Grenoble INP – ENSIMAG, UGA
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Projet Génie Logiciel

Documentation de conception

Yanis BOUHJOURA, Hugo DABADIE, Garance DUPONT-CIABRINI,
Marek ELMAYAN, Julien LALANNE

GL09 Groupe 2

27 janvier 2022

Table des matières

1 Introduction

La compilation d'un fichier deca a été découpé en trois étapes. La première produit qui, après une analyse lexicale et syntaxique (à partir d'un lexer et d'un parser) construit un arbre abstrait primitif du programme. Ensuite vient la vérification contextuelle, il s'agit d'enrichir l'arbre abstrait. Enfin il y a la génération du code, c'est-à-dire la production d'une suite d'instructions qui peut être lu par la machine abstraite ima. Pour réaliser ces deux dernières étapes, une structure de classes Java nous était fournie. Celle-ci repose majoritairement sur la notion d'héritage, ce qui permet de factoriser une grande partie du code. Ainsi notre travail a principalement consisté à implémenter les méthodes nécessaires à chaque classe. Nous allons maintenant décrire plus en détail la conception architecturale de notre projet.

2 Conception architecturale

2.1 Analyse lexicale et syntaxique - Etape A

La première étape de conception dans ce projet à été l'analyse lexicale et syntaxique du programme fourni en entrée. Il fallait pour cela implémenter un lexeur et un parseur permettants de valider ou de refuser syntaxiquement le programme.

2.1.1 Architecture de l'analyse lexicale et syntaxique

Pour créer ce lexeur et ce parseur, nous avons utilisé l'outil ANTLR (ANother Tool for Language Recognition), un outil permettant avec une syntaxe assez simple de faire de la reconnaissance de langage. Deux fichiers ANTLR (.g4) sont donc présents dans notre architecture, un pour le lexeur et l'autre pour le parseur. Lors de la compilation du projet, ces fichiers ANTLR sont alors convertis en deux fichiers Java(présents dans target/generated-sources/antlr4/fr/ensimag/ensimag/deca/syntax) pouvant alors être facilement intégrés au projet. Le lexeur définit des TOKENS que doit reconnaître le langage (analyse lexicale) et transforme ainsi une suite de caractères en une suite de TOKENS que peut interpréter le parseur. Ce dernier ionterprète donc uns suite de TOKENS et indique si elle est syntaxiquement correcte au langage. Si le programme n'est pas syntaxiquement correct, on peut ainsi renvoyer une erreur alors que s'il l'est, on peut initialiser l'arbre Java correspondant à chaque expression, chaque instance, etc... Cela nous sera utile plus tard pour réaliser une analyse contextuelle du code et générer le code assembleur.

Pour tester notre analyseur lexical et syntaxique, deux scripts de test sont présents : test_lex et test_synt et permettent de lancer l'analyse lexicale ou syntaxique sur un programme Deca ou sur ce que l'utilisateur entre dans l'entrée standard.

2.1.2 L'analyse lexicale et syntaxique pour l'affichage - La fonction print

Une première étape de la conception a été de permettre l'affichage sur la sortie standard à l'aide de fonctions intégrées au langage Deca. Pour cela, il a fallu implémenter un certain nombre de règles dans le lexeur et le parseur.

Dans le lexeur, il a tout d'abord fallu réserver cetrains mots comme 'print' ou 'println' ainsi que certains symboles comme '{' et '}' pour pouvoir créer un programme main. Enfin, il a fallu définir les types entier, flottant ainsi que les chaînes de caractères et pour finir les caractères spéciaux tels que le retour à la ligne.

Dans le parseur, il a donc fallu implémenter les règles correspondantes à commencer par la création d'un programme main, ainsi que la déclaration d'une liste d'instances et d'une instance. Cela nous permettait de créer un 'print'. Il fallait ensuite gérer les arguments de la fonction print, nous avons donc dû implémenter les règles syntaxiques associées à une liste d'expressions et à une expression.

Après avoir fait cela, nous pouvions reconnaître des programmes de la forme :

```
{
    print("Hello world", 7, 3.14);
}
```

2.1.3 L'analyse lexicale et syntaxique pour le langage sans objets

Après avoir fini de gérer l'affichage, il fallait pouvoir déclarer des variables, faire des calculs, et faire tout ce qui est possible de faire dans un langage sans objet. L'analyseur lexical et syntaxique avait alors besoin d'être étendu.

