Projet Génie Logiciel

Extension Tab

Réalisé par :
Mouez JAAFOURA, Ilyas MIDOU, Meriem KAZDAGHLI,
Salim KACIMI, Mehdi DIMASSI

Date : 23 janvier 2025

Grenoble INP - Ensimag & Phelma
Université Grenoble Alpes

Table des matières

1 Contexte de l'extension TAB **2**

2 Cahier des charges **2**
2.1 Déclaration de Table 2
2.2 Bibliothèque 2

3 Analyse Syntaxique **2**

4 Analyse Contextuel **5**
4.1 Règles pour les Déclarations 5
4.2 Règles pour les Types Valides 5
4.3 Règles pour les Initialisations 5
4.4 Règles sur les indices 5
4.5 Conception 6
4.6 Exemple 7

5 Analyse de Géneration de code **8**
5.1 Règles sur les indices 8
5.2 Règles sur les opérations : Division et modulo 8
5.3 Règles sur l'allocation de mémoire (pile) 9
5.4 Conception 9

6 Cahier des Charges : Bibliothèque **10**
6.1 Règles sur les opérations 10
6.2 MatrixFloat 10
6.3 Bibliothèque : ListFloat 11

7 Limitation **13**

8 Validation **14**

9 Conclusion **14**

10 Bibliographie **15**

1 Contexte de l'extension TAB

Dans le cadre de notre projet, nous avons entrepris le développement d'une extension TAB pour le langage de programmation Deca, en nous inspirant de la sémantique de Java et C. Cette approche vise à garantir que notre extension soit non seulement conforme aux standards établis par ces langages, mais aussi qu'elle réponde aux besoins spécifiques des utilisateurs de Deca. Nous prévoyons d'apporter les ajustements nécessaires pour adapter cette fonctionnalité au contexte particulier de Deca, tout en préservant la simplicité et l'essence qui caractérisent ce langage.

2 Cahier des charges

2.1 Déclaration de Table

- Le nom de la table doit être unique.
- Les types des colonnes doivent être valides et correspondre à ceux déclarés pour l'identifiant du tableau.
- Une table ne peut pas contenir plus de 10 000 cases limité a la taille de la pile.
- Les noms des colonnes ne peuvent pas être des mots-clés réservés du langage Deca.
- Chaque colonne doit avoir un type bien défini (aucun type indéfini ou ambigu n'est autorisé).
- Lors de la création d'un tableau non initialisé, tous les éléments du tableau sont définis par des valeurs par défaut selon leur type :
  - Pour les `int` et `float`, la valeur par défaut est `0`.
  - Pour les `boolean`, la valeur par défaut est `false`.
  - Pour les objets, la valeur par défaut est `null`.
- Les opérations effectuées sur les tables doivent respecter la cohérence des types (par exemple, aucune opération ne doit être réalisée entre un `int` et une chaîne de caractères).

2.2 Bibliothèque

- **Création de tableaux :** Permettre la création de tableaux multidimensionnels (nD) de taille fixe.
- **Affichage des tables :** Offrir la possibilité d'afficher les informations d'une table dans un format lisible.
- **Modification des éléments :** Permettre la modification d'un ou de plusieurs éléments d'un tableau.
- **Somme de tableaux :** Implémenter la somme de deux tableaux selon des règles définies.
- **Produit matriciel :** Réaliser le produit de deux matrices.
- **Transposition :** Calculer la transposée d'une matrice.
- **Déterminant :** Calculer le déterminant d'une matrice.

3 Analyse Syntaxique

Cette section présente les règles de grammaire nécessaires ainsi que les modifications à introduire pour intégrer l'extension `tab` dans le langage Deca. Ces règles couvrent la déclaration, l'initialisation et l'accès aux éléments d'une table.

Les règles suivantes définissent la création des tables, en s'inspirant de la syntaxe des tableaux en Java. Une table peut être déclarée avec le mot-clé `new`, suivi de son type et de ses dimensions, ou bien par une affectation directe de ses valeurs sauf pour les tableaux des objets.

