SCIENCES SORBONNE UNIVERSITÉ

Examen de fin de semestre POBJ
Licence 3 Informatique

C. Constantin, B. Lefebvre, J. Lejeune, A. Miné, D. Mercadier, Y. Thierry-Mieg

Mai 2018

2 heures – **Tout document papier autorisé**
Tout appareil électronique interdit

- Le barème est sur 22½ points et est donné à titre indicatif. La note finale sera bornée à 20 points.
- Dans le code java demandé vous ne gérerez pas les imports.
- Lors de la correction il sera tenu compte de la simplicité et de lisibilité de vos réponses.
- Il vous est conseillé de lire attentivement les exercices entièrement avant de composer.

On considère la construction d'expressions booléennes, comme

$$e = (x_0 \wedge x_1) \vee (x_2 \wedge \neg x_3)$$

Les variables $x_0 \ldots x_N$ sont des booléens, $\vee$ désigne la disjonction (*OR* logique), $\wedge$ désigne la conjonction (*AND* logique), et $\neg$ désigne la négation (*NOT* logique).

Le sujet est composé de quatre exercices relativement indépendants où l'on explore des représentations de ces expressions. Cependant, l'interface introduite dans l'exercice 2 est fortement réutilisée dans les exercices 3 et 4.

Exercice 1 – Arbre de syntaxe (4,5 points)

Notions testées : Classe, Interface, Instance, Composite.

On considère une implantation des expressions suivant le DP *Composite*. On aura

- Une interface `IBoolExpr` pour unifier le typage des expressions.
- Une classe `Constant` munie d'un booléen matérialisant les constantes "true" et "false".
- Une classe `Var` munie d'un entier i permettant de représenter une variable x_i.
- Une classe `Not` munie d'un opérande (son fils) permettant de représenter une négation $\neg e$.
- Une classe abstraite `BinOp` munie de deux références à ses fils représentant un opérateur binaire.
- Deux classes concrètes `And` et `Or` représentant les opérateurs binaires que sont le *OR* et le *AND* logique.

A titre d'exemple, l'expression donnée en introduction sera représentée par l'arbre de syntaxe de la Figure 1.

Question 1 1½ points

Sur un diagramme de classes UML, représenter cette situation. On fera figurer les classes et interfaces, leurs attributs, les associations qui les lient ainsi que les liens d'héritage et d'implémentation d'interface. Les méthodes et constructeurs ne sont pas demandés.

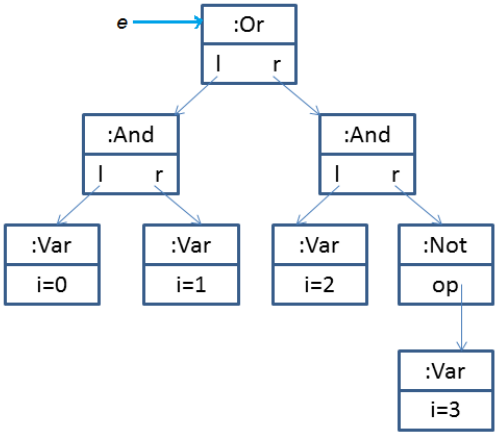
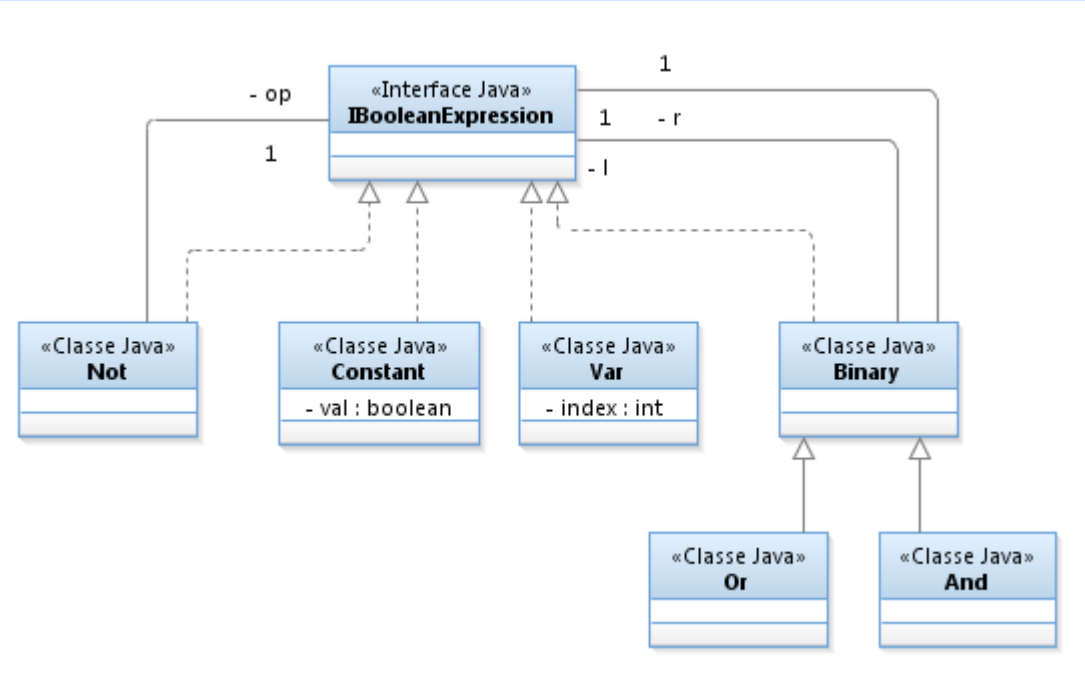


