TD10

Programmation en C (LC4)

Semaine du 2 avril 2007

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**

```
void decoupe(struct Liste * l,struct Liste ** l1,struct Liste** l2){
  *l1=l;
  if (l) *l2=l->suivant; else *l2=NULL;
  while (l) {
    struct Liste * tmp=l->suivant;
    if (tmp) {
      l->suivant=tmp->suivant;
      l=tmp->suivant;
      if (tmp->suivant) tmp->suivant=tmp->suivant->suivant; else tmp->suivant=NULL;
    } else l=NULL;
  }
}
```

Exercice 3 Écrire une fonction `void filtre(struct Liste * l)` qui efface le deuxième maillon de la liste, puis le quatrième, puis le sixième, ...

▶ **Exercice 4**

```
void filtre(struct Liste * l) {
  struct Liste * tmp;
  while (l) {
    tmp=l->suivant;
```

```
    if (tmp) {
      l->suivant=tmp->suivant;
      free(tmp);
    }
    l=l->suivant;
  }
}
```

2 Arbres

On travaille avec le type :

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

Exercice 5 Écrire une fonction qui teste si deux arbres ont le même squelette.

▶ **Exercice 6**

```
int meme_squelette(struct arbre *f,struct arbre *g) {
  if (!f&&!g) return 1;
  else if (f&&g) {
    return meme_squelette(f->gauche,g->gauche)&&meme_squelette(f->droite,g->droite);
  } else return 0;
}
```

Exercice 7 Écrire une fonction qui teste si un arbre `f` est un prefixe 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 8**

```
int prefixe(struct arbre *f,struct arbre *g) {
  if (!f) return 1;
  if (!g) return 0;
  if (f->valeur!=g->valeur) return 0;
  return prefixe(f->gauche,g->gauche)&&prefixe(f->droite,g->droite);
}
```

Exercice 9 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.

▶ **Exercice 10**

```
struct Liste_Noeuds;
struct Arbre {
  int valeur;
  struct Liste_Noeuds * fils;
};
struct Liste_Noeuds {
  struct Arbre * noeud;
```

```
  struct Liste_Noeuds * suivant;
};

int arite_max(struct Arbre * a) {
  int res=0;
  int tmp;
  int n=0;
  struct Liste_Noeuds *l=a->fils;
  while (l) {
    n++;
    tmp=arite_max(l->noeud);
    if (tmp>res) res=tmp;
    l=l->suivant;
  }
  if (n>res) return n; else return res;
}
```

3 Pointeurs vers des fonctions

Exercice 11 Quel est le type d'un pointeur vers une fonction prenant un `int` en argument et renvoyant un `int` en résultat ?

▶ **Exercice 12**

```
int (*)(int)
```

Exercice 13 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 14**

```
int pour_tout(struct Liste * l,int (*predicat)(int)) {
  while (l) {
    if ((*predicat)(l->valeur)) l=l->suivant; else return 0;
  }
  return 1;
}
```

Exercice 15 En utilisant la question précédente, écrire une fonction qui teste si tous les éléments d'une liste sont impairs.

▶ **Exercice 16**

```
int est_pair(int x) {
  return x%2==0;
}
int tous_pairs(struct Liste * l) {
  return pour_tout(l,&est_pair);
}
```

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, indépendament du type des objets manipulés.

Exercice 17 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 résulat renvoyé est le maillon de liste créé pour `x`.

Par exemple, si les objets stockés dans la liste sont des chaines de caractères, `inferieur` pourrait pointer vers une telle fonction :

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

▶ **Exercice 18**

```
struct Liste * insere(struct Liste * l, void * x,int (*inferieur)(void *,void *)) {
  struct Liste ** pere=NULL;
  while (l&&(*inferieur)(l->valeur,x)) {
    pere=&(l->suivant);
    l=l->suivant;
  }
  struct Liste * res=malloc(sizeof(struct Liste));
  res->valeur=x;
  res->suivant=l;
  if (pere) *pere=res;
  return res;
}
```