Document de conception

Equipe 10

Janvier 2021

Contents

1 Introduction **2**

2 Architecture **2**
- 2.1 Analyse lexicale et syntaxique . . . 2
  - 2.1.1 Syntaxe sans objet . . . 2
  - 2.1.2 Syntaxe avec objets . . . 2
- 2.2 Analyse et vérification contextuelle . . . 3
  - 2.2.1 Manipulation des environnements . . . 3
  - 2.2.2 Vérification contextuelle dans le package Tree . . . 4
- 2.3 Génération de code . . . 6
  - 2.3.1 Les expressions : *codeGenExpr()* . . . 6
  - 2.3.2 Les sauts : *codeGenSaut()* . . . 6
  - 2.3.3 Gestion des registres . . . 7
  - 2.3.4 Gestion des labels . . . 7
  - 2.3.5 Gestion de la pile . . . 7

3 Annexe **9**
- 3.1 Hiérarchie des classes dans le package Tree . . . 9

1 Introduction

Dans ce document, nous décrivons les démarches abordées pour parvenir à la conception du compilateur, programmé en Java, qui a pour but de vérifier et exécuter des programmes écrits en langage Deca. De plus, la description de l'architecture globale peut être une référence pour des développeurs dans le but éventuel de faire évoluer notre compilateur.

2 Architecture

2.1 Analyse lexicale et syntaxique

L'analyse lexicale et syntaxique est la première étape fondamentale dans notre compilateur. Elle a pour but de transformer le programme Deca fourni sous forme d'un arbre abstrait à travers différents tokens générés, arbre qui sera récupéré à la sortie de cette étape. Pour cela, l'implémentation a eu lieu au niveau des deux fichier DecaLexer.g4 et DecaParser.g4 du répertoire syntax (accessible à travers le chemin suivant : *src/main/antlr4/ensimag/deca/syntax*) pour faire, respectivement, les analyses lexicale et syntaxique. Au niveau du Lexer, nous ajoutons tous les caractères et mots réservés au langage Deca. Le Parser s'occupe quant à lui de récupérer les tokens générés par le lexer pour construire l'arbre du programme. Il va reconnaître des expressions puis ajouter à l'emplacement du token l'objet correspondant que ce soit un Assign, un IfThenElse ou autre.

2.1.1 Syntaxe sans objet

Comme le premier objectif était de compiler un programme affichant 'Hello, world!', nous nous sommes d'abord concentrés sur ce qu'il fallait faire pour créer l'arbre associé à ce programme. Pour cette étape, il suffit de compléter les fonctions ANTLR4 fournies en créant les objets correspondant à l'expression reconnu. Par exemple, si le token est PRINTLN, nous créons un objet de classe Println, présent dans le répertoire Tree. Enfin, il suffit de noter la bonne localisation des objets de l'arbre par rapport à leur position dans le programme.

2.1.2 Syntaxe avec objets

La partie avec objets est sensiblement similaire à la partie sans objet. Il a cependant fallu créer de nouvelles classes spécifiques à la programmation objet (This, MethodCall, DeclVar *etc*).

2.2 Analyse et vérification contextuelle

2.2.1 Manipulation des environnements

- **Implémentation de DecacCompiler.** Dans le DecacCompiler fourni, nous avons commencé par introduire un champ SymbolTable symbolTable qui stocke la table des symboles du programme Deca à compiler. Aussi, nous avons ajouté un autre champ envTypes (de type EnvironmentType) pour mémoriser les types prédéfinis ainsi que leurs noms et définitions. Chaque type de classe créé par l'utilisateur sera stocké dans cet environnement. Lors de la partie sans objet, nous avons travaillé dans un environnement unique sans empilement ce qui nous a permis d'ajouter un nouveau champ envExp dans DecacCompiler qui correspond à l'environnement local. Cette modification était un moyen pour mieux factoriser notre code et réduire le nombre de paramètres des méthodes de vérification. Mais, ceci n'était pas optimal pour la partie avec objets puisque les déclarations et les vérifications ne se font plus dans le même environnement global. Pour ne pas refaire le tour de toutes les classes dans le package Tree et mettre à jour toutes les méthodes, une des solutions pour faire les vérifications et les déclarations dans le bon environnement était de suivre les points suivants:

  Ø Créer une copie de l'environnement local récupéré depuis DecacCompiler.
  Ø Remplacer l'environnement local par celui de l améthode ou de la classe à analyser.
  Ø Faire les déclarations et les vérifications nécessaires dans cet environnement.
  Ø Remettre à jour l'environnement du DecacCompiler en y associant la copie conservée au préalable.

