Composition d'Informatique
Les Principes des Langages de Programmation (INF 321)
Promotion 2013

Sujet proposé par Eric Goubault

30 juin 2014

L'examen se compose de cinq exercices, indépendants, sauf le troisième qui dépend des deux premiers, le tout noté sur 24 points (ramené ensuite à 20). Le barême donné reste néanmoins indicatif. On attachera une grande importance à la clarté, à la précision et à la concision de la rédaction et des explications.

1 Exercice 1 (8 points): Intervalles en JAVA

Dans cet exercice, on souhaite remplacer l'arithmétique usuelle sur les nombres réels, par une arithmétique ensembliste, permettant d'évaluer des expressions arithmétiques directement sur des intervalles de valeurs.

Mathématiquement, un intervalle, noté $[a, b]$, est donné par une paire de nombres réels (a, b), avec $a \leq b$. On va implémenter dans cet exercice des opérations arithmétiques $\oplus$, $\otimes$, $\oslash$ et $\ominus$ prenant deux intervalles I et J, et renvoyant $K = I \odot J$ ($\odot = \oplus$, ou $\otimes$, ou $\oslash$ ou $\ominus$) "optimal", c'est-à-dire le plus petit possible, tel que

$$\{i \cdot j \mid i \in I, i \in J\} \subseteq K \tag{1}$$

($\cdot$ étant l'opération correspondante à $\odot$ sur les nombres réels, c'est-à-dire que $\cdot$ est respectivement $+$, $\times$, $/$ ou $-$). On voit facilement que:

- $[a, b] \oplus [c, d] = [a + c, b + d]$
- $[a, b] \otimes [c, d] = [min(a \times c, a \times d, b \times c, b \times d), max(a \times c, a \times d, b \times c, b \times d)]$
- $[a, b] \oslash [c, d] = [a, b] \otimes (1 \oslash [c, d])$ où:

$$1 \oslash [c, d] = \begin{cases} \left[\frac{1}{d}, \frac{1}{c}\right] & \text{si } c, d \text{ sont strictement positifs} \\ \left[\frac{1}{c}, \frac{1}{d}\right] & \text{si } c, d \text{ sont strictement négatifs} \\ \text{indéfini} & \text{sinon} \end{cases}$$

- $[a, b] \ominus [c, d] = [a - d, b - c]$

Néanmoins, les ordinateurs ne disposent que de calculs approchants les calculs dans les nombres réels, avec les nombres flottants (type `float` en JAVA, sous-ensemble $\mathbb{F}$ de $\mathbb{R}$, mathématiquement). On calculera donc plutôt avec des intervalles ayant des bornes nombres flottants.

Pour s'assurer que la propriété (1) reste vraie, il nous faut changer un peu les opérations $\oplus$, $\otimes$, $\oslash$ et $\ominus$. Pour ce faire, on suppose donné une fonction $R^{\uparrow} : \mathbb{R} \to \mathbb{F}$, associant à tout réel x le plus petit nombre flottant supérieur ou égal à x, et une fonction $R^{\downarrow} : \mathbb{R} \to \mathbb{F}$, associant à tout réel x le plus grand nombre flottant inférieur ou égal à x. $R^{\uparrow}$ (respectivement $R^{\downarrow}$) est appelé arrondi vers l'infini (respectivement, vers moins l'infini), et on écrira par exemple ($+_{\mathbb{R}}$, $-_{\mathbb{R}}$ sont les additions et soustractions dans les nombres réels, $\mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d} \in \mathbb{F} \subseteq \mathbb{R}$):

- $$[a, b] \oplus [c, d] = [R^{\downarrow}(a +_{\mathbb{R}} c), R^{\uparrow}(b +_{\mathbb{R}} d)] \quad (2)$$

- $$[a, b] \oplus [c, d] = [R^{\downarrow}(a -_{\mathbb{R}} d), R^{\uparrow}(b -_{\mathbb{R}} c)] \quad (3)$$

Question 1.1: En adaptant les formules ci-dessus, donner les règles de calcul pour $[a, b] \otimes [c, d]$ et $[a, b] \oslash [c, d]$, prenant deux intervalles de nombres flottants et renvoyant le produit et la division d'intervalles, vérifiant la propriété (1).