— **Exemple 1 :** `int[][] tab = new int[5][10];`
(Création d'une table avec 5 lignes et 10 colonnes, initialisée avec des valeurs par défaut 0.)

— **Exemple 2 :** `int[][] tab = [[1, 2], [1, 2]];`
(Création d'une table directement initialisée avec des valeurs spécifiques.)

— **Exemple 3 :** `A[] lisA = new A[5]; lisA[0] = new A(); ...`
(Création d'un tableau d'objets A avec 5 éléments nuls, puis remplissage de chaque case avec un objet A ou un de ses sous-types abstraits)

Même si, dans un tableau, on choisit de conserver les mêmes opérations arithmétiques, logiques ou de comparaison, nous devons nous assurer que ces opérations sont compatibles avec les types des éléments stockés dans ces tableaux. En d'autres termes, bien que les mêmes types d'opérations (comme l'addition, la multiplication, ou les comparaisons) puissent être appliquées à différents éléments d'un tableau, il faut vérifier que ces opérations sont valides pour le type de données que nous manipulons. Cela permet d'éviter des erreurs de type ou des comportements indéfinis.

Exemples :

Opérations arithmétiques sur un tableau d'entiers : Si nous avons un tableau d'entiers, nous pouvons effectuer des opérations arithmétiques comme l'addition, la soustraction, la multiplication, etc., entre les éléments du tableau.

Opérations arithmétiques sur un tableau d'entiers : Si nous avons un tableau d'entiers, nous pouvons effectuer des opérations arithmétiques comme l'addition, la soustraction, la multiplication, etc., entre les éléments du tableau.

```
{
    int array[] = [1, 2, 3, 4, 5];  // Declaration d'un tableau avec des
        valeurs initiales
    int i = 0;

    while (i < 5) {
        array[i] = array[i] + 2;  // Ajoute 2 a chaque element du tableau
        i = i + 1;
    }
}
// Resultat : array = [3, 4, 5, 6, 7]
```

Listing 1 – Exemple d'utilisation des operations arithmetiques

Opérations logiques sur un tableau de booléens : Si nous avons un tableau de valeurs booléennes, nous pouvons appliquer des opérations logiques comme && (et logique), || (ou logique) et ! (non logique).

```
{
    boolean[] flags = [true, false, true, false];
    int i = 0;
    while (i < 4) {
        flags[i] = !flags[i];  // Inverse chaque valeur booleenne
        i = i + 1;
    }
}
// Resultat : flags = [false, true, false, true]
```

Listing 2 – Exemple d'optimisation avec && dans Deca et IMA

Les contraintes imposées par le langage Deca nous ont forcés à modifier certaines règles afin de garantir le bon fonctionnement de notre extension TAB. En effet, la grammaire du langage Deca n'était initialement pas conçue pour gérer les tableaux dans les opérations. Cela a nécessité une modification de la syntaxe du langage Deca pour permettre une détection correcte des tableaux.

De plus, pour que les opérations expliquées ci-dessus puissent être effectuées sur ces tableaux, nous avons dû ajuster la grammaire et la construction des tableaux dans le langage, ainsi que la gestion des types et des opérations applicables.

Grammaire du langage Deca (extension TAB)

$$\textbf{dec\_var\_set} \rightarrow \textbf{type taille list\_decl\_var}$$

$$\textbf{taille} \rightarrow (\text{'['} \; \text{']'})^*$$

$$\begin{aligned}\textbf{primary\_expr} \rightarrow\ & \textbf{ident} \\ & \mid \textbf{ident}\,\text{'('}\,\textbf{list\_expr}\,\text{')'} \\ & \mid \text{'('}\,\textbf{expr}\,\text{')'} \\ & \mid \text{'}\textbf{readInt}\text{'}\,\text{'('}\,\text{')'} \\ & \mid \text{'}\textbf{readFloat}\text{'}\,\text{'('}\,\text{')'} \\ & \mid \textbf{new ident (} \;\; \text{'('}\,\textbf{expr}\,\text{')'} \;\; \mid \textbf{dim} \;\; \textbf{)} \\ & \mid \text{'('}\textbf{type}\text{')'}\,\text{'('}\textbf{expr}\text{')'} \\ & \mid \textbf{literal} \\ & \mid \textbf{tableau\_literal}\end{aligned}$$