FIGURE 1 – Arbre de syntaxe de l'expression $e = (x_0 \wedge x_1) \vee (x_2 \wedge \neg x3)$.

Solution:
Donc le sujet n'en parle pas, mais il nous faut une Expression pour chapeauter.



Barème :

- 25 % présence des 5 classes concrètes Const, Var, Not, Or, And.
- 25 % elles ont toutes un typage commun en l'interface Expression.
- 25 % BinOp est bien un Expression, munie des attributs left/right.
- 25 % attributs et associations présents de façon non redondante et suffisante pour capturer le problème.

On souhaite afficher les expressions, en utilisant la méthode standard **toString**. On impose que le code de cette méthode dans `BinOp` construise une expression parenthésée, dont l'opérateur (une String) est laissée aux soins d'une méthode abstraite `String getOp()`. On notera le OR avec "||", le AND avec "&&" et le NOT avec " !". Pour les variables, on concatène l'indice à un x. Globalement on souhaite obtenir pour l'expression e de l'introduction la String
"((x0&&x1)||(x2&& !(x3)))".

Question 2 2 points

En cohérence avec votre diagramme UML de la question précédente :

- donnez le code complet de la classe abstraite `BinOp` et de la classe `And` et de la classe `Var`. On impose que tous les attributs soient déclarés **private** et **final**. Pour ces trois classes, donnez le code de la déclaration, du constructeur et de la méthode `toString`.

Solution:

Binop.java

```java
package examMai2018.exo1;


public abstract class Binary implements IBooleanExpression {

	private final IBooleanExpression l;
	private final IBooleanExpression r;
	public Binary(IBooleanExpression l, IBooleanExpression r) {
		this.l = l;
		this.r = r;
	}

	public abstract String getOp ();

	@Override
	public String toString() {
		return "(" + l + getOp() + r + ")";
	}
```

And.java

```java

public class And extends Binary {

	public And(IBooleanExpression l, IBooleanExpression r) {
		super(l,r);
	}

	@Override
	public String getOp() {
		return "&&";
	}
```

Var.java

```java

public class Var implements IBooleanExpression {

	private int index;

	public Var(int index) {
		this.index = index;
	}

	@Override
```

```
        public String toString() {
                return "x"+index;
        }
```

Barème :

- 40 % classe Binop : 10% implements, abstract. 10 % les deux attributs typés expression,final 10% ctor , 10% toString
- 30 % classe And, 10% extends BinOp, 10% ctor, 10 % getOp
- 30 % classe Var, 10 % déclaration + implements, 10 % attribut final private, 10 % ctor + toString

Question 3 1 point

Quel est l'effet de déclarer un attribut `final` ? Quel intérêt cela peut-il présenter de déclarer tous les attributs `final` ?

Solution:

Attribut final = non modifiable après la construction, une seule affectation au total.

Tous les attributs final = garantir l'immuabilité, une propriété particulièrement utile pour faire de l'aliasing mémoire. Le passage par référence ne pose plus de problème d'encapsulation.

Barème :

- 50 % déf de final sur un attribut
- 50 % immuabilité abordée dans la réponse

Exercice 2 – Construction et Copie (4 points)

Notions testées : Factory, Génériques.

