Documentation - Extension TAB

L'extension que nous avons choisi est l'extension TAB, qui consistait à étendre le langage Deca avec les tableaux et proposer une librairie de calcul matriciel. - Documentation - Extension TAB - 1. Documentation utilisateur - 1.1. Utilisation des tableaux - 1.1.1. Déclaration - 1.1.2. Initialisation - Utilisation de `new` - Initilisation à partir d'un tableau - 1.1.3. Réaffectation - 1.1.4. Accession aux éléments - 1.2. Utilisation de la librairie : MathArray.decah - 1.5.1. Les méthodes de la librairie - 1.5.2. Utilisation de la librairie - 2. Choix de conception - 2.1. Implémentation du type primitif tableau/matrice pour le langage Deca - 2.1.1. Génération de l'arbre - 2.1.2. Génération du code assembleur - Allocation de la mémoire - Tableaux à 1 dimension - Tableaux à 2 dimensions (matrices) - Réassignation - 2.1.3 Initilisation en mémoire - 2.1.4 Accession à un élément - 2.1.5 Gestion des erreurs - 2.1.6 Méthode `length` - 2.1.7 Exemples de code - 2.2. Bibliothèque de calcul : *MathArray.decah* - 3. Spécifications - 3.1 Lexer et Parser - 3.2. Règles contextuelles - 3.3. Génération du code - 4. Analyse bibliographique - 4.1 Recherches - 4.2 Fonctionnalités à implémenter - 4.2.1 Basiques - 4.2.2 Avancées - 4.2.3 Optimisations - 4.3 Problèmes soulevés - 4.3.1 Utilisation mémoire - 4.3.2 Implémentation assembleur - 5. Méthode de validation - 5.1 Base de tests - 5.2 Utilisation de Jacoco - 6. Résulats de la validation de l'extension - 7. Amélioration # 1. Documentation utilisateur

Tout d'abord il faut savoir que cette extension n'affecte en aucun cas l'utilisation classique du langage Deca. Elle étend simplement les possibilités. Ici, comme pour un langage de programmation comme le Java, vous avez la possibilité d'utiliser des tableaux. Ce sont des objets stockant plusieurs éléments et qui permettent d'accéder aux éléments stockés à travers leur indice (place dans le tableau).

1.1. Utilisation des tableaux

1.1.1. Déclaration

Le type d'un tableau admet pour syntaxe : `typeTableau[]`. Pour déclarer un tableau, il faut utiliser la syntaxe suivante :

```
typeTableau[] nomDuTableau;
```

Le `typeTableau` peut être lui même le type d'un tableau, on obtient alors un tableau à 2 dimensions, une matrice. Par exemple pour la déclaration d'un tableau à 2 dimensions d'entiers, la commande a utilisé est la suivante :

```
int[][] tabInt;
```

1.1.2. Initialisation

Utilisation de `new` Il existe deux manières différentes d'initialiser un tableau.

La première utilise `new` pour initialiser le tableau. Elle permet de renseigner la taille du tableau et de créer ce tableau en mémoire.

Exemple :

```
int[] tab = new int[5];

// Syntaxe d'initialisation :
// typeTableau[] nomTableau = new typeTableau[taille];
```

Pour un tableau de dimension 2, l'initialisation est similaire :

```
int[][] tab = new int[][5]; //int[] est le type des sous éléments
```

La taille donnée en argument doit être un entier positif. Dans le cas contraire, une erreur est levée.

En utilisant cette méthode, l'ensemble des éléments du tableau sont initialisés à 0 en mémoire. Donc le code suivant affichera 0 sur la sortie standard :

```
int[] tab = new int[5];
print(tab[1]); //Affiche 0 sur la sortie standard
```

Initilisation à partir d'un tableau Une autre méthode pour initialisation un tableau est d'utiliser un autre tableau. Cela peut être un tableau déjà initialisé au préalable ou bien un tableau instantané. Les tableaux instantanés sont principalement utilisés pour l'initialisation des tableaux. La syntaxe des tableaux instantanés est la suivante :

```
{elem1, elem2, ..., elemn} // Tableau instantané de n éléments
```

