TD 3 : Retour sur les `struct` - Pointeurs

Langage C (LC4)
Semaine du 15 février 2010

1 Les nombres rationnels : en utilisant des pointeurs

Dans toute cette section, on utilisera le type **`int`** pour représenter les nombres entiers.
Le type `rational` est défini comme suit :

```
typedef struct {
  int numerator;
  int denominator;
} rational;
```

Question 1. Définir une fonction **`void`** `make_rational(int n, int d, rational * r)` qui étant donné deux **`int`** représentant des entiers n et d, crée le rationnel n/d, et une fonction **`void`** `print_rational(rational * r)` affichant à l'écran le rationnel r sous la forme n/d, **en utilisant une seule fois `printf`**.

Question 2. Écrire une fonction **`void`** `opp_rational(rational * rat, rational * opp)` qui calcule l'opposé d'un rationnel.

Question 3. Définir une fonction **`int`** `equal_rational(rational * r1, rational * r2)` qui renvoie 1 si les rationnels r_1 et r_2 sont égaux, et 0 sinon. (1 instruction)

Question 4. Définir une fonction **`void`** `sum_rational(rational * r1, rational * r2, rationnal * sum)` qui calcule la somme des deux rationnels donnés en argument (1 instruction).

2 Listes

Soit E un ensemble. Une liste d'éléments de E est un objet mathématique qui est soit une liste vide, soit une construction faite en ajoutant un élément de E (la tête) à une autre liste (la queue) :

$$l \quad ::= \quad \textbf{nil} \quad | \quad e :: l'$$

Si E est l'ensemble des entiers, partant de la liste vide et en ajoutant successivement en tête les valeurs 3, 5 et 1, on obtient la liste de taille 3 suivante :

$$1 :: 5 :: 3 :: \textbf{nil}$$

En C on peut manipuler des listes d'entiers grâce à la définition suivante :

```
typedef struct int_elem {
  int head;
  struct int_elem *tail;
} *int_list;
```

Une liste est un pointeur vers une structure `int_elem` qui contient le premier élément et un pointeur vers le reste de la liste. La liste vide est représentée par le pointeur `NULL`.
En pratique, `int_list` est donc un pointeur sur un élement de la liste.

2.1 Construction, affichage, destruction, copie

Question 5. Écrivez les fonctions :
- `int_list nil()` qui renvoie la liste vide
- `int_list cons(int head, int_list tail)` qui ajoute un élément `head` en tête d'une liste `tail`. Cette fonction utilise les éléments de la liste `tail` sans les copier, ils deviennent des éléments de la liste retournée.

Question 6. Écrivez la fonction `void print_list(int_list l)`, qui affiche une liste. La liste donnée en exemple plus haut sera affichée de la façon suivante :

```
[1,5,3]
```

Question 7. Écrivez la fonction `void free_list(int_list l)`, qui libère la totalité de la mémoire occupée par la liste `l`.

La liste donnée en exemple est construite grâce au code suivant :

```
int_list list0 = nil();
int_list list1 = cons(3, list0);
int_list list2 = cons(5, list1);
int_list list = cons(1, list2);
```

Question 8. Après avoir construit cette liste on souhaite libérer la mémoire qu'elle occupe. Parmi `l0`, `l1`, `l2` et `l`, sur quelle(s) variable(s) doit on appeler la fonction `free_list` ? Est-il judicieux de donner des noms de variables aux constructions intermédiaires, ou est-il préférable d'utiliser le code suivant ?

```
int_list l = cons(1, cons(5, cons(3, nil())));
```

Question 9. Écrivez la fonction `int_list copy_list(int_list l)`, qui produit une copie la liste `l`. (Indice : pour cette question, utiliser un appel récursif rend les choses plus façiles.)

Question 10. Donnez un exemple d'utilisation de cette fonction en construisant deux listes :
$4 :: 1 :: 5 :: 3 ::$ **nil**
$7 :: 1 :: 5 :: 3 ::$ **nil**
obtenues en ajoutant séparément 4 et 7 à la liste `l` précédente.

2.2 Manipulation des listes

Question 11. Écrivez les fonctions :
- `int empty(int_list l)` qui renvoie un booléen indiquant si la liste `l` est vide.

– **int** `size(int_list l)`, qui renvoie la taille de la liste `l`,
– **int** `equal(int_list list1, int_list list2)` qui renvoie un booléen indiquant si les deux listes `list1` et `list2` sont égales.

Question 12. Écrivez les fonctions :
– `int_list append(int_list list1, int_list list2)` qui renvoie la liste obtenue en regroupant sans changer l'ordre les éléments de la liste `list1` et ceux de la liste `list2`.
– `int_list reverse(int_list l)` qui renvoie la liste obtenue en renversant l'ordre des éléments de `l`.
Comme la fonction `cons`, ces fonctions consomment les éléments des listes qu'on leur donne en argument, ceux ci n'ont donc pas besoin d'être copiés.

2.3 Petits exercices

Question 13. Écrivez une fonction qui dédouble une liste, par exemple, $1 :: 5 :: 3 ::$ **nil** se dédouble en $1 :: 5 :: 3 :: 1 :: 5 :: 3 ::$ **nil**.

Question 14. Écrivez une fonction qui teste si une liste est un palindrome (c'est à dire égale à sa renversée).

2.4 Super bonus

Question 15. Écrivez la fonction `copy_list` sans utiliser la récursion.