

# Composition d'Informatique

## Mécanismes de la Programmation Orientée-Objet (INF371)

Promotion 2020

28 juin 2021

On accordera beaucoup d'importance à la clarté des réponses, ainsi qu'à la clarté et la concision du code.

### 1 Exercices

**Question 1** Soit la séquence d'instructions suivante :

- (1) PUSH(9)
- (2) PUSH(1)
- (3) ADD
- (4) PUSH(0)
- (5)
- (6) GTO(2)
- (7) STOP

a) Proposez une instruction à mettre dans l'emplacement vide pour que le programme boucle indéfiniment.

b) Proposez une instruction à mettre dans l'emplacement vide pour que le programme échoue sur un débordement de pile.

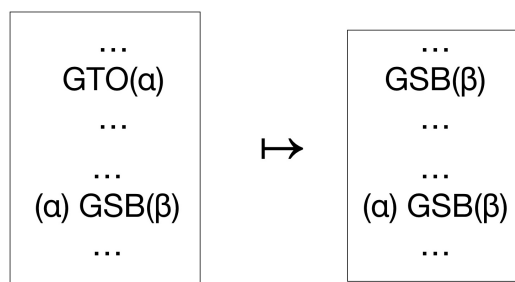
c) Proposez une instruction à mettre dans l'emplacement vide pour que le programme termine.  $\diamond$

*Solution.* a) POP ou ADD...

b) PUSH(0)...

c) GTO(7)  $\square$

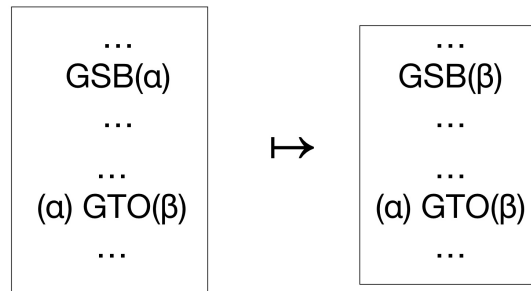
**Question 2** Un collègue vous propose l'optimisation suivante sur du code machine (remplacer un GTO par un GSB quand on est dans la situation de gauche).



Cette optimisation vous semble-t-elle correcte ? Sinon pourquoi ?  $\diamond$

*Solution.* Elle est fausse (on sera au mauvais endroit après le RET).  $\square$

**Question 3** Même question pour cette optimisation (remplacer un  $\text{GSB}(\alpha)$  par un  $\text{GSB}(\beta)$  quand on est dans la situation de gauche).



*Solution.* Elle est correcte. □

**Question 4** Soit la classe suivante :

```
class A {
    static int x = 0;
    int a, b;
    String s;
    A(int a, int b, String s) {
        this.a = a; this.b = b; this.s = s;
    }
    int f() { return (a + b); }
}
```

Quand on exécute l'instruction suivante, quelle est le nombre de mots mémoire de la séquence vers laquelle pointe la variable  $v$  :  $A \ v = \text{new } A(2, 3, "a");$  ?

Et après  $A \ v = \text{new } A(2, 3, "aaaa");$  ? ◇

*Solution.* Quatre mots dans les deux cas (trois champs et l'adresse de la table d'adressage). □

## 2 Expressions booléennes

*Dans cette partie on sera amené à utiliser quelques classes de la bibliothèque de Java dont les caractéristiques principales sont rappelées en appendice à la fin du sujet.*

On va représenter et manipuler des expressions booléennes. Ces expressions peuvent être :

- Soit les constantes `true` et `false`,
- soit des variables, qu'on notera  $x, y, z, \dots$
- soit construites avec les connecteurs  $\wedge$  (et),  $\vee$  (ou) et  $\neg$  (non).

Des exemples d'expressions sont donc :  $(\neg x \vee y)$ ,  $(x \wedge (\neg y \vee z))$ ,  $\neg(x \vee \text{false})$  etc...

On va représenter ces expressions de manière semblable à ce qu'on a vu en cours. Une expression sera représentée par un objet de classe `Expr`.

- `Expr` sera une classe abstraite,
- on aura des sous-classes concrètes `TrueExpr`, `FalseExpr` (pour les constantes), `VarExpr` (pour les variables), `AndExpr` (pour la conjonction), `OrExpr` (pour la disjonction), `NotExpr` (pour la négation),
- pour simplifier, on considèrera que les variables sont identifiées par un `int`.