Afin d'initialiser un tableau à partir d'un tableau instantané, la syntaxe a utilisé est la suivante :

```
typeTableau[] nomTableau = {elem1, elem2, ..., elemn};

//Par exemple pour initialiser un tableau d'entier
int[] tabInt = {2, 5, 8, 9};

//Pour un tableau de flottants
float[] tabFloat = {5.1, 45.6, 984.1};

//Pour une matrice d'entiers
int[][] matrixInt = {{1,2}, {3,5}, {5,74}};
```

À noter que les types des tableaux doivent correspondre pour pouvoir initialiser le tableau, sinon une erreur est levée. Exemple :

```
int[] tabInt = {1.0, 2.0} //Doit renvoyer une erreur
```

1.1.3. Réaffectation

Il est possible de réaffecter un tableau après qu'il ait été initialisé. Ce dernier est alors écrasé pour être remplacé par le nouveau tableau. Exemple :

```
int[] tab = new int[2]; //tab est un tableau de taille 2 où tous les éléments sont nuls

tab = {2, 4, 8, 7}; //Les éléments de tab sont écrasés et remplacés par le nouveau tableau
```

La taille du tableau est alors modifiée, ainsi que les éléemnts contenus. Toutefois, le nouveau tableau doit respecter le typage et être du même type que le tableau affecté. Dans le cas contraire, une erreur est levée. ### 1.1.4. Accession aux éléments

Pour accéder à un élement du tableau, cela se fait en utilisant un indice pour parcourir le tableau. L'indice est un entier compris entre 0 et la taille-1 du tableau , 0 représentant le premier élément du tableau et taille-1 l'indice du dernier élément.

La syntaxe pour accéder à un élément d'un tableau est la suivante :

```
nomTableau[indice];
```

Cela renvoie l'élément stocké à l'indice dans le tableau.

Dans le cas des tableaux à 1 dimension, le résultat renvoyé est directement exploitable par le programme de base. Exemple :

```
int[] tab = {2, 4, 8, 7};
int x = tab[0] + tab[2];
print(x); //Affiche 10 sur la sortie standard
print(tab[3]); //Affiche 7 sur la sortie standard.
```

Dans le cas des tableaux à 2 dimensions, l'accession dépend du niveau auquel il est effectué. La commande `tab[i]` renverra le sous tableau d'indice i qui est stocké dans le tableau. Pour accéder aux éléments des sous tableaux, la syntaxe a utilisé est la suivante :

```
nomTableau[indice1][indice2];
```

`indice1` correspond à l'indice du sous tableau dans le tableau et `indice2` correspond à l'indice de l'élément à accéder dans le sous tableau. Par exemple, pour une matrice d'entiers :

```
int[][] matInt = {{1, 2}, {4, 3}};
print(matInt[0][1]); //Affiche 2 sur la sortie standard
```

L'indice `indice1` doit être un entier compris entre 0 et taille-1 du tableau. L'indice `indice2` doit être un entier compris entre 0 et (taille du sous-tableau)-1. Si cela n'est pas le cas, une erreur `Index out of range` est levée.

1.2. Utilisation de la librairie : MathArray.decah

Pour utiliser les méthodes de la librairies, vous devez d'abord l'inclure en début de fichier:

```
#include "MathArray.decah"
```

Ensuite pour pouvoir utiliser les méthodes vous devez instancier un objet de la classe:

```
MathArray Calculator = new MathArray();
```

Enfin à travers l'objet instancié, vous pouvez appelé les méthodes :

```
// Les matrices A et B ont été initialisés precédemment
int[][] matrix = Calculator.multiplyInt(A, B)
```

2. Choix de conception

Le défi pour nous était d'étendre la langage Deca pour permettre à un développeur d'utiliser des tableaux, des matrices, mais aussi de proposer une librairie dans la bibliothèque standard qui permettrait d'effectuer des calculs sur ces objets.

Ainsi, la première étape consistait à implémenter le type primitif tableau/matrice pour le langage. Par la suite, il suffirait pour la librairie d'écrire une classe Deca contenant des méthodes effectuant des algorithmes comme par exemple la multiplication de deux matrices. Bien-sûr cette librairie dépend de l'aspect fonctionnelle du développement du langage Deca complet, car la librairie est simplement une classe.

