Université Cheikh Anta Diop

Ecole Supérieure Polytechnique

Département Génie Informatique

PROJET: Analyse et évaluation d'expressions arithmétiques(JAVA)

Membres du groupe

- Mariama Baldé
- Salif Diallo
- Ndeye Penda Diene
- Mame Sira Diop
- Ndeye Ndioro Diop

Professeur : Mr Fall

Année : 2023/2024

PLAN

Introduction

Ce projet a été initié pour développer un analyseur syntaxique et un évaluateur d'expressions en Java.

L'objectif de ce projet est de développer une application qui valide et évalue les expressions arithmétiques fournies par l'utilisateur. Le programme doit vérifier la syntaxe des expressions pour s'assurer qu'elles respectent les règles des opérations mathématiques. Une fois la syntaxe validée, il doit être capable de calculer la valeur de l'expression.

Le programme doit gérer les opérations arithmétiques de base comme l'addition, la soustraction, la multiplication et la division. En plus de cela, il doit prendre en charge l'utilisation des parenthèses pour définir explicitement l'ordre des opérations, garantissant ainsi des résultats précis même pour des expressions complexes.

Pour mener à bien ce projet, nous devons implémenter un analyseur syntaxique en utilisant une grammaire définie en BNF (Backus-Naur Form) comme spécifié dans l'énoncé. Cette grammaire formalise la structure des expressions arithmétiques, permettant ainsi de vérifier leur validité conformément aux règles établies. Grâce à cette approche, nous serons en mesure de déterminer avec précision si une expression donnée est syntaxiquement correcte.

Le programme devra également gérer les erreurs de syntaxe et fournir un retour approprié à l'utilisateur en cas de détection d'une expression incorrecte. Une fois que l'analyse syntaxique est réussie, le programme doit évaluer l'expression et afficher sa valeur.

Le programme doit également gérer les erreurs de syntaxe et fournir un retour approprié à l'utilisateur en cas de détection d'une expression incorrecte. En cas d'erreur de syntaxe, un message clair va indiquer le problème rencontré. Une fois que l'analyse syntaxique est réussie, le programme évalue l'expression arithmétique et affiche sa valeur. Cela garantit que l'utilisateur est informé des erreurs dans ses entrées et reçoit les résultats attendus lorsque l'expression est correcte.

I. .Contexte et Objectifs

Contexte du Projet :

En informatique, les analyseurs syntaxiques et les évaluateurs d'expressions sont essentiels pour interpréter, manipuler et traiter des données mathématiques. Ces outils vérifient la validité syntaxique des expressions arithmétiques et les convertissent souvent en une forme plus facilement évaluable, comme la notation postfixée (ou notation polonaise inversée).

L'analyse syntaxique décompose une expression en composants élémentaires (opérandes et opérateurs) et vérifie leur agencement selon les règles de syntaxe. Cela permet de détecter les erreurs et de prévenir l'évaluation d'expressions incorrectes.

L'évaluation des expressions se fait généralement après leur conversion en notation postfixée, simplifiant le calcul à l'aide de structures de données comme les piles (stacks). Cette conversion élimine la nécessité des parenthèses et permet une évaluation plus directe.

Objectifs du Projet :

Ce projet vise à concevoir et implémenter un analyseur syntaxique et un évaluateur d'expressions en Java. Ces composants doivent valider la structure syntaxique des expressions arithmétiques (notation infixée), les convertir en notation postfixée (RPN), puis évaluer ces expressions pour produire des résultats numériques précis.

En développant ce projet, on évalue des expressions arithmétiques et on crée un outil fondamental pour la manipulation des données mathématiques. Il implique la validation des expressions, leur conversion en notation postfixée et leur évaluation finale. Chaque étape requiert une compréhension approfondie des structures de données, des algorithmes et des compétences en programmation.

Ainsi, la création d'un analyseur syntaxique et d'un évaluateur d'expressions contribue significativement à la technologie sous-tendant de nombreux aspects de la manipulation des données mathématiques dans les systèmes informatiques modernes.

