# Corrigé de l'examen de programmation avancée

**ENSIIE, semestre 2**

mercredi 1er avril 2015

## Exercice 1 : Modularité et compilation séparée (7 points)

1. Le module `Tree` dépend de l'interface de `Tree`.
   Le module `Parser` dépend de l'interface de `Tree`.
   Le module `Parser` dépend de l'interface de `Parser`.
   Le module `Lexer` dépend de l'interface de `Parser`.
   Le module `Lexer` dépend de l'interface de `Lexer`.
   Le module `Main` dépend de l'interface de `Tree`.
   Le module `Main` dépend de l'interface de `Parser`.
   Le module `Main` dépend de l'interface de `Lexer`.
2. En C
   `gcc -Wall -ansi -c lexer.c`
   En Ocaml
   `ocamlc -c lexer.ml`
3. En C
   `gcc -Wall -ansi -o prog tree.o parser.o lexer.o main.o`
   En Ocaml
   `ocamlc -o prog tree.cmo parser.cmo lexer.cmo main.cmo`
4. Il faut recompiler les modules qui dépendent de l'interface de `Tree`, donc `Tree`, `Parser` et `Main` mais pas `Lexer`.
5. En C

```
CC=gcc -Wall -ansi

tree.o: tree.h
parser.o: tree.h parser.h
lexer.o: parser.h lexer.h
main.o: tree.h parser.h lexer.h

prog_c: tree.o parser.o lexer.o main.o
	$(CC) -o $@ $^
```

   En OCaml

```
%.cmo: %.ml
        ocamlc -c $<
%.cmi: %.mli
        ocamlc -c $<

tree.cmo: tree.cmi
parser.cmo: tree.cmi parser.cmi
lexer.cmo: parser.cmi lexer.cmi
main.cmo: tree.cmi parser.cmi lexer.cmi

prog: tree.cmo parser.cmo lexer.cmo main.cmo
        ocamlc -o $@ $^
```

6. En C : parser.h

```
typedef struct token_base* token;

token lpar();
token rpar();
token word(char*);
tree parse(char*, token (*) (char*));
```

lexer.h

```
token split(char*);
```

tree.h

```
typedef struct tree_base* tree;
tree empty();
tree node(char*, tree, tree);
void print_tree(tree);
```

En OCaml : parser.mli

```
type token
val lpar : token
val rpar : token
val word : string -> token
val parse : string -> (string -> token) -> Tree.tree
```

lexer.mli

```
val split : string -> Parser.token
```

tree.mli

```
type tree
val empty : tree
val node : string -> tree -> tree -> tree
val print_tree : tree -> unit
```

7. En C : `main.c`

```
#include "tree.h"
#include "parser.h"
#include "lexer.h"

int main(int argc, char**argv) {
  tree res = parse(argv[1], split);
  print_tree(res);
  return 0;
}
```

En OCaml : `main.ml`

```
let _ =
  let res = Parser.parse Sys.argv.(1) Lexer.split in
  Tree.print_tree res
```

## Exercice 2 : Bûcheron (ou comment hacher des arbres) (4 points)

1. a) 1 b) 1 c) 2
2. En C :

```
int tree_sum(ab a) {
  if (a)
    return (a->val + tree_sum(a->fg) + tree_sum(a->fd));
  else
    return 0;
}

int tree_hash(ab a, int m) {
   return (tree_sum(a) % m);
}
```

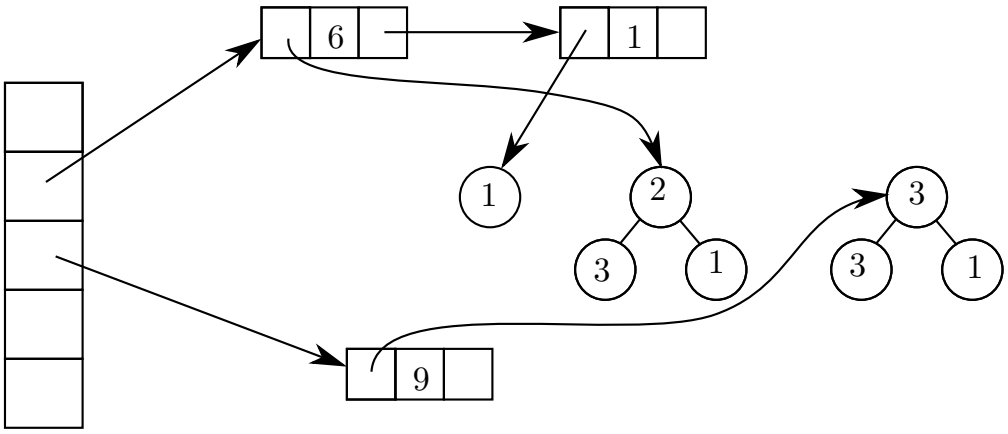
En OCaml :

```
let rec tree_sum a =
  match a with
```

```
    Vide -> 0
  | Noeud(fg,v,fd) -> v + tree_sum fg + tree_sum fd

let tree_hash a m =
  tree_sum a mod m
```

