Document de conception

Groupe 17
GUILLAUME Laure
LUU Duc Anh
EL RHATRIF Mohammed Amine
CHAKIR Sami
FARID Othmane

27 janvier 2017

1 Analyse lexicale et syntaxique

1.1 Description

Les fichiers concernés sont :

1. src/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4

2. src/antlr4/fr/ensimag/deca/syntax/DecaParser.g4

Les fichiers ci-dessus implémentent l'analyseur lexical et syntaxique. Ils sont écrits en langage Antlr qui prend en entrée la grammaire attribuée fixée dans la spécification et la traduit en un programme capable de reconnaître les lexèmes définis dans le DecaLexer, et transforme ce flux d'entrée au fur et à mesure en un arbre abstrait de syntaxes (Pour plus de détails veuillez vous référer au cahier de charge). La liste des lexèmes manipulables par notre compilateur se trouve dans DecaLexer. D'autres lexèmes peuvent être ajoutés à cette liste (Par exemple : For, do...while, etc), à cette fin on peut aussi ajouter d'autres règles à notre grammaire. La descrption du langage Antlr se trouve dans la partie "ANTLR : ANother Tool for Language Recognition" du cahier de charge.

1.2 Architecture

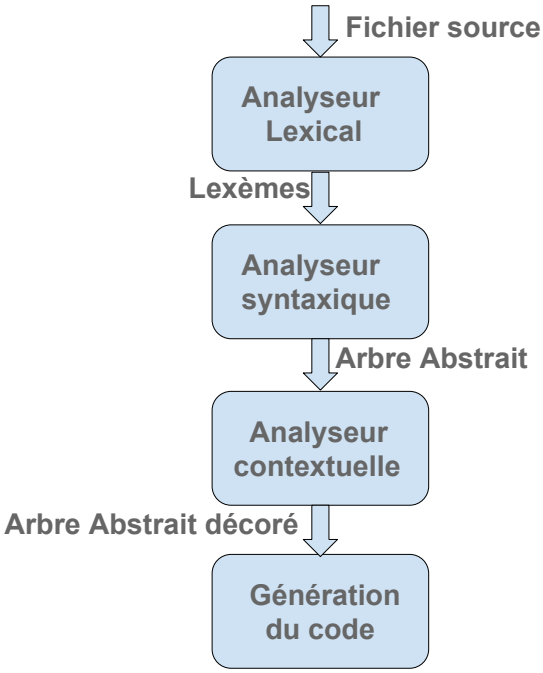

FIGURE 1 – Architecture.

2 Vérifications contextuelles

2.1 Architecture

2.1.1 Description

Les répertoires concernés sont :

1. src/main/java/fr/ensimag/deca/context/*

2. src/main/java/fr/ensimag/deca/tree/*

3. src/main/java/fr/ensimag/deca/context/tools/symbolTable.java

Cette partie est censée décorer l'arbre abstrait généré par l'analyse syntaxique. Elle utilise pour cela la grammaire attribuée fournie dans la spécification. Elle effectue également la détection d'erreurs contextuelles, ainsi que l'enrichissement de l'arbre. Cela se fait en trois passes. Il est à noter que pour le langage sans objet, la troisième passe suffit pour la vérification.

