Documentation de conception

Equipe 16

Ange Romuald Ossohou

Nadir Ait Lahmouch

Hamza Benjelloun

Oussama Fennane

Younes Zaibila

30 janvier 2020

Sommaire :

1° Liste des fichier java ajoutés :

Pour compléter le paquetage de dérivation de l'arbre, on ajouté les fichiers java suivants :

Partie objet :

- AbstractDeclField.java
- AbstractDeclMethod.java
- AbstractDeclParam.java
- AbstractMethodBody.java
- DeclField.java
- DeclMethod.java
- DeclParam.java
- ListDeclField.java
- ListDeclMethod.java
- ListDeclParam.java
- MethodAsmBody.java
- MethodBody.java
- MethodCall.java
- Return.java
- Selection.java
- This.java

Partie objet/sans objet:

- Cast.java
- InstanceOf.java
- New.java
- Null.java

Gestion de registre :

- RegisterManager.java

2° Conception de l'étape B (vérification contextuelle) :

La partie B a pour but de vérifier la syntaxe contextuelle de Deçà décrite dans la grammaire abstraite et concrète fournie par le polycopié, mais avant de commencer les passes nécessaires pour vérifier que le code deçà donné en entrée du compilateur est correct notre compilateur a besoin de deux environnements :
- l'environnement type qui contient tous les types prédéfinis (Int Type, Float Type, Boolean Type, Null Type, Void Type , la classe Objet...) et qui s'enrichit au fur et à mesure avec les nouvelles classes. Techniquement parlant elle contient une Map qui lie chaque Symbol avec sa définition.

- l'environnement Exp qui lie chaque méthode, variable, Class avec sa définition, elle contient au début de compilation que la méthode equals déjà défini dans la classe Object puis elle s'enrichit au fur et à mesure avec les différentes variables déclarées. Techniquement parlant l'Environnement dispose de deux attributs HashMap qui lie chaque Symbol avec sa définition et d'un attribut parent, parce qu'on va manipuler une liste chaînée des HashMap.

Comme notre compilateur a besoin dans chaque déclaration de vérifier que le type de la variable existe déjà on a modifié le constructeur du DécaCompiler et on a ajouté l'Environnement Type déjà prédéfini comme attribut de plus on a ajouté quelques fichiers qui représentent les non terminaux qui ne sont pas fournis par le squelette de nos enseignants par exemple les fichiers (Return. Java, New. Java, Null java ...).

Maintenant on a tous les fichiers nécessaires pour vérifier si le code donné en entrée est cohérent avec la syntaxe contextuelle de deçà décrite dans le document[Syntaxe Contextuelle]. Notre compilateur divise la vérification en 2 grandes parties.

-La première partie c'est de vérifier que les classes sont bien définies et cette partie elle-même contient 3 passes la première passe on vérifie que les classes sont bien déclarées, elle correspond à la fonction vérifylistclass qui est appelée par la racine de l'arbre abstraite Programme, et dans cette passe on remplit notre environnement type contenu dans le décacompiler, puis on passe à la deuxième passe qui consiste à vérifier que la déclaration des méthodes et fields au sein de chaque classe est bien respecté comme décrit dans le langage deçà, en effet dans cette passe on enrichit l'environnement Exp de chaque classe avec les définitions de chaque méthode, ensuite on passe à la troisième passe cette passe consister à vérifier que le body de chaque méthode est bien présenté cette passe qui ressemble bien à la passe faite dans la deuxième partie que je vais expliciter au-dessous.

-La deuxième partie, comporte une seule passe, elle commence par VerifyListDeclvar qui s'occupe de vérifier que le type choisi pour chaque variable

existe bien dans Environnement type et que le type de l'initialisation est cohérent avec le type déclaré c'est-à-dire on vérifie que le type de déclaration est lui-même le type de l'expression d'initialisation de plus dans le cas d'initialisation d'une variable du type Float avec un Int on fait un ConvFloat qui convertit l'entier a un réel ainsi on accepte l'initialisation d'une variable de type classe1 avec une autre variable qui a comme type une sous-classe de la classe1; puis on vérifie que le nom de variable n'existe pas déjà dans notre environnement exp alors on recourt à notre méthode get implémenter dans le fichier Environnement Exp.Jav qui fait un parcour récursif en partant de l'environnement courant jusqu'au null car en fait chaque environnement à son environnement parent par exemple si on définit une méthode dans une classe le parent de l'environnement exp de la méthode est l'environnement exp de la classe et le parent de l'environnement exp de la classe est null, la méthode get return le symbole s'il existe déjà sinon null, donc on peut facilement vérifier si le symbole existe ou non.
Après on passe au plus important c'est l'utilisation des variables déclarées ainsi vérifié le corps de chaque bloc, il s'agit de VerifyListInst qui parcourt une liste des instis et fait appel à verifyinst qui est une méthode abstraite du fichier Abstract Ints.java, et elle est implémentée dans les classes terminales: -(IfThenElse, While, New, Return …) , de plus on implémente verifyinst dans une autre classe importante qui est AbstractExpr qui donne naissance à une autre méthode qui est VerifyExpr, qui vérifie que les opérations binaires arithmétiques et de comparaison ainsi que les opérations unaires sont bien écrites dans le code source.

