GRENOBLE INP UGA Ensimag

Projet Génie Logiciel

Document de Conception

Groupe 41

Corentin HEUZÉ

Dorian VERNEY

Adrien AUBERT

Yuxuan LU

Yung Pheng THOR

Table de matière

1. Introduction

Ce document représente une documentation des choix de conception du compilateur decac. Il vise à faciliter la prise en main du compilateur en décrivant l'organisation générale de son implémentation.

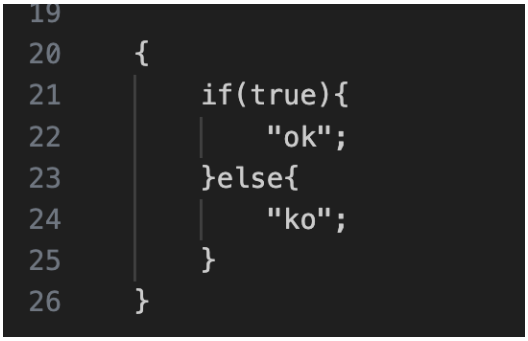
2. Architecture générale

Les fichiers source du projet peuvent être globalement divisés en deux parties distinctes. La première partie est associée à l'analyse lexicale et syntaxique, tandis que la deuxième englobe tous les éléments liés à la vérification contextuelle et à la génération du code. Dans cette section, nous examinerons les architectures ajoutées, en suivant les différentes étapes suivantes : l'analyse lexicale et syntaxique, l'analyse contextuelle et la génération de code.

2.1. Analyse lexicale et syntaxique

Les fichiers correspondant à cette première étape sont situés dans le répertoire *src/main/antlr4/fr/ensimag/deca/syntax*. Ce répertoire renferme les fichiers sources du générateur d'analyseur ANTLR. La transformation de la séquence de caractères du fichier d'entrée en séquence de lexèmes (analyse lexicale) est réalisée dans le fichier *DecaLexer.g4*, qui contient les différents tokens pris en compte dans cette phase. Ensuite, intervient l'étape de transformation de la séquence de lexèmes obtenue 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 en utilisant la commande *./test_lex fichier.deca*, tandis que celui de l'analyse syntaxique est obtenu grâce à la commande *./test_synt fichier.deca*. À titre d'exemple, ces deux commandes sont appliquées au code suivant :



./test_lex donne le résultat suivant :

```
OBRACE: [@0,609:609='{',<34>,20:0]
IF: [@1,615:616='if',<18>,21:4]
OPARENT: [@2,617:617='(',<32>,21:6]
TRUE: [@3,618:621='true',<4>,21:7]
CPARENT: [@4,622:622=')',<33>,21:11]
OBRACE: [@5,623:623='{',<34>,21:12]
STRING: [@6,633:636='"ok"',<46>,22:8]
SEMI: [@7,637:637=';',<37>,22:12]
CBRACE: [@8,643:643='}',<35>,23:4]
ELSE: [@9,644:647='else',<12>,23:5]
OBRACE: [@10,648:648='{',<34>,23:9]
STRING: [@11,658:661='"ko"',<46>,24:8]
SEMI: [@12,662:662=';',<37>,24:12]
CBRACE: [@13,668:668='}',<35>,25:4]
CBRACE: [@14,670:670='}',<35>,26:0]
```

./test_synt donne le résultat suivant :

```
`> [20, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [20, 0] Main
      +> ListDeclVar [List with 0 elements]
      `> ListInst [List with 1 elements]
         []> [21, 4] IfThenElse
             +> [21, 7] BooleanLiteral (true)
             +> ListInst [List with 1 elements]
             |  []> [22, 8] StringLiteral (ok)
             `> ListInst [List with 1 elements]
                []> [24, 8] StringLiteral (ko)
```

2.2. Analyse contextuelle:

Les modifications apportées dans cette section se concentrent principalement sur le répertoire *src/main/java/fr/enismag/deca/tree*. Les vérifications et la détection des erreurs contextuelles ont été implémentées dans les fichiers de ce répertoire. Tous les fichiers liés à la partie sans objet ont été fournis. Cependant, pour la partie objet, certains fichiers ont dû être ajoutés. Voici la liste des classes ajoutées, classées selon leurs utilités :