$$\textbf{dim} \rightarrow \text{'['}\,\textbf{expr}\,\text{']'}\,(\text{'['}\,\textbf{expr}\,\text{']'})^*$$

$$\begin{aligned}\textbf{select\_expr} \rightarrow\ & \textbf{primary\_expr} \\ & \mid \textbf{select\_expr} \;\; \text{'.'} \;\; \textbf{ident} \;\; (\text{'('}\,\textbf{list\_expr}\text{')'} \mid \epsilon) \\ & \mid \textbf{table\_element}\end{aligned}$$

$$\textbf{table\_element} \rightarrow (\text{'['}\,\textbf{expr}\,\text{']'})^+$$

$$\textbf{tableau\_literal} \rightarrow \text{'['}\,\textbf{tableau\_Integer} \mid \textbf{tableau\_Float} \mid \textbf{tableau\_Boolean}\,\text{']'}$$

$$\begin{aligned}\textbf{tableau\_Integer} \rightarrow\ & \text{'['}\,\textbf{tableau\_Integer}\,\text{']'} \;\; (\text{','}\,\text{'['}\,\textbf{tableau\_Integer}\,\text{']'})^* \\ & \mid \textbf{INT}\;(\text{','}\,\textbf{INT})^*\end{aligned}$$

$$\begin{aligned}\textbf{tableau\_Float} \rightarrow\ & \text{'['}\,\textbf{tableau\_Float}\,\text{']'} \;\; (\text{','}\,\text{'['}\,\textbf{tableau\_Float}\,\text{']'})^* \\ & \mid \textbf{FLOAT}\;(\text{','}\,\textbf{FLOAT})^*\end{aligned}$$

$$\begin{aligned}\textbf{tableau\_Boolean} \rightarrow\ & \text{'['}\,\textbf{tableau\_Boolean}\,\text{']'} \;\; (\text{','}\,\text{'['}\,\textbf{tableau\_Boolean}\,\text{']'})^* \\ & \mid (\textbf{TRUE}|\textbf{FALSE})\;(\text{','}\,(\textbf{TRUE}|\textbf{FALSE}))^*\end{aligned}$$

4 Analyse Contextuel

Les règles contextuelles à implémenter pour l'extension tab dans Deca concernent principalement la vérification de la cohérence des déclarations, des initialisations, et des accès aux tables.

4.1 Règles pour les Déclarations

— **Unicité du nom de la table :** Lors de la déclaration d'un tableau, il est impératif que son nom soit unique au sein de son contexte de déclaration. Cela signifie qu'un tableau ne peut pas partager le même nom qu'une autre variable déjà déclarée dans le même espace de noms.

— **Validation des types des éléments :** Lors de la déclaration d'un tableau, chaque type spécifié pour ses éléments doit être conforme aux types valides définis dans le langage Deca.

4.2 Règles pour les Types Valides

— `int` : Le tableau contient uniquement des nombres entiers.

— `float` : Le tableau contient uniquement des nombres à virgule flottante.

— `boolean` : Le tableau contient uniquement des valeurs booléennes, `true` ou `false`.

— `class` : La classe doit être déclarée au préalable.

4.3 Règles pour les Initialisations

— **Correspondance des dimensions :** Lors de l'initialisation d'un tableau, le nombre de dimensions spécifiées dans les crochets doit correspondre à la déclaration initiale du tableau. Cela garantit que la structure du tableau est conforme à ce qui a été défini au moment de sa déclaration. Exemple : int [ ] [ ] matrix = new int [2][2] ;

— **Compatibilité des types :** Les valeurs assignées à un tableau doivent être compatibles avec le type déclaré du tableau.Cela inclut le type des éléments ainsi que le respect des conversions autorisées (e.g., pas de conversion implicite entre int et boolean).

