Test
Corrigé

Exercice 1 Une fonction prend comme paramètres trois flottants représentant les longueurs des côtés d'un triangle. Elle renvoie ensuite une valeur entière indiquant s'il s'agit d'un triangle *scalène* (valeur 0), *isocèle* (valeur 1) ou *équilatéral* (valeur 2).

1.1. Produire une suite de tests pour ce programme : cas de test (Test Case), données de test (Test Data) et oracle.

Solution :

1. C1 : équilatéral ($a = b = c$)
   Données de test : a, b, c = 5, 5, 5
2. C2 : scalène ($a \neq b \wedge a \neq c \wedge b \neq c$)
   Données de test : (3, 4, 5)
3. C3 : isocèle $(a = b \vee a = c \vee b = c) \wedge \neg(a = b = c))$
   Données de test : (2, 2, 1), (3, 4, 3), (0.5, 1, 1)
4. C4 : invalides (nombres nuls ou négatifs, NaN, les trois réels ne permettent pas de construire un triangle (1, 2, 10 ne peuvent pas être les longueurs des côtés d'un triangle), etc.)

Pour l'oracle, voir les formules entre parenthèses.
On peut définir des fonctions :

— équilatéral(a, b, c) returns `a = b and b = c`

— scalène(a, b, c) returns `a != b and a != c and b != c`

— isocèle(a, b, c) returns ! équilatéral(a, b, c) and ! scalène(a, b, c)

— triangle(a, b, c) returns (a + b) > c and (b + c) > a and (c + a) > b

Ici, nous n'avons pas donné le résultat attendu car il était évident de part notre partition (le résultat de la fonction est 2, 0 ou 1). Pour C4, rien n'est précisé dans l'énoncé. Ce pourrait être une exception ou une autre valeur, par exemple -1.

1.2. Avez-vous fait seulement des tests fonctionnels ou aussi de robustesse ?

Solution : Puisque nous avons identifié les nombres négatifs, NaN... nous avons aussi fait du test de robustesse.

Si on pense au test aux limites, il faudrait certainement introduire un epsilon (constante) permettant de définir l'égalité de deux réels : $a = b \equiv |a - b| < \epsilon$

Exercice 2 Soit la spécification suivante :

```java
public static int search(List<?> list, Object element)
    // Effects: if list or element is null throw NullPointerException
    // if element is in list, returns the indice of one of its positions else -1
```

On considère la partition suivante basée sur la place de l'élément `element` dans la liste `list` :

— `element` est en début de `list`

— `element` est en fin de `list`

— `element` est à une position autre que début ou fin de `list`

2.1. Cette partition est-elle basée sur l'interface ou sur la sémantique de la fonction ?
Solution : Sur la sémantique car elle relie les deux paramètres en considérant que le deuxième devrait être un élément de la liste en premier paramètre.
2.2. Montrer que cette partition n'est pas disjointe.
Solution : Une liste réduite à un seul élément couvre les trois cas.
2.3. Montrer que cette partition n'est pas complète.
Solution : La liste vide ne correspond à aucune des catégories.
On peut aussi considérer une liste qui ne contient pas l'élément (qui est un cas nominal et non limite comme le précédent).
2.4. Proposer une nouvelle partition disjointe et complète.
Solution : Une classification qui s'appuie sur la sémantique pourrait être :

— l'élément n'est pas présent dans la liste
  — la liste est vide
  — la liste n'est pas vide

— l'élément est présent une et une seule fois
  — elt est en tête
  — elt est en fin
  — elt n'est ni en tête, ni en fin

— l'élément est présent plusieurs fois dans la liste
  — on peut envisager des cas similaires.

Faut-il donner une partition seulement sur l'interface ?
Remarque : l'oracle est simple lorsque la réponse est un entier différent de -1 : c'est un indice valide de la liste et l'élément à cet indice est égal à `element` ! Sinon, il faut parcourir la liste pour vérifier qu'elle ne contient pas l'élément si la fonction renvoie -1.

