Documentation de conception

Samy Ahjaou Hamza Bouihi Omar Guessous
Grégoire Sename Mamadou Thiongane

Groupe 9 – Équipe 42

26 janvier 2024

Table des matières

1 Introduction

Le développement d'un compilateur représente un défi significatif dans le domaine de la programmation informatique, mêlant des concepts complexes d'analyse syntaxique, de traitement du langage et d'optimisation du code. Ce document a pour but d'offrir une vue d'ensemble détaillée et spécifique de notre projet de compilateur, en se concentrant sur les aspects qui ne sont pas couverts dans la documentation standard fournie par les enseignants.

L'objectif principal de ce document est de guider les développeurs qui souhaiteraient maintenir, améliorer ou étendre les fonctionnalités de notre compilateur. Ainsi, au lieu de se plonger dans les détails de base du code, ce document met en lumière les spécificités de notre implémentation : l'architecture globale, les spécifications supplémentaires du code, et les choix des algorithmes et structures de données qui distinguent notre projet. Ces éléments sont cruciaux pour comprendre les décisions de conception sous-jacentes et les stratégies d'implémentation qui ont été adoptées.

Nous aborderons l'architecture de notre compilateur en mettant l'accent sur la liste des classes, leurs interactions et dépendances. Cette section aidera à comprendre comment les différentes composantes du compilateur fonctionnent ensemble pour réaliser l'analyse, la compilation et l'optimisation du code.

Ensuite, les spécifications sur le code du compilateur seront discutées, en mettant en évidence les modifications ou extensions que nous avons apportées. Les justifications de ces changements seront fournies pour donner un aperçu de notre processus de réflexion et des défis rencontrés.

Enfin, nous détaillerons les algorithmes et structures de données spécifiques que nous avons choisis ou développés, en expliquant pourquoi ils ont été privilégiés par rapport à d'autres options. Cette section vise à éclairer le lecteur sur les choix techniques qui sous-tendent le fonctionnement efficace de notre compilateur.

Ce document, en évitant les listings de code détaillés et en se concentrant sur des informations supplémentaires et contextuelles, servira de guide complet pour tout développeur entrant dans le projet, assurant ainsi une compréhension profonde et une facilité de transition pour la maintenance et l'évolution du compilateur.

2 Architecture générale

2.1 Objectif

Cette section a pour but de présenter une vue globale de l'architecture de notre compilateur, en mettant un accent particulier sur les évolutions et les ajouts réalisés lors de la transition vers la programmation orientée objet. Initialement, pour la partie sans objet du projet, l'architecture fournie par nos enseignants était adéquate et n'a nécessité aucune modification majeure. Cette continuité nous a permis de nous concentrer sur les aspects fondamentaux du traitement du langage sans devoir repenser l'infrastructure de base.

2.2 Vue d'ensemble de l'architecture

Lorsque nous avons abordé la partie orientée objet, une évolution de l'architecture s'est avérée nécessaire. À partir de l'étape A, correspondant à l'analyse syntaxique, nous avons commencé à intégrer des éléments supplémentaires dans notre architecture. Cette expansion a été guidée par les concepts présentés dans le polycopié, notamment les pages dédiées à la grammaire concrète et abstraite (pages 57 et 62).

Notre objectif était de construire sur les fondations de l'architecture sans objet existante tout en y ajoutant la flexibilité et les fonctionnalités requises par la programmation orientée objet. Ainsi, l'introduction de nouvelles classes, champs et méthodes est devenue une étape indispensable pour gérer les spécificités de cette approche. Nous avons soigneusement veillé à ce que ces ajouts s'alignent avec la structure originale pour assurer une cohérence et une intégrité architecturales tout en répondant aux exigences et aux possibilités offertes par l'orientation objet.

