



Projet Jeu Yōkai-no-mori

Rapport IA

Djaber Merah
Gautier Raimondi

Année 2018-2019

Table des matières

Table des matières	i
1 Introduction	3
1.1 Présentation du projet	3
2 Choix de l'algorithme	6
2.1 MiniMax	6
2.2 Alpa-Beta pruning	6
2.3 Machine Learning	7
2.4 MonteCarlo Tree Search	7
2.5 Choix de l'algorithme utilisé	7
2.6 Exigences de l'application	8
2.7 Décomposition du logiciel	8
3 Lexer & Parser	10
3.1 MiniJaJa	10
3.1.1 Grammaire	10
3.1.2 AST	11
3.1.3 Converter	14
3.2 JaJaCode	14
4 Gestion de la mémoire	15
4.1 Table des Symboles	15
4.2 La pile	17
4.3 Le tas	18
5 Gestion du MiniJaJa	20
5.1 Architecture de classes	20
5.2 Contrôleur de type	20
5.3 Interpréteur	20
5.4 Compilateur	22

6	Gestion du JaJaCode	23
6.1	Architecture de classes	23
6.2	Interpretor	23
6.3	Printer	23
7	Interface Homme-Machine	24
7.1	Différents onglets	24
7.2	Coloration Syntaxique et Breakpoints	24
7.3	Console et Output	25
8	Tests	26
9	Bilan	28
A	Grammaire MiniJaJa initiale	29
B	Grammaire JaJaCode initiale	30
C	Grammaire MiniJaJa modifiée	31

Remerciements

Avant de présenter ce rapport, nous tenons particulièrement à remercier notre enseignant en MOIA, Monsieur Fabrice Bouquet, ainsi que Madame Violeta Felea, enseignante en SCS, pour leurs cours qui ont été la base de ce projet. Nous les remercions aussi pour nous avoir aidés par la suite avec de nombreux conseils.

Conventions, Définitions et Acronymes

L'ensemble de ce document sera écrit en sans-sérif. De plus, chaque occurrence de mot ou acronyme explicité dans le glossaire ci-dessous sera un lien, et la première apparition de chaque entrée apparaîtra en gras dans le texte.

Mot	Définition
AST	Un AST (pour Abstract Syntactic Tree) est un arbre dont les nœuds internes sont marqués par des opérateurs et dont les feuilles (ou nœuds externes) représentent les opérandes de ces opérateurs. Autrement dit, généralement, une feuille est une variable ou une constante.

TABLE 1 – Glossaire de termes rencontrés dans le document

1 Introduction

Lors de notre première année de Master Informatique à l'UBFC, il nous a été donné comme projet de réaliser une intelligence artificielle pour le jeu de Yōkaï-no-mori, dont les règles seront détaillées plus bas. Pour des détails concernant la mise en place de la communication entre le client et le serveur de jeu, nous vous invitons à vous référer au rapport correspondant. Seule la communication entre le moteur IA et le client de jeu sera observée dans ce rapport.

1.1 Présentation du projet

Le projet Jeu Yōkaï-no-mori qui nous a été proposé consistait en l'écriture d'un serveur de jeu avec clients ainsi que d'un moteur IA pour ledit jeu.

Ce moteur IA devait être implémenté en Prolog avec communication avec Java à l'aide de la librairie Jasper.

Il devait respecter les règles du jeu présentées dans le sujet, à savoir :

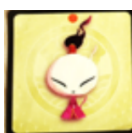
Yōkaï-no-mori est un jeu de stratégie à deux joueurs. Il se joue sur un plateau de 30 cases (5 colonnes de A à E et 6 lignes de 1 à 6). Chacun des joueurs dispose de 4 types de pions appelés yōkaï (représentés par 6 figurines, voir 1.1, les points indiquent les sens des déplacements) :

Le koropokkuru : *en gros l'équivalent du roi, qui peut se déplacer d'une case dans toutes les directions. Initialement, il y en a un par joueur.*

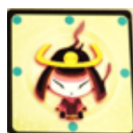
Le kodama (esprit de l'arbre) : *qui se déplace d'une case vers l'avant (pion au point rouge) mais qui peut être 'promu' en Kodama samouraï :] il peut alors se déplacer d'une case dans toutes les directions sauf les diagonales arrières (pion aux points bleus). Initialement, il y en a trois par joueur.*

Le kirin (licorne) : *qui peut se déplacer d'une case dans toutes les directions sauf les diagonales arrières. Initialement, il y en a deux par joueur.*

L'oni (ogre) : *qui peut se déplacer d'une case en avant (ou diagonales avant), et sur les diagonales arrières. Il peut aussi être promu en Super Oni : il peut alors se déplacer d'une case dans toutes les directions, sauf les diagonales arrières. Initialement, il y en a deux par joueur.*



Kodama



Kodama samouraï



Kirin



Koropokkuru



Oni



Super Oni

FIGURE 1.1 – Les pièces du jeu

Le tableau initial avec la position des pions est donné dans la figure 1.2. Il est à noter la zone de promotion correspondant aux deux lignes de fond de chaque camp. Pour savoir à qui appartient un pion, il suffit de regarder le sens dans lequel il est : tête vers le sud pour le joueur de la zone nord et inversement pour le joueur de la zone sud.

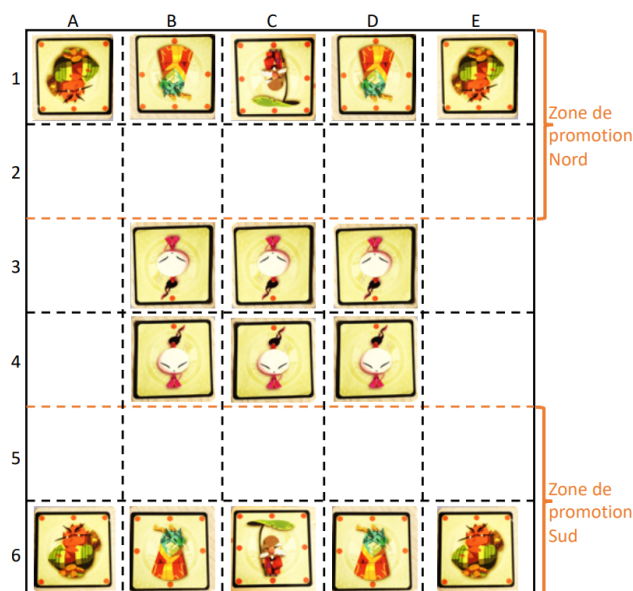


FIGURE 1.2 – Plateau initial

Chaque joueur joue à tour de rôle sans pouvoir passer son tour. Le joueur débutant la partie est celui situé au nord.