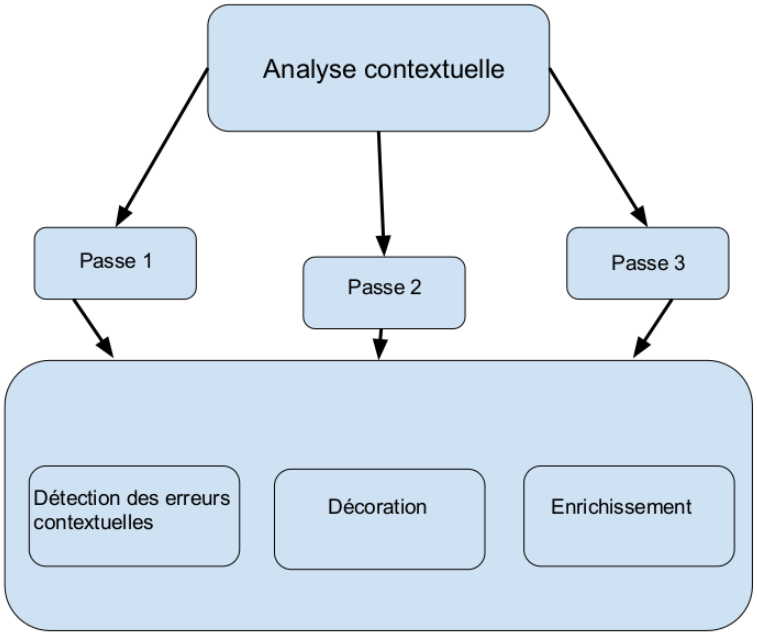

FIGURE 2 – Architecture.

Plus précisement, la vérification contextuelle se fait en deux temps :

1. Lever un message d'erreur dans le cas où le flux d'entrée ne respecte pas la grammaire attribuée fixée auparavant, c'est dans ce but qu'on a implémenté le domaine des attributs de la grammaire, en commençant par l'environnement type, et en finissant par l'environnement expression. Pour ce faire on a créé une nouvelle classe Environment_type, une classe qui ressemble à la conception de la classe Envornment_exp. Dans cette classe on enregistre les symboles et leurs type_Definition correspondants dans un dictionnaire et avec les getters et setters appropriés. Ce choix se justifie par le fait qu'il va nous aider à manipuler dans la suite d'une façon aisée notre Environment_type.
2. Décorer notre arbre syntaxique avec les typages stockés dans l'Environment_type et Environment_exp. Ces deux environnements sont enrichis au fur et à mesure qu'on effectue les trois passes de la vérification contextuelle.

2.1.2 Description de l'implémentation de chaque fonctionnalité

1. Verify : on vérifie les expressions contextuellement à l'aide des fonctions verify que nous avons implémentées dans les classes .java du dossier tree. Les fonctions utilisées sont les suivantes : verifyExpr,verifyRvalue,verifyCondition. Ces dernières utilisent l'environnmentexp et environnementType(champ du paramètre compiler de type Decac- Compiler) pour effectuer leur travail.
2. EnvrionnmentType :une classe qu'on a créée dans java/.../context qui contient un dictionnaire qui associe à chaque Symbol(classe declarée dans tools/symboleTable.java) une TypeDefinition. Pour cette classe on implémente toutes les fonctions nécessaires pour l'intialisation de l'environnment prédéfini, l'ajout d'un nouveau type(sera utilisé dans la partie avec objet), récupération d'une définition d'un type qui existe déjà dans l'environnement.
3. Détection des erreurs : on la gère dans les fonctions verify. En faisant des tests sur les types des fils d'un neoud, on lève des erreurs contextuelles avec un message et la position

de l'erreur dans le cas où ces types ne sont pas compatibles avec la définition de la règle en question.

4. Décoration et enrichissement : tous les paramètres nécessaires pour ces fonctionnalités sont disponibles après l'exécution du programme verifyProgram dans DecacCompiler.docompile(),car on intialise le type (ou la définition parfois si cela est nécessaire) à la fin de chaque traitement d'un noeud (fin de la fonction verify(Expr || Rvalue || Condition).

2.2 Analyse contextuelle pour les programmmes "avec objet"

Les programmes "avec objet" sont les fichiers sources qui peuvent contenir des classes avec les spécifications décrites dans le cahier de charge. De ce fait, l'architecture décrite ci-dessus est la même pour cette partie mais à quelques différences près ; en fait d'autres classes ont été créées, en particulier DeclClass, DeclField, DeclMethod, DeclParam, MethodBody et ListDeclParam. Ces classes permettent la compilation des fichiers instanciant des objets, tout en respectant les caractèristiques de la programmation avec objet (Encapsulation, Héritage ...).

Pour cette raison on a créé les classes suivantes et on a modifié d'autres qui existent déjà :

1. DeclClass : Cette classe hérite de la classe AbstractDeclClass qui à son tour hérite de tree, et permet d'implémenter la passe 1 et 2 et une partie de la passe 3 grâce aux méthodes, respectivement, verifyClass, veridyClassMembers et verifyClassBody. On a attribué à cette classe 4 champs.

