UGA – Grenoble INP

Projet Génie Logiciel

Documentation de conception

Grenoble, 26 janvier 2022.

Table des matières

1 Architecture du projet

1.1 Dépendance des classes

L'implémentation de ce projet repose en grande partie sur le principe d'hérédité de Java ; ainsi la mise en place de l'arbre des classes du package *Tree* fut primordial pour pouvoir avoir un meilleur point de vue sur la structure du projet. Des classes sont déjà présentes dans le squelette du projet (en bleu) et d'autres furent rajoutées au fur et à mesure (en rouge).

Dans cette représentation, les classes abstraites sont rectangulaires tandis que les feuilles sont en ovale. Pour ne pas surcharger l'arbre, certaines feuilles ont été déplacées en dessous.

Ainsi les différentes classes créées le furent pour pouvoir implémenter la notion de *class* dans le langage Deca :

— les classes *ListDeclParam, ListDeclMethod, ListDeclField* reprennent l'exemple déjà existant de *ListExpr* et permettent ainsi de s'occuper de classes ayant plusieurs déclarations de méthodes, d'attributs ou de paramètres.

— les classes *AbstractDeclParam* et *DeclParam* permettent de gérer l'implémentation des paramètres des différentes méthodes. Ainsi, *DeclParam* possède deux attributs de type AbstractIdentifier qui sont le nom et le type du paramètre concerné.

— les classes *AbstractDeclField* et *DeclField* gèrent l'implémentation des attributs d'une classe. Pour cela *DeclField* utilise quatre attributs reprenant le nom et le type de l'attribut, sa visibilité (*protected* ou *public*) ainsi que son initialisation.

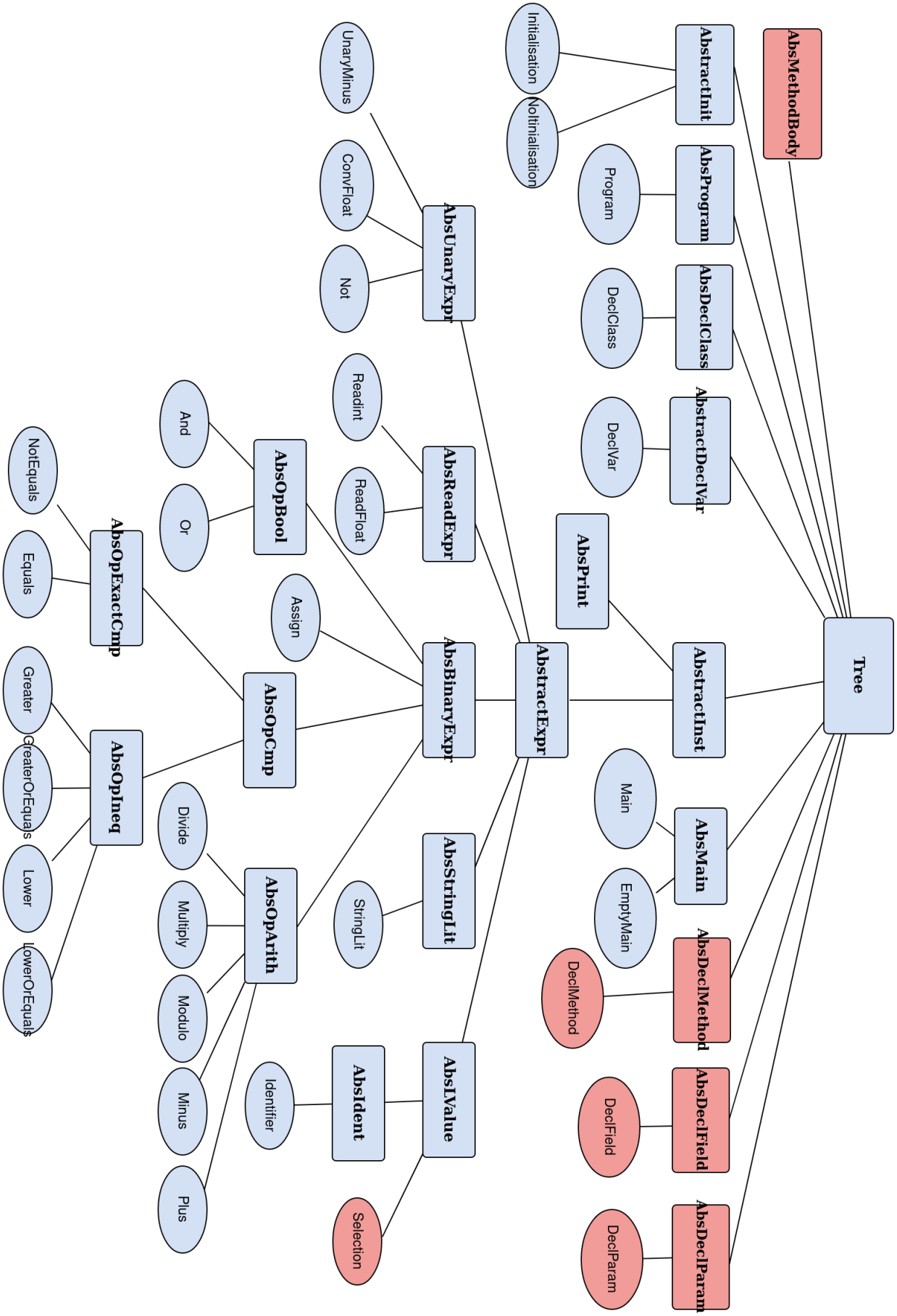
— les classes *AbstractDeclMethod* et *DeclMethod* permettent de gérer la déclaration de méthodes. Pour cela *DeclMethod* utilise comme attribut une liste de paramètres, son type (void, int, float, ...), son nom ainsi que ses différentes instructions (instructions gérées par la classe *AbstractMethodBody*).

