

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 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 ; le code de cette fonction est donné au listing 1.

Est-ce possible (cela fonctionne-t-il) ?

```

1  let rec get_vars l = function
2  | Var n -> if not(List.mem n l) then l @ [n] else l
3  | Not f -> get_vars l f
4  | And (f1, f2) ->
5      let l1 = get_vars l f1 in
6      let l2 = get_vars l1 f2 in l2
7  | Or (f1, f2) ->
8      let l1 = get_vars l f1 in
9      let l2 = get_vars l1 f2 in l2
10 | Imp (f1, f2) ->
11     let l1 = get_vars l f1 in
12     let l2 = get_vars l1 f2 in l2
13 | Equ (f1, f2) ->
14     let l1 = get_vars l f1 in
15     let l2 = get_vars l1 f2 in l2
16 | _ -> 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 ?

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*). Ne définissez dans votre programme aucune méthode *get_vars* et ne traitez pas 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 ? Vous pourrez aller plus loin sur cette question en étudiant le schéma *visiteur*? et en lisant cet [article](#).

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

```

1  type form =
2  | Top | Bot
3  | Var of string
4  | Not of form
5  | And of form * form
6  | Or of form * form
7  | Imp of form * form
8  | Equ of form * form;;

10  (* ----- *)
11  (* toString *)
12  let rec string_of_form = function
13  | Top -> "true"
14  | Bot -> "false"
15  | Var n -> n
16  | Not f -> "~" ^ (string_of_form f) (* ^ est concaténation *)
17  | And (f1, f2) ->
18    "(" ^ (string_of_form f1) ^ "/" ^ (string_of_form f2) ^ ")"
19  | Or (f1, f2) ->
20    "(" ^ (string_of_form f1) ^ "\" ^ (string_of_form f2) ^ ")"
21  | Imp (f1, f2) ->
22    "(" ^ (string_of_form f1) ^ "->" ^ (string_of_form f2) ^ ")"
23  | Equ (f1, f2) ->
24    "(" ^ (string_of_form f1) ^ "<->" ^ (string_of_form f2) ^ "));

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

29  (* ----- *)
30  (* simplification des formules *)

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

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

42  let simplif_imp = function
43  | (_, Top) | (Bot, _) -> Top
44  | (f, Bot) -> Not f
45  | (Top, f) -> f
46  | (l, r) -> Imp (l, r);;

48  let simplif_equ = function
49  | (f, Top) | (Top, f) -> f
50  | (_, Bot) | (Bot, _) -> Bot
51  | (l, r) -> Equ (l, r);;

```

```

53 let rec simplif_form = function
54 | And (f1, f2) ->
55     let f1' = simplif_form f1
56     and f2' = simplif_form f2 in
57     simplif_and (f1', f2')
58 | Or (f1, f2) ->
59     let f1' = simplif_form f1
60     and f2' = simplif_form f2 in
61     simplif_or (f1', f2')
62 | Imp (f1, f2) ->
63     let f1' = simplif_form f1
64     and f2' = simplif_form f2 in
65     simplif_imp (f1', f2')
66 | Equ (f1, f2) ->
67     let f1' = simplif_form f1
68     and f2' = simplif_form f2 in
69     simplif_equ (f1', f2')
70 | f -> f;;

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

77 (* ----- *)
78 (* évaluation des formules *)

80 let rec eval_form l = function
81 | Top -> true
82 | Bot -> false
83 | Var n ->
84     (try List.assoc n l
85      with Not_found -> failwith (n ^ " not in the assignment!"))
86 | Not f -> not (eval_form l f)
87 | And (f1, f2) ->
88     let f1' = eval_form l f1
89     and f2' = eval_form l f2 in
90     f1' && f2'
91 | Or (f1, f2) ->
92     let f1' = eval_form l f1
93     and f2' = eval_form l f2 in
94     f1' || f2'
95 | Imp (f1, f2) ->
96     let f1' = eval_form l f1
97     and f2' = eval_form l f2 in
98     (not f1') || f2'
99 | Equ (f1, f2) ->
100     let f1' = eval_form l f1
101     and f2' = eval_form l f2 in
102     f1' = f2';;

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

```