Documentation de conception

Grenoble-INP ENSIMAG

Groupe 19

Salaheddine ASLOUNE

Wen LUO

Anass BOUBKRI

El Mehdi KHADFY

Amine BELFKIRA

Première partie : Les architectures (liste des classes et leurs dépendances)

Deuxième partie : Spécifications sur le code du compilateur et leurs justifications

Troisième partie : Description des algorithmes et structures de données employés.

Introduction

Ce document constitue une documentation des choix de conception du compilateur decac . Elle facilite la prise en main du compilateur et décrit l'organisation générale de l'implémentation.

1. Architecture générale :

Les fichiers source du projet peuvent être grossièrement divisés en deux parties. La première partie est liée à l'analyse lexicale et syntaxique et la deuxième contient tout ce qui est lié à la vérification contextuelle et génération du code. Dans cette partie on traitera les architectures ajoutées, selon les différentes étapes suivantes: Analyse lexicale et syntaxique, analyse contextuelle et génération de code.

1.1. Analyse lexicale et syntaxique:

Les fichiers correspondant à cette première étape sont contenus dans src/main/antlr4/fr/ensimag/deca/syntsax. Ce répertoire contient les fichiers sources pour le générateur d'analyseur ANTLR. La transformation de la suite des caractères du fichier d'entrée en suite de lexèmes (analyse lexicale) est effectuée dans le fichier DecaLexer.g4. On y trouve les différents tokens pris en compte dans l'analyse lexicale. Ensuite, vient l'étape de la transformation de la suite de lexèmes obtenues en un arbre de syntaxe abstraite. Les programmes syntaxiquement corrects sont tous définis dans le fichier DecaParser.g4. Le résultat de l'analyse lexicale sur un fichier deca peut être affiché par la commande test_lex fichier.deca, et celui de l'analyse syntaxique est donné par la commande test_synt fichier.deca. En guise d'exemple on applique ces deux commandes sur le code suivant:

```
{
   boolean x = true;
   int y = 5;
   while (x) {
      if (y<2) {
            x = false;
      } else if (y<5) {
            y = y - 3;
      } else {
            y = y - 1;
      }
      println("good");
   }
   println("very Good !!");
}
```