3 Spécifications

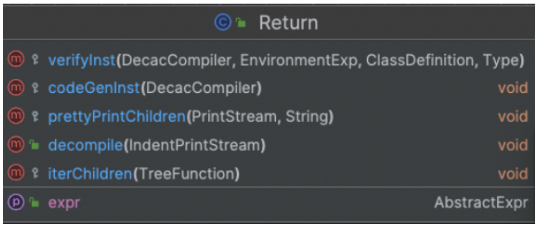
3.1 Analyse lexicale et analyse syntaxique

Pendant la réalisation de la partie A, qui englobe l'analyse syntaxique pour la programmation orientée objet, nous avons progressivement généré des fichiers Java dans le répertoire src/main/java/fr/ensimag/tree. Ces fichiers Java sont souvent des extensions de classes abstraites, notamment dans le cas des instructions telles que return.java, this.java, new.java, Selection.java, et Method.call. Chacun de ces fichiers Java contient des méthodes fondamentales telles que verifyInst, verifyExpr, decompile, codeGen, et prettyPrint, qui sont essentielles pour manipuler et transformer les objets du compilateur.

En ce qui concerne les attributs des classes, leur définition a été guidée par la grammaire abstraite présentée dans le polycopié. Pour illustrer ce processus, prenons l'exemple de la classe return. Cette classe a nécessité la création du fichier return.java qui suit une architecture spécifique permettant de représenter correctement le concept de retour dans le langage compilé. La mise en place de ces structures de classes personnalisées a été essentielle pour répondre aux besoins spécifiques de notre compilateur orienté objet.

Ici par exemple, nous avons compris que Return etait une instruction qui avait comme attribut une expression que nous avons appele expr, puis comme methode verifyInst, codeGenInst, prettyPrintChildren, decompile, iterChildren.

```
INST →
      EXPR
    | PRINT
    | IfThenElse[ EXPR LIST_INST LIST_INST]
    | NoOperation
    | Return[ EXPR]
    | While[ EXPR LIST_INST]
```



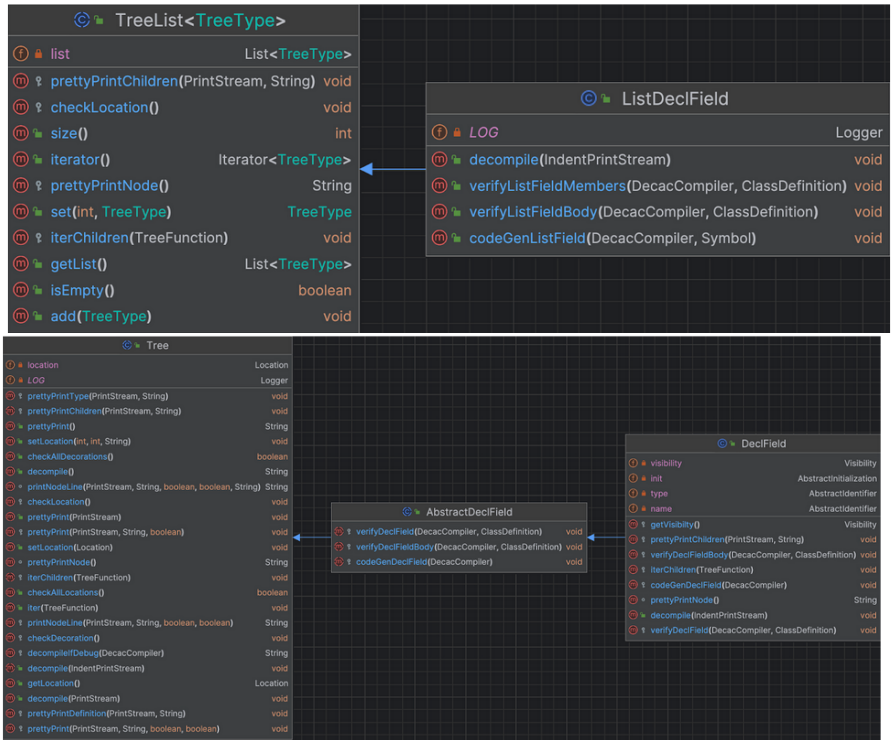
Nous avons suivi un processus similaire pour toutes les autres instructions et expressions, comme cela est clairement illustré dans le diagramme exhaustif de l'architecture de la partie orientée objet que vous trouverez en annexe.

