Corrigé de l'examen de programmation avancée

ENSIIE, semestre 2

mercredi 3 avril 2013

Exercice 1 : Compilation séparée (4 points)

1. En C : `gcc -Wall -ansi -c A.c`
   En OCaml : `ocamlc -c A.ml`
2. En C : `gcc -Wall -ansi -o prog A.o B.o C.o D.o E.o`
   En OCaml : `ocamlc -o prog D.cmo E.cmo C.cmo A.cmo B.cmo`
3. Il faut recompiler E et C.
4. En C :

```
prog: A.o B.o C.o D.o E.o
        gcc -Wall -ansi -o $@ $^

A.o: C.h D.h
B.o: C.h
C.o: C.h D.h E.h
D.o: D.h
E.o: E.h
```

   En OCaml :

```
prog: D.cmo E.cmo C.cmo A.cmo B.cmo
        ocamlc -o $@ $^

A.cmo: C.cmi D.cmi
B.cmo: C.cmi
C.cmo: C.cmi D.cmi E.cmi
D.cmo: D.cmi
E.cmo: E.cmi

%.cmo: %.ml
        ocamlc -c $<

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

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

1. En C, représentation standard :

```
struct arbre {
  int rang;
  struct arbre* fils[k];
};
```

En C, représentation FGFD :

```
struct arbre {
  struct arbre* fils_gauche;
  struct arbre* frere_droit;
};
```

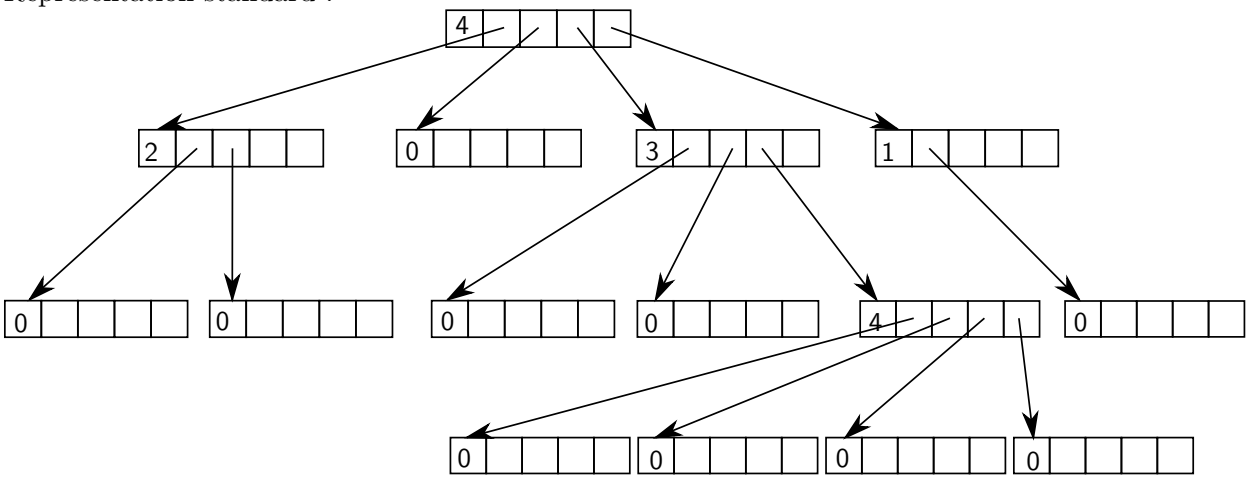
En OCaml, représentation standard :

```
type arbre = Noeud of int * (arbre array)
```

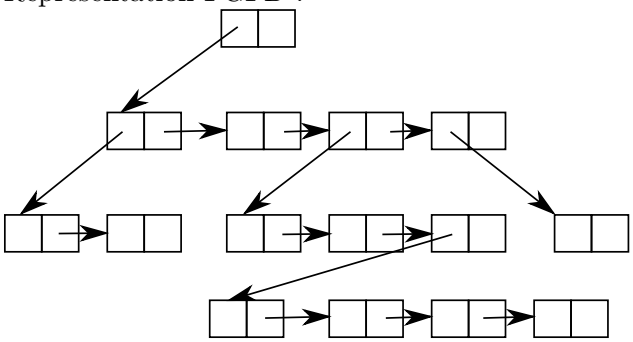
En OCaml, représentation FGFD :

