M1 Informatique - Ingénierie Logicielle
TD-TP Comparaison de l'architecture fonctionnelle et de l'architecture Objet.

Le schéma "Interpréteur"

Où l'on voit une implémentation "objet" des types "somme" (ou type concrets ou types algébriques) et les avantages et inconvénients respectifs de deux styles architecturaux, l'un privilégiant l'évolution fonctionnelle (facilité à ajouter une nouvelle fonction sans modifier le code existant), l'autre privilégiant l'évolution structurelle (facilité à ajouter un nouveau type de données).

Considérons le texte de programme donné en annexe A, issu d'un exercice donné dans le cours de L3 "Objets avancés", écrit en *OCaml*, qui définit un type somme et les fonctions qui permettent de représenter, de simplifier et d'évaluer des formules en logique booléenne. L'architecture de ce programme est fonctionnelle.

(Il est à noter que l'architecture objet utilise aussi des fonctions (les méthodes) mais elles sont distribuées autrement dans le texte du programme)

Question 1.

Imaginons que le programme de l'annexe A soit utilisable via sa version chargée en mémoire mais que son code source soit inaccessible.

Soit à définir, en *Ocaml*, dans un fichier source indépendant et sans pouvoir modifier le programme original de l'annexe A, une extension de ce programme, réalisant une fonction qui rende la liste de toutes les variables utilisées dans une formule logique donnée. Cela est-il réalisable ? On peut penser que oui, une proposition de code pour une telle fonction est donné au listing 1.

Cela fonctionne-t-il ?

```
1  (* fonction récursive prenant en arguments (1) une liste de strings l et (2) un élément f
2  de type form, rendant la liste l augmentée des noms des variables utilisées dans f *)
3
4  let rec get_vars l = function
5    | Var n -> if not(List.mem n l) then l @ [n] else l
6    | Not f -> get_vars l f
7    | And (f1, f2) ->
8       let l1 = get_vars l f1 in
9       let l2 = get_vars l1 f2 in l2
10   | Or (f1, f2) ->
11      let l1 = get_vars l f1 in
12      let l2 = get_vars l1 f2 in l2
13   | Imp (f1, f2) ->
14      let l1 = get_vars l f1 in
15      let l2 = get_vars l1 f2 in l2
16   | Equ (f1, f2) ->
17      let l1 = get_vars l f1 in
18      let l2 = get_vars l1 f2 in l2
19   | _ -> l;;
```

Listing (1) *– Ajout d'une fonction au programme de l'annexe A*

Question 2. Soit à définir, en Ocaml, dans un fichier source indépendant et sans modifier le programme original de l'annexe A, une extension de ce programme qui permet de lui ajouter l'opérateur logique *Xor*. Est-ce possible ? Si oui énoncez la solution.

Question 3. : Etudier le schéma de conception *Interpréteur*.

Question 4. : Appliquer le pattern *Interpréteur* à la réalisation, dans le langage à objet (OBJ) de votre choix, d'une version objet de l'évaluateur de formules logiques (programme `ObjFormEval`). Limitez vous pour cette question à une transposition de ce qui est donné en annexe A (ne pas ajouter la fonction get_vars ni l'opérateur logique *Xor*).

Question 5. Soit à définir, en OBJ dans un fichier source indépendant et sans modifier le programme original `ObjFormEval)`, une extension de ce programme qui permet de lui ajouter l'opérateur logique *Xor*. Est-ce possible ? Si oui faites le.

Question 6. Imaginons que votre programme `ObjFormEval)` soit utilisable via sa version chargée en mémoire mais que son code source soit inaccessible.
Soit à définir, en OBJ dans un fichier source indépendant et sans modifier le programme original `ObjFormEval)`, une extension de ce programme, réalisant une méthode `get_vars` qui rende la liste de toutes les variables utilisées dans une formule logique donnée.
Est-ce possible [1] ?