II. Implémentation

1. Architecture Logiciel

L'architecture logicielle est essentielle pour assurer qu'un analyseur syntaxique et un évaluateur d'expressions fonctionnent de manière efficace et fiable.

Ces composants doivent être conçus de manière modulaire, ce qui permet de séparer clairement les différentes responsabilités.

a. Analyseur Syntaxique

Ce composant vérifie la structure et la validité syntaxique des expressions mathématiques. Il décompose l'expression en opérandes (chiffres) et opérateurs (comme +, -, *, /), s'assurant que chaque élément est correctement agencé selon les règles de la syntaxe mathématique

b. Conversion Infixée vers PostFixée

Une fois l'analyse syntaxique réussie, l'expression infixée est convertie en notation postfixée (ou RPN). Cette transformation simplifie l'évaluation en éliminant la nécessité d'utiliser des parenthèses pour définir l'ordre d'exécution des opérations. Les expressions postfixées sont évaluées plus efficacement à l'aide de structures de données comme les piles, qui permettent un traitement linéaire et séquentiel des opérandes et opérateurs.

Ex:On suppose que l'on dispose d'une expression infixée où chaque opération est ''parenthésée''. Par exemple, ((5 * 3) + 4)).

Dans une première étape, on vérifie que l'expression infixe est correctement parenthésée.

Le principe : on part d'une pile vide et on lit l'expression infixe de la gauche vers la droite.

– Si la chaîne lue est une parenthèse ouvrante, on la place dans la pile ;

– Si la chaîne lue est une parenthèse fermante, on dépile la parenthèse ouvrante correspondante.

L'expression est correctement ''parenthésée'' si la pile est vide à la fin.

4. Écrire une fonction verif qui prend une expression infixée et qui vérifie qu'il y a autant de parenthèses ouvrantes que fermantes.

NB:Ceci est très important à comprendre car on essayera de prendre le chemin inverse pour évaluer de la droite vers la gauche.On fera l'exemple ci dessus dans la partie Test&Résultats
Source:https://cahier-de-prepa.fr/pc-kerichen/download?id=2097

c. Evaluation Expression Postfixée

Ce dernier composant effectue le calcul final de l'expression postfixée. Il utilise les opérandes empilés dans la pile pour exécuter les opérations selon l'ordre établi par la notation postfixée. Cette approche garantit une évaluation précise et efficace des expressions mathématiques, même lorsqu'elles sont complexes.

2. Détails Techniques(Code)

Dans l'implémentation on utilise des structures de données telles que des piles (stacks) et des boucles itératives pour gérer les opérations de manière efficace. Les algorithmes sont conçus pour être robustes et traiter une variété de scénari, y compris des expressions avec différents niveaux de complexité.
Notre code se compose de 4 classes principales:

- La Classe **EvaluateurExpression**:C'est là que se trouve toutes les méthodes nous permettant de faire une conversion infixée vers postfixée,l'évaluation des expressions postfixées et une partie de l'analyseur syntaxique

```java
import java.io.IOException;
import java.util.Scanner;
import java.util.Stack;
public class EvaluateurExpression {
    public EvaluateurExpression(){

    }

    // Vérifier si l'expression est valide
    public boolean estExpressionValide(String expression) {
        if (expression.equals(anObject:null)) {
            return false;
        }
        else{
            // Règles de validation de base : pas d'opérateurs consécutifs, parenthèses équilibrées, etc.
            String tabOperateur[]={"*","/","+","-"};
            for (String operateur1 : tabOperateur) {
                for (String operateur2 : tabOperateur) {
                    if (expression.contains(operateur1+operateur2)) {
                        return false;
                    }
                }
            }

            Stack<Character> stack = new Stack<>();
            for (char c : expression.toCharArray()) {
                if (c == '(') {
                    stack.push(c); //On ajoute a la pile
```