- **Décoration/enrichissement de l'arbre abstrait**. Dans le but de faciliter l'étape de génération de code, nous ajoutons des décorations aux niveaux des nœuds de l'arbre abstrait. Plus précisément, nous associons à chaque identificateur sa définition qui sera déterminée à travers les méthodes de vérification contextuelle (verifyExpr, verifyType ...) ainsi que le type pour les expressions. Ces décors sont ensuite affichés suite à l'appel des méthodes "prettyPrintChildren" qui permettent l'affichage de l'arbre correctement décoré. Quant à l'enrichissement de l'arbre, nous avons ajouté un nouveau nœud "ConvFloat" pour prendre en compte le cas où on associe un int à un float. On voit l'intervention de ce nœud au niveau de la méthode verifyRValue lorsque les deux opérandes sont de types différents. Ainsi, le cas (float ¡= int) ne sera pas considéré comme erreur contextuelle mais plutôt comme une mise à jour des types des opérandes.

2.2.2 Vérification contextuelle dans le package Tree

- **Compilateur sans objets**
  Pour la partie sans objet, le package tree était largement écrit avec toutes les classes fournies (sauf le nœud ConvFloat). Alors, pour faire la vérification contextuelle, on a fait le tour des fichiers du package pour implémenter à chaque fois la fonction verifyNœud dont le but est de vérifier les règles de la grammaire fournie au niveau du nœud sélectionné et ses fils si nécessaire. On lève une erreur contextuelle dans le cas échéant. Dans cette partie, toute la vérification contrxtuelle nécessite un seul parcours de l'arbre vu qu'on travaille dans un seul environnement.

- **Compilateur avec objets**
  Contrairement à la partie sans objet, la vérification se fait à travers 3 passes pour pouvoir gérer l'empilement des environnements des classes créées ainsi que ceux de leurs méthodes. Pour pouvoir faire les déclarations et la vérification, nous étions obligés de créer de nouvelles classes qui traduisent les nœuds manquants dans l'arbre abstrait comme ceux des champs et des méthodes (Voir annexe 4.1).

	classes ajoutées	Description
Les classes	Dot This New Null	Ces classes sont créées pour compléter l'implémentation des classes
Les méthodes	AbstractDeclMethod ListDeclMethod DeclMethod AbstractMethodBody MethodBody MethodAsmBody Return	En se référant au modèle de conception d'une classe, ces classes sont créées pour implémenter la déclaration des méthodes et leur vérification contextuelle
Les champs	AbstractDeclField ListDeclField DeclField	Toujours selon le modèle de création des classes et des méthodes, ces classes définissent les champs et la vérification des règles correspondantes de la grammaire
Les paramètres	AbstractDeclParam ListDeclParam DeclParam	Pareil que pour les champs mais associé aux paramètre
Appel des méthodes	AbstractMethodCall MethodCall	Ces classes servent à faire l'appel des méthodes et les vérifications nécessaires

Lors du premier parcours de l'arbre abstrait (passe 1), nous vérifions toutes les classes (nom et hiérarchie) à travers la fonction VerifyListClass

qui à son tour va faire appel à verifyClass. Si les règles de la grammaire sont respectées, ces classes seront déclarées et leurs environnements seront mis en place. Le deuxième parcours (passe 2) sert à vérifier les champs et la signature des méthodes des différentes classes via la fonction VerifyListClassMembers qui à son tour fait appel aux différentes fonctions de vérifications des méthodes (verifyListMethod, verifyMethodMembers , veriflParamMembers) et des champs (verifyListField, verifyFieldMembers). Les déclarations sont faites si aucune erreur contextuelle n'est détectée. Quant au troisième parcours (passe 3), nous nous concentrons sur les corps des classes, des méthodes et du "Main" du programme. Plusieurs fonctions de vérifications seront utilisées (verifyListClassBody, veriyClassBody, verifyMethodBody, verifyFieldBody. . . ect).

2.3 Génération de code

Concernant la génération du code assembleur, notre objectif était de mettre en place une architecture de code clair, utilisant au mieux le parcours de l'arbre et facile à débugguer.