On définit une interface générique pour construire des expressions booléennes. On y définit une méthode pour chaque noeud possible dans la formule. (NB : "true" et "false" étant des mots clés, on les préfixe avec un "m" ici, i.e. "mtrue" et "mfalse").

L'interface est générique et paramétrée par `BE`, censé représenter une expression booléenne.

IBoolExprFactory.java

```
public interface IBoolExprFactory<BE> {
       BE var (int index); // variable x_index
       BE not (BE e); // negation NOT d'une expression
       BE and (BE l, BE r); // AND logique
       BE or (BE l, BE r); // OR logique
       BE mtrue (); // expression VRAI
       BE mfalse (); // expresion FAUX
}
```

Question 1 1½ points

Donnez le code déclarant une factory d'expressions booléennes concrète permettant de construire les expressions définies en **exercice 1)**. Attention à correctement spécialiser le paramètre générique. On ne demande le code que de trois des méthodes : `and, var, mtrue`.

Solution:

BEF.java

```
public class BasicBoolFactory implements IBoolExprFactory<IBooleanExpression>
    {

        @Override
        public IBooleanExpression mtrue() {
                return new Constant(true);
        }
        @Override
        public IBooleanExpression var(int index) {
                return new Var(index);
        }
        @Override
        public IBooleanExpression and(IBooleanExpression l, IBooleanExpression r)
            {
                return new And(l,r);
        }
```

Barème :

- 25 % déclaration spécialisée sur implements
- 25 % * 3 pour les méthodes demandées

Supposons qu'on définisse une nouvelle méthode dans l'interface `IBoolExpr` de la manière suivante :

IBE.java

```
public interface IBoolExpr {
        <BE> BE copy (IBoolExprFactory<BE> facto);
}
```

Question 2 ½ point

Comment modifier la déclaration de `IBoolExprFactory` pour imposer que le type générique `BE` respecte l'interface `IBoolExpr` ?

Solution:

IBEF.java

```
package examMai2018;

public interface IBooleanExpressionFactory2<BE extends IBoolExpr> {
        // ...
}
```

Barème : binaire 0 ou 30 %. Note comptée avec la question suivante.

NB : Dans les autres exercices, on ne considérera pas que cette modification a été réalisée.

Question 3 1 point

Donnez le code de la méthode `copy` qu'il faut réaliser dans la classe `Var` et dans la classe `Not`.

Solution:

NB : on évite la classe And à cause des attributs privés l et r, c'est pénible.

Var.java

```
	@Override
	public <BE> BE copy(IBoolExprFactory<BE> facto) {
		return facto.var(index);
	}

}
```

BEF.java

```
public class Not implements IBooleanExpression {

	private IBooleanExpression op;

	public Not(IBooleanExpression op) {
		this.op = op;
	}

	@Override
	public String toString() {
		return "!("+op+")";
	}

	@Override
	public <BE> BE copy(IBoolExprFactory<BE> facto) {
		return facto.not(op.copy(facto));
	}
}
```

Barème :

- 30 % venant de la question précédente
- 30 % copy de Var
- 40 % le not avec sa récursion profonde. On ne demande que les lignes 16 et 17.

Question 4 1 point

Écrire un test JUnit4, qui construise à l'aide de la factory l'expression *e* donnée en introduction, et s'assure que la méthode `toString` produise bien le résultat attendu : "((x0&&x1)||(x2&& !(x3)))".

Solution:

TestString.java

```
package examMai2018.exo1;

import static org.junit.Assert.*;

import org.junit.Test;

import examMai2018.IBoolExprFactory;

public class TestToString {
```

```
	@Test
	public void testString() {
		IBoolExprFactory<IBooleanExpression> fac = new BasicBoolFactory();

		IBooleanExpression e = fac.or(fac.and(fac.var(0), fac.var(1)), fac.
		    and(fac.var(2), fac.not(fac.var(3))));

		assertEquals("((x0&&x1)||(x2&&!(x3)))",e.toString());
	}

}
```

Barème :

- 20 % déclaration du test
- 20 % initialisation de la facto
- 30 % construction de l'expression e
- 30 % assertion sur le résultat

Exercice 3 – Unicité et Cache (9 points)

Notions testées : Map, Decorator.