Exemple d'erreur :

```
int [ ] Table = new int [1];
Table[1] = "text";   Erreur : "text" n'est pas un int.
```

Lorsqu'un tableau est utilisé avec un type non compatible ou une dimension mal déclarée, cela peut entraîner des erreurs contextuelles. Par exemple, une tentative de traitement d'un tableau d'entiers comme un tableau de flottants sans conversion explicite génère une erreur.

```
Exemple d'erreur :float[] floatArray = new float[5];
int[] intArray = floatArray;
Erreur : tentative de conversion incompatible entre types.
```

4.4 Règles sur les indices

Il est important de vérifier que les indices utilisés pour accéder aux éléments d'un tableau sont bien des entiers. En effet, lors de l'évaluation d'une expression comme `tab[i-3]=1;`, le type de l'indice doit être un entier, car les indices d'un tableau doivent toujours être des entiers dans ce contexte. Toutefois, il est important de noter que les valeurs exactes des indices ne sont pas connues à l'avance, surtout dans le cas d'opérations arithmétiques comme `tab[i-3]`. Le calcul de l'indice peut dépendre de la valeur de `i`, et il est donc essentiel de s'assurer que cette valeur sera compatible avec l'accès aux éléments du tableau sans provoquer d'erreurs d'exécution ou d'index hors limites. Cette vérification sera effectuée dans la section de génération de code (codegen).

4.5 Conception

La méthode verifyExpr de la classe TableAllocation illustre une approche rigoureuse de l'analyse contextuelle pour l'allocation de tableaux dans l'extension TAB.

— Validation des dimensions du tableau :
Chaque dimension spécifiée dans une allocation est vérifiée pour s'assurer qu'elle est de type entier. Cela garantit que les indices des tableaux sont bien définis et compatibles avec les attentes du langage.
En cas de non-conformité, une erreur contextuelle explicite est levée avec une indication claire de la localisation du problème.

— Vérification du type de base :
Le type des éléments du tableau (par exemple, int, float, ou une classe) est vérifié en le comparant à l'environnement global des types (compiler.environmentType).
Cela permet de détecter les références à des types non définis ou incompatibles et de prévenir les incohérences au moment de l'allocation.

— Création de types tableaux à N dimensions :
La méthode prend en charge la construction dynamique de types tableaux en fonction des dimensions spécifiées.

Des méthodes spécifiques ont été implémentées dans les classes IntTableLiteral, FloatTableLiteral, BooleanTableLiteral, et StringTableLiteral. Les méthodes verifyExpr dans ces classes ont été implémentées pour :

— Valider les types des éléments du tableau :
Chaque élément du tableau est vérifié individuellement pour garantir qu'il correspond au type attendu (par exemple, int pour IntTableLiteral).

— Vérifier les dimensions du tableau :
La structure du tableau (1D, 2D, ou plus) doit respecter les règles syntaxiques et sémantiques du langage Deca.

— Créer des types spécifiques pour les tableaux :
Un type unique est généré dynamiquement pour chaque tableau, incluant des informations sur son type de données et ses dimensions (par exemple, 2D float pour un tableau bidimensionnel de nombres flottants).

— Préserver et retourner le type validé :
Une fois validé, le type du tableau est stocké dans l'objet courant et renvoyé pour une utilisation ultérieure dans les phases de compilation.

