# TP noté de programmation
# Durée 4h

15 avril 2003

Ce sujet comporte 3 pages.

Le but de ce sujet est d'étudier deux systèmes d'aide à l'écriture de mots.

## Remarques générales

Chaque partie est de durée équivalente donc ne passez pas 4 heures sur la même partie.

Les profils des fonctions vous sont donnés à titre indicatif. Rien ne vous empêche d'écrire des fonctions intermédiaires.

**Commentez votre code.**

Il vous est demandé de fournir, en plus du code, les traces d'exécution qui vous ont permis de tester vos fonctions (rajoutez les en commentaire à la fin du code).

Pour tester vos programmes, un dictionnaire vous est fourni. Il est cependant conseillé de tester en premier lieu vos fonctions avec votre propre dictionnaire (qui ne contiendrait, par exemple, que 3 ou 4 mots)

## [C] Écriture intuitive (système T9)

L'écriture intuitive est utilisée en particulier par les téléphones portables.

Comme les claviers (cf. tableau 1) de ceux-ci ne comportent que 8 touches permettant de taper des caractères, la méthode de saisie traditionnelle consiste pour l'utilisateur à appuyer plusieurs fois sur une même touche jusqu'à obtenir le caractère désiré. Pour saisir deux caractère consécutifs accessibles à partir de la même touche, on doit faire une courte pause entre les deux. Par exemple le mot « *actes* » correspondra à la séquence suivante : 2 *pause* 2 2 2 8 3 3 7 7 7 7.

| | | |
|---|---|---|
| 1 | 2 abc | 3 def |
| 4 ghi | 5 jkl | 6 mno |
| 7 pqrs | 8 tuv | 9 wxyz |

Tab. 1 – Exemple d'un clavier de téléphone simplifié

Le système T9 permet à l'utilisateur de taper seulement une fois la touche contenant la lettre désirée et de saisir ainsi toutes les lettres du mot souhaité. Le système détermine alors tous les mots existants (à partir d'un dictionnaire) parmi toutes les combinaisons de lettres possibles et les propose à l'utilisateur. Par exemple, pour le mot *actes*, l'utilisateur n'a plus qu'à taper 2 2 8 3 7. Le système propose alors les mots *actes*, *caves* et *caver* (tout dépend du dictionnaire utilisé).

Le but de cette première partie est d'écrire en **C** un programme qui permet à un utilisateur de saisir une séquence de chiffres et qui affiche en résultat la liste des mots accessibles pour ce code. On utilisera le dictionnaire `~serveur/PROG/Ocaml/dico.txt`.

1. Écrire une fonction `int calcul_code(char *)` qui calcule le code correspondant à une chaîne de caractères. Indication : on peut utiliser un tableau de 26 cases qui contient le code correspondant à chaque lettre. Par exemple, dans le tableau suivant, à la lettre *a* correspond la case 0 qui contient le code 2, à la lettre *b*, la case 1 contenant le code 2, etc...

   ```
   int code[26] = { 2, 2, 2, 3, 3, 3, ...};
   ```

   On accède facilement aux différentes cases du tableau en tenant compte du fait que le code ASCII est continu et strictement croissant. Ainsi, si `code[0]` correspond à la lettre *a*, `code['f'-'a']` correspond à la lettre *f*.
2. Écrire une fonction `liste lecture(char* nom_dico)` de lecture du dictionnaire qui stocke chaque mot et le code associé dans une liste chaînée triée en utilisant les fonctions disponibles dans `chaine.o` et `chaine.h` (dans le répertoire `~serveur/PROG/C`)
3. Écrire une fonction `int nb_codes(liste)` qui calcule le nombre de codes différents apparaissant dans une liste.
4. Écrire une fonction `liste* tableau(liste l, int taille_tab)` qui transforme la liste chaînée précédente en un tableau de listes chaînées dont chaque case contient la liste chaînée de **tous** les mots correspondant à un **même** code (cf figure 1).

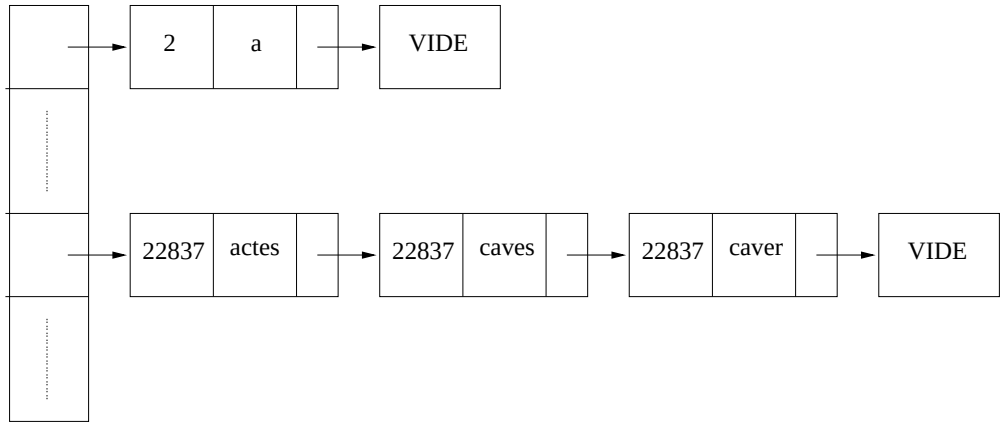


FIG. 1 – Exemple de tableau contenant des listes chaînées.