Correction:

- $$[a, b] \otimes [c, d] = [\quad \begin{array}{l} min(R^{\downarrow}(a \times_{\mathbb{R}} c), R^{\downarrow}(a \times_{\mathbb{R}} d), R^{\downarrow}(b \times_{\mathbb{R}} c), R^{\downarrow}(b \times_{\mathbb{R}} d)), \\ max(R^{\uparrow}(a \times_{\mathbb{R}} c), R^{\uparrow}(a \times_{\mathbb{R}} d), R^{\uparrow}(b \times_{\mathbb{R}} c), R^{\uparrow}(b \times_{\mathbb{R}} d))] \end{array}$$

- $$[a, b] \oslash [c, d] = [a, b] \otimes (1 \oslash [c, d])$$

  où:

  $$1 \oslash [c, d] = \begin{cases} [R^{\downarrow}(\frac{1}{d}), R^{\uparrow}(\frac{1}{c})] & \text{si c, d sont strictement positifs} \\ [R^{\downarrow}(\frac{1}{c}), R^{\uparrow}(\frac{1}{d})] & \text{si c, d sont strictement négatifs} \\ \text{indéfini} & \text{sinon} \end{cases}$$

  De fait l'énoncé marque distingue les cas c, d strictement positifs et c, d strictement négatifs, ce qu'il n'y a pas lieu d'être. Les deux réponses possibles ont été acceptées.

On va par la suite définir une classe `FloatIEEE` de nombres flottants avec des opérations d'addition, de soustraction, de multiplication et de division, paramétrées par un mode d'arrondi. Par exemple, quand le mode d'arrondi est fixé à `UP`, l'addition `plus` de deux nombres flottants a et b calculera $R^{\uparrow}(a +_{\mathbb{R}} b)$. Quand le mode d'arrondi est fixé à `DOWN`, l'addition de deux nombres flottants a et b calculera $R^{\downarrow}(a +_{\mathbb{R}} b)$.

Question 1.2: Définir un type de données Java `Direction` permettant de représenter naturellement les deux modes d'arrondi, `UP` et `DOWN`.

Correction: On utilise un type enum:

```
public enum Direction {
  UP, DOWN
}
```

On suppose donnée une classe de nombres flottants:

```
public class FloatIEEE {
  ...
  FloatIEEE(float x) { ... }
  public FloatIEEE getVal() { ... }
  public FloatIEEE setVal(float x) { ... }
  public static FloatIEEE
         plus(FloatIEEE x, FloatIEEE y, Direction d)
           throws OutOfRangeException { ... }
  public static FloatIEEE
         minus(FloatIEEE x, FloatIEEE y, Direction d)
           throws OutOfRangeException { ... }
  public static FloatIEEE
         div(Float IEEE x, FloatIEEE y, Direction d)
           throws DivByZeroException, OutOfRangeException
```

```
            { ... }
  public static FloatIEEE
        times(Float IEEE x, FloatIEEE y, Direction d)
          throws OutOfRangeException { ... }
}
```

Ainsi, pour des variables `x` et `y` de type `FloatIEEE`, on pourra calculer $R^{\uparrow}(\mathtt{x}+_{\mathbb{R}}\mathtt{y})$ et $R^{\downarrow}(\mathtt{x}\cdot_{\mathbb{R}}\mathtt{y})$) par des appels à `plus(x,y,UP)` et à `plus(x,y,DOWN)`, respectivement. De même, on pourra utiliser `minus(x,y,UP)` et `minus(x,y,DOWN)`, `times(x,y,UP)` et `times(x,y,DOWN)`), `div(x,y,UP)` et `div(x,y,DOWN)`. Ces opérations renverront une exception `OutOfRangeException` si le calcul rend un nombre non représentable en machine, et `DivByZeroException` si on divise un flottant par zéro.

On souhaite maintenant utiliser la classe `FloatIEEE` pour implémenter une classe intervalle de nombres flottants.

Question 1.3: Définir la classe `FloatInterval` d'intervalles de nombres flottants, comme un type produit de deux `FloatIEEE`.

Correction: On a tout simplement:

```
public class FloatInterval {
  FloatIEEE a;
  FloatIEEE b;
}
```

Question 1.4: Définir un constructeur prenant deux `FloatIEEE` et renvoyant un `FloatInterval`.

Correction: On rajoute à la définition de classe précédente le constructeur:

```
public class FloatInterval {
  FloatIEEE a;
  FloatIEEE b;
  FloatInterval(FloatIEEE x, FloatIEEE y) { a=x; b=y; }
}
```

Certains élèves ont rajouté la vérification $x \leq y$, en intervertissant éventuellement les deux bornes dans le cas contraire (ce qui a été compté comme bon également).

