TP numéro 2
Modules, Makefile

Programmation avancée, ENSIIE

Semestre 2, 2013–14

On reprend l'exemple des arbres binaires vus dans les TD et TP précédents. On souhaite utiliser une structure d'ensembles pour stocker les valeurs contenues dans les feuilles. On aura donc trois modules : le module `Ab` des arbres binaires comme vu en TP ; un module `Ensemble` définissant les ensembles d'entiers ; et un module `Main` où on écrira une fonction `ensemble_feuilles` qui retourne l'ensemble de valeurs contenues dans les feuilles, ainsi qu'une fonction principale de test qui crée un arbre, calcule l'ensemble de ses feuilles et l'affiche. Pour ceux qui ne l'aurait pas terminé à la séance précédente, le module `Ab` est disponible depuis la page : `http://www.ensiie.fr/~guillaume.burel/cours/IAP3/` Vous pourrez réaliser ce TP au choix en C ou en OCaml.

1. Quelles sont les relations de dépendance entre les modules ?
2. Écrire un premier `Makefile` correspondant à ce projet, pour C ou pour OCaml, sans utiliser de règles avec filtrage. L'exécutable final sera appelé `prog`.
3. Réécrire le `Makefile` pour y ajouter des règles avec filtrage.

   Remarque : dans GNU make, la règle suivante est ajoutée implicitement, il n'est donc pas nécessaire de le faire.

   ```
   %.o: %.c
           $(CC) -c $(CPPFLAGS) $(CFLAGS) $<
   ```

4. Écrire l'interface du module `Ensemble` (`ensemble.h` ou `ensemble.mli`) qui déclarera le type abstrait des ensembles d'entiers et des fonctions :
   - `empty` pour créer un ensemble vide ;
   - `mem` pour tester la présence d'un élément dans un ensemble ;
   - `add` pour ajouter un élément dans un ensemble ;
   - `uni` pour faire l'union de deux ensembles ;
   - `iter` pour appliquer une procédure à tous les éléments d'un ensemble.
5. Écrire le module `Main` (`main.c` ou `main.ml`).
6. Donner une première implémentation du module `Ensemble` (`ensemble.c` ou `ensemble.ml`) à l'aide de listes sans doublons.
7. Compiler (à l'aide de `make`), tester.
8. Dans le module `Main`, écrire une fonction `arbre_aleatoire` qui prend un entier `h` en argument et qui renvoie un arbre complet de hauteur `h` dont les valeurs des nœuds sont aléatoires. Pour générer les valeurs des nœuds, en C, on utilisera la fonction `rand` de `stdlib.h`, en OCaml on utilisera `Random.bits`.

9. Réécrire votre fonction principale pour que votre programme prenne un entier h en argument, construise un arbre aléatoire de hauteur h et calcule l'ensemble de ses feuilles.

   Rappel : pour accéder aux arguments de la ligne de commande en C, on définit la fonction `main` avec deux arguments `int argc, char **argv`. `argc` est le nombre d'arguments plus 1, `argv` est le tableau des arguments en tant que chaînes de caractères (le premier des éléments du tableau étant le nom du programme).

   On OCaml, on peut utiliser le tableau `Sys.argv`.

10. Donner une deuxième implémentation du module `Ensemble` à l'aide d'arbres AVL.

    En C : on créera un fichier `ensemble_avl.c` dans lequel il y aura un `#include "ensemble.h"`, et on ajoutera dans le `Makefile`, en plus des dépendances adéquates, une cible `prog_avl` qui sera produite en combinant `ab.o`, `ensemble_avl.o` et `main.o`.

    En OCaml : on copiera le fichier `ensemble.mli` en un fichier `ensemble_avl.mli`, et le fichier `main.ml` en un fichier `main_avl.ml` dans lequel on remplacera `Ensemble` par `Ensemble_avl`. On ajoutera dans le `Makefile`, en plus des dépendances adéquates, une cible `prog_avl` qui sera produite en combinant `ab.cmo`, `ensemble_avl.cmo` et `main_avl.cmo`.

11. Comparer les deux implémentations des ensembles pour des valeurs croissantes de h. On pourra utiliser la commande shell `time` pour mesurer le temps mis par le programme. Justifier les différences observées par rapport à la complexité des algorithmes utilisés.

S'il vous reste du temps et que vous avez choisi OCaml, compiler vos programmes à l'aide d'`ocamlopt` au lieu d'`ocamlc` et comparer les temps d'exécution.