Documentation de conception

GL37
Thomas Avare, Quentin Candaele, Helena Cazals, Germain Vu

22 janvier 2022

Ce document a pour vocation de décrire l'organisation générale de l'implémentation du compilateur. Cette documentation traite de l'architecture, les spécifications sur le compilateur autre que celles fournies et leurs justifications, ainsi que la description des algorithmes employés autres que ceux fournis et leurs justification.

Table des matières

1 Etape A : Analyse syntaxique et lexicale

1.1 Les classes représentant l'arbre

Pour l'analyse syntaxique nous avons créée un lexer et un parser en complétant les fichiers `DecaParser.g4` et `DecaLexer.g4` en s'appuyant sur les classes de parcours d'arbre de syntaxe déjà fournies pour la partie sans objet.

Pour la partie obejt nous avons eu à créer de nouvelles classess de parcours de l'arbre de dans le répertoire `../Projet_GL/gl37/src/main/java/fr/ensimag/deca/tree`. Nous avons simplement suivi les règles de la grammaire abstraite de deca pour créer cette nouvelle hiérarchie de classe. Nous avons créer des classes concrètes pour chaque terminal (`New.java`, `MethodCall.java` , `This.java`, `Selection.java`, `Return.java`, `DeclClass.java`, `DeclField.java`, `DeclMethod.java`, `DeclParam.java`, `MethodAsmBody.java`, `MethodBody.java`) qui héritent des non terminaux dont ils dérivent dans les règle de la grammaire absyraite. Nous avons donc aussi dû créer des super classes abstraites (`AbstractDeclClass.java`, `AbstractDeclField.java`, `AbstractDeclMethod.java`, `AbstractDeclParam.java`, `AbstractMethodBody.java`) qui héritent elles même de classe abstraites déjà fournies (dont elle dérivent).

Ainsi chaque règle de grammaire de syntaxe abstraite de la forme :

NON_TERMINAL → `terminal`↑*attribut_synthetise* [List de non terminaux]

Une classe abstraite **NON_TERMINAL** a été créée et une classe concrète `Terminal` qui hérite de la première. Si *attribut_synthetise* et la liste de non terminaux n'était pas nuls ils ont été implémentés en tant que attributs privés avec des getter et/ou setter nécessaires rajoutés en fonction des besoins des étapes suivantes.

Les dernières règles de la spécification deca de la forme : **LIST_NON_TERMINAL** → [**NON_TERMINAUX** *] ont été implémentés en tant que classe abstraite étendant les classes TreeList<Classe abstraite correspondant au non terminal>.

Les classes concernées sont `ListDeclClass.java`, `ListDeclField.java`, `ListDeclMethod.java`, `ListParam.java`.

Le Diagramme UML ci-dessous représente la hiérarchie des nouvelles classes créées et les attributs et les méthodes ajoutées en plus de celles qui devait être définie dans la classe obligatoirement (à cause de l'héritage des super classes abstraites donc exclus verifyXyz, decompile, prettyPrintChildren, iterChildren, codeGenXyz). Ces méthodes non explicités

sur le diagramme UML ont été implémentées selon les besoins de l'analyse conextuel et de la génération de code que nous détaillerons plus tard.

1.2 Spécification sur This et MethodCall

L'écriture des règles de syntaxe abstraite ont été faite suivant les spécification de la grammaire concrète déjà en partie écrite.

Nous avons donc rajouté pour chaque règle `ANTLR` du fichier `decaParser.g4` des instances de classe représentant l'abre de syntaxe en paramètre, ou en retour de règle selon les besoins ainsi que leur initialisation selon le token rencontré.

Le non terminal '`this`' est associé à la classe concrète `This.java` qui comporte un attribut boolean qui est faux si '`this`' est effectivement écrit dans le programme (conformément à la spécification). Ainsi, ceci nous a permis d'instancier la classe `MethodCall` avec et sans `this` lorsqu'il y avait appel de fonction sur l'instance courante (`this.m() :select_expression`), ou un appel de méthode simple (`<ident>.m()`). Il a donc été aussi nécessaire de le déclarer dans les fonctions de décompilation, afin de savoir si il était, oui ou non, effectivement présent.