Pour le lexeur, il a fallu réserver plusieurs autres mots comme 'if', 'then', 'else' pour les structures conditionnelles ou encore 'readInt' et 'readFloat' pour gérer l'entrée standard. Il fallait également reeconnaître tous les opérateurs booléens, arithmériques et tous les symboles spéciaux :

```
LT: '<';
GT: '>';
EQUALS: '=';
PLUS: '+';
MINUS: '-';
TIMES: '*';
SLASH: '/';
PERCENT: '%';
COMMA: ',';
OPARENT: '(';
CPARENT: ')';
OBRACE: '{';
CBRACE: '}';
EXCLAM: '!';
```

```
SEMI: ';' ;
EQEQ: '==';
NEQ: '!=';
GEQ: '>=';
LEQ: '<=';
AND: '&&';
OR: '||';
```

Dans le parseur, nous avons également dû implémenter les règles correspondantes : les déclarations de variables, les opérations unaires et binaires, les opérations de sélection, les types et les identificateurs.

En sortie de parseur, nous avions à ce moment soit une erreur dûe au fait que le langage ne soit pas syntaxiquement correct ou contienne des objets, soit un arbre syntaxique où chaque noeud est représenté par une classe Java et correspond à une règle syntaxique (exemple : une déclaration de variable, une opération booléenne, ...).

Un exemple typique de ce que pouvait reconnaître notre parseur à ce moment là serait :

```
{
    int i = readInt();
    i = 4 * i + 12;
    if (i < 314) {
        println("L'entier 4 x i + 12 est inférieur à ", 314);
    }
}
```

2.1.4 L'analyse lexicale et syntaxique pour le langage complet

Enfin, il fallait que notre analyseur puisse valider ou rejeter des programmes contenant des structures d'objets. Il fallait donc encore une fois mettre à jour le lexeur et le parseur.

Pour le lexeur, il n'y avait ici pas beaucoup de choses à faire, il a suffit de réserver les mots du vocabulaire de la programmation objets :

```
ASM : 'asm';
CLASS : 'class';
EXTENDS : 'extends';
INSTANCEOF : 'instanceof';
NEW : 'new';
NULL : 'null';
PROTECTED : 'protected';
RETURN : 'return';
THIS : 'this';
TRUE : 'true';
```

Dans le parseur cependant, il y avait beaucoup de choses à implémenter : la déclaration de classes, l'extension d'une classe à une autre, le corps d'une classe, la déclaration de champs de classe, de méthodes de classe, la visibilité des champs, les paramètres de méthodes, l'appel à une méthode, à un champ de classe, le cast et le comparateur 'instanceof'. Le processus est toujours le même : on vérifie si ce qu'on a parsé est syntaxiquement correct, si non on renvoie une erreur et si oui on créé un noeud de l'arbre syntaxique.

2.1.5 L'analyse lexicale et syntaxique pour l'extension - tableaux et matrices

Cette partie est détaillée dans la documentation de l'extension et ne sera pas développée ici.

2.1.6 La gestion des erreurs

Lors de l'analyse lexicale et syntaxique, si le programme donné n'est pas lexicalement ou syntaxiquement correct, une erreur est levée indiquant la ligne et la colonne auxquelles a été faite la faute. Cela permet de relever beaucoup d'erreurs permettant à l'utilisateur de savoir que son programme n'est pas syntaxiquement correct. De plus, si aucune erreur n'est levée ici, on sait que l'on peut passer à l'étape de vérification contextuelle sans qu'il y ait de problèmes liés à la syntaxe.

2.2 Analyse et vérification contextuelle - Etape B

Une fois l'arbre syntaxique créé, il faut vérifier que le contexte du programme est bien respecté c'est-à-dire vérifier que les types sont bien ceux attendus, que les paramètres donnnés lors de l'appel à une méthode sont bien ceux qui ont été précisés lors de la création de celle-ci, etc... Pour cela, nous allons parcourir l'arbre en vérifiant chaque noeud de celui-ci.

2.2.1 Architecture de l'arbre syntaxique et des méthodes de vérification

L'architecture adoptée est typique d'un arbre : Une classe Tree correspond à un noeud de l'arbre et toutes les classes créées (instances, expressions, ...) héritent de cette classe Tree. Chaque classe comporte des attributs correspondants aux noeuds enfants de cette classe. Par exemple, une opération binaire va avoir comme noeuds fils les deux expressions (gauche et droite) composant l'opération.

2.2.2 Les structures d'affichage

Une noeud AbstractPrint est créé pour gérer les fonctions d'affichage. Il possède comme noeuds fils tous les arguments à afficher. De cette classe dérivent deux classe Print et Println gérant respectivement l'affichage sans et avec retour à la ligne finale.

Lors de l'étape de vérification contextuelle, on s'assure ici que tous les arguments sont compatibles pour être affichés. On ne pourra pas print un objet par exemple.