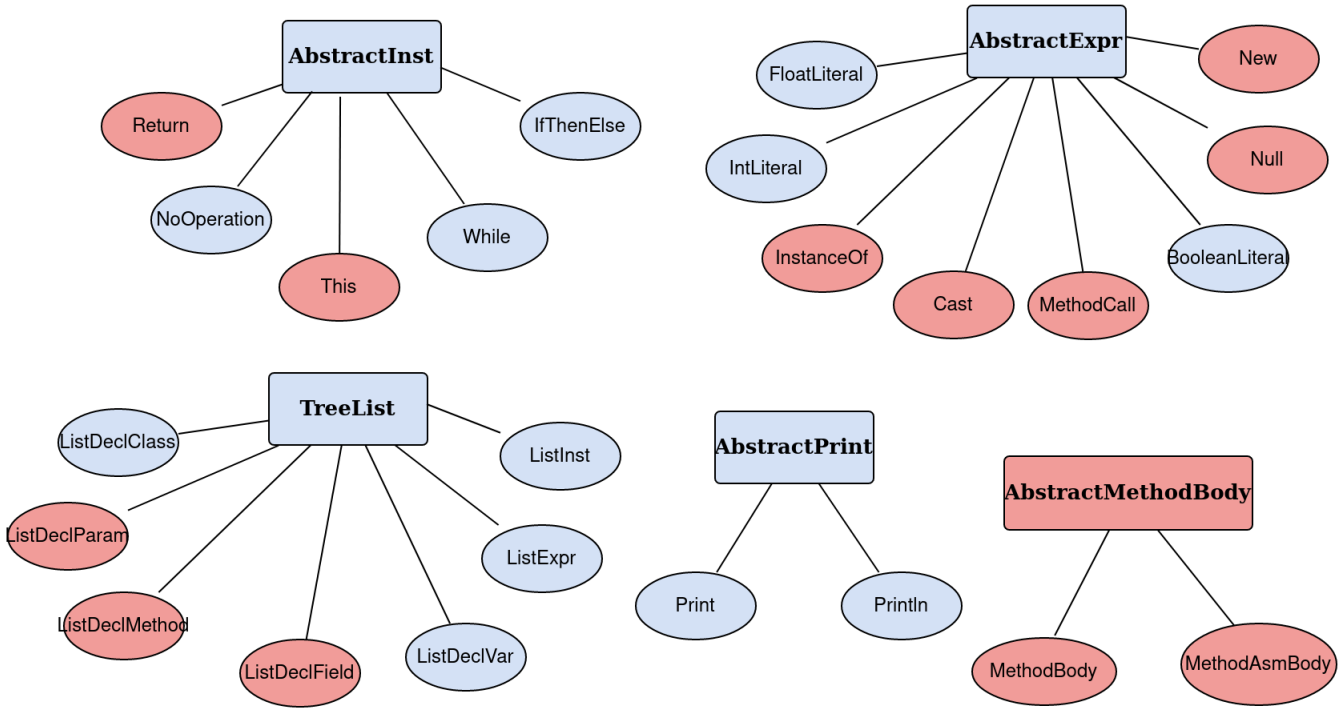
— les classes *AbstractMethodBody, MethodBody* et *MethodAsmBody* permettent de représenter le code des différentes méthodes. *MethodBody* et *MethodAsmBody* héritent toutes les deux de *AbstractMethodBody* qui elle même hérite de *Tree*. *MethodAsmBody* représente des méthodes bien particulières ayant directement des instructions en assembleur. Ainsi il suffit donc pour ce genre de méthode de recopier le code assembleur directement dans le fichier *.ass*.

— les classes *This, New, Null, Return* et *InstanceOf* ont été créées à la suite de l'utilisation de nouveaux mots clés liés au concept de classe.

