Compte-Rendu du projet /-/ Charles Bongiorno

Introduction

Ce projet s'inscrit dans le cadre du module d'ouverture de C++.
Notre objectif principal était de traiter des expressions mathématiques simple.
Exemple : (((6.5 * 2) - 8) + 7)

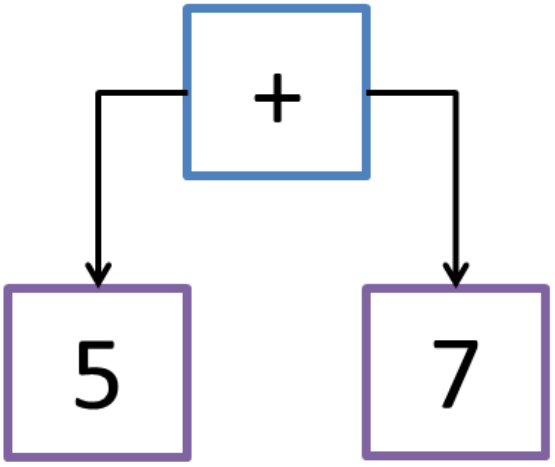
Ces expressions se limiteront aux opérateurs arithmétiques simple (addition, soustraction, multiplication, division) avec des constantes flottantes et des variables.
Dans un premier temps, ces expressions seront créées en dur dans le code.
Pour ensuite être construite à partir d'une chaine de caractères.

Cas d'utilisation

Lors du lancement du programme, vous serez invité à entrer une expression algorithmique simple, en notation polonaise inversé.
Actuellement, il n'y a pas de traitement en cas d'erreur de saisie, merci de vérifier vos entrées.
Le programme va ensuite afficher l'expression sous différentes formes.
De l'hypothèse où vous avez entré des variables nommées, il vous sera proposé de leur assigner une valeur.
La programme affichera alors l'expression, avec un résultat.

Représentation des classes

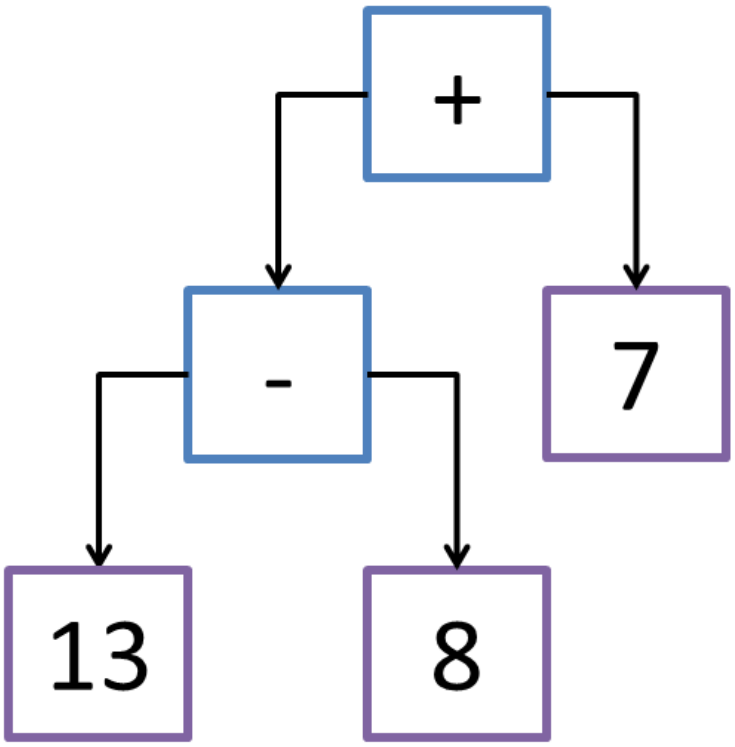
Afin de répondre au sujet, nous avons effectué une recherche sur un ensemble de classe pour matérialiser une expression mathématique.
Une expression est constitué (au minimum) d'un opérateur entre deux nombres.
Par exemple : *(5 + 7)*



