Projet Génie Logiciel

Extension

Aurélien VILMINOT
Damien CLAUZON
Guilherme KLEIN
Léon ROUSSEL
Pierre ARVY

26 janvier 2022

Résumé

Ce document vise à détailler le développement d'une extension pour le projet de Génie Logiciel. L'extension choisie se nomme TAB et porte sur les tableaux. Les tableaux étant absents de la version initiale de Deca, leur ajout passe par différentes phases. Tout d'abord, une partie de recherche pour comprendre comment adapter les différentes parties du projet à cette extension. Ensuite vient la conception des spécificités adaptées au langage Deca. Enfin, une partie d'implémentation pour ajouter la possibilité d'utiliser les tableaux dans le langage Deca.

Le but principal étant de pouvoir exploiter une bibliothèque de calcul matriciel, la partie sur des tableaux à une ou deux dimensions d'entiers et de flottants est prioritaire. Cette bibliothèque de calcul contient quelques fonctions de bases très utiles pour réaliser des calculs en algèbre.

Table des matières

1 Documentation Java

Deca étant un langage ressemblant à Java, nous nous intéressons donc aux différentes syntaxes du langage Java. Voici quelques liens qui vont nous être utiles pour étendre la grammaire de Déca donnée dans le polycopié. Dans cette partie, le but n'est pas de construire la grammaire Deca, mais d'étudier et d'extraire des informations sur la syntaxe de Java afin de pouvoir ensuite l'adapter au langage Deca. Des éléments Java qui ne sont pas présents dans le langage Deca seront donc présents dans cette partie, comme les types `short`, `long`, ...

1.1 Quelques liens

Pour commencer, voici quelques liens qui contiennent les principales informations. Les trois premiers sont issus de la documentation Java (version SE 17).

- Spécifications générales : The Java® Language Specification
- Section 2, grammaires : Chapter 2. Grammars
- Section 10, tableaux : Chapter 10. Arrays

Au-delà de la syntaxe Java officielle, voici un autre document moins officiel, mais plus simple à lire : Java BNF

1.2 Description sur les tableaux

1.2.1 Généralités

En Java, les tableaux sont des objets créés dynamiquement. Toutes les méthodes de la classe *Object* peuvent être appelées sur un tableau. Le nombre de composantes d'un tableau définit sa taille. Un tableau peut être vide. Tous les éléments d'un tableau sont d'un même type T. Le type du tableau est alors `T[]`. Un tableau peut lui-même contenir des tableaux. Le type des éléments d'un tableau est le type des éléments des sous-tableaux les plus profonds. Une fois le tableau créé, sa taille ne change pas. La taille d'un tableau peut être trouvé avec l'attribut `length`.

Quelques exemples de déclaration et quelques explications :

- `int[] tab;` ne créé pas les objets du tableau, cela créé uniquement la variable
- `int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };` créé effectivement un tableau de `int`, la partie à droite du signe égal est appelée l'initialiseur du tableau
- `int[] factorial = new int[3];` créé un tableau de `int` initialisé à 0. La partie à droite du signe égal est appelée le créateur de tableau
- `byte[] rowvector, colvector, matrix[];` est équivalent à `byte rowvector[], colvector[], matrix[][];`
- `float[][] f[][], g[][][], h[];` est équivalent à la liste de déclaration suivante `float[][][][] f; float[][][][][] g; float[][][] h;`

1.2.2 Initialiseur de tableau

L'initialiseur de tableau est une suite d'expressions séparées par des virgules et entourée par des accolades ({}). Une virgule peut apparaître derrière le dernier terme de la suite, elle sera ignorée.

Il doit y avoir une compatibilité entre le type du tableau et le type de la variable assignée. Pour le cas des tableaux, les différents cas suivants doivent être traités comme compatibles, et il doit donc y avoir une conversion entre le type de la valeur assignée et le type de la variable qui sera attribuée dans le tableau.

Les conversions possibles en Java sont répertoriées dans le lien suivant : Chapter 5. Conversions and Contexts.

- Si l'expression n'est pas compatible pour être convertie, une erreur est soulevée.
- Si l'espace alloué n'est pas assez grand, une erreur de type *OutOfMemoryError* est soulevée, sinon, l'espace est alloué et initialisé avec une valeur par défaut (0, false, null, ... selon le type).

— Les composantes sont ensuite initialisées avec les valeurs données dans le code source (entre accolades).
— Si l'initialisation d'une variable échoue, alors l'initialisation du tableau doit échouer.
— Si la composante est un tableau, alors l'initialisation de la variable doit être effectuée récursivement comme l'initialisation d'un tableau. Cela entraîne donc des déclarations comme celle-ci : `int ia[][] = { {1, 2}, null };`.

1.2.3 Créateur de tableau

Il est également possible de créer un nouveau tableau via une expression de création de tableau. Un exemple de cette création est `int[] factorial = new int[3];`.

Tout comme l'initialiseur de tableau, le créateur de tableau doit contenir le type primitif ou la classe des éléments du tableau. La taille du tableau peut être spécifié entre les crochets du créateur de tableau.

1.2.4 Accès à une composante

Pour accéder à une composante d'un tableau, il faut une expression qui référence un tableau suivie d'une expression qui retourne un index entouré de `[]`. Par exemple `A[2]` accède au troisième élément de A. L'index est une valeur de type int, ou de type short, byte, or char. Il n'est pas possible par exemple possible d'avoir un index de type long.

Les tableaux ont une origine à 0, ce qui signifie que les index possibles vont de 0 à n-1 où n est la taille du tableau. La vérification de l'accès à une composante est faite pendant l'exécution, si l'index est strictement négatif ou supérieur strict à n-1, une erreur de type *ArrayIndexOutOfBoundsException* est envoyée.