Voici un début d'implémentation :

```

abstract class Expr { }
class TrueExpr extends Expr { }
class FalseExpr extends Expr { }
class VarExpr extends Expr {
    int n; // l'identifiant de la variable
    VarExpr(int n) { this.n = n; }
}
class AndExpr extends Expr { ... } // a completer

```

**Question 5** Complétez ce qui précède en donnant les définitions des classes AndExpr et NotExpr (pas la peine d'écrire la définition de OrExpr qui est très similaire à AndExpr). ◇

*Solution.*

```

class AndExpr extends Expr {
    Expr left;
    Expr right;
    AndExpr(Expr a, Expr b) { left = a; right = b; }
}

class NotExpr extends Expr {
    Expr e;
    NotExpr(Expr e) { this.e = e; }
}

```

**Question 6** Equipez les sous-classes concrètes de Expr d'une méthode public String toString() qui affiche l'expression sous forme infixe. Les variables pourront être affichées simplement par leur numéro. Vous pouvez écrire V et F pour les constantes, et, respectivement, ET, OU et NON pour les connecteurs et vous n'êtes pas obligés d'optimiser le nombre de parenthèses.

Par exemple  $x \vee \neg y$  peut être affiché (1 OU (NON 2)) (si les variables x et y portent les numéros 1 et 2). ◇

*Solution.* Pas la peine de la mentionner dans Expr puisque c'est hérité de Object.

Par exemple, respectivement pour VarExpr, OrExpr AndExpr et NotExpr :

```

public String toString() { return("" + n); }
public String toString() {
    return( "(" + left.toString() + " OU " + right.toString() + ")" ); }
public String toString() {
    return( "(" + left.toString() + " ET " + right.toString() + ")" ); }
public String toString() {
    return( "(NON" + e.toString() + ")" ); }

```

## Valuations et tautologies

Une valuation est une fonction qui donne une valeur de vérité (de type bool) à chaque variable. On représente cela en Java comme un objet implémentant l'interface suivante :

```

interface Valuation { boolean value(int n); }

```

Etant donnée une valuation, toute expression booléenne a elle même une valeur de vérité. Par exemple considérons une valuation  $I$ , telle que  $I(x) = I(y) = \text{true}$  et  $I(z) = \text{false}$ . Alors, pour  $I$ , l'expression  $x \wedge (y \vee z)$  vaudra true et  $(x \wedge z) \vee (y \wedge z)$  vaudra false.

On ajoute à Expr une méthode abstraite pour calculer la valeur d'une expression :

```

abstract boolean eval(Valuation v);

```

**Question 7** Implémentez cette méthode dans les sous-classes concrètes de Expr. ◇

*Solution.* Pour TrueExpr :

```
boolean eval(Valuation v) { return true; }
```

Pour FalseExpr :

```
boolean eval(Valuation v) { return false; }
```

Pour VarExpr :

```
boolean eval(Valuation v) { return (v.eval(n)); }
```

Pour AndExpr :

```
boolean eval(Valuation v) { return (left.eval(v) && right.eval(v)); }
```

Pour OrExpr :

```
boolean eval(Valuation v) { return (left.eval(v) || right.eval(v)); }
```

Pour NotExpr :

```
boolean eval(Valuation v) { return (!e.eval(v)); }
```

Une *tautologie* est une expression dont la valeur vaut true pour n'importe quelle valuation. Par exemple  $x \vee \neg x$  ou  $y \vee \text{true}$  sont des tautologies. En revanche  $\text{true} \wedge y$  n'en est pas une.

**Question 8** Donnez deux autres exemples de tautologies. ◇

*Solution.* Par exemple  $\text{true}$ ,  $\text{true} \vee \text{false}$ ,  $\neg \text{false}$ ... □

**Question 9** On équipe la classe Expr d'une méthode pour ajouter à un HashSet de variables toutes les variables d'une expression (en fait les nombres correspondants aux variables).

```
void vars(HashSet<Integer> ev) { }
```

*Cette définition par défaut doit être réécrite pour les sous-classes de Expr susceptibles de contenir des variables.*

Par exemple si on invoque `e.vars(ev)` pour  $x \vee y$ , alors  $x$  et  $y$  doivent être ajoutées à `ev`.

Réécrivez cette méthode dans les classes concernées pour que ce soit le cas. ◇

*Solution.* Il faut réécrire la méthode pour les trois classes contenant potentiellement des variables.