À son tour, un joueur peut faire une seule des actions suivantes :

- Déplacer un pion sur une case vide.
- Déplacer un pion sur une case occupée par l'adversaire (capturer).
- Déposer sur le plateau un pion capturé (parachutage).

Le déplacement est propre à chaque yōkaï qui se déplace d'une case dans l'une des directions indiquées par ses points de couleur. La case de destination doit être vide ou occupée par un pion adverse, auquel cas le pion adverse sera capturé.

Si un kodama ou un oni se déplace dans la zone de promotion adverse, il est alors promu et passe dans la configuration à points bleus.

La capture d'un yōkaï adverse se fait en déplaçant un yōkaï sur la case le contenant. À ce moment, le yōkaï adverse est mis dans la réserve du joueur qui l'a capturé (et dépromu si nécessaire). Chaque joueur a ainsi sa propre réserve dans laquelle il met les pions qu'il capture à l'adversaire. Si, lors de la capture, le yōkaï déplacé rentre la zone de promotion adverse, il est promu.

Le dépôt d'un pion capturé (parachutage) se fait en prenant un pion de sa réserve (que l'on a précédemment capturé) et que l'on remet en jeu obligatoirement sur une case vide. Il est interdit de mettre un kodama sur une colonne déjà occupée par un autre de ses kodamas. Il est également interdit de mettre un kodama sur la dernière ligne du camp adverse. Si l'on dépose un pion dans la zone de promotion, celui-ci n'est pas promu. Il le sera lorsqu'il se déplacera dans celle-ci.

L'objectif final est de capturer le koropokkuru adverse. Néanmoins, la victoire est détectée dès lors qu'il

y a équivalent d'échec et mat, lorsque quelque soit le mouvement effectué par un joueur, la capture du koropokkuru est inévitable. De plus, si plus de 60 coups sont joués (30 par joueur), un match nul est déclaré.

Le moteur IA doit également être capable d'envoyer un coup en 6 secondes à compter de la réception du coup précédent et peut utiliser n'importe quelle technique.

2 Choix de l'algorithme

À la réception du sujet, nous avons commencé par établir une liste non-exhaustive des différentes méthodes utilisées en Intelligence Artificielle, et plus particulièrement, pour des raisons de simplicité de recherche, en GGP (General Game Playing). Nous avons remarqué que les plus notables étaient les suivants : MiniMax, Alpha-Beta pruning, Machine Learning et MonteCarlo Tree Search.

2.1 MiniMax

D'après Wikipedia, l'algorithme MiniMax *“est un algorithme qui s'applique à la théorie des jeux pour les jeux à deux joueurs à somme nulle (et à information complète) consistant à minimiser la perte maximum (c'est-à-dire dans le pire des cas)”*.

Le premier constat est donc qu'il s'applique bel et bien à notre cas.

Il consiste en une visite exhaustive de l'arbre de jeu jusqu'à une profondeur donnée (ou un noeud terminal), avec évaluation par une heuristique de chacune des feuilles de notre arbre de visite. Ces évaluations sont ensuite comparées et on garde la branche nous fournissant la plus haute valeur lorsque c'est notre coup ou la plus basse pour le coup de l'adversaire. Cela nous garantit le coup nous apportant la meilleure évaluation à profondeur donnée.

L'avantage principal de cet algorithme est qu'il nous assure le meilleur coup théorique à profondeur donnée, mais il nécessite pour cela un temps non négligeable. De plus, il est nécessaire de disposer d'une bonne heuristique pour que le meilleur coup relatif à cette heuristique se révèle être le meilleur coup en pratique.

2.2 Alpa-Beta pruning

L'algorithme alpha-beta, appelé en français élagage $\alpha\beta$, correspond à une technique permettant de réduire l'espace de recherche lors de l'application de l'algorithme précédent, MiniMax.

Il consiste à explorer seulement un arbre partiel des coups, en lieu et place de l'arbre exhaustif. Pour cela, on sauvegarde à tout instant deux valeurs, α et β , telle que α représente la valeur actuelle maximum pour un noeud max, et β la valeur minimum pour un noeud min. Ces valeurs sont mises à jour pour chaque noeuds, et permettent d'identifier un noeud garantissant un résultat moins intéressant qu'un noeud déjà exploré. Il n'est alors pas traité.

La technique alpha-bêta permet une réduction de l'espace de recherche MiniMax et donc théoriquement d'atteindre une profondeur supérieure. Il possède néanmoins toujours une temps de traitement conséquent et la nécessité de développer une heuristique.

2.3 Machine Learning

La branche du ‘Machine Learning’ est une branche de l’intelligence artificielle s’intéressant à l’étude statistique afin de développer des solutions. On peut citer par exemple les réseaux de neurones ou les algorithmes génétiques, faisant évoluer notre IA sur base de sélection naturelle.

Cette méthode permet d’obtenir des résultats généralement satisfaisants, mais demande un travail de compréhension et des pré-calculs relativement conséquents

2.4 MonteCarlo Tree Search

Enfin, l’algorithme MCTS est un algorithme de recherche heuristique. On y explore l’arbre des coups, mais pas exhaustivement, et sans profondeur maximale. En effet, on y réalise des simulations (‘roll-outs’) jusqu’à obtenir un noeud terminal (égalité ou victoire).

On utilise alors une structure d’arbre pour mémoriser les différents noeuds déjà explorés et leurs statistiques (nombre de simulations, pourcentage de victoires ...).

Théoriquement, et dans un monde idéal, les résultats de cet algorithme convergent vers MiniMax. Néanmoins, cela demande une puissance de calcul irréaliste, et n’est pas réalisable.

Cependant, cet algorithme permet le développement d’une IA sans l’écriture d’une heuristique qui peut être biaisée ou erronée, et alors influencer l’IA dans une mauvaise direction. De plus, l’algorithme est particulièrement efficace pour les jeux à haut taux de branchements, ce qui est le cas ici à cause des parachutages, car pouvant agrandir son arbre de manière asymétrique. Enfin, à tout moment, le résultat fourni par la méthode est sensé et peut être utilisé. Il est donc plus facile de fonctionner avec un principe de timeout.

Néanmoins, il est possible de rater une branche amenant vers une victoire assurée à cause du parcours non-exhaustif de l’arbre.