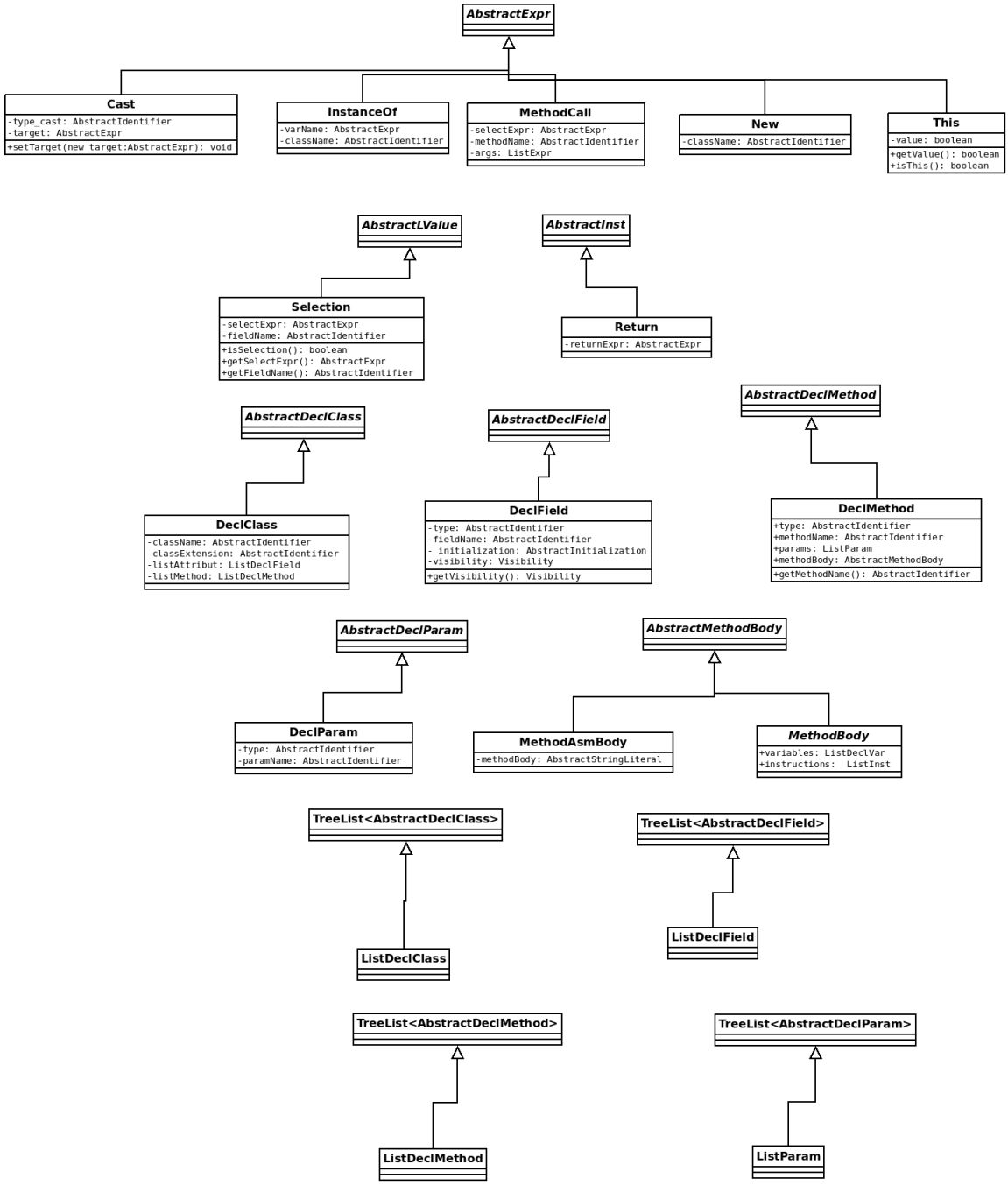

FIGURE 1 – Diagramme UML

2 Étape B : Vérification contextuelle

2.1 Vérification de l'arbre

Pour la vérification de l'arbre, nous avons effectué un parcours en profondeur en implémentant les différentes fonctions verifyXyz nécessaires à la vérification de la syntaxe contextuelle afin de respecter la grammaire attribuée deca c'est à dire la vérifications des types, le rajout de définition et de types. Nous avons aussi implémenté le renvoie des messages d'erreur adéquat aux règles de grammaire conformément à liste des erreurs contextuelles (manuel utilisateur).

Concernant le langage avec objet nous avons choisi de mettre à jour environnementType à chaque déclaration de nouvelle classe. Nous avons aussi pris soin de vérifier que chaque nouveau nœud de type AbstractIdentifier n'était pas défini deux fois.

