MMFAI
Cours de programmation

TP numéro 3
semaine du 26 octobre 1998

Struct(ures) de données. Listes. Arbres.

1 Structures de données

En C il est possible pour l'utilisateur de définir des structures de données en utilisant la construction `struct` qui corresponde aux declarations de types enregistrement en CamlLight ou Pascal.

Par exemple,

```
struct date {
            int annee;
            int mois;
            int jour;
};
```

définit un type enregistrement `date` avec des champs `annee`, `mois` et `jour` de type `int`.

Ensuite, on peut déclarer des variables de ce nouveau type :

```
struct date d;
```

et la déclaration alloue directement la mémoire nécessaire pour la structure.
On peut acceder aux champs par l'opérateur de séléction `.` comme en CamlLight :

```
d.annee=1997;
d.mois=11;
d.jour=24;
printf("La date est %02d/%02d/%d\n",d.jour,d.mois,d.annee);
```

Enfin, on peut utiliser la construction `typedef` pour assigner un nouveau nom à tout type déjà existant, avec la même syntaxe que pour une déclaration `extern` ou `static`, i.e.

`typedef` *typepreexistant* `NOUVEAU(X)NOM(S)`

Par exemple :

```
typedef struct date DATE, *DATEPTR;
```

permet ensuite de déclarer

```
DATE d;       /* struct date d   */
DATEPTR pd;  /* struct date *pd */
```

On peut aussi tout faire en un seul coup, comme dans l'exemple du type list plus avant.
Notons enfin que quand on manipule des structures on le fait le plus souvent en passant aux fonctions des pointeurs sur des structures et non pas les structures elles mêmes (on n'a pas envie de recopier des structures à chaque coup !), donc on trouve souvent la notation syntaxique `pd->annee` qui est *exactement équivalente à* `(*pd).annee`.

2 Manipulation de listes

Nous souhaitons nous constituer ici une librairie de fonctions utiles pour travailler sur des listes, et là nous sommes déjà confrontés à un choix de conception important :
- doit-on prévoir de faire toutes les manipulations de façon *destructive* en *modifiant la structure en place* (on fera alors bien attention à prévoir une fonction pour cloner une structure) ?
- ou veut-on, au contraire, ne rien modifier en place, pour travailler comme en CamlLight (on devra alors faire attention à libérer tout le temps la mémoire qui devient inutilisée) ?

Ici, nous allons d'abord explorer la deuxième solution, en programmant notre bibliothèque selon un approche algebrique aux types de données developpée au debut des années 80, qui suggère de decomposer les fonctions qui manipulent une structure de données en differentes classes : "constructeurs", "destructeurs" et "testeurs" (des noms qui expliquent déjà tout).

Exercice 2.1 (Le type de donnée intlist) *Pour la structure de données* `intlist` *définie comme suit*

```
typedef struct intlist {
        int data;
        struct intlist *suivant;
} intlist;
```

définissez les fonctions suivantes :
Coonstructeurs :
- `intlist *nil()` *construit une "intliste vide"*
- `intlist *cons(int data, intlist *l)` *construit la intliste* `cons(data,l)` *(la concatenation de la donnée data en tête de la intliste l)*

Testeurs :
- `int isnil(intlist *l)` *renvoie 1 si la intliste est vide et 0 sinon*

Destructeurs :
- `int hd(intlist *l)` *renvoie le premier élément de la intliste* `l`
- `intlist *tl(intlist *l)` *renvoie le reste de la intliste* `a`

Exercice 2.2 (Append et reverse)
Écrivez une fonction `void printintlist(intlist *l)` *qui imprime les éléments d'une intliste. Ensuite, écrire une fonction* `intlist *append(intlist *l1, intlist *l2)` *qui concatene deux listes, et une fonction* `intlist *rev(intlist *l)` *qui renverse une intliste.*

Rémarquez que si vous vous imposez d'écrire ces fonctions en utilisant uniquement *les fonctions definies ci-dessus, vous êtes obligés de ne pas faire de la modification en place. Mais rien ne vous empêche d'essayer d'écrire une version avec modification en place, en accedant directement à la structure.*

Testez vos fonctions et reflechissez aux avantages et désavantages des deux approches (en place vs. récopie).

3 Manipulation d'arbres

Exercice 3.1 (Le type de donnée intarbre) *Pour la structure de données* `intarbre` *définie comme suit*

```
typedef struct intarbre {
               int data;
```

```
                struct intarbre *fg;
                struct intarbre *fd;
} intarbre, *iaptr;
```

définissez les fonctions suivantes :
Constructeurs :
- `iaptr empty()` *construit un "intarbre vide"*
- `iaptr node(int data, iaptr g, iaptr d)` *construit l'intarbre* `node(data,g,d)`

Testeurs :
- `int isempty(iaptr a)` *renvoie 1 si l'intarbre est vide et 0 sinon*

Destructeurs :
- `int data(iaptr a)` *renvoie la donnée dans la racine de* `a`
- `iaptr filsg(iaptr a)` *renvoie le sousarbre gauche de* `a`
- `iaptr filsd(iaptr a)` *renvoie le sousarbre droit de* `a`

Exercice 3.2 (Parcours d'arbres) *Écrivez une fonction* `intlist *infixintarbre(iaptr a)` *qui renvoie la liste des noeuds d'un intarbre dans l'ordre de la visite infixe.*
Écrivez ensuite une fonction `void printintarbre(iaptr a)` *qui imprime les éléments d'un intarbre d'entiers dans cet ordre (utilisez* `printintlist`*).*

Exercice 3.3 (Miroir d'un arbre) *Écrivez une fonction* `iaptr miroir(iaptr a)` *qui construit l'arbre miroir de* `a`*, en utilisant* uniquement *les fonctions definies ci-dessus.*
Testez votre fonction.

Exercice 3.4 (Modification en place) *Écrivez maintenant, en accedant directement à la structure, une fonction* `void echange(iaptr a, iaptr b)` *qui échange les arbres pointés par* `a` *et* `b`*.*
Reécrivez la fonction `miroir` *à l'aide de cette fonction. Commentez sur la différence des deux approches.*

N'oubliez pas de vous déconnecter avant d'abandonner le clavier (voir notes du stage Unix).