On désire gérer les périodes d'occupation d'une salle. Il existe différentes structures d'arbres pour effectuer ce genre de gestion efficacement, en voici une possible :
On code un intervalle $[a, b]$ (où $a < b$ sont deux réels) par le couple `(a, b)`. On introduit donc le type :

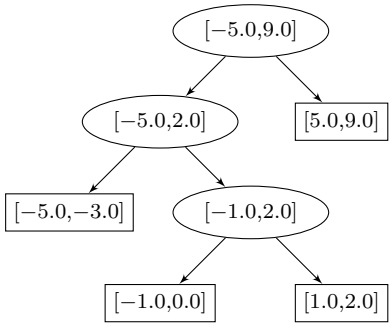
```
type intervalle = float * float;;
```

On utilisera pour stocker les plages d'occupation de la salle un arbre binaire dont les nœuds sont étiquetés par les intervalles. Les plages horaires réservées sont codées au niveau des feuilles. Tout nœud qui n'est pas une feuille possède exactement deux fils.

```
type arbre =
  | Vide
  | Feuille of intervalle
  | Noeud of arbre * intervalle * arbre;;
```

On dit que l'intervalle $(debut, fin)$ est *valide* si $debut < fin$.
On dit que l'arbre est *valide* s'il possède la propriété suivante : pour tout noeud n possédant deux fils f et g, si f est étiqueté par l'intervalle $[a_g, b_g]$ et si g est étiqueté par l'intervalle $[a_d, b_d]$, alors $a_g < b_g < a_d < b_d$, et n est étiqueté par l'intervalle $[a_g, b_d]$ (on dit que $[a_g, b_d]$ est l'*intervalle englobant* les intervalles $[a_g, b_g]$ et $[a_d, b_d]$).

Exemple : les plages d'occupation de la salle sont codés par les intervalles $[-5, -3]$, $[-1, 0]$, $[1, 2]$ et $[5, 9]$. Il existe différents arbres valides codant ces intervalles d'occupation, en voici un possible, les feuilles étant encadrées dans des rectangles et les nœuds internes dans des ovales :



Le constructeur `Vide` ne sert qu'à désigner l'arbre vide : il ne doit pas être utilisé dans un arbre qui n'est pas vide. Par exemple la construction `Noeud (Feuille (3.0, 5.0), (3.0, 5.0), Vide)` est interdite.

1) Coder l'arbre donné dans l'exemple précédent. Proposer (sans le coder) un arbre valide de géométrie différente qui possède les mêmes feuilles.

2) Écrire une fonction :

   ```
   englobe : arbre -> intervalle
   ```

   qui calcule le plus petit intervalle valide (pour l'inclusion) qui englobe tous les intervalles de réservation. Pour l'arbre donné en exemple, la valeur de retour doit être $(-5.0, 9.0)$
   Pour l'arbre vide on renverra l'intervalle $(0.0, -1.0)$ (qui n'est pas valide).

3) Écrire une fonction :

   ```
   est_valide : arbre -> bool
   ```

   qui permet de déterminer si un arbre est valide ou non.

4) On désire savoir si à une date x donnée la salle est occupée ou non. Écrire une fonction :

```
occupe : float -> arbre -> bool
```

qui effectue cette opération.

5) Définir une fonction :

```
intervalle_occupation : float -> arbre -> intervalle
```

qui permet à partir d'une date x donnée de déterminer l'intervalle d'occupation de la salle qui contient x, c'est–à–dire l'intervalle stocké dans une feuille et qui contient x. Si la salle n'est pas occupée à la date x on renverra l'intervalle `(0.0, -1.0)`.

6) Écrire une fonction :

```
intervalle_disponible : intervalle -> arbre -> bool
```

qui permet de savoir si la salle est vide pendant toute la durée d'un intervalle passé en paramètre. On considère que l'intervalle passé en paramètre est valide.

7) Écrire une fonction :

```
intervalle_partiellement_disponible : intervalle -> arbre -> bool
```

qui permet de savoir si, pendant la durée de l'intervalle passé en paramètre, il existe au moins un moment pendant lequel la salle est disponible.

8) Suppression d'un nœud : écrire une fonction

```
supprime : intervalle -> arbre -> arbre
```

qui permet de supprimer un intervalle $[\texttt{a}, \texttt{b}]$ *uniquement s'il s'agit d'une feuille* : on ne fera la suppression que si la feuille `Feuille (a,b)` existe explicitement, sinon on renvoie l'arbre de départ. Attention à faire remonter la modification au niveau des nœuds internes : l'arbre obtenu doit rester valide.

9) Écrire une fonction :

```
fusionne : arbre -> arbre -> arbre
```

qui prend en arguments deux arbres $\texttt{a}_1$ et $\texttt{a}_2$ pour lesquels on présuppose que toutes les dates apparaissant dans $\texttt{a}_1$ sont strictement inférieures à celles apparaissant dans $\texttt{a}_2$.
Cette fonction retourne l'arbre obtenu de la fusion de $\texttt{a}_1$ et de $\texttt{a}_2$. Vu ce qui est présupposé, la seule difficulté est de ne pas oublier que le constructeur `Vide` ne peut apparaître que dans l'arbre vide.

10) Écrire une fonction

```
scinde : intervalle -> arbre -> arbre * arbre * arbre
```

telle que si `a` est un arbre, alors `scinde (u, v) a` renvoie un triplet `(gauche, milieu, droite)` où :

- `gauche` est un arbre dont les feuilles sont les feuilles de `a` qui codent un intervalle `(i, j)` tel que $\texttt{j} < \texttt{u}$ (feuilles « à gauche » de l'intervalle)
- `droite` est un arbre dont les feuilles sont les feuilles de `a` qui codent un intervalle `(i, j)` tel que $\texttt{i} > \texttt{v}$ (feuilles « à droite » de l'intervalle)
- `milieu` est un arbre dont les feuilles sont les feuilles de `a` qui codent un intervalle `(i, j)` tel que $[\texttt{u}, \texttt{v}] \cap [\texttt{i}, \texttt{j}] \neq \emptyset$

11) En déduire une fonction :

```
insere : intervalle -> arbre -> arbre
```

qui permet l'insertion d'un intervalle dans un arbre. Au cas où l'intervalle que l'on insère intersecte d'autres intervalles stockés (au niveau des feuilles) dans l'arbre, on fusionnera les intervalles et on modifiera la géométrie de l'arbre en fonction de ces fusions. On privilégiera si possible des choix qui limitent la profondeur de l'arbre obtenu.