1.2.5 Assignement

La possibilité d'assignement d'une composante est vérifiée pendant l'exécution. Si un tableau est de type A[] et que l'on cherche à modifier une de ses composante, il est vérifié pendant l'exécution que la valeur assignée est de type A. Dans le cas où la valeur assignée n'est pas du même type que A, une erreur de type `ArrayStoreException` est soulevée.

1.2.6 Récapitulatif des méthodes et attributs d'un tableau

Les attributs et méthodes du tableau sont :
— `length`
— La méthode clone qui retourne une copie du tableau principal, mais qui partage encore les sous-tableaux.
— Tous les attributs et méthodes de la classe Object (sauf `clone`) par héritage des tableaux
Tous les tableaux ont également un object `Class` associé, ce qui permet de tester l'égalité de classe entre différents tableaux.

1.3 Syntaxe

Voici quelques parties de la syntaxe Java qui vont nous être utiles, notamment pour s'en inspirer lors de la modification des syntaxes Deca. Les quelques liens qui vont nous être utiles sont les suivants.
— 15.10. Array Creation and Access Expressions
— 10.6. Array Initializers
— 15.10.3. Array Access Expressions
— 15.8. Primary Expressions

1.3.1 ArrayCreationExpression

Les règles de grammaires suivantes décrivent la création des tableaux. On a le type qui est écrit par le non-terminal *PrimitiveType*. Puis les crochets avec *Dims* et *DimsExprs* en commençant par les crochets avec des index à l'intérieur.

ArrayCreationExpression :
  new PrimitiveType DimExprs Dims_{opt}

new ClassOrInterfaceType DimExprs Dims_{opt}
new PrimitiveType Dims ArrayInitializer
new ClassOrInterfaceType Dims ArrayInitializer

DimExprs :
DimExpr
DimExprs DimExpr

DimExpr :
[Expression]

Dims :
[ ]
Dims [ ]

1.3.2 ArrayInitializer

Les règles de grammaires suivantes ne servent pas toutes exclusivement pour les tableaux, mais elles aident à comprendre comment arriver au non terminal ArrayInitializer. Pour plus de règles, se référer à la partie 10.6. Array Initializers de la documentation Oracle.

FieldDeclaration :
$\text{FieldModifiers}_{\text{opt}}$ Type VariableDeclarators ;

VariableDeclarators :
VariableDeclarator
VariableDeclarators , VariableDeclarator

VariableDeclarator :
VariableDeclaratorId
VariableDeclaratorId = VariableInitializer

VariableDeclaratorId :
Identifier
VariableDeclaratorId [ ]

VariableInitializer :
Expression
ArrayInitializer

ArrayInitializer :
{ $\text{VariableInitializers}_{\text{opt}}$ $,_{\text{opt}}$ }

VariableInitializers :
VariableInitializer
VariableInitializers , VariableInitializer

1.3.3 ArrayAccess

Les règles concernant l'accès à une composante d'un tableau.

ArrayAccess :
ExpressionName [Expression]

PrimaryNoNewArray [Expression]

1.3.4 PrimaryExpression

Une des particularités de la syntaxe Java est la restriction de l'accès à un tableau. Un accès à un tableau ne peut pas être fait sur un créateur de tableau. Il y a donc une distinction dans la dérivation de *Primary*. Soit une dérivation vers une création de tableau, soit une dérivation vers un élément qui n'est pas une création de tableau.

Plus tard dans l'adaptation de la syntaxe abstraite de Deca, nous reviendrons sur ce point.

Primary :
PrimaryNoNewArray
ArrayCreationExpression

PrimaryNoNewArray :
Literal
ClassLiteral
this
TypeName . this
(Expression)
ClassInstanceCreationExpression
FieldAccess
ArrayAccess
MethodInvocation
MethodReference

2 Choix de l'implémentation en Deca

Dans cette partie, nous spécifions l'implémentation des tableaux en Deca. Cette dernière respectera la syntaxe de Java avec certaines fonctionnalités en moins.

2.1 Généralités

En Deca, les tableaux sont des objets créés dynamiquement. Toutes les méthodes de la classe `Object` peuvent être appelées sur un tableau. Le nombre de composantes d'un tableau définit sa taille. Un tableau peut être vide. Tous les éléments d'un tableau sont d'un même type T. Un tableau peut avoir une dimension ou plus, spécifiée par les crochets qui suivent le type. Une fois le tableau créé, la taille de ses dimensions ne change pas. La taille d'un tableau peut être trouvé avec l'attribut `length`. Par exemple `T[][] tab;` est un tableau de type `T[][]`, a deux dimensions, correspond à un tableau de tableaux de type `T[]`, et son attribut length renvoie le nombre de tableaux qu'il contient. Même si un tableau est un objet possédant les méthodes de la classe `Object`, il n'est pas considéré au niveau de la grammaire comme une classe.

Les tableaux supportent les types `int`, `float` et les classes créées par l'utilisateur.

Quelques exemples de déclaration et quelques explications :

- `int[] tab;` ne crée pas les objets du tableau, cela crée uniquement la variable.
- `int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };` créé effectivement un tableau de `int`, la partie à droite du signe égal est appelée l'initialiseur du tableau.
- `int[] factorial = new int[3];` créé un tableau de `int` initialisé à 0. La partie à droite du signe égal est appelée le créateur de tableau. La taille de chaque dimension est **EXIGÉE** dans le créateur de tableau.
- `int[][] matrix, bmatrix;` est équivalent à `int[][] matrix; int[][] bmatrix;`
- `float[][] f[][], g[][][], h[];` n'est **PAS** valide. Il faut initialiser les tableaux comme la liste de déclaration suivante : `float[][][][] f; float[][][][][] g; float[][][] h;`

2.2 Initialiseur de tableau

L'initialiseur de tableau est une suite d'expressions séparées par des virgules et entourée par des accolades ({}). Une virgule peut apparaître derrière le dernier terme de la suite, elle sera ignorée.