Fonctionnalités	Classes ajoutées
Déclaration des champs	AbstractDeclFields, ListDeclFields, DeclFields
Déclaration des paramètres	AbstractDeclParams, ListDeclParams, DeclParams
Déclaration des méthodes	AbstractDeclMethods, ListDeclMethods, DeclMethods
Traitement des méthodes	AbstractMethodBody, MethodBody, Return
Appel des méthodes	MethodCall
Gestion des casts	Cast
Déclaration d'un objet	New
Autres	InstanceOf, NullLiteral, Selection, This

2.3. Génération de code

Afin de générer le code assembleur, plusieurs classes ont été créées dans ./src/main/java/fr/ensimag/deca/codegen pour une meilleure gestion de l'implémentation et une meilleure structuration.

Fonctionnalités	Classes ajoutées
Gestion des registres disponibles ou non pour le calcul d'expressions	DummyRegisterAllocator
Création de messages adaptés pour la détection d'erreurs à l'exécution	ErrorMessage
Gestion de l'unicité des labels pour les expressions, les ifThenElse et les while	ManagementLabel
Gestion de la pile pour le placement des éléments dans la pile (Global base et Local base)	Stack
Petite classe représentant une position servant à garantir l'unicité des labels dans *ManagementLabel*	Position

Ces éléments sont instanciés dans la classe DecacCompiler afin d'avoir une constante accessibilité dans les méthodes de génération de code. Cela permet également de centraliser l'information et éviter de surcharger en paramètres les méthodes et ainsi gagner en efficacité.

3. Spécifications sur le code du compilateur et leurs justifications

3.1. Analyse lexicale et syntaxique

En ce qui concerne l'implémentation de l'étape d'analyse lexicale, pratiquement tous les tokens à définir ont été fournis dans le polycopié, accompagnés du pseudo-code qui les définit. Les quelques ajouts effectués sont les suivants :

- Distinction des deux types de commentaires
- Utilisation de la méthode doInclude dans l'inclusion de fichier

```
// Comments
COMMENTS: '//' (~('\n'))* { skip(); };
MULTI_LINE_COMMENTS: '/*' .*? '*/' { skip(); } ;

// Include
fragment FILENAME: (LETTER | DIGIT | '_' | '-' | '.')+ ;
INCLUDE: '#include' (' ')* '"'FILENAME'"'  { doInclude(getText()); };
```

Pour l'analyse syntaxique, le squelette de la plupart des règles a été fourni. Il a donc simplement fallu les implémenter. Tout cela a été fait dans le fichier *DecaParser.g4*.

3.2. Analyse contextuelle

Cette étape consistait principalement à mettre en œuvre les 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 pour effectuer les vérifications contextuelles. En réalité, 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 expliquées dans le polycopié. Les trois passes sont traitées respectivement par les méthodes *verifyListClass*, *verifyListClassMembers* et *verifyListClassBody*.

Dans la méthode *verifyListClassMembers*, deux nouvelles méthodes ont été intégrées : *verifyListDeclFieldMember* et *verifyListDeclMethodMember*. Ces méthodes sont responsables de la vérification contextuelle des déclarations de champs et de méthodes à l'intérieur d'une classe.

Pour la méthode *verifyListClassBody*, nous avons introduit deux nouvelles méthodes spécifiques : *verifyListDeclFieldBody* pour les champs et *verifyListDeclMethodBody* pour les méthodes.

Afin de mettre en œuvre ces méthodes, en particulier les méthodes qui en découlent *verifyClass*, *verifyClassMember* et *verifyClassBody*, nous avons ajouté dans la classe *EnvironmentType*, du dossier *src/main/java/fr/enismag/deca/context*, la classe *object*, le type *null* et les méthodes suivantes :

- *getType(Symbol key)*
- *getDefinition(Type type)*
- *declareClass(Symbol symbol, TypeDefinition typeDef)*
  Cette méthode permet l'ajout d'un nouveau type à l'environnement de type
- *subType(Type type1, Type type2)*
  Cette méthode permet de vérifier la relation de sous-typage.
- *assign_compatible(Type type1, Type type2)*
  Cette méthode permet de vérifier la compatibilité pour l'affectation.
- *cast_compatible(Type type1, Type type2)*
  Cette méthode permet de vérifier la compatibilité pour la conversion.

