

1. Représentation d'une Structure Booléenne Multi-niveaux (EBM)

1. Les différentes classes

1. La classe Ebm
2. La classe EbmVar
3. La classe EbmExpr

2. Présentation des méthodes récursives

3. Fonctionnement de la fonction `getType()`

4. Travail à Effectuer

1. Question 1
2. Question 2
3. Question 3
4. Question 4
5. Question 5 (facultative)

Représentation d'une Structure Booléenne Multi-niveaux (EBM)

Les différentes classes

Pour pouvoir implanter en mémoire une EBM il est nécessaire de disposer de trois classes:

- Une classe de base `Ebm`
- Une classe `EbmExpr` pour représenter les expressions (dérivant d'`Ebm`).
- Une classe `EbmVar` pour représenter les variables (dérivant d'`Ebm`).

Les squelettes de ces classes sont fournis dans l'énoncé `Ebm.h`, `EbmExpr.h` et `EbmVar.h`. Il vous est demandé de les compléter.

Une implantation de la classe `LogicValue` vous est fournie:

- `LogicValue.h`
- `LogicValue.cpp`

La classe `Ebm`

```
class Ebm {
public:
    enum Type          { Variable=1, Expression=2 };
    enum OperatorType { Not=1, And, Or, Xor, UndefinedOperator };
public:
    static Ebm*      parse      ( std::string );
public:
    unsigned int     literals   ();
    std::set<EbmVar*> support    ();
    LogicValue       eval      ();
    void             support    ( std::ostream& );
    void             display    ( std::ostream& );
};
```

Principales méthodes de la classe `Ebm`:

- `literals()` : retourne le nombre de littéraux de l'EBM. C'est à dire le nombre de fois que des variables

apparaissent dans l'expression.

- `support ()` : retourne le support de l'EBM sous forme d'ensemble (`set<>`) Le support est l'ensemble des variables nécessaire pour calculer la valeur de l'expression.
- `support (std::ostream&)` : Affiche le support dans un flux, voir l'exemple d'exécution du programme de test fourni plus bas.
- `display (std::ostream&)` : Affiche l'EBM sous sa forme humainement lisible, en notation *préfixée*.
- `eval ()` : Calcule la valeur de l'expression booléenne. Le type de retour est un `LogicValue`, c'est à dire une variable pouvant prendre trois valeurs: `Zero`, `One` et `Undefined` (cf. TME 2).
- `parse ()` : Une fonction statique transformant une chaîne de caractères en une EBM.

La classe `EbmVar`

```
class EbmVar: public Ebm {
public:
    // Operations sur les dictionnaires.
    static EbmVar*      get      ( unsigned int index );
    static EbmVar*      get      ( std::string name );
public:
    inline std::string  getName  ();
    inline LogicValue   getValue ();
    inline unsigned int  getIndex ();
    inline void         setValue ( LogicValue value );
private:
                                EbmVar  ( std::string name, LogicValue value=Low );
    virtual                  ~EbmVar  ();
private:
    static unsigned int      _maxIndex;
    static std::map<unsigned int,EbmVar*> _byIndex;
public:
    static EbmVar*          create  ( std::string name, LogicValue value=LogicValue::Zero );
    // Completer ici...
private:
    std::string              _name;        // Variable name.
    LogicValue               _value;      // Logical value.
    unsigned int             _index;      // Unique index (for ROBDD)
};
```

Attributs de la classe `EbmVar`:

- `_name` : Le nom de la variable (par ex. "a")
- `_value` : La valeur actuellement affectée à la variable (`Zero`, `One` ou `Undefined`).
- `_index` : Un index permettant d'identifier de façon unique la variable. Cet index trouvera son utilité dans le TME suivant, présentant les ROBDD.

Attributs *statiques* de la classe `EbmVar`:

- `_maxIndex` : Le compteur utilisé pour générer des index uniques (mécanisme identique à celui utilisé pour les vecteurs lors du TME1).
- `_byName` : Une table associative permettant de retrouver *n'importe quelle* variable sachant son nom.
- `_byIndex` : Une table associative permettant de retrouver *n'importe quelle* variable sachant son index.

Principales méthodes de la classe `EbmVar`:

- `getName ()` : accesseur retournant le nom.
- `getIndex ()` : accesseur retournant l'index.
- `getValue ()` : accesseur retournant la valeur.
- `setValue ()` : modificateur de la valeur.
- `get ()` : deux surcharge différentes pour retrouver une variable par son nom ou par son index.

- `create()` : crée *ou* retourne, si elle existe déjà un pointeur sur la variable demandée.

La classe `EbmExpr`

```
class EbmExpr : public Ebm {
private:
    OperatorType    _operator; // operateur de ce noeud
    std::list<Ebm*> _operands; // pointeurs sur les operandes

public:
                                EbmExpr    ( OperatorType, std::list<Ebm*>& operands );
    virtual                    ~EbmExpr    ();
    static EbmExpr*            Not        ( Ebm* );
    static EbmExpr*            Or         ( Ebm*, Ebm* );
    static EbmExpr*            Or         ( Ebm*, Ebm*, Ebm* );
    static EbmExpr*            Or         ( Ebm*, Ebm*, Ebm*, Ebm* );
    static EbmExpr*            Xor        ( Ebm*, Ebm* );
    static EbmExpr*            Xor        ( Ebm*, Ebm*, Ebm* );
    static EbmExpr*            Xor        ( Ebm*, Ebm*, Ebm*, Ebm* );
    static EbmExpr*            And        ( Ebm*, Ebm* );
    static EbmExpr*            And        ( Ebm*, Ebm*, Ebm* );
    static EbmExpr*            And        ( Ebm*, Ebm*, Ebm*, Ebm* );

public:
    inline OperatorType        getOperator ();
    inline std::list<Ebm*>&    getOperands ();
};
```

Attributs de la classe `EbmExpr`:

- `_operator` : le type de l'opérateur (dans `CaoTypes.h`, parmi `Not`, `And`, `Or` ou `Xor`).
- `_operands` : une liste de pointeurs vers les opérands.