Exercice 3 On considère l'algorithme du listing 1.
3.1. Donner les nœuds du graphe de contrôle (Control Flow Graph).
Solution : Les noeuds du graphe sont les blocs d'instructions...
Indiquer les instructions qui peuvent générer des branchements.
Solution : Les instructions qui engendrent des branchements sont :

1. les 4 if : ligne 2, 5, 8 et 10.

```java
void myFun(int a, int b, int c, int x) {
    if (b < c) {
        int d = 2 * b;
        int f = 3 * c;
        if ( x >= 0 && a >=0) {
            d = x;
            int e = c;
            if (d == 0) {
                f = f - e;
                if (d < a) {
                    d = a + 1;
                } else {
                    d = a - 1;
                }
                System.out.println(a);
            }
        }
    }
}
```

Listing 1: Exemple d'algorithme

Pour chacune de ces instructions, indiquer ses décisions / branches et ses conditions.

Solution : Le graphe de contrôle est donné en figure 1.

Les instructions qui génèrent des branchements sont les losanges. Les décisions sont indiquées à l'intérieur des losanges. Les branches sortent des losanges. Les conditions sont les sous-expressions liées par des opérateurs booléens.

3.2. Donner les chemins de ce graphe de contrôle.

Solution : Voir le schéma précédent en considérant tous les chemins allant du début (start) à la fin (end).

3.3. Identifier les définitions des variables et leur portée.

Solution : Les définitions de variables et leur portée sont indiquées par des blocs dans la figure 2.

3.4. Pour chaque variable, donner l'ensemble de ses utilisations. Préciser si ce sont des calculs *c-use* (computation) ou des conditions *p-use* (predicate).

Solution :

Variable	*def*	*c-use*	*p-use*
a	1	11, 13, 15	5, 10
b	1	3	2
c	1	4, 7	2
x	1	6	5
d	3, 6, 11, 13		8, 10
f	4, 9	9	
e	7	9	

