Université de Franche-Comté
Master informatique - 1ère année

Rapport de projet
Compilation et Génie Logiciel

Interprétation et compilation en programmation objet

Auteurs :

Awa Diallo
Guzal Khusanova
Virgil Manrique
Mohammad Nauval
Jérémy Navion
Mehdi Zemmoura

Année universitaire 2017 - 2018

Sommaire

Remerciements

Au terme de ce travail, nous souhaitons remercier pour leur aide et leurs conseils xxxxxxxxxxxxx, [titres].

Introduction

Dans le cadre de notre formation universitaire en première année de Master informatique, nous sommes amenés à réaliser un projet de compilation.

Contexte : pluridisciplinaire (COMP / GL) ; méthode agile par six ; rapport technique ; rapport GL séparé...

Les objectifs généraux de ce projet...

Nous débuterons par... Nous enchaînerons par..., et nous poursuivrons par... Pour terminer, nous ferons les bilans technique de ce projet.

Partie 1

Le projet de compilation

1 Le sujet

Objectifs détaillés...

2 Le cahier des charges

Contraintes fonctionnelles / non fonctionnelles ; complète le sujet...

3 Décomposition et architecture du logiciel

Découpage en plusieurs parties : GUI, analyseur, compilateur, etc.

Partie 2

Développement

Nous étudierons d'abord... Nous expliquerons..., nous évoquerons ensuite..., nous poursuivrons par..., et nous terminerons par...

1 Le contrôleur de type

2 L'analyseur

Dans la partie analyser il s'agit du module « Analyseur », qui contient des informations sur la grammaire, le parseur Minijaja et des utilitaires qui facilite la gestion du langage qui vont être plus détaillé prochainement.

Alors, premièrement, nous nous familiarisons avec des sous-parties (fichiers) intéressantes de l'analyseur et nous nous concentrons les principaux choix de réalisation. Ensuite la discussion porte sur des traitements des erreurs (des exceptions). Nous finissons par des tests réalisés.

2.1 La grammaire et parseur MiniJaja

Le fichier contient la grammaire MiniJaja décrit dans le langage JJTree . Ensuite à partir du fichier MiniJajaGrammar.jjt le parseur du Minijaja (MiniJajaGrammar.java), les nœuds arbres syntaxique abstraits (AST) et un visiteur de la grammaire sont génères à l'aide de compilateur javacc/jjtree.

Nous avons choisi d'utiliser les compilateurs javacc et jjtree qui permet de générer le parseur à partir de grammaire donné, car écrire notre propre parseur est plus complexe et plus couteux en temps. Alors la grammaire donnée à l'entrée aux compilateurs est sous forme BNF et en LL(1) (grammaire non-récursive gauche et non-ambigüe). Pour rendre la grammaire MiniJaja proposé dans le support du cours de Compilation en LL(1) nous avons utilisé les propriétés du langage Jjtree.

Prenons l'exemple de déclaration de variable et méthode :

decl → var | methode

var → typemeth ident vexp | typemeth ident [exp] |final type ident vexp

methode → typemeth ident (entêtes) vars instrs

Il faut noter que "typemeth ident" rend la grammaire ambiguë. Pour résoudre ce problème dans les paramètres de jjtree c'est possible de mettre lookhead = 3, dans ce cas avant prendre la décision pour la nœud déclaration pareurs analyse trois tokens qui suits d'abord. Mais nous avons décider de rendre la grammaire en LL(1). Alors le problème mentionné dessus est résolu à la manière suivante :

```
void Declaration() #void:
{}
{
  Constant()
| MethodType() Identifier() (Method() | Variable())

}
void Method() #void:
{}
{
   <OPEN_PARENTHESIS> Parameters() <CLOSE_PARENTHESIS>
  <OPEN_CURLY_BRACKET> Variables() Instructions()  <CLOSE_CURLY_BRACKE
}
```

Remarques : Le noeud AST #void ne sont pas généré

Le noeud Method a 5 enfants, il va chercher 5 dernière nœud avant l'appele de #Method(5). Alors c'est Instructions, Variables, Parametres, Identifier, MethodType

2.2 Visiteurs de la grammaire de Minijaja

Une classe MiniJajaGrammarVisitor.java est un visiteur de l'arbre syntaxique de MiniJaja généré. Deux visiteurs qui hérite de ce visiteur ont été développes afin de réduire le code de certaine implémentations de visiteur. Par exemple, classe privée RevokeVisitor de la classe MiniJajaCompiler a surchargé 4 méthodes de DefaultMiniJajaGrammarVisitor, pourtant la visiteur de MiniJaja contient 44 méthodes à surchargés.

