École nationale de l'informatique et des mathématiques appliquées de Grenoble

Projet Génie Logiciel

Année universitaire 2020-2021
Manuel Conception

Groupe 8 Equipe 40
Étudiants 2A

Table des matières

1 Architecture

1.1 Liste des classes

1.1.1 Pour la partie objet

En plus des classes déjà mises en place à la réception du projet, il a fallu implémenter presque entièrement les classes pour la partie objet de notre langage. Pour ce faire, différentes classes ont été créées.

Dans un premier temps il a été nécessaire de comprendre et d'implémenter des classes correspondant aux terminaux et au non terminaux principaux.

Dans un second temps, pour reprendre la hiérarchie de classe et d'arbre déjà présente dans le projet, nous avons dû créer les classes abstraites qui correspondaient à nos classes précédentes.

Enfin, il a été nécessaire de bien comprendre et d'adapter les extensions de chacune des classes pour faciliter l'implémentation de la partie B et C pour les objets du langage.

Nous avons donc la liste des classes suivantes :

- AbstractDeclField extends Tree
- AbstractDeclMethod extends Tree
- AbstractDeclParam extends Tree
- AbstractMethodBody extends Tree
- DeclField extends AbstractDeclField
- DeclMethod extends AbstractDeclMethod
- DeclParam extends AbstractDeclParam
- ListDeclField extends TreeList<AbstractDeclField>
- ListDeclMethod extends TreeList<AbstractDeclMethod>
- ListParam extends TreeList<AbstractDeclParam>
- MethodAsmBody extends AbstractMethodBody
- MethodBody extends AbstractMethodBody
- Return extends AbstractInst

1.1.2 Pour l'extension TAB

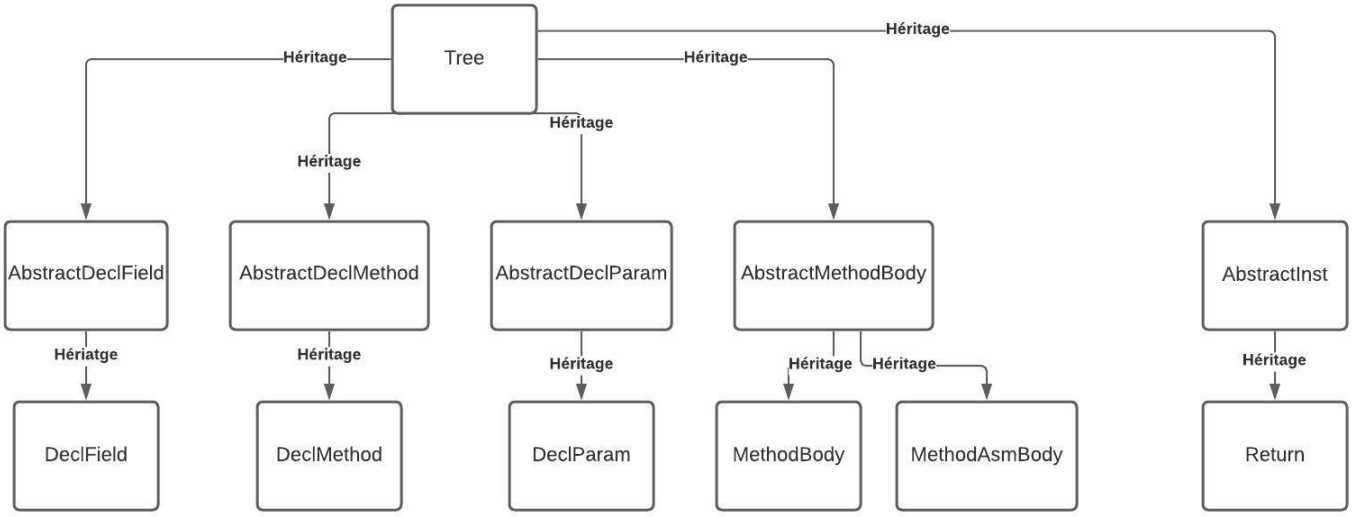
Liste des classes implémentées pour l'extension TAB :