Pour VarExpr :

```
void vars(HashSet<Integer> ev) { ev.add(n) }
```

Pour AndExpr et OrExpr :

```
void vars(HashSet<Integer> ev) {  
    left.vars(ev);  
    right.vars(ev);  
}
```

Pour NotExpr :

```
void vars(HashSet<Integer> ev) { e.vars(ev); }
```

**Question 10** Utilisez la méthode précédente pour ajouter à Expr une méthode qui rend sous forme de `LinkedList<Integer>` l'ensemble des variables d'une expression :

```
LinkedList<Integer> lvars() { ... }
```

On pourra utiliser les caractéristiques des classes `HashSet<E>` et `LinkedList<E>` qui sont rappelées à la fin de l'énoncé. ◇

*Solution.* On peut l'ajouter directement à Expr sans toucher aux sous-classes :

```
LinkedList<Integer> lvars() {
    LinkedList<Integer> r = new LinkedList<Integer>();
    HashSet<Integer> h = new HashSet<Integer>();
    vars(h);
    r.addAll(h);
    return(r);
}
```

**Question 11** Ecrivez une fonction qui étant donnée une valuation *v*, une variable *x* et un booléen *b* construit la valuation qui vaut *b* en *x* et est identique à *v* ailleurs.

```
static Valuation subst(Valuation v , int x, boolean b) { ... }
```

*Solution.*

```
static Valuation subst(Valuation v , int x, boolean b) {
    Valuation nv =
        m -> {if (m == x) return(b); return(v.value(m)); };
    return nv;
}
```

**Question 12** Sans chercher à être efficace, écrivez une fonction statique qui teste si une expression est une tautologie.

```
static boolean tauto(Expr e) { ... }
```

Que pouvez-vous dire de sa complexité? ◇

*Solution.* On se donne une fonction auxiliaire récursive pour tester toutes la valeurs sur chaque variable. La complexité est évidemment exponentielle par rapport au nombre de variables.

```
static boolean tauto(Expr e) {
    LinkedList<Integer> lv = e.lvars();
    Valuation v = (n -> false);
    return(aux(e, lv, v));
}

static boolean aux(Expr e, LinkedList<Integer> lv, Valuation v) {
    if (lv.isEmpty()) return (e.eval(v));
    int n = lv.remove();
    return(aux(e, lv, Gen.setTrue(n, v)) && aux(e, lv, Gen.setFalse(n, v)));
}
```

## BDDs

On se donne maintenant une autre manière de représenter les expressions booléennes : les diagrammes de décisions booléens ou BDDs (pour *binary decision diagrams*). Un BDD est soit :

- une constante booléenne (true ou false),
- soit un nœud  $\text{IfBDD}(x, l, r)$  où  $x$  est une variable propositionnelle et  $l$  et  $r$  sont eux-mêmes chacun un BDD.

La valeur d'un BDD de la forme  $\text{IfBDD}(x, l, r)$  est définie ainsi : si  $x$  vaut true, alors  $\text{IfBDD}(x, l, r)$  vaut la même valeur que  $l$ , si  $x$  vaut false, alors  $\text{IfBDD}(x, l, r)$  vaut la même valeur que  $r$ .

**Question 13** On se donne une classe abstraite BDD :

```
abstract class BDD {  
    abstract boolean value(Valuation v);  
}
```

Complétez en proposant trois sous-classes concrètes de BDD : TrueBDD, FalseBDD et IfBDD. ◇

*Solution.*

```
class FalseBDD extends BDD {  
    boolean value(Valuation v) { return false; }  
}  
class TrueBDD extends BDD {  
    boolean value(Valuation v) { return true; }  
}  
  
class IfBDD extends BDD {  
    int var;  
    BDD trueCase;  
    BDD falseCase;  
    IfBDD(int n, BDD t, BDD f) {  
        var = n;  
        trueCase = t;  
        falseCase = f;  
    }  
    boolean value(Valuation v) {  
        if (v.value(n)) return trueCase.value(v);  
        return falseCase.value(v);  
    }  
}
```

On peut traduire une Expr en BDD en utilisant le principe de l'expansion de Shannon : étant donné une expression  $e$  et une variable  $x$  apparaissant dans  $e$  on peut remarquer que pour toute valuation  $v$  on a :

- $e.\text{eval}(v)$  égal à  $e.\text{eval}(\text{subst}(v, x, \text{true}))$  lorsque  $v.\text{value}(x)$  vaut true.
- $e.\text{eval}(v)$  égal à  $e.\text{eval}(\text{subst}(v, x, \text{false}))$  lorsque  $v.\text{value}(x)$  vaut false.