2.3 Traitement des erreurs

2.4 Jeu de test

Pour tester les cas nominaux de la grammaire nous avons écrit la classe de test MiniJajaGrammarTest. Pour cela nous avons implémenter MiniJajaGrammarVisitor. Le test de grammaire est effectué à la manière suivante :

1. Nous avons code source MiniJaja syntaxiquement correct, cela veut dire que parseur ne leve pas une exception

2. On parse le code source à l'aide de MiniJajaGrammar.java et on obtient l'arbre syntaxique.

3. Notre visiteur parcours chaque nœud de notre arbre et construit au fur à mesure un rendu.

4. Pour déterminer que le test passe nous vérifions que code source MiniJaja donné à l'entrée du visiteur = rendu du visiteur.

3 Le compilateur

Dans cette partie du rapport la discussion porte sur les modules « JajaCode » et « Compiler ». Premier contient des classes nécessaires pour la construction d'un objet JajaCode et deuxième c'est un visiteur Minjaja, qui prend à l'entrée l'arbre MiniJaja et sort le Jajacode correspondant.

3.1 Conception et traitement des erreurs

Pour la construction de JajaCode le pattern « builder » a été mis en place. JajaCodeBuilder est constructeur de JajaCode qui contient une liste des instructions de JajaCode. Le compiler les remplie au fur à mesure. Une fois les

instructions de JajaCode sont construits, le methode build() généré l'objet JajaCode qui contient la séquence des instructions JajaCodede.

De plus, pour rendre les instructions de JajaCode compréhensible aux utilisateurs (au format chaîne de caractère) la classe JajaCodeRenderer mis en place . JajaCodeRenderer utilise le pattern visiteur et hérite de l'interface JajaCodeInstructions.

MiniJajaCompiler consiste de deux visiteurs. La classe MiniJajaCompiler et sa classe privée RevokeVisitor. Le visiteur RevokeVisitor est dédié au retrait de déclaration.

MiniJajaCompiler supposé de recevoir à l'entré le code source correct de MiniJaja, cela veut dire que avant de passé par compiler l'analyseur effectue l'analyse lexicale et syntaxique et contrôleur de type vérifier le compatibilité des typages, l'adéquation des valeurs. Alors le MiniJajaCompiler ne gère pas des erreurs.

3.2 Tests

La classe test MiniJajaCompilerTest teste la compilation du code source MiniJaja vers JajaCode. Pour cela les exemples de support de cours et des travaux dirigé sont utilisés. On donne à l'entrée MiniJajaCompiler un code source MiniJaja et reçoit en sortie l'objet JajaCode. Ensuite on convertit chaque instruction de JajaCode obtenu aux chaînes de caractères à l'aide JajaCodeRenderer. Alors on vérifier bien que compilation MiniJaja correspond bien à JajaCode que nous attend.

4 L'interpréteur MiniJaja

5 La gestion de la mémoire

Dans cette partie, nous allons examiner l'implémentation de l'état mémoire dans notre projet de compilation. La mémoire est l'un des éléments importants du projet et elle sert à gérer le stockage de tous les données déclarées dans un programme Minijaja. Ces données peuvent être des variables des différents types, des tableaux, et des déclarations des méthodes.

5.1 Conception

L'état mémoire du projet est composé de trois éléments : le dictionnaire de données, le tas, et la pile. Nous allons regarder la conception de chacun de ces trois éléments.