Question 1 1 point

Donnez le code d'un décorateur abstrait (au sens du *design pattern*) pour une `IBoolExprFactory<BE>`. On considère la version définie par cet énoncé, sans les éventuelles modifications que vous avez apportées dans l'exercice 2. Vous fournirez la déclaration, les attributs, le constructeur, et les méthodes `and` et `var`. Les attributs seront déclarés `private`. Le décorateur sera lui-même générique (pas de spécialisation de `BE`).

Solution:

AbstractDecorator.java

```
public abstract class AbstractBEFDecorator<BE> implements IBoolExprFactory<BE
    >{

	private IBoolExprFactory<BE> deco;

	public AbstractBEFDecorator(IBoolExprFactory<BE> deco) {
		this.deco = deco;
	}

	public BE var(int index) {
		return deco.var(index);
	}
	public BE and(BE l, BE r) {
		return deco.and(l, r);
	}
	public BE not(BE op) {
```

Barème :

- 20 % déclaration spécialisée sur implements
- 40 % attribut et ctor
- 40 % les deux méthodes demandées

On **impose** que les décorateurs concrets développés par la suite étendent ce décorateur abstrait.

Décorateur pour l'unicité.

Une table d'unicité a pour rôle de faciliter le partage en mémoire de sous arbres. Elle se réalise à l'aide d'une table associative (un *HashMap*) dont les valeurs sont égales aux clés. La table d'unicité UniqueTable<T> est munie d'une opération `T makeUnique(T obj)` qui cherche si l'objet est déjà présent dans la table ou non.

S'il est présent, i.e. il existe un élément *e* dans la table qui soit égal (au sens de *equals*) à *obj*, la méthode rend *e*. Sinon, l'objet *obj* est inséré dans la table et la méthode rend la référence de *obj*.

Question 2 1½ points

Donnez le code complet de la classe `UniqueTable<T>`.

Solution:

NB : c'est vu en cours, dans une version WeakReference un peu plus riche.

UniqueTable.java

```
package examMai2018.exo2;

import java.util.HashMap;
import java.util.Map;

public class UniqueTable<T> {

	private Map<T, T> map = new HashMap<>();

	T makeUnique (T obj) {
		T e = map.get(obj);
		if (e != null) {
			return e;
		} else {
			map.put(obj, obj);
			return obj;
		}
	}

}
```

Barème :

- 10 % déclaration de la classe générique
- 30 % attribut correctement initialisé
- 30 % recherche de l'élément
- 30 % insertion s'il n'existe pas

Question 3 1½ points

Une factory d'expressions peut être décorée pour utiliser une table d'unicité.

Écrivez un décorateur concret pour une factory d'expression, muni d'une table d'unicité et qui garantisse la construction d'une seule instance de chaque expression. C'est à dire que deux invocations à

une opération de la factory avec les mêmes paramètres doivent rendre le même objet. Fournissez la déclaration, les attributs, le constructeur, et les méthodes `and` et `var`.

Solution:

UniqueFactory.java

```
package examMai2018.exo2;

import examMai2018.IBoolExprFactory;

public class UniqueFactory<BE> extends AbstractBEFDecorator<BE> {

	private UniqueTable<BE> unique = new UniqueTable<>();
	public UniqueFactory(IBoolExprFactory<BE> deco) {
		super(deco);
	}
	public BE var(int index) {
		return unique.makeUnique(super.var(index));
	}
	public BE and(BE l, BE r) {
		return unique.makeUnique(super.and(l, r));
	}
	// ...

}
```

Barème :

- 20 % déclaration de la classe + extends sur deco abstrait,
- 20 % attribut correctement initialisé
- 20 % constructor (avec paramètre, invoquant super)
- 40 % les deux méthodes (-20 % si super n'est pas correctement utilisé)

Question 4 1 point

Quelles sont les contraintes sur les classes réalisant les expressions `BE` pour que le code de la table d'unicité fonctionne comme prévu ?

Solution:

Il faut que toutes les classes concrètes réalisent : `public boolean equals(Object o)` ; `public int hashCode()`.

Il est de plus souhaitable que toutes les expressions soient immuables (e.g. attributs final).

Barème :

- 40 % hashcode
- 40 % equals
- 20 % immuabilité cité dans la réponse

Décorateur mettant en place un cache.

Une factory d'expressions peut être décorée pour utiliser un cache. Pendant cet examen, on va traiter seulement le cas du `and`. Le traitement de l'autre opérateur binaire `or` serait similaire.