**Question 14** En partant de la remarque précédente, écrivez une fonction translate pour traduire une expression en BDD (ce peut être une fonction statique ou une méthode, à votre guise).

Indication : on pourra également utiliser la méthode lvars(). ◇

*Solution.* On teste sur chaque variable pour faire un grand peigne (de  $2^n$  feuilles s'il y a  $n$  variables).

```
static BDD translate(Expr e) {  
    LinkedList<Integer> lv = e.lvars(lv);  
    Valuation v = (n -> false); // en fait on pourrait prendre
```

```

    // n'importe quelle valuation
    return(traux(e, lv, v));
}

static BDD traux(Expr e, LinkedList<Integer> lv, Valuation v) {
    if (lv.isEmpty())
        if (e.eval(v)) return new TrueBDD();
        else return new FalseBDD();
    int n = lv.remove();
    return(new IfBDD(n, traux(e, lv, subst(v, n, true)),
                    traux(e, lv, subst(v, n, false))));
}

```

On va maintenant essayer d'optimiser la représentation sous forme de BDD.

**Question 15** Equipez les sous-classes de BDD d'une méthode de hachage `int hashCode()` correcte. On pourra prendre la fonction définie par :

$$\begin{aligned}
 h(\text{true}) &= 3 & h(\text{false}) &= 1 \\
 h(\text{IfBDD}(x, l, r)) &= x + 31(h(l) + 31h(r)) & & \diamond
 \end{aligned}$$

*Solution.* Pour `FalseBDD`, `TrueBDD` et `IfBDD` respectivement :

```

public int hashCode() { return 1; }
public int hashCode() { return 3; }
public int hashCode() {
    return(var+31*(trueCase.hashCode() + 31 * falseCase.hashCode()));
}

```

**Question 16** Implémentez la méthode `public boolean equals(Object o)` des sous-classes de BDD pour qu'elle teste correctement l'égalité structurelle entre BDD. ◇

*Solution.* Respectivement :

```

public boolean equals(Object o) {
    return(o instanceof FalseBDD);
}

public boolean equals(Object o) {
    return(o instanceof TrueBDD);
}

public boolean equals(Object o) {
    return((o instanceof IfBDD) &&
           ((IfBDD)o).trueCase.equals(trueCase) &&
           ((IfBDD)o).falseCase.equals(falseCase));
}

```

On veut éviter d'avoir en mémoire plusieurs copies d'un même BDD. Pour cela on va maintenir un cache.

On se donne une table d'association entre BDD :

```
static HashMap<BDD, BDD> cache = new HashMap<BDD, BDD>();
```

L'idée est la suivante :

— cache ne contiendra que des associations entre un BDD et lui-même.

- Lorsqu'on construira un nouveau BDD, on regardera dans `cache` si on a déjà une copie structurellement égale à ce BDD. Dans ce cas, on utilisera la version du cache. Sinon on ajoutera ce nouveau BDD au cache.

**Question 17** Ecrivez une fonction `static BDD retrieve(BDD e)` qui se comporte ainsi :

- Si un BDD structurellement égal à `e` est élément du cache, alors on renvoie cet élément du cache.
- Sinon on renvoie `e` après avoir ajouté `e` au cache. ◇

*Solution.*

```
static BDD retrieve (BDD b) {
    BDD r = cache.get(b);
    if (r != null) return r;
    cache.put(b, b);
    return b;
}
```

On peut remarquer que chaque élément du cache n'apparaît qu'une fois (la clé et la valeur sont physiquement égales). □

Un BDD est dit normalisé s'il vérifie les deux propriétés suivantes :

1. Si deux sous-parties du BDD sont identiques, elles sont partagées en mémoire.
2. Il n'y a pas de `IfBDD(x, l, r)` dont les deux branches `l` et `r` sont identiques.

On peut comprendre la deuxième clause comme : une (sous-)expression `IfBDD(x, e, e)` peut être remplacée par l'expression `e`.