2.5 Choix de l’algorithme utilisé

Dû à l’utilisation imposée de Prolog, nous avons rapidement mis de côté le Machine Learning, nécessitant des implémentations d’algorithmes complexes, et des grandes puissances de calcul.

Nous restait alors le choix entre le MiniMax, l’Alpha-Bêta, et le MCTS. Si l’utilisation de l’Alpha-Bêta par rapport au MiniMax paraissait évidente (à condition de réussir à l’implémenter), le doute subsistait par rapport au MonteCarlo Tree Search. Nous avons alors décidé de commencer par implémenter un MCTS sans trop d’amélioration, puis un AlphaBeta et de comparer les taux de victoires de chacun pour ensuite peaufiner l’algorithme utilisé.

Néanmoins, quelques problèmes de programmation et d’utilisation de la librairie Jasper nous ont amené à ne pouvoir tester notre IA que récemment, et il ne nous a donc pas été possible de comparer les résultats. Nous avons donc préféré prendre le risque et améliorer le MCTS, en essayant d’optimiser certains calculs, structures de données, et/ou transferts de résultats.

2.6 Exigences de l'application

Entre les exigences fournies dans le cahier des charges et celles que nous avons nous-même formulées, nous arrivons à la liste d'exigences suivantes :

Exigences fonctionnelles

- On veut pouvoir vérifier qu'un programme MiniJaJa est correct.
- On veut pouvoir vérifier qu'un programme MiniJaJa est bien typé.
- On veut pouvoir compiler tous les programmes MiniJaJa corrects et bien typés.
- On veut pouvoir interpréter un programme MiniJaJa correct et bien typé.
- On veut pouvoir interpréter un programme JaJaCode correct.
- On veut pouvoir ouvrir un fichier dans l'éditeur.
- On veut pouvoir éditer un fichier ouvert.
- On veut pouvoir sauvegarder les modifications d'un fichier ouvert.
- On veut pouvoir fermer un fichier ouvert.
- On veut pouvoir observer le JaJaCode résultant de la compilation.
- On veut pouvoir réaliser l'interprétation pas à pas de programmes MiniJaJa et JaJaCode.
- On veut pouvoir observer l'état mémoire à tout moment de l'interprétation.
- On veut pouvoir utiliser des fonctionnalités de breakpoints.
- On veut pouvoir effacer le contenu de la console de l'interface.

Exigences non-fonctionnelles

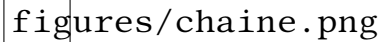
- La compilation et l'interprétation doivent être conformes aux règles décrites dans le cours.
- Les messages d'erreurs doivent s'afficher dans l'interface et être le plus clair possible. Ils doivent localiser l'erreur dans le fichier source.
- La réalisation des analyseurs lexical et syntaxiques doit utiliser Javacc ou Antlr.
- Les structures de données mémoire la pile et le tas doivent être mises en oeuvre le plus efficacement possible. La réalisation des opérations doit être basée sur l'accès direct. Une pile avec des entrées par hachage est la structure conseillée pour MEM et un système de ramasse-miette pour le tas. L'utilisation des techniques de hachage permet de satisfaire la contrainte d'accès direct au mieux.
- On doit pouvoir observer les états mémoire de fin d'exécution des programmes afin de les comparer.
- L'ensemble du projet doit être réalisé dans un même environnement cohérent.

2.7 Décomposition du logiciel

Afin de simplifier le travail en équipe, le projet a été décomposé en différents modules selon le raisonnement suivant :

Chaque module doit effectuer une tâche le plus indépendamment des autres modules possible et ne doit correspondre qu'à une partie continue de la chaîne d'exécution.

Disposant de la chaîne d'exécution disponible en figure 2.1, nous avons séparé l'application en six modules :

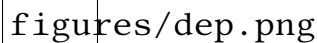


figures/chaine.png

FIGURE 2.1 – Chaîne d'exécution de l'application

- LexerParser : Ce module contient l'ensemble du code permettant de générer l'**AST** Objet MiniJaja, *id est* l'AST composé des classes que nous avons défini en Java correspondant à chaque opérande.
- LexerParserJJC : Ce module contient l'ensemble du code permettant de générer l'**AST** Objet MiniJaja, *id est* l'AST composé des classes que nous avons défini en Java correspondant à chaque opérande.
- Memory : Ce module contient l'ensemble des classes et fonctions permettant la gestion de la mémoire, donc de la pile et du tas.
- MiniJaJa : Ce module contient les classes correspondantes aux différentes instructions/opérandes disponibles en MiniJaja. Il contient également les visiteurs qui y sont liés.
- JaJaCode : Ce module contient les classes correspondantes aux différentes instructions/opérandes disponibles en JaJaCode. Il contient également les visiteurs qui y sont liés.
- GUI : Ce module contient les classes et ressources nécessaires au bon affichage de l'interface graphique et permettant l'interaction de l'utilisateur avec l'application.

Ces six modules respectent les règles de dépendances démontrées dans la figure 2.2



figures/dep.png

FIGURE 2.2 – Dépendances des différents modules définis

3 Lexer & Parser

La première étape d'un compilateur/interpréteur, indépendamment de la suite choisie par l'utilisateur ou du langage, est la lecture et l'analyse du code source afin d'en extraire un AST.

Pour cela, nous avons utilisé la librairie ANTLR (ANother Tool for Language Recognition), qui permet de générer automatiquement le code Java pour un Lexer et un Parser à partir d'un fichier de grammaire.

Pour simplifier l'utilisation de ANTLR, un petit script shell a été écrit permettant de générer et de placer les Lexers/Parsers au bon endroit dans notre arborescence de fichier.

Une fois le Lexer et le Parser généré, il ne restait plus qu'à convertir l'AST Antlr en AST utilisant notre propre architecture de classe. Ce travail est pris en charge par la classe Converter.

Le projet nécessitant de pouvoir reconnaître deux langages différents (tout du moins si l'on souhaite pouvoir interpréter directement du JaJaCode), cette structure de classe a dû être dédoublée, pour le MiniJaJa puis le JaJaCode.

3.1 MiniJaJa

3.1.1 Grammaire

Si une grammaire initiale était fournie, un certain nombre de modification y ont été apportées, que ce soit pour rendre la grammaire LL(1) ou pour simplifier la génération de l'arbre syntaxique.

On peut citer :

Les listes d'instructions et de déclarations

Dans un souci de simplicité, la règle de reconnaissance des listes de déclarations est passée de

```
1 decls : decl decls -> ^(DECLS decl ';' decls) | vnil
```

à

```
1 decls : (decl ';' )* -> ^(DECL decl*)
```

Cette modification s'applique aussi aux listes d'instructions, de déclaration de variables uniquement et même d'entêtes.

