Projet Génie Logiciel

Documentation : Conception

Gr2 - Gl10 - 2023 - 2024

Membres du groupe:

CHRIF M'HAMED Mohamedou
DIAB Dana
KONE Madou
MOHAMED AHMED Mohamed Lemine

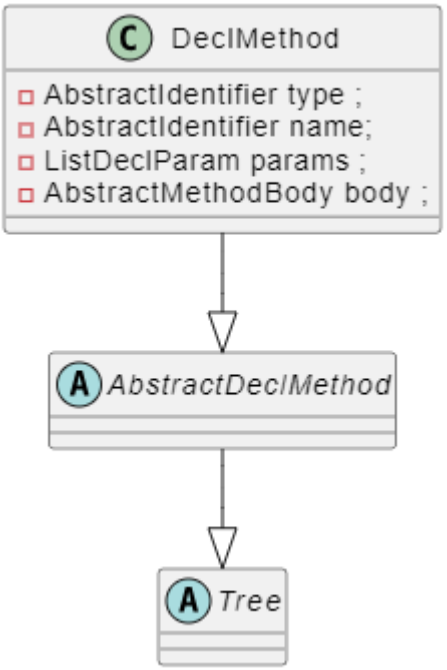
Sommaire

1. Conception du package tree : 2
2. Conception architecturale de l'étape B : 2
3. Conception architecturale de l'étape C : 6
4. Conclusion : 9

1. Conception du package tree :

Pour la conception du package **tree**, nous avons utilisé le patron de conception interprète.

Par exemple :
DECL_METHOD → DeclMethod[IDENTIFIER IDENTIFIER LIST_DECL_PARAM METHOD_BODY]

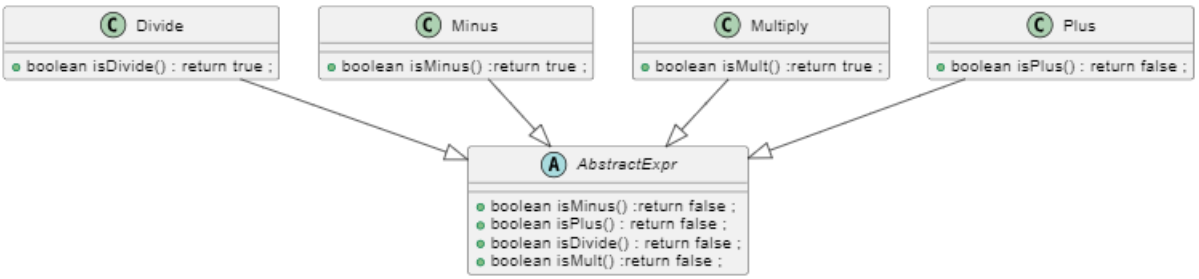


Et nous avons adopté la même démarche pour compléter la package tree pour **Deca Objet**. Cette conception nous a semblé être la plus simple et la plus utile pour ce que nous souhaitions programmer et développer.

2. Conception architecturale de l'étape B :

1. Conception pour les opérations

Dans l'étape B, nous avons besoin de connaître la nature de l'objet **AbstractExpr**, c'est-à-dire s'il s'agit d'une instance de **Plus**, **Minus**, .. etc. Cela permet de déterminer le type de retour d'une opération unaire ou binaire.
Pour cela, nous ajoutons quelques méthodes dans la hiérarchie du package `tree` comme le montre l'image suivante :



Dans la classe supérieure, on définit des méthodes telles que **isMinus()**, **isPlus()**, etc. Par défaut, ces méthodes retournent **false**.
Dans les classes dérivées, nous redéfinissons ces méthodes pour retourner **true** . Cela simplifie considérablement l'implémentation des fonctions utilitaires dont nous parlons dans le paragraphe suivant. De même, nous adoptons la même approche pour les autres opérations communes telles que **LT**(<), **GT** (>) , **And** (&&) ,... etc.

Nous avons préféré cela afin d'éviter d'avoir à utiliser une dizaine d'instructions "instanceof".

2. Opérations et prédicats sur les domaines d'attributs :

Pour les fonctions utilitaires, on crée un package appelé "util". À l'intérieur, il y a une classe "Util" où nous définissons toutes les fonctions utilitaires en tant que méthodes statiques. Ainsi, dans la partie B, si nous avons besoin d'une fonction utilitaire, nous l'appelons à travers la classe "Util". Par conséquent, nous n'avons pas besoin d'instancier la classe "Util", ce qui évite d'utiliser beaucoup de mémoire.