4.6 Exemple

```
{
    int a;
    int[]tab_entier=[10,5];
    print(tab_entier[0]," ",tab_entier[1]);
}
```

Listing 3 – Programme 1

Exemple d'arbre décoré de programme 1

```
‘> [5, 0] Program
   +> ListDeclClass [List with 0 elements]
   ‘> [5, 0] Main
      +> [5, 0] ListDeclVar [List with 2 elements]
      |  []> [7, 8] DeclVar
      |  ||  +> [7, 4] Identifier (int)
      |  ||  |  definition: type (builtin), type=int
      |  ||  +> [7, 8] Identifier (a)
      |  ||  |  definition: variable defined at [7, 8], type=int
      |  ||  ‘> [7, 8] NoInitialization
      |  []> [8, 9] DeclVar
      |      +> [8, 4] Identifier (int)
      |      |  definition: type (builtin), type=int
      |      +> [8, 9] Identifier (tab_entier)
      |      |  definition: variable defined at [8, 9], type=1D int
      |      ‘> [8, 20] Initialization
      |         ‘> [8, 20] TableInt [dimension=1]
      |            type: 1D int
      ‘> [5, 0] ListInst [List with 1 elements]
         []> [9, 4] Print
             ‘> [9, 10] ListExpr [List with 3 elements]
                []> [9, 10] ListElement
                ||  type: int
                ||  +> [9, 10] Identifier (tab_entier)
                ||  |  definition: variable defined at [8, 9], type=1D int
                ||  ‘> ListExpr [List with 1 elements]
                ||     []> [9, 21] Int (0)
                ||         type: int
                []> [9, 24] StringLiteral ( )
                ||  type: string
                []> [9, 28] ListElement
                    type: int
                    +> [9, 28] Identifier (tab_entier)
                    |  definition: variable defined at [8, 9], type=1D int
                    ‘> ListExpr [List with 1 elements]
                       []> [9, 39] Int (1)
                           type: int
```

5 Analyse de Géneration de code

5.1 Règles sur les indices

Les indices utilisés pour accéder aux éléments d'un tableau doivent obligatoirement être des entiers positifs ou nuls. En effet, dans le contexte de la gestion des tableaux, les indices commencent toujours à zéro et s'étendent jusqu'à la taille du tableau moins un. Toute tentative d'utilisation d'un indice négatif ou d'un type non entier (comme un flottant ou une chaîne de caractères) entraîne une erreur.car cela ne respecte pas les règles de la grammaire et du typage. De plus, l'utilisation d'indices dépassant la taille du tableau est strictement interdite, car cela provoquerait une erreur d'exécution, souvent appelée "ArrayIndexOutOfBoundsException" dans des langages comme Java. Il est donc impératif que les indices soient non seulement positifs, mais également compris dans les limites du tableau.

Exemple :

- `int[ ] myArray = new int[5];` valide
- `myArray[0] = 10;` valide
- `myArray[4] = 20;` valide
- `myArray[-1] = 15;` Erreur : indice négatif
- `myArray[5] = 25;` Erreur : dépassement de la taille du tableau
- `myArray[2.5] = 30;` Erreur Contextuelle :l'index doit étre de type int.

5.2 Règles sur les opérations : Division et modulo

Lors de l'exécution des opérations de division et de modulo, il est essentiel de vérifier que le diviseur n'est pas nul, car la division par zéro est une opération indéfinie en mathématiques et provoquerait une erreur d'exécution dans le programme.

La division consiste à diviser un nombre par un autre. Si le diviseur est égal à zéro, cela entraîne une erreur de division par zéro, ce qui peut entraîner un comportement imprévisible ou un plantage du programme. Il est donc crucial de s'assurer que le diviseur est non nul avant d'effectuer la division.

```
result = numerator / denominator;
```

Exemple :

```
int []tab = [10]; int y = 0;result = tab[0] / y;
```
(Erreur : division par zéro)

Dans cet exemple, la division de x par y entraînera une erreur car $y = 0$. Il faut vérifier que y n'est pas nul avant de procéder à la division.

L'opération de modulo (%) retourne le reste de la division entière de deux nombres. Comme pour la division, le modulo par zéro n'est pas défini et entraîne une erreur similaire.

```
result = numerator % denominator;
```

Exemple :

```
int []tab = [10]; int y = 0;result = tab % y;
```
(Erreur : modulo par zéro)

Dans cet exemple, le calcul de 10%0 entraînera une erreur, car le diviseur est nul. Encore une fois, il est nécessaire de vérifier que $y \neq 0$ avant d'effectuer cette opération.

Cette validation prévient les erreurs d'exécution et permet d'assurer que les résultats des opérations sont bien définis.

5.3 Règles sur l'allocation de mémoire (pile)

Dans notre système, la pile est limitée à 10 000 mots, ce qui impose des restrictions sur la quantité de mémoire qui peut être utilisée pour l'allocation des variables locales et les appels de fonctions. Chaque fois qu'une fonction est appelée, de la mémoire est allouée sur la pile pour stocker les variables locales et les informations de contrôle d'exécution (comme les adresses de retour). Si la pile dépasse cette limite, cela entraînera un dépassement de la pile, provoquant un plantage du programme et un message d'erreur débordement de la pile.