Les expressions

Afin de supprimer la récursivité à gauche les règles de grammaire correspondantes aux expressions ont été réécrites en

```
1 exp : ('!'^ exp1 | exp1) ('&&'^ exp1 | '||'^ exp1)*
2 ;
3
4 exp1 : (exp2) ('=='^ exp2 | '>'^ exp2)*
5 ;
6
7 exp2 : ('-'^ terme | terme) ('+'^ terme | '-'^ terme)*
8 ;
9
10 terme : fact ('*'^ fact | '/'^ fact)*
11 ;
12
13 fact : ident('[exp]' -> ^(TAB ident exp) | '(' listexp ')') -> ^(
14     | BOOL
15     | ent
16     | '(' exp ')') -> exp
17 ;
```

Ajout de print et printS

Bien que cela ne soit pas demandé dans le sujet, nous avons également ajouté deux instructions : print et printS qui affichent respectivement une valeur et un String. Cela se traduit par les règles de grammaire suivantes.

```
1 instr : 'print' '('(exp-> ^('print' exp) | String -> ^(PRINTS
2     String) ) ')';
3
4 String : '"' (.)* '"';
```

Nota Bene

Vous trouverez la grammaire complète en annexe C.

3.1.2 AST

Au delà de la simple analyse du code source du programme qui lui est passé, ANTLR permet également la construction d'un Arbre Syntaxique Abstrait suivant des règles prédéfinies dans la grammaire. Dans le cadre de ce projet, les principales règles sont les suivantes :

Déclaration de fonction

Lors de la déclaration d'une fonction, l'arbre syntaxique se construit de la façon suivante :

- Ajout d'une racine "METHOD"
- Ajout d'un fils Type
- Ajout d'un fils ident
- Ajout d'un fils pour chaque paramètre
- Ajout d'un fils pour la liste de déclaration de la fonction
- Ajout d'un fils pour la liste d'instructions de la fonction

Par exemple, pour le code :

```
1 int somme(int a, int b){  
2     int c;  
3     return a;  
4 };
```

Listing 3.1 – Code exemple en MiniJaJa - Déclaration de fonction

L'AST généré est visible en figure 3.1



figures/ASTMeth.png

FIGURE 3.1 – AST généré pour le code exemple - Déclaration de fonction

Déclaration de variable

Lors de la déclaration d'une variable, l'arbre syntaxique se construit de la façon suivante :

- Ajout d'une racine "VAR"
- Ajout d'un fils Type
- Ajout d'un fils ident
- Ajout d'un fils pour la valeur d'initialisation

Par exemple, pour le code :

```
1 int a = 5;
```

Listing 3.2 – Code exemple en MiniJaJa - Déclaration de variable

On obtient l'AST visible en figure 3.2



figures/ASTVar.png

FIGURE 3.2 – AST généré pour le code exemple - Déclaration de variable

Conditionnelle

Pour construire le noeud de l'arbre syntaxique correspondant à un if, ANTLR suit la méthode suivante :

- Ajout d'une racine "SI"
- Ajout d'un fils correspondant à la condition
- Ajout d'un fils correspondant à la liste d'instructions du then
- Ajout d'un fils correspondant à la liste d'instructions du else

Et, par exemple, pour :

```
1 if (true){  
2     print("Vrai");  
3 } else {  
4     print("Faux");  
5 };
```

Listing 3.3 – Code exemple en MiniJaJa - Conditionnelle

On obtient l'AST présent en figure 3.3



figures/ASTIf.png

FIGURE 3.3 – AST généré pour le code exemple - Conditionnelle

Boucle while

Pour traiter une boucle while, le raisonnement est très proche de celui utilisé pour le if :

- Ajout d'une racine "TANTQUE"
- Ajout d'un fils correspondant à la condition
- Ajout d'un fils correspondant à la liste d'instructions

Et, pour le code :

```
1 while(true){  
2     print("Woops !");  
3 };
```

Listing 3.4 – Code exemple en MiniJaJa - Boucle While

On obtient l'AST de la figure 3.4



figures/ASTWhile.png

FIGURE 3.4 – AST généré pour le code exemple - Boucle While

Expression

Pour traiter une expression, ANTLR va simplement ajouter une racine correspond à l'opérande, puis un fils par "paramètre" de cette opérande. Néanmoins, les opérations n'étant pas toute de même priorité, il peut être intéressant de regarder un exemple.

Typiquement, pour le code :

```
1 (- (a * 6 + 7) - 5) * 83 - 2
```

Listing 3.5 – Code exemple en MiniJaJa - Expression

On obtient l'AST de la figure 3.5

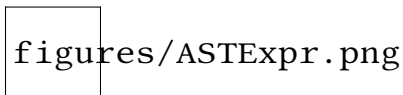


FIGURE 3.5 – AST généré pour le code exemple - Expression

3.1.3 Converter

Afin de pouvoir travailler plus efficacement sur les différents noeuds de l'AST, il était plus simple de les convertir vers nos propres objets plutôt que d'utiliser des noeuds standards dont seul un champ String change. Pour cela, nous utilisons la classe *Converter* qui s'occupe de réaliser des switch case sur ces champs String des noeuds afin de pouvoir instancier les bons objets parmi l'architecture définie dans le module MiniJaJa.

3.2 JaJaCode

Un travail équivalent a été réalisé pour pouvoir interpréter le JaJaCode. Néanmoins, un nombre beaucoup moins conséquent de modification ont été apportées à la grammaire et il ne semble pas intéressant de les détailler ici, les changements étant en substance les mêmes que pour le MiniJaJa.

4 Gestion de la mémoire

Dans notre application, le module Memory est le module regroupant les différentes structures et fonctions qui vont servir à stocker et manipuler les différents éléments composants la mémoire (variables, identifiant de fonctions, tableaux, constantes ...) pendant la vérification, la compilation et l'interprétation des programmes.

4.1 Table des Symboles

Une Table Des Symboles (TDS) est une centralisation des informations rattachées aux identificateurs d'un programme informatique : c'est une fonction accélératrice de compilation dont l'efficacité dépend de sa conception.

Elle centralise les informations sémantiques des identificateurs :

- Pour une variable : nom, type ...
- Pour un tableau : nom, dimension ...
- Pour une fonction : nom, type (de la valeur de retour), paramètres ...

