Programmation Fonctionnelle :
des concepts aux applications web
(NFP119)

Les types récursifs

1. On considère la définition du type `form` des formules propositionnelles suivant :

   ```
   type form =
     True | False | Not of form | And of form * form | Or of form * form
   ```

2. Ecrire une fonction `string_of_form : form -> string` qui affiche une formule donnée dans la syntaxe de OCaml.

3. Ecrire une fonction `eval : form -> bool` qui calcule la valeur de vérité d'une formule donnée.

4. Que fait la fonction `mystery` définie ainsi ?

   ```
   let mystery (f1, f2) = Not (And (Not f1, Not f2))
   ```

   Définir l'implication `impl` de la même manière.

5. On souhaite simplifier une formule en supprimant les occurrences de `Not (Not ...)`. Ecrire la fonction `simpl : form -> form` qui implante cette simplification.

6. On complète la définition précédente de `form` avec : `... | Atom of string` de manière à prendre en compte des variables propositionnelles. Modifier la définition de `eval` pour pendre aussi en argument un paramètre fonctionnel qui associe une valeur de vérité à chaque formule atomique. Son type est maintenant le suivant :

   ```
   eval : form -> (string -> bool) -> bool
   ```

7. Définir une variante de `eval` qui prend en paramètre une liste d'association pour coder les valeurs de vérité des formules atomiques. Son type est alors le suivant :

   ```
   eval' : form -> (string * bool) list -> bool
   ```

   On utilisera, pour définir `eval'` à partir de `eval`, la fonction `List.assoc` pour chercher la valeur de vérité associé à une formule atomique (cette fonction lève une exception si l'élément cherché n'est pas dans la liste). Son type est le suivant :

   ```
   List.assoc : 'a -> ('a * 'b) list -> 'b
   ```

   Remarquer que si `l` est de type `(string * bool) list` alors `fun x -> (assoc x l)` est de type `string -> bool` et en déduire la définition de `eval'`.

8. On définit le type `expr` des expressions arithmétiques suivant :

   ```
   type expr =
     Const of int | Minus of expr | Plus of expr * expr | Times of expr * expr
   ```

9. Ecrire une fonction `string_of_expr : expr -> string` qui affiche une expression donnée dans la syntaxe de OCaml.

10. Ecrire une fonction `eval_expr : expr -> int` qui évalue une expression donnée.

11. De manière à prendre en compte des variables dans les expressions arithmétiques, on complète la définition précédente de `expr` avec le constructeur :

```
| Var of string
```

Modifier la définition de `eval` pour pendre aussi en argument un paramètre `m` de type `(string * int) list`. Son type est maintenant le suivant :

```
eval_expr' : expr -> (string * int) list -> int
```

12. Compléter la fonction `string_of_expr : expr -> string` qui affiche une expression donnée dans la syntaxe de OCaml.

13. Ecrire une fonction `translate : form -> expr` qui traduit une formule en expression, en utilisant comme toujours les constantes 1 et 0 pour représenter `True` et `False`.

14. En déduire une nouvelle fonction d'évaluation des formules booléennes utilisant l'évaluation des expressions.

15. De manière à prendre en compte des déclarations locales dans les expressions arithmétiques, on complète la définition précédente de `expr` avec le constructeur :

    ```
    | Let of string * expr * expr
    ```

    Modifier la définition de la fonction `eval_expr' : expr -> (string * int) list -> int` en conséquence.

16. Modifier aussi la définition de `string_of_expr` en conséquence.

17. Modifier ensuite la définition de `string_of_expr` de manière à indenter les déclarations locales ainsi :

    ```
    let x =
      let y = (x + 1)
      in (y * 2)
    in (x * 3)
    ```

    On ajoutera pour cela un argument entier représentant la profondeur à la fonction `string_of_expr`.