```
type arbre = Noeud of (arbre option) * (arbre option)
```

2. Représentation standard :



Représentation FGFD :



3. a) En C, représentation standard :

```
struct arbre nieme_fils (struct arbre a, int i) {
  return *(a.fils[i]);
}
```

En C, représentation FGFD :

```
struct arbre nieme_fils (struct arbre a, int i) {
  struct arbre* res = a.fils_gauche;
  while (i > 0) {
    res = res->frere_droit;
    i--;
  };
  return *res;
}
```

En OCaml, représentation standard :

```
let nieme_fils a i =
  match a with
    Noeud (r, t) -> t.(i)
```

En OCaml, représentation FGFD :

```
let rec nieme_frere a i =
  if i = 0 then a
  else match a with
    Noeud(_, Some b) -> nieme_frere b (i-1)
  | Noeud(_, None) -> failwith "pas assez de frere"

let nieme_fils a i =
  match a with
    Noeud(Some f, _) -> nieme_frere f i
  | Noeud(None, _) -> failwith "pas de fils"
```

b) En C, représentation standard :

```
struct arbre ajoute_fils_a_gauche(struct arbre a, struct arbre b) {
  if (a.rang >= k) return a;
  a.rang++;
  int i;
  for (i = a.rang - 1; i > 0; i--) {
    a.fils[i] = a.fils[i-1];
  };
  a.fils[0] = &b;
  return a;
}
```

En C, représentation FGFD :

```
struct arbre ajoute_fils_a_gauche(struct arbre a, struct arbre b) {
  b.frere_droit = a.fils_gauche;
  a.fils_gauche = &b;
  return a;
}
```

En OCaml, représentation standard :

```
let rec decale_tableau t n =
  if n > 0 then begin
    t.(n) <- t.(n-1);
    decale_tableau t (n-1)
  end

let ajoute_fils_a_gauche a b =
  match a with
    Noeud(r, t) ->
      if r >= k then a
      else begin
decale_tableau t r;
t.(0) <- b;
Noeud(r + 1, t)
```

```
      end
```

En OCaml, représentation FGFD :

```
let ajoute_fils_a_gauche a b =
  match a, b with
  | Noeud(fga, fda), Noeud(fgb, None) ->
    Noeud(Some (Noeud(fgb, fga)), fda)
  | _ -> failwith "b a déjà un frere"
```

c) En C, représentation standard :

```
struct arbre ajoute_fils_a_droite(struct arbre a, struct arbre b) {
  if (a.rang >= k) return a;
  a.fils[a.rang] = &b;
  a.rang++;
  return a;
}
```

En C, représentation FGFD :

```
struct arbre ajoute_fils_a_droite(struct arbre a, struct arbre b) {
  struct arbre* res = a.fils_gauche;
  while (res)
    res = res->frere_droit;
  res->frere_droit = &b;
  return a;
}
```

En OCaml, représentation standard :

```
let ajoute_fils_a_droite a b =
  match a with
  Noeud(r, t) ->
    if r >= k then a
    else begin
      t.(r) <- b;
      Noeud(r+1, t)
    end
```

En OCaml, représentation FGFD :

```
let rec ajoute_frere_a_droite a b =
  match a with
    None -> Some(b)
  | Some(Noeud(fg, fd)) -> Some(Noeud(fg, ajoute_frere_a_droite fd b))

let ajoute_fils_a_droite a b =
  match a with
    Noeud(fg, fd) -> Noeud(ajoute_frere_a_droite fg b, fd)
```

d) En C, représentation standard :

```
struct arbre supprimer_fils(struct arbre a, int i) {
  a.rang--;
  for(; i < a.rang; i++)
    a.fils[i] = a.fils[i+1];
  return a;
```

```
}
```

En C, représentation FGFD :

