

# TD10

Programmation en C (LC4)

Semaine du 1 avril 2008

## 1 Listes

On commence par travailler avec le type de listes suivant :

```
struct Liste {
    int valeur ;
    struct Liste * suivant ;
};
```

**Exercice 1** Écrire une fonction

```
void decoupe (struct Liste * l, struct Liste ** l1, struct Liste ** l2)
```

qui prend une liste  $l$  en argument, et la découpe en deux listes :

- la liste formé du premier maillon de  $l$ , puis du troisième, puis du cinquième, ...
- la liste formé du deuxième maillon de  $l$ , puis du quatrième, puis du sixième, ... Le pointeur vers le premier maillon de la première (respectivement seconde) liste devra être écrit dans  $*l1$  (respectivement  $*l2$ ).

**Exercice 2** Écrire une fonction `void filtre(struct Liste * l)` qui efface le deuxième maillon de la liste, puis le quatrième, puis le sixième, ...

## 2 Arbres

On travaille avec le type :

```
struct arbre {
    int valeur ;
    struct arbre * gauche ;
    struct arbre * droite ;
};
```

**Exercice 3** Écrire une fonction qui teste si deux arbres ont le même squelette.

**Exercice 4** Écrire une fonction qui teste si un arbre  $f$  est un préfixe d'un arbre  $g$ , c'est-à-dire si l'on peut obtenir  $f$  à partir de  $g$  en remplaçant certains sous-arbres par des arbres vides.

**Exercice 5** Comment définir un type d'arbre où chaque nœud peut avoir un nombre variable de fils ? Écrire, pour ce type une fonction calculant l'arité maximale d'un nœud.

### 3 Pointeurs vers des fonctions

**Exercice 6** Quel est le type d'un pointeur vers une fonction prenant un int en argument et renvoyant un int en résultat ?

**Exercice 7** Écrire une fonction `int pour_tout(struct Liste * l, int (*predicat)(int))` qui prend en argument une liste l, et un pointeur predicat vers une fonction, et qui détermine si la fonction appliquée à tous les éléments de la liste retourne une valeur non-nulle.

**Exercice 8** En utilisant la question précédente, écrire une fonction qui teste si tous les éléments d'une liste sont impairs.

On travaille désormais avec le type suivant :

```
struct Liste {
    void * valeur ;
    struct Liste * suivant ;
};
```

L'idée est que l'on met dans le champ valeur un pointeur vers un objet dont l'on a fait oublier le type au compilateur. Cela permet d'écrire du code travaillant sur des listes, indépendamment du type des objets manipulés.

**Exercice 9** Écrire une fonction

```
struct Liste* insere (struct Liste* l, void* x ,
                    int (* inferieur) (void* ,void*))
```

qui prend en argument une liste, un pointeur x vers un objet de type inconnu, et un pointeur inferieur vers une fonction qui est supposée implémenter une relation d'ordre pour laquelle la liste l est triée ; et qui insère x dans la liste, de sorte qu'elle soit toujours triée. Le résultat renvoyé est le maillon de liste créé pour x. Par exemple, si les objets stockés dans la liste sont des chaînes de caractères, inferieur pourrait pointer vers une telle fonction :

```
int compare_chaine (char* s, char* t) {return strcmp(s,t) >0;}
```