- La Classe **CommandeLine**:Cette classe nous permet d'effacer l'écran le cmd de l'utilisateur et peut même afficher en plein écran le cmd si on utilise les OS(LINUX,WINDOWS,MACOS)

```java
public class CommandeLine {
    public void effacerConsole() {
        /**
         * Permet d'effacer la console
         */
        String osName = System.getProperty(key:"os.name").toLowerCase();
        String commande;
        // Vérifier le système d'exploitation
        if (osName.contains(s:"windows")) {
            // Système Windows
            commande = "cls"; // Commande pour effacer l'écran sur Windows
        } else if (osName.contains(s:"linux") || osName.contains(s:"unix") || osName.contains(s:"mac")) {
            // Système Linux ou Unix
            commande = "clear"; // Commande pour effacer l'écran sur Linux/Unix et mac
        } else {
            // Autres systèmes d'exploitation
            System.out.println(x:"Système d'exploitation non pris en charge.");
            return;
        }
        try {
            // Créer le processus
            ProcessBuilder processBuilder = new ProcessBuilder();
            if (osName.contains(s:"windows")) {
                processBuilder.command(...command:"cmd", "/c", commande);
            } else {
                processBuilder.command(commande);
            }
```

- La Classe **LireUtile** : C'est simplement la classe qui permet de supprimer les espaces dans la saisie de l'expression par l'utilisateur.Il renvoie l'expression même s'il on peut avoir des opérateurs additifs consécutifs ou multiplicatifs.Ceci est géré au niveau de **EvaluateurExpression**

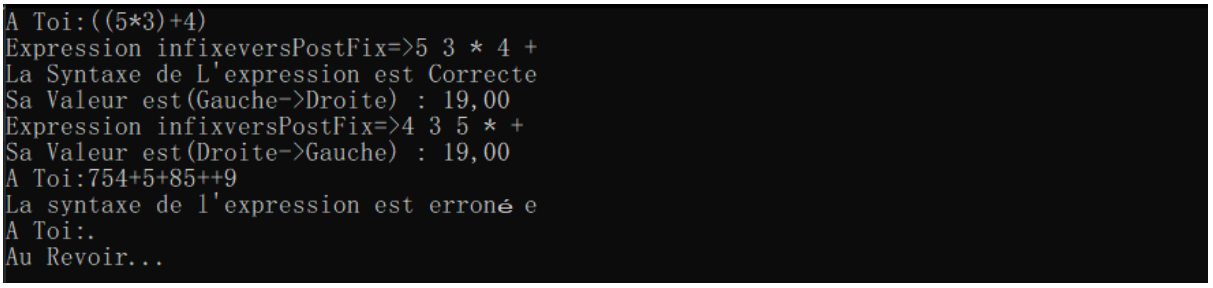
```java
    private StringBuilder expression = new StringBuilder();

    public LireUtile() {
        // Initialisation
    }

    public String lire_Utile() {
        expression.setLength(newLength:0); // Réinitialise l'expression
        System.out.print(s:"A Toi:");
        try {
            int caractere = System.in.read();
            expression.append(caractere);
            // Arrêter immédiatement si le premier caractère est un point
            if (caractere == '.') {
                expression.append(c:'.');
                return expression.toString();
            }

            boolean dernierCaractereNombre = false; // Indicateur pour vérifier si le dernier caractère lu était un nombre

            while (true) {
                caractere = System.in.read();

                // Vérifier si le caractère est '='
                if (caractere == '=') {
                    break;
                }

                char charActuel = (char) caractere;

                if (Character.isDigit(charActuel)) {
                    // Si le caractère actuel est un chiffre
```

- La Classe **Main** sert de point d'entrée principal de l'application Java, orchestrant l'initialisation, la gestion des entrées utilisateur, et la coordination des opérations d'analyse et d'évaluation d'expressions arithmétiques.


```java
    private StringBuilder expression = new StringBuilder();

    public LireUtile() {
        // Initialisation
    }

    public String lire_Utile() {
        expression.setLength(newLength:0); // Réinitialise l'expression
        System.out.print(s:"A Toi:");
        try {
            int caractere = System.in.read();
            expression.append(caractere);
            // Arrêter immédiatement si le premier caractère est un point
            if (caractere == '.') {
                expression.append(c:'.');
                return expression.toString();
            }

            boolean dernierCaractereNombre = false; // Indicateur pour vérifier si le dernier caractère lu était un nombre

            while (true) {
                caractere = System.in.read();

                // Vérifier si le caractère est '='
                if (caractere == '=') {
                    break;
                }

                char charActuel = (char) caractere;

                if (Character.isDigit(charActuel)) {
                    // Si le caractère actuel est un chiffre
```