`test_lex` donne le résultat suivant:

```
OBRACE: [@0,163:163='{',<32>,14:0]
IDENT: [@1,168:174='boolean',<49>,15:3]
IDENT: [@2,176:176='x',<49>,15:11]
EQUALS: [@3,178:178='=',<21>,15:13]
TRUE: [@4,180:183='true',<8>,15:15]
SEMI: [@5,184:184=';',<35>,15:19]
IDENT: [@6,189:191='int',<49>,16:3]
IDENT: [@7,193:193='y',<49>,16:7]
EQUALS: [@8,195:195='=',<21>,16:9]
INT: [@9,197:197='5',<43>,16:11]
SEMI: [@10,198:198=';',<35>,16:12]
WHILE: [@11,203:207='while',<12>,17:3]
OPARENT: [@12,208:208='(',<28>,17:8]
IDENT: [@13,209:209='x',<49>,17:9]
CPARENT: [@14,210:210=')',<29>,17:10]
OBRACE: [@15,211:211='{',<32>,17:11]
IF: [@16,220:221='if',<10>,18:7]
OPARENT: [@17,222:222='(',<28>,18:9]
IDENT: [@18,223:223='y',<49>,18:10]
LT: [@19,224:224='<',<39>,18:11]
INT: [@20,225:225='2',<43>,18:12]
CPARENT: [@21,226:226=')',<29>,18:13]
OBRACE: [@22,227:227='{',<32>,18:14]
IDENT: [@23,240:240='x',<49>,19:11]
EQUALS: [@24,241:241='=',<21>,19:12]
FALSE: [@25,242:246='false',<9>,19:13]
SEMI: [@26,247:247=';',<35>,19:18]
CBRACE: [@27,256:256='}',<33>,20:7]
ELSE: [@28,257:260='else',<11>,20:8]
IF: [@29,262:263='if',<10>,20:13]
OPARENT: [@30,264:264='(',<28>,20:15]
IDENT: [@31,265:265='y',<49>,20:16]
LT: [@32,266:266='<',<39>,20:17]
INT: [@33,267:267='5',<43>,20:18]
CPARENT: [@34,268:268=')',<29>,20:19]
OBRACE: [@35,269:269='{',<32>,20:20]
IDENT: [@36,282:282='y',<49>,21:11]
EQUALS: [@37,284:284='=',<21>,21:13]
IDENT: [@38,286:286='y',<49>,21:15]
MINUS: [@39,287:287='-',<24>,21:16]
INT: [@40,288:288='3',<43>,21:17]
```

```
SEMI: [@41,289:289=';',<35>,21:18]
CBRACE: [@42,299:299='}',<33>,22:8]
ELSE: [@43,300:303='else',<11>,22:9]
OBRACE: [@44,304:304='{',<32>,22:13]
IDENT: [@45,317:317='y',<49>,23:11]
EQUALS: [@46,319:319='=',<21>,23:13]
IDENT: [@47,321:321='y',<49>,23:15]
MINUS: [@48,322:322='-',<24>,23:16]
INT: [@49,323:323='1',<43>,23:17]
SEMI: [@50,324:324=';',<35>,23:18]
CBRACE: [@51,333:333='}',<33>,24:7]
PRINTLN: [@52,342:348='println',<14>,25:7]
OPARENT: [@53,349:349='(',<28>,25:14]
STRING: [@54,350:355='"good"',<45>,25:15]
CPARENT: [@55,356:356=')',<29>,25:21]
SEMI: [@56,357:357=';',<35>,25:22]
CBRACE: [@57,362:362='}',<33>,26:3]
PRINTLN: [@58,367:373='println',<14>,27:3]
OPARENT: [@59,374:374='(',<28>,27:10]
STRING: [@60,375:388='"very Good !!"',<45>,27:11]
CPARENT: [@61,389:389=')',<29>,27:25]
SEMI: [@62,390:390=';',<35>,27:26]
CBRACE: [@63,392:392='}',<33>,28:0]
```

`test_synt` quand lui donne:

```
`> [14, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [14, 0] Main
      +> ListDeclVar [List with 2 elements]
      |  []> [15, 11] DeclVar
      |  ||  +> [15, 11] Identifier (boolean)
      |  ||  +> [15, 11] Identifier (x)
      |  ||  `> Initialization
      |  ||       `> [15, 15] BooleanLiteral (true)
      |  []> [16, 7] DeclVar
      |     +> [16, 7] Identifier (int)
      |     +> [16, 7] Identifier (y)
      |     `> Initialization
      |          `> [16, 11] Int (5)
      `> ListInst [List with 2 elements]
      []> [17, 3] While
      ||  +> [17, 9] Identifier (x)
      ||  `> ListInst [List with 2 elements]
```

```
||    []> [18, 7] IfThenElse
||    ||  +> [18, 10] Lower
||    ||  |  +> [18, 10] Identifier (y)
||    ||  |  `> [18, 12] Int (2)
||    ||  +> ListInst [List with 1 elements]
||    ||  |  []> [19, 11] Assign
||    ||  |      +> [19, 11] Identifier (x)
||    ||  |      `> [19, 13] BooleanLiteral (false)
||    ||  []> [20, 16] IfThenElse
||    ||  ||  +> [20, 16] Lower
||    ||  ||  |  +> [20, 16] Identifier (y)
||    ||  ||  |  `> [20, 18] Int (5)
||    ||  ||  +> ListInst [List with 1 elements]
||    ||  ||  |  []> [21, 11] Assign
||    ||  ||  |       +> [21, 11] Identifier (y)
||    ||  ||  |       `> [21, 15] Minus
||    ||  ||  |       +> [21, 15] Identifier (y)
||    ||  ||  |       `> [21, 17] Int (3)
||    ||  `> ListInst [List with 1 elements]
||    ||    []> [23, 11] Assign
||    ||         +> [23, 11] Identifier (y)
||    ||         `> [23, 15] Minus
||    ||         +> [23, 15] Identifier (y)
||    ||         `> [23, 17] Int (1)
||    []> [25, 7] Println
||         `> ListExpr [List with 1 elements]
||         []> [25, 15] StringLiteral ("good")
[]> [27, 3] Println
     `> ListExpr [List with 1 elements]
          []> [27, 11] StringLiteral ("very Good !!")
```

