Documentation de conception

Groupe 44

COLIN-SOHY Mathéo

DIMITRIOU Tristan

EL HIT Oumayma

GAUX Antoine

HARRAUD Paul-Louis

Table des matières

1 Introduction

Ce document va présenter les différents choix réalisés lors de la conception et l'implémentation de la vérification contextuelle du programme source et de la génération du code assembleur associé.

2 Vérifications contextuelles

2.1 Généralités

Cette section va aborder la partie de vérifications contextuelles du programme afin d'accepter et uniquement accepter les programmes valides selon la grammaire contextuelle de deca. Cette vérification est effectuée en trois passes, que nous verrons plus en détail dans la suite de cette partie. Elle est entièrement gérée dans le code par un appel à la fonction `Program.verifyProgram(DecacCompiler)` qui s'exécute sans problème dans le cas d'un programme valide mais relève une erreur contextuelle si le programme est invalide (`ContextualError`, également signalée à l'utilisateur dans le message d'erreur qui commence alors par "[ContextualError]").

Note : dans cette partie, pour chacune des figures, les méthodes indiquées pour chacune des classes sont celles appelées durant la passe et celles-ci uniquement.

2.2 Environnement des types

Dans la spécification de deca, certains types sont prédéfinis à l'instar de `int`, `boolean` ou encore `Object` (ce sont les types `BUILTIN`), et l'utilisateur peut également en définir dans son programme sous la forme de classes. Afin de stocker ces informations, un environnement des types est créé avant la première passe contenant les types `BUILTIN`, puis enrichi des types définis par l'utilisateur lors de la première passe. Cet environnement permet alors de garder la correspondance entre les symboles définissant des types et leur définition.
La structure de donnée utilisée pour stocker ces informations est un map `Map<Symbol, Definition>` permettant donc de récupérer la définition associée à un symbole. Dès lors, il suffit d'avoir le symbole de l'objet actuel ou un accès au `SymbolTable` présent dans `DecacCompiler` pour obtenir la définition de l'objet voulu dans l'environnement des types. Le choix de la structure de dictionnaire semble assez évidente puisque l'on cherche à associer des `Symbol` à des `Definition`, qui sont deux objets qu'il serait difficile d'indexer. Ensuite, il est interdit dans un même environnement d'avoir deux éléments ayant le même nom ce qui est aussi le cas dans un dictionnaire, ce qui rend plus simple la détection de double-définitions pour le même symbole. Enfin, le faible nombre de variables dans un même environnement (rarement au delà de la vingtaine) permet de garder des complexités temporelle et spatiales raisonnables.

2.3 Environnement des expressions

Ce type d'environnement, très similaire au précédent (en particulier pour la strucuture de donnée utilisée), permet de garder la correspondance entre un symbole et sa définition dans le contexte actuel. On différencie alors trois contextes possibles :

- L'environnement des variables globales (celles définies dans le `main` du programme)
- L'environnement des éléments d'une classe (ses champs et ses méthodes)
- L'environnement des méthodes (ses paramètres et variables locales)

Ces environnements ont alors la possibilité de s'empiler, ainsi, si une variable n'est pas définie dans l'environnement actuel, le programme va chercher sa définition dans l'environnement parent et ainsi de suite jusqu'à trouver une définition ou atteindre l'environnement le plus haut, qui n'a plus de parent (dans ce cas, une erreur est retournée puisque la variable n'est donc pas définie).
Exemple, si on se trouve dans l'exécution de la méthode `printB` de la classe `B` qui hérite de `A`, la définition sera recherchée dans l'environnement de `printB`, puis dans celui de `B` et enfin celui de `A`.

