Ensimag Grenoble-INP

Compilateur DECA
Documentation de Conception

Auteurs :
Oussama Lamzaouri
Mohammed hadi Chamsi
Ziad Razani
Rong Li
Sofia Echarif

Tuteur :
Akram IDANI

27 janvier 2022

Table des matières

1 Architecture globale du projet

La compilation s'effectue en trois étapes. Dans un premier temps, le fichier .deca est donné en entrée du lexer afin de le réduire à un ensemble de mots (symboles, tokens) qui peut ensuite être donnés au parser. Le parser va construire un arbre syntaxique abstrait non décoré. Ensuite l'étape B ,la vérification contextuelle de l'arbre précédemment créé, vérifie que le programme écrit est valide et bien typé au sens des contextes. La dernière étape, l'étape C, génère le code assembleur pour la machine abstraite. L'arbre abstrait décoré passe à un programme assembleur (fichier .ass).

Nous implémentons donc, dans chacune de ces classes, également des méthodes permettant d'afficher des nœuds dans l'arbre abstrait primitif et de décompiler le fichier .deca, d'effectuer les vérifications contextuelles à la deuxième étape de compilation ainsi que des méthodes de génération de code .ass pour la troisième étape.

2 Étape A : Analyse lexicosyntaxique

Le compilateur associe à chaque "mot" du programme un symbole en vérifiant que "ces mots" appartient bien à la lexicographie du langage deca. Donc en ajoutant des mots et des règles de grammaire, on peut élargir ce langage.

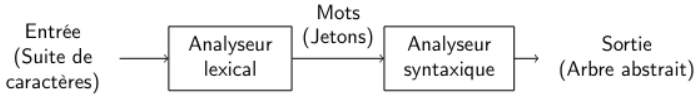

Figure 1 – Une passe dans l'étape A sur le programme source Deca.

- `src/main/antlr4/fr/ensimag/deca/syntax/DecaLexer.g4`
Ajouter tous les tokens au langage.
- `src/main/antlr4/fr/ensimag/deca/syntax/DecaParser.g4`
Ajouter tous les règles de grammaire du langage,
- `src/main/java/fr/ensimag/deca/syntax/DecaRecognitionException.java`
Ajouter des exceptions exécutées par le parser
- `src/main/java/fr/ensimag/deca/tree/`

Le paquetage `Tree` nous permet de construire l'arbre syntaxique abstrait à partir d'un fichier .deca. Toutes les classes dans ce paquetage découlent directement des différentes règles de grammaire qui permettent d'afficher des nœuds dans l'arbre abstrait primitif (`prettyPrintChildren(PrintStream s, String prefix)`, `iterChildren(TreeFunction f)`)et de décompiler le fichier .deca (`decompile(IndentPrintStream s)`).

Nous avons illustré la hiérarchie des classes qui héritent la classe Tree dans le figure 1.2 la page suivante.

Pour implémenter le concept de classe, `AbstractDeclMethod`, `DeclMethod`, `AbstractDeclField`, `DeclField`, `AbstractDeclParam`, `DeclParam`, `AbstractMethodBody`, `MethodBody`, `MethodAsmBody` ont été créées. Ce qui n'est pas indiqué dans le figure, ce sont les classes qui héritent la class `TreeList`. `ListDeclVar`, `ListInst`, `ListExpr`, `ListDeclClass` ont été fournis, aussi, nous avons créé `ListDeclMethod`, `ListDeclParam` et `ListDeclField` pour conserver une homogénéité dans l'arbre et de l'analyseur syntaxique.

Pour la partie "autre classes utiles" que l'on n'a pas listé, les classes `This`, `Null`, `New`, `Cast`, `InstanceOf` ont été créées comme une conséquence du concept de classe, ou on dit, Objet. L'implémentation des méthodes nécessite également de créer la classe `Return`. Les appels aux méthodes nécessite `MethodCall`, pour représenter un appel à une méthode. Il existe soit pour un appel dans une méthode d'une classe qui appelle une autre des méthodes de cette classe (ou superclass), soit pour un appel de méthode spécifiant l'instance de classe à laquelle il est appliqué. On doit aussi considérer l'appel comme `a.function(param)`, c'est-à-dire, l'accès aux paramètres, il est représenté par la classe `Selection` qui hérite donc de la classe `AbstractLValue`.

3 Étape B : Analyse contextuelle

La vérification contextuelle d'un programme Deca a besoin de trois passages. Il se fait avec les méthodes `verifyX` des classes dans le paquetage `Tree`, chacune de ces méthodes appel les méthodes `verifyX` de ces arbres fils et vérifie que les règles de grammaire correspondant à ce noeud sont bien respectées. Dans le cas échéant, une exception `ContextualError` est levée.

La première passe `verifyListClass` vérifie la déclaration des classes, ça veut dire, le nom des classes et la hiérarchie des classes. Si pour chaque classe, la super classe est déjà déclarée, la classe déclarée ne le soit pas déjà et en l'absence d'erreurs ajoute la classe a l'environnement types du compilateur, on passe à pass 2.

