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 :

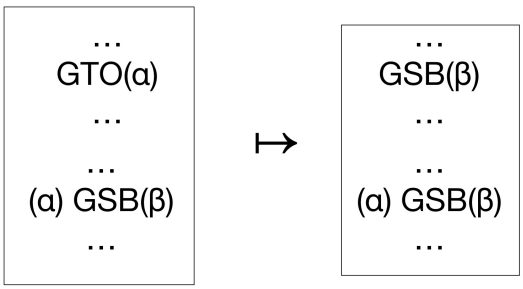
(1) PUSH(9)
(2) PUSH(1)
(3) ADD
(4) PUSH(0)
(5) ☐
(6) GTO(2)
(7) STOP

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.

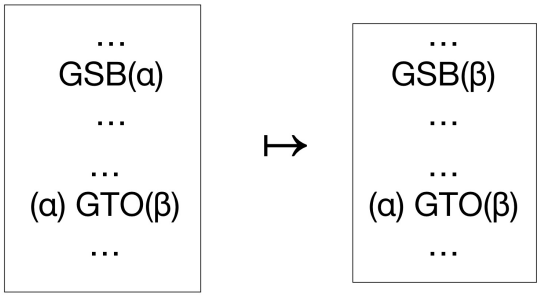
c) Proposez une instruction à mettre dans l'emplacement vide pour que le programme termine. ◇

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 ? ◇

Question 3 Même question pour cette optimisation (remplacer un GSB(α) par un GSB(β) quand on est dans la situation de gauche).



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 = new A(2, 3, "a");` ?

Et après `A v = new A(2, 3, "aaaa");` ? ◇

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`...
— soit construites avec les connecteurs $\wedge$ (et), $\vee$ (ou) et $\neg$ (non).
Des exemples d'expressions sont donc : $(\neg \texttt{x} \vee \texttt{y})$, $(\texttt{x} \wedge (\neg \texttt{y} \vee \texttt{z}))$, $\neg(x \vee \texttt{false})$ *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`). ◇

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). ◇

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 $I(x) = I(y) =$ `true` et $I(z) =$ `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`. ◇

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

Question 8 Donnez deux autres exemples de tautologies. ◇

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. ◇

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. ◇

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) { ... }
```

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 ? ◇

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 `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 `IfBDD`(x, l, r) est définie ainsi : si x vaut `true`, alors `IfBDD`(x, l, r) vaut la même valeur que l, si x vaut `false`, alors `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`. ◇

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

— `e.eval(v)` égal à `e.eval(subst(v, x, true))` lorsque `v.value(x)` vaut **`true`**.

— `e.eval(v)` égal à `e.eval(subst(v, x, false))` lorsque `v.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()`. ◇

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 :

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

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`. ◇

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. ◇

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. ◇

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. ◇

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. ◇

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 :

`add(E e)`	qui ajoute l'élément `e` à l'ensemble (s'il n'y apparait pas déjà).
`boolean contains(E e)`	qui indique si l'objet `e` 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 :

`add(E e)`	qui ajoute l'élément `e` à la liste
`E remove()`	qui enlève et renvoie le premier élément de la liste
`addAll(Collection<E> l)`	qui ajoute à la liste tous les éléments de `l`

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.