En ce qui concerne les fichiers .java qui héritent de la classe Tree, notre démarche a suivi une approche méthodique. Tout d'abord, nous avons fondé notre conception sur la grammaire du polycopié, de la même manière que nous l'avons fait pour les instructions et les expressions. Ensuite, nous nous sommes appuyés sur un exemple spécifique, en l'occurrence la déclaration d'une variable, pour élaborer notre structure. Ainsi, pour chaque élément du compilateur, nous avons créé un ensemble de fichiers qui reflètent une hiérarchie bien définie.

Cette hiérarchie se compose généralement de trois types de fichiers : Un fichier abstrait héritant de la classe Tree, qui établit la base pour la représentation de l'élément. Un fichier

de déclarations des attributs, où nous avons spécifié les attributs et les propriétés spécifiques nécessaires pour l'élément. Un fichier qui hérite de la classe TreeList, représentant une liste de plusieurs déclarations de cet élément particulier. Pour illustrer ce processus, prenons l'exemple de la déclaration des champs dans une classe. Notre hiérarchie de fichiers organisait la déclaration des classes de la manière suivante, avec chaque fichier remplissant un rôle spécifique dans le processus de compilation. Cette approche structurée et organisée a été fondamentale pour garantir que chaque élément du compilateur est correctement représenté, avec des attributs et des méthodes appropriés, tout en facilitant les étapes ultérieures de développement, notamment les étapes B et C.

```
LIST_DECL_FIELD → [DECL_FIELD*]
DECL_FIELD →
    DeclField↑Visibility[IDENTIFIER IDENTIFIER INITIALIZATION]
```



3.2 Analyse contextuelle

Une fois que nous avons achevé l'étape A, nous disposons désormais de toute l'architecture nécessaire pour passer à la suite de notre projet. Abordons maintenant la partie B, qui implique la décoration de l'arbre construit lors de l'étape précédente. Cette phase se divise en plusieurs sous-parties distinctes, chacune ayant un rôle spécifique à jouer.

3.2.1 Passe 1

Lors de la première passe, notre objectif est de vérifier le nom des classes et de construire la hiérarchie de classes. Pour ce faire, nous établissons un environnement initial, que nous stockons dans le fichier src/main/java/fr/ensimag/deca/context/EnvironnementType.java.

Cet environnement, connu sous le nom `env_types_predef`, contient les noms des différentes classes présentes dans le programme. Chacune de ces classes dispose d'un profil incomplet,

comprenant des informations sur sa super-classe, mais sans inclure l'environnement des champs et des méthodes de la classe.

Pour réaliser cette étape, des modifications ont été apportées au fichier src/main/java/fr/ensimag/deca/tree/decl plus précisément à la méthode verifyClass pour verifier si la super classe existe deja, si on peut heriter de cette superclasse et aussi si elle est deja declaree , comme on peut le voir dans le code ci-dessous :

```
protected void verifyClass(DecacCompiler compiler) throws ContextualError {
    EnvironmentType envTypes = compiler.environmentType;
    TypeDefinition def = envTypes.defOfType(superClass.getName());

    if (def == null) {
        throw new ContextualError("La classe mère n'existe pas", getLocation());
    }

    if (!def.isClass()) {
        throw new ContextualError("Vous ne pouvez pas hériter de ce que vous avez
        écrit", getLocation());
    }

    try {
        ClassDefinition cdef  = (ClassDefinition) def;
        ClassType ct = new ClassType(name.getName(), getLocation(),cdef);

        // ct.getDefinition().setNumberOfMethods(cdef.getNumberOfMethods());
        // ct.getDefinition().setNumberOfFields(cdef.getNumberOfFields());

        envTypes.declare_type(name.getName(), ct.getDefinition());
        name.setDefinition(ct.getDefinition());
        superClass.setDefinition(def);
        name.setType(ct);
    } catch (EnvironmentType.DoubleDefException e) {
        throw new ContextualError("Classe déjà définie", getLocation());
    }
}
```

3.2.2 Passe 2

Lors de la deuxième passe, nous concentrons nos efforts sur la validation des champs et des signatures des méthodes au sein des différentes classes du programme. Cette étape est cruciale pour garantir la cohérence et la conformité des structures de données et des fonctionnalités au sein de chaque classe.