2.4 Passe 1 : Nom et hiérarchie des classes

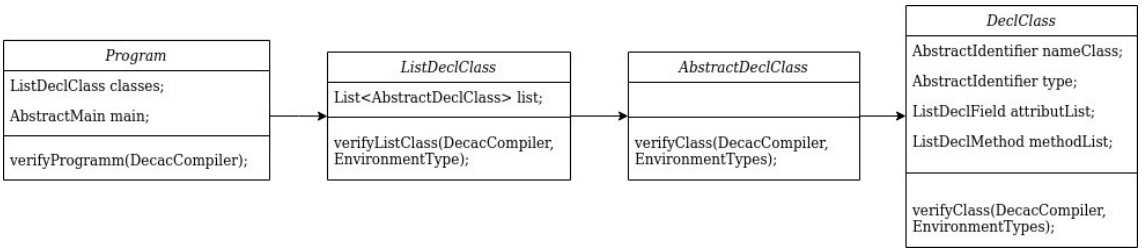

FIGURE 1 – Architecture simplifiée de la passe 1

Dans cette passe, l'objectif est donc de répertorier l'ensemble des classes définies par l'utilisateur, tout en vérifiant que leur nom est valide (ne pas utiliser deux fois le même identifiant) et dans le cas où elles enrichissent une classe mère, que celle-ci existe et est bien une classe.

La structure simplifiée de cette passe est indiquée dans la Figure 1. On effectue donc un simple parcours de la liste des classes définies par l'utilisateur dans son programme, en les ajoutant successivement à l'environnement des types. Lors de cet ajout, on garde également un lien avec la définition de la classe mère (`Object`, ou définie par le programme via `extends`), et on crée également une étiquette (`Label`) unique, qui sera le préfixe de celles des méthodes de cette classe, utiles pour la génération de code en aval.