Il doit y avoir une compatibilité entre le type du tableau et le type de la variable assignée. Le cas des tableaux est similaire à la compatibilité pour l'affectation existante en Deca.

Si l'espace alloué n'est pas assez grand, une erreur est soulevée, sinon, l'espace est alloué et initialisé avec une valeur par défaut (0, false, null, ...selon le type). Les composantes sont ensuite initialisées avec les valeurs données dans le code source (entre accolades). Si l'initialisation d'une variable échoue, alors l'initialisation du tableau doit échouer. Si la composante est un tableau, alors l'initialisation de la variable doit être effectuée récursivement comme l'initialisation d'un tableau.

2.3 Créateur de tableau

Il est également possible de créer un nouveau tableau via une expression de création de tableau. Un exemple de cette création est `int[][] factorial = new int[3][5];`. Tout comme l'initialiseur de tableau, le créateur de tableau doit contenir le type primitif ou la classe des éléments du tableau. La taille du tableau spécifié entre les crochets du créateur de tableau doit être une expression retournant un entier. Par exemple, int n; int[] factorial = new int[n = 5]; est valide mais `int[] factorial = new int[5.0];` est invalide. Toutes les dimensions doivent être spécifiées avec ce créateur de tableau. Par exemple, `int[][] factorial = new int[2][];` et `int[] factorial = new int[];` sont invalides.

2.4 Accès à une composante

Pour accéder à une composante d'un tableau, il faut une expression qui référence un tableau suivie d'une expression qui retourne un entier entouré de [ ]. Par exemple A[2] accède au second élément de A. Les tableaux ont une origine

à 0, ce qui signifie que les entiers possibles vont de 0 à n-1 où n est la taille du tableau. La vérification de l'accès à une composante est faite pendant l'exécution, si l'entier est strictement négatif ou supérieur strict à n-1, une erreur est envoyée.

2.5 Assignement

La possibilité d'assignement d'une composante est vérifiée pendant l'exécution. Si un tableau est de type A[] et que l'on cherche à modifier une de ses composante, il est vérifié pendant l'exécution que la valeur assignée est de type A. Dans le cas où la valeur assignée n'est pas du même type que A, une erreur est soulevée.

2.6 Récapitulatif des méthodes et attributs d'un tableau

Les attributs et méthodes du tableau sont :

- `length`
- Tous les attributs et méthodes hérités de la classe Object (sauf `clone`), c'est-à-dire la méthode `Object.equals`

Tous les tableaux ont également un object `Class` associé, ce qui permet de tester l'égalité de classe entre différents tableaux.

La création de tableaux se fera aussi de la même manière que pour le langage Java. Une nuance ajoutée est que la taille sera exigée en Deca. Cela n'est pas une grosse concession, car lors de la création d'un tableau pour effectuer des calculs, la taille est souvent déjà fixée.

Une fois ces non-terminaux introduit, il faut modifier la déclaration des variables. Nous ajoutons la possibilité de déclarer un tableau et la possibilité de l'initialiser. Cette déclaration de tableau peut se faire comme une variable du programme principale, mais aussi comme un attribut d'une classe.

2.7 Limitations liées au temps

La contrainte du temps nous pousse à créer une liste de priorité pour les fonctionnalités à implémenter, le but étant d'avoir au minimum la possibilité de créer des vecteurs et des matrices pour créer une bibliothèque de calculs matriciels.

- Tableaux d'`int` et de `float` (1)
- Initialisation et accès aux éléments (2)
- Tableaux de tout type primitif ou de classe (3)
- Tableaux n-dimensionnels (4)
- Méthode `clone` pour copier les éléments d'un tableau dans un autre (5)
- Syntaxe identique à celle de Java (ex : `int[][] tab` équivalent à `int[] tab[]`) (6)

En prenant en compte nos capacités techniques et le temps qu'il nous reste, nous choisissons de prendre plusieurs décisions :

- Implémenter (1) et (2), mais en implémentant uniquement le créateur de tableau. Cette structure laisse la possibilité de créer de grands tableaux pour les calculs et il est toujours possible d'initialiser les valeurs du tableau avec l'assignation après l'initialisation
- Se limiter à des éléments de type `int` et ou `float`
- Se limiter à des tableaux avec au plus 2 dimensions
- La méthode `clone` (5) et les déclarations équivalentes (6) ne sont pas implémentées

Ces choix simplifient grandement l'implémentation et nous assure presque d'avoir une version fonctionnelle à la fin du projet.

2.8 Récapitulatif sur les choix d'implémentation

Déclaration d'un tableau à 1 ou 2 dimensions

```
<type>[] <identifiant>;
<type>[][] <identifiant>;
```

Initialisation d'un tableau à 1 ou 2 dimensions

```
<type>[] <identifiant> = new <type> [<index>];
<type>[][] <identifiant> = new <type> [<index>][<index>];
```

La valeur de l'expression issue de `<index>` *doit être de type* `int`*, elle définit la taille du tableau.*

Assignation

```
1 int[] tableau;
2 int[] second_tableau = new int[<taille>];
3 tableau = new int[<taille>];
4 tableau = second_tableau;
```

Deux assignations ci-dessous à la ligne 3 et 4.

Accès au éléments

```
1 float composante;
2 float[] tableau = new float[<taille>];
3 tableau[<index>] = <flottant>;
4 composante = tableau[<index>];
```

La valeur de l'expression issue de `<index>` *doit être de type* `int`*, elle correspond à l'indice de la composante à laquelle on accède. L'index d'accès doit être un entier compris dans l'intervalle* $[0, n-1]$*, où* n *est la taille du tableau. La première composante est accessible avec l'index* 0*, la seconde avec l'index* 1 *et ainsi de suite.*

Accès à la taille

```
int taille = <identifiant>.length;
```

3 Adaptation des syntaxes Deca

3.1 Comment adapter la syntaxe ?

Les éléments présentés ci-dessus ne suffisent pas à recréer entièrement la syntaxe Deca. Il faut aller davantage en profondeur dans les différentes dérivations possibles pour s'assurer de toutes les possibilités. Pour cela, une bonne pratique peut être de dessiner un arbre avec les non-terminal. Cet arbre se transforme rapidement en un graphe pour la syntaxe Java à cause de certaines boucles (*Primary* dérive vers *(Expression)* et *Expression* dérive vers *Primary* en Java par exemple).

