

Documentation de l'extension TAB

Version 1.0 (release)

Sommaire

Introduction	3
Une analyse bibliographique	4
Choix de conception, d'architecture, et d'algorithmes	4
Déclaration de tableaux et initialisation	4
<i>ArrayLib.decah</i>	6
<i>MatrixLib.decah</i>	8
Partie A	12
Grammaire concrète :	13
Partie B	18
Partie C	19
Limites	19
Méthode de validation	20
Résultats de la validation de l'extension	21
Conclusion	22

Introduction

Nous avons choisi de traiter l'extension [TAB], une extension de Deca avec des tableaux et une bibliothèque de calcul matriciel. Nous devions ainsi faire une syntaxe hors-contexte, une syntaxe abstraite, une syntaxe contextuelle et la sémantique. Nous nous sommes également vu proposer une bibliothèque de calcul matriciel. Cette extension offrait quelques libertés dans sa réalisation et cela nous a beaucoup plu. De plus, il nous semblait difficile de faire des programmes intéressants sans les tableaux. Cela apportait quelque chose de nouveau et d'utile.

Spécification de l'extension

Pour utiliser les fonctionnalités de manipulation de tableaux de notre compilateur, vous devrez utiliser la commande avec l'option lors de la compilation de vos `decac -tab` fichiers. Cela permettra au compilateur de détecter l'utilisation de tableaux. Il n'est pas obligatoire d'inclure la bibliothèque pour déclarer et utiliser des tableaux, cependant si vous souhaitez utiliser des fonctionnalités avancées de manipulation de tableaux il vous faudra inclure *ArrayLib.decah*. De même, pour manipuler les matrices il faudra inclure *MatrixLib.decah*. Les informations sur la déclaration et l'utilisation de tableaux sont disponibles ci-après.

Une analyse bibliographique

Afin de pouvoir commencer l'implémentation de l'extension, nous avons débuté par une recherche bibliographique pour comprendre les différentes étapes nécessaires à la création et l'utilisation d'un tableau. Pour cela nous avons cherché la spécification des tableaux en java ([ArrayJava](#)) ou le tableau doit être déclaré avec un type puis des crochets afin de choisir la dimension, il est alors ensuite possible d'initialiser le tableau avec `{valeurs}` ou `new type[val1][val2]...[]()` avec `val1` obligatoire puis les suivantes qui peuvent être précisées ou non.

Nous avons ensuite trouvé la gestion des tableaux en mémoire dans java. Quand nous souhaitons créer un tableau, de l'espace est alloué pour un nouveau tableau de cette longueur et s'il n'y a pas suffisamment d'espace pour allouer le tableau, le compilateur lance une erreur. Sinon, un tableau à une dimension est créé de la longueur spécifiée et chaque composant du tableau est initialisé à sa valeur par défaut. Si des initialiseurs sont présents entre `{}`, le tableau est initialisé de gauche à droite.

Choix de conception, d'architecture, et d'algorithmes

Déclaration de tableaux et initialisation

Pour déclarer un tableau, on écrit : `int[] tab` pour un tableau de dimension une, `int[]...[] tab` pour un tableau plus grand.

Pour initialiser un tableau, `int[] tab = new int[N]()` est utilisé pour un tableau de dimension une de longueur N. Pour un tableau de dimension deux, on peut définir seulement le nombre de lignes ou le nombre de lignes et de

colonnes : `int[][] tab = new int[N][]()`, `int[][] tab = new int[N][M]()`. Ainsi, pour un tableau plus grand, on peut initialiser plus ou moins de longueurs : `int[]...[] tab = new int[N][N1]...[Nk]()` ou `int[]...[] tab = new int[N][N1]...[Nk][]...[]()`.

Il est également possible d'initialiser les tableaux explicitement : `int[] tab = {1, 2, 3}`, `int[][] tab = {{0, 1}, {0, 1}, {0, 1}}`. On peut aussi initialiser des tableaux avec des sous-tableaux de tailles variées, par exemple `int[][] tab = {{1}, {2, 2}, {3, 3, 3}}` ou `int[][][] tab = {{{1}, {2, 2}}, {{3, 3, 3}, {1}}, {}}`.