2.1. Implémentation du type primitif tableau/matrice pour le langage Deca

2.1.1. Génération de l'arbre

Création d'une classe **Table** qui hérite de *AbstractExpr* qui est instancié quand on initialise un tableau/matrice. Par exemple :

```
int[] tableau = {1, 2, 4, 2};
```

Ici on a un arbre avec une déclaration contenant : deux Identifier et un Table (l'initialisation) :

```
`> [1, 0] Main
      +> ListDeclVar [List with 1 elements]
      |  []> [2, 10] DeclVar
      |      +> [2, 4] Identifier (int[])
      |      +> [2, 10] Identifier (tableau)
      |      `> [2, 20] Initialization
      |         `> [2, 20] Table
```

```
|            +> ListExpr [List with 3 elements]
|            |  []> [2, 21] Int (1)
|            |  []> [2, 24] Int (6)
|            |  []> [2, 27] Int (3)
|            `> Int (3)
`> ListInst [List with 0 elements]
```

Création d'une classe **ElementTable** qui hérite de *AbstractIdentifier*. C'est un Identifier un peu spécial, qui contient de l'information en plus. Cette classe contient le symbol en lui même, l'Abstract Identifier du tableau "parent", et l'indice passé à l'intérieur des crochets.

Par exemple, si après la déclaration du tableau précédent, on ajoute :

```
tableau[0] = 5;
```

On obtient dans l'arbre syntaxique:

```
[]> [3, 15] Assign
             +> [3, 4] ElementTable (tableau[2])
             |  +> [3, 4] Identifier (tableau)
             |  `> [3, 12] Int (2)
             `> [3, 17] Int (2)
```

Pour gérer l'étape contextuel, nous avons dû créer une classe **TableType.java** qui hérite de *Type.java*, un peu différente des classes de type car elle contient un attribut sous-type qui est lui même un objet de type *Type*. Voici ses deux constructeurs :

```
public TableType(SymbolTable.Symbol name) {
        super(name);
        this.subType = null;

    }

    public TableType(SymbolTable.Symbol name, Type subType) {
        super(name);
        this.subType = subType;
    }
```

Il s'agissait ensuite de déclarer les types dans l'environnement de type. Pour les types classiques *int[]*, *int[][]* et idem avec *float* et *boolean*, ils sont déclarés dans *DecacCompiler.java*. Ensuite pour pouvoir créer des tableaux d'objet de classe défini par l'utilisateur, les types sont déclarés à la fin de la passe 1, dans la méthode *verifyClass* de *DeclClass.java*.

Pour ce qui est des matrices, un exemple permettra de comprendre le côté récursif de l'implémentation. Exemple: Modification d'une valeur d'une matrice de flottants

```
{
```

```
    float[][] tableau = {{1.5, 6.2}, {4.5, 1.2}};
    tableau[0][1] = 10.3;
}
```

L'arbre contextuelle obtenue est :

```
`> [1, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [1, 0] Main
      +> ListDeclVar [List with 1 elements]
      |  []> [2, 14] DeclVar
      |      +> [2, 4] Identifier (float[][])
      |      |  definition: type (builtin), type=float[][]
      |      +> [2, 14] Identifier (tableau)
      |      |  definition: variable defined at [2, 14], type=float[][]
      |      `> [2, 24] Initialization
      |         `> [2, 24] Table
      |            type: float[][]
      |            +> ListExpr [List with 2 elements]
      |            |  []> [2, 25] Table
      |            |  ||  type: float[]
      |            |  ||  +> ListExpr [List with 2 elements]
      |            |  ||  |  []> [2, 26] Float (1.5)
      |            |  ||  |  ||  type: float
      |            |  ||  |  []> [2, 31] Float (6.2)
      |            |  ||  |      type: float
      |            |  ||  `> Int (2)
      |            |  ||     type: int
      |            |  []> [2, 37] Table
      |            |      type: float[]
      |            |      +> ListExpr [List with 2 elements]
      |            |      |  []> [2, 38] Float (4.5)
      |            |      |  ||  type: float
      |            |      |  []> [2, 43] Float (1.2)
      |            |      |      type: float
      |            |      `> Int (2)
      |            |         type: int
      |            `> Int (2)
      |               type: int
      `> ListInst [List with 1 elements]
         []> [3, 18] Assign
             type: float
             +> [3, 4] ElementTable (tableau[0][1])
             |  definition: variable defined at [3, 4], type=float
             |  +> ElementTable (tableau[0])
             |  |  definition: variable defined at null, type=float[]
             |  |  +> [3, 4] Identifier (tableau)
```

```
|  |  |  definition: variable defined at [2, 14], type=float[][]
|  |  `> [3, 12] Int (0)
|  |     type: int
|  `> [3, 15] Int (1)
|     type: int
`> [3, 20] Float (10.3)
   type: float
```

