Correction TD de programmation fonctionnelle n°6

Licence d'informatique

▸ Exercice 1. Structure des formules

On utilisera le type `formule` suivant :

```
type formule =
    Atome of string
  | Valeur of bool
  | Non of formule
  | Imp of formule * formule
  | Et of formule * formule
  | Ou of formule * formule;;
```

1. Écrire une fonction qui calcule la hauteur d'une formule.

✂ --

Parler de la possibilité de matcher n'importe quoi avec _, ainsi que de la possibilité d'associer une même action à plusieurs motifs.

```
# let rec hauteur f = match f with
          Atome(_)
        | Valeur(_) -> 1
        | Non(x) -> 1+hauteur(x)
        | Imp(x,y)
        | Et(x,y)
        | Ou(x,y) ->
        let
           maximum a b = if a>b then a else b
        in
           1+(maximum (hauteur x) (hauteur y));;
val hauteur : formule -> int = <fun>
# hauteur (Ou(Valeur(false),Imp(Atome("a"),Valeur(true))));;
- : int = 3
```

-- ✂

2. Écrire une fonction qui calcule la taille d'une formule.

✂ --

```
# let rec taille f = match f with
          Atome(_)
        | Valeur(_) -> 1
```

```
        | Non(x) -> 1+taille(x)
        |   Imp(x,y)
        |   Et(x,y)
        | Ou(x,y) ->1+(taille x)+(taille y);;
val taille : formule -> int = <fun>
# taille (Ou(Valeur(false),Imp(Atome("a"),Valeur(true))));;
- : int = 5
```

3. Écrire une fonction qui teste si une formule est sans atome.

```
# let rec est_sans_atome f = match f with
          Atome(_) -> false
        | Valeur(_) -> true
        | Non(x) -> est_sans_atome(x)
        | Imp(x,y)
        | Et(x,y)
        | Ou(x,y) ->(est_sans_atome x) && (est_sans_atome y);;
val est_sans_atome : formule -> bool = <fun>
```

4. Écrire une fonction qui évalue une formule. On supposera que la formule est sans atome.

```
# let rec eval f = match f with
          Atome(_) -> failwith "La formule a un atome!"
        | Valeur(x) -> x
        | Non(x) -> not (eval x)
        | Imp(x,y) -> (not (eval x)) || (eval y)
        | Et(x,y) -> (eval x) && (eval y)
        | Ou(x,y) -> (eval x) || (eval y);;
val eval : formule -> bool = <fun>
# eval (Ou(Valeur(false),Imp(Valeur(false),Valeur(true))));;
- : bool = true
```

5. Écrire une fonction qui construise la liste sans doublon des atomes d'une formule.

```
# let rec liste_atomes =
  let rec fusion l m = match l with
       [] -> m
     | x::y -> if (appartient m x) then (fusion y m) else x::(fusion y m) in
function
       Atome(x) -> [x]
     | Valeur(_) -> []
```

```
      | Non(x) -> liste_atomes(x)
      | Imp(x,y)
      | Et(x,y)
      | Ou(x,y) -> (fusion (liste_atomes x) (liste_atomes y));;
val liste_atomes : formule -> string list = <fun>

# liste_atomes (Ou(Atome("a"),Imp(Atome("b"),Atome("a"))));;
- : string list = ["b"; "a"]
```
----

6. Écrire une fonction qui a pour argument une formule et un atome et qui teste si cette formule contient cet atome.

```
# let rec possede_atome f a = match f with
          Atome(x) -> x=a
        | Valeur(_) -> false
        | Non(x) -> possede_atome x a
        | Imp(x,y)
        | Et(x,y)
        | Ou(x,y) ->(possede_atome x a ) || (possede_atome y a);;
val possede_atome : formule -> string -> bool = <fun>
possede_atome (Ou(Valeur(false),Imp(Atome("a"),Valeur(true)))) "a";;
- : bool = true
possede_atome (Ou(Valeur(false),Imp(Atome("b"),Valeur(true)))) "a";;
: bool = false
```

▸ **Exercice 2. Simplification de formules**

Écrire une fonction permettant de simplifier une formule grâce aux règles suivantes :

1. $F \vee 0 = F$
2. $F \wedge 0 = 0$
3. $F \vee 1 = 1$
4. $F \wedge 1 = F$
5. $\neg 0 = 1$
6. $\neg 1 = 0$

Parler de la nécessité d'utiliser `begin...end` pour lever l'ambiguïté sur les portées des règles de pattern matching.

```
# let rec simplifier = function
        Atome(x) -> Atome(x)
```

```
        | Valeur(x) -> Valeur(x)
        | Non(x) ->
          begin
          match simplifier x with
                 Valeur(x3) -> Valeur(not x3)
               | x2 -> Non(x2)
          end
        | Et(x,y) ->
          begin
          match (simplifier x,simplifier y) with
                 (_,Valeur(false))
               | (Valeur(false),_) -> Valeur(false)
               | (x2,Valeur(_))
               | (Valeur(_),x2) -> x2
               | (x2,y2) -> Et(x2,y2)
          end
        | Ou(x,y) ->
          begin
          match(simplifier x,simplifier y) with
                 (_,Valeur(true))
               | (Valeur(true),_) -> Valeur(true)
               | (x2,Valeur(_))
               | (Valeur(_),x2) -> x2
               | (x2,y2) -> Ou(x2,y2)
          end
        | Imp(x,y) ->
          begin
          match(simplifier x,simplifier y) with
                 (_,Valeur(true))
               | (Valeur(false),_) -> Valeur(true)
               | (Valeur(true),x2) -> x2
               | (x2,Valeur(false)) -> Non(x2)
               | (x2,y2) -> Imp(x2,y2)
          end
val simplifier : formule -> formule = <fun>

simplifier (Ou(Valeur(false),Valeur(true)));;
```

--- ✂