**Question 18** Ecrivez une fonction qui normalise, ou essaye de normaliser au maximum, un BDD :

```
static BDD normalize(BDD e) { .. }
```

On pourra utiliser la variable globale `cache` et la fonction de la question précédente. ◇

*Solution.* Une possibilité est de commencer à ajouter une méthode `void norm()` à BDD.

```
// Pour TrueBDD et FalseBDD :
BDD norm() { return(Test.cache(this)); }

// pour IfBDD :
BDD norm() {
    BDD nl = Test.cache(trueCase.norm());
    BDD nr = Test.cache(falseCase.norm());
    if (nr.equals(nl)) return nr;
    return(Test.cache(new IfBDD(var, nl, nr)));
}

// et la fonction de normalisation :
static BDD normalize(BDD b) {
    BDD d = b.norm();
    if (b.equals(d)) return b;
    return(normalize(d));
}
```

Ici on itère `norm` jusqu'à ça ne bouge plus. On peut réfléchir à si c'est vraiment nécessaire ; je suis preneur d'arguments précis si vous en avez. On sera libéral dans la correction sur ce point de toute façon. □



**Question 19** Un collègue regarde le code et fait la remarque qu’il est utile, dans ce cas, de commencer le test d’égalité structurelle sur `IfBDD` (cad. la méthode `equals`) par un test d’égalité physique.

Pouvez-vous deviner l’argument du collègue ? (brièvement)

Ecrivez-la méthode `equals(Object o)` ainsi modifiée. ◇

*Solution.* A cause de l’utilisation du cache, il y a beaucoup de chances que deux BDDs structurellement égaux soient aussi physiquement égaux (car pointant vers le même objet du cache). Cela vaut donc la peine de commencer par vérifier s’ils sont physiquement égaux, auquel cas ils seront aussi structurellement égaux.

Sinon on fait le test classique. On peut se poser la question de si le test d’égalité physique suffit, mais c’est au mieux compliqué.

```
public boolean equals(Object o) {  
    if (o == this) return true;  
    // puis le code déjà donne
```

**Question 20** On prend une `Expr` e qui est une tautologie. On traduit cette expression en BDD avec la fonction `translate` de la question 14, puis on normalise le résultat avec `normalize` (question 18). Que peut-on dire du résultat ? Justifiez brièvement. ◇

*Solution.* Ce sera forcément un élément de `TrueBDD` (en fait l’unique copie de `TrueBDD` dans `cache`).

En effet, dans la traduction vers les BDDs, chaque variable apparaissant dans l’expression n’est considérée qu’une fois. Il ne peut donc y avoir de chemin de la racine du BDD vers une feuille `FalseBDD`. Les parties du BDD de la forme `IfBDD(x, TrueBbb, TrueBDD)` seront simplifiées vers `TrueBDD` et on recommencera jusqu’à arriver au BDD `trueBDD`. □

## Rappels sur quelques classes de la bibliothèque

### La classe `HashSet<E>`

Elle permet de gérer facilement un ensemble d’objets de classe `E`. Le constructeur `HashSet()` créé un nouvel ensemble vide. Elle dispose en particulier des méthodes suivantes :

<code>add(E e)</code>	qui ajoute l’élément <code>e</code> à l’ensemble (s’il n’y apparaît pas déjà).
<code>boolean contains(E e)</code>	qui indique si l’objet <code>e</code> appartient à l’ensemble.

Comme `LinkedList`, `HashSet` est une implémentation de `Collection`, et donc aussi de `Iterable` (voir ci-dessous).

### La classe `LinkedList<E>`

C’est une implémentation de listes chaînées. Elle est en particulier munie des méthodes suivantes :

<code>add(E e)</code>	qui ajoute l’élément <code>e</code> à la liste
<code>E remove()</code>	qui enlève et renvoie le premier élément de la liste
<code>addAll(Collection&lt;E&gt; l)</code>	qui ajoute à la liste tous les éléments de <code>l</code>

On rappelle que `LinkedList` et `HashSet` sont des implémentations de `Collection`, et donc aussi de `Iterable`, ce qui a pour conséquences que :

- L'argument de `addAll` peut être une `LinkedList` ou un `HashSet`.
- On peut itérer sur les éléments d'une `LinkedList` ou d'un `HashSet` avec la notation `for (E e : l) ....`

### **La classe** `HashMap<K,V>`

Elle est munie des méthodes suivantes :

- `put(K k, V v)` qui associe la valeur `v` à la clé `k`.
- `V get(K k)` qui renvoie la valeur associée à la clé `k` et renvoie `null` si aucune valeur n'est associée à `k`.

Le constructeur `HashMap()` crée une nouvelle table d'association vide.