1.2. Analyse contextuelle:

Les modifications effectuées dans cette partie concernent principalement le dossier src/main/java/fr/enismag/deca/tree. Les vérifications et la détection des erreurs contextuelles ont été implémentés dans les fichiers de ce répertoire. Les fichiers concernant la partie sans-objet ont tous été fournis. Cependant, on a dû ajouter certains fichiers pour la partie objet. Voici la liste des classes ajoutées selon leurs utilités:

Fonctionnalités	Classes ajoutées
Traitement des champs	AbstractDeclField, ListDeclField, DeclField.
Déclaration des méthodes	AbstractDeclMethod, ListDeclMethod,

	DeclMethod.
Traitement des méthodes	AbstractMethodBody, MethodBody, MethodAsmBody, Return.
Appel des méthodes	MethodCall.
Traitement des paramètres	AbstractDeclParam, ListDeclParam, DeclParam.
Gestion des casts	Cast.
Allocation dynamique de mémoire dans le tas	New.
Autres fonctionnalités nécessaire dans la partie objet	Selection, This, NullLiteral, InstanceOf.
Environnement des Types du programme	EnvironmentType

Les classes abstraites AbstractDeclField, AbstractDeclMethod, AbstractMethodBody et AbstractDeclParam étendent la classe Tree. Ces classes abstraites sont implémentées respectivement par DeclField, DeclMethod, MethodAsmBody et DeclParam.

Les classes MethodCall, Cast, New, This et InstanceOf étendent la classe AbstractExpr.

La classe Return étend la classe abstraite AbstractInst et la classe Selection étend la classe AbstractLValue.

La classe EnvironmentType, quant à elle, a été définie dans le dossier src/main/java/fr/enismag/deca/context.

1.3. Génération de code:

Pour la partie de génération du code aucune classe autre que celles citées précédement n'a été ajoutée. Il a suffi d'implémenter les méthodes nécessaires permettant de générer le code assembleur. Ces méthodes se résument principalement en cinq méthodes, qu'on redéfinit au sein de chaque classe où on en a besoin. Ces méthodes sont: codeGenInst, codeGenPrint, codeGenLabelFalse, codeGenLabelTrue et codeGenReg. Il y a aussi d'autres méthodes utilisées dans cette phase, telles que CodeGenClassTable permettant de générer la table des méthodes. Plus de détails sur ces méthodes seront données dans la section 2.3 de ce document.

2. Spécifications sur le code du compilateur et leurs justifications

2.1. Analyse lexicale et syntaxique:

En ce qui concerne l'implémentation de l'étape d'analyse lexicale, quasiment la totalité des tokens à définir ont été donnés dans le polycopié avec le pseudo-code qui les définit. Les quelques ajouts effectués sont les suivants:

- Utilisation de la méthode doInclude dans l'inclusion de fichier:
  `INCLUDE : '#include' (' ')* '"' FILENAME '"' {doInclude(getText()); };`
- Distinction des deux types de commentaires:
  `fragment COMMENT : (('/*' .*? '*/') | ('//' .*? (EOL|EOF)));`

Pour l'analyse syntaxique, le squelette de la plupart des règles ont été fournis. Il n'a donc fallu que les implémentés. En revanche, on a ajouté certaines règles pour pouvoir englober la partie objet. Tout cela a été fait dans le fichier DecaParser.g4.