5.1.1 Le Dictionnaire de Données

Comme son nom indique, le dictionnaire de données contient les données déclarées dans un programme Minijaja. Le dictionnaire de données est une table de hachage. Une table de hachage est une implémentation du tableau associatif, elle permet une association clé-valeur.

On peut accéder à chaque valeur du tableau par sa clé. L'accès se fait par une fonction de hachage qui transforme une clé en une valeur de hachage (un nombre) indexant les éléments de la table, ces derniers sont appelés *buckets* en anglais. Il nécessite de stocker dans les *buckets* la paire clé-valeur et pas uniquement la valeur.

Le fait de générer une valeur de hachage à partir d'une clé peut engendrer un problème d'une collision, c'est-à-dire que deux clé différents, pourront se retrouver associées à la même valeur de hachage et donc au même *bucket*. Pour résoudre ce problème, premièrement nous devons choisir une bonne fonction de hachage pour minimiser la collision. La fonction de hachage que nous avons choisi la fonction de hachage FNV1.

```
private int hash(final String key) {
    int fnv1Init32 = 0x811c9dc5;
    int fnv1Prime32 = 16777619;
    int hash = fnv1Init32;

    byte[] data = key.getBytes();
    for (byte aData : data) {
        hash ^= (aData & 0xff);
        hash *= fnv1Prime32;
    }
    return hash;
}
```

FIGURE 2.1 – Fonction de hachage FNV1

La fonction de hachage FNV1 n'est pas une fonction de hachage parfaite. Une fonction de hachage est dite parfaite si elle n'engendre aucune collision. En effet, nous avons toujours le problème de collision. Pour le résoudre, chaque case ou *bucket* contient une liste chaînée. Si deux clés se retrouvent au même *bucket*, les deux paires clé-valeur seront stockées dans cette liste.

Voici la représentation graphique de notre dictionnaire de données :

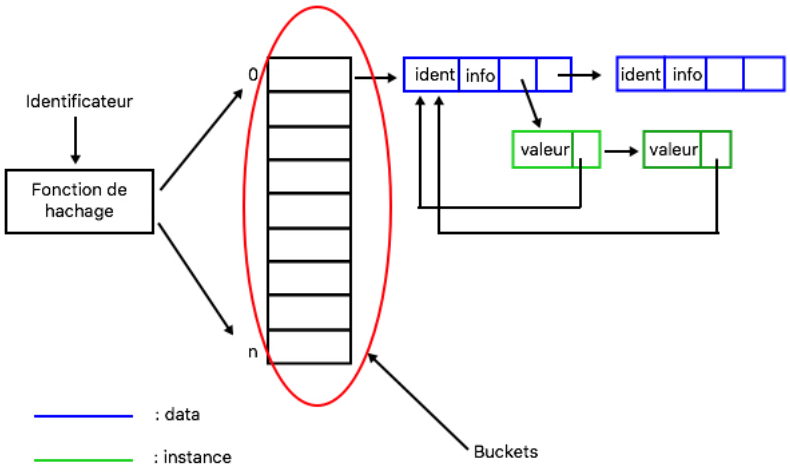

FIGURE 2.2 – Dictionnaire de données

L'image est notre dictionnaire de données. La table de hachage a pour capacité initiale **n**, c'est-à-dire qu'il contient n *buckets*. Comme mentionné avant, chaque *bucket* contient une liste de data. Data est un nœud de la liste, chaque data contient un identificateur, les informations concernant la variable, le tableau, ou la méthode déclarée. Chaque data possède une liste d'instances. Chaque instance possède une valeur et un pointeur vers son data pour récupérer les information concernant cette instance.

Notre dictionnaire de données sert à stocker les données déclarées dans un programme Minijaja. Par exemple, lors du déclaration d'une variable de type entier :

```
int test = 3;
```

Cette déclaration a pour l'identificateur la chaîne de caractères "test". Pour stocker cette déclaration dans le dictionnaire, on prends l'identificateur et on implémente la fonction de hachage à cet identificateur. Le résultat de cette fonction est un nombre qui indique l'indice du *bucket* dans lequel la donnée sera stockée. Si, par exemple la fonction de hachage transforme la chaîne "test" en un nombre 0, on crée un nœud data, ce nœud a pour l'identificateur

"test". La nature de l'objet est *variable* et le type d'objet est *integer*, ces deux informations sont aussi stockées dans le nœud data. Nous ajoutons aussi un nœud instance à la liste d'instances du nœud data, ce nœud instance a pour valeur 3. Nous ajoutons le nœud data à la liste chaînée du *bucket* à l'indice 0.