En ce qui concerne les options de compilation on a ajouté plusieurs attributs a la classe Compiler Option, un boolean pour chaque option afin que le compiler détecte bien l'option entrée par l'utilisateur, et une liste chaînée des chemins des fichiers qui est vide au début, de plus on a implémenté la méthode parseArgs qui prend en paramètres les arguments de compilation et essaye de détecter de quelle option s-il, et vérifie aussi si les options introduites sont compatibles entre eux par exemple on ne peut pas utiliser l'option v et p en même temps.
Parmi les options que notre compilateur sait gérer sont:

+ l'option b qui affiche une bannière indiquant le nom de l’équipe.

+ l'option p arrête decac après l’étape de construction de l’arbre, et affiche la décompilation de ce dernier (I.e. s’il n’y a qu’un fichier source à compiler, la sortie doit être un programme).

+ l'option v arrête decac après l’étape de vérifications c'est-à-dire après l'étape B.

+ l'option ne supprime pas les tests de débordement à l’exécution débordement arithmétique, débordement mémoire .

+ l'option r X limite les registres banalisés disponibles à R0 ... R{X-1}, avec X entre 4 et 16 .

+ l'option d'active les traces de debug. Répéter l’option plusieurs fois pour avoir plus de trace.

+ l'option P s’il y a plusieurs fichiers sources, lance la compilation des fichiers en parallèle (pour accélérer la compilation).

à titre d'information pour que l'option P fonction bien il faut que les fonctions et les variables utilisées dans la programmation soient non statices sinon il se peut que deux threads changent une variable statice à la fois et donc on n'aura pas le résultat souhaité.
Pour ajouter d'autres options il suffit d'ajouter le boolean qui correspond à cette option ainsi de modifier le switch dans la fonction parseArgs et ajouter l'option dans le manuel affiché par la fonction display usage lors d'une fausse utilisation de la commande decac.

3° Conception de l’étape C (Génération de code) :

RegisterManager:

Après plusieurs tentatives infructueuses de gérer bien les registres partagés entre les différentes sections (Opération Arithmétique, Opérations de comparaison, opérations unaires, opérations entre boolean ...), On a eu l'idée d'implémenter un RegisterManager qui s'occupe de l'allocation et le free de chaque registre. Notre RegisterManager dispose de plusieurs fonctions indispensables pour le bon fonctionnement de notre compilateur, On commence d'abord par expliciter les différents attribue de cette classe, On dispose d'une liste chaînée nommée ListDispo qui contient le numéro des registres disponibles pour le moment, et une autre liste chaînée ListPush qui contient un tuple qui (i, k) avec i le numéro de registre en question et le k est le nombre de fois qu'on a pu hé ce registre et on a un StackPush pour bien enregistrer l'ordre de hêtre qu'on a fait depuis le début de compilation, de plus il dispose d'un compteur qui est initialisé au début à 0 et un autre attribut qui calcule le nombre de hêtre qu'on a fait et enfin il dispose du compiler.

Maintenant explicitant les différentes fonctions de cette classe, elle dispose d'une méthode getfreeregister qui commence par vérifier si ListDispo est null ou non si elle est n'est pas null alors elle return le premier element de cette liste sinon elle fait un push du registre numéro "compteur" et elle est return ce registre pour qu l'utilise. De plus elle dispose d'une méthode Refresh qui prend en argument un registre et décrémenter le nombre de fois qu'il s'est pushé de plus elle fait pop à ce registre pour récupérer la valeur précédente enregistrée dans ce registre. Par ailleurs il existe aussi une autre méthode qui est très utile c'est reinit qui réinitialiser les listes du manager comme au début de compilation et enfin d'une méthode get qui

prend en argument un entier puis vérifie si cet entier est disponible dans ListDispo sinon elle fait Push puis elle incrémente le nombre de fois qu'on fait push à ce register.

Avec cette classe on a pu manipuler de très grosses opérations binaires, mais il reste encore beaucoup de travail à faire dans cette partie car c'est la plus dure à gérer.