3. Traitement Des Erreurs

Pour le traitement des erreurs, on a réussi la deuxième version ou l'erreur ne tue pas le programme mais l'évaluation uniquement comme dans le programme.Et tout ceci nous mène à donner le pourquoi du langage java dans ce projet



III. Avantages du Langage Java

Comme on a pu le voir l'utilisation des structures de données telles que des piles (stacks) et des boucles itératives pour gérer les opérations de manière efficace sont très importantes. Et, en se documentant on a constaté que Java est un langage de programmation évolué et fortement typé.

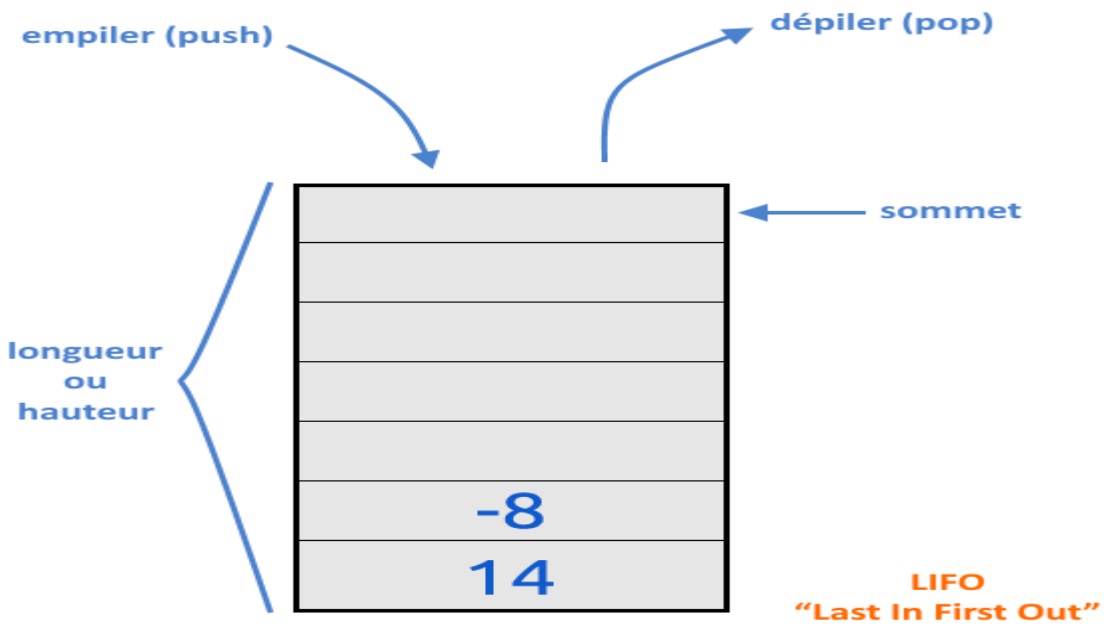
En effet, avec avec une multitude de bibliothèques comme Java.util.Stack pas besoin de créer beaucoup méthodes car on a les méthodes

- **Push (Empiler)** : Ajouter un élément au sommet de la pile.
- **Pop (Dépiler)** : Retourner et Retirer l'élément situé au sommet de la pile.
- **Peek (Regarder)** : Retourner simplement l'élément en haut de la pile sans le retirer.
- **empty (Est vide)** : Vérifier si la pile est vide.
- **isFull (Est pleine)** : Vérifier si la pile est pleine (dans le cas d'une pile de taille fixe)
- **firstElement(element qui est en bas index=0)**:Retourne le premier élément de la pile

En Java, l'utilisation des collections comme Stack offre une gestion plus sûre et plus pratique des données grâce à la gestion dynamique de la mémoire et aux méthodes intégrées pour la manipulation des éléments. Contrairement à C, où la gestion de la mémoire et des structures de

données est plus manuelle, Java simplifie ces aspects tout en offrant une performance comparable.