Dans un premier temps, pour les questions 5 et 6, on ignorera les exceptions renvoyées par les méthodes de la classe `FloatIEEE`, on fera donc comme si `plus` (et de façon similaire `minus`, `times` et `div`) était du type:

```
public static FloatIEEE plus(FloatIEEE x, FloatIEEE y);
```

Question 1.5: Définir une méthode `plusInterval` prenant deux `FloatInterval` et renvoyant un `FloatInterval` somme des deux premiers, selon la définition (2) et une méthode `minusInterval` prenant deux `FloatInterval` et renvoyant un `FloatInterval` différence des deux premiers, selon la définition (3).

Comme dit plus haut, on ignorera dans cette question les cas où les bornes des intervalles résultats ne sont pas représentables en machine.

Correction: On écrit les équations 2 et 3 en ignorant les exceptions:

```
public static FloatInterval plusInterval(FloatInterval I, FloatInterval J) {
    FloatInterval K;
    FloatIEEE inf, sup;
    if (I == null) return null;
```

```
    if (J == null) return null;
    inf = FloatIEEE.plus(I.a, J.a, DOWN);
    sup = FloatIEEE.plus(I.b, J.b, UP);
    K = new FloatInterval(inf, sup);
    return K;
}

public static FloatInterval minusInterval(FloatInterval I, FloatInterval J) {
    FloatInterval K;
    FloatIEEE inf, sup;
    if (I == null) return null;
    if (J == null) return null;
    inf = FloatIEEE.minus(I.a, J.b, DOWN);
    sup = FloatIEEE.minus(I.b, J.a, UP);
    K = new FloatInterval(inf, sup);
    return K;
}
```

Question 1.6: Définir une méthode `timesInterval` prenant deux `FloatInterval` et renvoyant un `FloatInterval` produit des deux premiers, selon la question 1.1 et définir une méthode `divInterval` prenant deux `FloatInterval` et renvoyant un `FloatInterval` division des deux premiers, selon la question 1 (on pourra s'aider d'une méthode `FloatInterval oneOver()`, à écrire, qui calculera l'inverse du `FloatInterval` sur lequel elle s'applique).

Comme dit plus haut, on ignorera dans cette question les cas où les bornes des intervalles résultats ne sont pas représentables en machine, par contre, on renverra `null` si on cherche à diviser par un intervalle contenant 0.

Correction: On écrit encore une fois de façon directe les équations correspondantes (en rajoutant néanmoins une fonction `Fmin` et `Fmax` pour calculer le minimum, respectivement maximum, de deux `FloatIEEE`):

```
public static FloatIEEE Fmin(FloatIEEE x, FloatIEEE y) {
  if (x.getVal() <= y.getVal())
    return x;
  else
    return y;
}

public static FloatIEEE Fmin(FloatIEEE x, FloatIEEE y) {
  if (x.getVal() <= y.getVal())
    return y;
  else
    return x;
}

public static FloatInterval timesInterval(FloatInterval I, FloatInterval J) {
    FloatInterval K;
    FloatIEEE inf, sup;
    if (I == null) return null;
    if (J == null) return null;
    inf = FloatIEEE.times(I.a, J.a, DOWN);
    inf = FloatIEEE.Fmin(inf, FloatIEEE.times(I.a, J.b, DOWN));
    inf = FloatIEEE.Fmin(inf, FloatIEEE.times(I.b, J.a, DOWN));
    inf = FloatIEEE.Fmin(inf, FloatIEEE.times(I.b, J.b, DOWN));
    sup = FloatIEEE.times(I.a, J.a, UP);
    sup = FloatIEEE.Fmax(inf, FloatIEEE.times(I.a, J.b, UP));
    sup = FloatIEEE.Fmax(inf, FloatIEEE.times(I.b, J.a, UP));
```

```
    sup = FloatIEEE.Fmax(inf, FloatIEEE.times(I.b, J.b, UP));
    K = new FloatInterval(inf, sup);
    return K;
}

public FloatInterval oneOver() {
    FloatInterval K;
    FloatIEEE inf, sup;
    if (a*b<=0)
      return null;
    if (a>0) {
      inf = div(new FloatIEEE(1), b, DOWN);
      sup = div(new FloatIEEE(1), a, UP);
    } else {
      inf = div(new FloatIEEE(1), a, DOWN);
      sup = div(new FloatIEEE(1), b, UP);
    }
    K = newFloatInterval(inf, sup);
    return K;
}

public static FloatInterval divInterval(FloatInterval I, FloatInterval J) {
    FloatInterval inv;
    if (I == null) return null;
    if (J == null) return null;
    inv = J.divOver();
    if (inv == null) return null;
    return timesInterval(I, inv);
}
```

On remarquera que `getVal()` aurait du retourner un `float` et non un `FloatIEEE` dans l'énoncé.

