Correction interro : Logique et Programmation Fonctionnelle

Licence d'informatique 3.1

——

▸ **Exercice 1.** ***Formules***

Simplifier le plus possible les formules suivantes (en donnant les équivalences intermédiaires) et repérer les tautologies.

1. $\neg(F \vee G) \rightarrow \neg G$

$$\begin{aligned}
&\Leftrightarrow \neg\neg(F \vee G) \vee \neg G \\
&\Leftrightarrow (F \vee G) \vee \neg G \\
&\Leftrightarrow F \vee (G \vee \neg G) \\
&\Leftrightarrow F \vee 1 \\
&\Leftrightarrow 1
\end{aligned}$$

Il s'agit donc d'une tautologie.

---

2. $\neg(F \wedge G) \rightarrow ((\neg F \rightarrow H) \rightarrow (G \wedge \neg F))$

$$\begin{aligned}
&\Leftrightarrow \neg\neg(F \wedge G) \vee (\neg(\neg\neg F \vee H) \vee (G \wedge \neg F)) \\
&\Leftrightarrow (G \wedge F) \vee (G \wedge \neg F) \vee (\neg F \wedge \neg H) \\
&\Leftrightarrow (G \wedge (F \vee \neg F)) \vee (\neg F \wedge \neg H) \\
&\Leftrightarrow (G \wedge 1) \vee (\neg F \wedge \neg H) \\
&\Leftrightarrow G \vee (\neg F \wedge \neg H)
\end{aligned}$$

---

3. $(F \rightarrow G) \wedge (G \vee H \vee F) \wedge (H \rightarrow F)$

$$
\begin{aligned}
&\Leftrightarrow (G \vee \neg F) \wedge (G \vee H \vee F) \wedge (\neg H \vee F)\\
&\Leftrightarrow (G \vee (\neg F \wedge (H \vee F))) \wedge (F \vee \neg H)\\
&\Leftrightarrow (G \vee (\neg F \wedge H) \vee (\neg F \wedge F)) \wedge \neg(\neg F \wedge H)\\
&\Leftrightarrow (G \vee (\neg F \wedge H) \vee 0) \wedge \neg(\neg F \wedge H)\\
&\Leftrightarrow (G \vee (\neg F \wedge H)) \wedge \neg(\neg F \wedge H)\\
&\Leftrightarrow (G \wedge \neg(\neg F \wedge H)) \vee ((\neg F \wedge H) \wedge \neg(\neg F \wedge H))\\
&\Leftrightarrow (G \wedge (F \vee \neg H)) \vee 0\\
&\Leftrightarrow (G \wedge (F \vee \neg H))
\end{aligned}
$$

4. $\neg(\forall y(\exists z(\neg A(z,y) \wedge B(y))))$

$$
\begin{aligned}
&\Leftrightarrow \exists y \neg(\exists z(B(y) \wedge \neg A(z,y)))\\
&\Leftrightarrow \exists y \forall z \neg(B(y) \wedge \neg A(z,y))\\
&\Leftrightarrow \exists y \forall z(\neg B(y) \vee A(z,y))\\
&\Leftrightarrow \exists y(\neg B(y) \vee \forall z A(z,y))
\end{aligned}
$$

▸ Exercice 2. ***Séquents***

On rappelle les règles de réduction des séquents.

$$
\begin{array}{llll}
G_\vee : \frac{\Gamma,F\vDash\Delta \quad \Gamma,G\vDash\Delta}{\Gamma,F\vee G\vDash\Delta} & D_\vee : \frac{\Gamma\vDash\Delta,F,G}{\Gamma\vDash\Delta,F\vee G} & \\
G_\wedge : \frac{\Gamma,F,G\vDash\Delta}{\Gamma,F\wedge G\vDash\Delta} & D_\wedge : \frac{\Gamma\vDash\Delta,F \quad \Gamma\vDash\Delta,G}{\Gamma\vDash\Delta,F\wedge G} & \\
G_\rightarrow : \frac{\Gamma\vDash\Delta,F \quad \Gamma,G\vDash\Delta}{\Gamma,F\rightarrow G\vDash\Delta} & D_\rightarrow : \frac{\Gamma,F\vDash\Delta,G}{\Gamma\vDash\Delta,F\rightarrow G} & \\
G_\neg : \frac{\Gamma\vDash\Delta,F}{\Gamma,\neg F\vDash\Delta} & D_\neg : \frac{\Gamma,G\vDash\Delta}{\Gamma\vDash\Delta,\neg G} & \\
G_\forall : \frac{\Gamma,A(t),\forall xA(x)\vDash\Delta}{\Gamma,\forall xA(x)\vDash\Delta} & D_\forall : \frac{\Gamma\vDash A(a),\Delta}{\Gamma\vDash\forall xA(x),\Delta} & \text{où } a \text{ est une variable fraîche}\\
G_\exists : \frac{\Gamma,A(a)\vDash\Delta}{\Gamma,\exists xA(x)\vDash\Delta} & D_\exists : \frac{\Gamma\vDash A(t),\exists xA(x),\Delta}{\Gamma\vDash\exists xA(x),\Delta} & \text{et } t \text{ est un terme}
\end{array}
$$

Prouver les formules suivantes par la méthode des séquents.