Dans cette partie, nous allons lister les fonctions utilitaires et leur signature en justifiant le choix des paramètres.

a. Relation de sous-typage :

```
/**
 * Relation de sous-typage :
 * @param compiler contient "env_types"
 * @param type1
 * @param type2
 * @return  boolean : indique si type1 est un sous type de type2

 */

public static boolean subType(DecacCompiler compiler , Type type1 , Type type2)
```

La fonction **subType** retourne un booléen qui indique si le type1 est un sous-type du type2. Pour cela, elle prend en paramètres deux objets **Type** : `type1` et `type2`, ainsi que l'objet **DecacCompile** pour accéder à l'environnement de types du compilateur.

b. Compatibilité pour l'affectation :

```
/**
 *  Compatibilité pour l'affectation :
 * @param compiler
 * @param type1
 * @param type2
 * @return boolean : indique si type2 peut etre affecte à type1
 */

public static  boolean  assignCompatibe(DecacCompiler compiler , Type type1 , Type type2)
```

La fonction **assignCompatible** indique si le type1 peut être affecté au type2. Elle prend en paramètre deux objets **Type** : `type1` et `type2`, ainsi que l'objet **Decacompiler** pour accéder à l'environnement de types du compilateur.

c. Compatibilité pour la conversion :

```
/**
 *  Compatibilité pour la conversion :
 * @param compiler
 * @param op
 * @param type
 * @return boolean : qui indique est ce que le type1 peut etre convertir à type2
 *
 */

public static boolean castCompatibel(DecacCompiler compiler , Type type1 , Type type2)
```

La fonction **castCompatible** indique si le type1 peut être converti au type2. Elle prend en paramètre deux objets **Type** : `type1` et `type2`, ainsi que l'objet **Decacompiler** pour accéder à l'environnement de types du compilateur.

d. L'opération **type_unary_op** :

```
/**
 *  @return : return le type d'une opération Uniare
 */
```

```
public static Type type_unaru_op(DecacCompiler compiler , AbstractExpr op , Type type)
```

La fonction **type_unary_op** retourne le type du résultat d'un opérateur unaire. Elle prend en entrée un objet **DecacCompiler** pour accéder au type prédéfini dans le compilateur (comme int, float, etc.), un objet **AbstractExpr** qui modélise l'opération unaire (UnaryMinus, Not), et un objet **Type**.

e. L'opération **type_arith_op**

```
/**
 * @return : return le type d'une opération binaire arthimétique
 */
public static  Type type_arith_op(DecacCompiler compiler ,Type t1 , Type t2)
```

La fonction **type_arith_op** retourne le type du résultat d'un opérateur arthimétique binaire. Elle prend en entrée un objet **DecacCompiler** pour accéder au type prédéfini dans le compilateur (comme int, float, etc.), un objet **AbstractExpr** qui modélise l'opération binaire (Plus , Mult), et un objet **Type**.

f. La méthode **type_binary_op :**

```
/**
 * @return : return le type d'une opération binaire
 */

public static  Type type_binary_op(DecacCompiler complier , AbstractExpr op , Type t1 , Type t2)
```

La fonction **type_binary_op** retourne le type du résultat d'un opérateur binaire. Elle prend en entrée un objet **DecacCompiler** pour accéder au type prédéfini dans le compilateur (comme int, float, etc.), un objet **AbstractExpr** qui modélise l'opération binaire (And ,OR ..), et un objet **Type**.

g. La méthode **type_instanceof_op :**

```
public static Type  type_instanceof_op(DecacCompiler compiler , Type type1 , Type type2)
```

La fonction **type_instanceof_op** retourne le type **boolean** si l'opération **instanceof** est valide et **null** sinon.
Elle prend en paramètre un objet **DecacCompiler** pour accéder au type **boolean** et deux objets de type **Type**.

3. Conception architecturale de l’étape C :

A. Sous-Langage HelloWorld :

Nous avons d'abord commencé par développer le sous-langage HelloWorld, pour lequel nous avons complété quelques fonctions. Ce langage n’a pas nécessité de conception ou d'effort, contrairement aux parties qui ont suivi.

