

Logique Ternaire

TP 2/3

Objectifs

- ★ Se familiarisez avec la syntaxe du C++.
- ★ Comprendre le compilateur

Contraintes

- Indentez vos fichier.
- La correction tiendra compte de la brièveté des méthodes que vous écrivez (évités les fonctions de plus de 25 lignes) ; n'hésitez pas à découper une méthodes en plusieurs sous-méthodes (privées) plus courtes.
- Votre code ne doit pas donner d'erreurs avec Valgrind (ni fuite mémoire, ni autre erreurs).
- Vous ne devez pas utiliser de fonction C quand un équivalent C++ existe.
- Pour l'UML, vous pouvez utiliser UMLet ou Umbrello.
- Les noms de classe commencent par une majuscule.
- Les noms de méthodes et d'attributs commencent par une minuscule.
- La convention de nommage des accesseur est `get_nom_attribut()` et `set_nom_attribut(...)`.
- Vous devez fournir un Makefile qui compile vos fichiers source et contient une règle clean ainsi qu'un programme de test.
- Le **code source et les diagrammes** doivent être *pusher* sur le git du TP

Préparation du TP

- Cloner votre répertoire sur votre compte
`git clone https://git` l'adresse qui vous a été attribué.
- Pendant le TP n'oubliez pas de comiter régulièrement
- Le répertoire contient un fichier `main.cc` qui vous servira de programme de test.
- N'oubliez pas de pusher l'ensemble

Le dépôt git ne doit pas contenir d'exécutable, ni de fichier objet, ni de fichier temporaire (*.)

Concept



Comme l'illustre le *strip* de xkcd, on a parfois besoin de pouvoir raisonner sur plus de deux valeurs (yes/no ou vrai/faux). Parfois la réponse peut être *je ne sais pas*.

En mathématique, il existe une logique sur trois valeurs (https://fr.wikipedia.org/wiki/Logique_ternaire), c'est cette logique que nous allons représenter dans ce TP. La figure suivante donne les tables de vérités des opérateurs NON, ET et OU.

A	B	NON A	A ET B	A OU B
T	T	F	T	T
T	F	F	F	T
T	U	F	U	T
F	T	T	F	T
F	F	T	F	F
F	U	T	F	U
U	T	U	U	T
U	F	U	F	U
U	U	U	U	U

TABLE 1 – T : true ; F : false ; U : Unknown

Nous allons donc représenter des expressions logiques de la logique ternaire. Une **expression logique** est constituée d'un ensemble de **variables logiques combinées par des opérations logiques**. Chaque **variable logique** peut prendre une **valeur ternaire** (soit *true*, *false* ou *unknown*). Les variables sont combinées par des opérateurs logiques, OR, AND et NOT. Un exemple d'expression logique *exp* est $exp = (a_1 \text{ AND } ((\text{NOT } a_2) \text{ OR } a_3))$, où a_1 , a_2 et a_3 sont des **variables logiques**. Si a_1 vaut *true*, a_2 vaut *false* et a_3 vaut *unknown*, alors l'expression *exp* vaut (*true AND ((NOT false) OR unknown)*), c'est-à-dire *true*.

Une expression logique ne sera pas directement manipulée. En fait on utilisera les types suivants :

- un *Atome* représente un atome logique. Il est constitué d'un nom unique et d'une valeur ternaire. De plus, la valeur ternaire *val* doit pouvoir être réinitialisée par l'utilisateur.
- un *Not* représente la négation d'une expression logique. Il est constitué de son unique opérande *opd* qui est une expression logique, et de son nom, "NOT".
- une expression logique binaire *ExpBin* correspond à une opération logique entre deux expressions logiques. Elle est constituée de deux opérandes, *opd1* et *opd2*, qui sont deux expressions logiques.
- un *And* est une expression logique binaire qui correspond au "AND" logique.
- un *Or* est une expression logique binaire qui correspond au "OR" logique.

Une expression logique est représentée par la classe abstraite *ExpLog*, dans le fichier d'en-tête *ExpLog.hh*.

→ Dessiner la hiérarchie de classe.

1 Les valeurs logiques

Proposer un type `ThreeVal_t`, qui permet de représenter les valeurs logiques ternaires (T,F,U).

Implémenter l'opérateur de sortie de flux (<<) pour afficher les valeurs ternaires. Le prototype de l'opérateur est le suivant (où `Type` est votre type) :

```
std::ostream& operator<<(std::ostream & out, Type val);
```

Vous pouvez maintenant l'utiliser comme ceci :

```
Type t = F;
std::cout << t << std::endl;
```

2 Classe Atome

Écrire la classe `Atome` qui permet de compiler le main suivant (fourni), on suppose que T, F et U représentent les valeurs ternaires. La fonction `toString()` retourne une chaîne de caractère de la forme `(a_1 = val)` où `a_1` est le nom (unique de l'atome généré automatiquement) et `val` sa valeur.