— l'appel aux méthodes et l'accès des paramètres a demandé une construction particulière. En effet, la classe *MethodCall* représente juste l'appel à une méthode et prend donc comme attribut la liste de paramètres donnés et le nom de la méthode. D'un autre coté la classe *Selection* s'occupe de sélectionner dans un objet une caractéristique de celui-ci, c'est à dire soit un attribut soit une méthode.

```
class A {
    int x;
    int method() { ... }
}
A objet = new A();
print(objet.x);             ici sélection d'un attribut
println(objet.method())     ici sélection suivie d'un appel à une méthode
```


Tree
AbsMethodBody
AbstractInit
Initialisation
NoInitialisation
AbsProgram
Program
AbsDeclClass
DeclClass
AbstractDeclVar
DeclVar
AbsPrint
AbstractInst
AbstractExpr
AbsMain
Main
EmptyMain
AbsDeclMethod
DeclMethod
AbsDeclField
DeclField
AbsDeclParam
DeclParam
AbsUnaryExpr
UnaryMinus
ConvFloat
Not
AbsReadExpr
ReadInt
ReadFloat
AbsBinaryExpr
AbsStringLit
StringLit
AbsLValue
AbsIdent
Identifier
Selection
AbsOpBool
And
Or
AbsOpCmp
Assign
AbsOpExactCmp
NotEquals
Equals
AbsOpIneq
Greater
GreaterOrEquals
Lower
LowerOrEquals
AbsOpArith
Divide
Multiply
Modulo
Minus
Plus



2 Implémentation de la Vérification Contextuelle

2.1 Gestions des environnements

Nous avons pris le parti de fusionner au maximum le code en ne créant qu'un seul type d'environnement générique que nous pouvons appliquer à plusieurs types d'environnements spécifiques. Cela donne une plus grande flexibilité dans la construction du code, mais cela force à générer à chaque compilation la construction des Types et des expressions génériques. Ainsi, au lancement du code, sont initialisés :

- Le type **void** pour les fonctions ne renvoyant pas de résultat
- Le type **boolean** pour la gestion des booléens
- Le type **float** pour la gestion des flottants
- Le type **int** pour la gestion des entiers naturels
- Le type **string** pour la gestion des chaînes de caractères
- Le type **null** pour les objets sans types appropriés
- La classe **object** qui est considérée comme super-classe immédiate de chaque classe construite dans l'analyse contextuelle
- La méthode **equals** qui permet la comparaison de deux objets, méthodes inhérentes à **object** et donc à toutes les classes générées.

Les environnements ainsi pré-générés servent de base aux parcours des arbres. Ces environnements permettent plusieurs choses :

- récupérer un type précis pour l'attribuer à une expression
- récupérer une définition de classe (qui reprend la même méthode mais en castant le type)
- déclarer un nouveau symbole avec la définition associée
- déclarer un nouvelle classe en associant sa définition associée

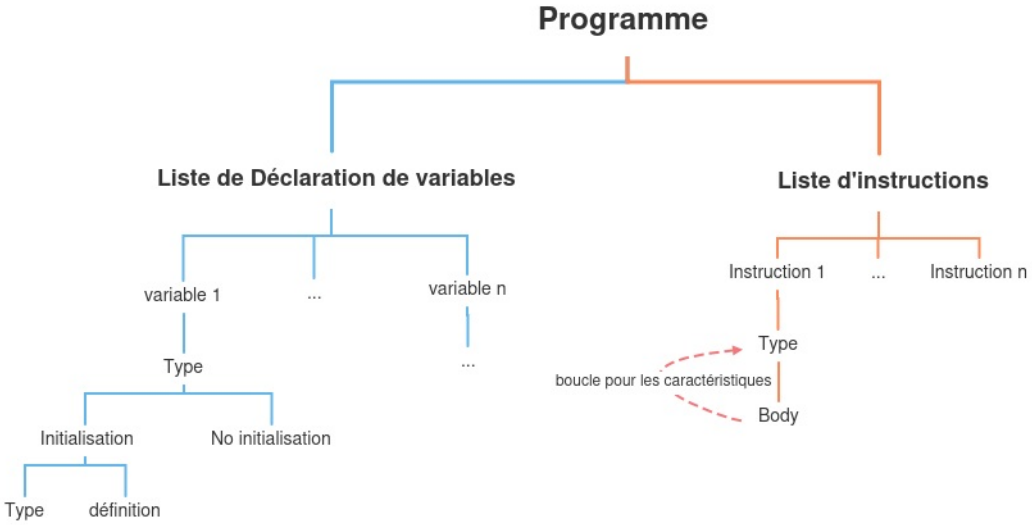
Nous avons décidé de séparer le travail sur les classes et les objets non classes afin de ne pas se retrouver avec des problèmes de typages, notamment plus tard lors des vérifications de types. En effet, séparer les deux entités permet de directement travailler avec des *Definition* ou des *ClassDefinition* indifféremment, sans avoir à se soucier d'un cast potentiel.

Nous avons donc construit des environnements chaînes qui permettent de facilement passer de l'un à l'autre, chacun étant pré-généré à l'initialisation de l'objet, et pouvant travailler sans distinction avec un objet simple ou une classe. Cet environnement de base va se faire compléter lors du parcours de l'arbre, afin de construire l'arbre contextuel de chaque classe, mais aussi de la fonction *main*, afin d'avoir un arbre de définition et de typage complet.