Pour accomplir cette tâche, nous capitalisons sur l'environnement construit au cours de la première passe. Plus précisément, nous héritons de l'environnement stocké dans l'attribut `en_types` , qui est transmis par le non-terminal program. Cette continuité dans la gestion de l'environnement nous permet de bâtir un environnement plus complet et informatif pour chaque classe.

Pour réaliser ces vérifications, nous avons mis en œuvre les méthodes verifyMembers dans différents fichiers, en suivant une approche méthodique pour garantir la cohérence de l'ensemble du processus. Notre démarche a commencé par la mise en place de la méthode verifyDecField dans le fichier DeclField.java. Cette première étape a été cruciale pour s'assurer de la compatibilité des informations entre la classe courante (currentClass) et sa super-classe. En poursuivant, nous avons ensuite implémenté la méthode verifyDeclMethod dans le fichier De-

clMethod. Cette étape a consisté à vérifier la signature de la méthode, en veillant à ce qu'elle corresponde à la méthode verifyDeclParam que nous avons également développée simultanément. La méthode verifyDeclParam prend en argument le compilateur et renvoie le type de chaque paramètre, à condition que ce type ne soit pas void. Ces types de paramètres sont ensuite ajoutés à la signature de la méthode.

Enfin, la méthode verifyDeclFMethod a été mise en place pour garantir que la signature de la méthode, ainsi que le type renvoyé par celle-ci, sont en conformité avec la méthode déclarée dans la super-classe. Cette vérification est essentielle pour maintenir la cohérence dans la hiérarchie des classes.

Voici à quoi ressemble notre implémentation de verifyDeclMethod qui représente une partie cruciale pour cette sous-partie :

```
protected void verifyDeclMethod(DecacCompiler compiler, ClassDefinition currentClass)
        throws ContextualError {
    Type typeReturn = type.verifyType(compiler);
    Signature sig = params.verifyListDeclParam(compiler);
    EnvironmentExp envExpSuper = currentClass.getSuperClass().getMembers();
    ExpDefinition def = envExpSuper.get(name.getName());
    // System.out.println(def + "\n\n\n");

    if (def != null && !def.isMethod()) {
        throw new ContextualError("methode déjà définie dans la classe mère en tant que
        field", getLocation());
    } else if (def != null && def.isMethod()) {
        MethodDefinition defMethode = def.asMethodDefinition("ce n'est pas une
        définition de méthode", getLocation());
        Signature sig2 = defMethode.getSignature();
        Type type2 = defMethode.getType();

        if (sig.isSameSignature(sig2) && (type2.sameType(typeReturn) || typeReturn
                .asClassType("Override impossible, verifiez  le type que renvoie votre
                fonction", getLocation())
                .isSubClassOf(type2.asClassType("Override impossible, verifiez  type
                que renvoie votre fonction",
                        getLocation())))) {
            try {
                MethodDefinition mDef = new MethodDefinition(type2, getLocation(), sig,
                defMethode.getIndex());
                name.setDefinition(defMethode);
                currentClass.getMembers().declare(name.getName(), mDef);

                // currentClass.incNumberOfMethods();

            } catch (EnvironmentExp.DoubleDefException e) {
                throw new ContextualError("methode déjà definie", getLocation());
            }
        } else if (!sig.isSameSignature(sig2)) {
            throw new ContextualError("Override impossible, verifiez la signature de
            votre fonction", getLocation());
        } else {
            throw new ContextualError("Override impossible, verifiez le type que
            renvoie votre fonction", getLocation());
        }
    } else if (def == null) {
        try {
            int indexPrevious = currentClass.getNumberOfMethods();
```

```
                MethodDefinition mDef = new MethodDefinition(typeReturn, getLocation(),
                sig, indexPrevious + 1);
                currentClass.incNumberOfMethods();
                currentClass.getMembers().declare(name.getName(), mDef);
                name.setDefinition(mDef);
            } catch (EnvironmentExp.DoubleDefException e) {
                throw new ContextualError("methode deja définie", getLocation());
            }
        }
    }
```

3.2.3 Passe 3

Au cours de la troisième passe, notre objectif principal est de réaliser une vérification exhaustive des blocs, des instructions, des expressions et des initialisations au sein du code source. Pour ce faire, nous construisons un environnement complet qui englobe non seulement les champs et les méthodes, mais également les paramètres des méthodes et les variables locales. Cette étape implique la mise en œuvre de différentes méthodes de vérification, notamment verifyDeclFieldBody, verifyDeclMethodBody, et verifyParamBody. Chacune de ces méthodes est conçue pour garantir la validité et la cohérence des éléments spécifiques du code. En ce qui concerne les champs, notre attention se porte sur la vérification de leur initialisation. Il est essentiel que les champs soient correctement initialisés pour éviter tout comportement indésirable du programme.