```
struct arbre supprimer_fils(struct arbre a, int i) {
  if (i == 0) {
    a.fils_gauche = a.fils_gauche->frere_droit;
    return a;
  };
  struct arbre* b = a.fils_gauche;
  while (i > 1) {
    b = b->frere_droit;
    i--;
  };
  b->frere_droit = b->frere_droit->frere_droit;
  return a;
}
```

En OCaml, représentation standard :

```
let rec decaler_droite t i n =
  if i < n then begin
    t.(i) <- t.(i+1);
    decaler_droite t (i+1) n
  end

let supprimer_fils a i =
  match a with
    Noeud(r, t) ->
      decaler_droite t i (r - 1);
      Noeud(r - 1, t)
```

En OCaml, représentation FGFD :

```
let rec supprimer_frere a i =
match a, i with
  Noeud(fg, Some(Noeud(fdg,fdd))), 1 -> Noeud(fg, fdd)
| Noeud(fg, Some fd), _ -> Noeud(fg, Some (supprimer_frere fd (i-1)))
| Noeud(_, None), _ -> failwith "pas assez de frere"

let supprimer_fils a i =
  match a, i with
    Noeud(Some(Noeud(fgg,fgd)), fd), 0 -> Noeud(fgd, fd)
  | Noeud(Some fg, fd), _ ->
    Noeud(Some(supprimer_frere fg i), fd)
  | Noeud(None, _), _ -> failwith "pas assez de fils"
```

4. Soit n le nombre de sous-arbres de la racine de `a`

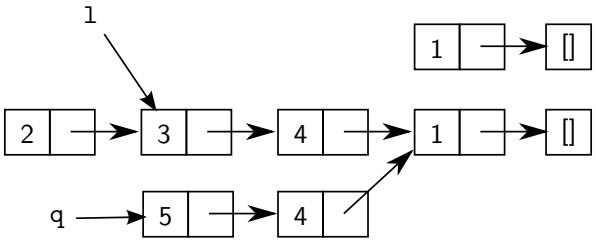
Fonction	représentation standard	représentation FGFD
`nieme_fils`	$\mathcal{O}(1)$	$\mathcal{O}(i)$