2.5 Passe 2 : Champs des classes et signature des méthodes

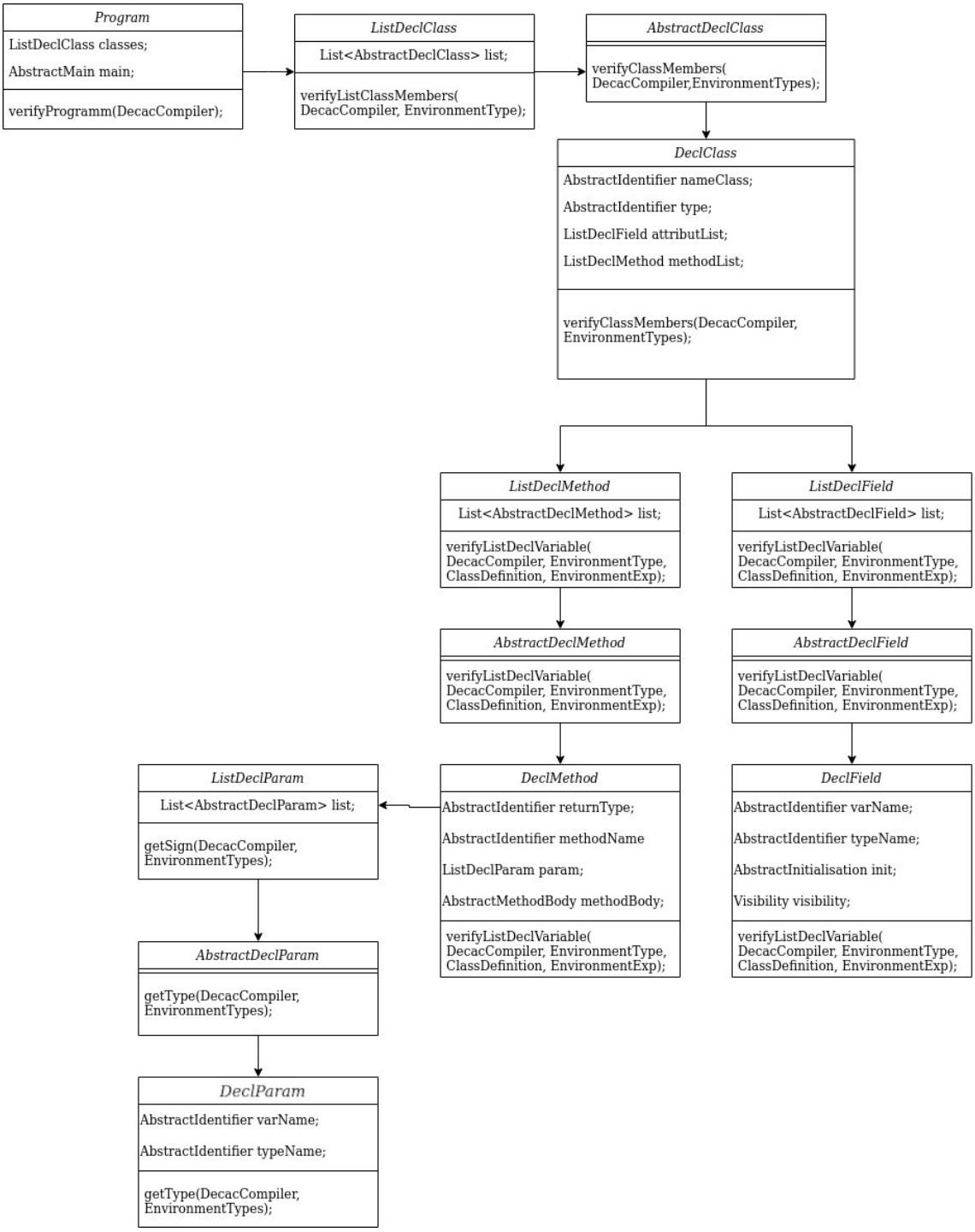

FIGURE 2 – Architecture simplifiée de la passe 2

Dans cette seconde passe, l'objectif est de répertorier pour chacune des classes, ses champs, ses méthodes et pour chacune de ces dernières, leur signature et type de retour. C'est durant cette passe que les héritages de champs et de méthodes ainsi que leurs redéfinitions sont gérées par le compilateur, permettant durant la génération de code, de faire appel à la méthode voulue. Les sources d'erreurs contextuelles pour cette passe sont nombreuses : redéfinition de champs ou de méthodes, types inconnus tant pour les champs, les types de retour ou dans la signature de chaque méthode.

La structure simplifiée de cette passe est indiquée dans la Figure 2. Avec le même parcours de la liste des classes définies dans le programme, on ajoute pour chacune, dans son environnement de classe, la définition de ses différents champs et de ses différentes méthodes. Lors de ces ajouts, on crée, pour chacune des méthodes,

son étiquette (`Label`) unique, avec le préfixe créé durant la passe 1.