- ArrayLiteral extends AbstractExpr
- ArraySelection extends AbstractLValue
- IdentifierArray extends AbstractIdentifier
- NewArray etxends AbstractExpr

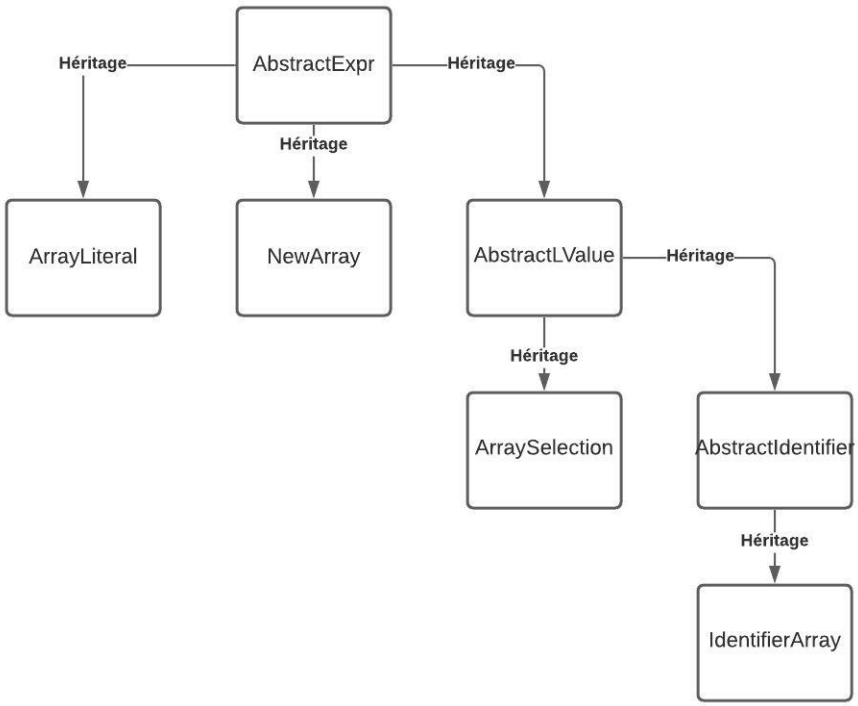
Pour plus de précisions, la documentation de l'extension est disponible.

1.2 Dépendances

1.2.1 Pour la partie objet



1.2.2 Pour l'extension TAB



1.3 Modifications des classes préexistantes

1.3.1 Changement dû au PUSH POP de registre dans une méthode

Au cours de la création de la génération de code pour la partie objet, il a été nécessaire de mettre en place le PUSH et POP des registres à utiliser lors de l'appel d'une méthode. Le

problème de ces PUSH POP était qu'il fallait prévoir le nombre de registre qu'il sera nécessaire de réserver en début de méthode.

Pour ce faire, nous avons décidé de nous inspirer des fonctions addFirst déjà préexistantes pour le programme. Seul problème, lors de l'utilisation de addFirst pour les commandes PUSH de registres en début de méthode, cela considérait le programme tout entier et mettait donc les commandes concernées en début de programme. Il a donc fallu adapter notre réflexion à ce problème.

Il a été décidé de mettre en place un programme auxiliaire, rattaché à un booléen Aux. Ainsi, lorsqu'on veut savoir le nombre de registre à PUSH pour la méthode on setAux à True, puis on utilise programmeAux pour passer une première fois dans la génération de code du corps de la méthode. Une fois cet appel fait, on remet la pile et le tas dans l'état avant la réalisation du programme auxiliaire.

