Corrigé du DS2

Représentations d'ensembles avec des intervalles

Partie A – Fonctions générales

```
# type intervalle == int * int;;
Type intervalle defined.

# let bien_formé (i : intervalle) = fst i <= snd i;;
bien_formé : intervalle -> bool = <fun>
```

A.1

```
# let disjoints (i1 : intervalle) (i2 : intervalle) = snd i1 < fst i2 || snd i2 < fst i1;;
disjoints : intervalle -> intervalle -> bool = <fun>
```

A.2

```
# let fusion i1 i2 = min (fst i1) (fst i2), max (snd i1) (snd i2);;
fusion : 'a * 'b -> 'a * 'b -> 'a * 'b = <fun>
```

Partie B – Représentation par des listes triées d'intervalles

```
# type liste == intervalle list;;
Type liste defined.
```

B.1

```
# let rec ajouter (i : intervalle) (l : liste) = match l with
        | [] -> [i]
        | a :: q when snd i < fst a -> i :: l
        | a :: q when fst i > snd a -> a :: (ajouter i q)
        | a :: q -> ajouter (fusion a i) q;;
ajouter : intervalle -> (int * int) list -> (int * int) list = <fun>
```

B.2

B.3

```
# let rec appartenir v = function
        | [] -> false
        | a :: q as l -> (fst a <= v && v <= snd a) || (appartenir v l);;
appartenir : 'a -> ('a * 'a) list -> bool = <fun>
```

B.4

```
# let rec verifier = function
        | [] -> true
        | a :: [] -> true
        | a :: b :: q -> (snd a < fst b) && verifier (b :: q);;
verifier : ('a * 'a) list -> bool = <fun>
```

Partie C – Représentation par des arbres binaires

C.1

```
# type arbre =
        | Vide
        | Feuille of intervalle
        | Noeud of arbre * arbre;;
Type arbre defined.

# let fusion_arbre = fun
        | Vide d -> d
        | g Vide -> g
        | g d -> Noeud (g, d);;
fusion_arbre : arbre -> arbre -> arbre = <fun>
```

C.2

C.3

```
# let rec englobant = function
        | Vide -> failwith "pas d'intervalleenglobant"
        | Feuille i -> i
        | Noeud (g, d) -> fst (englobant g), snd (englobant d);;
englobant : arbre -> int * int = <fun>
```

C.4

```
# let rec verifier = function
        | Vide | Feuille _ -> true
        | Noeud (g, d) -> (verifier g) & (verifier d)
              & (snd (englobant g) < fst (englobant d));;
verifier : arbre -> bool = <fun>
```

C.5

```
# let rec appartenir v = function
        | Vide -> false
        | Feuille i -> fst i <= v && v <= snd i
        | Noeud (g, d) -> appartenir v g || appartenir v d;;
appartenir : int -> arbre -> bool = <fun>
```

C.6

```
# let rec decouper v = function
        | Vide -> Vide, Vide, Vide
        | a when v < fst (englobant a) -> Vide, Vide, a
        | a when snd (englobant a) < v -> a, Vide, Vide
        | Feuille n as a -> Vide, a, Vide
        | Noeud (g, d) -> let g1, g2, g3 = decouper v g and d1, d2, d3 = decouper v d in
              fusion_arbre g1 d1, fusion_arbre g2 d2, fusion_arbre g3 d3;;
decouper : int -> arbre -> arbre * arbre * arbre = <fun>
```

C.7

```
# let rec ajouter i a =
        let rec ajouter_aux i = function
                | Vide -> Vide, i, Vide
                | Feuille (l, r) as f when r < fst i -> f, i, Vide
                | Feuille (l, r) as f when snd i < l -> Vide, i, f
                | Feuille (l, r) -> Vide, fusion i (l, r), Vide
                | Noeud (g, d) ->
                        let g1, g2, g3 = ajouter_aux i g and d1, d2, d3 = ajouter_aux i d in
                                fusion_arbre g1 d1, fusion g2 d2, fusion_arbre g3 d3
in let g', i', d' = ajouter_aux i a in
        fusion_arbre (fusion_arbre g' (Feuille i')) d';;
ajouter : int * int -> arbre -> arbre = <fun>
```

Partie D – D'une représentation à l'autre

D.1

```
# let rec arbre_vers_liste = function
        | Vide -> []
        | Feuille n -> [n]
        | Noeud (g, d) -> (arbre_vers_liste g) @ (arbre_vers_liste d);;
arbre_vers_liste : arbre -> intervalle list = <fun>
```

D.2

```
# let rec liste_vers_arbre = function
        | [] -> Vide
        | [a] -> Feuille a
        | a :: q -> Noeud (Feuille a, (liste_vers_arbre q));;
liste_vers_arbre : intervalle list -> arbre = <fun>
```