B. Sous-Langage SansObjet:

Contrairement au premier incrément, cette partie a nécessité davantage d’efforts. Avant d’entamer la programmation, il a fallu réfléchir à l’architecture de notre code afin de partir d’une base solide. L'étape C en général repose énormément sur la conception et l’architecture établies lors de l'étape B. L'étape C nous a également parfois aidé à prendre les bonnes décisions ou à modifier légèrement la conception de l'étape B.

Dans ce sous-langage, nous avons simplement décidé de redéfinir la méthode codeGenInst au niveau de presque toutes les classes. Étant donné qu'en entrée, nous avons un objet de type Instr dont nous ne connaissons pas le type concret, l'appel à **codeGenInst** sur les instructions dans main fait automatiquement appel à celle dans la classe nécessaire.

Par exemple, pour { 1+2; }, si l'on part de **codeGenProgram** puis **codeGenMain**, cette dernière va appeler le **codeGenListInst** sur la liste d'instructions qui est censée, dans ce cas, contenir une seule instruction. Elle va ensuite appeler **codeGenInst** des classes allant de **BinaryExpr** jusqu'à **Add** dans notre hiérarchie, laquelle à son tour va appeler codeGenInstr de la classe **IntLiteral**.

Nous avons suivi cette logique dans la plupart de la partie SansObjet. Nous avons tenté de tirer parti du polymorphisme offert par le langage Java pour développer le sous langage SansObjet.

En ce qui concerne les opérateurs de comparaison, ainsi que les expressions booléennes (And et Or), nous avons adopté une approche légèrement différente. Nous avons essayé de mettre en place leur codage, comme expliqué dans [GenCode], en créant une nouvelle classe qui code toutes ces expressions en utilisant des instances de la classe. Cependant, nous avons rapidement réalisé que cette méthode n'était pas facile à utiliser et à intégrer dans le package Tree. Par conséquent, nous avons subdivisé cette classe en petits fragments que nous avons intégrés dans les classes correspondantes (Lower, Equals, And, etc.) sous la forme de méthodes portant le nom **codeCondition**.

Les résultats ont montré que cette conception a été efficace jusqu'à maintenant.

En ce qui concerne la gestion des registres, nous avons élaboré un algorithme visant à déterminer le registre dans lequel se trouve le résultat de l'expression et à permettre la réutilisation des registres une fois l'évaluation achevée. La classe RegisterManager a été dédiée à la gestion des registres, comprenant plusieurs attributs, dont:

- numR, qui représente le numéro du premier registre libre (ou le dernier utilisé), initialisé à 2,
- regMax, fixé à 15 par défaut ou à X dans le cas de l'option -r X, indiquant le nombre total de registres pouvant être utilisés,
- maxOver, un booléen, qui est à true si nous avons dépassé nos capacités en registres, signalant ainsi le début de l'utilisation de la pile pour stocker nos calculs intermédiaires.
- registerOfOp1, qui va contenir le numéro du registre qui contient le résultat de la partie gauche dans les expressions binaires.

A la fin de l'évaluation totale d'une expression, on remet numR à 2.

Par exemple, si on applique l'algorithme sur l'expression (2+3) * 4, nous aurons l'arbre d'appels représenté dans la figure suivante :