2.2 Parcours de vérification

2.2.1 Partie sans objet

La vérification contextuelle de la partie sans objet est plutôt simple. Il s'agit d'un simple parcours en profondeur de l'arbre des dépendances syntaxiques créé par le parser. Pour mieux comprendre ce parcours, nous pouvons regarder comment se décompose un programme sans objet, dans le langage Deca. Le schéma ci-dessous représente comment fonctionne la vérification contextuelle d'un programme sans objet.

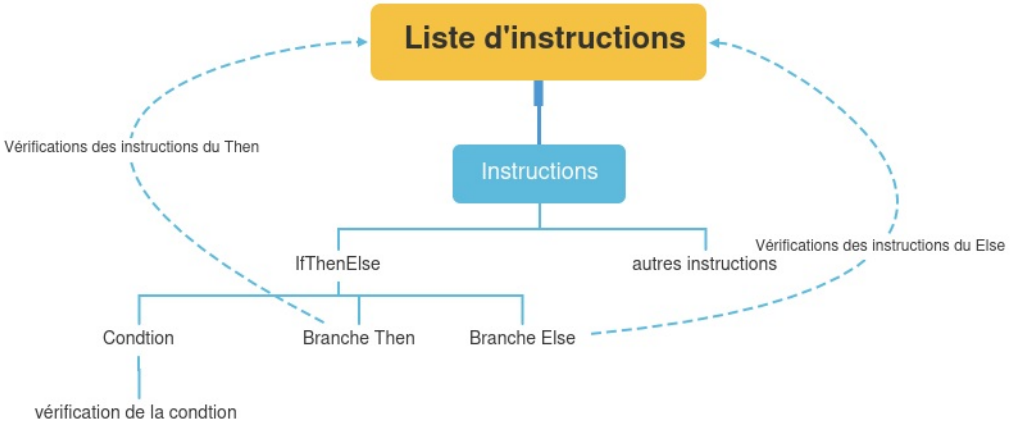


On peut donc constater que sont d'abord traitées les déclarations puis les instructions. Nous allons voir plus en détail le fonctionnement de quelques instructions, mais en général, chaque partie d'une instruction est vérifiée. Puis on descend progressivement l'arbre en vérifiant les expressions qui composent le corps de l'instruction.

— cas d'une **expression** (*Assign, opération arithmétique*). Ce cas est simple puisqu'il suffit de récursivement vérifier toutes les parties de l'expression et de relever s'il y a un problème de typage ou de définition quelque part.

— cas d'un **print**. Il faudra ici vérifier le type de chaque argument, et afficher une erreur s'ils font partie des types interdits

— Cas d'un **ifThenElse**. Le schéma ci-dessous explique plus en détails le fonctionnement de la vérification :



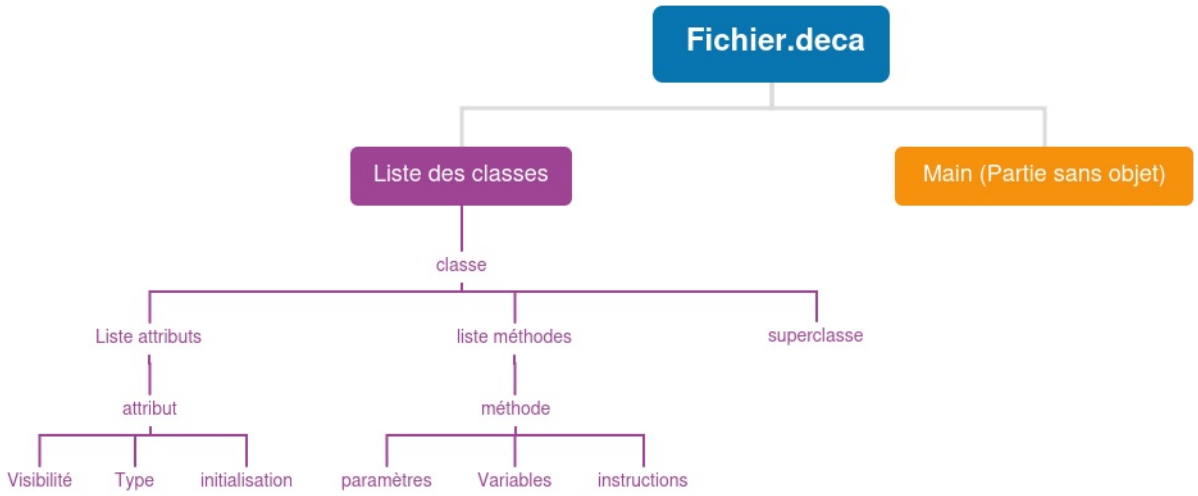
On remarque alors que nous avons une boucle de vérification du contenu pour les instructions du *then* et du *else*. Cette boucle permet la génération des structures de type :

```
if (){ ...
}else if(){
    ...
}else if(){
    ...
}else{
    ...
}
```

— cas d'un **while**. Le principe est le même que pour le **IfThenElse** : on vérifie la condition et on boucle la vérification sur le corps des instructions du **while**.