Lors de la définition d'un identificateur, on insère une nouvelle entrée dans la TDS. Dans le cadre de ce projet, la déclaration d'un identificateur est à priori unique.

Identification

L'identification est le processus de recherche d'un identificateur dans la TDS. On utilise une table de hachage, indexée par le nom de l'identificateur, afin de permettre un accès rapide aux informations, contenues dans la TDS, concernant ce dernier.

À chaque utilisation d'un identificateur dans le programme, on cherche l'entrée correspondante dans la TDS. Si le processus d'identification échoue, cela signifie que l'identificateur n'a pas été déclaré au préalable dans le programme.

Table de hachage

Une table de hachage est une structure de données permettant une association clé-valeur (tableau associatif), elle garantit un accès rapide, et en temps constant, aux identificateurs contenus dans la TDS.

Il s'agit d'un tableau ne comportant pas d'ordre (contrairement à un tableau ordinaire qui est indexé par des entiers). On accède à chaque valeur du tableau par sa clé grâce à une fonction de hachage qui transforme cette dernière en une valeur de hachage (un nombre) indexant les éléments de la table.

N'importe quelle fonction transformant un identificateur en un nombre peut servir de fonction de hachage mais elle ne sera pas nécessairement efficace. En effet, il peut arriver que plusieurs clés différentes se retrouvent associées à une même valeur de hachage, ce qui donne lieu à une collision.

Le nombre de collisions générées par la fonction de hachage qui ne pourra plus garantir un accès direct aux valeurs qu'elle contient (un index pouvant correspondre à plusieurs valeurs différentes).

Pour des performances optimales, il est donc nécessaire d'utiliser une fonction de hachage qui génère le moins de collisions possibles (idéalement aucune). Nous avons décidé d'implémenter l'algorithme de *DJB2* qui est considéré par beaucoup comme l'une des fonction de hachage les plus efficaces.

Portée d'un identificateur

La portée d'un identificateur correspond à sa zone d'existence (le bloc dans lequel il a été déclaré). Ainsi, plusieurs identificateurs portant un nom identique peuvent coexister et référencer des éléments différents.

La table des symboles globale (correspondant à la portée globale de la classe) contient, en plus de ses définitions, d'autres tables des symboles correspondant chacune à une sous-portée (une pour la fonction *main* et une pour chaque autre fonction).

Alors, à la lecture du programme :

- Si on rencontre un identificateur dans une déclaration, on vérifie son absence dans la TDS correspondante à la portée courante.
- Si on rencontre un identificateur dans une partie exécutable, on vérifie son existence dans la TDS correspondante à la portée courante. Si l'identification échoue et que la portée courante correspond à une méthode, alors on vérifie l'existence de l'identificateur dans la TDS. Si cette identification échoue, alors il y a erreur.

```
1 class C { // Début de la portée globale
2     int x = 0;
3     int f(int p) { // Début de la portée de f
4         int c = 6;
5         return p+c ;
6     } ; // Fin de la portée de f
7
8     main { // Début de la portée du main
9         int y = 2;
10        x = 3;
11        y += f(x) ;
12    } // Fin de la portée du main
13 }
```

Listing 4.1 – Exemple de définition de portée

Implémentation et Utilisation

Dans le code, cette notion de Table Des Symboles se traduit par un ensemble de classes : *Scope*, *SymbolTable* et des classes ...*Symbol*.

Les classes *Symbol* correspondent simplement à l'objet qui sera stockée dans notre TDS pour chaque élément du programme. Si tous ont en commun de mémoriser l'adresse, le nom et le type de l'élément, il en existe trois variétés :

- *MethodSymbol* : Représente une méthode dans la TDS. Elle stocke en plus une liste de paramètres et une référence vers l'objet correspondant à la méthode pour l'interprétation.
- *TabSymbol* : Représente un tableau dans la TDS. Elle stocke en plus, si on la connaît, la taille du tableau.
- *VarSymbol* : Représente une variable/constante dans la TDS. Elle ne dispose d'aucune particularité.

Pour stocker l'adresse d'un élément dans la pile, nous avons décidé plutôt de stocker un décalage par rapport à l'adresse de début du Scope. Cela sera traité plus en détail dans la partie sur l'interprétation MiniJaJa.

La classe *Scope* n'est alors qu'une classe qui retient son id, son Scope parent, son adresse actuelle dans la pile (le décalage cumulé des déclarations du scope) et une HashTable des Symbols contenus dans le Scope.

La classe *SymbolTable* revient alors simplement à une HashTable de Scope, un pour chaque portée.

Lors de l'exécution de l'application, dans le cas de l'interprétation du MiniJaJa, la table des symboles est générée lors du contrôle de type puis passée à l'interpréteur. Dans le cas de l'interprétation du JaJaCode, elle est calculée à la volée.

4.2 La pile

La pile sert à la mémorisation des variables ainsi que leur valeur, leur type (ex : entier), leur nature (ex : méthode) et leur identifiant. Ces éléments sont représentés sous la forme d'un quadruplet, insérés dans la pile au fur de leur rencontre durant l'interprétation. Cette structure de donnée n'est utilisée que pendant l'interprétation étant une structure dynamique.

La pile est définie dans la classe *MyStack*, et représentée par une liste chaînée sur laquelle on effectue les opérations suivantes :

- Ajout d'un élément : via la fonction `push(Quadruplet)` qui va ajouter un élément au sommet de la pile
- Retrait d'un élément : via la fonction `pop()` qui retourne le sommet de la pile tout en le retirant
- Permutation : via la fonction `swap()` qui inverse l'ordre des deux éléments les plus récents dans la pile. Cette méthode servira dans le retrait des variables ainsi que pour retourner les résultats des méthodes
- Affichage : via la fonction `toString()` qui nous permettra d'apercevoir l'état de notre pile, par exemple lors de l'exécution pas à pas
- Affectation d'une valeur : via la fonction `affectVal(ident, valeur)` qui donne une valeur à une variable définie par son identificateur en mettant cette valeur dans le champ VAL du Quadruplet correspondant
- Déclaration : via les fonction `decl*Sorte*(ident, valeur, nature, sorte)` qui, pour chaque sorte de déclaration, construisent le quadruplet et l'ajoute à la pile

Un exemple d'affichage de la pile pendant l'exécution pas à pas d'un programme JaJaCode peut être vu dans la figure 4.1.