Pour accéder à un élément `i` d'un tableau `t`, on écrit `t[i]`. Pour afficher une matrice `mat` :

```
int i = 0, j = 0;
while (i < mat.length) {
    while (j < mat[i].length) {
        print(mat[i][j], ' ');
        j = j + 1;
    }
    i = i + 1;
    println();
}
```

Concernant les méthodes de manipulation avancées des tableaux et des matrices, nous avons créé *ArrayLib.decah* qui contient les fonctions pour les tableaux (array, de dimension une) et *MatrixLib.decah* qui contient les fonctions pour les matrices.

Nous avons d'abord écrit les fonctions en java car cela s'écrit et se test plus simplement. Une fois que nous étions certains de la qualité de notre code, nous l'avons traduit en decah. Ainsi, nous avons remplacé les boucles "for" par des boucles "while", nous nous sommes assurés que les déclarations de variables se faisaient en début de fonctions, etc. Nous avons testé le code

decah de deux façons, tout d'abord nous l'avons remis en java et testé ainsi, ensuite nous avons créé des tests en deca sur chacune des méthodes.

ArrayLib.decah

ArrayLib.decah présente trois fonctions pour afficher les tableaux pour les trois types différents (flottant, entier et booléen) : `void printTabFloat(float[] tab)`, `void printTabInt(int[] tab)`, `void printTabBoolean(boolean[] tab)`.

De même, des fonctions permettent de remplir le tableau passé en argument de l'élément donné : `void fillFloat(float[] tab, float element)`, `void fillInt(int[] tab, int element)`, `void fillBoolean(boolean[] tab, boolean element)`. Cette fonction n'a pas été implémentée pour Objet car dans le tableau, on remplit avec une copie d'un seul élément en mémoire. Si on remplit avec un objet (`Rational` par exemple), il s'agirait d'une seule copie du même objet (on modifierait le même `Rational`).

Les méthodes de recherche d'un élément, `int searchFloat(float[] tab, float element)`, `int searchInt(int[] tab, int element)`, `int searchBool(boolean[] tab, boolean element)`, `int searchObject(Object[] tab, Object element)`, renvoient un entier qui indique la position dans le tableau `tab` du premier élément recherché, s'il a été trouvé. Sinon, l'entier renvoyé vaut -1.

Des méthodes de trie ont été créées pour trier les tableaux d'entiers et de flottants, `void quicksortFloat(float[] tab)`, `void quicksortInt(int[] tab)`. Ces méthodes font respectivement appel à `void quicksortRecFloat(float[] tab, int low, int high)`, `void quicksortRecInt(int[] tab, int low, int high)`. Cela permet au développeur d'entrer uniquement le tableau en paramètre est de toujours le trier entièrement. Enfin ces méthodes appellent récursivement `int partitionFloat(float[] tab, int low, int high)`, `int partitionInt(int[] tab, int low, int high)`.

L'algorithme de tri rapide, aussi appelé tri pivot, est l'un des algorithmes les plus efficaces de tri avec une complexité moyenne pour n éléments en $O(n \log(n))$. La complexité dans le pire des cas est quadratique, cette situation est évitable en implémentant correctement l'algorithme. Pour cela, nous nous sommes appuyés sur l'algorithme ci-dessous :

// Tri une partition du tableau (entre l'indice low et high), divise en deux partitions, puis tri ces parties

algorithm quicksort(A, low, high) **is**

// vérification sur les indices entrés en paramètres

if low >= high || low < 0 **then**

 return

// Appel à la fonction partition, renvoie l'index pivot

 p := partition(A, low, high)

// tri les deux partitions

 quicksort(A, low, p - 1) *// À gauche du pivot*

 quicksort(A, p + 1, high) *// À droite du pivot*

// Divise le tableau en deux partitions

algorithm partition(A, low, high) **is**

 pivot := A[high] *// Le dernier élément est le pivot*

// Pivot index temporaire

 i := low - 1

for j := low **to** high - 1 **do**

if A[j] <= pivot **then**

 i := i + 1

 swap A[i] **with** A[j]

// Met le pivot à la bonne position

 i := i + 1

 swap A[i] **with** A[hi]

return i *// renvoie l'index du pivot*

Des méthodes de comparaisons de tableaux ont également été réalisées `boolean compareFloat(float[] A, float[] B)`, `boolean compareInt(int[] A, int[] B)`, `boolean compareIntFloat(int[] A, float[] B)`, `boolean compareBool(boolean[] A, boolean[] B)`, `boolean compareObject(Object[] A, Object[] B)` et permettent de comparer des tableaux d'entiers, de flottants, de booléens, d'objets, mais aussi de comparer un tableau d'entier à un tableau de flottant.