Un cache permet de retrouver la valeur résultant d'un calcul si on l'a déjà calculée. Ici les opérandes du calcul sont les deux expressions l et r passées à `BE and (BE l, BE r)`. Le cache doit donc associer à une paire de `BE` (les opérandes) une occurrence de `BE` (le résultat).

On souhaite définir une classe `Pair<T>` qui porte en attribut deux occurrences de `T` (les opérandes). Elle va servir de clé dans une table de hash, en conséquence définissez le `hashCode` comme le *XOR* des `hashCode` des deux opérandes. Le XOR (noté ^ en Java) de deux entiers est une opération commutative, mais le *AND* est lui même commutatif ($l \wedge r = r \wedge l$). L'égalité avec `equals` doit donc être définie pour que deux paires (l_1, r_1) et (l_2, r_2) soient considérées égales si $l_1 = l_2$ et $r_1 = r_2$ ou symétriquement si $l_1 = r_2$ et $r_1 = l_2$. Autrement dit, $(l, r) = (l, r)$ mais aussi $(l, r) = (r, l)$. On va considérer à cette question que les `T` manipulés sont déjà uniques en mémoire ; on pourra donc les comparer directement à l'aide de l'égalité référentielle.

Question 5 $1\frac{1}{2}$ points

Donnez le code complet de cette classe `Pair<T>`.

Solution:

Pair.java

```
package examMai2018.exo2;

public class Pair<T> {
	private T l;
	private T r;
	public Pair(T l, T r) {
		this.l = l;
		this.r = r;
	}
	@Override
	public int hashCode() {
		return l.hashCode() ^ r.hashCode();
	}
	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Pair other = (Pair) obj;
		return (l==other.l && r==other.r) || (l==other.r && r==other.l);
	}

}
```

Barème :

- 10 % déclaration Pair<T>
- 20 % déclaration des deux attributs + constructeur
- 20 % hashCode
- 50 % equals, dont 20 % sur le test correct avec instanceof ou getClass + typecast, et 20 % le test est symétrique, 10 % le test ne récurse as plus d'un niveau (equals ok, ssi. on démarre par un test obj==this).

Question 6 1½ points

Écrivez un décorateur concret pour une factory d'expression, muni d'un cache pour l'opération `and` c'est à dire une `Map<Pair<BE>,BE>` (on utilisera un `HashMap`), et qui redéfinit le `and` de manière à d'abord chercher dans le cache le résultat. Si on le trouve, on rend directement le résultat. Sinon on calcule le résultat et on ajoute ce résultat au cache avant de le retourner.

Solution:

CacheFactory.java

```
public class CacheBEF<BE> extends AbstractBEFDecorator<BE> {

	public CacheBEF(IBoolExprFactory<BE> deco) {
		super(deco);
	}

	private Map<Pair<BE>,BE> cacheAnd = new HashMap<Pair<BE>, BE>();

	@Override
	public BE and(BE l, BE r) {
		Pair<BE> op = new Pair<BE>(l, r);
		BE res = cacheAnd.get(op);
		if (res == null) {
			res = super.and(l, r);
			cacheAnd.put(op,res);
		}
		return res;
	}
}
```

Barème :

- 20 % déclaration de la classe + extends sur deco abstrait,
- 20 % attribut correctement initialisé
- 20 % constructor (avec paramètre, invoquant super)
- 40 % le and utilise correctement le cache

Assemblage des décorations.

Question 7 1 point

Écrivez un test JUnit 4, qui construise deux occurrences de la formule $x_0 \wedge x_1$ à l'aide d'une factory d'expressions uniques et munie d'un cache (attention à l'ordre des décorations !), et échoue si l'égalité logique (à l'aide de `equals`) entre les deux occurrences rend $false$ ou si l'égalité référentielle rend $false$.