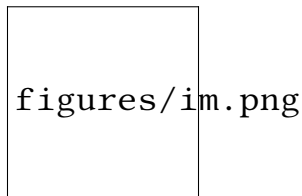


FIGURE 4.1 – Exemple d’affichage de la pile

4.3 Le tas

Le tas est la composante de la mémoire qui vient compléter la pile. Il représente un bloc mémoire tel qu’on le connaît. Il stocke en temps normal toutes les informations relatives aux objets dynamiques, et dans notre cas, relatives uniquement aux tableaux, étant donné que MiniJaJa ne gère pas les objets.

Il est composé de trois éléments :

- Un tableau d’entier qui représente le bloc mémoire effectif
- Un tableau de ‘Blocs’ qui sauvegarde la décomposition du bloc mémoire
- Une table de hachage afin de stocker les références

Le tas se base sur quatre fonctionnalités principales : l’allocation, la dés-allocation, l’accès et enfin la réorganisation.

L’allocation

Lors de l’allocation d’un tableau de taille t , la gestion de la mémoire va chercher s’il existe un bloc de mémoire continu pouvant accepter un tableau de cette taille.

Si un tel bloc existe, la mémoire correspondante est allouée et l’espace restant du bloc est découpé en bloc de taille correspondante à une puissance de 2. Une référence est également ajoutée dans la table de hachage.

Si un tel bloc est introuvable, la gestion de la mémoire devrait lancer la procédure de Garbage Collector (ramasse-miettes). Néanmoins, cette fonctionnalité n’étant pas implémentée à l’écriture de ce rapport, il est impossible de commenter ici son fonctionnement. L’application se contente actuellement de lever une exception "Full Heap".

La dés-allocation

La dés-allocation, qui survient en cas de synonymie ou de sortie de bloc, revient simplement à décrémenter de un le compteur de référence du bloc correspondant et de vérifier s’il est toujours référencé. Si ce n’est pas le cas, il est supprimé de la table de hachage.

Néanmoins, aucune libération de mémoire n’est effectué à ce moment, cela se déroule durant l’appel au Garbage Collector.

Accès

Lors de l'accès à l'indice i du tableau t , l'interpréteur va chercher dans la pile l'adresse du tableau dans le tas puis accéder à l'espace mémoire d'adresse $@t + i$ si cela est possible. Dans le cas contraire, une exception `IndexOutOfBoundsException` est levée.

Réorganisation

La réorganisation correspond à la dernière étape du Garbage Collector. Elle consiste à condenser l'ensemble des blocs alloués au début de notre espace mémoire et de reconstruire l'espace disponible en blocs de puissance de 2. Encore une fois, le Garbage Collector n'étant pas encore implémenter, il est impossible de documenter la mise en oeuvre technique de cette fonction.

5 Gestion du MiniJaJa

Comme cela a pu être dit précédemment dans ce rapport, nous utilisons tout un ensemble de classes pour nous permettre d’instancier un AST correspondant au programme MiniJaJa que nous traitons.

5.1 Architecture de classes

L’architecture que nous avons définie suit les diagrammes de classes présents en figure 5.1 et 5.2.


 figures/Figure41.jpg

FIGURE 5.1 – Diagramme de classes de l’architecture MiniJaJa


 figures/Figure42.jpg

FIGURE 5.2 – Diagramme de classes de l’architecture MiniJaJa

Nous utilisons ensuite un pattern Visiteur pour nous permettre d’exécuter des fonctions sur chacune de ces classes. Nous avons défini 3 visiteurs : TypeChecker, Interpreter et Compiler.

5.2 Contrôleur de type

Le premier visiteur à être exécuté est TypeChecker. Il effectue des contrôles sémantiques sur l’AST généré et renvoyé par le Converter. Il est également chargé de construire la table des symboles.

En d’autres termes, il vérifie à la fois le type des expressions mais aussi le nombre de paramètres d’une fonction.

Pour construire la table des symboles, l’instance du visiteur se souvient de l’id du scope courant et l’utilise afin de créer une référence à chaque déclaration rencontrée.

5.3 Interpréteur

Le second, et plus imposant, visiteur est Interpreter. Il effectue l’interprétation, aussi bien directe que pas à pas d’un AST MiniJaJa.

Si l’idée derrière un interpréteur est simple à appréhender, l’implémentation dispose de plusieurs particularités qu’il faut détailler.

Interprétation basique

Pour l'interprétation simple, *id est* l'interprétation directe du MiniJaJa à l'exception des fonctions, le visiteur n'a besoin que d'une référence au tas et à la pile.

En effet, il va rajouter/accéder/modifier les éléments dans la pile ou le tas en fonction de quel noeud est rencontré, mais aucun traitement supplémentaire n'est nécessaire.

Typiquement, le code de l'interprétation de l'instruction + est de la forme :

```
1 final int lExpVal = (int) mjjAdd.getLeftExp().accept(this);
2 final int rExpVal = (int) mjjAdd.getRightExp().accept(this);
3 //lExpVal + rExpVal = ?
4 return lExpVal + rExpVal;
```

Listing 5.1 – Code de l'interprétation de Add

Interprétation des fonctions et Scope

Néanmoins, lorsque nous avons rajouté les fonctions au spectre des instructions qui peuvent être gérées par l'interpréteur, un problème est apparu. Il a fallu trouver un moyen de gérer les différents scopes et donc également le stockage des adresses des variables dans la TDS, qui est créée statiquement et reste inchangée durant l'interprétation.

Pour cela, nous avons décidé de passer par deux changements de la base de code.

Le premier, survenu directement dans la table des symboles, correspond à remplacer le stockage de l'adresse d'un symbole par le stockage du décalage de ce symbole dans la pile par rapport au début du scope. Par exemple, pour le code

```
1 int f(){
2     int a;
3     int b;
4 }
```

Alors, le décalage de a est 0 tandis que celui de b est 0+taille(a). En pratique, et étant donné notre structure de pile, la fonction taille() renvoie toujours 1.

Le second changement est lui intervenu dans le visiteur. En effet, pour pouvoir utiliser le décalage en lieu et place de l'adresse, il était nécessaire de se souvenir de l'adresse de début du scope. Pour cela, nous utilisons une Pile de Pair<String, Integer> qui se souvient dans l'ordre des derniers Scopes dans lesquels nous sommes rentrés et quelle était la taille la pile à ce moment là.

Ces deux changements permettent alors d'utiliser efficacement les scopes et il serait même possible, si cela était dans la spécification du langage, de définir des variables dans des boucles while et qu'elle ne soient pas accessibles d'en dehors.

Interprétation pas à pas

Une autre des difficultés rencontrées concerne l'interprétation pas à pas.