Nous avons également créé des méthodes pour copier des tableaux d'entiers, de flottants et de booléens dans de nouveaux tableaux : `float[] deepCopyFloat(float[] tab)`, `int[] deepCopyInt(int[] tab)`, `boolean[] deepCopyBool(boolean[] tab)`. De la même façon, nous avons fait des méthodes pour copier une partie de tableau : `float[] deepCopyOfRangeFloat(float[] tab, int deb, int fin)`, `int[] deepCopyOfRangeInt(int[] tab, int deb, int fin)`, `boolean[] deepCopyOfRangeBool(boolean[] tab, int deb, int fin)`. Nous déclarons donc de nouveau tableau qui se remplissent des éléments sélectionnés. Si le tableau est plus grand que celui copié, il sera rempli de 0 à la suite des éléments du tableau initial.

Enfin, *ArrayLib.decah* présente deux méthodes pour convertir des tableaux d'entiers en flottants et inversement : `int[] castArrayFloatToInt(float[] tab)`, `float[] castArrayIntToFloat(int[] tab)`.

MatrixLib.decah

Concernant *MatrixLib.decah*, de nombreuses méthodes sont similaires à celles de *ArrayLib.decah*.

On retrouve en effet trois méthodes qui permettent d'afficher les matrices d'entiers, de flottants et de booléens : `void printMatFloat(float[][] matrice)`, `void printMatInt(int[][] matrice)`, `void printMatBool(boolean[][] matrice)`.

Des fonctions permettent de remplir le tableau passé en argument de l'élément donné : `void fillFloat(float[][] matrice, float element)`, `void fillInt(int[][] matrice, int element)`, `void fillBoolean(boolean[][] matrice, boolean element)`.

Les méthodes de recherche d'un élément : `int[] searchFloat(float[][] matrice, float element)`, `int[] searchInt(int[][] matrice, int element)`, `int[] searchBool(boolean[][] matrice, boolean element)`, `int[] searchObject(Object[][] matrice, Object element)`.

On peut également comparer deux matrices avec : `boolean compareFloat(float[][] A, float[][] B)`, `boolean compareInt(int[][] A, int[][] B)`, `boolean compareIntFloat(int[][] A, float[][] B)`, `boolean compareBool(boolean[][] A, boolean[][] B)`.

Enfin, des méthodes permettent de faire des copies de matrices : `float[][] deepCopyFloat(float[][] matrix)`, `int[][] deepCopyInt(int[][] matrix)`, `boolean[][] deepCopyBool(boolean[][] matrix)`, et des copies sur des parties de matrices : `float[][] deepCopyOfRangeFloat(float[][] matrix, int linStart, int colStart, int linEnd, int colEnd)`, `int[][] deepCopyOfRangeInt(int[][] matrix, int linStart, int colStart, int linEnd, int colEnd)`, `boolean[][] deepCopyOfRangeBool(boolean[][] matrix, int linStart, int colStart, int linEnd, int colEnd)`.

Outre ces fonctions, *MatrixLib.decah* permet de calculer la trace des matrices : `float traceFloat(float[][] matrice)`, `int traceInt(int[][] matrice)`.