Le principal défi est d'ajouter les fonctionnalités de cette extension sans casser les fonctionnalités déjà existantes. Il ne faut pas que l'ajout d'éléments dans les grammaires modifie le comportement de ce qui est déjà en place.

3.2 Adaptation de la lexicographie

3.2.1 Modifications

La liste des mots réservés ne change pas. Nous ajoutons cependant **2 symboles spéciaux** : '**[**' et '**]**'. Les crochets servent pour définir les tableaux et pour accéder aux éléments d'un tableau. Ces deux crochets sont donc à ajouter dans le lexer.

- **OBRACKET** : '['
- **CBRACKET** : ']'

La création d'un identifiant pour les tableaux est également faite. Ce terminal sera utilisé plus tard dans la syntaxe concrète.

IDENT_TAB : **IDENT** (**[]**) $*$

Il s'agit d'un identifiant suivi de crochets directement après, sans espace superflu.

3.3 Adaptation de la syntaxe concrète

Les modifications réalisées aboutissent à des créations de règles et des créations de non-terminaux. Afin de simplifier l'implémentation, nous avons essayé d'imiter au maximum les comportements de la grammaire initiale de Deca, tout en s'inspirant des idées de la grammaire Java.

Les remarques en italique font références aux noms des non-terminaux en Java.

3.3.1 Explications

La première modification va venir modifier le non-terminal **type**. En effet, le type peut maintenant être `int[][]` ou `float[]` par exemple. Nous décidons donc d'ajouter un non-terminal **ident_tab**. Pour plus de facilité, les crochets devront directement suivre le type primitif sans espace entre les crochets (ceci a été décrit dans la partie sur la lexicographie). Il en résulte que `int []` ou `int[ ]` ne seront pas reconnus.

Il vient donc la modification des deux règles suivantes.

type→**ident**|**ident_tab**

ident_tab→**IDENT_TAB**

Il vient ensuite la construction du non-terminal **array_creator_expr** qui sera utilisé pour créer un tableau. Le non-terminal **dim_expr** est fait d'une suite d'expression entre crochets. Chaque expression entre crochets donne la taille de la dimension. Il est à noter que **expr** ne peut pas dériver vers ϵ, il n'y a pas de possibilité de ne pas donner de taille à une dimension.

Il vient donc la création des deux règles suivantes.

array_creator_expr→**'new' ident dim_expr**

dim_expr→'[' **expr** ']' ('[' **expr** ']')∗

Pour interagir avec les tableaux, il y a besoin d'une structure d'accès. L'accès à un tableau se fait avec un tableau et l'index de la composante que l'on cherche à obtenir. Le tableau peut être issu d'un appel de méthode ou d'un attribut d'une classe (résultant de **select_expr**), d'un identifiant ou d'une expression parenthésé (résultant de **primary_expr**), etc. Il faut donc que le non-terminal faisant référence au tableau dans la structure de l'accès soit **select_expr**. L'index peut être issu d'une expression. La vérification du type de l'expression sera faite dans la partie contextuelle. Cette première réflexion donne donc la structure suivante.

array_access→**select_expr** '[' **expr** ']'

Cette structure d'accès peut rendre un tableau si la dimension du tableau initial est strictement supérieure à 1, ou une valeur de même type que le type primitif du tableau. Le choix le plus naturel et qui coïncide le mieux avec la syntaxe de Java, est d'ajouter la dérivation **primary_expr** → **array_access**. Cela permet de conserver la structure initiale de la grammaire.

Cette dérivation permet par ailleurs de créer des accès en cascade de manière récursive. `tab[5][4]` sera donc reconnu comme 2 accès. Le premier accès s'effectue sur le tableau `tab`, et le second sur le tableau `tab[5]`. Il en résulte la règle suivante.

primary_expr→...| **array_access**

C'est ici que se présente la principale difficulté de la création de cette grammaire. Il ne faut pas créer d'ambiguïté au niveau de l'accès et de la création d'un tableau. Le lecteur remarquera que la structure **array_creator_expr** n'a pas encore été intégré à la grammaire existante, aucune dérivation n'est possible vers ce non-terminal.

Le plus naturel serait de l'ajouter comme une dérivation possible de **primary_expr**, il en résulterais donc la dérivation suivante : **primary_expr** → **array_creator_expr**.

Il apparaît alors un problème. `new int[5][2]` doit-il être interprété comme la création d'un tableau d'entier à deux dimensions de taille $(5, 2)$, ou comme l'accès de la troisième composante du tableau qui vient d'être créé ? Quand l'utilisateur écrit `new int[5][2]`, pense-t-il écrire `(new int[5][2])` ou `(new int[5])[2]` ? Sans parenthéser, il est impossible de savoir et c'est pour cela qu'il y a ambiguïté.

Il faut donc lever l'ambiguïté. Pour cela, il faut être certain que l'expression **select_expr** ne puisse pas dériver vers **array_creator_expr** directement.

C'est là que la syntaxe de Java aide à construire la nôtre. Il y a choix à effectuer après le non-terminal *Primary* en Java. Soit vers *ArrayCreationExpression* (l'équivalent de notre non-terminal **array_creator_expr**), soit vers *PrimaryNoNewArray* (qui est l'équivalent de notre non-terminal **select_expr**).