Question 1.7: Dans cette question, on traite maintenant les exceptions renvoyées par les méthodes de la classe `FloatIEEE`. Comment modifier les méthodes `plusInterval`, `minusInterval` et `timesInterval` pour renvoyer une exception `OutOfRangeException` si l'une des deux bornes calculées n'est pas représentable en machine?

Comment modifier la méthode `divInterval` afin qu'elle renvoie une exception `OutOfRangeException` si l'une des deux bornes n'est pas représentable en machine ou une exception `DivByZeroException` dans le cas où la division n'est pas définie?

Ecrire les bouts de code modifiés en conséquence.

Correction Les fonctions `plusInterval`, `minusInterval` et `timesInterval` se contentent de renvoyer les éventuelles exceptions lancées durant le calcul. C'est différent dans le cas de `divInterval` pour l'exception `DivByZeroException`: `divInterval` lance explicitement cette exception dès que $0 \in [\texttt{J.a}, \texttt{J.b}]$ et pas seulement quand `J.a`=0 ou `J.b`=0.

On traite juste le cas de `timesInterval` et `divInterval`, les cas de `plusInterval` et `minusInterval` étant similaires à celui de `timesInterval`:

```
public static FloatInterval timesInterval(FloatInterval I, FloatInterval J)
  throws OutOfRangeException {
    FloatInterval K;
    FloatIEEE inf, sup;
    if (I == null) return null;
    if (J == null) return null;
    inf = FloatIEEE.times(I.a, J.a, DOWN);
    inf = FloatIEEE.Fmin(inf, FloatIEEE.times(I.a, J.b, DOWN));
    inf = FloatIEEE.Fmin(inf, FloatIEEE.times(I.b, J.a, DOWN));
    inf = FloatIEEE.Fmin(inf, FloatIEEE.times(I.b, J.b, DOWN));
```

```
    sup = FloatIEEE.times(I.a, J.a, UP);
    sup = FloatIEEE.Fmax(inf, FloatIEEE.times(I.a, J.b, UP));
    sup = FloatIEEE.Fmax(inf, FloatIEEE.times(I.b, J.a, UP));
    sup = FloatIEEE.Fmax(inf, FloatIEEE.times(I.b, J.b, UP));
    K = new FloatInterval(inf, sup);
    return K;
}

public FloatInterval oneOver()
  throws OutOfRangeException, DivByZeroException {
    FloatInterval K;
    FloatIEEE inf, sup;
    if (a*b<=0)
      throws DivByZeroException;
    if (a>0) {
      inf = div(new FloatIEEE(1), b, DOWN);
      sup = div(new FloatIEEE(1), a, UP);
    } else {
      inf = div(new FloatIEEE(1), a, DOWN);
      sup = div(new FloatIEEE(1), b, UP);
    }
    K = newFloatInterval(inf, sup);
    return K;
}

public static FloatInterval divInterval(FloatInterval I, FloatInterval J)
  throws OutOfRangeException, DivByZeroException {
    FloatInterval inv;
    if (I == null) return null;
    if (J == null) return null;
    inv = J.divOver();
    if (inv == null) return null;
    return timesInterval(I, inv);
}
```

2 Exercice 2 (4 points) : Domaine des intervalles

On considère maintenant les intervalles de nombres réels augmentés des infinis. Formellement un intervalle $I \in \mathbb{I}$ est soit $[a, b]$, avec $a, b \in \mathbb{R} \cup \{-\infty, \infty\}$, $a \leq b$ soit un élément que noté $\perp$. L'ordre considéré sur les intervalles est l'ordre d'inclusion pour les éléments différents de $\perp$, c'est-à-dire que $[a, b] \leq [c, d]$ ssi $a \geq c$ et $b \leq d$, et sinon $\perp \leq I$, pour tout $I \in I$.

Question 2.1: Prouver que $\mathbb{I}$ muni de l'ordre $\leq$ défini plus haut est un treillis complet.

Correction: Soit $A \subseteq \mathbb{I}$ un sous-ensemble du domaine des intervalles. Que $\perp \in A$ ou pas, cela ne changera pas l'éventuel supremum de A, car $\perp$ est le plus petit élément de $\mathbb{I}$. On suppose donc que A est constitué d'intervalles $I = [a_I, b_I]$. Soit $A_\cup = [inf\{a_I \mid I \in A\}, sup\{b_I \mid I \in A\}]$ et $A_\cap = [sup\{a_I \mid I \in A\}, inf\{b_I \mid I \in A\}]$. Ces inf et ces sup de nombres réels augmentés des infinis existent toujours dans les réels augmentés des infinis, donc $A_\cup$ et $A_\cap$ sont bien définis. On prouve maintenant que $A_\cup = \cup\{I \in A\}$ et $A_\cap = \cap\{I \in A\}$ les sup et inf respectivement de l'ensemble d'intervalles A, avec l'ordre sur les intervalles.