Dans le contexte de verifyDeclMethodBody, nous avons étendu nos vérifications pour inclure les paramètres des méthodes. Pour réaliser cette vérification, nous avons développé une fonction additionnelle, verifyDeclParamBody, qui joue un rôle essentiel. Cette fonction prend en argument le compilateur, localEnv (représentant l'environnement des expressions des paramètres, créé à l'intérieur de la méthode verifyDeclMethodBody), ainsi que currentClass. La fonction verifyDeclParamBody parcourt chaque paramètre, vérifie son type, l'ajoute à l'environnement, et le déclare correctement. Ce processus est fondamental pour garantir que les méthodes fonctionnent conformément à leurs spécifications, en tenant compte des types et des valeurs attendues des paramètres.

De plus, au cours de cette phase, nous avons également implémenté la méthode verifyExpr pour les instructions spécifiques à la programmation orientée objet, à savoir This, Return, New, Selection, et MethodCall. Ces instructions présentent des comportements particuliers et doivent être vérifiées avec précision pour garantir une interopérabilité correcte dans le cadre de la programmation orientée objet.

3.3 Génération de code

Par soucis de praticité, nous avons choisi d'ajouter des attributs à la classe `DecacCompiler` dans le but d'avoir accès à ceux-ci à n'importe quel endroit de notre arbre. Après coup, nous regrettons de ne pas avoir plus utilisé le répertoire `src/main/java/fr/ensimag/deca/codegen`, ce qui aurait rendu notre code plus structuré.

L'attribut entier `d` nous permet d'allouer correctement l'espace mémoire dans la pile pour chaque instanciation de variable.

`idReg` permet de gérer l'indice du registre que l'on manipule lors d'une instruction assembleur et d'incrémenter celui-ci pour en manipuler un autre. De plus, grâce à la méthode `compiler.getOptions().regMax()`, nous sommes en mesure de manipuler exclusivement les registres qui nous sont autorisés (par exemple, lors de la compilation avec `decac -r 4`).

`errors` est un tableau de booléens dont les éléments sont initialisés à `false` puis, lorsque que le code assembleur génère une instruction susceptible d'engendrer une erreur, nous affectons au booléen correspondant la valeur `true`, ce qui permet de générer uniquement les labels des erreurs nécessaires lors de la compilation d'un programme.

`tstoCurr` et `tstoMax` permettent de calculer l'entier `#d1` présent dans l'instruction `TSTO #d1`. En effet, nous utilisons `tstoCurr` pour compter le nombre maximal d'instructions `PUSH` consécutifs puis pour mettre à jour `tstoMax`, qui aura pour valeur finale celle de l'entier `d1` attendue.

`addSP` est un compteur qui est incrémenté à chaque initialisation de variables, méthodes... pour pouvoir placer SP au bon endroit lors de l'instruction `ADDSP #addSP`.

L'attribut `currentProgram` correspond à l'objet de classe `IMAProgram` sur lequel les instructions du compilateur sont générées. Les méthodes `beginBlock` et `endBlock` permettent de créer une instance de la classe `IMAProgram` afin de générer indépendamment les instructions d'un bloc donné puis d'ajouter cette suite d'instructions à la fin du programme principal et ainsi gérer l'ordre de génération des instructions.

4 Algorithmes et structures de données

4.1 Structures de données

L'attribut `classAdresses` est une table de hachage `Map<Symbol, DAddr>` qui associe à chaque classe son adresse dans la mémoire. En effet, cela est utile pour retrouver l'adresse de la superclasse dans la table des méthodes.