2.2. Analyse contextuelle:

Cette étape consistait principalement en l'implémentation des redéfinitions de la méthode verifyExpr dans les classes du dossier src/main/java/fr/enismag/deca/tree. Lors du premier parcours du code deca, nous appelons cette méthode afin d'effectuer les vérifications contextuelles. En fait, la vérification commence d'abord dans la classe Program présente dans src/main/java/fr/enismag/deca/tree, par la méthode verifyProgram. Cette méthode effectue les trois passes de vérification expliqués dans le polycopié. Les trois passes sont traitées respectivement par les méthodes verifyListClass, verifyListClassMembers et verifyListClassBody. Afin d'implémenter ces méthodes et en particulier les méthodes qui en découlent verifyClass, verifyClassMember et verifyClassBody, nous avons ajoutés dans la classe EnvironmentExp, du dossier src/main/java/fr/enismag/deca/context, les méthodes suivantes:

- `void add(EnvironmentExp otherEnvironment) throws ContextualError`

Cette méthode permet l'ajout du contenue de l"environnement donné en paramètre à l'environnement champ de la classe EnvironmentExp.

- `void addMethods(EnvironmentExp otherEnvironment,ClassDefinition cls, ClassDefinition SuperCls) throws ContextualError`

Cette méthode permet d'ajouter les méthodes déclarées dans le paramètre otherEnvironment et ceux déclarés dans la super-classe tout en gérant la redéfinition des méthodes héritées.

Après la vérification des classes, le programme lance la vérification du main qui lance à son tour la vérification des déclarations de variables globales et des instructions. Pour ce faire, on a eu besoin d'un environnement de types disponibles à l'utilisation. On a donc implémenté la classe EnvironmentType dans le répertoire src/main/java/fr/enismag/deca/context. Cette classe est instanciée au début du

programme pour déclarer les types BuiltIn, puis la classe Object, et après chaque déclaration d'une classe, un nouveau type est ajouté à cet environnement.

En ce qui concerne la décoration de l'arbre abstrait, il a suffi d'implémenter les méthodes prettyPrintChildren et iterChildren, tout en ajoutant les définitions à chaque nœud de l'arbre.

2.3. Génération de code:

Cette phase consistait principalement en l'implémentation de la méthode codeGenInst et des autres méthodes, annoncés dans la section 1.3, lorsqu'il est nécessaire.

CodeGenInst permet de générer le code assembleur de la classe dans laquelle elle est redéfinie. Et si nécessaire, elle met la valeur de l'opération dans un registre R. L'accès à cette valeur par la suite est assez simple, il suffit de prendre le dernier registre occupé.

```
   @Override
    protected void codeGenInst(DecacCompiler compiler){
        compiler.addInstruction(new LOAD(value,
genRegister(compiler)));
    }
```

src/main/java/fr/ensimag/deca/tree/IntLiteral.java

CodeGenReg fait en général la même chose que CodeGenInst et retourne en plus le registre utilisé. Ce registre peut être un offset de GB ou LB, ou bien un R registre selon le besoin.

```
    protected DVal codeGenReg(DecacCompiler compiler) {
        codeGenInst(compiler);
        return
Register.getR(compiler.getRegManager().getIndexLastRegister());
    }
```

src/main/java/fr/ensimag/deca/tree/AbstractExpr.java

CodeGenPrint est responsable de générer le code assembleur qui permet l'affichage pour les instructions **print** et **println.** On a aussi ajouté la méthode CodeGenPrintHexa qui fait pareil pour les instructions printx et printlnx.

```
 @Override
    protected void codeGenPrint(DecacCompiler compiler) {
        compiler.addInstruction(new LOAD(codeGenReg(compiler),
```

```
Register.R1));
        if (getDefinition().getType().isInt()) {
            compiler.addInstruction(new WINT());
        }
        else {
            compiler.addInstruction(new WFLOAT());
        }
    }
```