Notre table de hachage a pour *load factor* 0.75, c'est-à-dire que si le nombre d'éléments atteint 75% de la capacité initiale, nous augmentons la capacité de la table de hachage. Ceci est fait pour diminuer la taille de la liste chaîne que chaque *bucket* possède pour garder la complexité de la recherche d'un élément dans la table.

L'intérêt de l'utilisation de la table de hachage est la complexité de la recherche d'un élément. Comme les tables ordinaires, la table de hachage permet un accès en O(1) en moyenne. Toutefois, comme plusieurs paires clé-valeur peuvent se trouver dans un même *bucket*, le temps d'accès dans le pire cas est de O(n).

5.1.2 Le tas

Le tas sert à stocker les données d'un tableau. Lors que l'utilisateur déclare un tableau d'une taille dans le programme Minijaja, les données de ce tableau sont stockées dans le tas. Le tas lui-même est un tableau.

Lors que le tas est initialisé, Le tas est initialisé avec une capacité donnée ou une capacité par défaut. Le capacité par défaut est de 256. Voici le tas après l'initialisation. Supposons que le tas a pour capacité sa capacité par défaut. Le tas possède un bloc libre de taille 256 à l'adresse 0.

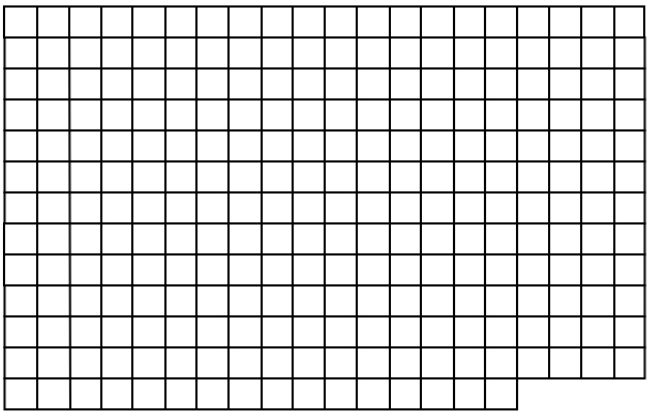

FIGURE 2.3 – Initialisation du tas

Nous avons aussi une table pour mémoriser les adresses des blocs libres dans le tes. Chaque case du tableau contient des adresses des blocs libres dont la taille est de 2 ^ l'indice de la case dans le tableau. Par exemple, la case 0 contient des adresses des blocs libres de taille 2 ^ 0 = 1.

Après l'initialisation du tas, la table d'adresses des blocs libres du tas indique qu'il y a un bloc libres de taille 256 à l'adresse 0.

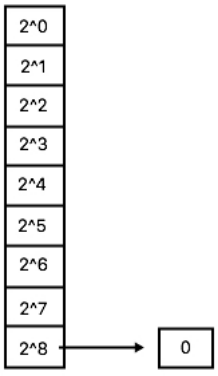

FIGURE 2.4 – La table d'adresses des blocs libres du tas après l'initialisation du tas

Imaginons que l'utilisateur déclare un tableau "t1" de taille 10. Nous par-

courons la table d'adresses pour chercher un bloc dont la taille est égale ou supérieure à 10. Nous avons trouvé un bloc de taille 256 à l'adresse 0. Ce bloc est ensuite découpé.

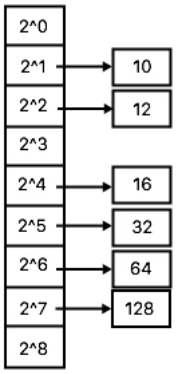

FIGURE 2.5 – Le tableau d'adresses des blocs libres du tas après la déclaration d'un tableau de taille 10

Nous pouvons regarder qu'il nous reste 6 blocs avec la taille totale de 246. Les éléments du tableau "t1" seront stocké dans le tas dans le bloc de taille 10 situé à l'adresse 0 jusqu'à l'adresse 9.