2.6 Passe 3 : Vérification des initialisations, expressions et instructions

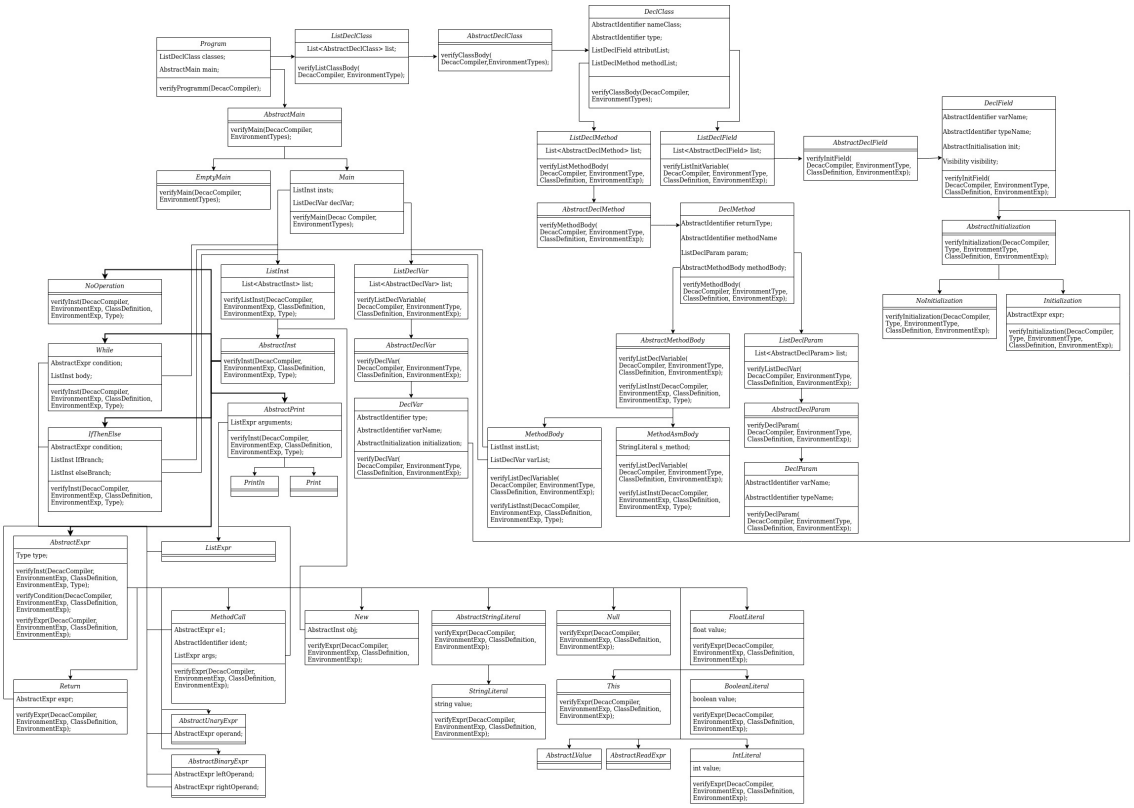

FIGURE 3 – Architecture simplifiée de la passe 3

Cette dernière passe est la plus ramifiée, la plus complexe mais aussi celle qui va relever le plus d'erreurs contextuelles possibles. Elle est également essentielle pour la génération de code puisque c'est elle qui va faire le lien entre les variables et leurs définitions à chacune de leurs utilisations. Elle permet en particulier de distinguer deux variables qui auraient le même nom dans deux environnements différents et de choisir la définition adéquate à chacun des appels sur le symbole.

La structure simplifiée de cette passe est indiquée dans la Figure 3. Elle commence par un parcours de la liste des classes. Dans chacune d'elles, on vérifie l'initialisation des champs, puis le corps des méthodes. Une fois ce parcours effectué, la vérification contextuelle est également faite sur le programme principal du fichier. Durant cette passe, les appels de méthode, la définition des variables et les différents calculs peuvent être vérifiés puisque toutes les informations nécessaires (nom des nouvelles classes, signatures des méthodes, ...) ont pu être récupérées par le programme lors des passes précédentes.

2.7 Impact de l'architecture sur les différents sous langages

Les passes 1 et 2 ne sont nécessaires que pour le langage avec-objet et par conséquent inutile pour les langages hello-world et sans-objet. Cependant, la passe 3 doit bien être effectuée pour les différents sous-langages, mais sans avoir nécessairement besoin de l'ensemble des fonctionnalités. Ainsi, il est possible de modifier une grande partie du code sans aucun risque pour le langage hello-world ni pour la langage avec-objet