2.2.3 Les litéraux

Héritent de la classe Tree également les classes IntLiteral, FloatLiteral et BoolLiteral qui permettent de stocker des litéraux (i.e. directement des expressions de la forme '3', '5.3' ou 'true'). Ces classes ont comme attributs les valeurs enregistrées par le litéral. Lors de la vérification, on déclare simplement que le noeud est du type correspondant (int, float ou bool), il n'y a rien à vérifier contextuellement.

2.2.4 La déclaration des variables

Lors de l'entrée dans un nouvel environnement (main, class ou method) on commence par déclarer toutes les variables locales. Le noeud ListDeclVar de l'arbre est donc là pour stocker toutes ces déclarations sous forme d'un ArrayList. Les noeuds enfants sont de la forme DeclVar et ont pour attributs le type de la variable, son identifier (nom) ainsi qu'une potentielle initialisation. Lors de l'étape de vérification, on teste alors que le type n'est pas Void, que la variable n'est pas déjà déclarée dans l'environnement local et on lance les vérifications sur les noeuds fils.

2.2.5 Les opérations unaires et binaires

Maintenant que l'on peut proprement déclarer des variables et des litéraux il est intéressant de gérer les opérations sur ceux-ci. Deux classes abstraites sont alors ici pour définir de manière globale les opérations : AbstractUnaryExpr et AbstractBinaryExpr pour gérer les opérations unaires ou binaires.

Les opérations unaires sont :

— Le moins unaire qui a pour noeud fils seulement l'opérande. On lance alors la vérification sur cette opérande puis on vérifie que l'expression est bien un entier ou un flottant.

— Le Not dans lequel on vérifie que l'opérande est booléenne.

— Le ConvFloat dans lequel on vérifie que l'opérande est un entier.

Les opérations binaires sont :

— Les opérations booléennes Or et And dans lesquelles on vérifie les opérandes et le fait que celles-ci soient bien booléennes.

— Les opérations de comparaison et arithmétiques dans lesquelles on vérifie les opérandes et le fait que celles-ci soient bien de types compatibles (int et float) avec l'utilisation possible du noeud ConvFloat.

— Les assignations de variables '=' dans lesquelles on vérifie que l'assignation soit compatible.

2.2.6 Les structures de contrôle

Enfin, pour faire fonctionner le langage sans objet, il faut créer des structures de contrôle (if et while) en créant les noeuds IfThenElse et While.

Le IfThenElse a pour noeuds fils la condition ainsi que les listes d'instances représentants les branches then et else. Lors de l'étape de vérification contextuelle, on se contente d'appeler la fonction de vérification sur ces trois noeuds fils.

Le While a pour noeuds fils la condition ainsi que la listes d'instances représentants le corps du while. Lors de l'étape de vérification contextuelle, on se contente d'appeler la fonction de vérification sur ces trois noeuds fils.

2.2.7 La déclaration de classes

La première étape pour la partie objet du compilateur était la déclaration de classes. Pour cela un noeud ListDeclClass a été créé contenant dans une HashMap les différentes classes déclarées. Les noeuds DeclClass ont eux pour noeuds fils les identifieurs de la classe et de la super classe déclarée, ainsi que des listes de déclarations de champs et de méthodes. Pour la vérification contextuelle, on vérifie ici que la super classe a déjà été déclarée, que la classe que l'on essaie de déclarer ne l'est pas déjà et on ajoute à l'environnement local la nouvelle classe. Enfin, on lance les vérifications sur les fields et les méthodes.

2.2.8 La déclaration de fields

Comme dit précédemment, chaque noeud DeclClass a pour fils une liste de déclarations de champs. Pour chaque field, on vérifie que celui-ci n'existe pas déjà dans l'environnement local, qu'il n'est pas de type void et qu'il est bien initialisé.

2.2.9 La déclaration de méthodes

La dernière étape pour la vérification d'une classe est la vérification de ses méthodes. Celles-ci sont stockées dans une ListDeclMethod. Chaque méthode a pour attributs un type, un identificateur (nom de la méthode) ainsi qu'une liste d'arguments et un corps de fonction. Lors de la vérification, on teste si la signature de la fonction correspond bien à des arguments valides, si l'on est en train de créer une nouvelle fonction ou si on souhaite l'override, si la méthode a déjà été définie dans la même classe ou si les arguments ne correspondent pas avec la super définition. Enfin, dans un nouvel environnement local (corps de la fonction) on déclare les paramètres de la méthode et on vérifie que le corps de la méthode est correct contextuellement.