2.3.1 Les expressions : *codeGenExpr()*

Une grande partie des noeuds de l'arbre étendent la classe *AbstractExpr*, en particulier pour le langage Deca sans objet. C'est pourquoi, nous avons eu l'idée de réaliser une fonction qui est héritée dans toutes les classes filles de cette classe mère, le but étant de créer une méthode récursive qui à chaque noeud la lance sur ses enfants au fur et à mesure que nous descendions dans l'arbre. D'où la naissance de la méthode *codeGenExpr*
La structure de cette méthode nous a été inspirée par les diapositives sur la partie C (cf. diapo 5/14). Tout d'abord, nous avons réalisé cette méthode dans le cas décrit dans les diapositives (opérations arithmétiques sur les entiers), puis avons cherché à étendre celle-ci sur les autres expressions. Ainsi, *codeGenExpr* est devenu notre méthode centrale lorsqu'il fallait gérer les calculs, assignations etc.
De plus, la structure de cette méthode présente plusieurs avantages : le premier est que nous mettons en paramètre le registre dans lequel nous souhaitons mettre le résultat de l'expression, ce qui nous permet de déjà gérer, en partie, la disponibilité des registres (cf. section 2.3.3). Le second est que chaque redéfinition de cette méthode est indépendante des autres. En effet dans chaque noeud la méthode a une fonction bien précise et ne dépend pas du résultat obtenu sur ses classes filles. Cela nous a permis d'avoir une grande souplesse lors du développement et de repérer les erreurs très rapidement, simplement en suivant le parcours de l'arbre lors de la génération du code.

2.3.2 Les sauts : *codeGenSaut()*

Un fois confronté aux structres de contrôles, nous nous sommes rendus compte que la méthode codeGenExpr ne suffisait pas à gérer toutes les opérations, et en particulier les instructions qui engendrait des instructions de type saut en assembleur. D'où la création de la méthode *codeGenSaut()*.
Ici encore une partie de la structure de la méthode nous a été donnée dans le polycopié ([**Gencode**] section 7.2). Nous avons alors implémenté la méthode pour les cas présentés dans le polycopié. Puis il nous a fallu étendre la méthode pour les autres expressions d'où l'ajout en paramètre d'un registre, qui avait la même fonction que celui de *codeGenExpr*. En effet, certaines expression *codeGenSaut()* faisaient appel à *codeGenExpr* pour obtenir un résultat ou alors reprenaient la même forme que cette dernière, ce qui faisait de la répétition de code à une ou deux lignes près mais nous n'avions pas de meilleure solution.
Ainsi, les avantages de cette méthode sont les mêmes que pour *codeGenExpr* si ce n'est qu'elle permet en plus de séparer les opérations engendrant des sauts des autres opérations.

2.3.3 Gestion des registres

Il était important de savoir quel registre était occupé et quel registre était libre lors de la génération du code, afin de pouvoir les utiliser. Pour ce faire, nous possédons une liste représentant l'état des différents registres. Ainsi, lorsque nous voulons utiliser un registre, nous changeons son état pour "occupé", et nous le libérons seulement lorsque les calculs que nous voulons effectuer sont terminés. Dans le cas où le dernier registre est utilisé, il est alors possible de stocker son contenu en haut de la pile, pour pouvoir facilement le récupérer une fois les opérations terminées. Cette dernière opération est gérée dans le code de *codeGenExpr* et *codeGenSaut*.

2.3.4 Gestion des labels

Les labels, en code assembleur, permettent d'aller de certaines parties du code à une autre sans passer par toutes les étapes intermédiaires. Ils représentent en quelques sortes des "balises" indiquant par exemple où commence une méthode. Dans le DecacCompiler, nous avons mis en place une Map regroupant tous les labels créés durant la génération du code. Cela nous évite de recréer plusieurs fois le même Label et il est simple de les récupérer pour les mettre à la suite des différentes instructions de saut. En parallèle, nous possédons également une Map des différents labels d'erreur. Ainsi, nous pouvons écrire le saut vers une erreur avant d'avoir générer le code de celle-ci, code se trouvant à la fin du programme.

2.3.5 Gestion de la pile

Lors de l'exécution du programme assembleur, il est essentiel de connaître à chaque instant l'état de la pile et de pouvoir tester des potentiels débordements. Nous avons donc implémenté plusieurs compteurs dans l'objet DecacCompiler afin de pouvoir stocker des variables, mais égalmement pour calculer les débordements de piles.
Nous possédons alors un compteur *GB*, un compteur *LB* et un compteur de variables temporaire. Les compteurs *GB* et *LB* nous permettent de savoir où nous stockons les variables afin de pouvoir facilement les retrouver dans la pile par la suite. Le compteur temporaire nous permet de compter le nombre de *PUSH* effectué dans un bloc, pour ensuite placer au début de ce bloc la commande *TSTO*, testant les débordements de pile avant qu'ils n'arrivent effectivement.
En effet, après avoir parcouru un bloc entier (le bloc principal *main* ou les blocs de fonction), nous pouvons alors prédire la place maximale qui va être attribuée dans la pile. Ainsi, il suffit de placer un *TSTO* en début de bloc, suivi d'un *BOV* sautant au label d'erreur *stack_overflow*, et d'un *ADDSP #d*, réservant *d* place en haut de la pile.