2.2 Enrichissement de l'arbre

Pour les cast, pour la conversion implicite, nous avons créé une nouvelle classe `ConvFloat` qui étend la classe abstraite `AbstractUnaryExpr` qui remplace le noeud courant dans l'analyse. Nous procédons de la même manière pour les conversions explicites d'un type int vers un type float.

2.3 les environnements : EnvironnementExp et EnvironnementType

Les `EvironnementExp` et `EnvironnementType` sont représentés par deux linkedlist contenant la définition des symbole et les pointeurs sur l'environnement supérieur (qui est null pour EnvironnementType). `EnvironnementType` est quant à lui était ajouté en tant que variable globale dans le `DecacCompiler` passé en paramètre de chaque `verifyXyz`.

3 Etape C : génération de code

3.1 Gestion des registres

Nous avons créé la classe `registerManager` pour tout ce qui concerne l'accès au registre dans les instructions mais aussi pour l'accès à la pile. Le nom peut porter à confusion. La classe `main` contient un attribut de classe `registerManager`. C'est un attribut static et public donc accessible et utilisable par toutes les autres classes.

Cette classe contient quatre attributs :

— `register` : c'est une instance de Register défini dans le package `pseudocode`. Il va permettre principalement de renvoyer les différents registres dans les différentes méthodes (Rm, GB, LB, SP)
— `freeRegister` : c'est une liste de booléen. Elle est de taille fixe 13, et représente l'état des différents registres R3 à R15, s'ils sont utilisés ou non par une instruction lors de l'installation .
— `nbvar` : c'est un entier util dans le `main`. Il indique combien de variables et de méthodes ont été déclarées dans la pile (sur GB) dans le programme principal.
— `nb_lb` : c'est un entier util lors de la déclaration de variable dans les méthodes. En effet, celles-ci sont rajoutées à la pile à partir de l'adresse du registre LB. Cet attribut permet d'éviter de définir au même endroit dans la pile les variables.

A partir de ces 4 attributs, les différentes méthodes définies vont permettre aux instructions d'avoir accès aux éléments de la mémoire.

La fonction `getRegister` est utilisée dans les différentes fonctions `codeGen` des instructions. Elle retourne l'adresse d'un des registres, le premier qui est disponible, et modife la liste `freeRegister` et motfie l'attribut du registre concerné en false pour notifier que le registre n'est plus disponible. Si il n'y a aucun registre de disponible, elle renvoie null.

La méthode `freeRegister`(GPRegister) permet de notifier le `RegisterManager` qu'un registre n'est plus utilisé , et modifie l'attribut `freeRegister` pour mettre à true l'indice correct. Il aurait peut-être été nécessaire de changer le nom de cette méthode car elle peut porter à confusion avec l'attribut.

Ensuite , il y a de nombreuses fonctions `get` , qui donnent accès à des registres particuliers. `getR2` renvoie le registre R2, ce registre est particulier dans notre implémentation et c'est pourquoi il a un accesseur propre. Il y a aussi des accesseurs pour R1 et R0, ce fut nécessaire pour les `print` ou les `return` dans les méthodes. Et enfin il y a `getSP` et `getLB` qui renvoient ces registres particuliers (SP et LB).

La méthode `getLBMemory` est une méthode utilisée uniquement pour les déclarations de variable dans les méthodes. En effet, ces variables sont rajoutées à la base locale (LB) dans la pile. Elle récupère l'attribut `nb_lb` et retourne l'adresse `nb_lb`(LB), qui sera l'adresse de la variable. Ensuite elle incrémente de 1 la valeur de l'attribut `nb_lb`.

La méthode `freeLBMemory` sera appelée à la fin de la génération de code d'une méthode. Elle remet à 1 l'attribut `nb_lb` car cet attribut sera à nouveau nécessaire lors de la génération de code de la prochaine méthode.

La fonction `getStackMemory` permet aux instructions d'accéder à la pile. Sa fonction première est de renvoyer des adresse aux variables lors de leur déclaration ainsi qu'aux méthodes lors de la création de la table des méthodes. Les variables du +main+ sont stockées dans la pile GB (globale base). Elle est aussi utilisée quand il n'y a plus de registres disponibles. Elle utilise l'attribut `nb_lb` pour cela. Elle est reliée à la 2ème fonction `freeRegister`. Cette méthode fut utile et fonctionnelle lors de la partie sans-objet mais elle peut parfois poser problème dans le langage Deca complet. En effet, elle utilise GB pour accéder à la mémoire, et donc elle ne prend pas en compte les modifications de la pile lors d'appels de méthodes. Il aurait été nécessaire avec plus de temps de la modifier et d'utiliser plutôt les registres SP et LB ainsi que les instructions ima ADDSP et SUBSP, et de créer une autre fonction à part.