2.2.10 L'appel à une méthode

Enfin, on peut créer des classes avec le mot clé 'new' et y appeler des méthodes. Le noeud MethodCall a alors été créé. Il stocke l'identifier de la méthode, les ar-

guments avec lesquels celle-ci a été appelée ainsi que l'objet auquel la méthode fait référence. Pour la vérification contextuelle, on teste ici que les arguments sont bien ceux attendus par la méthode, et que celle-ci est bien déclarée dans l'objet qu'elle référence.

2.2.11 L'analyse contextuelle pour l'extension

Cette partie est détaillée dans la documentation de l'extension et ne sera pas développée ici.

2.3 Génération de code - Etape C

La génération du code commence dans le fichier DecacCompiler.java qui fait appel à la classe Program dans la méthode doCompile(). Ensuite le code va être généré en accord avec l'arbre construit aux étapes précédentes.

2.3.1 Structure du code assembleur généré

Un fichier assembleur généré va toujours commencer par les instructions qui vérifient qu'il n'y a pas de débordement de la pile. Mais pour les écrire, des informations sur le code qui suit sont nécessaires, ainsi les instructions ne sont insérées en début de fichier qu'après la génération du reste du code.
Ainsi la première étape va être de construire la table des méthodes des différentes classes. Comme dans tous les cas, la classe Object est définie par défaut, son code sera toujours généré (une optimisation possible aurait été de le faire uniquement lorsqu'un objet est instancié). Tous le code la concernant est généré grâce au fichier codegen/ObjectClass.java. Ensuite on s'occupe des tables de méthodes des classes définies par l'utilisateur, cela est fait majoritairement dans DeclClass.java. Cependant avant d'écrire le code il faut gérer les redéfinitions de méthodes et d'attributs entre classe mère et classe fille.
Cette première passe terminée, on s'occupe ensuite du codage du programme principal. On commence par la déclaration des variables (globales) puis on génère le code des instructions. Ensuite pour chaque classe on s'occupe de générer le code d'initialisation des attributs puis des méthodes. Enfin le fichier termine par les labels des erreurs qui peuvent être potentiellement appelées pendant l'exécution.

2.3.2 Les expressions

L'ensemble des instructions dérive de la classe abstraite AbstractInst dans laquelle on définit une méthode abstraite codeGenInst() qui s'occupe de générer le code ainsi il faut l'adapter pour chaque sous-classe.
De cette super classe dérive la classe AbstractExpr qui regroupe toutes les expressions. On peut encore ensuite distinguer les expressions binaires et les expressions

unaires. Ainsi, comme le codeGenInst() se ressemble pour certaines catégories d'expression, il a été factorisé dans les classes de plus haut niveau. Pour les opérateurs, arithmétiques et booléens, on a utilisé des mnémoniques afin de pouvoir garder cette factorisation. Enfin les fonctions codeGenInst() prennent en arguments le numéro du registre courant ainsi le registre utilisé est propagé et lors d'une opération binaire qui nécessite un second registre, on appellera codeGenInst() sur le second opérateur avec le registre n+1. De plus on ajoute un test avant d'exécuter la fonction et s'il n'y a plus de registre disponible (par défaut 16 mais peut être limité avec l'option -r X du compilateur) on utilise les instructions PUSH et POP pour sauvegarder momentanément les registres puis les restaurer.

2.3.3 Les structures de contrôle

Les points d'entrée des structures de contrôle sont les classes IfThenElse et While dérivant de la classe Tree. Dans la méthode codeGenInst de celles-ci, on créé différents labels en fonction du booléen renvoyé par la condition. Cependant, le comportement de ces deux structures est différent. En effet, lors d'un test If, on saute à un label Else si le résultat du test est false. Lors d'un test While, on saute au label startWhile si le résultat est True. Les structures pouvant être présentes dans un test booléen sont les opérations booléennes, les opérations de comparaison, les litéraux booléens, les identifiers, les appels à des méthodes et l'opération InstanceOf. Dans tous ces noeuds, nous avons donc créé une méthode codeGenJump qui génère le code assembleur réalisant la consigne suivante : évaluer le résultat du test booléen et sauter au label donné en argument en fonction du résultat et de la structure appelante (If ou While).