`Taille maximale de la pile:` Le programme utilise une pile limitée à 10 000 mots. Chaque déclaration de variable locale ou chaque appel de fonction augmente la taille de la pile. Il est donc crucial de s'assurer que l'ensemble des allocations ne dépasse pas cette taille pour éviter un dépassement de pile. Par exemple, l'utilisation de variables locales ou de tableaux locaux dans une fonction augmente la consommation de la pile.

`Variables locales et tableaux:`

Les variables locales, y compris les tableaux, sont allouées sur la pile. Par exemple, une déclaration de tableau comme :

```
int tab[500];  // 500 mots de memoire sur la pile
int var = 10;  // 1 mot de memoire sur la pile
int tab[10][10];  // 100 mots de memoire sur la pile
```

5.4 Conception

Lors de la génération de code pour les tableaux, une attention particulière doit être portée à la manière dont les données sont stockées dans la mémoire. Afin de maximiser l'utilisation de l'espace mémoire disponible, les éléments des tableaux sont stockés de manière contiguë dans le tas. Cela signifie que chaque élément du tableau occupe une case mémoire consécutive, ce qui permet un accès rapide et efficace aux éléments à l'aide d'index.

Dans le cas des tableaux unidimensionnels, chaque élément est alloué de manière séquentielle, ce qui signifie que si un tableau 'tab' de taille 'n' est déclaré, il occupera 'n' cases contiguës dans le tas. Cette approche est simple et efficace pour des tableaux classiques où chaque élément est de taille fixe (par exemple, un tableau d'entiers ou de flottants).

Cependant, dans le cas des tableaux multidimensionnels, comme les matrices, la gestion de la mémoire devient plus complexe. Une matrice de taille $m \times n$ (ou $m \times n \times z$, etc.) est également allouée dans des cases contiguës, mais la mémoire est "aplatissante" pour tenir compte de toutes les dimensions. Par exemple, une matrice $m \times n$ sera allouée dans $m \times n$ cases contiguës. Cela signifie que, bien que l'accès aux éléments de la matrice soit basé sur deux indices (par exemple, $mat[i][j]$), la mémoire est en fait organisée sous forme linéaire, en stockant tous les éléments successivement dans la mémoire. Pour une matrice $m \times n$, les éléments sont accessibles en suivant la règle suivante : l'élément situé à l'indice (i, j) sera placé à l'emplacement linéaire $i \times n + j$, ce qui permet d'utiliser une simple opération d'indexation pour accéder aux éléments de la matrice.

Cette approche garantit une gestion efficace de la mémoire, tout en assurant que l'accès aux éléments du tableau ou de la matrice est effectué de manière rapide et fiable.

6 Cahier des Charges : Bibliothèque

6.1 Règles sur les opérations

— **Somme/Différence de deux matrices :** Il faut que les matrices possèdent la même dimension.

— **Multiplication :** Pour que l'opération de multiplication soit possible entre deux matrices, leurs dimensions doivent respecter la règle suivante : le nombre de colonnes de la première matrice doit être égal au nombre de lignes de la seconde matrice. Par exemple, si la première matrice est de dimensions $A \in \mathbb{R}^{m \times n}$ et la seconde matrice est $B \in \mathbb{R}^{n \times p}$, alors leur produit $C = A \times B$ est défini, et la matrice résultante C sera de dimensions $m \times p$. Si cette condition n'est pas respectée, la multiplication n'est pas possible.
De plus, chaque élément c_{ij} de la matrice résultante C est calculé comme suit :

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

où a_{ik} représente l'élément de la i-ème ligne et de la k-ème colonne de la matrice A, et b_{kj} représente l'élément de la k-ème ligne et de la j-ème colonne de la matrice B. Ainsi, chaque case de C est obtenue en effectuant le produit scalaire entre la i-ème ligne de A et la j-ème colonne de B.