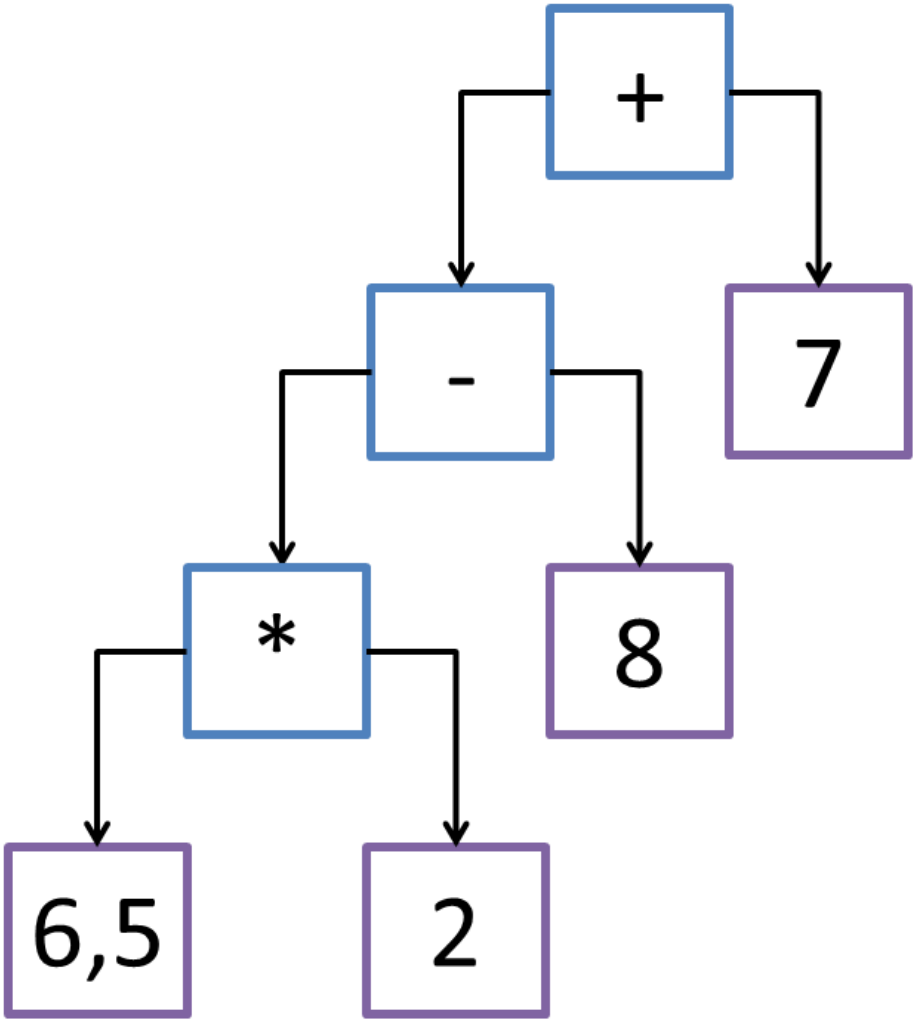
Ici le + est l'opérateur (encadré en bleu) et les constances des opérandes (encadré en violet).
(nota: nous utiliserons le même code couleur pour les exemples suivants, et le vert sera utilisé pour les variables)

Les opérateurs qui seront utilisés dans ce projet sont addition, soustraction, multiplication, division.
Nous allons donc avoir une classe mère 'opérateur' composé de deux opérandes (droite et gauche) et des classes qui en hériterons (+ - * /).
De même, nous aurons une classe générique 'opérande' et des classes filles 'constante' et 'variable'.