2. DeclField : Cette classe hérite de la classe AbstractDeclField qui à son tour hérite de tree. Dans cette classe on implémente les différents vérifications contextuelles concernant les champs de la classe(par exemple on vérifie si les champs existent avec le même nom dans une classe mère, aussi on lui donne le bon index, etc), puis on ajoute chaque champ dans l'Env_expr local où il est déclaré.

3. DeclMethod : Cette classe hérite de la classe AbstractDeclMethod qui à son tour hérite de tree. La vérification de cette classe est similaire à celle de DeclField à quelques différences près. En fait, dans cette classe on trouve le VerifyXYZ des deuxième et troisième passes, et on trouve aussi la liste des paramètres de la méthode. L'ajout de la méthode dans l'Env_expr local se fait sur plusieurs étapes, puisque l'on vérifie s'il s'agit d'un redéfinition ou pas de la méthode, ce qui nécéssite de fouiller dans les classes mères s'il existe une méthode avec le même nom et si oui on doit vérifier la compatibilité de la signature de ces deux méthodes.

4. MethodBody : Cette classe hérite de la classe AbstractMethodBody qui à son tour hérite de tree. Cette classe est dédiée à la troisième passe où on vérifie le corps des méthodes, il s'agit d'examiner la liste de déclarations et liste des instructions de chaque méthode.

5. MethodCall : Cette classe hérite de la classe AbstractMethodCall qui à son tour hérite de tree. On emploie cette classe pour appeler une méthode qui existe dans l'Env_expr local concerné, ce qui demande de vérifier la compatibilité des signatures entre la méthode appelante et la methode appelée. (Pour plus de détails, veuillez vous référer à

l'implémentation de la classe src/.../MethodCall.deca)

6. DeclParam : Cette classe hérite de la classe AbstractDeclParam qui à son tour hérite de tree. On ajoute ici chaque paramètre à l'Env_Expr local concerné mais avant on vérifie si un autre paramètre a le même nom, si oui un message d'erreur est levé. Et d'ailleurs c'est pas la seule vérification qu'on fait dans cette classe ; on peut se référer à cette classe(src/.../tree/DeclParam.java) pour voir les autres vérifications.

7. Selection : Cette classe hérite de la classe AbstractLValue. Cette classe permet d'appeler les champs et les méthodes via l'Env_expr local de la classe concernée. On vérifie, entre autres, la visibilité des champs, et on lève une erreur si on n'a pas le droit d'accéder au champ qu'on veut sélectionner.

8. This : cette classe hérite de la classe AbstractExpr, et sert à designer l'objet courant de la classe dans laquelle elle est utilisée. On vérifie, entre autres, que this n'est pas appelé dans le main.

9. ListDeclClass : Cette Classe hérite de la classe TreeList<AbstractDeclClass>. Elle permet de lancer la vérification sur chaque classe de la liste des classes déclarées dans le fichier source.

10. ListDeclField : Cette Classe hérite de la classe TreeList<AbstractDeclField>. Elle permet de lancer la vérification sur chaque champ de la liste des champs déclarés dans le fichier source.

11. ListDeclParam : Cette Classe hérite de la classe TreeList<AbstractDeclParam>. Elle permet de lancer la vérification sur chaque paramètre de la liste des paramètres déclarés dans le fichier source.

12. ListDeclMethod : Cette Classe hérite de la classe TreeList<AbstractDeclParaMethod>. Elle permet de lancer la vérification sur chaque méthode de la liste des méthodes déclarées dans le fichier source.

13. New : Cette classe hérite de la classe AbstractExpr. Dans cette classe, on vérifie l'instanciation des classes dans le fichier source.

La figure ci-dessous expose ces différents classes et les dépendances qui les relient :