Ce dernier point d'équivalence entre *PrimaryNoNewArray* et **select_expr** est plus difficile à comprendre. En Java, les accès aux attributs ou aux méthodes se font **après** *Primary*, alors qu'il se font dans **select_expr** en Deca. Le choix de dériver vers *PrimaryNoNewArray* ou vers *ArrayCreationExpression* doit donc se faire "au-dessus" de **select_expr** en Deca.

Afin de créer ce choix, nous créons un non-terminal **choose_expr** "entre" **unary_expr** et **select_expr**. Il en résulte donc les créations et modifications suivantes.

unary_expr→'-' **unary_expr**|→'!' **unary_expr**|**choose_expr**

choose_expr→**select_expr** *(PrimaryNoNewArray)* |**array_creator_expr** *(ArrayCreationExpression)*

Ainsi, la règle **array_access** →**select_expr** '[' **expr** ']' ne permet pas d'avoir directement une création de tableau dans **select_expr**. Donc, `new int[5][3]` est donc obligatoirement une création de tableau à deux dimensions et l'accès à une composante d'un tableau qui vient d'être créé se fait avec des parenthèse (`(new int[5])[3]`).

3.3.2 Résumé des modifications

Voici donc un résumé des règles et des non-terminaux ajoutés.
[Adaptation règle]
unary_expr
→**'-' unary_expr**
| →**'!' unary_expr**
| **choose_expr**

[Ajout non-terminal]
choose_expr
→**select_expr***(primary no new array)*
| **array_creator_expr**

[Ajout règle]
primary_expr
→**ident**
|**ident '(' list_expr ')'** *(method invocation)*
| **'(' expr ')'**
| **' readInt ' '(' ')'**
| **' readFloat ' '(' ')'**
| **'new' ident '(' ')'** *(class instance creation expression)*
| **'(' type ')' '(' expr ')'**
| **literal**
| **array_access**

[Ajout non-terminal]
array_access
→**select_expr '[' expr ']'**

[Ajout non-terminal]
array_creator_expr
→**'new' ident dim_expr**

[Ajout non-terminal]
dim_expr
→**'[' expr ']' ('[' expr ']')***

[Ajout d'une règle]
type
→**ident**
| **ident_tab**

[Ajout non-terminal]
ident_tab
→**IDENT_TAB**

3.3.3 Remarques

Malgré ces modifications, il reste tout de même des différences entre la syntaxe Java et la syntaxe Deca. Par exemple le cast d'une variable est une expression primaire en Deca, le cast est donc prioritaire sur les autres opérations (notamment la sélection), cela n'est pas le cas en Java.

3.4 Adaptation de la syntaxe abstraite

3.4.1 Explications

La construction de la syntaxe abstraite est importante, car elle détermine partiellement la décompilation ainsi que la syntaxe contextuelle. Il s'agit de récupérer les structures créer précédemment et d'ajouter les dérivations possibles.

La création d'un tableau est composée d'un identifiant et de la liste des index. Nous créons le terminal NewArray ainsi que la règle suivante.

EXPR →NewArray [**IDENTIFIER LIST_EXPR**]

L'accès à un tableau peut servir lors d'une assignation à gauche du signe égal ou comme une simple expression. Nous créons donc ArrayAccess, qui prendra en argument un tableau et un index, et qui sera une dérivation de **LVALUE** via la règle suivante.

LVALUE →ArrayAccess[**EXPR EXPR**]

Les tableaux étant vus comme des variables, les règles de déclaration des variables et des attributs ne sont pas modifiés, tout comme l'initialisation des variables et des attributs.

DECL_VAR →DeclVar [**IDENTIFIER IDENTIFIER INIZIALIZATION**]

DECL_FIELD →DeclField ↑*Visibility* [**IDENTIFIER IDENTIFIER INIZIALIZATION**]

INITIALIZATION →Initialization [**EXPR**] | NoInitialization

3.4.2 Résumé des modifications

Il en résulte donc l'ajout des règles suivantes.

EXPR
→…
| NewArray [**IDENTIFIER LIST_EXPR**]

LVALUE
→…
| ArrayAccess[**EXPR EXPR**]

3.4.3 Remarques

Après ces ajouts de règles, il faut vérifier que les règles de la grammaire ne sont pas modifiées. Cette grammaire abstraite a pour règle :
Une propriété importante de cette grammaire est d'une part qu'un nom de nœud donné n'apparaît que dans une seule règle, et d'autre part qu'un non-terminal donné apparaît au plus une fois en tant que membre droit d'une règle d'inclusion. Ce critère syntaxique simple garantit la non-ambiguïté de la grammaire ainsi définie.

Les terminaux ajoutés sont bien présents une seule fois dans la syntaxe abstraite. Et il n'y a pas de non-terminaux ajoutés. La règle ci-dessus est donc bien conservée après la modification de la grammaire.

3.5 Adaptation de la décompilation

Cette partie est sûrement la plus simple, il suffit de reprendre la syntaxe abstraite et d'y ajouter les attributs. Les notations sont les mêmes que celle du polycopié du projet.

3.5.1 Explications

Toute la décompilation des types (`int [ ] [ ]` par exemple) se fait à l'intérieur même du non-terminal **IDENTIFIER**, il n'y a donc aucun changement à prévoir pour les types.

Pour le terminal NewArray, il faut décompiler l'identifiant ainsi que les expressions qui se situe dans les crochets. La notation utilisée n'est pas très conventionnelle ({dims := ϵ} {dims := dims.'['.e.']'}), mais il faut la comprendre de cette manière : l'attribut 'dims' est calculé récursivement. Il est vide à l'initialisation, et on concatène l'expression courante de la liste entre crochets à chaque itération.