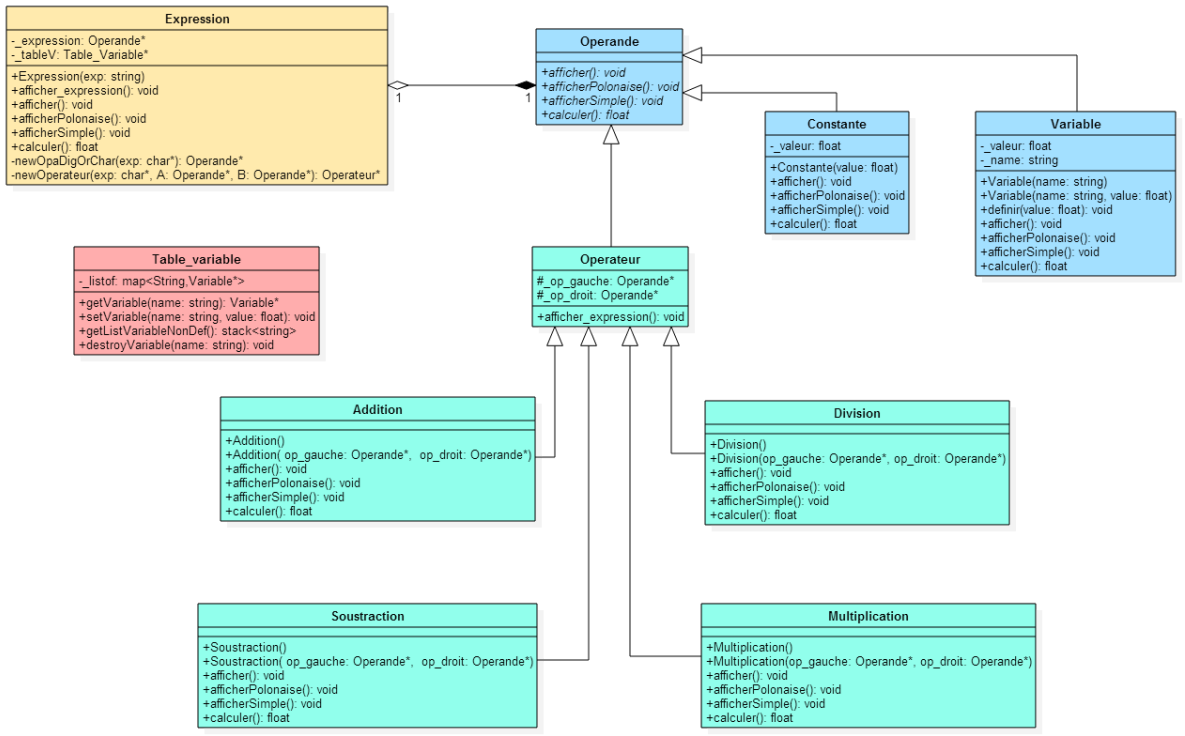
Maintenant, prenons le cas d'une expression plus longue : *((13 - 8) + 7)*



Dans cette exemple, l'opérateur '-' a remplacé l'opérande '5'.
Nous généralisons donc un opérateur comme une sorte d'opérande.
Ainsi nous pouvons cascadé les opérateurs les uns sous les autres afin de construire de longue expression.



Nous arrivons donc à la représentation statique des classes :



(voir l'image)

Détails des classes

Classe Expression

Fonction de création d'une expression à partir d'une chaine de caractére Cette classe fait office d'interface pour l'utilisation des expressions.

Fonctions membres publiques

Expression (std::string exp) Création de la classe **Expression**. Permet d'analyser une expression sous forme d'une chaine de caractère et d'en construire l'opérande associée.
void afficher_expression ()
void afficher ()
void afficherPolonaise ()
void afficherSimple ()
void calculer ()

Fonctions membres privées

Operande * newOpaDigOrChar (char *exp) Création d'un **Operande**, **Constante** ou **Variable** suivant si c'est un nombre ou pas.
Operateur * newOperateur (char *exp, **Operande** *A, **Operande** *B) Création d'un **Operateur**, suivant le caractère de l'opération à réaliser.

Attributs privés

Operateur * _expression
Table_Variable * _tableV

Classe Operande

Opérande d'une opération.