.



On a notamment les classes StringBuilder et StringBuffer qui sont deux classes en Java utilisées pour manipuler des chaînes de caractères de manière efficace, notamment lorsque des opérations de concaténation ou de modification fréquentes sont nécessaires.

Pour StringBuilder on a :

la Mutabilité :

- StringBuilder est mutable, ce qui signifie que son contenu peut être modifié après sa création.

la Performance :

- StringBuilder est non synchronisé, ce qui le rend plus rapide que StringBuffer dans les environnements non multi-threads.

une utilisations recommandée :

- Préférable lorsque la synchronisation n'est pas requise, par exemple dans des opérations de manipulation de chaînes dans des applications non multi-threadées.

Pour StringBuffer on a:

Mutabilité :

- StringBuffer est également mutable, permettant des modifications de contenu après sa création.

Synchronisation :

- Contrairement à StringBuilder, StringBuffer est synchronisé, ce qui garantit la sécurité des opérations dans un environnement multi-threadé en permettant l'accès simultané sans erreur.

Performance :

- En raison de sa synchronisation, StringBuffer peut être plus lent que StringBuilder dans des environnements non multi-threads.

Utilisations recommandées :

- Utilisé lorsque la manipulation de chaînes nécessite la synchronisation, ce qui est typique dans des applications multi-threadées.

Beaucoup d'autres éléments montrent que le bon choix se portait sur java.

IV. Tests & Résultats

Dans notre code, nous avons inclus des mécanismes de gestion d'erreur robustes pour assurer la fiabilité des résultats.

On va montrer détails par détails ceux qui nous a permis d'aboutir aux différents résultats

On va prendre comme exemple celui du libellé du projet

Expression=3+125*5/-6+10

Pour faire l'évaluation de la gauche vers la droite on va d'abord vérifier si notre syntaxe est correcte et voici la partie du code qui s'en occupe **EvaluerExpression<estExpressionValide**

```java
// Vérifier si l'expression est valide
public boolean estExpressionValide(String expression) {
    if (expression.equals(anObject:null)) {
        return false;
    }
    else{
        // Règles de validation de base : pas d'opérateurs consécutifs, parenthèses équilibrées, etc.
        String tabOperateur[]={"*","/","+","-"};
        for (String operateur1 : tabOperateur) {
            for (String operateur2 : tabOperateur) {
                if (expression.contains(operateur1+operateur2)) {
                    return false;
                }
            }
        }

        Stack<Character> stack = new Stack<>();
        for (char c : expression.toCharArray()) {
            if (c == '(') {
                stack.push(c); //On ajoute a la pile
                // System.out.println(stack);
            }
            else if (c == ')') {
                if (stack.isEmpty()) {
                    System.out.println(x:"Parentheses non equilibre");
                    return false;
                }
                // System.out.println("Before Pop ");
                // System.out.println(stack);
                stack.pop(); //supprime le dernier objet qui est en haut
                // System.out.println("After Pop ");
                // System.out.println(stack);
            }
        }
        // System.out.println(stack.isEmpty());
    return stack.isEmpty();
    }
}
```

On va faire une vérification pour savoir si on a des combinaisons d'opérations additif ou multiplicatif deux a deux.Dans tabOperateur on fait toutes les combinaisons possibles et si l'on trouve 1 que sa soit **++,+-,+*,+/,...** Alors l'expression est incorrecte false.

Cependant qu'en est il des parenthèses?

On va prendre ce petit exemple et décommenter les print
Ex2:(58+9)+(47+7-2))

```
A Toi:(58+9)+(47+(7-2))=
Before Pop
[(]
After Pop
[]
Before Pop
[(, (]
After Pop
[(]
Before Pop
[(]
After Pop
[]
Expression infixeversPostFix=>58 9 + 47 7 2 - + +
```

Comme on le voit il va toujours ajouter quand on des parenthèses mais si on a une parenthèse fermante il fera **pile.pop** qui enlève le top de la pile.