`VTable` est une classe héritée de la classe `HashMap<Symbol, Label[]>` et utilisée dans un algorithme que nous détaillerons dans la section suivante pour la construction de la table des méthodes pour la passe 1 de la partie "avec objet".

`ObjectClass` est une classe du répertoire `src/main/java/fr/ensimag/deca/codegen` contenant l'ensemble des méthodes permettant de gérer les instructions en assembleur associées à la classe `Object`.

L'attribut `sourceFiles` de la classe `DecacOptions`, initialement une `ArrayList<File>` a été modifiée en `HashSet<File>` de sorte qu'un fichier apparaissant plusieurs fois sur la ligne de commande ne soit compilé qu'une seule fois.

4.2 Algorithmes

Pour les expressions arithmétiques, nous avons suivi les algorithmes de la slide 5/14 de la présentation sur l'étape de génération de code, présentés en figure 2. Dans notre code, cela se traduit par l'implémentation des méthodes `mnemo`, `dVal` et `codeGenExpr`.

Opérande d'une expression atomique

```
<dval(IntLiteral↑n)> := #n
<dval(Identifier↑symb)> := @symb
<dval(_[ _ _] )> := ⊥
```

Mnémonique d'un opérateur binaire

```
<mnemo(Plus)> := ADD
<mnemo(Minus)> := SUB
<mnemo(Mult)> := MUL
```

Code pour calculer *e* dans R*n* (utilisant uniquement R0 et R*n* ... RMAX)

```
<codeExp(e, n)>
avec <dval(e)>≠⊥
  := LOAD <dval(e)>, Rn

<codeExp(op[ e1 e2] , n)>
avec <dval(e2)>≠⊥
  := <codeExp(e1 , n)>
     <mnemo(op)> <dval(e2)>, Rn
```

```
<codeExp(op[ e1 e2] , n)>
avec <dval(e2)>=⊥ et n=MAX
  := <codeExp(e1 , n)>
     PUSH Rn ; sauvegarde
     <codeExp(e2 , n)>
     LOAD Rn, R0
     POP Rn  ; restauration
     <mnemo(op)> R0, Rn
```

```
<codeExp(op[ e1 e2] , n)> avec <dval(e2)>=⊥ et n<MAX
  := <codeExp(e1 , n)>
     <codeExp(e2 , n+1)>
     <mnemo(op)> Rn+1, Rn
```

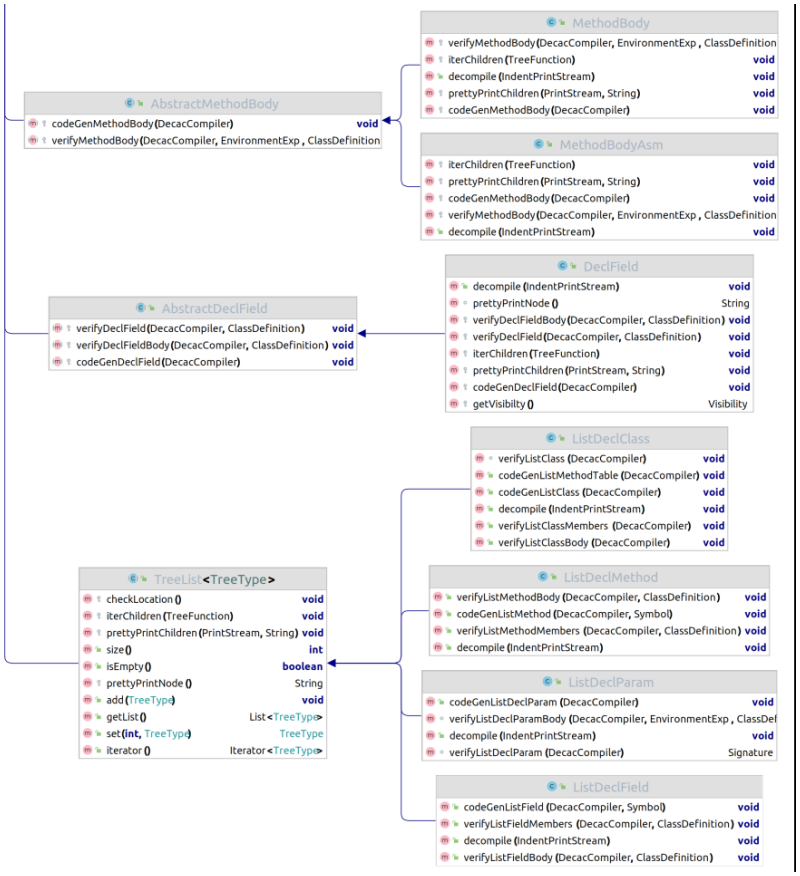
FIGURE 2 – Génération de code pour expressions arithmétiques