Solution:

TestEqual.java

```
package examMai2018.exo2;

import static org.junit.Assert.*;

import org.junit.Test;

```

```java
import examMai2018.IBoolExprFactory;
import examMai2018.exo1.BasicBoolFactory;
import examMai2018.exo1.IBooleanExpression;

public class TestEqual {

	@Test
	public void testEquality() {
		IBoolExprFactory<IBooleanExpression> fac = new BasicBoolFactory();
		// deco 1 : Unicité
		fac = new UniqueFactory<>(fac);
		// deco 2 : cache
		fac = new CacheBEF<>(fac);
		// deux expressions
		IBooleanExpression be = fac.and(fac.var(0), fac.var(1));
		IBooleanExpression be2 = fac.and(fac.var(0), fac.var(1));
		// equals
		assertTrue(be.equals(be2));
		// égalité référentielle
		assertTrue(be==be2);
	}

}
```

Barème :

- 20 % déclaration du test
- 50 % construction de la factory, décorée deux fois
- 20 % construction de deux instances distinctes de e
- 30 % résultats attendus correctement testés.

Exercice 4 – Diagramme de Décision Binaire (5 points)

Notions testées : Raisonnement, Récursion, Algorithmique, Structures de Données.

Les expression construites jusqu'ici n'ont pas une forme canonique ; par exemple on peut construire l'expression $\neg\neg x_0$ qui ne sera pas considérée égale à x_0. Les diagrammes de décision binaires réduits et ordonnés ou ROBDD (Reduced Ordered Binary Decision Diagram) fournissent une forme canonique ainsi qu'une représentation compacte des expressions booléennes.

La représentation s'appuie sur un arbre, dit de décision, où les variables de la formule sont rencontrées par ordre strictement décroissant. Les nœuds de l'arbre sont des décisions à la manière d'un if-then-else : si la variable est vraie, aller voir le fils "true" sinon aller voir le fils "false".

La version de base est peu compacte, elle correspond plus ou moins à la table de vérité de la fonction, représentée par un graphe. Mais elle est canonique, il n'y a qu'une représentation possible d'une fonction booléenne. Un diagramme de décision pour l'expression e est donné Figure 3.

Chaque nœud de l'arbre porte un indice i de variable et possède deux fils $siVrai$ et $siFaux$, correspondant à la sous formule où la variable x_i est vraie (ou respectivement fausse).

Il existe deux nœuds terminaux, correspondant à la formule "vrai" et à la formule "faux". Ces nœuds sont représentés par des nœuds dont les deux fils sont null, et ayant la variable d'indice -1 pour "vrai" et -2 pour "faux".

On suppose dans la manipulation des ROBDD que tous les nœuds sont uniques, i.e. ils sont construits à l'aide d'une factory comme celle développée en exercice 3. Cela permet une réduction importante de la

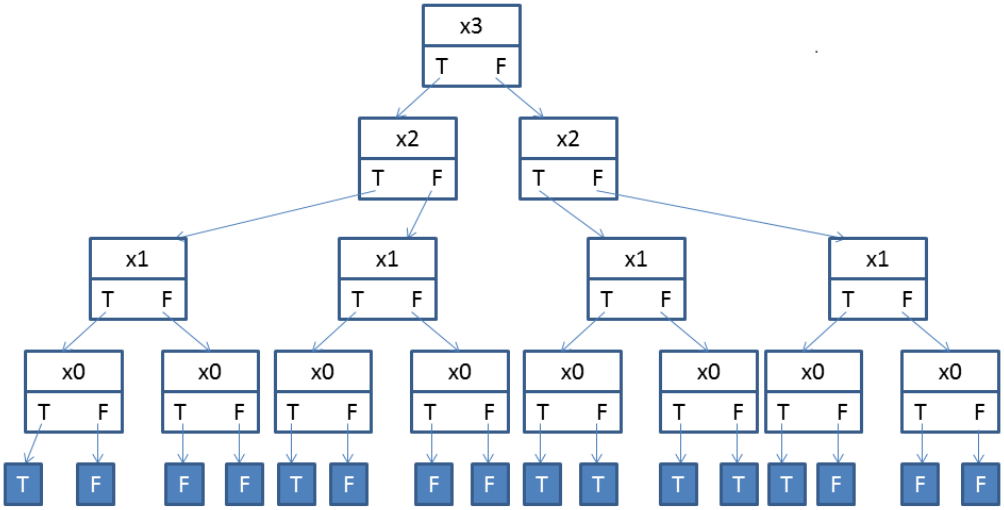
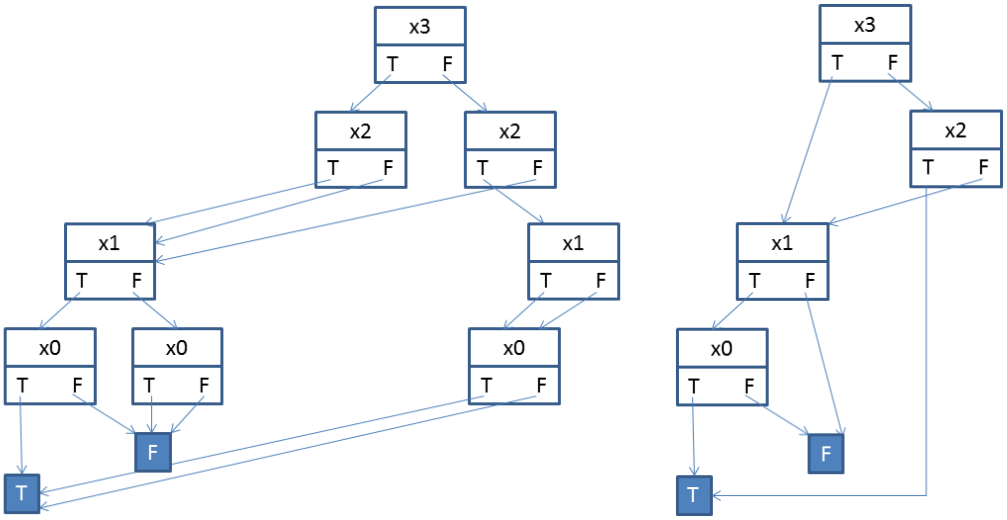