En effet, il fallait pour cela être capable de pauser l'interprétation avant chaque nouveau noeud. Pour pallier ce problème, nous avons utilisé les mots clés **wait** et **synchronized** de Java ainsi que les Thread, et donc, par nécessité due à JavaFX, les Services et Tasks.

Lors de l'interprétation pas à pas, l'interprétation se fait donc sur un Thread différent de l'interface et chaque clic de l'utilisateur sur `nextStep` provoque un `notify` de ce Thread et donc l'exécution du prochain noeud.

5.4 Compilateur

Enfin l'application devant permettre de compiler un code source MiniJaJa, il était nécessaire de proposer un visiteur permettant la compilation à partir de l'AST.

Pour réaliser la compilation, nous nous sommes simplement basés sur les règles de compilation vues en cours et qui n'ont pas nécessité de modification outre-mesure, si ce n'est du à nos quelques modifications de la grammaire.

6 Gestion du JaJaCode

Tout comme le MiniJaJa, le JaJaCode dispose d'une architecture de classe qui lui est propre que cela soit pour simplifier la compilation du MiniJaJa ou directement sa compilation.

6.1 Architecture de classes

L'architecture que nous avons définie suit le diagramme de classe présent en figure 6.1.


 figures/JJC.jpg

FIGURE 6.1 – Diagramme de classes de l'architecture JaJaCode

Nous utilisons ensuite ici aussi un pattern Visiteur pour nous permettre d'exécuter des fonctions sur chacune des classes. Il y a deux visiteurs JaJaCode : Interpreter et Printer.

6.2 Interpreter

Le langage JaJaCode étant plus simple (*id est* plus bas niveau) que le MiniJaJa, son interprétation est elle aussi simplifiée.

En effet, il n'est ici plus question de gérer ni scope ni TDS et toutes les variables sont directement mises en pile. De fait, toutes les variables présentes en pile sont accessibles depuis le point actuel du programme (à la différence du MiniJaJa) et il suffit de "remonter" la pile pour trouver l'objet qui nous intéresse.

De même, le JaJaCode ayant une structure d'une instruction par ligne, il est très simple d'implémenter une interprétation pas à pas, il nous suffit de stocker le numéro de la ligne actuellement interprété.

6.3 Printer

De même, le Printer ne présentait aucune difficulté et consistait juste à concrétiser les différents objets manipulés jusque lors.

7 Interface Homme-Machine

Conformément aux exigences fonctionnelles, notre application propose les contrôles suivant :

- Ouvrir un Fichier
- Créer un Fichier
- Fermer un Fichier
- Sauvegarder un Fichier
- Sauvegarder un Fichier Sous ...
- Sauvegarder tous les Fichiers
- Compiler un Fichier MiniJaJa
- Compiler & Interpréter un Fichier MiniJaJa
- Interpréter un Fichier MiniJaJa
- Interpréter Pas à Pas un Fichier MiniJaJa
- Éditer un Fichier ouvert

7.1 Différents onglets

La première fonctionnalité ‘challengante’ de l’interface graphique a été de gérer l’existence de différents onglets d’édition de code, pouvant pointer chacun vers un fichier différent, qu’il soit MiniJaJa ou JaJaCode. Il était donc nécessaire de modifier les menus contextuels dépendamment de l’extension du fichier ouvert dans l’onglet actif.

Pour l’ensemble de ces problématiques, nous avons mis en place une classe `TabFile`, héritant de la classe `JavaFX tab`, et qui nous permet de gérer nous-même plus profondément les onglets ouverts.

Pour l’adaptation des menus contextuels au fichier ouvert, nous avons mis en place un listener sur la liste de nos `TabFile`, permettant ainsi de détecter chaque changement de focus et d’adapter l’affichage.

De même, un listener a été placé sur la partie éditable de l’onglet afin de détecter toute modification effectuée par l’utilisateur et de lui proposer la sauvegarde du fichier.

7.2 Coloration Syntaxique et Breakpoints

Afin de mettre en place la coloration syntaxique du code éditable et les breakpoints, nous nous sommes tourné vers une bibliothèque JavaScript déjà existante, `CodeMirror.js`, que nous avons inclus dans une `WebView`.

Cela nous permet de profiter de la souplesse du combo HTML/CSS/JS en terme d’édition et d’affi-

chage de contenu textuel sans avoir à repenser la roue. De plus, CodeMirror propose la possibilité d'éditer les fichiers 'mode', responsables de la coloration syntaxique. Nous avons donc créé un fichier 'mode' pour chacune des extensions possible : mjj et jjc.

7.3 Console et Output

Afin de permettre à l'utilisateur d'accéder aux informations renvoyées par le compilateur, interpréteur ou même le programme, une console a été mise en place dans l'interface.

Pour l'utiliser, nous avons redirigé les `System.out` et `System.err` vers un `Stream` auquel est associé un listener pour mettre à jour la console conjointement.

8 Tests

Afin de rester sur un cap cohérent durant l'ensemble du projet, une des premières tâches réalisée a été de créer un ensemble de fichiers de tests, permettant d'effectuer des tests d'acceptations.

On dispose de tests qui fonctionnent et disposent d'un résultat attendu, mais aussi de fichiers spécifiques qui sont pensés pour échouer face à un test bien défini.

On peut citer par exemple :

```
1 class C {
2     int value = 5;
3     main
4     {
5         int data = 3;
6         int sum = 0;
7         boolean b = false;
8         if (value > 0) {
9             data++;
10        };
11        if (data > 0) {
12            value = value - 1;
13        };
14        if (data == value) {
15            sum = value + data;
16            b = true;
17        };
18        if (b && (sum > 0)) {
19            value = 0;
20        };
21        if (!b || (sum == 0)) {
22            data = 0;
23        };
24        print("Sum : ");print(sum);
25        print("Data : ");print(data);
26        print("Value : ");print(value);
27        // Resultats attendus :
28        // sum = 8
29        // data = 4
30        // value = 0
31    }
32 }
```

Listing 8.1 – Test If

```

1 class C{
2     int a = 8;
3     int somme(int a){
4         print("Call Somme! \n");
5         if (a>5){
6             return somme(a-1);
7         };
8         return a;
9     };
10    main{
11        int c = 5;
12        print(somme(a));
13    }
14 }

```

Listing 8.2 – Test Récursif

```

1 class fail{
2     main
3     {
4         inexistantType inexistant = 5;
5     }
6 }

```

Listing 8.3 – Test Fail Syntaxique Type

```

1 class fail{
2     main
3     {
4         int nombre = 10;
5         boolean b = nombre;
6     }
7 }

```

Listing 8.4 – Test Fail Sémantique Type

9 Bilan

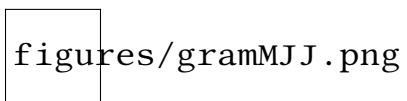
Nous sommes satisfait du résultat technique. En effet, les différentes exigences fournies par le cahier des charges sont respectées et nous sommes content du travail réalisé.