Pour les expressions booléennes, encore une fois, nous avons suivi les algorithmes détaillés en pages 221 et 225 du polycopié du projet. Cela se traduit dans notre code par l'implémentation de la méthode `code`.

Pour construire la table des étiquettes `vTable`, on l'initialise en associant au symbole de la classe `Object` une liste contenant l'étiquette de la méthode `equals` puis, lors d'un héritage ou d'une redéfinition, on associe au symbole de la classe une liste contenant les étiquettes des méthodes héritées. Ensuite, pour chaque méthode définie ou redéfinie dans cette classe, on ajoute son étiquette à la liste au bon indice à l'aide de l'attribut `index` de sa `MethodDefinition`.

5 Annexe


AbstractDeclParam
verifyDeclParamBody (DecacCompiler, EnvironmentExp , ClassDefiniti
codeGenDeclParam (DecacCompiler) void
verifyDeclParam (DecacCompiler) Type
DeclParam
iterChildren (TreeFunction) void
prettyPrintChildren (PrintStream, String) void
verifyDeclParamBody (DecacCompiler, EnvironmentExp , ClassDefiniti
decompile (IndentPrintStream) void
verifyDeclParam (DecacCompiler) Type
codeGenDeclParam (DecacCompiler) void
AbstractDeclClass
verifyClass (DecacCompiler) void
codeGenMethodTable (DecacCompiler) void
codeGenClass (DecacCompiler) void
verifyClassMembers (DecacCompiler) void
verifyClassBody (DecacCompiler) void
DeclClass
codeGenMethodTable (DecacCompiler) void
decompile (IndentPrintStream) void
verifyClass (DecacCompiler) void
prettyPrintChildren (PrintStream, String) void
codeGenClass (DecacCompiler) void
verifyClassBody (DecacCompiler) void
verifyClassMembers (DecacCompiler) void
iterChildren (TreeFunction) void
AbstractDeclMethod
verifyDeclMethod (DecacCompiler, ClassDefinition) void
verifyDeclMethodBody (DecacCompiler, ClassDefinition) void
codeGenMethodTable (DecacCompiler, Symbol) void
codeGenMethod (DecacCompiler, Symbol) void
DeclMethod
decompile (IndentPrintStream) void
iterChildren (TreeFunction) void
verifyDeclMethod (DecacCompiler, ClassDefinition) void
codeGenMethodTable (DecacCompiler, Symbol) void
codeGenMethod (DecacCompiler, Symbol) void
prettyPrintChildren (PrintStream, String) void
verifyDeclMethodBody (DecacCompiler, ClassDefinition) void
This
iterChildren (TreeFunction) void
verifyExpr (DecacCompiler, EnvironmentExp , ClassDefinition) Type
prettyPrintNode () String
isImplicit () boolean
decompile (IndentPrintStream) void
prettyPrintChildren (PrintStream, String) void
Return
setExpr (AbstractExpr) void
prettyPrintChildren (PrintStream, String) void
iterChildren (TreeFunction) void
verifyInst (DecacCompiler, EnvironmentExp , ClassDefinition, Type)
decompile (IndentPrintStream) void
getExpr() AbstractExpr
codeGenInst (DecacCompiler) void
Tree
prettyPrint (PrintStream) void
getLocation () Location
prettyPrint (PrintStream, String, boolean, boolean) void
prettyPrintNode () String
checkDecoration () void
prettyPrintChildren (PrintStream, String) void
prettyPrint () String
setLocation (int, int, String) void
prettyPrintDefinition (PrintStream, String) void
setLocation (Location) void
checkLocation () void
decompile (PrintStream) void
decompile () String
iter (TreeFunction) void
decompile (IndentPrintStream) void
prettyPrint (PrintStream, String, boolean) void
checkAllDecorations () boolean
checkAllLocations () boolean
decompileIfDebug (DecacCompiler) String
printNodeLine (PrintStream, String, boolean, boolean, String) String
printNodeLine (PrintStream, String, boolean, boolean) String
iterChildren (TreeFunction) void
prettyPrintType (PrintStream, String) void
AbstractInst
verifyInst (DecacCompiler, EnvironmentExp , ClassDefinition, Type)
codeGenInst (DecacCompiler) void
decompileInst (IndentPrintStream) void
New
codeGenInst (DecacCompiler) void
prettyPrintChildren (PrintStream, String) void
setExpr (AbstractIdentifier) void
getIdent() AbstractIdentifier
decompile (IndentPrintStream) void
verifyExpr (DecacCompiler, EnvironmentExp , ClassDefinition) Type
iterChildren (TreeFunction) void
Selection
prettyPrintChildren (PrintStream, String) void
decompile (IndentPrintStream) void
verifyExpr (DecacCompiler, EnvironmentExp , ClassDefinition) Type
codeGenInst (DecacCompiler) void
iterChildren (TreeFunction) void
MethodCall
decompile (IndentPrintStream) void
prettyPrintChildren (PrintStream, String) void
codeGenInst (DecacCompiler) void
verifyExpr (DecacCompiler, EnvironmentExp , ClassDefinition) Type
iterChildren (TreeFunction) void