3.5. Donner l'ensemble des paires *def-use* pour chaque variable du programme.

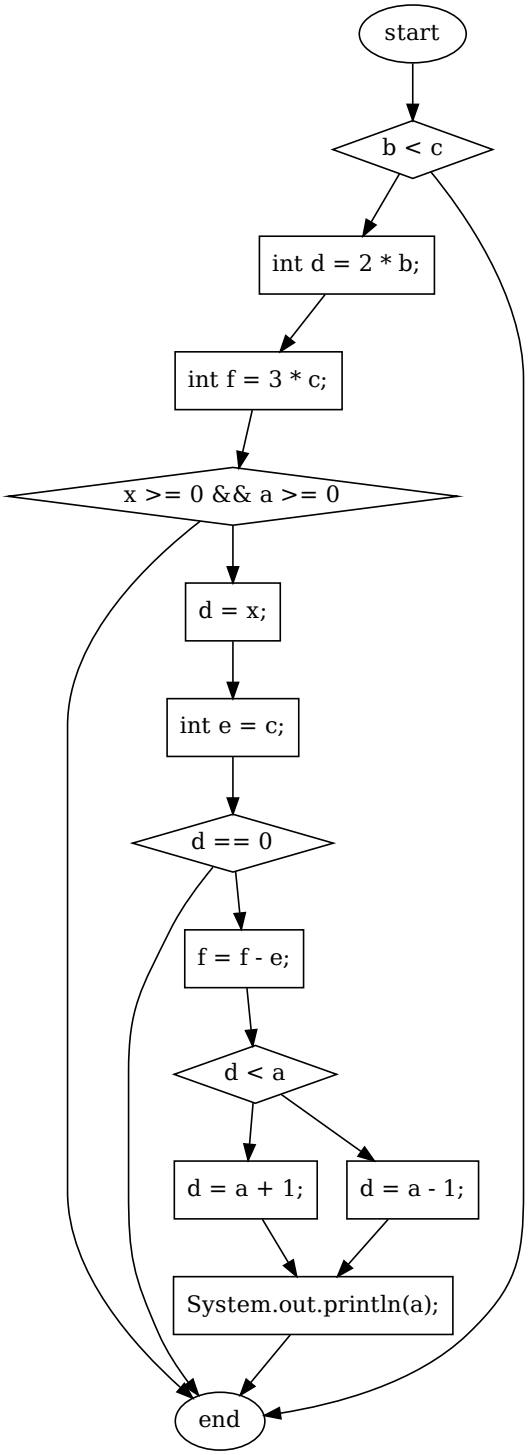

FIGURE 1 – Graphe de contrôle

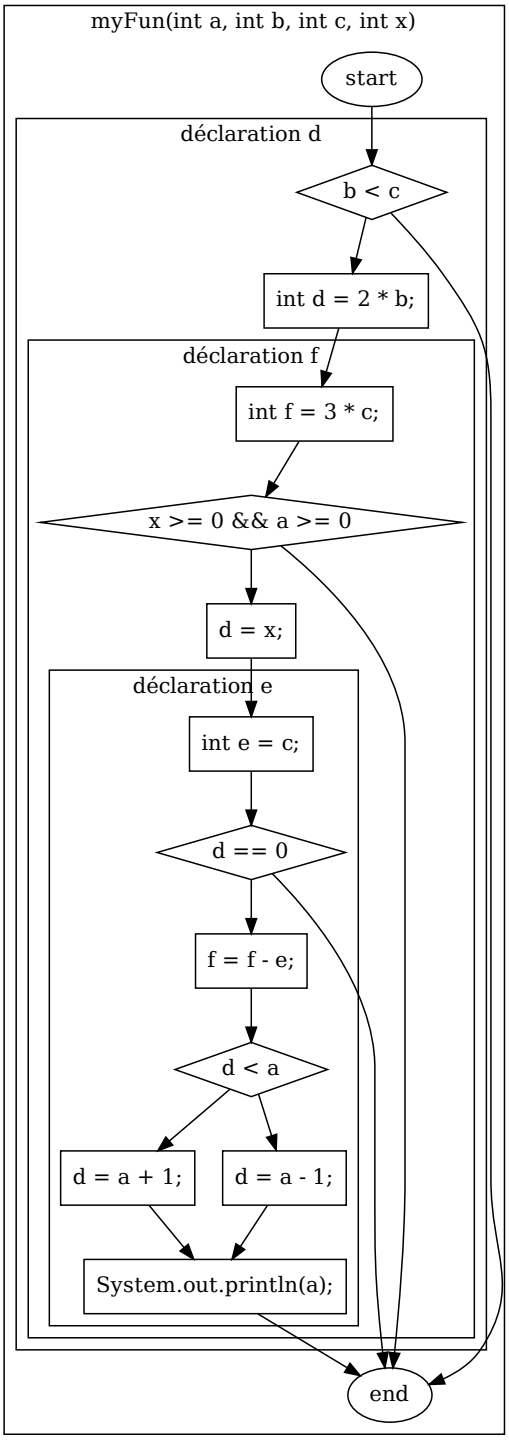

FIGURE 2 – Portée des variables

Solution :

Variable	*def-use*
`a`	(1, 5), (1, 10), (1, 11), (1, 13), (1, 15)
`b`	(1, 2), (1, 3)
`c`	(1, 2), (1, 4), (1, 7)
`x`	(1, 5), (1, 6)
`d`	(6, 8), (6, 10)
`f`	(4, 9)
`e`	(7, 9)

Exercice 4 Soit le programme ci-dessous :

```
1  void foo(boolean a, boolean b, boolean c) {
2      if (a || (b && c)) {
3          out.println("ok");
4      }
5      out.println("fin");
6  }
```

4.1. Donner les éléments à couvrir pour chacun des critères suivants : instructions (I), décisions (D), conditions (C), décisions/conditions (DC), conditions multiples (MC), MC/DC (Modified Condition/Decision Condition).

