Corrigé de l'examen de programmation fonctionnelle en Objective Caml

Christian Rinderknecht

Mardi 22 avril 2003

Exercice A :

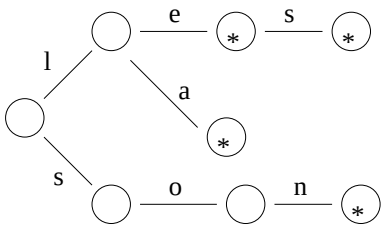
```
# fun (x,y) -> (y+1, not x);;
- : bool * int -> int * bool = <fun>
# fun p -> (snd p, fst p);;
- : 'a * 'b -> 'b * 'a = <fun>
# fun f -> fun (x,y) -> f(x+1) + f y;;
- : (int -> int) -> int * int -> int = <fun>
# fun f -> fun (x,y) -> (f(x+1), f y);;
- : (int -> 'a) -> int * int -> 'a * 'a = <fun>
# fun f -> fun (x,y) -> (f x, f y);;
- : ('a -> 'b) -> 'a * 'a -> 'b * 'b = <fun>
# fun f -> fun (x,y) -> (f x + 1, f y);;
- : ('a -> int) -> 'a * 'a -> int * int = <fun>
```

Exercice B :

```
# fun n b -> if b then n+1 else n-1;;
- : int -> bool -> int = <fun>
# fun n -> (true, n+1);;
- : int -> bool * int = <fun>
# List.map (fun n -> if n mod 2 = 0 then true else false);;
- : int list -> bool list = <fun>
# fun l -> List.map fst l;;
- : ('a * 'b) list -> 'a list = <fun>
# fun l x -> List.assoc x (List.map (fun (x,y) -> (y,x)) l);;
- : ('a * 'b) list -> 'b -> 'a = <fun>
# List.combine;;
- : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

Exercice C :

Un arbre peut représenter une collection de mots de sorte qu'il soit très efficace de vérifier si une séquence donnée de caractères est un mot valide ou pas. Dans ce type d'arbre, appelé un trie*, les arcs ont une lettre associée, chaque nœud possède une indication si la lettre de l'arc entrant est une fin de mot et la suite des mots partageant le même début. La figure suivante montre le* trie *des mots « le », « les », « la » et « son », l'astérisque marquant la fin d'un mot :*



On utilisera le type Caml suivant pour implanter les tries :

```
type trie =
  { mot_complet : string option; suite : (char * trie) list }
```

Le type prédéfini `type 'a option = None | Some of 'a` *sert à représenter une valeur éventuellement absente. Chaque nœud d'un* trie *contient les informations suivantes :*

- *Si le nœud marque la fin d'un mot* `s` *(ce qui correspond aux nœuds étoilés de la figure), alors le champ* `mot_complet` *contient* `Some s`*, sinon ce champ contient* `None`*.*
- *Le champ* `suite` *contient une liste qui associe les caractères aux nœuds.*

C.1 : *Écrire la valeur Caml de type* `trie` *correspondant à la figure ci-dessus.*

```
{mot_complet=None;
 suite=[('l',{mot_complet=None;
              suite=[('e',{mot_complet=Some "le";
                           suite=[('s',{mot_complet=Some "les";
                                        suite=[]})]});
                     ('a',{mot_complet=Some "la";
                           suite=[]})]});
        ('s',{mot_complet=None;
              suite=[('o', {mot_complet=None;
                            suite=[('n', {mot_complet=Some "son";
                                          suite=[]})]})]})]}
```

C.2 : *Écrire une fonction* `compte_mots` *qui compte le nombre de mots dans un* trie.

```
let rec compte_mots {mot_complet=m; suite=l} =
    (match m with Some _ -> 1 | None -> 0)
  + List.fold_left (fun n (_,t) -> n + compte_mots t) 0 l
```

C.3 : *Écrire une fonction* `select` *qui prend en argument un* trie *et une lettre, et renvoie le* trie *correspondant aux mots commençant par cette lettre. Si ce* trie *n'existe pas (parce qu'aucun mot ne commence par cette lettre), la fonction devra lancer une exception* `Absent`*, que l'on définira. On utilisera la fonction prédéfinie* `List.assoc` *pour effectuer la recherche dans la liste des sous-arbres* `suite`*.*

```
exception Absent
```

```
let select lettre trie =
  try
    List.assoc lettre trie.suite
  with Not_found -> raise Absent
```

C.4 : *Écrire une fonction* `recherche` *qui vérifie si une chaîne de caractères est un mot dans un* trie *donné. La fonction devra prendre un argument supplémentaire* `i`*, qui représente la position dans le mot associée au nœud courant. Le* i^e *caractère d'une chaîne* `s` *s'obtient en écrivant* `s.[i]`*, le premier caractère étant numéroté* 0*. La longueur de la chaîne* `s` *s'écrit* `String.length s`*. On emploiera la fonction* `select`*.*

```
let rec recherche chaine index trie =
  if index = String.length chaine
  then match trie.mot_complet with
         Some _ -> true
       | None -> false
  else try
         let fils = select chaine.[index] trie
         in recherche chaine (index+1) fils
       with Absent -> false
```

Exercice D :
On s'intéresse aux expressions booléennes écrites à l'aide des connecteurs `or` *(« ou » booléen),* `and` *(« et » booléen),* `not` *(négation booléenne), des constantes* `true` *et* `false`*, et de variables. Par exemple :* $(x \text{ or } y)$ *and* $not(x \text{ and } y)$*.*

D.1 : *Définir un type Caml* `bool_exp` *pour ces expressions.*

```
type bool_exp =
  Or  of bool_exp * bool_exp
| And of bool_exp * bool_exp
| Not of bool_exp
| Var of string
| True
| False
```

D.2 : *Écrire une fonction* `eval` *permettant d'évaluer de telles expressions. Cette fonction devra prendre en paramètre un environnement associant les noms de variables à leur valeur.*

```
let rec eval env = function
  Or (e1, e2)  -> eval env e1 || eval env e2
| And (e1, e2) -> eval env e1 && eval env e2
| Not (e)      -> not (eval env e)
| Var (s)      -> List.assoc s env
| x            -> x
```

D.3 : *Les connecteurs booléens considérés satisfont en particulier les identités*

$$\begin{aligned} not(a\ or\ b) &= not(a)\ and\ not(b) \\ not(a\ and\ b) &= not(a)\ or\ not(b) \\ not(not(a)) &= a \\ not(true) &= false \\ not(false) &= true \end{aligned}$$

Ces identités permettent de transformer toute expression booléenne en une expression équivalente où les négations ne sont appliquées qu'à des variables. Par exemple, l'expression $not(x\ and\ (y\ or\ not(z)))$ *peut être transformée en* $not(x)\ or\ not(y\ or\ not(z)))$ *puis en* $not(x)\ or\ (not(y)\ and\ not(not(z))))$ *et enfin en* $not(x)\ or\ (not(y)\ and\ z)$.
Écrire une fonction Caml `normalise` *qui réalise cette transformation.*

```
let rec normalise = function
  Or (e1, e2)         -> Or (normalise e1, normalise e2)
| And (e1, e2)        -> And (normalise e1, normalise e2)
| Not (Or (e1, e2))   -> normalise (And (Not e1, Not e2))
| Not (And (e1, e2)) -> normalise (Or (Not e1, Not e2))
| Not (Not (e))       -> normalise (e)
| Not (True)          -> False
| Not (False)         -> True
| x                   -> x
```