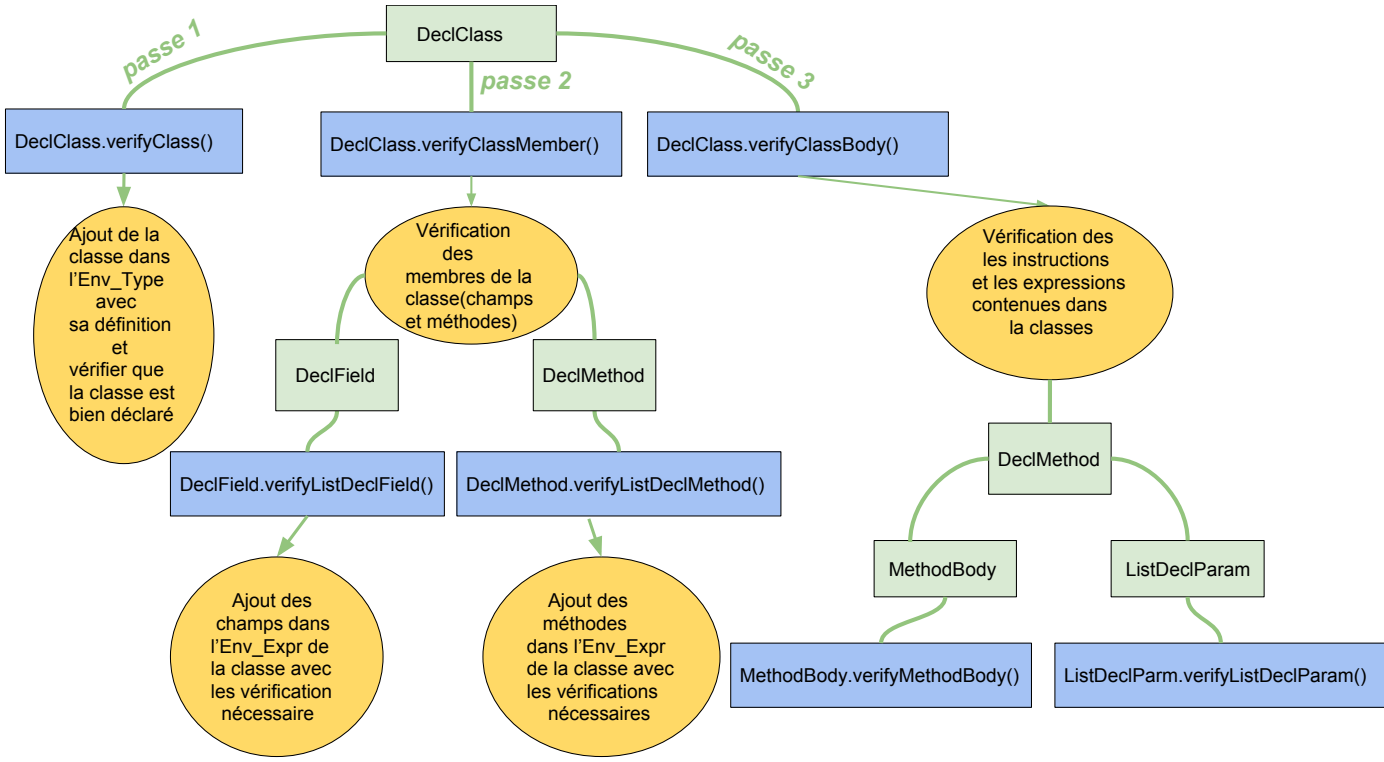

FIGURE 3 – liste des classes et leurs dépendances.

Remarque :
La figure ci-dessus n'explicite pas toutes les méthodes et classes qu'on emploie dans les vérifications contextuelles de notre programme, il s'agit d'une vue globale de ce qui se passe en cas de la déclaration d'une classe. La vérification du programme principal ne figure pas dans ce schéma.

3 Génération du code d'assembleur

3.1 Description

L'objectif de l'étape C est de générer du code assembleur à partir d'un arbre d'un programme Deca contextuellement correct. Les principaux problèmes dans ce travail sont la gestion des registres et la pile, la gestion des erreurs d'exécution (division par zéro, l'erreur d'entrée/sortie,..) ainsi que l'ajout des parties qui permenttent de générer les instructions d'assembleur.

3.2 Liste des composants

- **src/main/java/fr/ensimag/deca/tree/** : Implémentation des parcours de l'arbre abstrait,
- **src/main/java/fr/ensimag/deca/codegen/** : Des classes utiles à la génération du code,
- **src/main/java/fr/ensimag/deca/ima/pseudocode** : La syntaxe abstraite d'un programme en langage d'assemblage IMA,
- **src/test/java/fr/ensimag/deca/codegen/** : Une classe permet d'afficher les codes d'assembleur générés sur la console,
- **src/test/deca/codegen/** : Cas de tests en Deca.