EXPR ↑r → NewArray [**IDENTIFIER** ↑type **LIST_EXPR** ↑dims {dims := ϵ} {dims := dims.'['.e.']'}] {r := 'new '.type.dims}

Le terminal Access a lui une décompilation très simple à comprendre.

LVALUE ↑r → Access[**EXPR** ↑e **EXPR** ↑index] {r := e.'['.index']'}

3.5.2 Résumé des modifications

EXPR ↑r
→ …
| NewArray [**IDENTIFIER** ↑type **LIST_EXPR** ↑dims {dims := ϵ} {dims := dims.'['.e.']'}]
{r := 'new '.type.dims}

LVALUE ↑r
→ …
| Access[**EXPR** ↑e **EXPR** ↑index]
{r := e.'['.index']'}

3.6 Adaptation de la syntaxe contextuelle

L'ajout de nouveaux types permet de ne pas retoucher au terminal Initialization.

L'ajout du terminal ArrayAccess comme **lvalue** implique que l'index d'accès soit de type `int` et que l'identifiant réfère à un tableau. Les tableaux étant limités à une et deux dimensions et de type primitif `int` ou `float`, on obtient la condition présente ci-dessous. L'affection renvoie le type de l'élément accédé. Il faut enlever une dimension pour obtenir ce nouveau type.

lvalue $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type$
→ …
→ ArrayAccess [
expr $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_1$
expr $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type_2$
]
condition $type_1 \in$ {int[], int[][], float[], float[][]} et $type_2$ = int

$$\textbf{\textit{affection}}\ type := \begin{cases} \text{int si } type_1 = \text{int[]} \\ \text{float si } type_1 = \text{float[]} \\ \text{int[] si } type_1 = \text{int[][]} \\ \text{float[] si } type_1 = \text{float[][]} \end{cases}$$

L'ajout du terminal NewArray vérifie que le type primitif est `int` ou `float`, et que la dimension un ou deux. Pour plus de lisibilité, c'est dans le non-terminal **expr_index** qu'est vérifié le type de l'index.

expr $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type$
$\rightarrow \ldots$
$\rightarrow$ NewArray [
type $\downarrow env\_types \uparrow type$
$\{dimension := 0\}$
[(**expr_index** $\downarrow env\_types \downarrow env\_exp \downarrow class$ $\{dimension := dimension + 1\}$) $*$]
]
condition $type \in$ {int, float} ET $dimension \in \{1, 2\}$.

$$\textbf{\textit{affection}}\ type := \begin{cases} \text{int[] si } type = \text{int et } dimension = 1 \\ \text{int[][] si } type = \text{int et } dimension = 2 \\ \text{float[] si } type = \text{float et } dimension = 1 \\ \text{float[][] si } type = \text{float et } dimension = 2 \end{cases}$$

expr_index $\downarrow env\_types \downarrow env\_exp \downarrow class$
$\rightarrow$ **expr** $\downarrow env\_types \downarrow env\_exp \downarrow class \uparrow type$
condition $type$ = int

4 Implémentation

4.1 Analyse syntaxique

Cette partie nécessite la modification de deux éléments : le lexer et le parser.

4.1.1 Implémentation Lexer

Comme précisé dans la partie sur les modifications lexicographiques (cf. 3.2), la reconnaissance des crochets nécessite l'ajout de deux éléments dans le lexer.

```
    OBRACKET : '[';
CBRACKET : ']';
```

Cela entraîne également l'ajout du terminal **IDENT_TAB**.

```
    IDENT_TAB :
(LETTER | '$' | '_')(LETTER | DIGIT | '$' | '_')*((OBRACKET)(CBRACKET))+;
```

4.1.2 Implémentation Parser

Les modifications du parser sont toutes des conséquences de la modification de la grammaire abstraite (cf. 3.3.2). Il s'agit donc d'utiliser la même façon d'implémenter que pour la partie initiale du langage Deca. En effet, une fois la grammaire abstraite modifiée, il n'y a aucune différence avec l'implémentation des différentes parties 'Hello World' , 'sans-objet' ou 'avec objets'.

Voici tout de même un petit exemple de modification du fichier `DecaParser.g4` :

```
1 array_creator_expr returns[NewArray tree]
2    // t for type
3    : NEW t=ident d=dim_expr{
4       assert($t.tree != null);
5       assert($d.tree != null);
6       $tree = new NewArray($t.tree, $d.tree);
7       setLocation($tree, $NEW);
8    }
9    ;
```

Ci-dessus l'ajout du non-terminal de la création de tableau. A la ligne 6, se situe la création de l'objet NewArray avec le type et la liste d'expression en argument.

Les classes instanciées par le parser ont été créées dans le dossier `src/main/java/fr/ensimag/deca/tree`. Ces classes implémentent également les fonctions de vérification de la partie suivante.

4.2 Analyse contextuelle

Les modifications de la syntaxe contextuelle qui sont effectuées (cf. 3.6) sont très légères. Cela rend donc la tâche assez facile pour l'adaptation du code Java.

Le fonctionnement avec 2 dimensions au maximum et avec un nombre de types primitifs restreints permet de simplifier le fonctionnement au niveau des types. Nous choisissons de créer une classe par type, il y en a ainsi 4 : *VectorFloatType*, textitVectorIntType, *MatrixFloatType* et *MatrixIntType*.

L'ajout de ces types dans l'environnement se réalise dans le script *DecaCompiler.java*.

Pour la création des fonctions de vérification, il suffit de suivre les modifications de la syntaxe contextuelle (cf. 3.6). Les erreurs à lever sont toutes répertoriées dans la liste des erreurs. Ces erreurs traitent notamment de la vérification de la dimension et du type des index.

4.3 Génération de code

Comme il a été vu pendant la syntaxe abstraite, deux règles de la syntaxe abstraite sont à implémenter : la création d'un tableau avec NewArray, et l'accès à un élément avec ArrayAccess. L'accès au champ `length` est aussi possible sur un tableau. Détaillons ces trois éléments en expliquant également leur implémentation en assembleur.