Fonctions membres publiques

Operande ()
virtual ~Operande ()
virtual void afficher ()=0 Afficheur par défaut de l'opérande.
virtual void afficherPolonaise ()=0 Afficheur de l'opérande en version polonaise.
virtual void afficherSimple ()=0 Afficheur de l'opérande en mode simplifié

virtual float calculer ()=0 Permet d'éffectuer l'opération contenu.

Classe Constante

Héritage de **Operande**
Valeur définie d'une expression.

Fonctions membres publiques

Constante (float value)
Constante (const **Constante** &o)
~Constante ()
void afficherPolonaise () Afficheur de l'opérande en version polonaise.
void afficherSimple () Afficheur de l'opérande en mode simplifié.
float calculer () Permet d'éffectuer l'opération contenu.

Attributs privés

float _valeur

Classe Variable

Héritage de **Operande**
Valeur inconnue d'une expression La valeur assigné est défini à NULL par défaut.

Fonctions membres publiques

Variable (std::string name) Constructeur par défaut.
Variable (std::string name, float value) Constructeur avancé
~Variable () Destructeur.
void definir (float value) Assignation d'une valeur à la variable.
void afficher () Afficheur par défaut de l'opérande.
void afficherPolonaise () Afficheur de l'opérande en version polonaise.
void afficherSimple () Afficheur de l'opérande en mode simplifié
float calculer () Permet d'éffectuer l'opération contenu.

Attributs privés

std::string _name
float _value = NULL

Classe Operateur

Héritage de **Operande**
Opérateur d'une opération.

Fonctions membres publiques

Operateur (void)
Operateur (**Operande** *op_gauche, **Operande** *op_droit)
Operateur (const **Operateur** &o)
virtual ~Operateur ()
void afficher_expression () Affiche l'opération et son résultat.

Attributs protégés

Operande * _op_gauche
Operande * _op_droit

Classe Table_Variable

Héritage de **Singleton< Table_Variable >**
Tableau des variables, association entre une lettre et une variable La gestion des variables se fait à l'aide d'une map qui associe une lettre avec un pointeur vers une variable.

Fonctions membres publiques

void setVariable (std::string name, float value) Assignation d'une valeur à une variable Recherche dans les variables enregistrées (_listof) celle ayant comme nom 'name'. Assignation d'une valeur pour cette variable s'il elle existe, création de la variable dans le cas contraire.

Variable * getVariable (std::string name) Obtetion d'une variable Test l'existance d'une variable, la crée dans le cas contraire Et la retourne.
void destroyVariable (std::string name) Destruction d'une variable.
std::stack< std::string > getListVariableNonDef () Getter d'un la liste des variables n'ayant pas de valeur assigniée Création d'un stack pour le retour (list). Parcours des variables enregistrées (_listof). Ajout dans le stack des variables n'ayant pas de valeur assigniée (item.second->calculer() == NULL).

Fonctions membres privées

Table_Variable ()
~Table_Variable ()

Attributs privés

std::map< std::string, Variable * > _listof

Construction d'une expression à partir d'une chaine de caractères

Voici le détail de la construction d'une expression à partir d'une chaine de caractère.
Dans la version actuelle, ce programme ne sait traiter que les expressions sous forme polonaise inversé.
Nous utiliserons l'exemple *"10 X * 7 +"*
*((10 * X) + 7)*

Je parcours ma chaine de gauche à droite, par bloc de trois.
Je trouverai ainsi, d'après le schéma d'une notation polonaise inversé :