src/main/java/fr/ensimag/deca/tree/Identifier.java

CodeGenLabelTrue est utilisée dans les opérations sur les booléens. Elle admet en paramètre, en plus du DecaCompiler, un label. Cette méthode génère le code assembleur permettant un branchement conditionnel au label si la condition du booléen est vérifiée. De même pour le CodeGenLabelFalse si la condition n'est pas vérifiée.

```
   @Override
    protected void codeGenLabelTrue(DecacCompiler compiler , Label
next)
    {
        codeGenCMP(compiler);
        compiler.addInstruction(new BGT(next));
        verifyPop(compiler);
    }

  @Override
    protected void codeGenLabelFalse(DecacCompiler compiler, Label
next)
    {
        codeGenCMP(compiler);
        compiler.addInstruction(new BLE(next));
        verifyPop(compiler);
    }
```

src/main/java/fr/ensimag/deca/tree/Greater.java

CodeGenClassTable qui permet de générer la partie du code assembleur qui construit la table des méthodes. La méthode CodeGenClass a été implémentée dans DeclClass.java.

CodeGenClass est responsable de générer le code assembleur de l'initialisation des champs de la classe ainsi que le code de chaque méthode a l'aide de la fonction codeGenMethode et codeGenRestore. La méthode CodeGenClass a été implémentée dans DeclClass.java.

3. Description des algorithmes et structures de données employés.

Tout d'abord, nous utilisons un tableau de longueur fixe 16 (puisque ima n'a que seize registres) pour savoir si un registre est utilisé. Bien que nous ayons déclaré que la longueur du tableau était de 16, nous n'opérerons pas sur les premier et deuxième registres (R0, R1) qui sont des registres scratch. La raison pour laquelle nous ne déclarons pas un tableau de longueur quatorze est que nous n'avons pas à faire d'addition et de soustraction à chaque fois en fonction de l'indice donné.Nous avons également deux HashMap. Le première HashMap est utilisé pour stocker la relation entre les symboles et leurs offset GB. Cette idée de HashMap Symbol-Indice a été utilisée dans le traitement des méthodes pour faire la liaison entre chaque variable local et l'indice du registre qu'il l'a contient. Cette HashMap est une variable Statique dans la classe Identifier.java. Elle est réinitialisée au début du traitement de chaque méthode. La deuxième HashMap est utilisée pour stocker la relation parent-enfant entre chaque classe. La première raison du choix du HashMap est que sa complexité pratique de recherche est théoriquement O(1). La deuxième raison est que nous voulons enregistrer les paires clé-valeur. La paire clé-valeur du deuxième HashMap est relativement complexe. Sa valeur clé est l'indice GB de la classe parent, et la "valeur" de la valeur est l'ensemble(Set) des indices GB de la sous-classe. La raison du choix d'un ensemble est que, étant donné une classe parent, nous voulons obtenir toutes ses différentes sous-classes. Et ce HashMap est généralement appelé lorsque le mot clé instanceof est traité.La première HashMap est utilisée beaucoup plus que la deuxième. En effet, chaque fois qu'une variable globale est déclarée ou appelée, nous allons rechercher son index GB via cette première HashMap. Il en est de même pour les fonctions de chaque classe.

Pour gérer l'insuffisance des registres, une variable **needPop** a été déclarée dans la class RegistreManager dont on a l'accès par le biais du DecaCompiler. Cette variable est incrémentée lors de chaque PUSH et décrémentée après chaque POP. Par conséquent, elle représente le nombre de POP restant à faire pour récupérer toutes les données. Le PUSH est généralement fait dans la méthode genReg qui permet de récupérer un registre en appelant la méthode getIndexRegistre. Dans le cas où aucun registre n'est disponible, cette méthode return -1. Dans ce cas, la méthode genReg effectue un PUSH du dernier registre utilisé et incrémente la variable **needPop**. La méthode qui avait besoin de ce registre effectue une vérification. Elle est effectuée à la fin de presque toutes les méthodes codGenInst. Si un PUSH a été fait et selon la méthode en question, si elle a besoin de ce registre comme la méthode codeGenInst de AbstractOpArith qui fait un LOAD puis POP, elle appelle la méthode doLoadPop, sinon, comme la méthode de la class AbstractOpCmp qui n'a besoin que d'un POP, elle fait appelle à la méthode verifyPop.