3. La complexité de la fonction `tree_sum` vérifie la relation de récurrence $C(n) = C(n_1) + C(n_2) + 1$ où $n$, $n_1$ et $n_2$ sont les nombres de noeud respectivement dans l'arbre `a`, son fils gauche et son fils droit. Par conséquent, la complexité de `tree_sum` est en $O(n)$, et celle de `tree_hash` aussi.

4. 


## Exercice 3 : Tri par tas (9 points)

1. On montre par induction la propriété $P(a)$ : « si $a$ est ordonné en tas et est non vide, alors la valeur maximale contenue dans $a$ est à la racine. »

   Montrons $P(\texttt{Vide})$ : trivial.

   Supposons $P(fg)$ et $P(fd)$, montrons $P(\texttt{Noeud}(fg, v, fd))$.

   Supposons que $\texttt{Noeud}(fg, v, fd)$ est ordonné en tas.

   Montrons que $v$ est plus grande que toutes les valeurs des nœuds de $fg$ et de $fd$.

   Soit $x$ un noeud de l'arbre. Sans perte de généralité, supposons que $x$ est dans $fg$. (L'autre cas est symétrique.)

   Puisque $x$ est dans $fg$, $fg$ n'est pas vide. Par ailleurs, par définition d'être ordonné en tas qui parle de chaque nœud de l'arbre, le sous-arbre $fg$ est aussi ordonné en tas.

   En appliquant l'hypothèse de récurrence sur $fg$ on sait donc que la valeur maximale de $fg$ est à la racine de $fg$, qui est donc plus grande que celle en $x$.

   Par définition d'être ordonné en tas, $v$ est plus grande que la valeur à la racine de $fg$, elle est donc plus grande que celle en $x$.

   CQFD.

2. En C :

```c
int est_ordonne_tas(ab a) {
  if (!a) return 1;
  if (a->fg && a->fg->val > a->val) return 0;
  if (a->fd && a->fd->val > a->val) return 0;
  return (est_ordonne_tas(a->fg) && est_ordonne_tas(a->fd));
}
```

En OCaml :

```
let rec est_ordonne_tas a =
  match a with
    Vide -> true
  | Noeud(fg,v,fd) ->
    match fg,fd with
    | Vide, Vide -> true
    | Vide, Noeud(_,vd,_) -> vd <= v && est_ordonne_tas fd
    | Noeud(_,vg,_), Vide -> vg <= v && est_ordonne_tas fg
    | Noeud(_,vg,_), Noeud(_,vd,_) ->
      vg <= v &&
      vd <= v &&
      est_ordonne_tas fg &&
      est_ordonne_tas fd
```

3. La complexité de la fonction `est_ordonne_tas` vérifie la relation de récurrence $C(n) = C(n_1) + C(n_2) + k$ où $n$, $n_1$ et $n_2$ sont les nombres de noeud respectivement dans l'arbre `a`, son fils gauche et son fils droit, et $k$ est une constante. Par conséquent, la complexité de `est_ordonne_tas` est en $O(n)$.

4. En C :

```c
int est_tas_aux(int* t, int i, int l) {
  if (i >= l) return 1;
  if (2*i+1 < l && t[i] < t[2*i+1]) return 0;
  if (2*i+2 < l && t[i] < t[2*i+2]) return 0;
  return (est_tas_aux(t, 2*i+1, l) && est_tas_aux(t, 2*i+2, l));
}

int est_tas(int* t, int l) {
  return est_tas_aux(t, 0, l);
}
```

En OCaml :

```
let rec est_tas_aux t i l =
  if i >= l then true
  else if 2*i+1 < l && t.(i) < t.(2*i+1) then false
```

```
  else if 2*i+2 < l && t.(i) < t.(2*i+2) then false
  else est_tas_aux t (2*i+1) l && est_tas_aux t (2*i+2) l

let est_tas t l =
  est_tas_aux t 0 l
```

