Corrigé de l'examen de programmation avancée

ENSIIE, semestre 2

mercredi 30 mars 2011

Exercice 1 : Arbres (4 points)

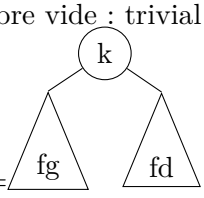
1. En OCaml :

```
let rec miroir a = match a with
    Vide -> Vide
  | Noeud(fg, i, fd) -> Noeud(miroir fd, i , miroir fg)
```

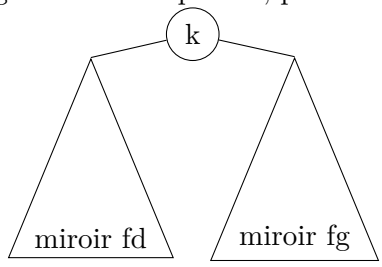
En C :

```
arbre miroir(arbre a) {
  arbre tmp;
  if (a == NULL) return NULL;
  tmp = a->fg;
  a->fg = miroir(a->fd);
  a->fd = miroir(tmp);
  return a;
}
```

Si on note C_n la complexité de la fonction `miroir`, on a $C_n = C_{n_1} + C_{n_2} + k$ avec $n = n_1 + n_2 + 1$ et k une constante correspondant au nombre d'opérations différentes des appels récursifs dans le corps de `miroir`. On peut montrer que $C_n = k \times n$ par récurrence sur n (en supposant abusivement que $C_0 = 0$).

3. Par induction sur les arbres :
   - Arbre vide : trivial
   - $a =$ 

: le parcours préfixe de a visite les nœuds en commençant par k, puis les nœuds de fg dans l'ordre préfixe, puis les nœuds de fd dans l'ordre préfixe.

   Le miroir de a vaut 

Le parcours postfixe du miroir de a visite les nœuds du miroir de fd dans l'ordre postfixe, puis ceux du miroir de fg dans l'ordre postfixe, puis k.
Par hypothèse d'induction, cela revient à visiter les nœuds de fd dans l'ordre inverse du parcours préfixe, puis ceux de fg dans l'ordre inverse du parcours préfixe, puis k. CQFD.

Exercice 2 : Modularité (3 points)

1. Chaque module peut être développé indépendamment une fois que l'interface a été fixée, on peut donc séparer les tâches via des modules différents.
2. Un module créé dans un projet peut être réutilisé dans un autre projet sans avoir besoin de tout reprendre.
3. On peut modifier un module sans avoir à modifier l'ensemble du code, du moment que son interface ne change pas.

Exercice 3 : Makefile (3 points)

En OCaml

```
OCAMLC=ocamlc

%.cmo:%.ml
	$(OCAMLC) -c -o $@ $<
%.cmi:%.mli
	$(OCAMLC) -c -o $@ $<

main.cmo: parseur.cmi evaluation.cmi affichage.cmi main.cmi
parseur.cmo: lecture_de_fichiers.cmi expressions.cmi parseur.cmi
evaluation.cmo: expressions.cmi environnement.cmi evaluation.cmi
affichage.cmo: expressions.cmi affichage.cmi
lecture_de_fichiers.cmo: lecture_de_fichiers.cmi
expressions.cmo: expressions.cmi
environnement.cmo: environnement.cmi

OBJ=environnement.cmo expressions.cmo lecture_de_fichiers.cmo affichage.cmo \
    evaluation.cmo parseur.cmo main.cmo

prog: $(OBJ)
	$(OCAMLC) -o $@ $(OBJ)
```

En C

```
main.o: parseur.h evaluation.h affichage.h main.h
parseur.o: lecture_de_fichiers.h expressions.h parseur.h
evaluation.o: expressions.h environnement.h evaluation.h
affichage.o: expressions.h affichage.h
```

```
lecture_de_fichiers.o: lecture_de_fichiers.h
expressions.o: expressions.h
environnement.o: environnement.h

OBJ=environnement.o expressions.o lecture_de_fichiers.o affichage.o \
    evaluation.o parseur.o main.o

prog: $(OBJ)
        $(CC) -o $@ $(OBJ)
```

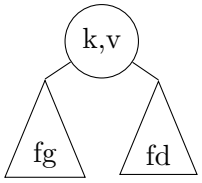
Exercice 4 : Arbres binaires de recherche (6 points)

1. On parcours l'arbre en allant dans le fils gauche tant qu'il n'est pas vide, on renvoie la valeur dans le nœud qui n'a pas de fils gauche.
2. En OCaml

   ```
   let rec plus_petite a = match a with
       Nil -> failwith "hypothèse non vérifiée"
     | Bin(Nil, (_,v), _) -> v
     | Bin(fg, _, _) -> plus_petite fg
   ```

   En C

   ```
   value plus_petite(dict d) {
     while(d->fg != NULL) d = d->fg;
     return d->val;
   }
   ```

3. Par induction sur l'arbre :
   – Arbre vide : ok car on ne montre la correction que pour un arbre non vide
   – 

   : deux cas :
     – si fg est vide, d'après l'invariant des ABR, les clefs de fd sont plus grandes que k, donc la plus petite clefs de l'arbre est bien k et est associée à v.
     – si fg n'est pas vide, d'après l'invariant des ABR, les clefs de fg sont plus petites que k et celles de fd plus grandes que k, donc la plus petite se trouve dans fg. Comme fg n'est pas vide, on peut appliquer l'hypothèse d'induction à fg, `plus_petite(fg)` renvoie la valeur associée à la plus petite clef de fg qui est également la plus petite clef de l'arbre en entier.
4. La complexité de l'algorithme est en $O(h)$ où h est la hauteur du nœud contenant la plus petite clef. Dans le cas des arbres binaires simples, on a donc une complexité dans le pire des cas en $O(n)$ (arbre déséquilibré en forme de peigne vers la gauche) et en moyenne en $O(\log n)$.
5. Les arbres AVL sont toujours bien équilibrés, donc la complexité est en $O(\log n)$ en moyenne comme dans le pire des cas.

Exercice 5 : Choix de structures de données (4 points)

1. Ici, c'est le temps en moyenne de la recherche qui importe. Les tables de hachage ont une complexité en $O(1)$ contrairement aux ABR en $O(\log n)$ et aux listes d'association en $O(n)$, il faut donc choisir les tables de hachage.
2. Ici, c'est la complexité des trois opérations dans le pire des cas qui importe. Les listes d'associations, les tables de hachage et les ABR simples ont une complexité de la recherche et de la suppression dans le pire des cas en $O(n)$, tandis que cette complexité est en $O(\log n)$ pour les arbres AVL. On choisit donc des arbres AVL.
3. On définit une structure de donnée similaire à une table de hachage mais en utilisant des arbres AVL au lieu de listes d'association en cas de collision. En moyenne, en supposant la fonction de hachage uniforme, on a toujours une complexité en $O(1)$. Dans le pire des cas, on n'a que des collisions, ce qui donne une complexité égale à une constante plus celle dans un arbre AVL, ce qui donne une complexité en $O(\log n)$.