1. $F \rightarrow ((\neg F \vee \neg G) \rightarrow (F \wedge \neg G))$

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$F, \neg F \vee \neg G \vDash F$}
\AxiomC{}
\UnaryInfC{$F \vDash F, \neg G$}
\UnaryInfC{$F, \neg F \vDash \neg G$}
\AxiomC{}
\UnaryInfC{$F, \neg G \vDash \neg G$}
\BinaryInfC{$F, \neg F \vee \neg G \vDash \neg G$}
\BinaryInfC{$F, \neg F \vee \neg G \vDash F \wedge \neg G$}
\UnaryInfC{$F \vDash (\neg F \vee \neg G) \to (F \wedge \neg G)$}
\UnaryInfC{$\vDash F \to ((\neg F \vee \neg G) \to (F \wedge \neg G))$}
\end{prooftree}
$$

2. $(F \to G) \to ((\neg G \wedge F) \to H)$

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$\neg G, F \vDash F, H$}
\AxiomC{}
\UnaryInfC{$G, F \vDash G, H$}
\UnaryInfC{$G, \neg G, F \vDash H$}
\BinaryInfC{$F \to G, \neg G, F \vDash H$}
\UnaryInfC{$F \to G, \neg G \wedge F \vDash H$}
\UnaryInfC{$F \to G \vDash (\neg G \wedge F) \to H$}
\UnaryInfC{$\vDash (F \to G) \to ((\neg G \wedge F) \to H)$}
\end{prooftree}
$$

3. $(\exists x \forall y A(y,x)) \to (\forall x \exists y A(x,y))$

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$A(b,a), \forall y A(y,a) \vDash A(b,a), \exists y A(b,y)$}
\UnaryInfC{$A(b,a), \forall y A(y,a) \vDash \exists y A(b,y)$}
\UnaryInfC{$\forall y A(y,a) \vDash \exists y A(b,y)$}
\UnaryInfC{$\forall y A(y,a) \vDash \forall x \exists y A(x,y)$}
\UnaryInfC{$\exists x \forall y A(y,x) \vDash \forall x \exists y A(x,y)$}
\UnaryInfC{$\vDash (\exists x \forall y A(y,x)) \to (\forall x \exists y A(x,y))$}
\end{prooftree}
$$

4. $(\exists x \forall y (A(x,y) \wedge B(y))) \to (\forall x (B(x) \wedge \exists y A(y,x)))$

$$
\begin{prooftree}
\AxiomC{}
\UnaryInfC{$A(a,b), B(b), \forall y (A(a,y) \wedge B(y)) \vDash B(b)$}
\AxiomC{}
\UnaryInfC{$A(a,b), B(b), \forall y (A(a,y) \wedge B(y)) \vDash A(a,b), \exists y A(y,b)$}
\UnaryInfC{$A(a,b), B(b), \forall y (A(a,y) \wedge B(y)) \vDash \exists y A(y,b)$}
\BinaryInfC{$A(a,b), B(b), \forall y (A(a,y) \wedge B(y)) \vDash B(b) \wedge \exists y A(y,b)$}
\UnaryInfC{$A(a,b) \wedge B(b), \forall y (A(a,y) \wedge B(y)) \vDash B(b) \wedge \exists y A(y,b)$}
\UnaryInfC{$\forall y (A(a,y) \wedge B(y)) \vDash B(b) \wedge \exists y A(y,b)$}
\UnaryInfC{$\forall y (A(a,y) \wedge B(y)) \vDash \forall x (B(x) \wedge \exists y A(y,x))$}
\UnaryInfC{$\exists x \forall y (A(x,y) \wedge B(y)) \vDash \forall x (B(x) \wedge \exists y A(y,x))$}
\UnaryInfC{$\vDash (\exists x \forall y (A(x,y) \wedge B(y))) \to (\forall x (B(x) \wedge \exists y A(y,x)))$}
\end{prooftree}
$$

▸ **Exercice 3.** ***Récursion***

Écrire le code CAML des fonctions récursives suivantes.

1. *la factorielle d'un entier :* `val fact : int -> int`.