De plus, nous avons implémenté des méthodes pour les calculs arithmétiques de matrices. Nous avons fait l'addition : `float[][] addFloat(float[][] A, float[][] B)`, `int[][] addInt(int[][] A, int[][] B)`, la soustraction : `float[][] subFloat(float[][] A, float[][] B)`, `int[][] subInt(int[][] A, int[][] B)`, la multiplication par un scalaire : `float[][] multScalarFloat(float[][] A, float lambda)`, `int[][] multScalarInt(int[][] A, int lambda)`, la multiplication de deux matrices entre elles : `float[][] multMatrixFloat(float[][] A, float[][] B)`, `int[][] multMatrixInt(int[][] A, int[][] B)`, la multiplication d'une matrice par un vecteur : `float[] multArrayMatrixFloat(float[] A, float[][] B)`, `int[] multArrayMatrixInt(int[] A, int[][] B)`, et de la même façon la multiplication d'un vecteur par une matrice. Il est également possible de convertir une matrice d'entiers en flottants et inversement : `float[][] castMatrixIntToFloat(int[][] matrix)`, `int[][] castMatrixFloatToInt(float[][] matrix)`.

À cela nous avons ajouté des méthodes pour des calculs matriciels plus complexes tels que le déterminant, le rang ou l'inverse. Pour ce faire, nous avons fait une recherche sur les différents algorithmes possibles. Après avoir comparé les possibilités, nous avons opté pour celui de Gauss-Jordan, décrit ci-dessous :

Gauss-Jordan

```

r = 0                                     (r est l'indice de ligne du dernier pivot trouvé)
Pour j de 1 jusqu'à m                     (j décrit tous les indices de colonnes)
|  Rechercher max(|A[i,j]|, r+1 ≤ i ≤ n).
|  |  Si A[k,j] ≠ 0 alors                 (A[k,j] est le pivot)
|  |  r=r+1                             r désigne l'indice de la future ligne servant de pivot)
|  |  Diviser la ligne k par A[k,j] (Normalisation de la ligne de pivot)
|  |  Si k ≠ r alors
|  |  |  Échanger les lignes k et r (On place la ligne du pivot en position r)
|  |  Fin Si

```

```

| | Pour i de 1 jusqu'à n           (On simplifie les autres lignes)
| | | Si i≠r alors
| | | | Soustraire à la ligne i la ligne r multipliée par A[i,j] (de façon à
annuler A[i,j])
| | | Fin Si
| | Fin Pour
| Fin Si
Fin Pour
Fin Gauss-Jordan

```

Nous avons ainsi traduit cela en decah, nous avons nommé cette méthode : `IntFloat gaussJordan(float[][] matrix)`. Elle prend en argument des matrices flottantes. On peut tout de même appliquer la méthode Gauss-Jordan à des matrices d'entiers en utilisant les méthodes de conversion.

L'algorithme de Gauss-Jordan permet d'en déduire le déterminant de la matrice passée en argument. En effet, ce dernier se calcul suivant la formule :

$$\det(A) = (-1)^p \cdot \prod_{j=1}^n (A[k, j])$$

Nous initialisons donc le déterminant à 1 lors de l'implémentation de l'algorithme. À chaque permutation effectuée nous multiplions le déterminant par -1 et nous le multiplions également à chaque fois par le nouveau pivot. La fonction `float det(float[][] matrix)` renvoie ainsi ce déterminant calculé dans la méthode gaussJordan.

De même, l'algorithme permet de calculer le rang de la matrice. On initialise le rang à la taille de la matrice, et à chaque colonne où le pivot est nul, le rang prend la valeur rang-1. La fonction `float rang(float[][] matrix)` renvoie ainsi ce rang calculé dans la méthode gaussJordan.

Enfin, à partir de cette méthode nous avons pu calculer l'inverse d'une matrice avec : `float[][] inverse(float[][] matrix)`. Cette méthode concatène la matrice entrée en paramètre à une matrice identité de la même taille. On applique ensuite Gauss-Jordan à cette nouvelle matrice. On obtient alors une matrice identité sur la partie droite, et la matrice inverse sur la partie gauche. On renvoie alors la partie gauche.

Pour certaines méthodes, nous avons eu besoin de vérifier les dimensions des matrices entrées en arguments. Nous avons besoin de renvoyer un message d'erreur en cas de problème. C'est pourquoi nous avons défini en bas de Matrix :

```
void matrixError()  
asm ("WSTR \"Erreur de dimension\"  
WNL  
ERROR");
```

Cela permet d'afficher le message "Erreur de dimension", de faire un retour à la ligne et d'afficher un message d'erreur. `matrixError()` est ainsi appelée lors de la conversion des matrices, du calcul du rang, du déterminant, de l'inverse, de la trace, des opérations sur les matrices (multiplication, addition, soustraction), et lorsque l'on fait l'algorithme de Gauss-Jordan.