FIGURE 2 – Diagramme de décision ordonné de l'expression $e = (x_0 \wedge x_1) \vee (x_2 \wedge \neg x3)$.

taille des diagrammes, cf. Figure 3, gauche. Ce diagramme est le même qu'en figure 2, mais les sous arbre égaux ont été fusionnés en mémoire.



(a) Diagramme de décision ordonné et **sous arbres partagés**

(b) ROBDD de l'expression $e = (x_0 \wedge x_1) \vee (x_2 \wedge \neg x3)$.

Question 1 $1\frac{1}{2}$ points

Écrivez la classe `ITE` permettant de représenter un nœud de l'arbre de décision. On souhaite qu'elle

- soit immuable.
- porte un indice de variable (un entier), et les références de ses deux fils, eux même des `ITE`.
- comporte la méthode standard *equals* ; elle ne doit pas faire de récursion sur l'arbre entier, étant donné que les nœuds sont uniques par construction, on se contentera d'égalités référentielles pour comparer les fils d'un nœud.
- Pour faciliter la suite du code, on définira des accesseurs en lecture (*getter*) pour les attributs.

Solution:

ITE.java

```java
package examMai2018.exo2;

public class ITE {

	private final ITE ifT ;
	private final ITE ifF ;
	private final int var;

	public ITE(int var, ITE ifT, ITE ifF) {
		this.var = var;
		this.ifT = ifT;
		this.ifF = ifF;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (!(obj instanceof ITE))
			return false;
		ITE other = (ITE) obj;
		if (var != other.var)
			return false;
		return ifT==other.ifT && ifF == other.ifF;
	}
```

Barème :

- 30 % déclaration, attributs + constructeur
- 20 % getter pour les trois attributs.
- 50 % equals (dont 25 % sur le fait qu'on ne récurse pas tout l'arbre).

L'objectif est maintenant de construire une `ITEFactory` pour produire des ROBDD. Le reste de l'exercice décompose en étapes la construction de cette classe. On répondra aux trois questions suivantes en fournissant la classe `ITEFactory` permettant de construire des ROBDD. C'est une implantation de `IBoolExprFactory<ITE` tel que définie par l'énoncé (donc sans les modifications apportées en exercice 2).

Question 2 1 point

Réduction des fils égaux

Les ROBDD sont réduits par rapports aux diagrammes de décision par la règle suivante : un noeud dont les deux fils sont égaux peut être "sauté", on lie son père directement à ses fils. Cette réduction simplifie grandement les diagrammes de décision (cf. Figure 3, droite). Par rapport à la version à gauche de la même expression, les noeuds ayant même fils *siVrai* et *siFaux* ont été simplifiés.

Pour obtenir cet effet, définir dans `ITEFactory` une méthode privée `ITE newITE(int indexVar, ITE ifTrue, ITE ifFalse)`. Cette méthode opère la réduction suivante : si $siVrai = siFaux$ on rend *siVrai*. Sinon on construit un nouveau noeud avec les paramètres fournis.