2.3.4 L'initialisation des variables

Les consignes données dans le poly spécifient que lorsqu'une variable locale est initialisée sans valeur alors on lui affecte la valeur par défaut 0. Cependant pour les variables globales instanciées sans valeur à l'initialisation le comportement était indéfini. Ainsi nous avons décidé que plutôt que de gérer les appels à des variables globales non initialisées, nous appliquerions la même politique que les variables locales aux variables globales i.e. une initialisation par défaut à 0.
Au niveau de la conception chaque variable déclarée possède un attribut de type AbstractInitialization qui peut être soit Initialisation ou NoInitialization avec chacun sa fonction codeGen() qui ajoute les instructions en accord avec le principe énoncé précédemment.
De plus, lorsque l'on instancie une variable quelque soit son type (globale, locale, attribut, paramètre, ...) on stocke son adresse dans l'attribut Operand de la classe expDefinition qui est associé à chaque variable. Ainsi plus tard dans la compilation un appel à getOperand() permet de récupérer l'adresse à laquelle est stockée une variable.

2.3.5 L'affichage - la fonction print()

La génération du code pour l'affichage d'expressions, la fonction print() en deca, se fait un peu différemment des instructions classiques. En effet comme les fonctions assembleurs ne prennent pas de registre en argument et affichent toujours la valeur stockée dans le registre R1, on doit dans un premier temps appeler la fonction qui va générer le code de l'expression pour l'évaluer puis la fonction codeGenPrint(), redéfinie dans chaque classe susceptible d'être affichée, va stocker cette valeur dans le registre R1 et va ajouter l'instruction correspondant au bon type de valeur à afficher.

2.3.6 Les spécifications du langage objet

Le langage objet qu'il nous était tenu d'implémenter, avait ses spécificités par rapport aux instructions que l'on trouvait dans la partie principale (Main) d'un programme.
Tout d'abord la déclaration des variables en tant qu'attribut de classe se fait dans DeclField qui est appelé pour chaque élément de ListDeclField. Pour savoir à quelle adresse enregistrer l'attribut on stocke puis récupère son index dans la classe FieldDefinition attribuée à l'élément concerné.
Ensuite vient la déclaration des méthodes, elle est faite similairement dans DeclMethod et ListDeclMethod, cependant il faut ensuite pour chaque méthode sauvegarder les paramètres de la méthode au bon endroit de la pile. Pour cela on setOperand() avec l'index récupéré dans ParamDefinition de la même manière que pour les Fields.
Enfin il nous reste à générer le body de la méthode. Comme celui-ci est structuré comme le programme main (ListDeclVar puis ListDeclInst) on applique les mêmes fonctions de génération de code. La seule différence est au niveau de l'initialisation des variables car celles-ci sont locales à la méthode ainsi on utilise un booléen qui est pris en argument de la fonction pour les différencier des variables globales.
L'ensemble de ce code est généré à la suite du code du programme principale mais est appelé au cours de l'exécution de celui-ci grâce à des Labels.

2.3.7 Les erreurs à l'exécution

Toute la gestion des erreurs assembleur, susceptibles d'être levées à l'exécution, est regroupé dans le fichier *AssError.java*. On définit dans ce fichier un label pour chaque erreur possible ainsi que les méthodes permettant d'ajouter les instructions qui permettent de sauter au bon label correspondant. Un booléen est associé à chaque type d'erreur, initialisé à *false*, dès qu'une erreur est appelée on le passe à *true* ainsi il permet de savoir quelles erreurs il est utile d'afficher à la fin du fichier. Grâce à cette optimisation il n'y a pas de code superflu généré pour les erreurs.

2.3.8 Les options de decac

— L'option -n permet de ne créer aucune ligne de code assembleur liée aux erreurs, ainsi elle est à utiliser une fois que l'on sait que le programme deca écrit ne provoque aucune erreur, afin d'avoir des fichiers moins volumineux et une exécution plus efficace.

— L'option -P permet de compiler plusieurs fichiers en parallèle et récupère automatiquement les ressources fournies pas l'appareil qui exécute la commande

— L'option -c est une option personnalisée qui permet d'afficher les commentaires de code dans le fichier assembleur ainsi s'il y a une erreur à l'exécution on peut plus facilement débugger mais lors d'une compilation classique le fichier .ass créé est moins volumineux.

3 Conclusion

Sur la partie commune du projet, nous avons gardé la structure de départ qui nous était donnée et nous avons implémenté les méthodes des différentes classes Java. Le principe d'héritage d'un langage orienté objet comme Java à permis de factoriser une grande partie du code cependant il peut rendre plus difficile sa prise en main. Ainsi afin d'avoir une meilleure vue d'ensemble un diagramme UML est disponible dans *./docs/diagrams*. Le grand nombre de classes ne permettant pas de fournir un diagramme lisible sans zoomer on a donc décidé de ne pas l'inclure dans ce document.
En ce qui concerne l'extension nous nous sommes inspirés du squelette de base pour une intégration plus facile mais pour plus détails sur les algorithmes et la structure spécifique veuillez consulter la documentation de l'extension.