Partie A

Concernant la partie A de l'extension. Nous avons d'abord modifier le lexer pour détecter les `[` (OBRACKET) et `]` (CBRACKET). Pour pouvoir créer des tableaux nous avons dû modifier la grammaire. Pour cela, nous avons ajouté ou modifié plusieurs terminaux et non terminaux :

- `tab_expr` qui permet de détecter l'initialisation du tableau en détectant `{expr,...}` ou `{}` et qui dérive de `assign_expr` pour permettre le `int[] a = {}` et dérive vers `or_expr` pour détecter les `abstarctExpr`
- `select_expr` ou nous avons rajouté la détection de `select_expr[expr]` afin de pouvoir par exemple récupérer l'élément d'un champ Tableau

d'une classe. Pour cela nous avons créé une classe ArraySel.java qui possède en champ deux expressions : une pour la sélection et l'autre pour l'index du tableau.

- primary_expr que nous avons modifié pour y ajouter un NEW tabIdent () qui dérive donc vers tabIdent et qui permet de détecter des déclaration de la forme new int[]()
- type dans lesquels nous avons créé une nouvelle branche IDENT([])+ qui nous permet de créer un nouveau type de la forme nomtype suivi d'un nombre (>1) de []. Pour cela nous avons créé une nouvelle classe Array.java qui contient trois champs: un symbol name, un abstractExpr IdentifierName et un int Level.
- enfin nous avons créé un terminal nommé tabident qui dérive de primary_expr qui détecte les tokens de la forme Ident ([expr])+([])* pour reconnaître les appels de tableaux de la forme a[1][] . Nous avons aussi créé une classe nommée TabIdentifier.java avec cinq champs : Symbol name, Symbol localType, int level, ListExpr listeposs, Definition definition.

Ce qui donne la grammaire ci-dessous.

Grammaire concrète :

prog

→list_classes main EOF

main

→ε

| block

block

→'{' list_decl list_inst '}'

list_decl

→decl_var_set*

decl_var_set

→type list_decl_var ';' ;

list_decl_var

→(decl_var) (';' decl_var)*

decl_var

→ident ('=' expr) ?

list_inst

→(inst)*

inst

→expr ';' ;

| ';' ;

| 'print' '(' list_expr ')' ';' ;

| 'println' '(' list_expr ')' ';' ;

| 'printx' '(' list_expr ')' ';' ;

| 'printlnx' '(' list_expr ')' ';' ;

| if_then_else

| 'while' '(' expr ')' '{' list_inst '}'

| 'return' expr ';' ;

if_then_else

→'if' '(' expr ')' '{' list_inst '}'

('else' 'if' '(' expr ')' '{' list_inst '}')*

('else' '{' list_inst '}') ?

list_expr

→(expr (';' expr)*) ?

expr

→assign_expr

assign_expr

→tab_expr

({ condition: expression must be a "lvalue" } '=' assign_expr

| ε)

tab_expr
→ '{' ((tab_expr) (',' tab_expr)*) ? '}'
| or_expr

or_expr
→ and_expr
| or_expr '||' and_expr

and_expr
→ eq_neq_expr
| and_expr '&&' eq_neq_expr

eq_neq_expr
→ inequality_expr
| eq_neq_expr '==' inequality_expr
| eq_neq_expr '!=' inequality_expr

equality_expr
→ sum_expr
| inequality_expr '<=' sum_expr
| inequality_expr '>=' sum_expr
| inequality_expr '>' sum_expr
| inequality_expr '<' sum_expr
| inequality_expr 'instanceof' type

sum_expr
→ mult_expr
| sum_expr '+' mult_expr
| sum_expr '-' mult_expr

mult_expr
→ unary_expr
| mult_expr '*' unary_expr
| mult_expr '/' unary_expr
| mult_expr '%' unary_expr

unary_expr

→ '-' unary_expr
| '!' unary_expr
| select_expr

select_expr

→ primary_expr
| select_expr '.' ident
 ('(' list_expr ')'
 | ε)
| select_expr '[' expr ']'