Après la vérification des classes, le programme procède à la vérification du main, qui à son tour initie la vérification des déclarations de variables globales et des instructions.

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

3.3. Génération de code

Concernant l'implémentation de génération de code, le but était de factoriser un maximum en utilisant intelligemment l'héritage pour produire un code fonctionnel, maintenable et scalable en vue de potentiels ajouts de fonctionnalités. Dans chaque classe (y compris les classes abstraites), du répertoire *./src/main/java/fr/enismag/deca/tree* les méthodes *codeGenInst* et *codeGenPrint* sont utilisées pour produire le code assembleur dans le fichier .ass. Haut dans la hiérarchie, plusieurs fonctions principales, servant à produire le code assembleur de chaque expressions, appellent automatiquement les *codeGensInst* adéquates grâce au typage largement présent. Avec cette technique, il est uniquement nécessaire de faire appel aux instructions assembleur dans les classes représentant les feuilles de l'arbre, avec souvent, l'utilisation de fonctions *instr*, *instrBin* pour encore plus de factorisation et de clarté.

A propos de l'initialisation des éléments; variables, classes, méthodes, une méthode *codeGenDecl[elem]* est implémentée afin d'initialiser l'élément dans la pile, que ce soit en global ou en local pour une fonction ou même dans le tas pour un field.

```
@Override
public void codeGenDeclVar(DecacCompiler compiler)
{
    compiler.add(new Line(comment:""));
    compiler.add(new Line(decompile()));

    DAddr dAddr =  compiler.getStack().newGlobal();
    if (type.getType().isClass()) {
        this.initialization.codeGenInit(compiler, dAddr);
    } else if (!(initialization instanceof NoInitialization)){
        this.initialization.codeGenInit(compiler, dAddr);
    }
    varName.getVariableDefinition().setOperand(dAddr);
}
```

Un autre point important est la clarté du code assembleur produit. Le fichier .ass se compose d'abord de la partie construction de la *table des méthodes*, puis de la partie *Main* et *end_Main,* de la partie *déclaration des classes* avec les *champs* et *méthodes* et enfin la gestion des potentielles *erreurs* à l'exécution.

4. Description des algorithmes et structures de données employés

4.1. Partie Contextuelle

Pour la partie contextuelle, *HashMap* est utilisé pour stocker *EnvTypes* et *EnvExp*, compte tenu de la possibilité d'une augmentation exponentielle du nombre de classes pouvant être exécutées dans le programme. Par conséquent, nous avons besoin d'une structure de données suffisamment rapide pour identifier rapidement l'objet correspondant à chaque exécution. *EnvType* est utilisé pour stocker le symbole de type et sa définition de type correspondante, tandis que *EnvExp* est utilisé pour stocker les symboles de variables, de classes, de méthodes, de champs et de paramètres à chaque niveau, ainsi que leurs définitions d'expression correspondantes. L'utilisation de la structure *HashMap* permet d'effectuer ces opérations en temps constant *O(1)*.

4.2. Partie Codegen

Afin de gérer correctement la génération de code, il a été nécessaire d'utiliser des structures de données adaptées.
Dans la classe MangementLabel, les HashMap sont très utiles pour stocker les labels que nous déclarons pour les différentes expressions booléennes, les expressions ifThenElse ainsi que pour les While. Un label doit être associé à sa position et doit être récupéré depuis sa structure de donné d'où il est stocké avec un coût minimal. C'est pourquoi, la structure HashMap est adéquate puisqu'elle répond aux problématiques avec un coût en O(1) pour un *get*.
Par ailleurs, afin de créer la table des méthodes en début de génération de code, une structure de type *ArrayList* nous a été utile afin de gérer les redéfinitions de méthodes, les méthodes héritées… pour chaque classe des fichiers .deca. Avec celle-ci, tester l'appartenance à une classe ou encore récupérer la définition d'une méthode héritée a été facilement réalisable *et* efficace.