De plus, nous avons réalisé un tableau de booléen appelé RegisterUsed, initialisé à faux. Lors de la première exécution du corps de la méthode, lorsqu'un registre i est utilisé, on passe RegisterUsed[i] à true. Ainsi, la fonction codGenBody dans la classe MethodBody retournera le nombre de true dans le tableau RegisterUsed (non utilisé pour le cas d'une méthode en assembleur, classe MethodAsmBody). Dans le cas où aucun registre n'est utilisé, on PUSH quand même un registre si il y a des déclarations de variables.

2 Spécifications sur le code

2.1 Rappels pour les méthodes *codeGenInst, codeGenDecl, codeGenPrint, codeGenBranch*

Ces méthodes sont centrales dans la conception de notre code. Elles sont très adaptées au côté objet de notre architecture : le code généré dépend de la nature de l'objet considéré. Ainsi, *codeGenInst* est redéfini pour chaque instruction (Assignation, Opérations..), *codeGenDecl* pour tout type de déclaration (variable, classe, méthode ..), *codeGenPrint* permet de géré l'affichage selon le type donné en argument au print, cela concerne toute expression pouvant être de type entier, float ou string, c'est donc regroupé dans AbstractExpr (sauf pour ArraySelection, cf 3.4 Extension). Enfin, la méthode *CodeGenBranch* concerne tous les blocs conditionnels et est donc centrale pour les boucles while et les blocs IfThenElse. Toute personne portant de l'intérêt à notre projet doit donc s'intéresser en priorité à ces méthodes là, et avant tout à une méthode qui leur est souvent commune : *codeExp*.

2.2 La méthode codeExp

Rapidement dans le projet il nous a fallu mettre en place cette méthode qui allait s'avérer être une des plus importantes de notre conception. Elle s'est imposée à nous quand nous avons commencé à nous intéresser au problème de la gestion des registres. L'idée est simple, nous voulions une méthode qui lorsque qu'elle est appelée, évalue l'objet dans un registre donné. Nous avons donc créé pour toute expression (première déclaration dans AbstractExpr) la méthode *void codeExp(DecacCompiler compiler, int n)*. L'argument *compiler* est bien sur indispensable à toute méthode générant du code, et l'entier *n* désigne le numéro du registre dans lequel doit être placé le résultat. Ainsi, nous avons ensuite écrit (Override) pour tout les cas d'expression lorsque c'était nécessaire. Le grand avantage de cette méthode est qu'elle s'incorpore très bien à l'architecture orientée objet de ce projet, et de plus elle facilite grandement la gestion des registres. En effet, l'utilisateur a à sa disposition l'option **-r X** du compilateur, lui permettant de réduire le nombre de registres utilisés par le code assembleur (initialement 16), cette donnée est stockée dans le champ RegisterMax de la classe DecacCompiler (cf section suivante). Ainsi, lorsque l'évaluation d'une expression dans le registre n nécessite plus d'un registre, on appelle usuellement codeExp sur les registres n+1, n+2 ..etc mais il faut veiller à ne jamais dépasser RegisterMax. Dans ce cas particulier, il faut utiliser les instructions PUSH POP pour préserver l'état d'autres registres avant de les utiliser. Toute modification du code (pour maintenance ou amélioration) doit veiller à cela.

2.3 Les champs entiers du compilateur : RegisterMax, kSP, kGB et kLB

Tout au long du développement, nous avons du rajouter des constantes dans la classe DecacCompiler. Citons les principales et leur but.
Nous avons besoin souvent de savoir le registre maximal utilisable (cf paragraphe CodeExp ci-dessus), cette donnée est stockée dans le champ RegisterMax, initialisée par défaut à 15 (16 registres, registre max = R15).
Pour les déclarations de variables (globales ou locales), il faut aussi un compteur pour établir un

offset (1(GB), 2(GB)..), pour cette donnée on se sert des entiers kGB et kLB. Enfin, pour tester et prévenir un débordement de pile, on utilise l'instruction assembleur *TSTO*. Néanmoins, pour factoriser ces instructions répétitives (cf Document gestion énergétique) il nous faut là encore un compteur enregistrant la position maximal du pointeur SP au cours du programme. Cette donnée est calculée à l'aide des grandeurs kSP et maxSP.