6.2 MatrixFloat

La bibliothèque MatrixFloat est un ensemble de fonctions pour la manipulation des matrices de nombres flottants, conçu pour offrir des opérations basiques et avancées sur les matrices.
Fonctionnalités principales :

— 1. Accès aux éléments

```
float getElement(float[][] matrix, int row, int col)
```

Permet d'accéder à un élément spécifique de la matrice.
Utilité : Facilite la lecture des éléments sans devoir manipuler directement les indices.

— 2. Affichage

```
void printMatrix(float[][] matrix, int rows, int cols)
```

Affiche le contenu de la matrice dans un format lisible.
Utilité : Outil de débogage ou de visualisation des matrices.

— 3. Opérations basiques **Addition et soustraction**

```
float[][] add(float[][] A, int rowsA, int colsA, float[][] B, int rowsB
    , int colsB)
float[][] subtract(float[][] A, int rowsA, int colsA, float[][] B, int
    rowsB, int colsB)
```

Effectuent respectivement l'addition et la soustraction élément par élément.

Multiplication

```
float[][] multiply(float[][] A, int rowsA, int colsA, float[][] B, int
    rowsB, int colsB)
```

Réalise la multiplication matricielle, avec vérification préalable de la compatibilité des dimensions.

— 4. Propriétés avancées **Transpose**

```
float[][] transpose(float[][] matrix, int rows, int cols)
```

Renvoie une matrice transposée en inversant lignes et colonnes.
Utilité : Essentiel pour de nombreuses applications mathématiques et statistiques.

Trace

```
float trace(float[][] matrix, int rows, int cols)
```

Calcule la somme des éléments diagonaux de la matrice.
Utilité : Permet d'obtenir rapidement une propriété fondamentale d'une matrice carrée.

Déterminant

```
float determinant(float[][] matrix, int size)
```

Calcule le déterminant d'une matrice carrée
Utilité : Indique si une matrice est inversible

6.3 Bibliothèque : ListFloat

La classe générique **ListFloat** permet de gérer dynamiquement des listes de nombres flottants. Elle fournit des fonctionnalités avancées, telles que la redimension des tableaux, le tri rapide et des méthodes de recherche efficaces.

Fonctionnalités principales

— 1. Initialisation

```
void init()
```

Initialise la liste avec une capacité par défaut de 10. **Utilité** : Prépare les structures nécessaires pour stocker les données.

— 2. Ajout d'éléments

```
void add(float value)
```

Ajoute un élément à la liste, avec redimension automatique si nécessaire. **Utilité** : Permet de gérer dynamiquement la taille de la liste

— 3. Accès aux éléments

```
float get(int index)
```

Renvoie l'élément à l'index donné. Affiche un message d'erreur si l'index est hors limites.
Utilité : Facilite l'accès aux données stockées.

— 4. Suppression d'un élément

```
void remove(int index)
```

Supprime l'élément à un index spécifique et ajuste les autres éléments. **Utilité** : Permet de gérer la liste de manière dynamique.

— 5. Taille actuelle de la liste

```
int getSize()
```

Renvoie le nombre d'éléments actuellement stockés dans la liste. **Utilité** : Fournit une méthode rapide pour connaître la taille de la liste.

— 6. Tri rapide (Quicksort)

```
void quickSort()
void quickSortRecursive(int low, int high)
int partition(int low, int high)
```

Implémente l'algorithme Quicksort pour trier les éléments de la liste. **Utilité** : Offre un tri performant pour des

— 7. Recherche binaire

```
int binarySearch(float target)
```

Recherche un élément dans une liste triée en utilisant l'algorithme de recherche binaire. **Utilité** : Permet une recherche rapide grâce à la division du problème.

— 8. Recherche linéaire

```
int linearSearch(float target)
```

Recherche un élément dans la liste, élément par élément. **Utilité** : Utilisée lorsque la liste n'est pas triée.

Détails des algorithmes

Tri rapide (Quicksort) Le tri rapide utilise une approche récursive pour diviser et trier la liste :

— Sélection d'un pivot.
— Réorganisation des éléments pour placer les plus petits avant le pivot et les plus grands après.
— Application récursive à chaque sous-liste.

Avantage : Complexité moyenne $O(n \log n)$.