Dans la suite du code, on construira les nouveaux noeuds avec cette méthode (un genre de "constructeur intelligent") plutôt qu'avec des `new ITE(i,t,f)`.

Solution:

BDDFac.java

```
public class BDDFactory implements IBoolExprFactory<ITE>{

	private ITE newITE(int var, ITE ifT, ITE ifF) {
		if (ifT == ifF) {
			return ifF;
		} else {
			return new ITE(var,ifT,ifF);
		}
	}
```

Barème :

- 20 % déclaration de la factory, implements IBEF avec spécialisation du générique
- 30 % déclaration private de la méthode newITE
- 50 % réduction demandée correctement réalisée

Question 3 1 point

Terminaux true/false, Variables if-then-else

Définir à présent dans la classe `ITEFactory` :

- `ITE mtrue(), ITE mfalse()` rendent des noeuds de fils "null" et de variable respectivement -1 et -2. Ces noeuds seront déclarés comme des constantes, i.e. des attributs déclarés `private static final`.
- `ITE var(int index)` rend un noeud de variable *index* et de fils *siVrai* à "vrai" et *siFaux* à "faux".

Solution:

BDDFac.java

```

	public static final ITE True = new ITE(-1,null,null);
	public static final ITE False = new ITE(-2,null,null);

	@Override
	public ITE mtrue() {
		return True;
	}
	@Override
	public ITE mfalse() {
		return False;
	}

	@Override
	public ITE var(int var) {
		return newITE(var,True,False);
	}
```

Barème :

- 25 % déclaration correcte des constantes True/False
- 25 % méthodes mtrue/mfalse
- 50 % méthode var construit un ITE avec les bon fils, en appui sur newITE.

Question 4 1½ points

Récursions pour not et and

On demande également de fournir les deux opérations *not* et le *and*.

Pour la négation, `ITE not(ITE e)` rend "vrai" si e est "faux", "faux" si e est "vrai". Sinon, on construit un noeud de même variable que l'original, mais dont les fils sont récursivement la négation des fils du noeud original :

$$\neg ITE(i,t,f) = ITE(i,\neg t,\neg f)$$

.

Pour le ET logique `ITE and(ITE e1, ITE e2)` on sépare trois cas :

- Cas terminaux : $e \wedge e = e$; $true \wedge e = e \wedge true = e$; $false \wedge e = e \wedge false = false$
- Cas propagation : Si $e1.var > e2.var$, alors

$$ITE(i1,t1,f1) \wedge e2 = ITE(i1,t1 \wedge e2, f1 \wedge e2)$$

  Le cas où $e2.var > e1.var$ est symétrique.
- Cas même variable : Si $e1.var = e2.var$, alors

$$ITE(i,t1,f1) \wedge ITE(i,t2,f2) = ITE(i,t1 \wedge t2, f1 \wedge f2)$$

  .

(Notons que le `or(ITE e1, ITE e2)` ne diffère de and que par les cas terminaux : $e \vee e = e$; $true \vee e = e \vee true = true$; $false \vee e = e \vee false = e$. Son code n'est pas demandé.)

Écrire le code des méthodes *not* et *and* dans `ITEFactory`.

Solution:

BDDFac.java

```
@Override
public ITE not(ITE op) {
	if (op == True) {
		return False;
	} else if (op == False) {
		return True;
	} else {
		return newITE(op.getVar(),not(op.getIfTrue()), not(op.
		    getIfFalse()));
	}
}

@Override
public ITE and(ITE l, ITE r) {
	if (l == True) return r;
	if (r == True) return l;
	if (r==False || l == False) return False;
	if (l==r) return l;

	if (l.getVar() < r.getVar()) {
		return newITE(r.getVar(), and(r.getIfTrue(),l), and(r.
		    getIfFalse(),l));
	} else if (r.getVar() < l.getVar()) {
```

```
                    return newITE(l.getVar(), and(l.getIfTrue(),r), and(l.
                        getIfFalse(),r));
                }
                assert l.getVar() == r.getVar();

                return newITE(l.getVar(), and(l.getIfTrue(),r.getIfTrue()), and(l.
                    getIfFalse(), r.getIfFalse()));
```

Barème :

- 30 % cas not correctement traité : cas terminaux et propagation
- 40 % cas terminaux et propagation "simple" du and
- 30 % cas propagation asymétrique