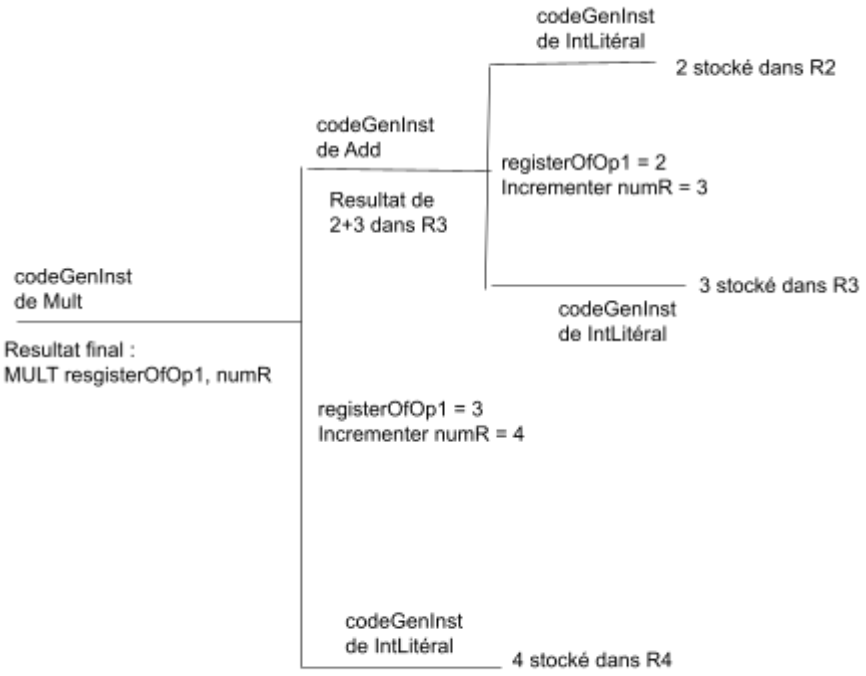


Figure 1 : Exemple de gestion de registres (Algorithme implémenté dans la classe AbstractBinaryExpr)

Le seul point qui manquait était la gestion de la pile lorsqu'on n'avait plus de registres libres, et donc avec l'option -r. Nous avons introduit **maxOver** à ce moment-là, mais nous avons rencontré un bug qui a retardé la mise en œuvre de l'option -r pour le rendu intermédiaire. Cela a été résolu dans les heures qui ont suivi l'heure limite du rendu intermédiaire.

De plus un autre choix qu'on a eu à faire était la modification du prototype des fonctions codeGenPrint. En effet, étant dans la classe AbstractExpr, on avait besoin de l'attribut printHex(qui se trouve dans la classe AbstractPrint) afin de gérer l'affichage hexadécimal, la meilleure solution qu'on a trouvé était de modifier le prototype de codeGenPrint car les autres solutions demandaient à créer des attributs statiques ou même de faire des test instanceof, qui plus est, ces méthodes semblaient ne pas fonctionner.

En conclusion, nous avons pris conscience que pour être à l'aise dans le développement en général, et plus particulièrement dans celui de ce compilateur, il est crucial de partir d'une bonne architecture de classes et de conception. Les algorithmes ont été facilement trouvés, et nous n'avons pas eu besoin de recourir à des structures de données complexes.

C. Langage Complet :

N'ayant pas pu terminer cette partie dans le délai imparti, nos choix de conception n'ont pas couvert l'ensemble des classes.

1. Passe 1 :

   Cette passe concernait la génération de la table de méthode.

   Pour cette partie, on a eu recours à une pile qui permettait d'obtenir,à partir d'une classe, toutes les méthodes de ses parents jusqu'à Object tout en prenant en compte les redéfinitions, c'est-à dire, prendre la méthode la plus récemment redéfinie. Cela étant fait, il suffisait juste de faire un parcours de méthode classe par classe pour générer la table des méthodes.

   Cependant pour la table de méthode de Object, on a été beaucoup manuel car la classe object n'est pas définie explicitement dans un code deca et donc il ne fait pas partie de la liste des classes.

2. Passe 2 :

La deuxième passe comporte les points suivants : codage des champs de chaque classe, codage des méthodes de chaque classe (déclarations et instructions), codage du programme principal (déclarations et instructions).

La génération de code pour le langage Déca Complet présente quelques différences par rapport à celle du sous-langage SansObjet, principalement en raison de l'introduction d'un tas et de nouveaux pointeurs sur la pile, tels que LB. Cela nous a obligés à définir une méthode codeGen pour les variables globales dans main et une autre pour les variables locales à une méthode. Quelques autres changements légers et l'ajout de méthodes ont été mis en place, mais en général, nous avons progressé sur ce langage Complet de la même manière que pour le Déca SansObjet.

Malheureusement, nous n'avons pas pu terminer tout ce que nous souhaitions faire et implémenter avant le rendu final. La conception et l'architecture de cette partie peuvent être revues en vue d'améliorations.

4. Conclusion :

En conclusion, l'architecture et la conception dans un projet de génie logiciel sont essentielles pour assurer la clarté, la maintenabilité et l'évolutivité du code. Une structure bien définie facilite la gestion des changements et favorise la réutilisabilité du code. Une conception réfléchie impacte directement la qualité du code, permettant une meilleure gestion de projet et une adaptation efficace aux évolutions des exigences. En somme, un investissement initial dans une architecture solide est crucial pour le succès à long terme d'un projet logiciel, particulièrement pour ce projet de compilateur Decac.