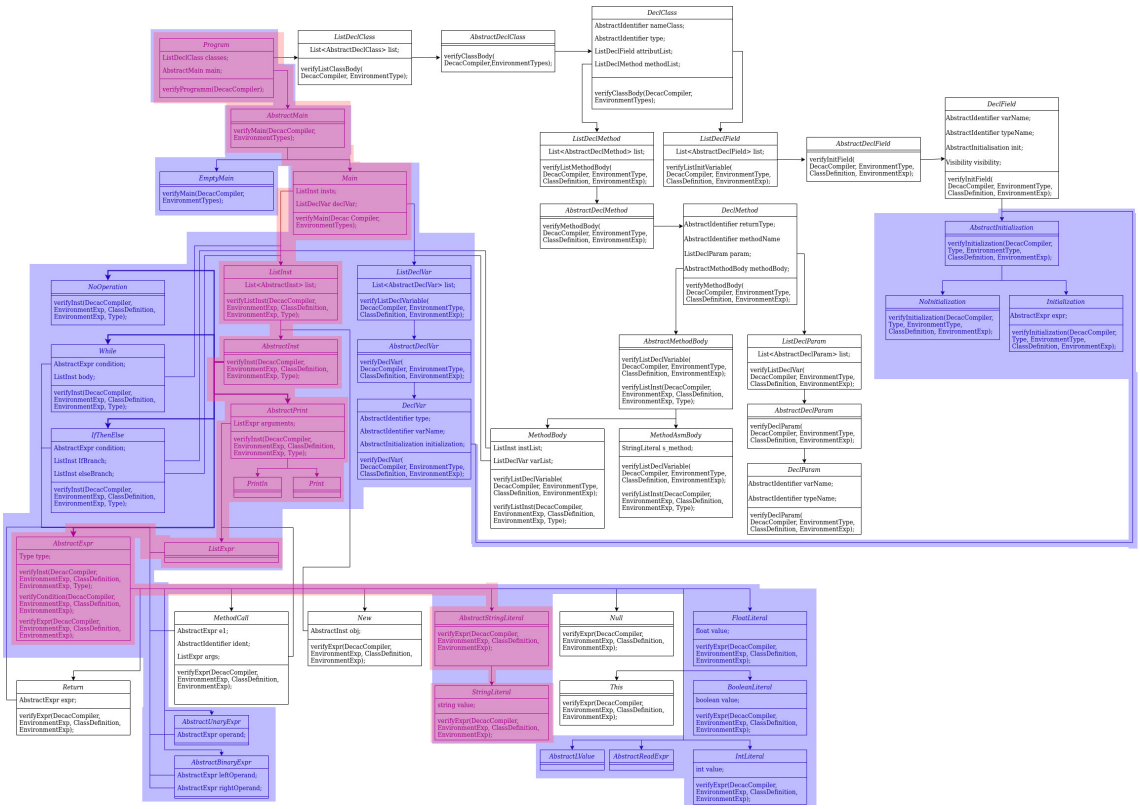

FIGURE 4 – Découpage de l'architecture par sous-langage de deca (hello-world en rouge, sans-objet en bleu)

3 Génération de code

3.1 Généralités

Cette section va aborder l'aspect génération de code du projet. On va donc montrer les points importants de l'implémentation. On parlera spécifiquement des classes ajoutés dans le package gencode, de comment on implémente la génération, des principales difficultés et de comment on résout les problèmes qu'elles engendrent. Voici une représentation légèrement simplifiée de l'architecture (on ne met en valeur que les fonctions de génération de code, et certaines autres qui sont utiles à la génération).
Un point important de la génération de code est que le code assembleur est très loin d'être optimisé. Nous avons préféré faire des choix simplificateurs permettant d'implémenter plus de fonctionnalités de meilleurs qualités. Nous reviendrons sur l'impact de ce choix dans un autre document.

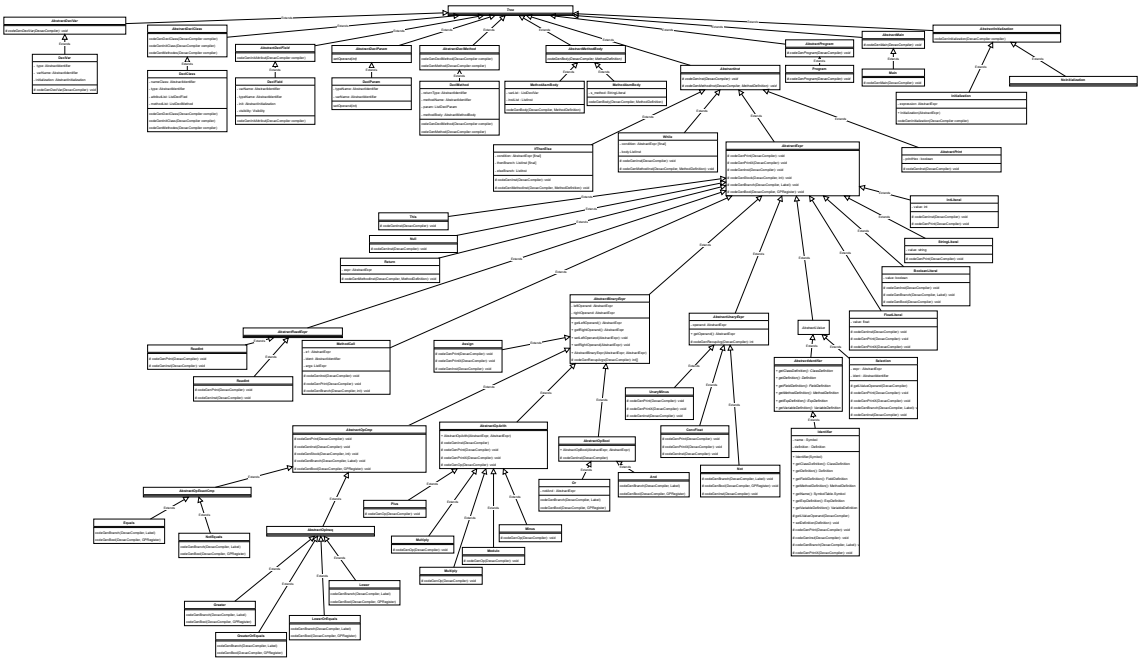

FIGURE 5 – Architecture simplifiée de la génération de code

3.2 Package gencode

Le package gencode contient 3 classes.