2.2.2 Vérifications des classes

Nous pouvons maintenant nous intéresser à la composition d'un programme classique Deca utilisant des classes. Le schéma ci-dessous représente la composition d'un fichier Deca :



La construction d'une classe demande beaucoup de vérifications supplémentaires par rapport à la partie sans objet, et elle se décompose en trois phases qui représentent les trois étapes de la construction d'une classe.

Phase 1 Elle représente l'initialisation de la classe et la récupération de la super-classe potentielle. Une classe sans super-classe se voit attribuer par défaut la classe **object** comme parent. On récupère ainsi

l'environnement parent et on créé un nouvel objet de type *class*. On peut alors fixer le type et la définition de l'environnement courant, tout en vérifiant que ce type peut bien être déclaré, pour éviter les conflits de nomenclature. En résumé, cette phase permet de définir l'environnement général de la nouvelle classe fraîchement créée. On se situe donc au niveau de l'étage supérieur du schéma.

Phase 2 La phase 2 correspond à l'initialisation des attributs et des méthodes. Cela correspond à la phase intermédiaire du schéma. Elle permet de récupérer les informations de la super-classe concernant les attributs et les méthodes, vérifiant si la redéfinition de méthode est conforme à la signature et aux paramètres de la méthode mère. Cette phase permet de définir la structure de la classe et de constater la cohérence de l'environnement local avec l'environnement parent.

Phase 3 La phase 3 est la plus proche de la vérification du sans objet. En effet, puisque le squelette de la classe a été vérifié par les phases précédentes, il suffit juste de vérifier un à un :

— La déclaration des attributs, en suivant la même méthodologie que la déclaration de variable du sans objet.

— La déclaration des méthodes, en suivant la même méthodologie qu'un programme **main** classique. Néanmoins, il faut faire particulièrement attention aux valeurs de retour de l'expression renvoyée par *return* car elle doit coïncider avec la signature de la méthode.

2.3 Un exemple de vérification de l'arbre

L'exemple suivant permet de mettre du concret sur la vérification d'un programme Deca. Voici un programme Deca simple utilisant plusieurs fonctionnalités du compilateur.

```
class A {
   int x ;
   void setX(int a) {
        println("j'utilise le setX de A");
        this.x = a ;
   }
   int getX() {
      return this.x;
   }
}

class B extends A {

    void setX(int a) {
        println("j'utilise le setX de B");
        this.x = a;
   }
}

{
   B a= new B() ;
   a.setX(2) ;
   println("a.getX() = ", a.getX()) ;
}
```