2.4 Création des types prédéfinis

La création des types prédéfinis du langage se fait dans la classe DecacCompiler grâce à un attribut EnvironmentType. A la création d'une classe DecacCompiler, les types prédéfinis seront rajoutés à l'environnement à travers la méthode *createPredefTypes*. Dans un même temps, la table des symboles est également créée à travers un attribut *symbols* permettant de garantir l'unicité des noms d'identifiant. Le code permettant de générer la classe de base Object est fourni par une DeclClass sauvegardée dans DecacCompiler et initialisée à la compilation avant la vérification contextuelle.

La classe EnvironmentType ainsi que EnvironmentExp ont été implémentées avec un HashMap permettant de lier le symbole de nom à une Definition. L'empilement est géré par une recherche récursive du nom dans l'environnement puis celui du parent si la recherche est infructueuse. L'union disjointe est, elle, gérée par la levée d'une exception si le symbole de nom apparaît déjà dans le Map.

La création des messages d'erreur s'est faite via des attributs *static publique* de la classe *ContextualError* pour chaque erreur possible.

2.5 Les 3 possibilités d'appel d'un objet

Lors d'une assignation avec les objets ou lors d'un appel de méthode, plusieurs possibilités sont à spécifier. En effet 3 cas sont possibles pour désigner l'objet (exemple avec une classe A) :

- Faire une déclaration A a = new A() puis l'utiliser à l'aide de a.*un field* pour une assignation ou a.*une méthode* pour un appel de méthode.
- Utiliser directement le new : (new A()).*un field* pour une assignation ou (new A()).*une méthode* pour un appel de méthode.
- Utiliser un this (seulement possible dans une classe) : this.*un field* pour une assignation ou this.*une méthode* pour un appel de méthode. Ces trois possibilités sont toutes les trois acceptées par la grammaire mais il a fallu les différencier lors de l'écriture de la génération de code pour une assignation et pour un appel de méthode.

2.6 Override d'une méthode

Lors de la mise en place de la partie objet de notre langage, il a été nécessaire de gérer la réécriture d'une méthode par une classe fille. Par exemple, si une classe A, qui hérite d'Objet par définition, a elle même la déclaration d'une méthode equals avec le même type de retour, le même nom et les mêmes paramètres cela correspond à une réécriture. Dans cet exemple, si on

fait un appel de la méthode equals pour une objet A, on veut que cette méthode corresponde à celle de la fille et non pas celle de la mère.

Pour cela nous avons dû dans un premier temps mettre en place un moyen de comparer les méthodes pour savoir reconnaître un cas d'override. Nous avons donc mis en place un HashMap avec comme clés les noms des méthodes (String) et comme valeur associée l'AbstractMéthod correspondante. Avec ce HashMap nous avions une première comparaison dans les méthodes déclarées ce qui permettait de voir les premières occurrences. Pour confirmer ces occurrences, nous avons ensuite comparé les signatures pour être sûr de la réécriture de la méthode. Si le nom et la signature étaient les mêmes dans ce cas, nous pouvons affirmer que nous sommes face à un override.

Une fois l'override confirmé, nous avons mis en place une dissociation de cas entre la déclaration d'une méthode et l'override d'une méthode. Tout d'abord nous avons, dans la méthode codeGenListDeclMethod, mis l'index de la méthode mère pour la méthode fille, ainsi au yeux de la classe fille, la méthode mère n'avait plus d'indice, donc impossible de l'utiliser. De plus nous avons dissocié les méthodes de déclaration de méthode avec en particulier les méthodes codeGenDeclMethod et codeGenDeclMethodOverride dans la classe DeclMethod.

