Sans ré-écrire les réponses aux questions précédentes, écrivez les lignes du script ***compare.sh*** permettant pour chaque fichier de données :

- d'exécuter le premier programme en redirigeant ses entrées depuis ce fichier, et ses sorties vers un fichier de même nom mais dont l'extension est remplacée par `.1` (dans le répertoire courant),
- d'exécuter le deuxième programme en redirigeant ses entrées depuis ce fichier, et ses sorties vers un fichier de même nom mais dont l'extension est remplacée par `.2` (dans le répertoire courant),

— de comparer les deux fichiers de sortie produits,
— d'afficher un message dans le cas où ces deux fichiers sont différents.

```
./$1 < $f > $pref.1
./$2 < $f > $pref.2
diff $pref.1 $pref.2
if [ $? -eq 0 ]
then
        echo resultat identique pour $f
else
        echo les resultats pour $f different
fi
```