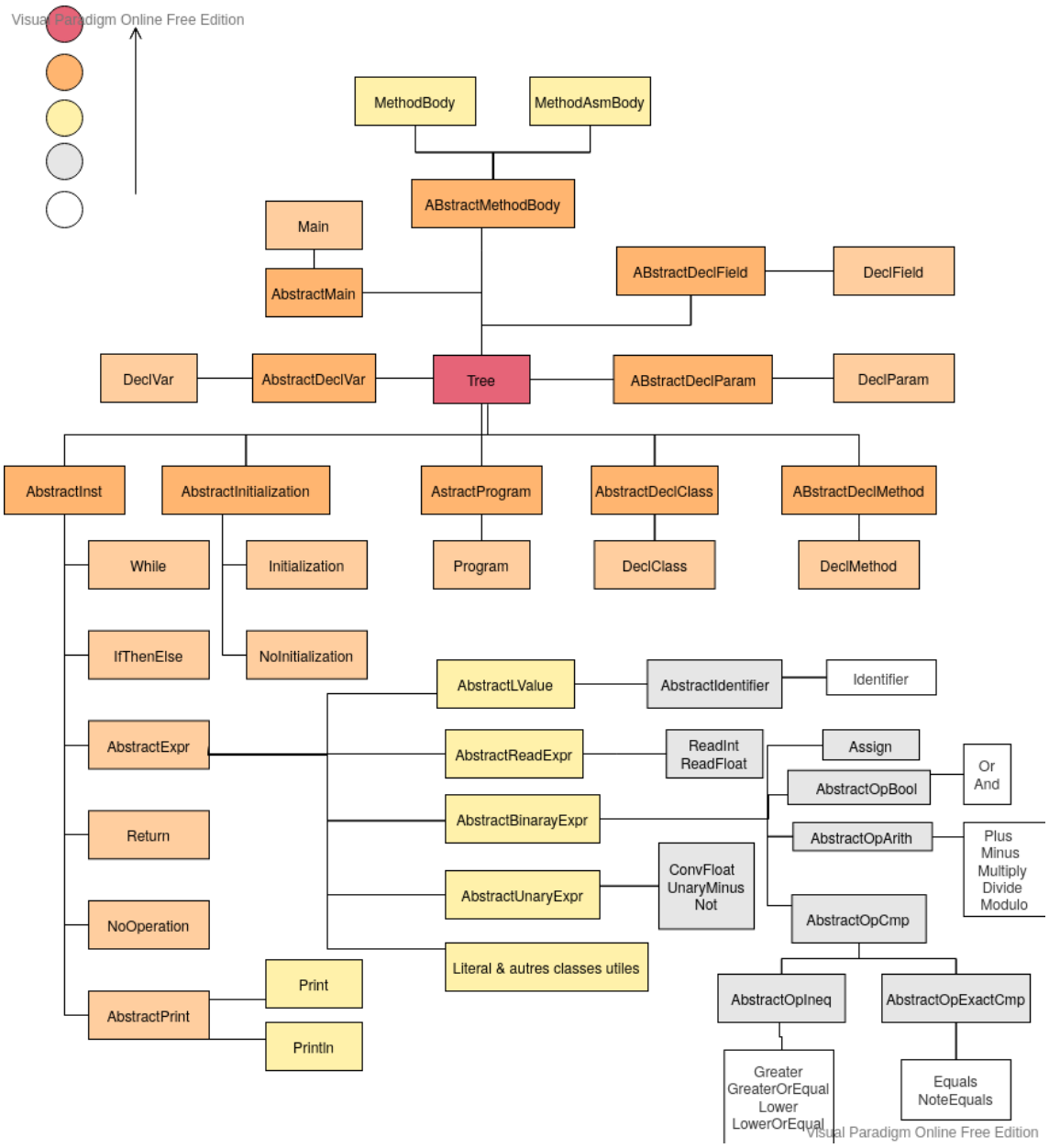

FIGURE 2 – hiérarchie des classes

La deuxième passe vérifie les déclarations des champs et la signature des méthodes. `verifyListClassMembers` appelle les methodes `verifyListField` et `verifyListDeclMethod` , elles appellent respectivement le méthodes `verifyDeclField`, qui vérifie si les champs et méthodes d'une classe écrasent les champs ou méthodes de ces superclasses, initialisent les définitions de ces champs et méthodes et les ajoutent a l'environnement members de la `ClassDefinition`, et `verifyDeclMethod` qui appelle la méthode `verfyDeclParam` pour vérifier les types des paramètres de la méthode, et ensuite les ajouter à la signature de la méthode.

Enfin, la troisième passe `verifyListClassBody` vérifie le corps des méthodes. Cette méthode appelle `verifyMethodBody` qui vérifie contextuellement l'initialisation des champs, les paramètres de la méthode et le corps des méthodes, en prenant l'environnement local(de la classe).