`ajoute_fils_a_gauche`	$\mathcal{O}(n)$	$\mathcal{O}(1)$
`ajoute_fils_a_droite`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`supprimer_fils`	$\mathcal{O}(n-i)$	$\mathcal{O}(i)$

5. L'accès à un fils est plus rapide dans la représentation standard.

La représentation standard est plus avantageuse si on ajoute les sous-arbres à droite, alors que la représentation FGFD est plus avantageuse si on les ajoute à gauche.

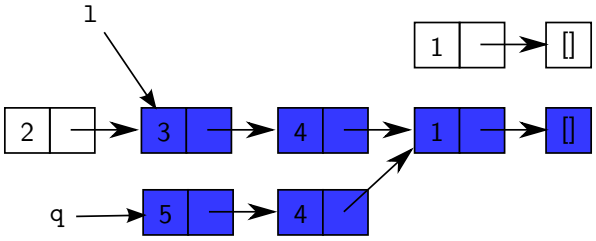
La place occupée en mémoire est meilleure pour la représentation FGFD dans le cas où beaucoup de noeuds ne sont pas complets (c'est-à-dire ont un nombre de noeuds strictement inférieur à k), ce qui est le cas des feuilles par exemple.

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

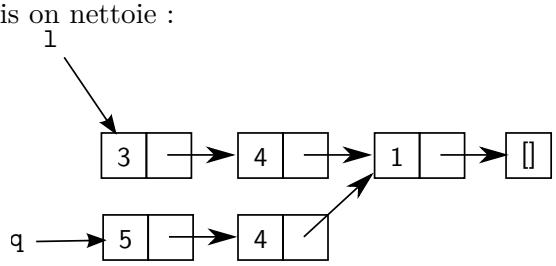
1. 

   Les parties accessibles sont celles à partir de `l` et `q`.

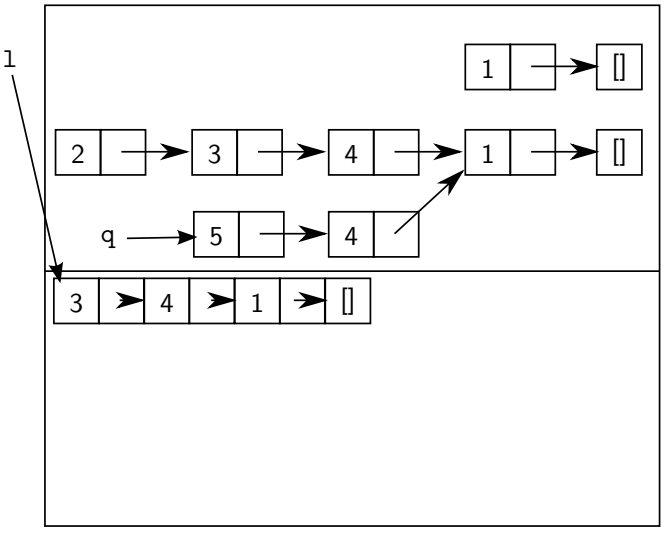
2. On marque à partir des variables du programme :



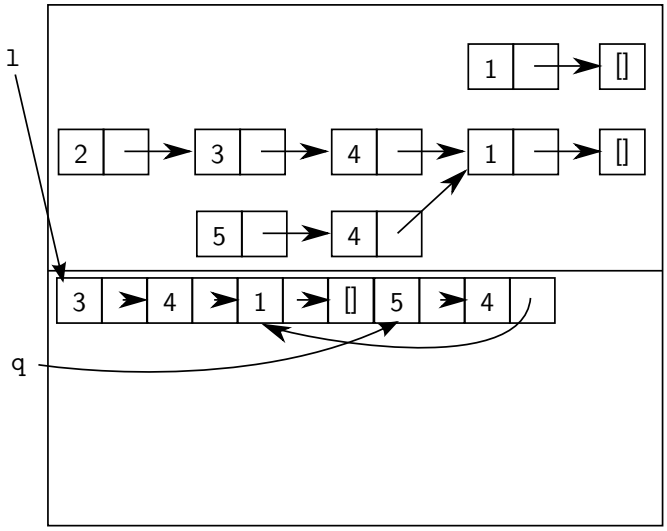
   Puis on nettoie :



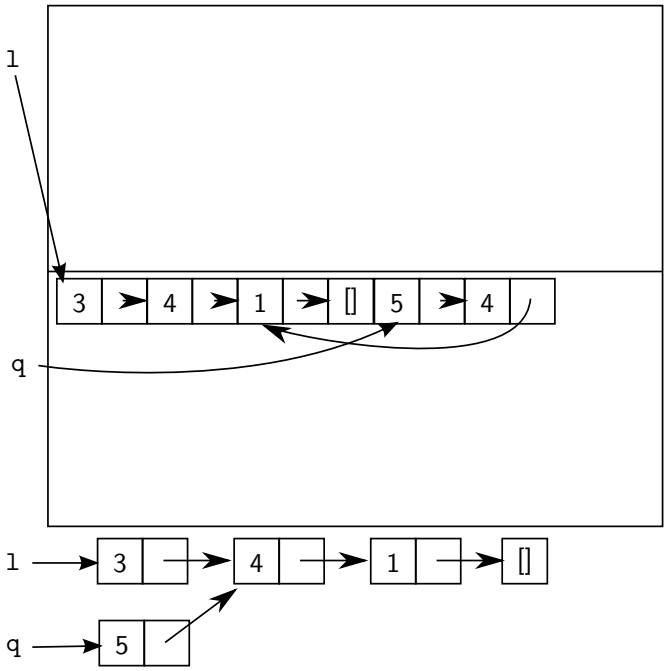
3. On recopie d'abord `l` (par exemple)



Puis `q`



Et on change de moitié de mémoire :



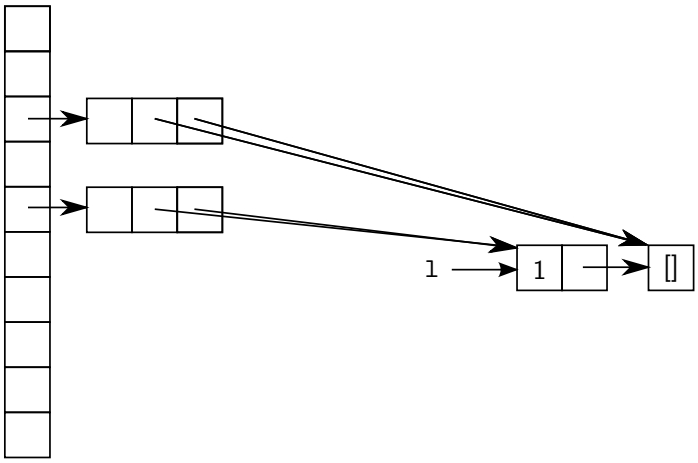
4.

5.
```
let table = creer 251