— `ErrorManager`, qui s'occupe de générer, si besoin, le code de gestion d'erreur. Il est par exemple appelé lors de l'instruction new pour générer le test de tas plein.
— `StackTracker`, qui suit l'état de la pile, et permet donc de placer correctement les éléments sur la pile. On l'utilise surtout pour tester au tout début du programme le dépassement mémoire.
— `RegisterManager`, qui s'occupe d'allouer les registres *GlobalPurpose* et sauvegarder les contenus lors d'appel de fonction.

`ErrorManager` est donc appelé dès que l'instruction BOV doit être ajouté dans le programme assembleur. Séparer ainsi les tests d'erreurs permet une gestion très simple et claire de l'option du compilateur pour ignorer les erreurs (option -n).

`StackTracker` est utilisé de deux manières différentes dans la génération. Au début du programme lors de la création de la table des méthodes, on l'utilise comme repère pour placer les éléments correctement et enregistrer leur emplacement. Dans la suite de la génération de code, on évite de l'utiliser pour 2 raisons. La première est que l'on ne suit pas l'évolution suite à un appel de méthode (l'instruction BSR rajoute 2 éléments dans la pile, on ne les prend pas en compte), la deuxième est qu'une erreur de tracking ferait planter tout le programme, de manière très peu explicite.

`RegisterManager` stocke N booléens dans un tableau, chacun indiquant si oui ou non le registre est vacant (donc libre). Nous avons choisi le tableau comme structure de stockage pour plusieurs raisons :

— On accède de nombreuses fois à cette structure, il faut donc un accès rapide et peu cher
— Un tableau de booleen est léger en mémoire

— La taille du tableau est définie au début de la compilation car le nombre de registre est constant durant la compilation
— Il n'y a pas de problème à n'utiliser que certains registres et pas d'autres, donc le parcours peut se faire de droite à gauche ou de gauche à droite (on a choisi la première solution).

3.3 Les différentes méthodes

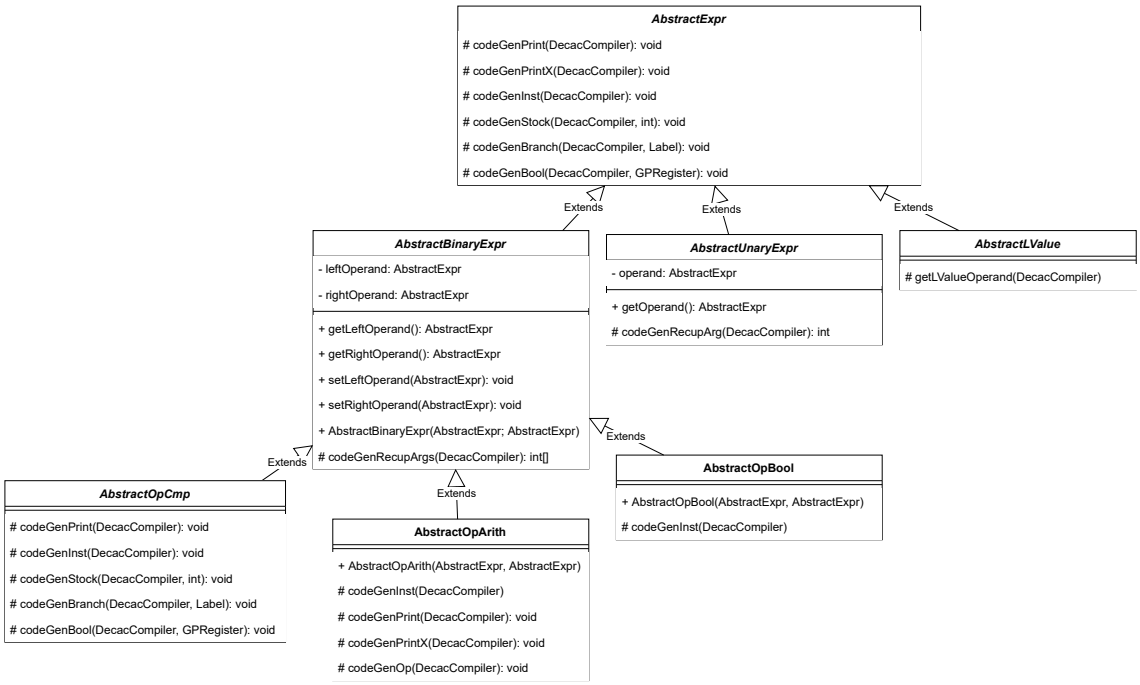

FIGURE 6 – Coeur abstrait simplifié de l'architecture

De la même manière que les sources données utilisaient plusieurs méthodes `codeGen` (en particulier Inst et Print), nous avons décidé de suivre cette logique en en créant d'autres. On obtient finalement :