Génération du code :

Sémantique à l'étape C :

a) Génération du code d'une expression :

Pour une meilleure factorisation du code vous allez trouver dans le fichier AbstractExpr.java une fonction de génération de code codeGenExpr qui sert à générer le code pour une expression donnée. Et c'est cette même fonction qu'on surcharge dans les différentes classes filles qui étendent la classe mère AbstractExpr.java.
Cette fonction a comme attribut de retour une DVal qui étends la classe Operand. Et donc cette valeur peut-être soit un Immediate (String ou Int), un GPregistre, une DAddr ou encore une adresse de registre (o(R)) avec o un offset et R un registre appartenant à {R1, R2, R3 ….., R15}.
Dans la plupart de ces fonctions on a besoin de registres dans lequel on stockera la valeur de retour. Et pour ce faire, on récupère un registre libre fournit pas la classe RegistreManager.java. On se sert alors selon le besoin des instructions de la machine abstraite LOAD et STORE. Mais comme cette fonction est souvent utilisée de manière récursive, il se peut que la valeur de retour soit un registre, et dans ce cas on se sert directement de ce dernier pour y stocker de la valeur quand vous faire monter dans l'arbre.

b) Expressions arithmétiques binaires :

Pour gérer les expressions binaires comportant ainsi les instructions arithmétiques comme l'addition la soustraction la division et le modulo. On utilise la même fonction suce mentionné pour générer l'expression de Charles Aubert en droite et gauche.

Par souci de factorisation la fonction de génération se trouve dans le fichier espace, on n'y génère donc les expressions gauche et droite, hélas pour éviter tout écrasement dans le cas où les deux registres de retour sont identiques on teste ce cas et on l'évite. S'ensuit la génération du code de l'opération grâce là aussi à une méthode abstract codeGenOp dans la classe mère, et qui prend en paramètre le compilateur et les deux les expressions droite et gauche. Cette fonction génère donc l'opération trouvée dans le programme et est surchargée dans les classes :

PLUS
MULTIPLY
MODULO
UNARYMINUS
NOT
MINUS
CONVFLOAT

Par exemple, pour l'addition et la multiplication les codeGenOp sont les suivants :

```
      @Override
    void codeGenOp(DecacCompiler compiler, GPRegister left, DVal
right) {
      compiler.addInstruction(new ADD(right,left));
    }

      @Override
      void codeGenOp(DecacCompiler compiler, GPRegister left, DVal
right) {
      compiler.addInstruction(new MUL(right, left));
      }
```

La dernière étape dans la fonction qui génère des expressions arithmétiques binaires est de tester si oui ou non Overflow il y a. Et cela se fait que dans le cas ou l'option '-n' est absente de la commande de compilation. Le cas échéant donc, on génère une instruction 'BOV' avec une étiquette 'overflow !error' avant de retourner le registre où est stocké le résultat de l'opération.

Remarque :
Une possibilité d'optimisation qui est absente dans le code et qu'on peut rajouter par la suite et celle d'utiliser les instructions de shift pour calculer les expressions ou les opérandes sont des immédiats.

c) Expressions arithmétiques unaires :

Concerne les expressions arithmétiques unaires, on se sert là aussi de la fonction codeGenExpr, dans laquelle on commence par charger l'opérande dans un registre libre grâce à l'instruction LOAD avant de générer l'opération avec codeGenOp . Pour l'opération UnaryMinus on utilise l'instruction OPP, tandis qu'eux pour la conversion implicite vers un float on utilise FLOAT et pour finir on utilise l'opération SEQ qui prend le registre contenant l'opérande pour générer l'opération 'not'. Exemple de codeGenOp pour les expressions arithmétiques unaires :

```
@Override
void codeGenOp(DecacCompiler compiler, GPRegister left, DVal right) {
compiler.addInstruction(new OPP(right, left));
}
```

d) Codage des structures de contrôle :

Conditionnelles :

Avant de parler de ces deux structures il faut comprendre les deux fonctions codeGenInstPos et codeGenInstNeg se trouvant dans la classe AbstractExpr :

```
     void codeGenInstPos(DecacCompiler compiler, Label
etiquette){}
     void codeGenInstNeg(DecacCompiler compiler, Label
etiquette){}
```

Ces deux fonctions génèrent un branchement selon si l'expression est évaluée respectivement à TRUE ou FALSE.

Et pour générer donc du code pour le ifthenelse, on applique à l'expression condition (attribut de la classe IfThenElse.java) la fonction codeGenInstNeg() pour générer du code de branchement si celle-ci est évaluée à FALSE. Aussi, on applique la fonction de génération d'une liste d'instructions à l'attribut thenBranch. Par la suite on ajoute un branchement inconditionnel à l'étiquette de la fin selon si il y a un else ou pas. Et on finit par générer à la successivement une étiquette pour l''else', les instruction de la branche else et l'étiquette de fin.

