Examen de programmation avancée

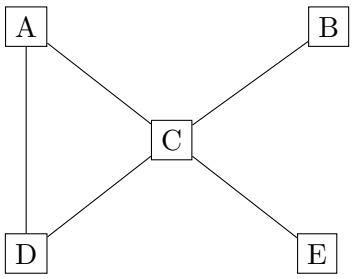
ENSIIE, semestre 2

mercredi 3 avril 2013

Durée : 1h45.
Tout document personnel autorisé (pas de prêt entre voisins). L'usage de la calculatrice ou de tout autre appareil électronique n'est pas autorisé.
Ce sujet comporte 4 exercices indépendants, qui peuvent être traités dans l'ordre voulu. Il contient 3 pages.
Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 20 points.

Exercice 1 : Compilation séparée (4 points)

On a choisi de découper un projet selon les modules suivants :



Un module m1 est relié par une arête vers le bas à un autre module m2 si m1 dépend de (l'interface de) m2. On suppose que chaque module sauf A et B a une interface explicite. On pourra considérer au choix que les modules sont des fichiers C ou OCaml.

1. Quelle commande faut-il écrire manuellement pour compiler le module A ? En OCaml, on supposera que les interfaces des modules sont déjà compilées.
2. En supposant que tous les modules sont compilés, quelle commande faut-il taper manuellement pour faire l'édition de liens ?
3. On modifie l'interface de E. Quel(s) module(s) faut-il recompiler ?
4. Écrire le `Makefile` correspondant au projet. On n'oubliera pas d'écrire une cible pour produire l'exécutable final qu'on appellera `prog`.

Exercice 2 : Représentation d'arbres à k-fils (8 points)

Dans un arbre k-aire, chaque nœud peut avoir de 0 à k fils. Pour manipuler de tels arbres, deux représentations sont utilisées en général :

- Dans la représentation standard, chaque nœud possède un tableau de taille k pour stocker les fils, plus une variable `rang` qui indique le nombre effectif de fils pour ce nœud.
- Dans la représentation fils gauche/frère droit (FGFD), chaque nœud possède un lien `fils_gauche` vers son fils le plus à gauche, ainsi qu'un lien `frere_droit` qui pointe vers son frère droit s'il existe.

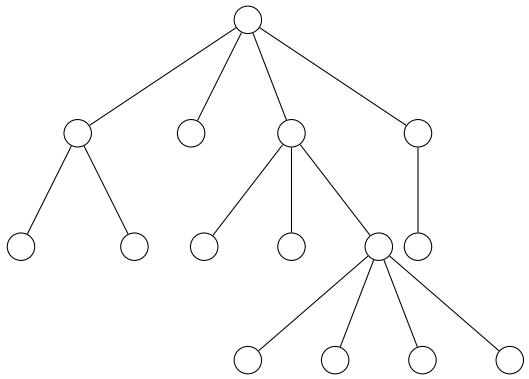
1. Définir le type des arbres k-aires pour chacune des deux représentations, au choix en C ou en OCaml.

   Pour la représentation FGFD en C, on emploiera des pointeurs pour `fils_gauche` et `frere_droit`, en utilisant le pointeur NULL dans le cas où ils ne sont pas présents.

   Pour la représentation FGFD en OCaml, on utilisera le type `option` pour `fils_gauche` et `frere_droit`, avec

   `type 'a option = None | Some of 'a`

   (Si le nœud n'existe pas, on utilise `None`, si c'est le nœud `n` on utilise `Some n`.)
2. Donner une représentation en mémoire de l'arbre 4-aire suivant :



   pour chacune des représentations.
3. Pour chacune des deux représentations, écrire des fonctions :
   a) `nieme_fils` qui prend en argument un arbre `a` et un entier `i` et qui retourne le `i`$^{\text{e}}$ fils de la racine de `a` ;
   b) `ajoute_fils_a_gauche` qui prend en argument deux arbres `a` et `b` et qui ajoute `b` comme sous-arbre le plus à gauche de la racine de `a` ;
   c) `ajoute_fils_a_droite` qui prend en argument deux arbres `a` et `b` et qui ajoute `b` comme sous-arbre le plus à droite de la racine de `a` ;
   d) `supprimer_fils` qui prend en argument un arbre `a` et un entier `i` et qui supprime le `i`$^{\text{e}}$ fils de la racine de `a`.
4. Pour chacune des deux représentations, donner la complexité dans le pire des cas de chacune des fonctions ci-dessus, en fonction, le cas échéant, de l'argument entier `i`, et en fonction du nombre de sous-arbres de la racine de `a`.
5. Quels avantages et inconvénients ont les deux représentations l'une par rapport à l'autre (en terme de complexité, de mémoire utilisée, etc.) ?

Exercice 3 : Garbage collector et partage maximal (5 points)

On considère le programme OCaml suivant :

```
let l = 1 :: [] in
let q = 1 :: [] in
let l =
  let h = 2 :: 3 :: 4 :: q in
  List.tl h
in
```

```
let q = 5 :: 4 :: q in
  (* ici *) l, q
```

1. Représenter l'état de la mémoire au point indiqué par le commentaire `(* ici *)`. Quels sont les parties accessibles depuis le programme ?
2. Appliquer un algorithme de garbage collection de type « marquer et nettoyer ».
3. Appliquer un algorithme de garbage collection copiant.

On veut maintenant faire du partage maximal sur les listes, en utilisant la technique de hash-consing.

4. Représenter les listes `l` et `q` au point `(* ici *)` avec partage maximal.
5. Écrire les constructeurs intelligents `nil : 'a list` et `cons : 'a -> 'a list -> 'a list` qui construisent les listes par hashconsing.
6. Réécrire le programme ci-dessus avec ces constructeurs intelligents.
7. Représenter le déroulement en mémoire du programme avec les constructeurs intelligents, en incluant la table de hachage servant au hashconsing.

Exercice 4 : Fonctions de hachage (3 points)

On veut écrire des fonctions de hachage pour des chaînes de caractères. On cherche donc des fonctions qui prennent en entrée une suite finie de caractères et qui retournent un entier compris en 0 et $m - 1$.

Une méthode générale est de combiner itérativement la valeur de chaque caractère (vu comme un entier) avec un accumulateur :

```
hash_accu := 0
pour i allant de 0 à la longueur de s - 1
  hash_accu := combine(hash_accu, s[i])
retourner hash_accu modulo m
```

Dans la suite, on considérera que $m = 256$.

1. On utilise le "ou exclusif" comme fonction pour combiner : `combine(x, y) = x xor y`. Quelle est la valeur de hachage de la chaîne `"chien"` ? Pour rappel, la représentation en binaire des caractères est la suivante : 'c' = 01100011, 'h' = 01101000, 'i' = 01101001, 'e' = 01100101, 'n' = 01101110.
2. Quelle est la valeur de hachage de la chaîne `"niche"` ?
3. Plus généralement, que se passe-t-il pour les anagrammes ?
4. Par conséquent, quelle propriété mathématique la fonction `combine` doit-elle ne pas vérifier pour éviter les collisions ?
5. Que penser de la fonction `combine(x, y) = x * 19 + y` ?
6. Proposer une méthode pour obtenir une fonction de hachage sur les arbres binaires d'entiers.