let hashcons l =
  try rechercher table l
  with Not_found ->
    inserer table l l;
    l

let nil = hashcons []

let cons x l = hashcons (x :: l)
```

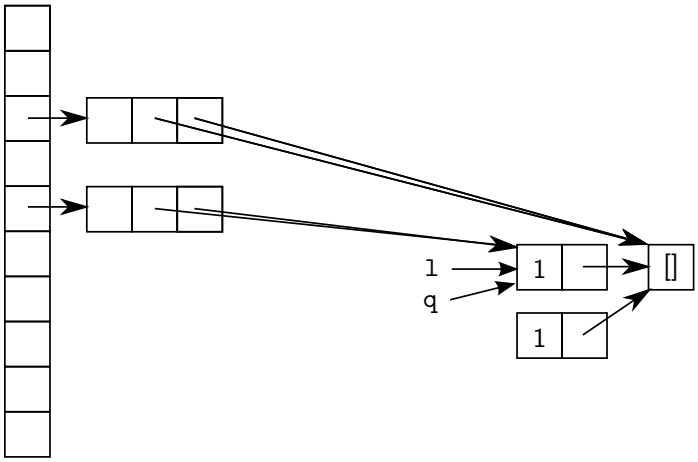
6.
```
let l = cons 1 nil in
```