✂ -

```
let rec fact = function
  | 0 -> 1
  | n -> n*(fact (n-1))
```

Récursion terminale (plus efficace car on fait l'appel récursif en dernier) ; le résultat intermédiaire est passé en argument :

```
let fact =
  let rec aux r = function
    | 0 -> r
    | n -> aux (n*r) (n-1) in
  aux 1
```

- ✂

2. *la puissance d'un entier par un autre :* `val puiss : int -> int -> int`.

✂ -

```
let rec puiss x = function
  | 0 -> 1
  | y -> x*(puiss x (y-1))
```

Récursion terminale :

```
let puiss x =
  let rec aux r = function
    | 0 -> r
    | n -> aux (x*r) (n-1) in
  aux 1
```

- ✂

3. *la suite de Fibonacci, définie par* $u_0 = u_1 = 1$ *et* $u_n = u_{n-1} + u_{n-2}$ *pour* $n > 1$ *:* `val fibo : int -> int`.

✂ -

```
let rec fibo = function
  | 0 | 1 -> 1
  | n -> fibo (n-1) + fibo (n-2)
```

On remarque que cette fonction n'est pas efficace car à chaque étape on fait deux appels récursifs, ce qui fait que le calcul `fibo n` va appeler la fonction en tout autour de 2^n. Une solution utilise la fonction suivante, qui ne fait qu'un appel.

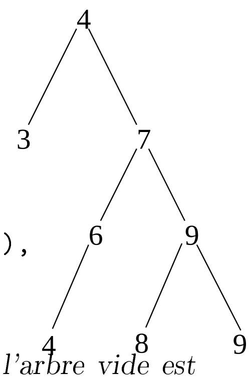
- ✂

4. *une fonction qui renvoie un couple composé des deux termes consécutifs* (u_n, u_{n+1}) *de la suite de Fibonacci (sans utiliser la précédente) :* `val fibo2 : int -> int * int`.

```
let rec fibo2 = function
  | 0 -> (1, 1)
  | n -> let (x, y) = fibo2 (n-1) in (y, x+y)
```
Récursion terminale :
```
let fibo2 =
  let rec aux (x, y) = function
    | 0 -> (x, y)
    | n -> aux (y, x+y) (n-1) in
  aux (1, 1)
```

▸ **Exercice 4.** ***Arbres***

On définit le type suivant d'arbres binaires étiquetés par des entiers :




```
type arbre = Vide | Noeud of int * arbre * arbre
```

Ainsi, l'arbre a représenté ci-contre est défini par :

```
let a = Noeud (4, Noeud (3, Vide, Vide),
                  Noeud (7, Noeud (6, Noeud (4, Vide, Vide), Vide),
                            Noeud (9, Noeud (8, Vide, Vide),
                                      Noeud (9, Vide, Vide))))
```

1. *On définit la hauteur d'un arbre de la façon suivante : la hauteur de l'arbre vide est 0 ; celle d'un autre arbre est 1 plus la hauteur maximale de ses deux sous-arbres. Par exemple, la hauteur de a est 4. Écrire une fonction récursive renvoyant la hauteur d'un arbre :* `val hauteur : arbre -> int`.

```
let rec hauteur = function
  | Vide -> 0
  | Noeud (_, g, d) -> let max a b = if a > b then a else b in
      1+(max (hauteur g) (hauteur d))
```

2. *Supposons que notre arbre soit trié, c'est à dire que l'étiquette d'un nœud soit strictement supérieure à toutes celles de son fils gauche et inférieure ou égale à celle de son fils droit (c'est le cas dans l'exemple). Écrire une fonction qui renvoie vrai si un entier donné est présent dans un arbre, faux sinon :* `val cherche : int -> arbre -> bool`.

```
let rec cherche i = function
  | Vide -> false
  | Noeud (j, _, _) when i = j -> true
  | Noeud (j, g, _) when i < j -> cherche i g
  | Noeud (_, _, d) -> cherche i d
  ;;
```

3. *Écrire une fonction récursive qui renvoie la somme de tous les entiers présents dans l'arbre :* `val somme : arbre -> int`.

```
let rec somme = function
  | Vide -> 0
  | Noeud (i, g, d) -> i+somme g+somme d
```

4. *Écrire une fonction récursive qui prend en argument une fonction et un arbre, applique la fonction à chaque nœud de l'arbre, et renvoie l'arbre correspondant :* `val map : (int -> int) -> arbre -> arbre`. *Par exemple,* `map factorielle a` *renverrait :*

```
Noeud (24, Noeud (6, Vide, Vide),
           Noeud (5040, Noeud (720, Noeud (24, Vide, Vide), Vide),
                        Noeud (362880, Noeud (40320, Vide, Vide),
                                       Noeud (362880, Vide, Vide))))
```

```
let rec map f = function
  | Vide -> Vide
  | Noeud (i, g, d) -> Noeud (f i, map f g, map f d)
```