3 Description des algorithmes et structure de données employés

3.1 La génération branchements et opérations booléennes

La génération des branchements, comme décrite dans la partie 7 du sujet [Codage des expressions], a été conçu à travers une méthode de signature *void codeGenBranch(DecacCompiler compiler, boolean evaluate, Label label)*

Pour chaque objet, permettant ainsi de suivre l'arbre et de demander la génération de code pour chaque opération booléenne ou de comparaison. Ainsi, <Code(C, vrai,E)> sera généré par la méthode codeGenBranch(compiler, true, E) appliquée au noeud décrivant la condition C. Ces méthodes permettent d'implémenter les conditionnelles et les boucles exactement comme décrit dans la partie 8 [Codage des Structures de contrôle].

3.2 Indices de méthodes et de champs

Lors de l'implémentation de la partie objet du langage, nous avons du gérer l'appel de méthode ainsi que la sélection de champ, tout en prenant en compte l'héritage et la notion de réécriture des méthodes (Override). Pour accéder à la bonne méthode ou au bon champ dans ces cas de figure, nous avons dû construire un indice de méthode et un indice de champ pour chaque méthode et champ d'une famille de classe. Ces indices sont définis dans les classes MethodDefinition.java et FieldDefinition.java du package context. Prenons un exemple pour y voir plus claire :

```
1  Class A {
2
3      int x = 1;
4
5      int getAttribut() {
6          return x;
7      }
8      void setAttributA(int x) {
9          this.x = x;
10     }
11 }
12
13 Class B extends A {
14
15     int y = 2;
16
17     int getAttribut() {
18         return y;
19     }
20     void setAttributB(int y) {
21         this.y = y;
22     }
23 }
```

Ils ne faut oublier que toute classe hérite de la classe Object. Ainsi, pour la table des méthodes de A, nous aurons dans l'ordre :

- Equals (indice 0)
- getAttribut (indice 1)
- setAttributA (indice 2)

Tandis que pour la table des méthodes de B nous aurons :

- Equals (indice 0)
- getAttribut (indice 1)
- setAttributA (indice 2)
- setAttributB (indice 3)

Ainsi, l'indice de méthode permet de connaitre l'offset à effectuer pour récupérer cette dernière. Le principe est le même avec les champs. Pour plus de détails, le lecteur pourra aller voir dans le fichier ListDeclMethod.java de fr.ensimag.deca.tree où il y a dans le cas d'un Override réécriture de l'indice de méthode.

3.3 HashMap pour Override une méthode

Comme expliqué précédemment, il a été nécessaire de mettre en place un Hashmap afin d'écrire notre override de méthode. En effet, avec un hashmap avec comme clé le *String* nom de la méthode et en valeur la méthode en elle même, il était possible de facilement retrouver si une méthode était réécrite dans une classe fille.

3.4 Extension

Les éléments développés dans cette partie feront écho au document traitant de notre extension, n'hésitez pas à le consulter en complément.

3.4.1 Algorithme pour la sélection d'élément d'un tableau

Voici notre algorithme de sélection d'un élément dans un tableau (écrit en pseudocode) :

```
{
    // Notons : float[]...[] tab = new float[t1][t2]...[t(n-1)][tn], voici notre
    algorithme pour l'instruction de selection "tab[k0][k1]...[k(n-1)][kn]"
    int somme = 0;
    int facteur = 1;
    for (i allant de n a 0) {
        -> verifier si ki est out of range;
        somme = somme + facteur * ki;
        facteur = facteur * (taille de la dimension i);
    }
    return somme;
}
```

C'est donc cet algorithme qui est utilisé dans la méthode codeExp de la classe ArraySelection.java. Il est important que cette méthode codeExp soit différente dans ce cas-ci. En effet,

nous avons du rajouter un argument (par Overload) qui est un booléen pour renseigner si il faut évaluer la VALEUR de l'élément dans le registre n ou bien son ADRESSE. Ce besoin découle du fait que pour une instruction d'assignation, il faut l'adresse de l'élément, pour y rentrer sa nouvelle valeur (tab[i][j] = 1), tandis que dans d'autres cas, ils nous faut sa valeur : (x = x + tab[i][j]). Par défaut, codeExp fera le deuxième cas, mais il faut savoir cette particularité de conception, qui peut s'avérer pratique.