```
let q = cons 1 nil in
let l =
  let h = cons 2 (cons 3 (cons 4 q)) in
  List.tl h
in
let q = cons 5 (cons 4 q) in
  l, q
```

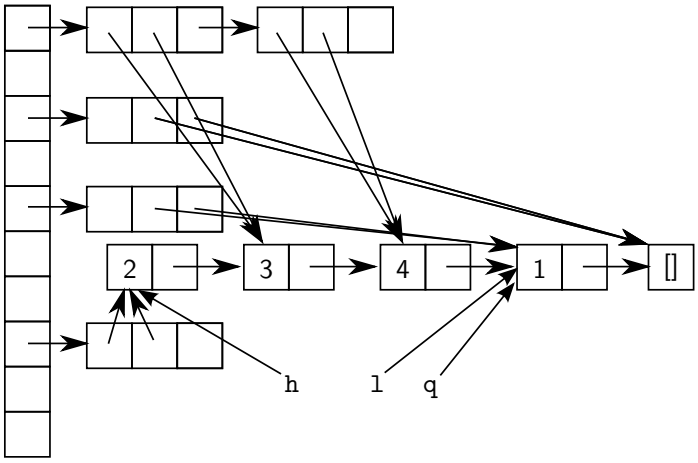
7. Après la première ligne :



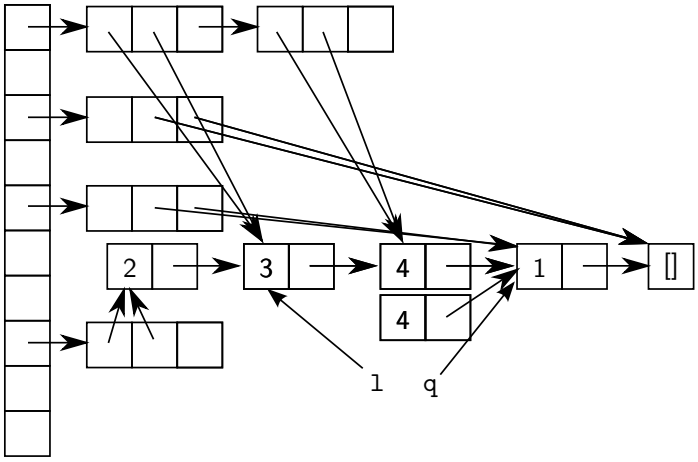
Après la deuxième :



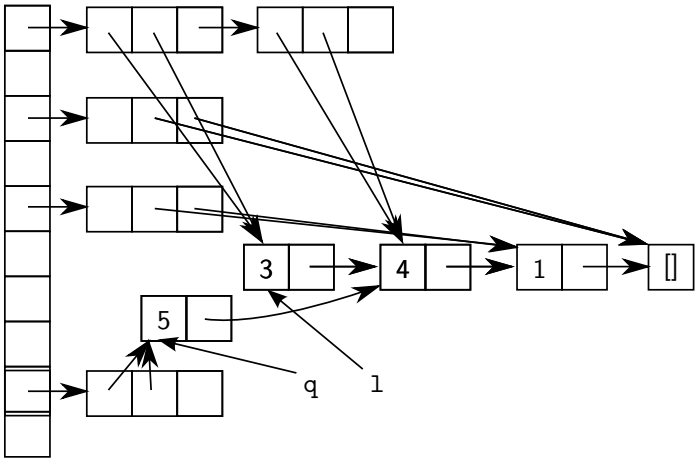
Après la quatrième ligne :



Après le `cons 4 q` de la ligne 7 :



À la fin :



Exercice 4 : Fonctions de hachage (3 points)

1. La valeur de hachage vaut 00000000 `xor` 'c' `xor` 'h' `xor` 'i' `xor` 'e' `xor` 'n',
   soit 01100011 `xor` 01101000 `xor` 01101001 `xor` 01100101 `xor` 01101110,
   soit 00001011 `xor` 01101001 `xor` 01100101 `xor` 01101110,
   soit 01100010 `xor` 01100101 `xor` 01101110,

soit 00000111 `xor` 01101110,
soit 01101001.

2. La valeur de hachage vaut 00000000 `xor` 'n' `xor` 'i' `xor` 'c' `xor` 'h' `xor` 'e',
   soit 01101110 `xor` 01101001 `xor` 01100011 `xor` 01101000 `xor` 01100101,
   soit 00000111 `xor` 01100011 `xor` 01101000 `xor` 01100101,
   soit 01100100 `xor` 01101000 `xor` 01100101,
   soit 00001100 `xor` 01100101,
   soit 01101001.
3. Si on a un anagramme $c_1 \dots c_n$ et $c_{\pi(1)} \dots c_{\pi(n)}$ avec une permutation π, la valeur de hachage vaudra 0 `xor` c_1 `xor` $\cdots$ `xor` $c_n = 0$ `xor` $c_{\pi(1)}$ `xor` $\cdots$ `xor` $x_{\pi(n)}$ dans les deux cas. Il y aura donc collision entre anagrammes.
4. La fonction combine ne doit pas être commutative, sinon la valeur de hachage est la même pour les anagrammes, ce qui entraîne des collisions.
5. La fonction `combine(x, y) = x * 19 + y` n'est jamais commutative (on peut vérifier que si `combine(x,y)` = `combine(y,x)` alors `x` = `y`), elle ne pose a priori pas de problème, au moins pour les anagrammes.
6. On peut définir une fonction de hachage de façon récursive :
   - si l'arbre est vide, on retourne une certaine constante (par exemple 0) ;
   - si on a un noeud contenant l'entier i avec comme fils fg et fd, on calcule récursivement la valeur de hachage h_{fg} du fils gauche et h_{fd} du fils droit, et on retourne `combine(`h_{fg}`, combine(`h_{fd}`, `i`))`.