On le vérifie ici juste pour $A_\cup$: dans un premier temps, on voit que $A_\cup$ est un majorant de A:

$$I \leq A_\cup$$

$(I \in A)$ car $a_I \geq inf\{a_J \mid J \in A\}$ et $b_I \leq sup\{a_J \mid J \in A\}$. Enfin, si $[a,b]$ majore tous les intervalles I de A, alors a minore tous les a_I, $I \in A$, et donc $inf\{a_J \mid J \in A\} \geq a$ par définition de l'inf. De même, b majore tous les b_I, $I \in A$, et donc $sup\{b_J \mid J \in A\} \leq b$ par définition du sup. Ceci implique que $A\cup \leq [a,b]$, et donc que $A_\cup$ est le plus petit majorant des éléments de A.

Soit $F : \mathbb{I} \to \mathbb{I}$ la fonctionnelle sur $\mathbb{I}$ qui à I associe $F(I) = ([-1000, 0] \cup (I \oplus [1,1])) \cap [-10000, 100]$ où $\oplus$ est l'extension naturelle de la somme d'intervalles définie à l'exercice 1, pour tenir compte des bornes infinies. Dans le cas qui nous intéresse:

$$\begin{array}{rcll} [a,b] \oplus [1,1] & = & [a+1, b+1] & \text{si } a, b \in \mathbb{R} \\ [a,\infty] \oplus [1,1] & = & [a+1, \infty] & \text{si } a \in \mathbb{R} \text{ et } b = \infty \\ [-\infty, b] \oplus [1,1] & = & [-\infty, b+1] & \text{si } b \in \mathbb{R} \text{ et } a = -\infty \\ [-\infty, \infty] \oplus [1,1] & = & [-\infty, \infty] & \text{sinon} \end{array}$$

Question 2.2: Prouver, en utilisant un théorème du cours, que F admet un plus petit point fixe.

Correction: Comme $\mathbb{I}$ est un treillis complet, on peut utiliser le théorème de Tarsky dès lors que F est croissante. On pourrait également utiliser le théorème de Kleene en prouvant également que F est continue (car $\mathbb{I}$ treillis complet implique $\mathbb{I}$ CPO). On donne ici la première solution.

On considère $[a,b] \leq [c,d]$ et on prouve que $F([a,b]) \leq F([c,d])$. Tout d'abord on note que:

$$F([a,b]) = \begin{cases} [sup(-10000, inf(-1000, a_I + 1)), inf(100, sup(0, b_I + 1))] & \text{si } a, b \in \mathbb{R} \\ [sup(-10000, inf(-1000, a_I + 1)), 100] & \text{si } a \in \mathbb{R} \text{ et } b = \infty \\ [-10000, inf(100, sup(0, b_I + 1))] & \text{si } b \in R \text{ et } a = -\infty \\ [-10000, 100] & \text{sinon} \end{cases}$$

Donc on regarde les cas suivants:

- $[a,b] = [c,d] = [-\infty, \infty]$, donc $F([a,b]) \leq F([c,d])$
- $[a,b] = [-\infty, b]$ et $[c,d] = [-\infty, d]$ (on passe le cas $[c,d] = [-\infty, \infty]$, facile), donc $F([a,b]) = [-10000, inf(100, sup(0, b_I + 1))]$, $F([c,d]) = [-10000, inf(100, sup(0, d_I + 1))]$, donc il suffit de prouver que $b_I \leq d_I$ implique $inf(100, sup(0, b_I + 1)) \leq inf(100, sup(0, d_I + 1))$. C'est vrai par des propriétés élémentaires des inf et des sup sur les réels (croissants en leurs deux arguments).
- $[a,b] = [a, \infty]$ et $[c,d] = [c, \infty]$ (on passe le cas $[c,d] = [-\infty, \infty]$, facile) ainsi $F([a,b]) = [sup(-10000, inf(-1000, a_I + 1)), 100]$, $F([c,d]) = [sup(-10000, inf(-1000, c_I + 1)), 100]$ et on conclut de même que $c_I \geq a_I$ implique