4.3.1 Création d'un tableau

L'aspect le plus important à voir en premier est la représentation d'un tableau dans notre langage.
Un tableau est une adresse mémoire qui pointe vers une portion du tas démarrant par la taille du tableau et suivie par les éléments du tableau. Les éléments du tableau sont tous du même type, et peuvent être des entiers, des flottants, ou des tableaux.
Lors de la création d'un tableau par l'instruction `new`, ce dernier est initialisé avec des valeurs par défaut. Pour un tableau à 1 dimension, les éléments du tableau sont initialisés à 0 ou 0.0. Pour un tableau à 2 dimensions, les éléments du tableau sont des tableaux dont les éléments sont initialisés à 0 ou 0.0.

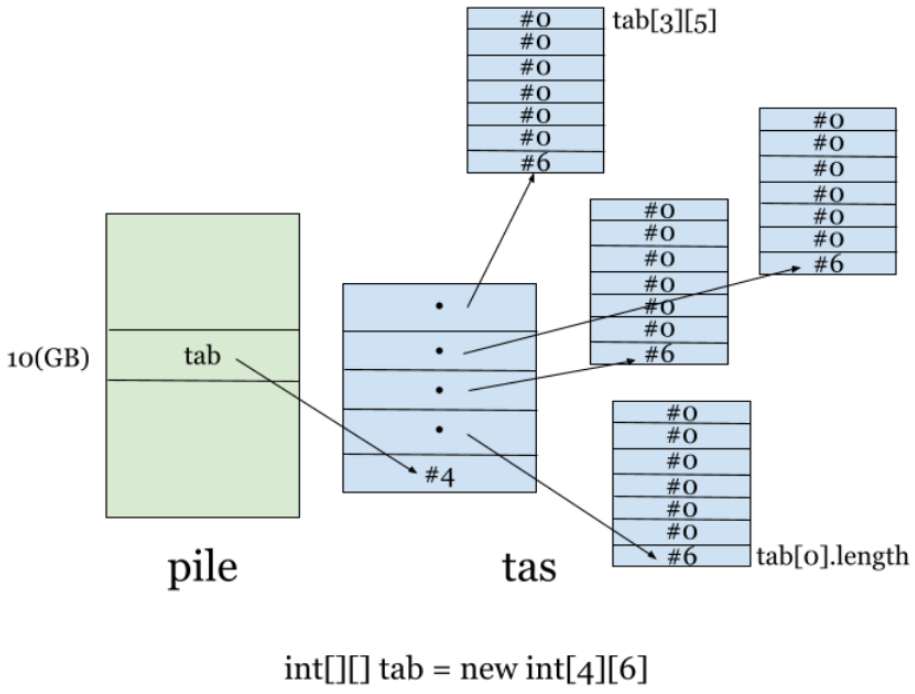

FIGURE 1 – État de la pile et du tas lors de la création d'un tableau 2D

La taille du tableau est calculée comme une expression entière, puis elle stockée dans un registre. Un espace mémoire est réservé dans le tas par l'instruction NEW en ajoutant 1 à la taille, afin de pouvoir stocker la taille du tableau ainsi que les éléments du tableau. La taille du tableau est stockée dans la première case de l'espace créé.
Une boucle est faite sur la condition $taille \neq 0$ pour remplir les éléments du tableau, dans laquelle l'adresse mémoire du tableau est incrémenté d'un mot, et la taille décrémentée de 1 à chaque tour de boucle. A la fin, on associe au tableau l'adresse mémoire qui contient sa taille.
Nous prenons également soin qu'aucun registre en cours d'utilisation ne soit modifié après la création d'un tableau.

4.3.2 Accès à la taille

Comme décrit dans la section précédente, la taille d'un tableau est stockée dans la première case de l'espace mémoire associée au tableau. Il suffit donc d'accéder au champ `length` de façon similaire au champ d'une classe, c'est-à-dire en chargeant le contenu de l'index 0 de l'adresse mémoire du tableau dans un registre.

4.3.3 Accès à un élément

L'accès à un élément se fait comme l'accès à la taille, en chargeant le contenu de l'index $n + 1$ de l'adresse mémoire du tableau dans un registre, où n est l'index accédé dans le tableau. Ce comportement est compatible avec les tableaux 1D et 2D.

4.4 Limitations

L'accès à un élément est en $\mathcal{O}(n)$ car nous n'avons pas trouvé d'instruction IMA permettant d'accéder au n^{eme} élément suivant une adresse mémoire, où n est contenu dans un registre. L'instruction LEA ne permet pas de le faire qu'avec un immédiat.

Nous avons donc fait le choix d'être le plus général possible en acceptant des index comme des expressions et non des immédiats, ce qui cause une complexité non négligeable pour l'accès à un élément.

Des optimisations auraient pu être faites cependant, par exemple en distinguant les immédiats des autres expressions pendant la génération de code.

La taille du tableau n'est pas vérifiée à sa création, à l'inverse de l'accès à un élément, ce qui peut causer une erreur d'exécution IMA quand la taille est négative.

Il n'est pas possible d'utiliser les commande de type `print` sur les tableaux. Il est donc conseillé de les afficher en itérant sur chaque élément.

5 Méthode de validation

Contrairement aux autres parties du compilateur, la validation de l'extension s'est entièrement faite en développant de manière dirigée par les tests. En effet, la plupart de ceux-ci (pour chaque étape) ont été écrits avant de développer les différentes parties concernées. Cela a permis d'assurer que le résultat du développement était conforme aux spécifications des grammaires étendues décrites précédemment.

Pour le *parser*, des dizaines de tests de conformité passés en "oracle" ont été écrits en avance pour tester la bonne implémentation de la grammaire concrète décrite précédemment. Tous ces tests se retrouvent dans l'arborescence, dans le répertoire `/src/test/deca/syntax/extension`. On y retrouve des tests valides et invalides comme pour la validation du compilateur.