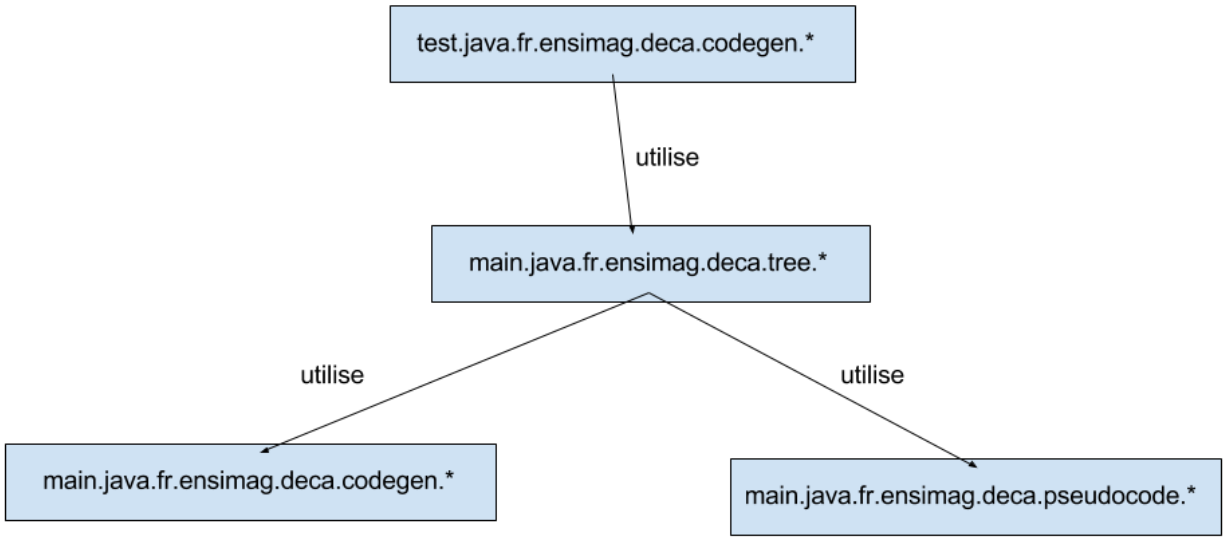

FIGURE 4 – La dépandance des composants.

3.3 Les spécifications

3.3.1 Sur la gestion de mémoire

La classe MemoryManagement dans le package **fr.ensimag.deca.codegen** a comme rôle de gérer les registres utilisés ainsi que les variables d'états dans la partie génération du code de compilateur.

Pour la gestion des registres, on utilise un tableau *avaRegs[]* des booléens pour enregistrer l'état des registres. Par ailleurs, pour traiter quelques problèmes de :

— Nombre des registres maximal : on utilise une variable *RMAX* qui vaut 15 par défault et une méthode *setRMAX(int)* qui permet de changer cette valeur dans l'option de compilateur.

— Prise d'un registre libre : on fait un parcours entre *R2* et *RMAX* sur le tableau *avaRegs[]* et on récupère le premier registre qui n'est pas encore occupé.

— Prise du dernier registre utilisé pour enregistrer : pour ce problème, on a une variable *lastReg* qui contient le numéro du dernier registre. Cette valeur est très utile pour la partie génération des instructions d'assembleur.

— Liberté d'un registre qui n'est plus nécessaire : on implémente une méthode qui permet de changer la valeur associée à ce registre dans le tableau *avaRegs* et on ajoute cette méthode dans les instrucions qui permettent de générer une instruction assembleur. Par exemple : Si on ajoute les valeurs dans les registres *R2* et *R3* par l'instruction : **ADD R2, R3** (R3 := V[R3] + V[R2]), après cette instruction, le registre R2 n'est plus nécessaire, alors on peut le libérer.

```
public ADD(DVal op1, GPRegister op2) {
    super(op1, op2);

    if (op1 instanceof GPRegister) {
        freeRegister(((GPRegister) op1).getNumber());
    }
}
```

FIGURE 5 – Libération du registre après l'utilisation.

— Liberté de tous les registres après la génération du code du block Main : cette action est utile pour la partie traitement du sous-langage *Deca essentiel*. Les registres vont être liberés avant le codage des méthodes. Cela est facile à implémenter par un parcours sur le tableau *avaRegs*.