Affectation et initialisation à la fois pour les sous-tableaux : une difficulté

2.1.2. Génération du code assembleur

Allocation de la mémoire Les tableaux et matrices peuvent représenter une grande quantité d'éléments à stocker en mémoire. Nous avons donc choisi comme choix de conception de stocker les tableaux et matrices non pas dans la pile mais dans le tas. En effet, ce dernier possède en général un espace de stockage plus grand que la pile, mais qui induit un management de la mémoire plus contraignant que pour la pile (nous en reparlerons par la suite).

Tableaux à 1 dimension À la déclaration d'un tableau par les commandes `typeTab[] tab = new typeTab[X]` ou `typeTab[] tab = {a,b,c}`, à partir de la taille précisé `X` ou de la taille récupérée du tableau initialisateur, un espace mémoire dans le tas est réservé pour le tableau `tab` à l'éxécution grace à la commande assembleur `NEW`. Cet espace correspond au nombre de mots nécessaire pour stocker chaque élément, ainsi qu'un mot supplémentaire pour stocker la taille du tableau. L'adresse du premier mot du bloc alloué est ensuite stocké dans la pile.

Exemple d'allocation mémoire d'un tableau à 1 dimension de taille 3 au niveau assembleur:

```
LOAD #3, R4
LOAD #1, R3
ADD R4, R3
NEW R3, R3
STORE R4, 0(R3)
```

Tableaux à 2 dimensions (matrices) Pour les éléments de types `typeTab[][]`, l'allocation en mémoire s'effectue en deux temps : une première comme pour un tableau à 1 dimension, puis, lorsque l'on initialise les sous éléments, une deuxième allocation similaire à la précédente où, au lieu de stocker les adresses des sous tableaux dans la pile, ces dernières sont stockées dans le tas, dans les mots du premier tableau initialisé. On obtient donc un tableau d'adresse de références à des tableaux de sous éléments, dont la référence se trouve dans la pile.

Réassignation Lors de la réassignation d'un tableau déjà existant, un nouveau tableau est créé en mémoire et la référence est modifiée pour pointer vers ce tableau.

2.1.3 Initilisation en mémoire

Avant d'initialiser les éléments, la taille du tableau déterminé est inséré comme premier élément du bloc alloué dans le tas. Une fois l'allocation en mémoire effectuée, le tableau est initialisé selon les éléments décrits en 1.1.2. Cela est effectué au niveau assembleur par une boucle `while` sur les éléments à initialiser. On rappelle que pour une initialisation avec la commande `tab = new typeTab[X]`, tous les éléments du tableau sont initialisés à 0.

2.1.4 Accession à un élément

Pour accéder à un élément `tab[x]`, on stocke la valeur de l'indice additionée de 1 dans le registre `RY` et on retrouve grace à l'identifiant `tab` l'adresse du tableau qui est stockée dans la pile. Celle-ci nous donne l'adresse du premier élément du bloc que l'on stocke dans un registre `RX`. Pour accéder à l'élément d'indice `x` et le stocker dans le registre `RZ`, on utilise alors la commande assembleur `LOAD 0(RX, RY), RZ` compréhensible par **ima**. Pour cela, une classe **RegisterOffsetDouble** a été rajouté qui dérive de **RegisterOffset** permettant de réaliser cette instruction.

```
public class RegisterOffsetDouble extends RegisterOffset {

    private final Register register2;

    public RegisterOffsetDouble(int offset, Register register, Register register2){
        super(offset, register);
        this.register2 = register2;
    }

    public Register getRegister2(){
        return this.register2;
    }

    @Override
    public String toString() {
        return this.getOffset() + "(" + this.getRegister() + ", " + this.getRegister2()  +
    }
}
```

Pour les matrices, l'accession est similaire mais se déroule juste au niveau du sous tableau.