L'arbre décoré obtenu est le suivant : nous allons commenter les différents ajouts de décoration.

```
 > [1, 0] Program
+> ListDeclClass [List with 2 elements]
|  []> [1, 6] DeclClass
```

```
|  ||  +> [1, 8] Identifier (Object)   ---> Attribution automatique de Object en superclass
|  ||  |  definition: type (builtin), type=Object
|  ||  +> [1, 6] Identifier (A)
|  ||  |  definition: type defined at [1, 6], type=A
|  ||  +> ListDeclField [List with 1 elements]
|  ||  |  []> [2, 7] PUBLIC DeclField
|  ||  |      +> [2, 3] Identifier (int)
|  ||  |      |  definition: type (builtin), type=int  ---> Récupération du type
                                                            préconstruit
|  ||  |      +> [2, 7] Identifier (x)
|  ||  |      |  definition: field defined at [2, 7], type=int  ---> Attribut initialisé
|  ||  |      `> NoInitialization
|  ||  `> ListDeclMethod [List with 2 elements]
|  ||     []> [3, 3] DeclMethod
|  ||     ||  +> [3, 3] Identifier (void)
|  ||     ||  |  definition: type (builtin), type=void
|  ||     ||  +> [3, 8] Identifier (setX)
|  ||     ||  |  definition: method defined at [3, 3], type=void
|  ||     ||  +> ListDeclParam [List with 1 elements]
|  ||     ||  |  []> [3, 13] DeclParam
|  ||     ||  |      +> [3, 13] Identifier (int)
|  ||     ||  |      |  definition: type (builtin), type=int
|  ||     ||  |      `> [3, 17] Identifier (a)
|  ||     ||  |         definition: parameter defined at [3, 13], type=int
|  ||     ||  `> [3, 20] MethodBody
|  ||     ||     +> ListDeclVar [List with 0 elements]
|  ||     ||     `> ListInst [List with 2 elements]
|  ||     ||        []> [4, 8] Println
|  ||     ||        ||  `> [4, 16] ListExpr [List with 1 elements]
|  ||     ||        ||     []> [4, 16] StringLiteral (j'utilise le setX de A)
|  ||     ||        ||         type: string
|  ||     ||        []> [5, 8] Assign
|  ||     ||            type: int
|  ||     ||            +> [5, 8] Selection
|  ||     ||            |  type: int
|  ||     ||            |  +> [5, 8] This
|  ||     ||            |  |  type: A
|  ||     ||            |  `> [5, 13] Identifier (x)  ---> récupération du bon identifiant
|  ||     ||            |     definition: field defined at [2, 7], type=int
|  ||     ||            `> [5, 17] Identifier (a)
|  ||     ||               definition: parameter defined at [3, 13], type=int
|  ||     []> [7, 3] DeclMethod
|  ||         +> [7, 3] Identifier (int)
|  ||         |  definition: type (builtin), type=int
|  ||         +> [7, 7] Identifier (getX)
|  ||         |  definition: method defined at [7, 3], type=int
|  ||         +> ListDeclParam [List with 0 elements]
|  ||         `> [7, 14] MethodBody
|  ||            +> ListDeclVar [List with 0 elements]
|  ||            `> ListInst [List with 1 elements]
|  ||               []> [8, 6] Return
|  ||                   `> [8, 13] Selection ---> Controle de la valeur de retour
|  ||                      type: int
```

```
|  ||                      +> [8, 13] This
|  ||                      |  type: A
|  ||                      `> [8, 18] Identifier (x)
|  ||                         definition: field defined at [2, 7], type=int
|  []> [12, 6] DeclClass
|      +> [12, 8] Identifier (A)
|      |  definition: type defined at [1, 6], type=A ---> récupération de la super classe
|      +> [12, 6] Identifier (B)
|      |  definition: type defined at [12, 6], type=B
|      +> ListDeclField [List with 0 elements]
|      `> ListDeclMethod [List with 1 elements]
|         []> [14, 4] DeclMethod
|             +> [14, 4] Identifier (void)
|             |  definition: type (builtin), type=void
|             +> [14, 9] Identifier (setX)
|             |  definition: method defined at [14, 4], type=void
|             +> ListDeclParam [List with 1 elements]
|             |  []> [14, 14] DeclParam
|             |      +> [14, 14] Identifier (int)
|             |      |  definition: type (builtin), type=int
|             |      `> [14, 18] Identifier (a)
|             |         definition: parameter defined at [14, 14], type=int
|             `> [14, 21] MethodBody
|                +> ListDeclVar [List with 0 elements]
|                `> ListInst [List with 2 elements]
|                   []> [15, 8] Println
|                   ||  `> [15, 16] ListExpr [List with 1 elements]
|                   ||     []> [15, 16] StringLiteral (j'utilise le setX de B)
|                   ||         type: string
|                   []> [16, 8] Assign
|                       type: int
|                       +> [16, 8] Selection
|                       |  type: int
|                       |  +> [16, 8] This
|                       |  |  type: B
|                       |  `> [16, 13] Identifier (x)
|                       |     definition: field defined at [2, 7], type=int
|                                 ---> Attribut de A
|                       `> [16, 17] Identifier (a)
|                          definition: parameter defined at [14, 14], type=int
`> [20, 0] Main
   +> ListDeclVar [List with 1 elements]
   |  []> [21, 5] DeclVar
   |      +> [21, 3] Identifier (B)
   |      |  definition: type defined at [12, 6], type=B
   |      +> [21, 5] Identifier (a)
   |      |  definition: variable defined at [21, 5], type=B
   |      `> [21, 6] Initialization
   |         `> [21, 8] New
   |            type: B
   |            `> [21, 12] Identifier (B)
   |               definition: type defined at [12, 6], type=B
   `> ListInst [List with 2 elements]
```

```
[]> [22, 3] MethodCall
||  type: void
||  +> [22, 3] Identifier (a)
||  |  definition: variable defined at [21, 5], type=B ---> Méthode bien redéfinie
||  +> [22, 5] Identifier (setX)
||  |  definition: method defined at [14, 4], type=void
[]> [23, 3] Println
    '> [23, 26] ListExpr [List with 2 elements]
       []> [23, 11] StringLiteral (a.getX() = )
       ||  type: string
       []> [23, 26] MethodCall
           type: int
           +> [23, 26] Identifier (a)
           |  definition: variable defined at [21, 5], type=B
           +> [23, 28] Identifier (getX)
           |  definition: method defined at [7, 3], type=int ---> Méthode bien hérité
                                                                  de A
```

On peut ici voir que la grande complexité de la Partie B avec objet est de pouvoir chaîner les environnements et de réutiliser les environnements parents dans les environnements enfants. La pré-construction des types et des environnements permet de mieux faire le lien entre environnements et facilite l'héritage d'attributs et de méthodes.

3 Implémentation de la Génération de Code

Pour la partie de la génération de code assembleur nous avons essayé de respecter le squelette de code déjà fourni. Afin de respecter le principe de hiérarchie des classes, nous avons factorisé au maximum notre code en créant la fonction génératrice de code **CodeGenReg()** d'abord dans la classe *AbstractExpr* et qui sera redéfinie dans les différentes feuilles de l'arbre en fonction des besoins. De même dans la classe *AbstractInst* la méthode **codeGenInst()** est créée pour générer le code des instructions et elle sera redéfinie dans les différentes feuilles (*IfThenElse, While, Return, AbstractPrint, ...*).