Afin de procéder à la vérification contextuelle, il est nécessaire de mettre en place des environnements. Un environnement représente un dictionnaire où l'on associe à chaque identifiant (symbole x, `int`) sa définition. Cela rendra la gestion des déclarations plus faisable, et facilitera l'identification des différents identifiants dans les instructions. Pour réaliser cela, les deux classes `EnvironmentExp` et `EnvironmentType` ont été mises à notre disposition. Il est à noter que la déclaration de variables ayant le même nom devrait être possible dans certains cas de figure, (Classe dont l'un des attributs est, par exemple, x, l'une des méthodes de la même classe déclare une variable locale x ; ou encore deux classes l'une héritant de l'autre ayant toutes les deux un attribut x). Afin de réaliser cela, un empilement d'environnements d'expressions est nécessaire. Cet empilement se fait en

spécifiant l'environnement parent lors de chaque création d'un nouvel environnement.

4 Étape C : Génération du code

On pourrait distinguer quatre parties dans cette étape, ces quatre parties sont facilement remarquables dans tout code assembleur généré par notre compilateur. En effet, regardons le code assembleur généré pour le programme deca suivant :

```
1  class A {
2      int a = 1;
3      int getA() {
4          return this.a;
5      }
6  }
7
8  {
9      A a = new A();
10     println(a.getA());
11 }
```

FIGURE 1.3 – Exemple de programme Deca

```
; start main program
; Main program
        TSTO #6
        BOV StackOverflowError
        ADDSP #6
        LOAD #null, R0
        STORE R0, 1(GB)
        LOAD code.Object.equals, R0
        STORE R0, 2(GB)
        LEA 1(GB), R0
        STORE R0, 3(GB)
        LOAD code.Object.equals, R0
        STORE R0, 4(GB)
        LOAD code.A.getA, R0
        STORE R0, 5(GB)
```

```
; Beginning of main instructions:
        TSTO #2
        BOV StackOverflowError
        ADDSP #2
        NEW #2, R2
        BOV HeapOverflowError
        LEA 3(GB), R0
        STORE R0, 0(R2)
        PUSH R2
        BSR init.A
        POP R2
        STORE R2, 6(GB)
        STORE R2, 0(SP)
        LOAD 0(R2), R2
        BSR 2(R2)
        LOAD R0, R1
        WINT
        WNL
        HALT
```

1. Construction de la table des méthodes

Dans cette partie, on empile pour chaque classe, et dans l'ordre de déclarations des classes, un pointeur vers l'adresse de la pile où sont rangées les étiquettes des méthodes de la classe parentes, puis des étiquettes de méthodes susceptibles d'être appelées lorsqu'une instance de la classe est créée, ces étiquettes incluent également celles des classes parentes n'ayant pas été redéfinies.

2. Génération de code pour le programme principal

C'est dans cette partie qu'on génère le code des instructions qui seront exécutées. En premier lieu, si le programme comporte une déclaration de variables, des cases seront réservées dans la pile, en dessous de la table des méthodes, pour ces variables et contiendront leurs éventuels initialisations. Ensuite, du code est généré pour la liste d'instructions. Les instructions ainsi que les initialisations peuvent êtres des expressions comportant des opérations logiques ou arithmétiques, des nombres, des identifiants, des appels de méthodes, des sélections ou des allocations.

```
code.Object.equals:
        PUSH R2
        LOAD -3(LB), R2
        CMP -2(LB), R2
        BNE code.Object.equals.false
        LOAD #1, R0
        BRA code.Object.equals.end
code.Object.equals.false:
        LOAD #0, R0
code.Object.equals.end:
        POP R2
        RTS
init.A:
        LOAD -2(LB), R1
        LOAD #1, R0
        STORE R0, 1(R1)
        RTS
code.A.getA:
        PUSH R3
        LOAD -2(LB), R3
        LOAD 1(R3), R0
        POP R3
        RTS
```

```
ZeroDivisionError:
        WSTR "Error: Use zero as division"
        ERROR
StackOverflowError:
        WSTR "Error: Stack Overflow"
        ERROR
HeapOverflowError:
        WSTR "Error: Heap Overflow"
        ERROR
NullObjectError:
        WSTR "Error : Object is null"
        ERROR
; end main program
```

3. Génération de code pour les méthodes

Dans cette partie, du code est généré pour toutes les méthodes des classes, le corps d'une méthode ressemble, en général, au corps du programme principal, sauf qu'ici, on travail dans une zone locale de la pile, où l'on doit retrouver l'adresse de l'instance de la classe appelante, puis les paramètres de la méthode. Par analogie, pour chaque méthode, un espace sera consacré pour les éventuels variables locales déclarées, dans la pile en dessous des paramètres, puis du code sera généré pour la liste d'instructions. Pour chaque classe, il existe une méthode `init` dont l'utilisateur ne soupçonne pas l'existence, qui sera appelé à chaque création d'une nouvelle instance de cette classe et dont le but est d'initialiser les attributs.

4. Génération de code pour le traitement des erreurs sémantiques

Dans cette partie, du code est généré pour spécifier le comportement de l'exécutable lorsqu'une erreur sémantique est rencontrée. Généralement, comme le montre cet exemple, ce comportement se résume à une terminaison anormale du programme avec un message d'erreur.