2.1.5 Gestion des erreurs

L'accession à un élément avec un indice incorrect (supérieur à la taille ou négatif) est traité au niveau assembleur en deux étapes : une première étape pour vérifier que l'indice est positif, une deuxième pour vérifier que l'indice est bien strictement inférieur à la taille du tableau. Si ces conditions ne sont pas vérifiées, cela renvoie l'erreur à l'éxécution : `Erreur : Index out of range`.

2.1.6 Méthode `length`

Afin de permettre de récupérer la taille d'un tableau en temps constant, la taille du tableau est directement implémenté dans les données stockées dans le tas. En effet, le premier mot du bloc alloué à un tableau stocke la longueur du tableau donnée à son initialisation. C'est à partir de la taille indiquée à cet emplacement que l'on fait les comparaisons d'indice pour l'erreur **index out of range**. ### 2.1.7 Exemples de code

Pour le morceau de code suivant en Deca :

```
{
    int[][] tab = {{1,2},{4,5}};
    tab[1][1];
}
```

Le code assembleur généré est (les parties **exceptions** et **initialisation** ne sont pas représentées ici) :

```
; Main program
DebutMain:
; Beginning of main declarations
        LOAD #2, R4
        LOAD #1, R3
        ADD R4, R3
        NEW R3, R3
        STORE R4, 0(R3)
        LOAD #2, R5
        LOAD #1, R4
        ADD R5, R4
        NEW R4, R4
        STORE R5, 0(R4)
        LOAD #1, R5
        STORE R5, 1(R4)
        LOAD #2, R5
        STORE R5, 2(R4)
        STORE R4, 1(R3)
        LOAD #2, R5
        LOAD #1, R4
        ADD R5, R4
```

```
        NEW R4, R4
        STORE R5, 0(R4)
        LOAD #4, R5
        STORE R5, 1(R4)
        LOAD #5, R5
        STORE R5, 2(R4)
        STORE R4, 2(R3)
        STORE R3, 1(LB)
        ADDSP #1
; Beginning of main instructions:
        LOAD 1(LB), R3
        LOAD #1, R5
        CMP #0, R5
        BLT io_error_index_out_of_range
        CMP 0(R3), R5
        BGE io_error_index_out_of_range
        ADD #1, R5
        LOAD 0(R3, R5), R3
        LOAD #1, R4
        CMP #0, R4
        BLT io_error_index_out_of_range
        CMP 0(R3), R4
        BGE io_error_index_out_of_range
        ADD #1, R4
        LOAD 0(R3, R4), R3
        HALT
; end main program
```

2.2. Bibliothèque de calcul : *MathArray.decah*

Comme expliqué précedemment, cette librairie est simplement une classe. Ainsi elle contient des méthodes de calcul, qui prennent en paramètre un ou deux objets de type matrice/tableau, et renvoie un objet de type matrice/tableau. Par exemple, la méthode qui permet de retourner la matrice résultante de la multiplication de deux matrices d'entiers :

```
// Return multiplication of two int matrices
    int[][] multiplyInt(int[][] A, int[][] B) {
        int nbLinesA = length(A);
        int nbColumnsA = length(A[0]);
        int nbLinesB = length(B);
        int nbColumnsB = length(B[0]);
        int i = 0;
        int j = 0;
        int k = 0;
        int[][] matrix = new int[][nbLinesA];
```

```
        if (nbColumnsA != nbLinesB) {
            assem();
        }
        while (i < nbLinesA) {
            matrix[i] = new int[nbColumnsB];
            while (j < nbColumnsB) {
                while (k < nbColumnsA) {
                    matrix[i][j] = matrix[i][j] + A[i][k]*B[k][j];
                    k = k + 1;
                }
                j = j + 1;
            }
            i = i + 1;
        }
        return matrix;
    }
```

Dans le cas où l'utilisateur passe en paramètre deux tableaux/matrices de taille différentes, ou alors dans le cas de la multiplication si le nombre de colonnes de la première matrice n'est pas égale au nombre de ligne de la deuxième, une ligne de code assembleur et générer grâce à *asm* et permet de sauter à une erreur. Comme vous pouvez le voir dans l'exemple précédent, nous appelons la méthode *assem()* à la première instruction dans le cas décrit précedemment. Voici le code de cette méthode qui est présente dans la librairie:

```
void assem() asm (
    "BRA io_incorrect_sizes"
);
```

3. Spécifications

Voici les règles qui ont permis d'obtenir les résultats précédents.