3.1 Gestion de la Mémoire, Registres, Labels

3.1.1 RegisterManager

Ayant à notre disposition plusieurs registres, nous avons décidé de créer une classe s'occupant de la gestion des registres : *RegisterManager*. Cette classe, nous permettra d'accéder au registre courant mais aussi s'occupera de toute l'allocation et de libérer les registres. Cela permet de toujours vérifier qu'on n'alloue pas trop de registres ou qu'on ne libère pas de registre non alloué. Par défaut, on considère que nous disposons bien de 16 registres mais cette valeur peut être modifiée à l'aide de la commande :

```
decac -r X fichier.deca
```

avec $4 \leq X \leq 16$

3.1.2 MemoryManager

Pour la gestion de la mémoire nous avons voulu prendre en attribut, dans notre classe *MemoryManager*, l'indice courant des adresses de Base Global (GB), de Base Locale (LB) et le maximum de l'indice possible pour la Base Locale. Ainsi, cette classe va pouvoir gérer les différentes allocations des adresses locales et globales. Elle permet aussi de renvoyer des *RegisterOffset* à l'aide des indices courants comme *offset*.

3.1.3 LabelManager

Cette classe permet la gestion des Labels et garantit l'unicité des Labels à l'aide de l'attribut entier **counter** et ainsi permet d'empêcher la création de deux Labels portant le même nom. Elle permet aussi l'unicité des Labels d'erreurs en les stockant comme attribut et en les créant dès leur premier appel.

On aurait pu optimiser le code en ajoutant une méthode qui ne crée le Label d'erreur et ne l'ajoute au code assembleur que lorsque ce Label est utilisé (notamment avec l'instruction BOV). Cependant, par manque de temps, nous avons ajouté tous les Labels d'erreurs à la fin du fichier *.ass* qu'ils soient utiles au programme ou non.

3.2 Génération de code Sans Objet

3.2.1 l'Affichage sur la Sortie Standard

Pour gérer les fonctions *print/println* nous avons décidé de créer deux méthodes **codeGenPrint()** et **codeGenPrintHexa()** qui permettent d'écrire soit en hexadécimal soit en décimal pour les flottants, et qui gèrent de la même manière les chaînes de caractères et les entiers.

Ce n'est que dans la méthode **codeGenInst()** de la classe *AbstractPrint* que l'une ou l'autre des méthodes est appelée en fonction de l'attribut booléen **printHexa**.

3.2.2 les Opérations

Pour les Opérations Arithmétiques, par souci de factorisation du code, la méthode **codeGenReg()** est redéfinie dans la classe *AbstractBinaryExpr* et fait appel à une méthode **geneInstru()** qui permet de générer les instructions en assembleur des Opérations Arithmétiques ($+, /, \times, \%, -$).

La méthode **codeGenReg()** permet, en fonction des registres libres ou non, d'allouer les registres qui vont être utilisés par l'opération. De plus, elle va aussi convertir automatiquement un entier en flottant lors d'une opération mixte entre les deux types. Ainsi, l'opération 2×3.2 est possible sans avoir à transtyper l'entier en flottant. Enfin, elle va vérifier les cas de dépassement (OverFlow) en faisant un appel au Label d'erreur correspondant. Par souci d'efficacité cet appel n'est effectué que quand un OverFlow est possible grâce à la méthode booléenne **canOverFlow()**. Cela regroupe donc les opérations entre flottants et les divisions ou modulo par 0. Ainsi, pour l'opération 2×3.2 le programme assembleur donne :

```
LOAD #2, R2 ; on met 2 dans le registre R2
FLOAT R2, R2 ; on le transforme en flottant
MUL #0x1.99999ap1, R2 ; on fait une multiplication avec 3.2 écrit en hexadécimal
BOV overFlowError.0 ; BOV = branchement if OVerflow
HALT
```

Concernant les opérations booléennes de comparaison ($<, \leq, >, \geq, ==, \neq$), dans la classe *AbstractOpCmp*, une comparaison entre les deux valeurs va d'abord être faite puis, en fonction de l'opération, soit 1 est mis dans le registre ou soit celui-ci est remis à 0. Ainsi, pour l'instruction ci-dessous nous obtenons le programme assembleur suivant :

```
    boolean b = true == true;
```

```
LOAD #1, R2
CMP #1, R2 ; comparaison de AbstractOpComp
SEQ R2      ; SEQ = Set if EQuals
STORE R2, 1(GB)
HALT
```

3.2.3 les Conditions et les Boucles

Pour gérer les Conditions et les Boucles nous avons utilisé des Labels. En effet pour une condition il suffit, après la comparaison faite dans *AbstractOpCmp*, de brancher en fonction du résultat sur le Label suivant à l'aide des instructions de branchements conditionnels.