Pour valider l'*analyse contextuelle*, et le code des classes Java rajoutées dans l'arborescence et qui constituent celle-ci, des tests unitaires ont été écrits en avance en plus des dizaines de tests de conformité eux aussi passés en "oracle". Ces derniers se retrouvent dans le répertoire `/src/test/deca/context/extension`.

Enfin, pour valider la génération de code, et ce de manière similaire au compilateur, des tests dits "boîte-noire" et des tests de conformité ont été fournis, et que l'on peut retrouver dans le répertoire `/src/test/deca/codegen/extension`.

La validation de tous ces tests a été automatisée avec le script `test-ext.sh` qui se trouve dans le répertoire `/src/test/script`. Celui-ci permettait de surveiller la non-régression du code lorsque des modifications étaient apportées pour reconnaître les grammaires décrites ci-dessus. Finalement, le compilateur est conforme aux différentes spécifications décrites dans les parties 2 et 3. Cela se vérifie largement avec les tests fournis, et notamment ceux qui utilisent la librairie de calcul matriciel décrite dans la partie suivante (situés dans le répertoire `/src/test/deca/codegen/extension/valid/library`). On y retrouve des déclarations de tableaux, l'initialisation de ceux-ci, l'assignement et enfin l'accès à des composantes. À noter cependant que les tests sur la génération de code ne sont pas suffisants. En effet, certaines erreurs ne sont pas encore traitées, comme l'initialisation d'un tableau à taille négative.

6 Bibliothèque de calcul matriciel

Comme spécifié sur l'extension, une bibliothèque rudimentaire de calcul matriciel a été implémentée en Deca (fichier `MatrixLibrary.decah`). Cette librairie doit être ajoutée à tout fichier Deca qui souhaite l'utiliser avec la commande suivante :

```
1 #include "MatrixLibrary.decah";
```

Définissant une classe, elle doit être ajoutée avant le code correspondant au bloc principal du programme. Après ça, il suffit d'instancier la classe `AlgebraLib` dans du code Deca, et d'appeler les méthodes de calcul matriciel sur l'objet correspondant.

En plus de tester concrètement l'implémentation de l'extension, et donc de la valider, elle propose à un utilisateur des fonctions de calcul matriciel, et vectoriel. En effet, des fonctions élémentaires (affichage, somme…) ont également été implémentées pour ces objets (correspondants à des tableaux). Les composantes de toutes les matrices et de tous les vecteurs sont des flottants. Cela est nécessaire pour réaliser certaines opérations basiques sur celles-ci (division…). A noter que la limitation de la taille du tas restreint les opérations à des "petites" matrices (moins de 10 000 composantes).

Chaque fonction du fichier `MatrixLibrary.decah` est documentée ci-dessous (en plus d'être commentée dans le code), comme sur le manuel utilisateur.

1. **Méthode :** `float[][] uniformMatrix(int height, int width, float value)`
   Retourne une matrice dont les composantes sont initialisées à la valeur `value`.

2. **Méthode :** `float[] uniformVector(int len, float value)`
   Retourne un vecteur dont les composantes sont initialisées à la valeur `value`.

3. **Méthode :** `void printMatrix(float[][] matrix)`
   Affiche une matrice sur la sortie standard.

4. **Méthode :** `void printVector(float[] vector)`
   Affiche un vecteur sur la sortie standard.

5. **Méthode :** `float[] linspace(float min, float max, float step)`
   Retourne un vecteur dont les composantes sont comprises entre `min` compris et `max` non compris, et espacées par le pas `step`.

6. **Méthode :** `float[][] reshape(int height, int width, float[] vector)`
   Retourne une matrice de dimensions `height` x `width` correspondant au vecteur `vector` redimensionné.

7. **Méthode :** `boolean isSquarred(float[][] mat)`
   Retourne `true` si la matrice `mat` est carrée, `false` sinon.

8. **Méthode :** `boolean sameSize(float[] vec1, float[] vec2)`
   Retourne `true` si les vecteurs `vec1` et `vec2` sont de même taille, `false` sinon.

9. **Méthode :** `boolean sameDim(float[][] mat1, float[][] mat2)`
   Retourne `true` si les matrices `mat1` et `mat2` sont de même taille, `false` sinon.

10. **Méthode :** `float[][] addMat(float[][] mat1, float[][] mat2)`
    Retourne une matrice issue de l'addition des matrices `mat1` et `mat2`.

11. **Méthode :** `float[][] subMat(float[][] mat1, float[][] mat2)`
    Retourne une matrice issue de la soustraction des matrices `mat1` et `mat2`.

12. **Méthode :** `float[][] multMat(float[][] mat1, float[][] mat2)`
    Retourne une matrice issue de la multiplication des matrices `mat1` et `mat2`.

13. **Méthode :** `float[] addVector(float[] vector1, float[] vector2)`
    Retourne un vecteur issu de l'addition des vecteurs `vector1` et `vector2`.

14. **Méthode :** `float[] subVector(float[] vector1, float[] vector2)`

Retourne un vecteur issu de la soustraction des vecteurs `vector1` et `vector2`.

15. **Méthode :** `float multVector(float[] vec1, float[] vec2)`
    Retourne le flottant correspondant au produit scalaire (euclidien) entre les vecteurs `vec1` et `vec2`.

16. **Méthode :** `float[][] transpose(float[][] mat)`
    Retourne la matrice correspondant à la transposée de la matrice `mat`.

La plupart de ces méthodes vérifient que les dimensions des paramètres sont cohérentes avant de passer au calcul matriciel/vectoriel. Lorsque cela n'est pas le cas, `null` est renvoyé. Une erreur est alors levée à l'exécution.