  $sup(-10000, inf(-1000, a_I + 1)) \leq sup(-10000, inf(-1000, c_I + 1))$.
- $[a,b]$ et $[c,d]$ avec a, b, c et d réels (finis). Donc $F([a,b]) = [sup(-10000, inf(-1000, a_I + 1)), inf(100, sup(0, b_I + 1))]$ et on utilise pour conclure encore la croissante du sup, de l'inf et de l'addition, donc des fonctions $x \to sup(-10000, inf(-1000, x + 1))$ et $x \to inf(100, sup(0, x + 1))$.
- $[a,b]$ avec a, b réels (finis), donc $F([a,b]) = [sup(-10000, inf(-1000, a_I + 1)), inf(100, sup(0, b_I + 1))]$ et:
  - $[c,d] = [-\infty, \infty]$: $F([c,d]) = [-10000, 100]$ et $[sup(-10000, inf(-1000, a_I + 1)), inf(100, sup(0, b_I + 1))] \leq [-10000, 100]$, trivialement.
  - $[c,d] = [-\infty, d]$ avec d réel (fini). On conclut comme précédemment pour la borne inf, et par la croissante de $x \to inf(100, sup(0, x + 1))$ pour la borne sup.
  - $[c,d] = [c, \infty]$ avec c réel (fini): de même que plus haut.

Question 2.3: Quel est le plus petit point fixe de F? Y en a t-il d'autres?

Correction: On écrit $F([a,b]) = [a,b]$. Dans le cas où a, b sont des réels (finis) on obtient les équations:

$$\begin{array}{lcl} sup(-10000, inf(-1000, a+1)) & = & a \\ inf(100, sup(0, b+1)) & = & b \end{array}$$

Donc $a = -10000$ si $inf(-1000, a+1) \leq -10000$ et $a = inf(-1000, a+1)$ sinon. Pour la deuxième équation, nécessairement on ne peut avoir $a = a+1$ donc $inf(-1000, a+1) = -1000$, $a \geq -1001$ et $a = -1000$. La première équation n'est pas possible car $a = -10000$ implique $inf(-1000, a+1) = -9999$ qui n'est pas inférieure ou égale à -10000.

On a également $b = 100$ si $sup(0, b+1) \geq 100$, c'est-à-dire si $b \geq 99$. Donc $b = 100$ est une solution. Sinon $sup(0, b+1) = b$ si $b \leq 99$, ce qui n'est pas possible.

Les autres cas $a = -\infty$ et $b = \infty$ n'apportent aucune possibilité de point fixe supplémentaire. Il n'y a donc qu'une solution, l'intervalle $[-1000, 100]$, qui est nécessairement le plus petit point fixe de F.

3 Exercice 3 (4 points): Implémentation du domaine des intervalles

On identifiera par la suite `null` avec le $\perp$ du domaine des intervalles de l'exercice 2.

Question 3.1: Programmer dans la classe `FloatInterval` de l'exercice 1, une méthode `FloatInterval sup(FloatInterval I, FloatInterval J)` (respectivement `FloatInterval inf(FloatInterval I, FloatInterval J)`, et `Boolean leq(FloatInterval I, FloatInterval J)`) qui implémente le sup (respectivement l'inf, et l'ordre sur les intervalles) des intervalles `I` et `J` dans le treillis $(\mathbb{I}, \leq)$ de l'exercice 2.

Correction: On a le code suivant:

```
public static FloatInterval sup(FloatInterval I, FloatInterval J) {
  FloatInterval K;
  FloatIEEE in, su;
  if (I==null) return J;
  if (J==null) return I;
  inf = FloatIEEE.Fmin(I.a, J.a);
  sup = FloatIEEE.Fmax(I.b, J.b);
  K = new FloatInterval(in, su);
  return K;
}

public static FloatInterval inf(FloatInterval I, FloatInterval J) {
  FloatInterval K;
  FloatIEEE in, su;
  if (I==null) return null;
  if (J==null) return null;
  inf = FloatIEEE.Fmax(I.a, J.a);
  sup = FloatIEEE.Fmin(I.b, J.b);
  if (in <= su)
    K = new FloatInterval(in, su);
  else
    K = null;
  return K;
}
```

```
public static Boolean leq(FloatInterval I, FloatInterval J) {
  if (I==null) return true;
  if (J==null) return (I==null);
  if (I.a.getVal() < J.a.getVal()) return false;
  if (I.b.getVal() > J.b.getVal()) return false;
  return true;
}
```

On donne l'interface Java suivante:

```
public interface Functional {
  public static FloatInterval F(FloatInterval I);
}
```

Question 3.2: Programmer la fonctionnelle F de cet exercice dans une classe implémentant l'interface `Functional`. On ne réécrira pas la méthode `plusInterval` que l'on supposera renvoyer `null` si au moins l'un de ses deux arguments est égal à `null`.