Boucle While :

Pour la boucle tant que, c'est très simple, on ajoute les étiquettes nécessaires, la liste des instructions et on applique le codeGenInstPos() sur la condition.

e) Opérations de comparaisons :

Pour les opérations de comparaison on utilise des fonctions spécifiques qu'on va lister après. On commence par évaluer les membres de droite et de gauche et on exécute la comparaison après.

Les fonctions utiles pour cette partie :

```
 1° protected abstract void codeGenInstCmpPos(DecacCompiler
compiler, Label label);
 2° protected abstract void codeGenInstCmpNeg(DecacCompiler
compiler, Label label);

 3° protected void fetchCondPos(DecacCompiler compiler, GPRegister
r){};
 4° protected void fetchCondNeg(DecacCompiler compiler, GPRegister
r){};
```

Les première et deuxième fonctions permettent de brancher au bout de code précédé par le paramètre étiquette, selon si l'expression a été évaluée à vrai ou à faux. Elles utilisent selon les classes où elles sont surchargées les instructions :

BEQ
BNE
BGE
BLE
BLT

Les troisième et quatrième fonctions comment leur nom l'indiquent positionnent dans les registres passés en paramètre la valeur de la condition en suivant si on utilise pour chaque classe le fetchCondPos ou fetchCondNeg. On utilise pour cela les instructions :

SEQ
SNE
SLE
SGE
SGT
SLT

Exemple de fonctions surchargées dans la classe Great.java :

```
      @Override
      protected void codeGenInstCmpPos(DecacCompiler compiler,
Label label) {
            compiler.addInstruction(new BGT(label));
      }

      @Override
      protected void codeGenInstCmpNeg(DecacCompiler compiler,
Label label) {
            compiler.addInstruction(new BLE(label));
      }

      @Override
      protected void fetchCondPos(DecacCompiler compiler,
GPRegister r) {
            compiler.addInstruction(new SGT(r));
      }

      @Override
      protected void fetchCondNeg(DecacCompiler compiler,
GPRegister r) {
      compiler.addInstruction(new SLE(r));
      }
```

f) Opérateurs booléens :

Pour les opérations avec des AND et des OR, le code n'est pas facotrisé et tout se passe dans les classes respectives And.java et Or.java.
On évalue le membre de droite et qui sera exécuté seulement après avoir vu le membre de gauche.
Le membre de gauche est évalué en branchement (codeGenInstNeg() pour And et codeGenInstPos() pour Or) et si il est évalué à faux ou à vrai on regarde pas le membre de droite.
codeGenInstNeg() et codeGenInstPos() sont eux aussi surchargés pour éviter justement l'évaluation des deux membres (évaluation paresseuse) .

Les étiquettes sont eux aussi ajoutés pour permettre les bons branchement.

Remarque :
L'opérateur Or ne marche pas pour des expressions très simples (exemple : false II false). Et cela à cause d'une mauvaise gestion de registres.

g) Assignation :

Pour l'opération d'assignation, c'est assez simple. On charge les opérandes gauche (un register Offset) et droite (DVal) et on stocke ce dernier dans le premier.

h) Déclaration de variables :

Pour déclarer une variable, la fonction ajoutée par nos soins codeGenDeclVar est là pour ça !
On commence par récupérer l'initialisation (valeur retournée par codeGenExpr surchargée dans Initialization. Java et NoInitialization), puis selon si c'est une variable globale ou locale (boolean local = true), on stocke cette initialisation dans respectivement Register.GB et Register.LB avec un offset correspodant à la position de la variable dans le code.

i) ReadInt et ReadFloat :

Pour ces deux fonctions demandant à l'utilisateur de rentrer soit un int ou un float, on utilise les deux instructions RINT ET RFLOAT en tachant d'ajouter un branchement vers une erreur de type 'INPUT/OUTPUT Error' si le nombre entré ne correspondait pas (passer un int pour un float ou l'inverse).

j) Cast d'objets et instanceOf :

Pour le cast d'objet, on a besoin d'un algorithme qui permet de tester si oui ou non une classe A est instanceof une classe B. L'algorithme en question est généré par la fonction instanceOfProgram présente dans le fichier Program.java. Ce dernier repose sur le même principe d'une boucle tant que dont on sort que si effectivement la première classe ou l'une de ses superclasses est égale à la deuxième, ou si on est arrivé à la classe Object.

k) Génération des tables de méthodes:

Pour la génération des tables des méthodes on s'est servi de la fonction get All Methods qu'on a implémenté dans la classe ClassDefinition qui return une TreeSet des définitions des méthodes triée selon leur indice de déclaration de plus on a ajouté deux autres méthodes dans la classe getAdrTableMethodes et setAdrTableMethodes qui enregistre l'indice de la méthode dans la pile afin de les utilisés à chaque fois qu'on déclare un nouveau variable de type classe.