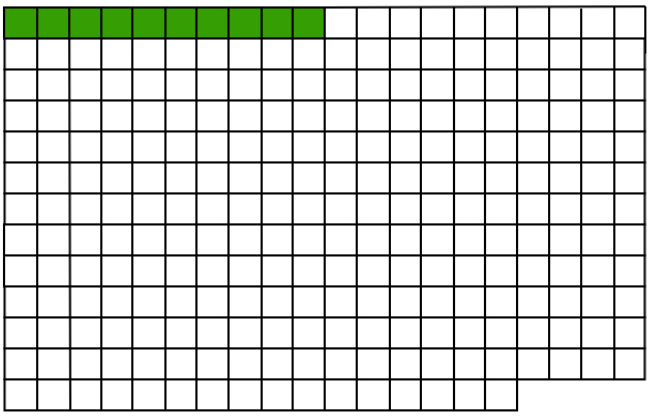

FIGURE 2.6 – Le tas après la déclaration du tableau de taille 10

Le bloc en vert est le bloc destiné à stocker les éléments du tableau "t1".

Supposons que l'utilisateur déclare ensuite un tableau "t2" de taille 3. Nous parcourons la table d'adresses de blocs libres, nous trouvons un bloc de taille 4 à l'adresse 12, nous découpons ce bloc.

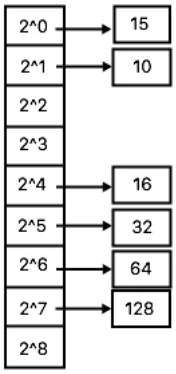

FIGURE 2.7 – La table d'adresses des blocs libres du tas après la déclaration d'un tableau de taille 3

En effet, les éléments du tableau "t2" seront stockés dans le bloc situé l'adresse 12 jusqu'à l'adresse 14.

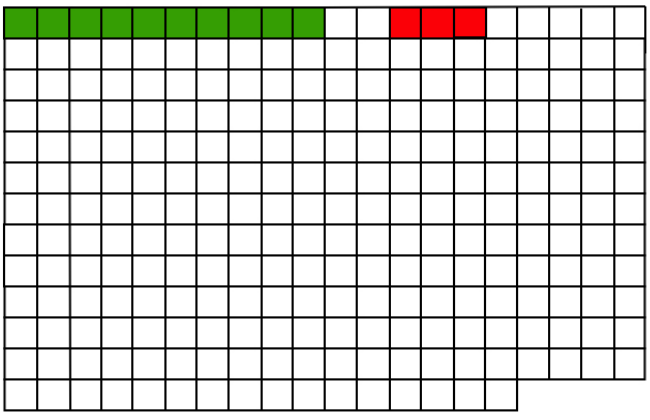

FIGURE 2.8 – Le tas après la déclaration du tableau de taille 3

Le bloc en rouge est le bloc destiné à stocker les éléments du tableau "t2".

Un problème apparaît. Comme ce que nous pouvons voir, il y a deux espaces libres entre le bloc en vert et le bloc en rouge, c'est le problème de fragmentation. Ces deux espaces risquent d'être perdu. Pour résoudre ce problème, nous faisons en sorte qu'à chaque déclaration du tableau, le bloc destiné à stocker ses données est aligné au plus gauche possible. Nous faisons aussi des fusionnement des blocs qui nécessitent d'être fusionnés.