Correction: Le code suivant répond à la question:

```
public class FunctionF implements Functional {
 public static FloatInterval F(FloatInterval I) {
  return inf(sup(new FloatInterval(new FloatIEEE(-1000), new FloatIEEE(0)),
                 plus(I, new FloatInterval(new FloatIEEE(1), new FloatIEEE(1)))),
                  new FloatInterval(new FloatIEEE(-10000), new FloatIEEE(100)));
  }
}
```

Question 3.3: Créer une méthode `FloatInterval Lfp(Functional F)` pour la classe `FloatInterval` qui calcule, quand elle termine, le plus petit point fixe de toute fonctionnelle `F` de type `Functional`, en supposant `F` continue en son argument intervalle.

Correction: Le code de la méthode en question est:

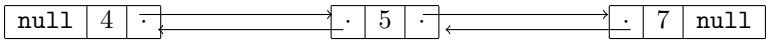
```
public static FloatInterval Lfp(Functional F) {
  FloatInterval Xn = null;
  FloatInterval Xsn = FunctionF.F(Xn);
  while (leq(Xn, Xsn)) {
    Xn = Xsn;
    Xsn = FunctionF.F(Xn);
  }
  return Xn;
}
```

Remarque: La fonctionnelle F de l'exercice 2 est en fait la sémantique "abstraite" (au sens de l'interprétation abstraite, inventée par Patrick et Radhia Cousot en 1976, au coeur de plusieurs systèmes de validation de programmes, utilisés dans l'industrie) pour synthétiser l'invariant de boucle du programme suivant, dans le domaine des intervalles:

```
/*@ requires x>=-1000 && x<=0
  @*/
  int P(int x) {
    while (-10000<=x && x<=100)
      x = x+1;
    return x;
  }
```

4 Exercice 4 (6 points) : Liste doublement chaînée

On souhaite implémenter dans cet exercice une liste d'entiers doublement chaînée, c'est-à-dire contenant, pour chaque noeud, un pointeur non seulement vers son successeur, mais également vers son prédécesseur. Comme dans le cours, on peut représenter graphiquement les liens entre noeuds par un graphe, par exemple, la liste `l` suivante est doublement chaînée:



Question 4.1: Définir une classe `DoublyLinkedList` implémentant les listes doublement chaînées d'entiers ainsi que son constructeur, prenant en argument un entier, et deux listes doublement chaînées.

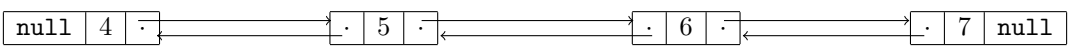
```
public class DoublyLinkedList {
  int n;
  DoublyLinkedList succ;
  DoublyLinkedList pred;
  DoublyLinkedList(i, s, p) {
    n=i;
    succ=s;
    pred=p;
  }
}
```

Question 4.2: Définir la liste `l` du début de cette section, avec des appels du constructeur de la question 4.1, et suffisamment de variables intermédiaires.

```
DoublyLinkedList N7 = new DoublyLinkedList(7, null, null);
DoublyLinkedList N5 = new DoublyLinkedList(5, N7, null);
DoublyLinkedList N4 = new DoublyLinkedList(4, N5, null);
N7.pred = N5;
N5.pred = N4;
```

Dans les questions qui suivent, on suppose que les listes doublement chaînées considérées sont strictement croissantes, c'est-à-dire que la valeur entière contenue dans un noeud est inférieure strictement à la valeur entière contenue dans le noeud suivant, et supérieure strictement à la valeur entière contenue dans le noeud précédent.

Question 4.3: Donner une implémentation de la méthode d'insertion `DoublyLinkedList Insert(int x)` d'un entier `x` dans une liste doublement chaînée d'entiers croissants, afin que le résultat soit toujours une liste doublement chaînée croissante. Ainsi `l.Insert(6)` doit renvoyer la liste `l1` suivante:



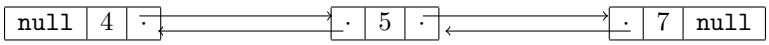
On est libre de choisir un mode de partage quelconque entre le résultat et l'argument de `Insert`, que l'on décrira en quelques mots. De même on est libre de choisir une solution itérative ou récursive.

