TDs de Logique. Feuille 6. Caml

Exercice 1.— Ecrire une fonction `combine` qui transforme deux listes $[a_1; a_2; \ldots; a_n]$ et $[b_1; b_2; \ldots; b_n]$ en la liste $[(a_1, b_1); (a_2, b_2); \ldots; (a_n, b_n)]$.

Exercice 2.— Ecrire une fonction `split` qui fait l'opération inverse.

Exercice 3.— Ecrire une fonction `flat` telle que `flat [[1;2;3];[4;5]]=[1;2;3;4;5]`.

Exercice 4.— Définir le type des arbres binaires et écrire la fonction `flat` vue en cours avec accumulateur.

Exercice 5.— Donner le type de la fonction suivante

```
let rec f x = function
      [] -> []
    |y::l -> (x,y)::f x l
;;
```

Dire ce que calcule `f` et réécrire `f` avec `map`.
En utilisant `f`, écrire une fonction `p` qui calcule le produit cartésien de deux listes, c'est-à-dire la liste de tous les couples dont le premier élément est dans la première liste et le second dans la seconde liste. Par exemple,

```
p [1;2] [true;false]
= [(1,true);(1,false);(2,true);(2,false)]
```

Exercice 6.— Définir un type somme pour représenter les propositions. On prendra un constructeur pour chaque connecteur.

Exercice 7.— Renommer le type `(string * bool) list` en `interpretation`.

Exercice 8.— Ecrire une fonction `evalue` qui prend en argument une formule F et une interpretation I et calcule $\bar{I}(F)$. On utilisera la fonction prédéfinie `assoc` de type

```
assoc : 'a -> ('a * 'b) list -> 'b = <fun>
```

Exercice 9.— Ecrire une fonction qui calcule les atomes d'une formule F. On utilisera la fonction prédéfinie `mem` qui teste l'appartenance d'un élément à une liste.

Les flux

```
#[<'1;'2;'3>];;
- : int stream = <abstr>
#let s=[<'1;'2;'3>];;
s : int stream = <abstr>
#let t=[<'4;'5>];;
t : int stream = <abstr>
#[<s;t>];;
- : int stream = <abstr>
#stream_next  [<s;t>];;
- : int = 1
#stream_next  [<s;t>];;
- : int = 2

#let rec saute_blancs s =
     match s with
  [<'(` `|`\t`|`\n`)>] -> saute_blancs s
| [< 'c >]              -> [< 'c ; saute_blancs s >]
| [< >]                 -> s
;;
saute_blancs : char stream -> char stream = <fun>
```

On définit le type suivant pour manipuler des expressions arithmétiques :

```
#type expression = Var of char
                 | Plus of expression * expression
                 | Minus of expression * expression
                 | Mult of expression * expression
                 | Div of expression * expression
;;
```

Ecrire des fonctions pour transformer une chaîne de caractères rentrée au clavier représentant une expression arithmétique sous forme infixe complètement parenthésee, en une expression du type ci-dessus.