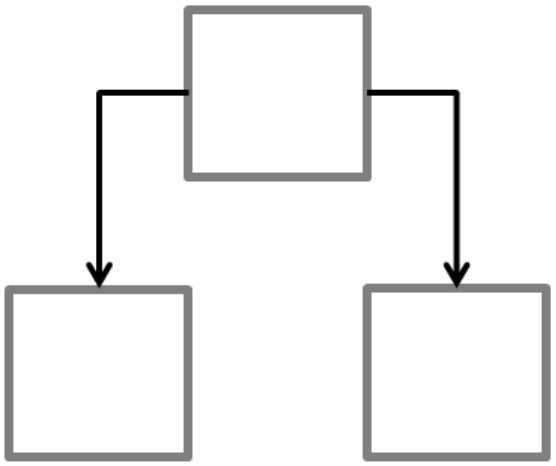
- un premier opérande
- un second opérande
- un opérateur

Pour chaque opérande l'algorithme se demandera s'il s'agit d'une constante (nombre) ou d'une constante.
Dans le premier cas, il créera une nouvelle constante, dans le seconde, il utilisera la classe **Table_Variable** pour obtenir un pointeur vers une variable.
Ensuite, pour chaque opérateur, un comparaison du caractère permettra de connaitre l'opération à effectuer.

Enfin, on recommence l'algorithme avec comme 1er opérande, le pointeur vers l'opérateur précédemment créé ainsi que la suite de la chaine de caractère.

Voici une animation résumant l'algorithme:

10 X * 7 +



Les variables

Les variables se comportent comme des constances, à l'exception près que s'il n'y a pas de valeur défini, elles bloqueront le processus de calcul d'une expression.
Lors de leur initialisation, elle ne possède pas de valeur (défini à null dans le code) mais on pourra faire l'assignation à l'aide de la méthode : void definir(float value)

Gestion avec la table des variables

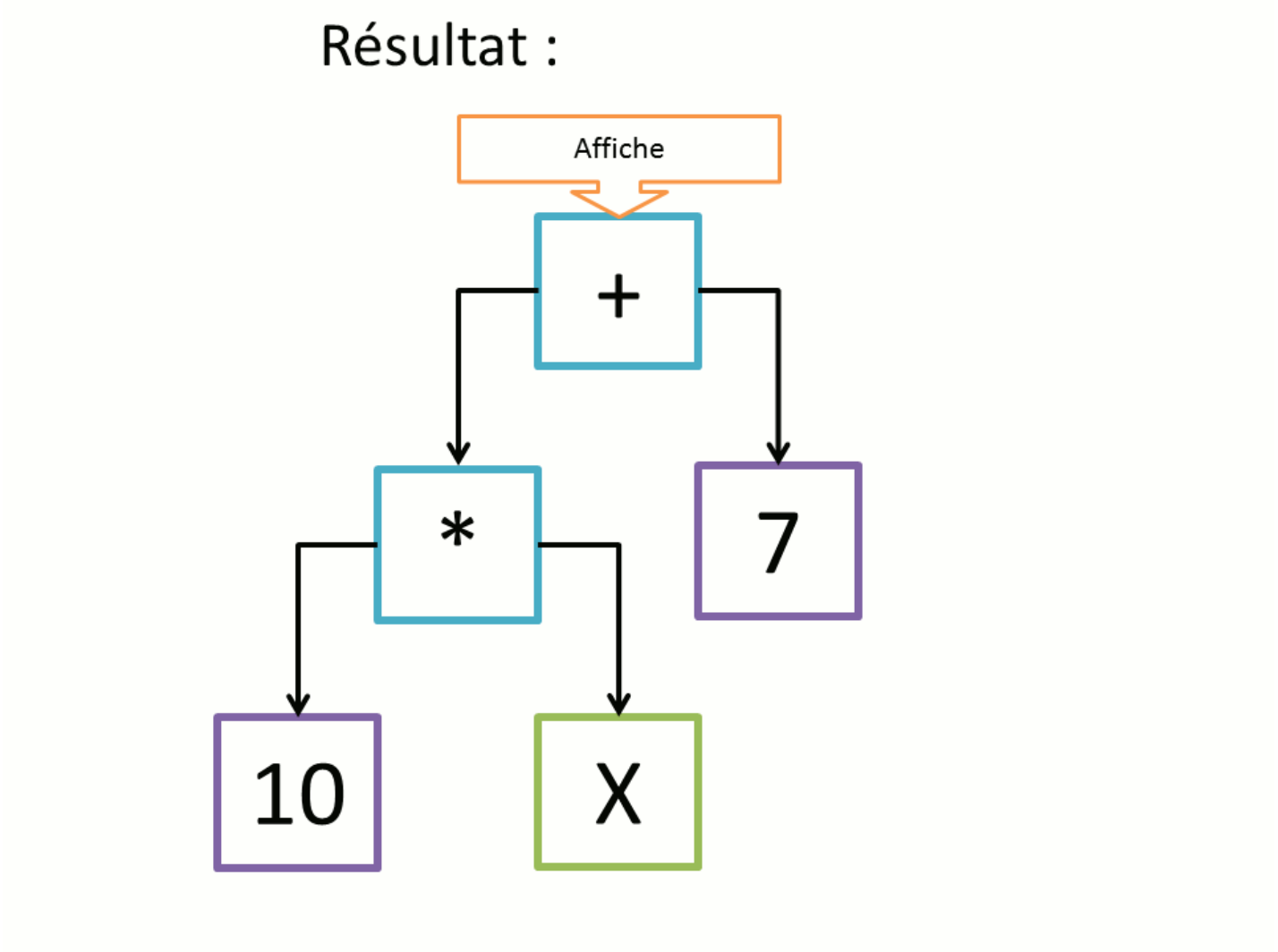
Afin de mutualiser les variables, j'utilise un gestionnaire de variable.
Ce dernier sera un singleton afin qu'il n'y ai qu'une seule instance de la classe dans le programme.
Avec ceci, nous pourrons avoir une seule instance par variable même si nous utilisons plusieurs fois le même caractère dans le programme.