5. En C :

```
void tamiser(int* t, int l, int i) {
  if (2*i+1 >= l) /* Feuille ou au-dela */ return;
  if (2*i+1 == l - 1) /* Un seul fils a gauche */
    {
      if (t[i] < t[l-1])
        echange(t, i, l-1);
      return;
    };
  /* Noeud avec deux fils */
  if (t[2*i+1] < t[2*i+2]) {
    /* fils droit plus grand que le gauche */
    if (t[i] < t[2*i+2]) {
      echange(t, i, 2*i+2);
      tamiser(t, l, 2*i+2);
    }; }
  else {
    /* fils gauche plus grand que le droit */
    if (t[i] < t[2*i+1]) {
      echange(t, i, 2*i+1);
      tamiser(t, l, 2*i+1);
    }; };
}
```

En OCaml :

```
let rec tamiser t l r =
  if 2*r+2 < l then (* deux fils *)
    if t.(2*r+2) > t.(2*r+1) then begin
      (* fils droit plus grand que fils gauche *)
      if t.(2*r+2) > t.(r) then
        begin
          echange t r (2*r+2);
          tamiser t l (2*r+2)
        end end
    else begin
      (* fils gauche plus grand que fils droit *)
      if t.(2*r+1) > t.(r) then
```

```
        begin
          echange t r (2*r+1);
          tamiser t l (2*r+1)
        end end
  else if 2*r+1 < l then (* un seul fils a gauche *)
    if t.(2*r+1) > t.(r) then begin
      echange t r (2*r+1);
      tamiser t l (2*r+1)
    end
(* on ne fait rien dans les autres cas *)
(* (feuille ou en dehors de l'arbre)   *)
```

6. Informellement le pire des cas est quand il faut faire descendre la valeur de la racine à une des feuilles. Soit $h$ la hauteur de l'arbre représenté par le tableau, la complexité dans le pire des cas satisfait la relation de récurrence $C(h) = C(h-1)+1$. (On ne s'applique récursivement que sur un des sous-arbres.) Or la hauteur $h$ de l'arbre est égale à $\log_2(n)$ où $n$ est la longueur du tableau. La complexité est donc en $O(\log(n))$.
7. Le nœud le plus à droite pour lequel il faut commencer à faire le tamisage est le père de la dernière case du tableau, celle à la position $n-1$. Ce père est à la position $\lfloor \frac{(n-1)-1}{2} \rfloor = \lfloor \frac{n-2}{2} \rfloor = \lfloor \frac{n}{2} - 1 \rfloor = \lfloor \frac{n}{2} \rfloor - 1$.
8. En C :

```
void ordonner_en_tas(int* t, int l) {
  int j;
  for (j = l/2 - 1; j >= 0; j--)
    tamiser(t, l, j);
}
```

   En OCaml :

```
let rec ordonner_en_tas_aux t l i =
  if i >= 0 then begin
    tamiser t l i;
    ordonner_en_tas_aux t l (i-1)
  end

let ordonner_en_tas t l =
  ordonner_en_tas_aux t l (l/2  - 1)
```

9. Dans le pire des cas, on fait $\frac{n}{2}$ tamisages, chacun ayant une complexité en $O(\log n)$, on a donc une complexité en $O(n \log(n))$.
10. En C :

```
void tri_par_tas_aux(int* t, int l) {
```

```
  if (l < 2) return;
  /* t est suppose ordonne en tas */
  echange(t, 0, l-1);
  tamiser(t, l - 1, 0);
  tri_par_tas_aux(t, l-1);
}

void tri_par_tas(int* t, int l) {
  ordonner_en_tas(t, l);
  tri_par_tas_aux(t, l);
}
```

En OCaml :

```
let rec tri_par_tas_aux t l =
  if l > 1 then begin
    (* t est suppose ordonne en tas *)
    echange t 0 (l-1);
    tamiser t (l-1) 0;
    tri_par_tas_aux t (l-1)
  end

let tri_par_tas t l =
  ordonner_en_tas t l;
  tri_par_tas_aux t l
```

11. On fait d'abord une fois `ordonner_en_tas`, qui a une complexité en $O(n\log(n))$, puis on fait $n-1$ fois un échange et un tamisage de complexité $O(1+\log(n))$. Au final, on a donc une complexité en $O(n\log(n))$ (la meilleure possible pour un tri de tableau).