Les adresses dans la pile de chaque variable sont définies grâce à une méthode `setOperand` et on peut y accéder via une méthode `getOperand`.

3.2 Principes communs dans la génération de code

Lors de la génération de code des différentes instructions on utilise souvent les mêmes processus. Il faut se diriger vers les `codeGenInst` ou `codeGenPrint` des classe. Le résultat d'une instruction , qu'elle soit binaire , unaire ou une simple récupération d'une variable, est toujours placée dans la registre R2. Cela nous permet de faciliter l'enchainement des différentes exécutions d'une ligne de code Deca, car on sait toujours où retrouver le résultat.

Par exemple, dans le code 2, nous avons la génération de code d'un plus. On a récupéré un registre qui sera référencé par la variable R3. On génère d'abord l'instruction de gauche. On sait que le résultat se trouvera dans le registre R2. On déplace ce résultat dans le registre référencé par R3 avant de générer le code de la partie de droite. Et enfin on peut réaliser l'opération entre les deux registres. Le résultat sera encore une fois placé dans le registre R2, ainsi si une autre opération attend le résultat de cette opération, elle saura où récupérer le résultat.

La génération de code se fait généralement de gauche à droite. Pour un assign par contre , on génère le code de la partie gauche d'abord, on récupère dans le registre R2 le résultat et puis on récupère l'adresse de l'identifier avec la méthode `getOperand` et on store le résultat à l'adresse de la variable.

3.3 Génération de code du lanage avec Objet

Génération de la tables des méthodes : Nous avons créé une classe `MethodTable` dans le package codeGen, qui représente la table des méthodes d'une classe. Elle a comme attribut :

- `addr_super_class` qui est l'adresse de la table des méthodes de la super clasListse
- `addr_class` qui est l'adresse de la table de la classe actuelle
- `class_def` qui est la définition de la classe. Cet attribut sera util pour reconnaitre à quelle classe appartient la table des méthodes.
- ensuite nous avons 3 ArrayList : `list_label`, `method_list` et `addr_list`. Ces 3 listes associent un label à son adresse et à la définition de sa méthode. Ces 3 listes sont remplies au fur et à mesure , mais simultanément à chaque fois, de la déclaration des méthodes dans une classe.

Ensuite au niveau des méthodes, la méthode `addMethod` ajoute un triplet definition/label/adresse à chacune des liste correspondante. Il y a plusieurs accesseurs pour obtenir soit la définition de la classe ou l'adresse de cette classe ou celle de la super$_classe$.

La méthode `addSuperTable` permet de rajouter la table des méthodes de la super classe à celle de la fille. Elle récupère les label et définitions et réserve une nouvelle adresse. Enfin dans cette classe, nous avons une méthode qui permet de générer le code de la table. Cette méthode sera appelé en tout début de programme.

Ensuite, nous avons créé une classe `ListMethodTable` qui représente la suite de table des méthodes.

4 L'extension TRIGO

4.1 Utilisation de la bibliothèque

La classe `Math.decah` contient les algorithmes retenus pour réaliser la classe `Math.decah` (voir la documentation de l'extension).

Pour utiliser la bibliothèque dans son programme deca, il suffit d'ajouter la ligne `#import math.decah` en début de fichier.

Vous pourrez désormais calculer les fonctions trigonométriques de bases. D'autres fonctions sont implémentées indirectement et peuvent-être également appelées.

Pour calculer le cosinus de 18.0 par exemple, ils suffit :

— De créer une nouvelle instance de la classe Math avec la commande `Math m = new Math()`
— De faire appel à la méthode cosinus de notre instance `float f = m.cos(18.0)`

Pour plus de détail sur les différentes méthodes utilisables, vous pourrez vous référez à la partie ci-dessous,

4.2 Architecture de la bibliothèque

La classe `Math.decah` contient de quoi calculer :

La valeur absolue d'un flottant :
`float _abs(float val)` : Retourne la valeur absolue d'une valeur flottante.

— Si l'argument n'est pas négatif, l'argument est retourné.
— Si l'argument est négatif, la négation de l'argument est retournée.

Le factoriel d'un entier :
`int _factoriel(int val)` :
— Retourne le factoriel d'une valeur entière.

La racine carré d'un flottant :
`float _sqrt(float x)` : Renvoie la racine carrée positive correctement arrondie d'une valeur float.