L'algorithme fonctionne à l'aide d'une map (sorte de tableau associatif) entre un caractère (string) et une valeur (int).
Lors que l'on demande l'instance d'une variable avec son nom, le gestionnaire va regarder s'il possédé déjà une variable avec ce nom.
Si c'est le cas, il retourne l'instance de la variable, sinon il crée une nouvelle instance, retient le pointeur et renvoit l'instance.

Parcours de l'expression

Afin d'afficher l'expression ou de calculer son résultat, il nous faut parcourir cette ensemble d'opération. Ce parcours va s'effectuer par récursivité.
De manière générale, on demande l'information aux autres membres (opérande gauche et droite), on ajoute notre information et on renvoie.

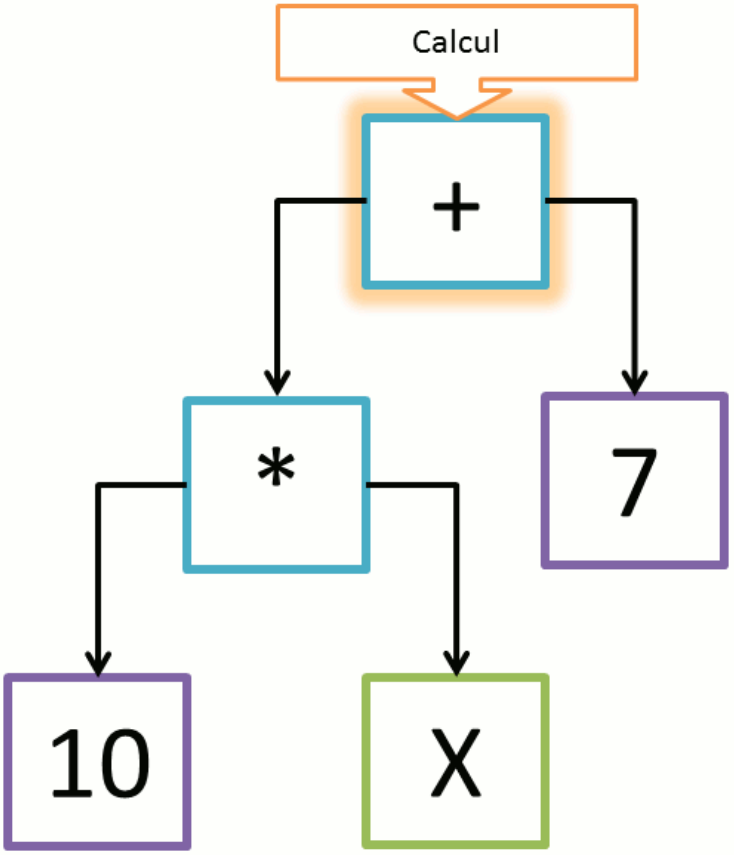
Voici une animation résumant l'algorithme:



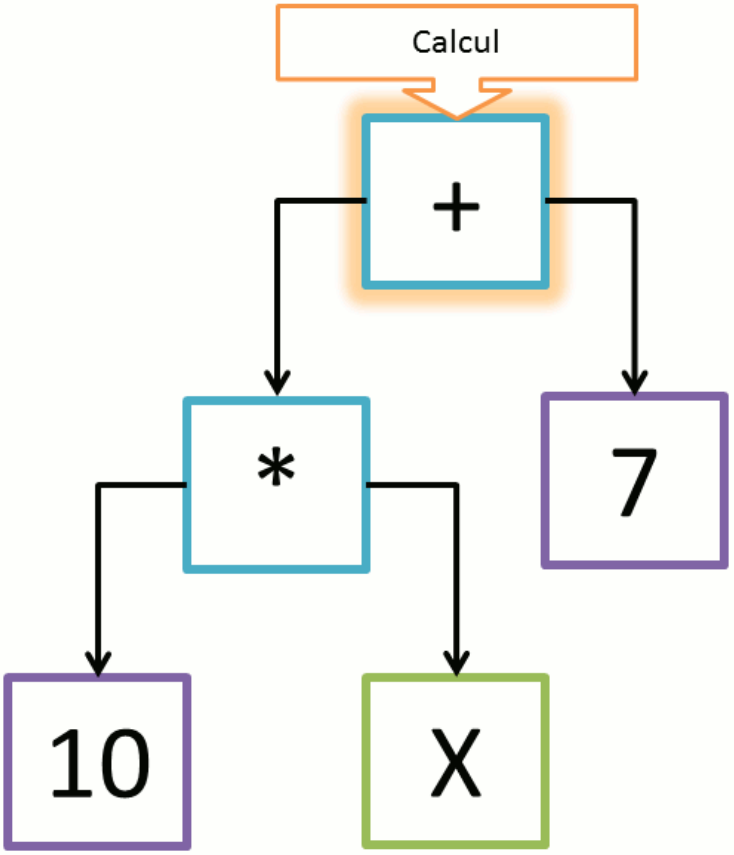
Cas de calculer

Lors de l'appel à la méthode calculer, il y a aura quelques différences de fonctionnement dû aux variables.
Comme vu précédemment, par récursivité, l'appel à la méthode va se propager dans toute l'expression.
Pour une constante, la valeur de retour sera celle de la constante.
Pour un variable, si une valeur est assignée, son comportement sera identique à une constante.
En revange, s'il n'y a pas de valeur assignée, la valeur null sera renvoié : la valeur est inconnue.
Enfin, pour un opérateur, cela effectuera l'opération entre les deux opérandes, et retournera le résultat.
Une specificité existe, s'il reçoit une valeur null d'un des deux opérandes, l'opérateur renverra lui aussi la valeur null, afin d'indiquer que le calcul n'est pas possible.
De même une division par zéro, retournera la valeur null pour les même raisons.

Voici une animation résumant l'algorithme dans le cas où X a une valeur assignée :



Voici une animation résumant l'algorithme dans le cas où X n'a pas de une valeur assignée :



Crédits

Projet_Expression
Module d'ouverture C++
Isen-Toulon
Par Charles BONGIORNO