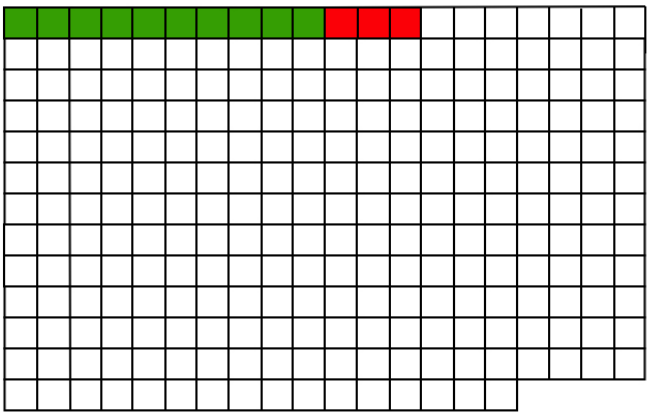

FIGURE 2.9 – Le nouveau bloc est aligné au plus gauche possible

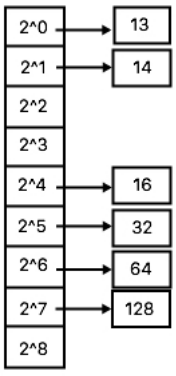

FIGURE 2.10 – La table d'adresses des blocs libres après l'alignement du bloc t2 au plus gauche possible

La même opération est faite lors de la suppression du bloc du tas. De cette

manière, les blocs libres sont toujours regroupés à droite des blocs assignés. Et il n'y a pas de blocs libres situés entre des blocs assignés.

Nous avons aussi une table de symbole pour mémoriser les blocs assignés. Dans notre exemple, la table de symbole est comme ci-dessus :

Identificateur	adresse	taille	références
1	0	10	1
2	10	3	1

FIGURE 2.11 – La table de symbole du tas

L'identificateur du bloc est la clé de cette table. Pour chaque clé, nous avons l'adresse du bloc, la taille du bloc, et aussi le nombre de références pour ce bloc.

5.1.3 La pile

La pile sert à stocker les instances des variables, des tableaux, et des méthodes déclarées dans le programme Minijaja. Par exemple, lors de la déclaration des deux variables de type entier :

```
int i = 3;
int j = 4;
```

Lors de la déclaration de ces deux variables, nous ajoutons à la table de hachage deux nœuds data, et nous ajoutons aussi deux instances à la pile.

<i,INTEGER,VARIABLE,3>
<j,INTEGER,VARIABLE,4>

Toutes les opérations se baseront sur la pile. Dans le cas où l'instance est un tableau, sa valeur est l'identificateur du bloc contenant les éléments du tableau dans le tas. Et dans le cas où l'instance est une méthode, sa valeur est le nœud AST de la méthode.

5.1.4 La Mémoire

La mémoire contient trois éléments : le dictionnaire de données, le tas, et la pile. Lors de la déclaration d'une donnée, l'identificateur et les informations

de la donnée sont stocké dans le dictionnaire de données, la valeur de la donnée est stockée dans la pile. Dans le cas de la déclaration d'une tableau, ses éléments sont stockés dans le tas.

Les opérations qui peuvent être effectuées par la mémoire sont :

```java
public interface IMemory {
    void push(Instance instance);
    Instance pop();
    Instance peek();
    void swap();
    void declVar(String identifier, EnumType type, Value value);
    void identVal(String identifier, EnumType type, int index);
    void declCst(String identifier, EnumType type, Value value);
    void declTab(String identifier, EnumType type, Value value);
    void declMeth(String identifier, EnumType type, Value value);
    void retirerDecl(String identifier);
    void affecterVal(String identifier, Value value);
    void affecterValT(String identifier, Value value, int index);
    void affecterType(String identifier, EnumType type);
    void expParam(ListExpression listExpression, Parameters parameters);
    Value val(String identifier);
    Value valT(String identifier, int index);
    EnumObject objet(String identifier);
    EnumType sorte(String identifier);
    Parameters parametre(String identifier);
    Declarations declaration(String identifier);
    Instructions corps(String identifier);
}
```

FIGURE 2.12 – Les opérations de la mémoire

5.2 Traitement des erreurs

Les erreurs peuvent apparaître lors des opérations dans la mémoire. Lors que une erreur est produite, une exception est levée. Par exemple, dans le cas où l'utilisateur essaie d'affecter une valeur à une variable dans la pile, mais la pile est vide. Une exception du type NullStackException est levée.

```java
/**
 * Assigns a value to a data instance.
 * @param identifier    identifier of the data instance whose value will be modified.
 * @param value         the value that will be assigned.
 */
@Override
public void affecterVal(String identifier, Value value) {
    if (stack.isEmpty()) {
        throw new EmptyStackException();
    }
```

FIGURE 2.13 – L'affectation de la variable dans le cas où la pile est vide

Il existe plusieurs autres exceptions dans la mémoire. Ces exceptions sont aussi utile pour le contrôle de type.