5. Écrire une fonction `liste dicho(liste*, int)` qui trouve tous les mots correspondant à un code donné en effectuant une recherche dichotomique[1] de ce code dans le tableau de listes chaînées. Afficher alors tous les mots correspondant à ce code.

# [OCaml] Écriture prédictive

Étant donnée une chaîne de caractères, l'écriture prédictive consiste à déterminer tous les mots (pris dans un dictionnaire) dont la chaîne est un préfixe et à les présenter à l'utilisateur sous la forme d'une liste dans laquelle il pourra sélectionner le mot souhaité.

On représentera les mots sous la forme d'un arbre (cf figure 2) où chaque lettre correspond à un nœud. Il faut en plus pouvoir déterminer la lettre sur laquelle un mot se termine. En effet, il faut pouvoir reconnaître, par exemple, que *moi* est un mot mais pas *moin*. Il est donc nécessaire d'ajouter un moyen de distinguer la fin d'un mot. On pourra, par exemple, ajouter un booléen à chaque nœud de l'arbre valant `true` si la lettre correspondant à ce nœud constitue bien la dernière d'un mot depuis la racine de l'arbre.

[1] L'idée de base d'une recherche dichotomique est de comparer la valeur recherchée avec celle du milieu du tableau. Si la valeur est plus petite, on continue la recherche dans la première moitié du tableau ; si elle est plus grande, on continue dans la seconde. Indication : on peut facilement réaliser cette recherche grâce à une fonction récursive prenant en arguments les bornes du sous-tableau considéré.

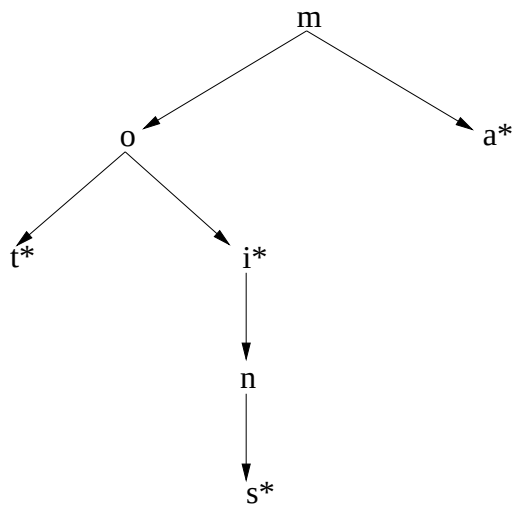


FIG. 2 – Représentation des mots *mot*, *moi moins* et *ma*

Ainsi, après avoir représenté tous les mots du dictionnaire, pour obtenir tous les mots accessibles à partir d'un préfixe donné, il suffira de trouver le sous-arbre correspondant à ce préfixe.

1. Définir le type `arbre` permettant de représenter tous les mots commençant par une même lettre. À chaque nœud est associé une lettre, un booléen indiquant si le chemin depuis la racine forme un mot et le sous-arbre de ses suffixes, représenté par une liste d'arbres. Une feuille de l'arbre (dernière lettre d'un mot qui n'est le préfixe d'aucun autre mot) aura donc une liste de fils vide.
2. Écrire une fonction `arbre_de_mot : string -> arbre` qui transforme un mot en arbre.
   **Indication** : il existe un grand nombre de fonctions prédéfinies dans le module `String` qui permettent de manipuler facilement les chaînes de caractères. Par exemple `String.length` renvoie la longueur d'un mot etc...
3. Écrire une fonction `lecture : string -> arbre list` de lecture du dictionnaire qui stocke chaque mot commençant par la même lettre dans un même arbre. Étant donné qu'il y a au moins un mot commençant par chacune des lettres de l'alphabet, vous obtiendrez 26 arbres.
   **Indication** : Une solution peut être de parcourir l'arbre jusqu'à trouver l'endroit où il faut ajouter le sous arbre correspondant puis d'utiliser la fonction définie ci-dessus. Par exemple, si le mot *ma* existe déjà et que l'on souhaite ajouter dans l'arbre le mot *maman* il suffit d'ajouter le suffixe *man* (que l'on peut obtenir grâce à la fonction `String.sub`) au bon endroit, c'est-à-dire d'insérer le sous-arbre correspondant au suffixe dans l'arbre.
4. Écrire une fonction `sous_arbre : string -> arbre list -> arbre` qui renvoie le sous-arbre correspondant à un préfixe donné.
5. Écrire une fonction `affiche : arbre -> unit` qui affiche sur la sortie standard les mots contenus dans un arbre.
6. Écrire une fonction qui affiche, pour un préfixe donné, tout les mots accessibles dans le dictionnaire qui ont le même préfixe.