MethodBody
verifyMethodBody(DecacCompiler, EnvironmentExp , ClassDefinition
iterChildren(TreeFunction) void
decompile(IndentPrintStream) void
prettyPrintChildren(PrintStream, String) void
codeGenMethodBody(DecacCompiler) void
AbstractMethodBody
codeGenMethodBody(DecacCompiler) void
verifyMethodBody(DecacCompiler, EnvironmentExp , ClassDefinition
MethodBodyAsm
iterChildren(TreeFunction) void
prettyPrintChildren(PrintStream, String) void
codeGenMethodBody(DecacCompiler) void
verifyMethodBody(DecacCompiler, EnvironmentExp , ClassDefinition
decompile(IndentPrintStream) void
DeclField
decompile(IndentPrintStream) void
prettyPrintNode() String
verifyDeclFieldBody(DecacCompiler, ClassDefinition) void
verifyDeclField(DecacCompiler, ClassDefinition) void
iterChildren(TreeFunction) void
prettyPrintChildren(PrintStream, String) void
codeGenDeclField(DecacCompiler) void
getVisibilty() Visibility
AbstractDeclField
verifyDeclField(DecacCompiler, ClassDefinition) void
verifyDeclFieldBody(DecacCompiler, ClassDefinition) void
codeGenDeclField(DecacCompiler) void
ListDeclClass
verifyListClass(DecacCompiler) void
codeGenListMethodTable(DecacCompiler) void
codeGenListClass(DecacCompiler) void
decompile(IndentPrintStream) void
verifyListClassMembers(DecacCompiler) void
verifyListClassBody(DecacCompiler) void
TreeList<TreeType>
checkLocation() void
iterChildren(TreeFunction) void
prettyPrintChildren(PrintStream, String) void
size() int
isEmpty() boolean
prettyPrintNode() String
add(TreeType) void
getList() List<TreeType>
set(int, TreeType) TreeType
iterator() Iterator<TreeType>
ListDeclMethod
verifyListMethodBody(DecacCompiler, ClassDefinition) void
codeGenListMethod(DecacCompiler, Symbol) void
verifyListMethodMembers(DecacCompiler, ClassDefinition) void
decompile(IndentPrintStream) void
ListDeclParam
codeGenListDeclParam(DecacCompiler) void
verifyListDeclParamBody(DecacCompiler, EnvironmentExp , ClassDef
decompile(IndentPrintStream) void
verifyListDeclParam(DecacCompiler) Signature
ListDeclField
codeGenListField(DecacCompiler, Symbol) void
verifyListFieldMembers(DecacCompiler, ClassDefinition) void
decompile(IndentPrintStream) void
verifyListFieldBody(DecacCompiler, ClassDefinition) void