primary_expr

→ ident
| tabident
| ident '(' list_expr ')'
| '(' expr ')'
| 'readInt' '(' ')'
| 'readFloat' '(' ')'
| 'new' ident '(' ')'
| 'new' tabident '(' ')'
| '(' type ')' '(' expr ')'
| literal

type

→ ident
| IDENT ('[' ']') +

literal

→ INT
| FLOAT
| STRING
| 'true'
| 'false'
| 'this'
| 'null'

ident

→ IDENT

tabident

→ IDENT ('[' expr ']')+('[' ']')*

/**** Class related rules ****/

list_classes

→ (class_decl)*

class_decl

→ 'class' ident class_extension '{' class_body '}'

class_extension

→ 'extends' ident

| ε

class_body

→ (decl_method

| decl_field_set)*

decl_field_set

→ visibility type list_decl_field ';'

visibility

→ ε

| 'protected'

list_decl_field

→ decl_field

(';' decl_field)*

decl_field

→ ident ('=' expr) ?

decl_method

→ type ident '(' list_params ')' (block

| 'asm' '(' multi_line_string ')' ';')

list_params

→ (param(';' param)*) ?

multi_line_string
 → STRING
 | MULTI_LINE_STRING
 param
 → type ident

Partie B

Pour la partie contextuelle, nous avons choisit de suivre les règles suivantes :

tabident ↓ *env_types* ↑ *type*

→ Identifier ↑ *name*

condition : (__, *type*) , *env_types(name)*

[(**expr** ↓ *env_types* ↓ *env_exp* ↓ *class* ↑ *type*)]*

condition : *type* = int

type ↓ *env_types* ↑ *type*

→ Identifier ↑ *name* ('[' '']*)

condition : (__, *type*) , *env_types(name)*

expr ↓ *env_types* ↓ *env_exp* ↓ *class* ↑ *type*

→ [(**expr** ↓ *env_types* ↓ *env_exp* ↓ *class* ↑ *type*)*]{*type_r* := *type*}

condition : assign_compatible(*env_type*, *type*, *type_r*)

expr ↓ *env_types* ↓ *env_exp* ↓ *class* ↑ *type*

→ **expr** ↓ *env_types* ↓ *env_exp* ↓ *class* ↑ *type₁*

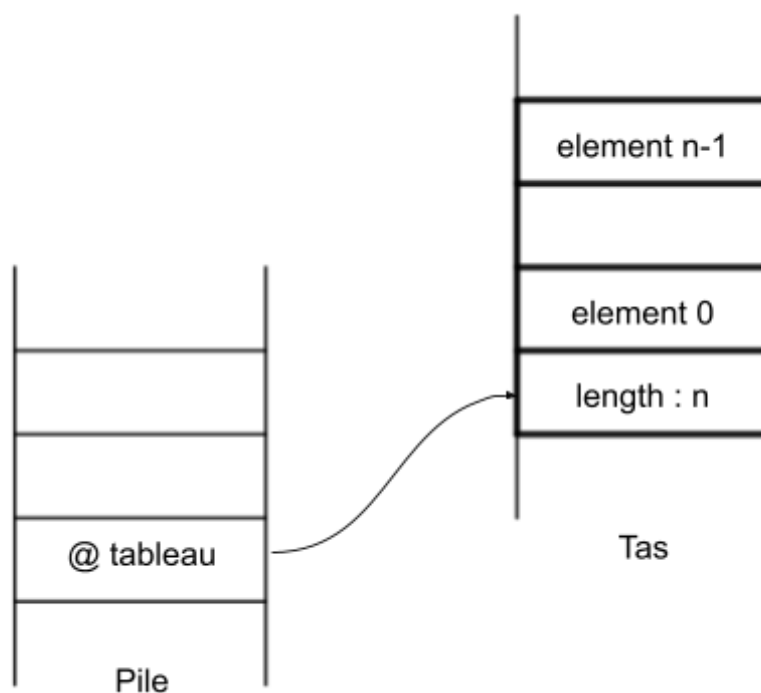
condition : *type₁* = array

expr ↓ *env_types* ↓ *env_exp* ↓ *class* ↑ *type₂*

condition : *type₂* = int

Partie C

Pour la gestion des tableaux dans la mémoire nous avons choisi de créer un élément dans le tas contenant la longueur du tableau puis la liste de ses éléments.