Si tout se passe bien l'expression devra être transformé de infixe a postfixe

Comment on transforme en PostFixe

```java
public String reecritureExpressionGD(String expression) {
    StringBuilder resultat = new StringBuilder();
    Stack<Character> pile = new Stack<>();
    for (int i =0 ; i <expression.length(); i++) {
        Character c = expression.charAt(i);
        if (Character.isDigit(c)) {
            StringBuilder tmp=new StringBuilder();
            tmp.append(c);
            // System.out.println(tmp);
                while (i<=expression.length()-2 && Character.isDigit(expression.charAt(i+1)) ) {
                    tmp.append(expression.charAt(++i));
                    // System.out.println(tmp);
                }
            // System.out.println("Fin");
            resultat.append(tmp);
        }
        else if (c == '(') {
            pile.push(c);
        }
        else if (c == ')') {
            while (!pile.isEmpty() && pile.peek() != '(') {
                resultat.append(c:' ').append(pile.pop());
            }
            pile.pop(); // Enleve la parenthese ouvrante
        }
        else if (estOperateur(c)) {
            resultat.append(c:' ');
            //System.out.println(pile);
            while (!pile.isEmpty() && aPriorite(c, pile.peek())) {
                resultat.append(pile.pop()).append(c:' ');
            }
            // System.out.println("Ajout pile");
            pile.push(c);
            // //System.out.println(pile);
        }
    }
    while (!pile.isEmpty()) {
        resultat.append(c:' ').append(pile.pop());
    }
    return resultat.toString();
}
```

C’est cette méthode qui va transformer l’expression infixe en postfixe
Et celui ci s’occupe du calcul:

```java
// Évaluer une expression postfixée
public  double calculExpressionGaucheDroite(String expression) {
    Stack<Double> pile = new Stack<>();
    try (Scanner scanner = new Scanner(expression)) {
        while (scanner.hasNext()) {
            if (scanner.hasNextDouble()) {
                pile.push(scanner.nextDouble());
            } else {
                double b = pile.pop();
                double a = pile.pop();
                switch (scanner.next()) {
                    case "+":
                        pile.push(a + b);
                        break;
                    case "-":
                        pile.push(a - b);
                        break;
                    case "*":
                        pile.push(a * b);
                        break;
                    case "/":
                        pile.push(a / b);
                        break;
                }
            }
        }
    }
    return pile.pop();
}
```