Recherche binaire La recherche binaire divise la liste triée en deux à chaque itération :

— Vérification de la position centrale.
— Réduction de la recherche à la moitié pertinente.

Avantage : Complexité $O(\log n)$.

Exemples d'utilisation

et tri

```
ListFloat list;
list.init();
list.add(5.0);
list.add(2.0);
list.add(9.0);
list.quickSort();
list.print(); // Affiche : 2.0 5.0 9.0
```

— Recherche d'un élément

```
int index = list.binarySearch(5.0);
if (index != -1) {
    print("  lment   trouv     l'index : " + index);
} else {
    print("  lment   non trouv .");
}
```

7 Limitation

En raison de la contrainte de temps imposée par le projet, nous n'avons pas pu effectuer une large gamme de tests sur notre extension. Cette limitation pourrait entraîner la présence de bugs non détectés, notamment dans des cas particuliers ou des scénarios non envisagés.

Bien que nous ayons conçu et validé les fonctionnalités principales de notre extension, un test approfondi, incluant des tests de charge, des tests sur des cas limites, ou encore des scénarios d'utilisation complexes, reste indispensable pour garantir une robustesse totale.

Nous recommandons, dans les futures itérations, d'investir du temps dans :

— L'ajout de cas de tests supplémentaires pour couvrir davantage de scénarios.

— L'amélioration de la couverture de code en intégrant des tests unitaires et fonctionnels plus exhaustifs.

— L'automatisation complète des tests pour détecter rapidement les régressions lors des ajouts de nouvelles fonctionnalités.

En raison de la contrainte de temps, nous n'avons pas pu résoudre le problème suivant :

```
    void allocate(int col, int row) {
        println(col, " ", row);
        list = new int[col][row];
    }
```

Il est possible de créer un tableau avec une taille fixe, comme `new int[2]`, mais il n'est pas possible de créer un tableau avec une taille dynamique, comme `new int[col]`, lorsque `col` est un paramètre de la méthode.

Le problème réside dans le fait que `col`, en tant que paramètre de méthode, n'existe plus une fois que l'exécution sort de la méthode. Cela entraîne une limitation dans la gestion dynamique des tableaux à l'intérieur de la méthode.

8 Validation

Nous avons créé un nouveau répertoire `extension`, qui est organisé en plusieurs sous-répertoires spécifiques : `context`, `synt`, et `codegen`. Chacun de ces sous-répertoires est lui-même divisé en deux catégories : `valides` et `invalides`. Dans chaque catégorie, nous avons ajouté des tests correspondant à chaque partie, afin de vérifier le bon fonctionnement de notre extension.

De plus, nous avons développé un script shell permettant d'exécuter automatiquement ces tests. Ce script facilite la vérification des résultats des tests et assure une couverture complète de notre extension. Enfin, en ajoutant ces tests dans le fichier `pom.xml`, nous avons amélioré la couverture globale des tests, contribuant à une validation plus robuste de notre solution.

9 Conclusion

En s'inspirant des standards établis par des langages comme Java et C, nous avons introduit une gestion efficace des tableaux multidimensionnels, permettant d'effectuer des opérations avancées telles que le produit matriciel, le calcul de déterminants, et des manipulations dynamiques des listes.

Cette extension apporte des fonctionnalités adaptées aux besoins des développeurs en renforçant la gestion des types, l'efficacité des algorithmes, et en assurant une syntaxe intuitive pour les utilisateurs de Deca.

Enfin, l'extension TAB illustre l'importance d'une conception rigoureuse, intégrant des vérifications contextuelles et des règles syntaxiques claires, pour garantir une utilisation fiable .

10 Bibliographie

— **Documentation C++ Reference** :
https://en.cppreference.com/w/

— **Java Language Specification (SE 17)** :
https://docs.oracle.com/javase/specs/jls/se17/html/jls-15.html

— **Java Language Specification (SE 20)** :
https://docs.oracle.com/javase/specs/jls/se20/html/jls-10.html

— **Java BNF Grammar** :
https://cs.au.dk/ amoeller/RegAut/JavaBNF.html