Néanmoins, on peut déplorer la différence de cadence entre la R1 et la R2. Si les objectifs de la R1 ont été atteints sans grande difficulté, la fin du projet s'est déroulé dans la peur de la deadline.

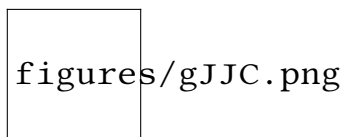
Cela est dû à plusieurs facteurs, parmi lesquels une baisse du niveau d'investissement moyen, une concentration accrue envers les tests, et quelques difficulté techniques liées notamment à l'interprétation des méthodes et à l'interprétation pas à pas.

Il est également appréciable d'avoir pu mettre à profit les compétences acquises en cours de Compilation et d'avoir se rendre compte du fonctionnement concret d'un compilateur.

A Grammaire MiniJaJa initiale



B Grammaire JaJaCode initiale



C Grammaire MiniJaJa modifiée

```
1 grammar MiniJaJa;
2
3 options {
4     backtrack    = false;
5     k            = 1;
6     output       = AST;
7     ASTLabelType = CommonTree;
8 }
9
10 tokens {
11     VAR;
12     METH;
13     PARAM;
14     CST;
15     VARS;
16     BLOCK;
17     VEC;
18     CALL;
19     TABLEAU;
20     CLASSE;
21     IDENT;
22     DECLS;
23     INSTRS;
24     VIDE;
25     AFFECTATION;
26     INCREMENT;
27     SOMME;
28     TAB;
29     RETOUR;
30     SI;
31     TANTQUE;
32     APPELI;
33     NBRE;
34     LISTEXP;
35     APPELE;
36     PRINTS;
37 }
38
39 @header{package fr.femtost.disc.m1comp4.lexerParser.AntlrGenerated
    ;}
```

```

40
41 @lexer::header {package fr.femtost.disc.m1comp4.lexerParser.
    AntlrGenerated;}
42
43
44 axiom : fichier EOF-> fichier
45 ;
46
47 fichier : classe*
48 ;
49
50 classe : 'class' ident '{'decls methmain'}' -> ^(CLASSE ident decls
    methmain)
51 ;
52
53 ident : IDF -> ^(IDENT IDF)
54 ;
55
56 decls : (decl ';'*) -> ^(DECLS decl*)
57 ;
58
59 decl : typemeth ident ((vexp -> ^(VAR ident typemeth vexp) | '['
    exp ']' -> ^(TABLEAU ident typemeth exp)) | methode-> ^(METH
    typemeth ident methode?)) | finalvar
60 ;
61
62 varz : (vars ';'*) -> ^(VARS vars*)
63 ;
64
65 vars : typemeth ident (vexp -> ^(VAR ident typemeth vexp) | '[' exp
    ']' -> ^(TABLEAU ident typemeth exp))
66 | finalvar
67 ;
68
69 finalvar : 'final' type ident vexp -> ^(CST type ident vexp)
70 ;
71
72 vexp : '=' exp -> exp
73 | -> VIDE
74 ;
75
76 methode : '(' (type ident (',' type ident)*)? ')' '{'varz instrs
    '}' -> ^(PARAM type ident)* varz? instrs?
77 ;
78
79 methmain : 'main' '{'varz instrs'}' -> ^('main' varz? instrs?)
80 ;
81
82 instrs : (instr ';'*) -> ^(INSTRS instr*)
83 ;

```

```

84
85 instr : ident(
86     ('=' exp -> ^(AFFECTATION ident exp) | '++' -> ^(INCREMENT
87     ident) | '+=' exp -> ^(SOMME ident exp))
88     | '(' listexp ')' -> ^(APPELI ident listexp?)
89     | '[' exp ']' ('=' exp -> ^(AFFECTATION ^(TAB ident exp) exp)
90     | '++' -> ^(INCREMENT ^(TAB ident exp) ) | '+=' exp -> ^(SOMME ^(
91     TAB ident exp) exp))
92     )
93     | 'return' exp -> ^(RETOUR exp)
94     | 'if' exp '{' instrs '}' ('else' '{' instrs '}' )? -> ^(SI exp
95     instrs instrs?)
96     | 'while' '(' exp ')' '{' instrs '}' -> ^(TANTQUE exp instrs?)
97     | 'print' '(' (exp -> ^('print' exp) | String -> ^(PRINTS String)
98     ) ')'
99 ;
100
101 listexp : (exp (',' exp)*)? -> ^(LISTEXP exp*)
102 ;
103
104 exp : (!'^ exp1 | exp1) ('&&'^ exp1 | '||'^ exp1)*
105 ;
106
107 exp1 : (exp2) ('=='^ exp2 | '>'^ exp2)*
108 ;
109
110 exp2 : ('-'^ terme | terme) ('+'^ terme | '-'^ terme)*
111 ;
112
113 terme : fact ('*'^ fact | '/'^ fact)*
114 ;
115
116 fact : ident('[exp]' -> ^(TAB ident exp) | '(' listexp ')') -> ^(
117     APPELE ident listexp) | -> ident)
118     | BOOL
119     | ent
120     | '(' exp ')' -> exp
121 ;
122
123 typemeth : VOID
124     | type
125 ;
126
127 type : TYPE
128 ;
129
130 ent : INT -> ^(NBRE INT)
131 ;
132
133 VOID : 'void'

```

```

128 ;
129
130 TYPE : 'int' | 'boolean'
131 ;
132
133
134 BOOL : 'true' | 'false'
135 ;
136
137 INT      : '0'..'9'+
138 ;
139
140 IDF      : ('a'..'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9'|'_' )*
141           //Commence par une lettre (minuscule ou majuscule)
142 ;
143
144 String : '"' (.)* '"'
145 ;
146 COMMENT : '/*' ( options {greedy=false;} : . )* '*/' {$channel=
147           HIDDEN;} //Commentaire sur plusieurs lignes
148 ;
149 ILCOMMENT : '// ' ~( '\r' | '\n' )* {$channel=HIDDEN;} //Inline
150           comments
151 ;
152 WS      : ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;} //
153           Whitespace

```