Et donc si on veut faire de la droite vers la gauche:
il suffit de, pour la réécriture de l’expression, de commencer par la gauche vers droite puis de changer les parenthèses ouvrantes en parenthèses fermantes et une petite subtilité pour le calcul au lieu d’avoir a-b on aura b-a et pour (a/b) on aura (b/a) ce qui va permettre de calculer de la droite vers la gauche

```java
public  String DroiteGauche(String expression) {
    StringBuilder resultat = new StringBuilder();
    Stack<Character> pile = new Stack<>();
    for (int i = expression.length()-1; i >=0; i--) {
        Character c = expression.charAt(i);
        if (Character.isDigit(c)) {
            StringBuilder tmp=new StringBuilder();
            tmp.append(c);
            while (i>=1 && Character.isDigit(expression.charAt(i-1)) ) {
                tmp.append(expression.charAt(--i));
            }
            tmp.reverse();
            resultat.append(tmp);
        }
        else if (c == ')') {
            pile.push(c);
        }
        else if (c == '(') {
            while (!pile.isEmpty() && pile.peek() != ')') {
                resultat.append(c:' ').append(pile.pop());
            }
            pile.pop(); // Enleve la parenthese fermante
        }
        else if (estOperateur(c)) {
            if (i!=0) {
                resultat.append(c:' ');
            }
            //System.out.println(pile);
            while (!pile.isEmpty() && aPriorite(c, pile.peek())) {
                resultat.append(pile.pop()).append(c:' ');
            }
            // System.out.println("Ajout pile");
            pile.push(c);
            // //System.out.println(pile);
        }
    }
    while (!pile.isEmpty()) {
        resultat.append(c:' ').append(pile.pop());
    }
    return resultat.toString();
}
```

```java
public  double evaluerExpressionDroiteGauche(String expression) {
    // Convertir l'expression infixée en postfixée (notation polonaise inversée)
    String reecritureExpressionDG = DroiteGauche(expression);
    System.out.println("Expression infixversPostFix=>"+reecritureExpressionDG);
    // Évaluer l'expression postfixée
    return evaluerExpressionDG(reecritureExpressionDG);
}

    // Évaluer une expression postfixée
    public  double evaluerExpressionDG(String expression) {
        Stack<Double> pile = new Stack<>();
        try (Scanner scanner = new Scanner(expression)) {

            while (scanner.hasNext()) {
                if (scanner.hasNextDouble()) {
                    pile.push(scanner.nextDouble());
                }
                else {
                    double b = pile.pop();
                    double a = pile.pop();
                    switch (scanner.next()) {
                        case "+":
                            // System.out.println("a="+a+"et b="+b);
                            pile.push(b + a);
                            break;
                        case "-":
                            // System.out.println("a="+a+"et b="+b);
                            pile.push(b - a);
                            break;
                        case "*":
                            // System.out.println("a="+a+"et b="+b);
                            pile.push(b * a);

                            break;
                        case "/":
                        // System.out.println("a="+a+"et b="+b);
                            pile.push(b / a);
                            break;
                    }
                }
            }
        }
        return pile.pop();
    }
```

2. Résultats

```
A Toi:3+125*5/7-6+10
Expression infixeversPostFix=>3 125 5 * 7 / + 6 - 10 +
La Syntaxe de L'expression est Correcte
Sa Valeur est(Gauche->Droite) : 96,29
Expression infixversPostFix=>10 6 + 7 5 / 125 * - 3 +
Sa Valeur est(Droite->Gauche) : 76,29
A Toi:45+95-58*2*(7*2)-(4-9)=
Before Pop
[(]
After Pop
[]
Before Pop
[(]
After Pop
[]
Expression infixeversPostFix=>45 95 + 58 2 * 7 2 * * - 4 9 - -
La Syntaxe de L'expression est Correcte
Sa Valeur est(Gauche->Droite) : -1479,00
Expression infixversPostFix=>9 4 - 2 7 * 2 * 58 * - 95 - 45 +
Sa Valeur est(Droite->Gauche) : -1489,00
A Toi:.
Au Revoir...
```

V. Bilan & Perspectives D’améliorations

- **Réussites et Perspectives :**
  Ce projet nous a permis de développer une solution fonctionnelle et efficace pour la validation et l'évaluation d'expressions arithmétiques en Java. Nous avons surmonté avec succès les défis liés à la gestion des structures de données et à l'optimisation des algorithmes pour assurer la précision et la rapidité des calculs.
- **Perspectives Futures :**
  Pour améliorer notre application, nous envisageons d'ajouter le support de fonctions mathématiques avancées et de variables, ainsi que de l'intégrer dans des applications plus larges nécessitant des capacités de calcul sophistiquées. Par exemple on peut introduire les logarithmes,les exponentielles modulo et d’autres fonctions.

Conclusion

Ce projet nous a permis de développer une solution fonctionnelle et efficace pour la validation et l'évaluation d'expressions arithmétiques en Java. Nous avons conçu un système modulaire et robuste qui sépare clairement les responsabilités de chaque composant, allant de l'analyse syntaxique à la conversion en notation postfixée, jusqu'à l'évaluation finale des expressions.

Les défis rencontrés, notamment la gestion des erreurs de syntaxe et l'optimisation des algorithmes de calcul, ont été surmontés grâce à une compréhension approfondie des structures de données et des algorithmes en Java. Les avantages du langage Java, tels que ses bibliothèques riches et ses mécanismes de gestion de la mémoire, ont été pleinement exploités pour garantir la performance et la fiabilité de notre application.