En outre, dans cette classe, il y a certaines variables d'état utiles pour le codage :

— Nombre des variables globales : pour cette partie, une variable d'état est initialisée à 0 au début et augemente automatiquement chaque fois qu'il y a une déclaration dans le

block Main.

— Nombre des valeurs sauvegardées dans la pile : ce sont les registres empilés dans la pile par l'instruction **PUSH Rm** dans le cas où tous les registres sont occupés. Cela est implémenté comme le nombre des variables globales ppur initinaliser une variable d'état à 0 et l'incrémente quand on a une instruction **PUSH**.

— Taille des tables des méthodes : cette variable donne le nombre de mots occupé par les tables des méthodes et est utile pour la déclaration des variables globales dans le block Main.

— Nombre des variables locales : cette valeur est utile dans le codage des sous-programmes comme les méthodes. On l'utilise pour la vérification de débordement local.

Ces variables sont aussi utiles pour l'ajoutat des codes de la vérification de débordement à la fin de la génération du code.

3.3.2 Sur la génération des instructions d'assembleur

On considère le programme principal qui contient des sous-programmes. On génère le code pour les sous-programmes et on les ajoute sur le programme principal après.

Dans cette partie, des codes utiles sont ajoutés dans les méthodes *codeGen*(IMAProgram)*. Les implémentations des *codeGen** dans les classes, qui correspondent aux feuilles de l'arbre abstraite, ajoutent les instructions concernées dans le programme assembleur dans la plupart des situations.

— Codage des tables des méthodes : pour faciliter cette partie, on ajoute, dans la **fr.ensimag.deca.context.ClassDefinition**, une variable qui contient l'adresse de la table des méthodes et une collection *vtable* qui présente la table des étiquettes des méthodes. Avec chaque déclaration d'une classe, on construit sa *vtable* en copiant toutes les étiquettes dans sa classe-mère, en mettant à jour pour ajouter les équitettes de ses méthodes. Puis, on génère du code assembleur en utilisant cette *vtable* et l'adresse de la table des méthodes ci-dessus.

— Codage des champs : on référence un champ d'un objet en utilisant l'adresse indirecte qui est construite par l'index du champ et l'adresse de l'objet. Le codage de la sélection de champs consiste à enregistrer l'adresse de l'objet dans un registre, puis enregistrer l'adresse du champ en utilisant son index. Le codage d'initialisation contient la sélection de champs et enregistre la valeur initiale dans cette adresse comme une déclaration expliquée ci-dessous.

— Codage des méthodes : le codage d'une méthode contient les déclarations des variables locales et les codages des instructions (des expressions) qui sont expliquées ci-dessous. Le problème dans cette partie est la sauvegarde des registres qui seront utilisés. Pour le faire, on utilise une collection *pushedRegs* mise dans
fr.ensimag.deca.codegen.MemoryManagement
pour sauvegarder les registres utilisés. Chaque fois qu'un nouveau registre est utilisé, on le met dans *pushedRegs*. Après le codage de méthode, on récupère chaque registre **Rm**

et ajoute des instructions **PUSH Rm** au début et **POP Rm** à la fin de la partie de codage de cette méthode.

— Codage des déclarations : le compilateur associe l'identifier de variable à son adresse dans la pile pour utiliser la méthode *setOperand* dans la définition de variable associée à cette déclaration. L'adresse de variable globale dans la pile est calculée via la taille des tables des méthodes et le nombre des variables globales. Celle de la variable locale est l'adresse indirecte avec son déplacement par apport à **LB** (base locale). S'il y a une initialisation, le codage de l'expression concernée est géré et la valeur récupérée est dans le dernier registre utilisé *(lastReg)*. On enregistre cette valeur dans le mot qui correspond à la variable par l'instruction **STORE lastReg daddr**. Dans le cas il n'y a pas d'initialisation, il n'y a non plus le code assembleur explicite.