3 Annexe

3.1 Hiérarchie des classes dans le package Tree

NB : Les classes ajoutées sont surlignées en vert

- fr.ensimag.deca.tree.Tree
  - fr.ensimag.deca.tree.AbstractDeclClass
    - fr.ensimag.deca.tree.DeclClass
  - fr.ensimag.deca.tree.AbstractDeclField
    - fr.ensimag.deca.tree.DeclField
  - fr.ensimag.deca.tree.AbstractDeclMethod
    - fr.ensimag.deca.tree.DeclMethod
  - fr.ensimag.deca.tree.AbstractDeclParam
    - fr.ensimag.deca.tree.DeclParam
  - fr.ensimag.deca.tree.AbstractDeclVar
    - fr.ensimag.deca.tree.DeclVar
  - fr.ensimag.deca.tree.AbstractInitialization
    - fr.ensimag.deca.tree.Initialization
    - fr.ensimag.deca.tree.NoInitialization
  - fr.ensimag.deca.tree.AbstractInst
    - fr.ensimag.deca.tree.AbstractExpr
      - fr.ensimag.deca.tree.AbstractBinaryExpr
        - fr.ensimag.deca.tree.AbstractOpArith
          - fr.ensimag.deca.tree.Divide
          - fr.ensimag.deca.tree.Minus
          - fr.ensimag.deca.tree.Modulo
          - fr.ensimag.deca.tree.Multiply
          - fr.ensimag.deca.tree.Plus
        - fr.ensimag.deca.tree.AbstractOpBool
          - fr.ensimag.deca.tree.And
          - fr.ensimag.deca.tree.Or
        - fr.ensimag.deca.tree.AbstractOpCmp
          - fr.ensimag.deca.tree.AbstractOpExactCmp
            - fr.ensimag.deca.tree.Equals
            - fr.ensimag.deca.tree.NotEquals
          - fr.ensimag.deca.tree.AbstractOpIneq
            - fr.ensimag.deca.tree.Greater
            - fr.ensimag.deca.tree.GreaterOrEqual
            - fr.ensimag.deca.tree.Lower
            - fr.ensimag.deca.tree.LowerOrEqual
        - fr.ensimag.deca.tree.Assign
      - fr.ensimag.deca.tree.AbstractLValue
        - fr.ensimag.deca.tree.AbstractIdentifier
          - fr.ensimag.deca.tree.Identifier
        - fr.ensimag.deca.tree.Dot
      - fr.ensimag.deca.tree.AbstractMethodCall

- fr.ensimag.deca.tree.MethodCall
- fr.ensimag.deca.tree.AbstractReadExpr
  - fr.ensimag.deca.tree.ReadFloat
  - fr.ensimag.deca.tree.ReadInt
- fr.ensimag.deca.tree.AbstractStringLiteral
  - fr.ensimag.deca.tree.StringLiteral
- fr.ensimag.deca.tree.AbstractUnaryExpr
  - fr.ensimag.deca.tree.ConvFloat
  - fr.ensimag.deca.tree.Not
  - fr.ensimag.deca.tree.UnaryMinus
- fr.ensimag.deca.tree.BooleanLiteral
- fr.ensimag.deca.tree.CastExpr
- fr.ensimag.deca.tree.FloatLiteral
- fr.ensimag.deca.tree.InstanceOf
- fr.ensimag.deca.tree.IntLiteral
- fr.ensimag.deca.tree.New
- fr.ensimag.deca.tree.Null
- fr.ensimag.deca.tree.This
- fr.ensimag.deca.tree.AbstractPrint
  - fr.ensimag.deca.tree.Print
  - fr.ensimag.deca.tree.Println
- fr.ensimag.deca.tree.IfThenElse
- fr.ensimag.deca.tree.NoOperation
- fr.ensimag.deca.tree.Return
- fr.ensimag.deca.tree.While

- fr.ensimag.deca.tree.AbstractMain
  - fr.ensimag.deca.tree.EmptyMain
  - fr.ensimag.deca.tree.Main
- fr.ensimag.deca.tree.AbstractMethodBody
  - fr.ensimag.deca.tree.MethodAsmBody
  - fr.ensimag.deca.tree.MethodBody
- fr.ensimag.deca.tree.AbstractProgram
  - fr.ensimag.deca.tree.Program
- fr.ensimag.deca.tree.TreeList<TreeType>
  - fr.ensimag.deca.tree.ListDeclClass
  - fr.ensimag.deca.tree.ListDeclField
  - fr.ensimag.deca.tree.ListDeclMethod
  - fr.ensimag.deca.tree.ListDeclParam
  - fr.ensimag.deca.tree.ListDeclVar
  - fr.ensimag.deca.tree.ListExpr
  - fr.ensimag.deca.tree.ListInst