5.3 Choix de tests

Nous effectuons des tests fonctionnels pour chaque fonctionnalité de la mémoire et ses composants. Par exemple, la table de hachage est un composant de la mémoire, nous faisons des tests pour s'assurer que la table de hachage fonctionne correctement. Nous avons un test qui s'assure que la table de hachage augmente sa capacité si le nombre des éléments contenus dans la table de hachage atteint 75% de sa capacité, nous avons aussi un test qui s'assure que lors de l'insertion d'une valeur avec une clé qui existe déjà dans la table de hachage, l'ancienne valeur est supprimée et remplacée par la nouvelle valeur. Voici la capture de ce test.

```
@Test
public void putExistingElement() {
    HashMap<String, Data> table = new HashMap<>();
    Data data = new Data( identifier: "test", EnumType.INTEGER, EnumObject.VARIABLE);
    table.put("test", data);
    Assert.assertTrue(table.get("test").getType().equals(EnumType.INTEGER));

    Data data2 = new Data( identifier: "test", EnumType.BOOLEAN, EnumObject.VARIABLE);
    table.put("test", data2);
    Assert.assertTrue(table.get("test").getType().equals(EnumType.BOOLEAN));
}
```

FIGURE 2.14 – Un exemple du test fonctionnel de la table de hachage

Nous pouvons remarquer aussi que pour chaque test fonctionnel, nous déclarons de nouveau l'élément que nous testons. Dans l'image ci-dessus, nous déclarons la table de hachage au début du test. Nous n'utilisons pas une seule table de hachage pour tous les tests. Nous avons décidé de faire ce choix pour que chaque test ne dépend pas des autres test, autrement dit, chaque test est indépendant et peut être exécuté individuellement.

Cependant, dans certains cas, nous avons besoin de tester une fonctionnalité avant de tester une autre fonctionnalité parce que la deuxième fonctionnalité a besoin d'appeler la première fonctionnalité. Par exemple, pour tester la méthode d'affectation d'une valeur à une variable dans la pile, nous avons besoin d'utiliser la méthode de la déclaration de la variable. C'est la raison pour laquelle nous devons être surs que la méthode de la déclaration de la variable fonctionne normalement avant de pouvoir l'utiliser. Autrement dit, nous avons besoin de tester la déclaration de la variable avant l'affectation de la valeur à une variable.

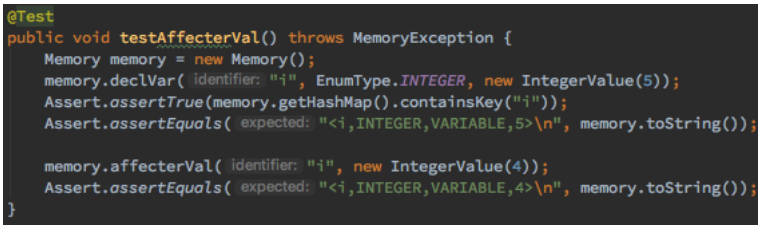

FIGURE 2.15 – Le test de l'affectation d'une valeur à une variable

6 L'interpréteur JajaCode

7 L'Interface Homme-Machine

Xxxx xxxxx....

7.1 Cahier des charges

7.2 Choix de réalisation

7.3 Conception

7.4 Implémentation

7.5 Tests

7.6 Bilan et résultats

Partie 3

Bilan

1 Technique et Fonctionnel

2 Personnel

Sur le plan personnel, ce projet...

Conclusion

Annexe A

Titre

Table des figures

Résumé

Dans le cadre...

As part of...

Mots clés Université de Franche-Comté...

Key words University of Franche-Comté...