Correction: On considère le code suivant:

```
public DoublyLinkedList Insert(int x) {
 DoublyLinkedList res;
 if (x<n) {
    res = new DoublyLinkedList(x,this,null);
    this.pred = res;
 }
 if (x==n)
    res = this;
 if (x>n) {
    if (succ==null) {
        succ = new DoublyLinkedList(x,null,this);
        res = this;
    }
    else {
       DoublyLinkedList s = succ.Insert(x);
       res = new DoublyLinkedList(n,s,pred);
       s.pred = res;
    }
 }
 return res;
}
```

Ici on a choisit de partager le résultat avec l'argument (pas de recopie du `this`). Le sujet ne spécifiait pas clairement si l'on considérait, comme dans la solution plus haut, qu'il y avait un "début" de liste à partir duquel on cherchait à insérer, uniquement à droite, ou si on devait éventuellement insérer à gauche. Ces différents cas de figure ont été comptés comme justes.

Question 4.4: Ecrire la méthode `DoublyLinkedList Delete(int x)` qui efface le noeud contenant la valeur `x` dans une liste doublement chaînée croissante. Ainsi, `l1.Delete(6)` sur la liste `l1` de la question précédente doit renvoyer la liste de la figure suivante:



De même qu'à la question 4.3, on pourra choisir de partager ou pas les cellules de la liste argument de `Delete` avec son résultat (que l'on qualifiera en quelques mots) et une solution itérative ou récursive.

Correction: On considère le code suivant:

```
public DoublyLinkedList Delete(int x) {
  DoublyLinkedList res;
  if (x == n) {
    res = succ;
    res.pred = null;
  }
  if (x < n)
    res = this;
  if (x > n) {
    if (succ != null) {
      DoublyLinkedList s = succ.Delete(x);
      res = new DoublyLinkedList(n, s, null);
      s. pred = res;
    }
  }
  return res;
}
```

Ici on a choisit de partager le résultat avec l'argument (pas de recopie du `this`).

5 Exercice 5 (4 points) : Preuve à la Hoare

Compléter les assertions JML associées aux codes suivants, que l'on justifiera brièvement, en citant brièvement les règles de vérification associées, du cours 7. On pourra, si on préfère, utiliser le format de la logique de Hoare du cours et compléter les assertions entre chaque instruction du code.

Question 5.1: On considère d'abord la méthode `F`:

```
/*@ requires Y...;
  @ ensures X==0;
  @*/
int F(int Y) {
  int X
  X = Y;
  return X;
}
```

Correction: La solution est:

```
/*@ requires Y==0;
  @ ensures X==0;
  @*/
int F(int Y) {
  int X
  X = Y;
  return X;
}
```

Question 5.2: On considère ensuite la méthode `G` suivante:

```
/*@ requires ...;
  @ ensures Y==5;
  @*/
int G(int Z, int W) {
  int Y;
  if (X == 0)
    Y = Z + 1;
  else
    Y = W + 2;
  return Y;
}
```

Correction: La solution est:

```
/*@ requires ((X==0) => (Z==4)) || ((X!=0) => (W==3));
  @ ensures Y==5;
  @*/
int G(int Z, int W) {
  int Y;
  if (X == 0)
```

```
    Y = Z + 1;
  else
    Y = W + 2;
  return Y;
}
```

En utilisant le formalisme de la preuve à la Hoare du cours, les assertions correspondant à chaque instruction sont:

```
{ (X = 0 ⇒ Z = 4) ∨ (X != 0 ⇒ W = 3) }
  if (X == 0)
  { Z = 4 }
    Y = Z + 1 ;
  { Z = 4 ∧ Y = 5 }
  else
  { W = 3 }
  Y = W + 2;
  { W = 3 ∧ Y = 5 }
{ Y = 5 }
```

(en utilisant successivement les règles pour les conditionnelles et pour les affectations).

Question 5.3: On consière enfin la méthode `H` suivante:

```
/*@ ensures Z=...;
  @*/
int Z(int x, int z) {
  int X = x;
  int Z = z;
  /*@ loop_invariant
    @    ...
    @*/
  while (X != 0) {
    Z = Z - 1;
    X = X - 1;
  }
  return Z;
```

Correction: La solution est:

```
/*@ ensures Z=z-x;
  @*/
int Z(int x, int z) {
  X = x;
  Z = z;
  /*@ loop_invariant
    @    x-X==z-Z;
    @*/
  while (X != 0) {
    Z = Z - 1;
    X = X - 1;
  }
  return Z;
```

Les assertions correspondant à chaque ligne du code, dans le format des preuves à la Hoare du cours est:

```
{ X = x ∧ Z = z }
  while (X != 0) {
    { (Z − X = z − x ∧ X != 0) ⇒ ((Z − 1) − (X − 1) = z − x) }
    Z = Z − 1;
    { Z − (X − 1) = z − x }
    X = X − 1;
    { Z − X = z − x }
  }
{ Z = z − x }
```

(en utilisant successivement les règles pour les boucles et pour les affectations).