Ainsi il n'y a pas de problème pour créer et manipuler des tableaux imbriqués.

Limites

Après avoir implémenté et testé notre extension, nous pouvons constater certaines limites. Tout d'abord, les flottants induisent un calcul approximatif notamment pour Gauss-Jordan, le calcul de déterminant et d'inverse. Les arrondis peuvent manquer de précision et conduire à des résultats pas tout à fait exacts.

De plus, lors de la manipulation de tableaux et de matrices, il est possible d'attribuer des valeurs ailleurs que dans l'initialisation, i.e. que l'on peut écrire pour `tab` défini par `float[] tab;` des valeurs plus loin dans le code : `tab = {2, 3}`. En java, il n'est possible d'attribuer les valeurs qu'à l'initialisation. Cependant, cela permet tout de même d'offrir plus de liberté au développeur, et est donc une option qui peut être bénéfique.

Concernant les tableaux d'objet, le sous-typage et le sur-typage ne sont pas forcément gérés. Par exemple, un objet de type A rempli d'objets de type B fonctionne. Il n'y a pas de détection du type.

De surcroît, nous n'avons pas implémenté de fonctions pour la concaténation des tableaux et des matrices. C'est d'ailleurs pourquoi dans la méthode du calcul de l'inverse d'une matrice, nous concaténons nous même la matrice entrée en paramètre avec la matrice identité.

Enfin, les performances de nos algorithmes auraient pu être améliorées. En effet, nous les avons codé en decah, il est nécessaire de les recompiler à chaque fois par notre compilateur. Cela est peu efficace : notre code n'est pas aussi performant que si les algorithmes avaient été écrits directement en assembleur.










Méthode de validation

Pour la validation de notre extension, nous nous sommes servis de notre architecture de test. Nous avons augmenté le nombre de tests et ainsi, nous avons agrandi notre couverture. Nous avons ainsi pu tester le nouveau lexer et le nouveau parser. Nous avons également ajouté des tests sur chacune des fonctions de *ArrayLib.decah* et *MatrixLib.decah*. Il s'agit donc de tests pour les tableaux à une dimension et matrices à deux dimensions, de type

entier, flottant et booléen. Nous avons également réalisé des tests spécifiques à la partie context et codegen.

Les tests sont organisés de la façon suivante, dans `src/test/deca/extension`, il y a quatre dossiers contenant chacun des tests, eux-mêmes contenus dans les dossiers `valid` et `invalid`. Le premier dossier, nommé “`parser`” présente les tests sur la détection des nouveaux terminaux. Le dossier “`lexer`” contient les tests sur la détection des ‘[’ et ‘]’. Concernant “`context`”, ce dossier porte sur la création de tableau de la mauvaise forme, de mauvaise dimension et sur l’utilisation de tableau dans une fonction. Enfin, le dossier “`codegen`” présente des tests de toutes les fonctions de `ArrayLib.decah` et `MatrixLib.decah`.

Résultats de la validation de l’extension

Element	Missed Instructions	Cov.	Missed Branches	Cov.
 TabIdentifier		59%		59%
 Array		35%		25%
 ArrayLiteral		81%		61%

Nous avons réalisé des tests pour chaque fonction mais ceux-ci ne sont pas visibles sur Jacoco. On constate que le coverage des tests unitaires n’est pas complet notamment à cause de certaines erreurs qui n’ont pas été testées.

Conclusion

Finalement, nous avons réussi à implanter une version de tableau déclarable dans deca qui ressemble à la version de java et qui est fonctionnelle sur la partie testée mais il manque quelques tests sur des parties mineures de l'extension. Nous avons aussi réussi à implanter un nombre assez important de fonctions utiles pour le calcul matriciel ou sur des tableaux mais d'autres fonctions pourraient être rajoutées par un futur utilisateur du compilateur. Nous avons personnellement bien apprécié cette partie de l'extension car nous étions assez libres de choisir ce que nous voulions faire.