Examen ELFE

janvier 2006 - durée 3h

documents de cours autorisés - livres et calculatrices interdits

Programmation Fonctionnelle

Vous disposez de 3h pour traiter la partie "Programmation logique" et la partie "Programmation Fonctionnelle".

Vous êtes libre de gérer votre temps comme vous le souhaitez entre ces deux parties. Cependant il vous faut rendre des copies séparées pour chacun des thèmes.

Exercice 1 : *Construction de listes*

Réalisez une fonction f qui à partir d'une liste d'entiers, tous supposés positifs non nuls, construit la liste des listes de la forme $[1; 2; \ldots; n]$ pour chaque entier n de la liste de départ.

Par exemple :

```
# f [5; 1; 3] ;;
- : int list list = [[1; 2; 3; 4; 5]; [1]; [1; 2; 3]]
```

Exercice 2 : *Expressions booléennes*

On considère le type suivant pour représenter les expressions booléennes

```
type expr_bool =
    Vrai
  | Faux
  | Et of expr_bool*expr_bool
  | Ou  of expr_bool*expr_bool
  | Non of expr_bool ;;
```

les constructeurs Vrai, Faux, Et, Ou et Non ayant la signification implicite.

Q 1 . Il est bien connu que les opérateurs `ou` et `et` sont commutatifs. Écrivez une fonction qui teste l'égalité de deux expressions de type `expr_bool` en tenant compte de cette remarque.

Q 2 . Complétez la déclaration du type expr_bool pour permettre la représentation des expressions booléennes avec variables (les variables étant des chaînes de caractères).

Q 3 . Réalisez la fonction nommée `evaluer` de type `expr_bool -> (string * bool) list -> bool` telle que l'expression

```
evaluer expr intp ;;
```

calcule la valeur de vérité de l'expression `expr`, les valeurs des variables étant définies dans l'interprétation `intp` des variables.

Q 4 . Réalisez la fonction nommée `variables` calculant la liste des variables contenues dans une expression booléenne. Dans cette liste toute variable n'apparaît qu'une fois.

Q 5 . Réalisez la fonction nommée `interpretations` qui calcule la liste des interprétations possibles d'une liste de variables. Par exemple, pour une variable on doit avoir deux interprétations

```
interpretations ["a"] ;;
- : (string * bool) list list = [["a", false] ; ["a", true]]
```

et pour deux variables on doit avoir quatre interprétations

```
interpretations ["a" ; "b"] ;;
- : (string * bool) list list =
  [["a", false ; "b", false] ; ["a", false ; "b", true] ;
   ["a", true ; "b", false] ; ["a", true ; "b", true]]
```

Q 6 . On rappelle qu'une formule est une tautologie si elle est vraie dans toutes les interprétations de ses variables. Réalisez le prédicat `tautologie` qui vérifie si l'expression qu'on lui passe en paramètre est une tautologie.

Solutions

Exercice 1

Solution

```
let f l =
  let rec g n l =
    if n=1 then 1::l
    else g (n-1) (n::l)
  in
    List.map (function x -> g x []) l ;;
```

Exercice 2

Q 1 .
Solution

```
let rec expr_egales exp1 exp2 =
  match (exp1, exp2) with
    | (Vrai,Vrai) -> true
    | (Faux,Faux) -> true
    | (Et(e11,e12),Et(e21,e22)) -> (expr_egales e11 e21) && (expr_egales e12 e22)
    | (Ou(e11,e12),Ou(e21,e22)) -> (expr_egales e11 e21) && (expr_egales e12 e22)
    | (Non(e1),Non(e2)) -> expr_egales e1 e2
    | (_,_) -> false ;;
```

Q 2 .
Solution

```
type expr_bool =
    Vrai
  | Faux
  | Var of string
  | Et of expr_bool*expr_bool
  | Ou  of expr_bool*expr_bool
  | Non of expr_bool ;;
```

Q 3 .
Solution

```
let rec evaluer expr intp = match expr with
| Vrai -> true
| Faux -> false
| Nom n -> List.assoc n intp
| Et (e1,e2) -> (evaluer e1 intp) && (evaluer e2 intp)
| Ou (e1,e2) -> (evaluer e1 intp) || (evaluer e2 intp)
| Non e1 -> not (evaluer e1 intp);;
```

Q 4 .
Solution Voici une solution avec une fonction auxiliaire.

```
let rec variables_aux l expr = match expr with
| Nom n -> if (List.mem n l ) then l else n::l
| Et (e1,e2) -> let l1 = variables_aux l e1 in variables_aux l1 e2
| Ou (e1,e2) -> let l1 = variables_aux l e1 in variables_aux l1 e2
| Non e1 -> variables_aux l e1
| _ -> l;;

let variables e1 = variables_aux [] e1;;
```

Biensûr on pourrait cacher la fonction auxiliaire pour en faire une fonction locale, mais pour des raisons de lisibilité je préfére laisser comme cela.

Une autre manière de faire, pourrait être de ne pas se préoccuper des doublons, de trier la liste puis, de supprimer les doublons. Cela évite de multiple parcours de la liste d'accumulation des variables, mais cela à un prix, car il se peut que cette liste grossisse artificiellement...

Q 5 .
Solution

```
let rec interpretations l = match l with
| [] -> [ [] ]
| t::q ->  let s = interpretations q in
           (List.map (fun x -> (t,false)::x) s) @
           (List.map (fun x -> (t,true)::x) s) ;;
```

Q 6 .
Solution

```
let tautologie exp =
 let v = variables exp in
 let i = interpretations v in
 not (List.mem false (List.map (fun x -> evaluer exp x) i));;
```