Annexe A : un évaluateur de formules en logique booléenne - Architecture fonctionnelle

```
type form =
  | Top | Bot
  | Var of string
  | Not of form
  | And of form * form
  | Or of form * form
  | Imp of form * form
  | Equ of form * form;;

 (* ---------------------------- *)
 (* toString *)
let rec string_of_form = function
  | Top -> "true"
  | Bot -> "false"
  | Var n -> n
  | Not f -> "~" ^ (string_of_form f) (*^ est concaténation *)
  | And (f1, f2) ->
    "(" ^ (string_of_form f1) ^ "/\\" ^ (string_of_form f2) ^ ")"
  | Or (f1, f2) ->
    "(" ^ (string_of_form f1) ^ "\\/" ^ (string_of_form f2) ^ ")"
  | Imp (f1, f2) ->
    "(" ^ (string_of_form f1) ^ "->" ^ (string_of_form f2) ^ ")"
  | Equ (f1, f2) ->
    "(" ^ (string_of_form f1) ^ "<->" ^ (string_of_form f2) ^ ")";;

let f = Imp (And (Var "A", Var "B"), Or (Not (Var "C"), Top));;
print_endline (string_of_form f);;

(* ---------------------------- *)
(* simplification des formules *)

let simplif_and = function
  | (f, Top) | (Top, f) -> f
  | (_, Bot) | (Bot, _) -> Bot
  | (l, r) -> And (l, r);;

let simplif_or = function
  | (_, Top) | (Top, _) -> Top
  | (f, Bot) | (Bot, f) -> f
  | (l, r) -> Or (l, r);;

let simplif_imp = function
  | (_, Top) | (Bot, _) -> Top
  | (f, Bot) -> Not f
  | (Top, f) -> f
  | (l, r) -> Imp (l, r);;
```

1. Vous pourrez aller plus loin sur cette question en étudiant le schéma *visiteur*? et en lisant cet article.

```
let simplif_equ = function
  | (f, Top) | (Top, f) -> f
  | (_, Bot) | (Bot, _) -> Bot
  | (l, r) -> Equ (l, r);;

let rec simplif_form = function
  | And (f1, f2) ->
     let f1' = simplif_form f1
     and f2' = simplif_form f2 in
     simplif_and (f1', f2')
  | Or (f1, f2) ->
     let f1' = simplif_form f1
     and f2' = simplif_form f2 in
     simplif_or (f1', f2')
  | Imp (f1, f2) ->
     let f1' = simplif_form f1
     and f2' = simplif_form f2 in
     simplif_imp (f1', f2')
  | Equ (f1, f2) ->
     let f1' = simplif_form f1
     and f2' = simplif_form f2 in
     simplif_equ (f1', f2')
  | f -> f;;

let f = And (Var "A", Or (Var "B", Top));;
let f' = simplif_form f;;
print_endline (string_of_form f);;
print_endline (string_of_form f');;

(* ---------------------------- *)
(* évaluation des formules *)

let rec eval_form l = function
  | Top -> true
  | Bot -> false
  | Var n ->
     (try List.assoc n l
      with Not_found -> failwith (n ^ " not in the assignment!"))
  | Not f -> not (eval_form l f)
  | And (f1, f2) ->
     let f1' = eval_form l f1
     and f2' = eval_form l f2 in
     f1' && f2'
  | Or (f1, f2) ->
     let f1' = eval_form l f1
     and f2' = eval_form l f2 in
     f1' || f2'
  | Imp (f1, f2) ->
( let f1' = eval_form l f1
     and f2' = eval_form l f2 in
     (not f1') || f2'
  | Equ (f1, f2) ->
     let f1' = eval_form l f1
     and f2' = eval_form l f2 in
     f1' = f2';;

let f = Imp (Var "A", Imp (Var "B", Var "A"));;
let l = [("A", true); ("B", true)];;
eval_form l f;;
```