ENSIMAG

Projet GL

GL 59

Documentation de conception

WALID BAKIR

WALID ABIDI

ABDELOUAHAB MESTASSI

EL MEHDI SALAH-EDDINE

YASSIN EL KHALDI AHANNACH

1 Étape A

Cette partie du projet consistait à compléter les deux fichiers de syntaxe antlr4 ; DecaLexer.g4 et DecaParser.g4.

Cette partie servait à vérifier la syntaxe du code écrit en Deca. Le lexeur était assez facile, principalement en copiant et collant tous les jetons autorisés dans Deca à partir du poly.

Le parseur, cependant, a nécessité le remplissage de quelques classes java, l'ajout d'attributs et de constructeurs. Nous vérifions chaque jeton et fragment figurant dans le lexeur et vérifions leur placement et la manière correcte de les écrire dans un fichier Deca.

Pour la partie Sans Objet, les fichiers java et les types de retour pour chaque symbole terminal et non terminal étaient déjà donnés. Mais pour la partie avec objet, nous avons dû tout écrire à partir de zéro. Mais nous nous sommes appuyés sur la façon dont la première partie du fichier a été écrite et nous nous en sommes inspirés pour écrire le reste.

Nous n'avons pas trouvé trop difficile de tester cette partie, car nous n'avions pas d'autre partie à tester. Nous pouvions simplement écrire des instructions et les tester sur place, même si parfois nous nous sommes rendu compte que certaines choses étaient fausses après la réalisation des autres parties.

2 Étape B

Lors de cette étape, on vérifie la syntaxe contextuelle du programme Deca. On récupère l'arbre abstrait généré par l'étape A et on le décore en

ajoutant les définitions associées aux noeuds Identifier et les types associés aux noeuds qui dérivent de Expr. Dans le cas d'une erreur qu'il faut générer, on lève une exception ContextualError en spécifiant un message indiquant l'origine de l'erreur.

Le code gérant cette partie se trouve dans deux répertoires :

1) src/main/java/fr/ensimag/deca/context : ce répertoire contient

- Une classe abstraite **Type** qui décrit un type Deca, avec des métodes permettant de tester le type effectif d'une expression. Chaque classe qui implémente Type doit redéfinir sa méthode correspondante. Par exemple, IntType redéfinit la méthode boolean isInt() pour retourner true.

- Une classe abstraite **Definition** qui décrit la définition d'un identificateur. Cet identificateur peut être une classe (ClassDefinition), un identificateur dans une expression (ExpDefinition), un champ de classe (FieldDefinition), une méthode (MethodDefinition), un paramètre de méthode (ParamDefinition), un type (TypeDefinition), ou une variable (VariableDefinition). Chacune de ces classes redéfinit ses propres méthodes (Par exemple, MethodDefinition redéfinit boolean isMethod()) et définit des méthodes additionnelles utiles au traitement de la classe.

- Les deux classes décrivant les environnements : **EnvironmentExp** et **EnviromentType** : La première associe le symbole d'un identificateur à son ExpDefinition et la deuxième lui associe son TypeDefinition. La structure de donnée choisie pour représenter cette association est un HashMap qui prend comme clé le symbole de l'identificateur et comme valeur ExpDefinition ou TypeDefinition. Pour réaliser le chaînage des environnements, ces deux classes possèdent un attribut parentEnvironment correspondant à l'environnement de la classe mère d'une classe donnée.

- Une classe **Signature** qui décrit la signature d'une méthode. Une signature est une liste de types. La structure de donnée utlisée pour cette liste est une ArrayList<Type>.

2) src/main/java/fr/ensimag/deca/tree :

Ce répertoire contient un ensemble de classes abstraites **AbstractXyz** qui correspondent aux non-terminaux de la grammaire ainsi que les sous-classes correspondantes aux terminaux de la grammaire. Chaque classe abstraite possède une méthode abstraite verifyXyz. C'est cette méthode qui permet de faire les vérifications contextuelles et générer les erreurs. Cette méthode est ensuite implémentée dans les classes représentant les terminaux de la grammaire. Par exemple, la classe Program possède la méthode verifyProgram dont le corps appelle les méthodes des trois parcours de vérifications contextuelles. Dans le cas d'une erreur contextuelle, on arrête la vérification et cette méthode lève l'exception ContextualError qui prend en paramètres un message indiquant l'origine de l'erreur et l'emplacement dans le fichier deca du code provoquant l'erreur.

3 Étape C

La gestion des registres était un point très importante car c'est la base de génération du code assembleur pour ceci nous avons créé une classe **"RegsManager.java"** dans **"java.fr.ensimag.deca.codegen"**, qui contient plusieurs structures de données qui servent à enregistrer des informations comme les registres libres, les registres utilisés et aussi la position courante dans la pile ainsi que le nombre des variables locales et le nombre des paramètres et plusieurs autres information.

Toutes les variables de cette classe étaient privées pour éviter tout changement non désiré qui peut mener à des erreurs lors de la génération du code.

Les structures de données utilisées :

Des ArrayLists pour enregistrer l'état des registres que ce soit les libres ou les registre qui sont passés dans la pile qui sont manipulées à l'aide des fonctions suivante :

- **public GPRegister getReg() :** retourne un registre non utilisé
- **public void freeReg(DVal reg) :** libère le registre passé en paramètre
- **public void pushall() :** génère des instructions pour faire un push pour tous les registres utilisés
- **public void popall() :** génère des instructions pour faire un push pour tous les registres utilisés

— **public void pushregister(GPRegister reg) :** push le registre passé en paramètre dans la pile
— **public void freeall() :** libère tous les registres

Pour la gestion des Labels, premièrement nous avons créé une classe pour gérer les labels utilisés dans quelque classe comme "IfThenElse", “While”, “OR”et “AND”. Mais après nous avons changé de conception en mettant pour chaque classe ces propres labels qui seront définis dans chaque classe.Ce concept des Label est aussi un point fort du langage assembleur car il permet de passer à plusieurs endroits du code pour l'exécuter.

Aussi pour faciliter la génération du code nous avons implémenté la fonction “codePreGen” qui sert à chercher l'opérand du registre ou on enregistre le résultat d'une expression avant d'effectuer d'autres opérations.

Dans la classe Program où débute toute génération de code, on distingue entre deux cas : le cas sans objet ou il n y a pas de déclaration de classes (classes.size ==0) et le cas du langage deca avec objet (deca complet) ou il faut générer la table de méthode avec la méthode codeGenListTabledeMethode et ensuite la génération du code pour le programme principal par la méthode codeGenMain et enfin on génère de code des méthodes pour chaque classe en utilisant codeGenMethode. Dans cette dernière partie nous avons utilisé IMAprogram pour diviser le programme en plusieurs sous-programmes afin de générer le code bloc par bloc et aussi pour pouvoir gérer les Pushs et les Pops.et pour ceci on a introduit quelques modifications sur la classe decacCompiler.