3.4.2 Algorithme pour les ArrayLiteral

Contrairement à l'allocation des tableaux par l'instruction *new*, rentrer un tableau manuellement (ArrayLiteral, avec la syntaxe décrite dans la documentation utilisateur) nécessite d'allouer ET d'initialiser. L'allocation est identique à celle de la l'instruction *new*. Mais nous avons créé un algorithme récursif pour l'initialisation des valeurs (cf initValueRec et initValue dans la classe ArrayLiteral.java). Pour expliquer dans les grandes lignes le principe, voici l'idée. Le problème d'une sélection standard, dont nous avons détaillé l'algorithme dans le paragraphe précédent, est qu'à l'instant de la génération de code nous ne connaissons pas forcément les coordonnées de la sélection. En effet, le code suivant est valide :

```
{
    int n = 1;
    int[][] tab = new int[3][3];
    tab[2*n][n+n/1] = 2;
}
```

Ainsi, il faut tout traduire et tout faire en langage assembleur (cf ArraySelection.java). Il était donc hors de question en terme de performance de procéder de même pour une initialisation de la forme, c'est à dire de considérer ce qui suit :

```
{
    int[] tab = {1, 2, 3};
    // equivaut
    int[] tab = new int[3]
    int i = 0;
    while (i < 3) {
        tab[i] = i+1;
        i++;
    }
}
```

Il est clair que dans le premier cas, il était possible de faire mieux, et d'économiser des instructions d'assembleurs. C'est pourquoi nous avons écrit ces algorithmes mentionnés plus haut. Décrivons en bref leur fonctionnement. Depuis la classe ArrayLiteral nous avons accès à une liste d'expressions qui est en fait la liste d'éléments du tableau. On applique alors à cette liste le procédé récursif suivant :

- Si le premier élément est un tableau, alors pour chaque sous tableau, on transmet son indice dans la liste et on initialise ses valeurs (récursion).

- Sinon, il s'agit d'un objet, on l'initialise donc à partir de la liste de coordonnées/indices reçues.

Voici un exemple pour mieux comprendre :
On souhaite initialiser en mémoire le tableau {{10, 20}, {30, 40}} :
On alloue donc un bloc de taille 5 (=4+1) sur le tas pour les éléments et un deuxième bloc de taille 2 pour la taille des dimensions (2 et 2). (cf document extension pour plus de détails).
L'adresse retournée par l'instruction NEW est placée dans le registre R_n.
On commence la récursion :
=> initialisation de {{10, 20}, {30, 40}} :
Le premier élément est un tableau. On est dans le premier cas.

- appel 1 : indice 0, initialisation de {10, 20}

- appel 2 : indice 1, initialisation de {30, 40}

=> initialisation de {10, 20} :
Le premier élément est un entier. On est dans le deuxième cas.

- on calcule l'offset pour les coordonnées (0, 0) grâce à l'algorithme de sélection, on trouve 0. On rentre donc 10 dans la case mémoire 1(R_n)

- on calcule l'offset pour les coordonnées (0, 1) grâce à l'algorithme de sélection, on trouve 1. On rentre donc 20 dans la case mémoire 2(R_n)

=>initialisation de {30, 40} :
Le premier élément est un entier. On est dans le deuxième cas.

- on calcule l'offset pour les coordonnées (1, 0) grâce à l'algorithme de sélection, on trouve 2. On rentre donc 30 dans la case mémoire 3(R_n)

- on calcule l'offset pour les coordonnées (1, 1) grâce à l'algorithme de sélection, on trouve 3. On rentre donc 40 dans la case mémoire 4(R_n).