3.1 Lexer et Parser

Ajout de 3 tokens dans le Lexer :

```
OBRACKET: '['
CBRACKET: ']'
LENGHT: 'length'
```

Et modification du token IDENT :

```
IDENT : (LETTRE | '_' | '$') (LETTRE | '_' | '$' | CHIFFRE )*
    (OBRACKET CBRACKET)? (OBRACKET CBRACKET)?;
```

dans le but d'autoriser les types xxx[] et xxx[][];

Ajout des règles suivantes dans le Parser :

```
unary_expr -> LENGTH OPARENT primary_expr CPARENT
primary_expr -> OBRACE list_expr CBRACE {
            assert($args.tree != null);
                $tree = new Table($args.tree);
                setLocation($tree, $OBRACE);
                }
        | NEW type OBRACKET expr CBRACKET {
            assert($type.tree != null);
                assert($expr.tree != null);
                $tree = new Table($type.tree, $expr.tree);
                setLocation($tree, $NEW);
        }
```

3.2. Règles contextuelles

Pour la règle

`primary_expr -> NEW type OBRACKET expr CBRACKET`

Le type de expr doit être *int*. Ainsi dans le *verifyExpr* de Table on lance une erreur contextuelle si le type retourné par le *verifyExpr* de expr n'est pas *int*.

Pour la règle :

`primary_expr -> OBRACE list_expr CBRACE`

Le type des expressions de list_expr doivent être identiques. De plus, le type des expressions doit correspondre au sous-type du tableau.

3.3. Génération du code

La génération du code assembleur se fait de la même manière que pour le reste du projet, avec la commande `decac` et avec les mêmes options disponibles. Toutefois, il est conseillé pour l'utilisation de cette extension d'augmenter la taille du tas à l'aide de l'option `-p nnn` lors de l'appel de **ima**.

4. Analyse bibliographique

4.1 Recherches

Deca étant un sous langage de Java, nous avons dans un premier temps fait des recherches sur la documentation Java concernant les tableaux. Pour construire notre syntaxe et l'ensemble des fonctionnalités de base, nous sommes donc inspirés principalement de pages de documentation (ici et ici). C'est à partir de ces recherches que nous avons pu construire la manière de réaliser l'extension et de déterminer les différentes fonctionnalités a implémenter. Nous avons donc déterminer des méthodes précises de notre bibliothèque.

Pour les choix de conception, et notamment des choix de l'utilisation de la pile ou du tas, nous avons utiliser une documentation décrivant les différents avantages et inconvénients de chaque espace mémoire, ce qui nous a permis de faire nos choix de conception.

4.2 Fonctionnalités à implémenter

Nous avons classé les fonctionnalités en 3 catégories, selon leur importance : ### 4.2.1 Basiques

- L'addition et la soustraction ;
- La multiplication de deux matrices ;
- La multiplication par une constante.

Pour les fonctionnalités définies ci-dessus, nous allons utilisé un algorithme naïf qui fonctionnera sur les matrices de petites dimensions.

4.2.2 Avancées

- Taille d'une matrice ;
- Calcul du déterminant : applications à des matrices LU pour le calcul du déterminant ;
- Transposition ;
- Inversion de matrice.

4.2.3 Optimisations

- Matrices creuses ;
- Stockage optimisé ;
- Multiplications : utilisation de l'algorithme de Strassen pour des matrices carrées de grande taille.