Implémenter la méthode `evaluate()` qui retourne la valeur ternaire de l'atome.

```
int main() {
    Atom a(T);
    Atom b;
    Atom c(b);
    b = T;
    cout << a.toString() << endl;
    cout << b.toString() << endl;
    cout << c.toString() << endl;
    c = false;
    a = b = F;
    cout << a.toString() << endl;
    cout << b.toString() << endl;
    cout << c.toString() << endl;
    cout << a.evaluate() << endl;
}
```

```
(a_0 = T)
(a_1 = T)
(a_2 = U)
(a_0 = F)
(a_1 = F)
(a_2 = F)
F
```

3 ExpNot

La classe `ExpNot` représente une expression logique à une opérande. Écrire la classe correspondante au main suivant et qui réalise l'affichage donné.

Implémenter la méthode `evaluate()` qui retourne la négation ternaire de l'atome.

```
int main() {
    Atom a(T);
    ExpNot n1(a);
    ExpNot n2(n1);
    cout << n1.toString() << endl;
    cout << n2.toString() << endl;
    cout << n2.evaluate() << endl;
}
```

```
NOT(a_0 = T)
NOTNOT(a_0 = T)
T
```

4 Classe ExpBin

La classe abstraite `ExpBin` contiendra le constructeur d'une expression à deux opérandes.

5 Classe ExpOr et ExpAnd

Écrire ces deux classes qui sont des expressions binaires et permettent de compiler le main suivant :

```

int main(){
    Atom a(T);
    Atom b;
    Atom c(F);
    ExpNot n1(a);
    ExpAnd and1(n1,b);
    ExpAnd and2(c,b);
    ExpOr or1(and1,and2);
    cout << and1.toString() << std::endl;
    cout << or1.toString() << "==" <<
    or1.evaluate() << endl;
}

```

```

(NOT(a_0 = T) AND (a_1 = U))
((NOT(a_0 = T) AND (a_1 = U)) OR ((a_2 = F)
AND (a_1 = U))) = F

```

6 Plus d'opérations

Ajouter les opérations manquantes pour que le main suivant fonctionne et affiche le bon résultat.

```

int main()
{
    Atom a(U),b(U),c(U);
    ExpNot nota(a);
    ExpAnd and1(a , b);
    ExpAnd and2(nota , c);
    ExpAnd and3(b , c);
    ExpOr or1(and1,and2);
    ExpOr or2(or1,and3);

    cout << or2.toString() << endl;

    while(or2 == U)
    {
        if(a == U)
        {
            a = T;
            continue;
        }
        if(b == U)
        {
            b = T;
            continue;
        }
        if(c == U)
        {
            c = F;
            continue;
        }
    }
    cout << or2.toString() << endl;
}

```

```

(((a_0 = U) AND (a_1 = U)) OR (NOT(a_0 = U) AND (a_2 = U))) OR ((a_1 = U) AND (a_2 = U))
(((a_0 = T) AND (a_1 = T)) OR (NOT(a_0 = T) AND (a_2 = U))) OR ((a_1 = T) AND (a_2 = U))

```

7 Lecture d'un fichier DIMACS

Nous allons utiliser nos structures pour représenter des formules de logique sous forme normale conjonctive (CNF). La formule est une conjonction de clauses où chaque clause est une disjonction de littéral. Les littéraux sont nos variables booléennes.

Par exemple :

```
(a_0 OR a_1 OR a_2) AND (a_3 OR NOT(a_4)) AND (NOT(a_0) OR a_3)
```

est une formule CNF.

Un problème classique est de trouver une valeur pour chacune des variables qui rend la formule vraie. Dans l'exemple précédent, $a_0 = T$, $a_1 = F$, $a_2 = T$, $a_3 = T$ et $a_4 = F$ résout le problème. Ce problème est un problème NP-complet classique mais il existe de nombreux solveurs (appelés SAT-solver) qui sont capables de résoudre des formules avec des millions de littéraux.

Il existe un format de fichier pour définir des formules sous forme CNF, le format DIMACS. Le format est défini comme montré sur la figure 1

File format

The benchmark file format will be in a simplified version of the DIMACS format:

```
c
c start with comments
c
c
p cnf 5 3
1 2 3 0
4 -5 0
-1 4 0
```

- The file can start with comments, that is lines beginning with the character `c`.
- Right after the comments, there is the line `p cnf nbvar nbclauses` indicating that the instance is in CNF format; `nbvar` is the exact number of variables appearing in the file; `nbclauses` is the exact number of clauses contained in the file.
- Then the clauses follow. Each clause is a sequence of distinct non-null numbers between `-nbvar` and `nbvar` ending with 0 on the same line; it cannot contain the opposite literals `i` and `-i` simultaneously.
- Positive numbers denote the corresponding variables.
- Negative numbers denote the negations of the corresponding variables.

FIGURE 1 – Format DIMACS

Ajouter les méthodes manquantes pour construire une expression logique à partir d'un fichier au format DIMACS.

Bonus : Proposer un algorithme pour résoudre les formules CNF.

Bonus : Remplacer les tests du main par des tests unitaires en utilisant la bibliothèque *catch* (cf. TP1)