— Codage des expressions arithmétiques : les expressions arithmétiques contient deux opérandes *lOp* (opérande à gauche) et *rOp* (opérande à droite). En général, on implémente cette partie pour générer du code d'assembleur pour le *lOp*, récupérer le registre qui contient le résultat *(lastReg)*, générer du code d'assembleur pour le *rOp*, récupérer le registre qui contient le résultat *(lastReg)*, et utiliser ces deux registres dans la feuille concernée par l'expression arithmétique.

```
lvalue.codeGenInst(compiler);
reg = getLastUsedRegisterToStore();
rvalue.codeGenInst(compiler);
val = getLastUsedRegisterToStore();
```

FIGURE 6 – Gencode et récupération du résultat.

```
@Override
protected void codeGenInst(IMAProgram compiler) {
    super.codeGenInst(compiler);
    compiler.addInstruction(new ADD(val, reg));
    setLastUsedRegister(reg.getNumber());
}
```

FIGURE 7 – Utilisation de la feuille.

— Codage des expressions booléennes : les booléens ont deux valeurs false ou true. Lors de leur déclaration, le compilateur enregistre donc soit un 1 pour true soit un 0 si le booléen est false. Cependant un booléen peut aussi être représenté par une opération booléenne. Si c'est le cas le compilateur compare les deux opérandes et si l'inéquation est vérifiée, le compilateur renvoie un 1 sinon un 0.

— Codage des structures de contrôle : le *if* et le *while* nécessite l'introduction de labels. Pour le *if*, les appels de labels se font lorsque la condition contenue dans le *if* n'est pas verifiée. Ainsi si celle-ci est vérifiée on rentre dans le *if* puis après avoir exécuté les instructions du *if* on va à la fin du *if*. Si la condition n'est pas vérifiée alors le compilateur

va a l'instruction *else* ou *if else* suivante.

Pour le *while* c'est l'inverse, lorsque la condition est respectée le compilateur va au début du *while* qui contient les instructions à l'interieur du *while* sinon, il continue et sort du *while*. Cela est dû au fait que les instructions du *while* sont avant la condition. Ainsi les opérations booléennes devaient réagir de manière différente à l'appel de l'une ou l'autre fonction, pour cela nous avons introduit une variable globale pour signaler par quelle fonction elles étaient appelées.

3.3.3 Sur la gestion des erreurs d'exécution

Les codes pour la vérification de débordement et le traitement sont ajoutés après la génération du code du programme (ou sous-programme) parce que c'est à ce moment où on sait combien de mots de la pile on va utiliser.

Une autre type d'erreur de débordement se passe quand on fait une opération arithmétique sur les flottans. Donc il faut vérifier le type des opérandes concernées par opérations et puis ajouter le code de vérification après l'instruction arithmétique (**MUL**, **DIV**).

Pour l'erreur *division par zéro*, on ajoute une vérification avant l'instruction concernée par l'opération arithmétique ainsi qu'un signal par le compilateur (par une variable d'état) qu'il faut ajouter le traitement de cette erreur.

La vérification de l'erreur *déréférencement null* est ajouté dans la partie gencode de MethodCall après l'instruction enregistrement d'adresse de l'objet dans un registre.

L'erreur *input error* est vérifiée dans la partie gencode des ReadInt et ReadFloat.

3.4 Description des algorithmes et structures de données

Pour stocker les étiquettes des méthodes (*vtable*), on utilise un dictionnaire, ici c'est un *TreeMap<Integer, String>*. La raison qu'on choisit cette structure vient de l'architecture de la table des étiquettes : chaque méthode a un index et une équitette, l'index est unique dans la *vtable*, par contre, la redéfinition d'une méthode dans la sous-classe change seulement l'étiquette et maintient l'index. Ainsi, comme le Key est l'index, la Valeur est l'étiquette, la méthode put(K, V) de dictionnaires permet de résoudre ce problème facilement. En plus, on utilise le *TreeMap* pour que les étiquettes soient triées selon leur index asssocié, cela garantit que les tables des méthodes sont bien implémentées.

Pour stocker les registres utilisés dans le codage des méthodes (*pushedRegs*), on utilise une pile. Alors au moment où on a généré le code pour une méthode, on récupère un registre au sommet de la pile, ajoute un **PUSH** au début et un **POP** à la fin du codage, puis on récupère un autre registre et refait jusqu'à ce que la pile soit vide. Cela garantit que les registres soient sauvegardés et restaurés exactement.