4.3 Problèmes soulevés

4.3.1 Utilisation mémoire

En mathématiques et en calcul matriciel, on est souvent amené à manipuler des matrices de grandes tailles qui peuvent comporter des milliers voire des millions d'éléments, mais dont la plupart sont nuls. Ces matrices sont appelées matrices creuses et utilisent une quantité non négligeable de mémoire pour stocker simplement des 0. Il peut donc être bon de réflechir sur des structures de données mieux adaptés à ce genre de matrices comme le Compressed Column Storage, le Compressed Spare Raw Format ou encore le format Harwell-Boeing.

4.3.2 Implémentation assembleur

Le problème de l'implémentation de ces méthodes est leur compléxité et leur application au niveau machine de ces dernières. Nous ne manquons pour le

moment de recul et de temps sur ces méthodes d'optimisation pour les proposer de manière fonctionnelle dans notre bibliothèque.

5. Méthode de validation

5.1 Base de tests

Afin de valider la conformité de notre extension, nous avons utilisé des jeux de tests écrits préalablement au développement qui nous permetraient de vérifier le fonctionnement, au moins basique, de notre bibliothèque. Nous avons procédé de la même manière que pour tout le reste du projet, avec l'utilisation régulière de la commande `mvn test` qui permettaient de vérifier non seulement le succès des tests par rapport aux résultats attendus, mais aussi de confirmer que l'implémentation effectuer de l'extension dans le *Lexer* et le *Parser* n'ont pas affecté les fonctionnalités préalablement établies.

Les méthodes de passage de tests ont été réalisés par palier correspondant à des fonctionnalités précises, comme l'ajout de deux matrices, le calcul de la longueur d'un tableau ou encore la multiplication de matrices. Chaque étape devait être totalement validée avant de pouvoir passer à la suite.

Une fois l'étape validée, un oracle était généré et venait s'ajouter aux oracles déjà existant pour s'arruser que le codage de la nouvelle fonctionnalité de cassait pas le travail déjà réaliser.

5.2 Utilisation de Jacoco

Afin de vérifier la couverture des nouveaux tests sur le code de l'extension, l'utilisation de **Jacoco** est un outil essentiel. L'utilisation de la commande `mvn verify` permet de vérifier la couverture de nos tests. Une couverture trop faible est synonyme de jeux de tests mal conçu, et l'objectif et de fournir une couverture sur les parties du code rajoutées qui soit supérieur à la couverture moyenne des jeux de tests sans l'extension. Grâce à **Jacoco**, nous pouvons aussi cibler précisement les portions de tests non parcourus et ainsi ajuster les tests en conséquence, notamment pour les tests invalides qui doivent soulever des exceptions.

Grâce à cela, nous avons un couverture moyenne de 84% sur les nouvelles fonctionnalités implémentées. Les tests fournissent donc une bonne couverture (couvertrue moyenne totale de 82%).

6. Résulats de la validation de l'extension

Nous avons vu avec Jacoco que les couvertures de tests présentes permettaient de valider en partie le bon fonctionnement de notre programme. Cependant, les pourcentages manquants peuvent encore être réduits, notamment au sujet des

tests invalides qui lèvent des erreurs. En effet, bien que nos tests soient complets sur les opérations valides de la bibliothèque, ils restent assez léger sur les parties qui doivent lever une exception.

Les parties basiques de l'extension semblent bien implémentés, toutefois les fonctions avancées n'ont pas pu être entièrement codées, faute de temps.

7. Amélioration

Actuellement l'abre contextuelle est cohérent pour la création d'un tableau d'objet d'une classe déclaré dans le fichier Deca et la compilation ne lance pas d'erreur. Il y a cependant une erreur à l'exécution. Cette erreur est lié à la liaison entre l'extension et le développement du langage avec objet. Je pense qu'il ne faudrait pas très longtemps pour corriger cet aspect et pouvoir manipuler des tableaux d'objet quelconque.

Une amélioration possible de notre extension est la gestion intelligente de la mémoire et notamment de la libération mémoire des tableaux. En effet, pour le moment, la libération en mémoire des tableaux lorsqu'ils ne sont plus utilisé n'est pas implémenté, ce qui peut entraîner des problèmes de remplissages du tas. Les recherches sur les méthodes ont été effectués (utilisation de `DEL` en assembleur) mais nous n'avons pas eu le temps de proposer une implémentation fonctionnelle de la fonctionnalité.