Cas spéciaux :

— Si l'argument est un infini positive, alors le résultat est positif.
— Si l'argument est un zéro positif ou un zéro négatif, alors le résultat est le même que l'argument.
— Sinon, le résultat est la valeur float la plus proche de la véritable racine carrée mathématique de la valeur de l'argument calculé à partir de l'algorithme de Héron.

fonction puissance :
`float puiss(int a, int b)` : Renvoie la valeur du premier argument élevé à la puissance du deuxième argument.

Le résultat calculé doit être à quelques ulp du résultat exact.

Fonctions trigonométriques :
`float sin(float f)` : Renvoie le sinus trigonométrique d'un angle(Méthode Cordic amélioré + décalage n_π).

Cas spéciaux :

— Si l'argument est zéro, alors le résultat est un zéro avec le même signe que l'argument.
— Le résultat calculé doit être à moins de quelques ulp du résultat exact.
— Les résultats doivent être semi-monotones.

`float cos(float f)` : Renvoie le cosinus trigonométrique d'un angle (Méthode Cordic amélioré + décalage n_π).

— Le résultat calculé doit être à moins de quelques ulp du résultat exact.
— Les résultats doivent être semi-monotones.

`float acos(float f)` : Renvoie l'acos trigonométrique d'un angle (déduit de l'atan).

— Le résultat calculé doit être à moins de quelques ulp du résultat exact.
— Les résultats doivent être semi-monotones.

`float atan(float f)` : Renvoie l'Atan trigonométrique d'un angle (Application de la moyenne arithmétique-géométrique).

— Le résultat calculé doit être à moins de quelques ulp du résultat exact.
— Les résultats doivent être semi-monotones.

ULP
`float ulp(float f)` : Renvoie la taille d'un ulp de l'argument.

ULP : L'unité de précision élémentaire sur les nombres flottants est l'intervalle entre deux flottants représentables (≪ Unit in the last place ≫)

autre
`float _tab(int i)` : Substituer le tableau statique qui n'est pas implémenté en DECA (pour le calcul de

5 Les points à améliorer/ faire

5.1 Partie objet

Nous avons identifié les parties ne fonctionnant pas sur notre compilateur. Pour la plupart d'entre elles, elles sont dues à un manque de temps pour fournir le compilateur en temps et en heure. C'est en partie dû à la sous estimation de l'étape C de la partie objet.

L'instruction `instanceOf` n'est pas compilable.Les étapes B et C ne sont pas implémentées pour permettre son utilisation. Il faudrait un temps supplémentaire de 1 ou 2 jours pour permettre une implémentation correcte.

Quant au `cast`, son implémentation ne permet de convertir un objet d'une classe vers sa super classe. De même les étapes B et C sont à modifier pour permettre cela.

Les appels de méthodes (`methodeCall`) sont aussi à améliorer car lorsque l'on appelle une méthode dans une autre nous avons l'obligation d'utiliser l'expression de sélection `this.nomMethode` (ex : `this.m()` au lieu de `m()`).

L'extension rendu le jour de la deadline ne compilait pas correctement à cause d'une méthode appelée avant sa définition dans le fichier `Math.deca`. Le problème n'était donc pas lié à un défaut de notre compilateur mais à un manque de compréhension de la syntaxe deca de notre part. Ce problème a été réglé rapidement depuis en définissant la méthode `_tab` en début de fichier. Ainsi nous pouvons inclure notre extension dans tous fichier deca.

Une des limitation de notre compilateur est aussi le nombre d'opérande sur une même ligne . Si il est trop grand il y a des chances que le nombre de registre soit insuffisant et il faut alors accéder à la pile. Cela marchait bien pour la partie sans objet, mais avec objet la pile est modifiée régulièrement et nous n'avons pas eu le temps de modifier RegisterManager.java pour répondre à cette problématique .

Nous avons aussi un problème lors de l'extension de classe : la classe fille récupère les attributs de la classe mère, et ceci n'a pas été implémenté, seul les champs déclarés dans la classe fille sont pris en compte

Enfin, lors du rendu de notre compilateur, la génération de code dans le cas d'un paramètre dans une méthode ne se faisait pas car le getVaribleDefintion n'était pas implémenté. On ne pouvait donc pas print autre chose que que le problème a été réglé depuis.

```
this.getLeftOperand().codeGenInst(compiler);
compiler.addInstruction(new LOAD(r2, (GPRegister)r3));
this.getRightOperand().codeGenInst(compiler);
compiler.addInstruction(new ADD(r3, r2));
Main.rmanager.freeRegister((GPRegister)r3);
```

FIGURE 2 – generation de code instruction plus