Cependant, pour les conditions de type If Then Else, cela devient plus compliqué. En effet, il faut pouvoir gérer aussi les else if ; pour cela nous avons rajouté un booléen **local** qui permet d'indiquer si nous avons affaire à des conditions imbriquées ou non. Si cela n'est pas le cas, il faut donc créer le Label correspondant à la fin du If. De même, le else peut ne pas exister. S'il est présent il est donc important de créer un nouveau Label correspondant au début des instructions du else. Ainsi, pour un programme simple comme celui-ci on obtient un code assembleur suivant :

```
if(true==true){
    println("Hello");
}else{
    println("Bonjour");
}
```

```
LOAD #1, R2
CMP #1, R2 ; comparaison de OpComp
BNE startelse.2 ; BNE = Branchement if Not Equals
WSTR "Hello"
WNL
BRA endif.1
startelse.2:
WSTR "Bonjour"
WNL
endif.1:
HALT
```

Enfin, pour les boucles, encore une fois nous avons utilisé des Labels permettant d'indiquer le début de la condition, les instructions dans la boucle, et la fin de la boucle c'est à dire la suite du programme. La condition de la boucle étant gérée comme précédemment, cela n'est donc pas très compliqué à mettre en place dans la méthode **codeGenInst()**.

3.2.4 l'Assignation

Pour gérer l'assignation des variables à leur valeur, la méthode **codeGenReg()** est redéfinie dans la feuille *Assign*. Cette méthode va d'abord chercher un registre libre dans lequel mettre la valeur, s'il n'en a pas, elle va allouer un registre dans la zone locale. Ensuite, elle va assigner la valeur à la variable en utilisant les instructions STORE et LOAD.

3.3 Génération de code pour les Classes

Concernant l'implémentation des Classes nous avons fait le choix de ne pas trop avoir à modifier le code sans objet et donc de coder les différentes instances des classes comme une variable ordinaire. Ainsi, nous avons pu conserver notre code sans avoir beaucoup de changements à faire pour gagner beaucoup de temps. L'implémentation se repose donc sur deux passes différentes : une qui va d'abord construire la table des méthodes et l'autre l'initialisation des attributs, le codage des méthodes et du programme principal.

3.3.1 VTable : la Table des Méthodes

Pour l'implémentation des méthodes nous avons du rajouter une nouvelle classe : la table des méthodes, *VTable*. Cette classe qui est appelée à la création de la classe permet de stocker l'adresse du début de la classe dans la pile et un tableau de taille du nombre de méthodes. Ce tableau stocke des *LabelOperand*, des Labels utilisés comme opérandes. Les classes créées ainsi vont aussi stocker dans leur table les méthodes de leur classe mère grâce à l'implémentation lors la création de la *VTable*.

L'un des problèmes auquel nous nous sommes heurtés lors de l'implémentation des classes fut la création de la *VTable* de la classe Object. En effet, on considère que la classe Object est la classe mère de toutes les autres, et ses méthodes doivent donc apparaître dans toutes les tables de toutes les classes. Or, pour pouvoir créer la *VTable* nous devions rentrer dans la méthode **codeGenClass** dans *DeclClass*, cependant cette méthode n'est appelée que pour les classes créées par l'utilisateur. Ainsi, nous avons décidé de rajouter une partie de code qui ne sera atteint que la première fois et qui correspondrait à l'implémentation de la classe Object. Cette petite astuce se trouve donc dans la classe *ListDeclClass* et permet donc de créer la table d'Object avant de créer toutes les autres, avec l'utilisation de l'entier **first**.

De plus, nous avons stocké la Table des méthodes d'Object comme attribut dans la classe *DeclClass* pour pouvoir y accéder en tout temps sans avoir à la recréer à chaque fois. Cependant, dans cette implémentation nous n'avons pas eu le temps de réussir à correctement accéder à la méthode, **Equals**, d'Object dans toutes nos classes.

3.3.2 Initialisation des Attributs

Pour chaque attribut d'une classe, s'il n'est pas initialisé par l'utilisateur nous avons fourni une valeur par défaut à stocker à l'adresse correspondant à l'attribut. Cette valeur par défaut correspond donc à :

Type	Valeur par Défaut
int	0
float	0.0
class	null
boolean	false

3.3.3 Les Méthodes

Pour pouvoir tester le débordement de la pile, avec l'instruction TSTO, il fallait d'abord connaître la taille de la pile effective. Or, l'instruction TSTO doit être placée en début de bloc, car cette taille dépend du nombre de registres utilisés, du nombre de variables et d'autre paramètres. Pour résoudre ce problème nous avons décidé de découper en bloc précis les différentes parties du code, ainsi nous pouvons utiliser la méthode **addfirst()** de *DecacCompiler* qui permet, à la fin, d'ajouter tout en haut du bloc notre instruction avec la bonne taille pour la pile directement. Pour découper en bloc le code pour les différentes classes nous avons créé deux méthodes **startBlock()** et **endBlock()** qui s'occupent de créer le bloc, d'ajouter les registres sauvegardés au début du bloc et les libérer à la fin du bloc, calculer la taille de la pile, ...

3.3.4 l'Appel de Méthode

Pour pouvoir coder l'appel aux différentes méthodes, on ne peut pas directement utiliser l'instruction BSR qui s'occupe d'appeler la méthode. En effet, il faut allouer de la place pour les variables locales, récupérer les différents paramètres de la méthode et vérifier qu'il n'y a pas de déréférencement de null.