Solution :

- I : lignes 2, 3, 5
- D : condition de la ligne 2 valant une fois vrai (décision vraie) et une fois faux (décision fausse)
- C : chaque conditions simples de la condition de la ligne 2 prend les valeurs vrai et faux (décision simple)
- DC : combine D et C, autrement dit décision sur la condition complète et sur chaque condition simple
- MC : toutes les combinaisons des conditions simples de la condition de la ligne 2 (c'est-à-dire les valeurs vrai et faux pour chaque condition simple). Cela conduit à construire la table de vérité correspondant aux conditions simples.
- MC/DC : critère DC étendu en prenant toutes les valeurs des conditions simples qui provoquent le changement de la condition complète.

4.2. Donner les jeux de tests du programme couvrant les critères et illustrer qu'ils sont différents.

Solution :

- I : donner la valeur vrai à `a` est suffisant (toutes valeurs de `b` et `c` conviennent)
- D : la DT précédente (décision vraie) doit être complétée par une activation de la décision fausse. `a` doit prendre la valeur faux, `b` ou `c` doivent prendre la valeur faux

— C : nous pouvons prendre `(vrai, faux, faux)` et `(faux, vrai, vrai)`. La décision complète n'est pas couverte car la condition complète ne prend que la valeur vrai.

— DC : nous pouvons prendre les mêmes valeurs pour les trois variables. La décision complète est alors couverte.

— MC : il faut construire la table de vérité pour les trois variables.

— MC/DC : si `b` ou `c` prend la valeur faux, alors `a` change la décision complète quand elle change de valeur, il faut donc prendre des DTs pour ces 2 cas. Si `a` prend la valeur faux, alors les changements de `b` et `c` peuvent changer la décision complète, il faut donc prendre des DTs pour ces 2 cas. Nous pouvons donc prendre : `(vrai, faux, vrai)`, `(faux, faux, vrai)`, `(faux, faux, vrai)`, `(faux, vrai, vrai)`, `(faux, vrai, faux)`. Nous avons 6 DTs au lieu de 8 en MCs.

Exercice 5 Considérons l'algorithme de l'exercice 3 (listing 1).
5.1. Donner les éléments à couvrir pour chacun des éléments suivants : *all-defs*, *all-use*, *all-p-use*, *all-c-use*, *all-def-use-paths*.
Solution :

Critère	Lignes
all-defs	1, 3, 4, 6, 7, 9, 11, 13
all-use	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13, 15
all-p-use	2, 5, 8, 10
all-c-use	3, 4, 6, 7, 9, 11, 13, 15
all-def-use-paths	voir la table *def-use* de l'exercice 3

5.2. Donner des jeux de tests couvrant les critères et illustrer qu'ils sont différents.
Solution :

Critère	Lignes
all-defs	
all-use	
all-p-use	
all-c-use	
all-def-use-paths	

Exercice 6 Soit la méthode Java du listing 1.
6.1. Donner son graphe de contrôle.
6.2. Donner une suite de tests TS_n qui couvre tous les nœuds du graphe de contrôle.
6.3. La suite TS_n couvre-t-elle tous les arcs ? Si oui, indiquer les données de tests qui effectuent la couverture des arcs. Sinon, ajouter des données de test pour obtenir une suite de tests TS_a qui couvre tous les arcs.
6.4. Indiquer quelles sont les lignes de code correspondant aux définitions de la variable `result` (ensemble *defs(`result`)*). Même question pour les ensembles d'utilisation en calcul *c-use(`result`)* et d'utilisation en prédicats *p-use(`result`)*.
6.5. Donner une suite de tests TS_d qui couvre le critères *all-p-uses* pour `result`.

Listing 1 – La méthode Java maxSum

```
1  // Outputs result = 0 + 1 + 2 + ... + |value|
2  // if results > maxInt the error
3  static void maxSum(int maxInt, int value) {
4     int result = 0;
5     int i = 0;
6     if (value < 0) {
7        value = - value;
8     }
9     while (i < value && result <= maxInt) {
10       i++;
11       result = result + i;
12    }
13    if (result <= maxInt) {
14       System.out.println(result);
15    } else {
16       System.out.println("error");
17    }
18 }
```