Principales méthodes de la classe `EbmVar`:

- `getOperator()` : accesseur, retourne le type de l'opérateur.
- `getOperands()` : accesseur, retourne une référence sur la liste des opérands (pour éviter une copie inutile de la liste).
- `Or()` : trois surcharges construisant une expression `Or` entre ses `N` opérands. Le but de ces différentes fonctions est de cacher la construction de la liste des opérands et l'allocation de l'`EbmExpr` résultante. Noter que cela implique une construction *bottom-up* de l'EBM complète. Voir l'exemple fourni.
- `And()` et `Xor()` : idem.

Présentation des méthodes récursives

Les méthodes `literals()`, `support()`, `display()` et `parse()` de la classe `Ebm` sont récursives. On définit le niveau de profondeur de l'arbre d'une EBM comme le nombre d'arcs `N` nécessaire pour atteindre la racine. Dans l'EBM `e4`, la racine `e4` a une profondeur nulle, `e1` une profondeur de 1 et `a` une profondeur de 2.

Récursion

Considérons l'exemple de la fonction `literals()`. Dire qu'elle est récursive signifie que son calcul au niveau de profondeur `N` peut se décomposer en calculs *de cette même fonction* appliquée sur le niveau `N+1`. Par exemple, le calcul de `literals()` de `e4` peut s'exprimer comme la somme des fonctions `literals()` appliquée à ses opérands (`e0`, `e1`, `e2` et `e3`):

Condition d'arrêt

On comprend bien, que l'on ne va pas rappeler indéfiniment `literals()`, à une profondeur donnée, il faudra bien d'arrêter. Le critère d'arrêt est simple: lorsque l'on atteint une `EbmVar`, le noeud n'a pas d'opérandes, et son nombre de littéraux vaut 1.

En conclusion, la fonction récursive aura la structure générale suivante:

```
LITERALS(e)
  IF e IS_A variable:
    RETURN 1
  ELSE
    sum = 0
    FOR EACH operand OF e:
      sum += LITERALS(operand)
    RETURN sum
```



La figure suivante représente l'ordre dans lequel les noeud de l'arbre de l'EBM sont parcourus lors d'une récursion.



État de la pile d'appel des fonctions a un instant donné du parcours récursif. Le chemin actuellement parcouru est (e4, e1, a)



La pile d'appel des fonctions membres à un instant donné.

Attention : il est très important de bien distinguer le code la fonction, qui est écrit une fois (dans `Ebm`) et l' *objet* auquel elle s'applique. Nous avons ainsi la même fonction membre `display()` mais sur trois objets différents : `e4`, `e1` et `a`.



Fonctionnement de la fonction `getType()`

Dans la présentation de la récursion, nous avons vu qu'il était nécessaire, pour choisir entre la récursion et l'arrêt, de savoir distinguer une variable (`EbmVar`) d'une expression (`EbmExpr`).

Pour cela nous allons implanter la fonction `getType()`.

La fonction `getType()` a les caractéristiques suivantes:

- Elle est déclarée virtuelle pure dans la classe `Ebm`, c'est à dire qu'elle **doit** être sur-définie dans les classes dérivées **et** qu'elle n'a aucune implémentation au niveau de la classe `Ebm`.
- Dans la classe `EbmVar` elle retourne **Variable**.
- Dans la classe `EbmExpr` elle retourne **Expression**.

De cette façon, lorsqu'elle est appelée dans les fonctions récursives définies dans `Ebm`, elle retournera la valeur associée au *type dynamique* de l'`Ebm` en cours de traitement. C'est à dire `EbmVar` ou `EbmExpr`.



Travail a Effectuer

Question 1

Implanter `EbmVar`:

- Ajouter le dictionnaire manquant.
- Ajouter la fonction de création statique.
- Implanter les méthodes *ordinaires*.
- Les constructeurs & destructeurs ne sont pas vides: que doivent-il contenir?

Question 2

Implanter `EbmExpr`:

- Penser à une façon simple pour les opérateurs logiques d'appeler le constructeur générique.

Question 3

Implanter `Ebm`:

- Le squelette du fichier vous est fourni: `Ebm.cpp`. Il contient le code de la fonction `Ebm::parse()`
- Implanter la déclaration (dans `Ebm`) et les deux sur-définitions de la fonction `getType()` (dans `EbmVar` & `EbmExpr`).
- Implanter les fonctions `Ebm::literals()`, `Ebm::support()`, `Ebm::eval()` et `Ebm::display()`.

Question 4

Tester l'ensemble avec le programme suivant: `Main.cpp`.

Question 5 (facultative)

Il existe une autre façon d'implanter le mécanisme de récursivité ne nécessitant pas de fonction `getType()`, suggérez une approche alternative (utilisant plus intensivement l'héritage).