— `void codeGenInst(DecacCompiler compiler)` réalise l'instruction et, si un résultat peut être émis, le stocke dans le registre `R1` .
— `void codeGenMethodInst(DecacCompiler compiler, MethodDefinition methodDefinition)` réalise dans beaucoup de cas l'exact identique que `codeGenInst` . Cependant, cette méthode est utilisée pour la génération de code des méthodes, car l'accès à la `MethodDefinition` est parfois indispensable dans notre implémentation.
— `int codeGenStock(DecacCompiler compiler)` réalise l'instruction et stocke le résultat de l'instruction dans un registre, le numéro du registre est l'entier renvoyé. Si l'entier est négatif, le résultat a été stocké dans la pile à la place.
— `void codeGenRecupArgs[s](DecacCompiler compiler)` récupère l'argument [ou les arguments], les place dans des registres (ces registres peuvent être scratch).
— `void codeGenPrint[X](DecacCompiler compiler)` réalise l'instruction et affiche le résultat.
— `void codeGenBranch(DecacCompiler compiler, Label lb)` réalise le test booléen associé pour mettre les codes conditions aux bonnes valeurs, puis génère l'instruction de branchement adéquate.
— `void codeGenBool(DecacCompiler compiler, GPRegister rDest)` réalise le test booléen et stocke sous forme d'un entier le résultat dans un registre spécifié (0 représentant false, 1 représentant true).
— `void codeGenPrintBool(DecacCompiler compiler)` réalise l'instruction et affiche true ou false en fonction du résultat booleen de l'instruction. Cette fonction n'est pas utilisée mais est présente. Si la syntaxe du langage l'autorisait, on pourrait ainsi afficher des booleens.
— `void codeGenOp(DecacCompiler compiler)` réalise l'instruction associé à l'opération (`Plus` par exemple).

Il y a d'autres fonctions de ce type, mais elle ne sont utiles qu'une fois (`codeGenProgram`) par exemple. On les trouve toutes dans la figure 5. Un point très important ici est que la "réalisation de l'instruction" est très souvent uniquement gérée par `codeGenInst` et les autres méthodes l'appellent.

Par exemple, le code de `AbstractExpr.codeGenStock` :

```
protected int codeGenStock(DecacCompiler compiler){
this.codeGenInst(compiler);
[...]
return i;}
```

Il ne faut donc pas appeler les deux méthodes.

3.4 Le stockage de valeurs temporaires

Cette partie concerne la gestion des registres et le stockage des valeurs temporaires, comme par exemple les résultats intermédiaires lors d'un calcul. Nous avons réalisé ce travail en deux temps. Nous avons commencé par établir un système extrêmement simple pour permettre d'avancer dans le projet plus rapidement, puis nous avons fait la transition vers un système moins naïf avant de nous attaquer à la partie objet.

La première idée est de stocker toutes les valeurs temporaires dans la pile. De cette manière, il n'y a aucun problème d'allocation de registre, car ils ne sont utilisés que ponctuellement pour faire des opérations. On se libère de toutes allocations de registres et donc de tout code associé. Cela pose un problème qui est une place mémoire plus grande, car certains programmes qui ne dépasseraient pas en utilisant les registres dépassent en utilisant la pile. On perd donc en codes fonctionnels.

Nous avons utilisé cette stratégie jusqu'à la fin du langage sans objet, et nous avons fait la transition vers la méthode plus classique à ce moment là. La gestion de l'allocation est géré par le `RegisterManager` , lui même principalement appelé dans `codeGenStock` et `codeGenRecupArg[s]`. Ces méthodes sont définies à un haut niveau d'abstraction (dans `AbstractExpr` pour `codeGenStock`) pour factoriser le code.

3.5 Les booléens

Les booléens sont utilisés de deux manières différentes, soit en tant que condition pour choisir la prochaine instruction (le branchement), soit comme une valeur que l'on calcule et que l'on stocke pour plus tard (une valeur).

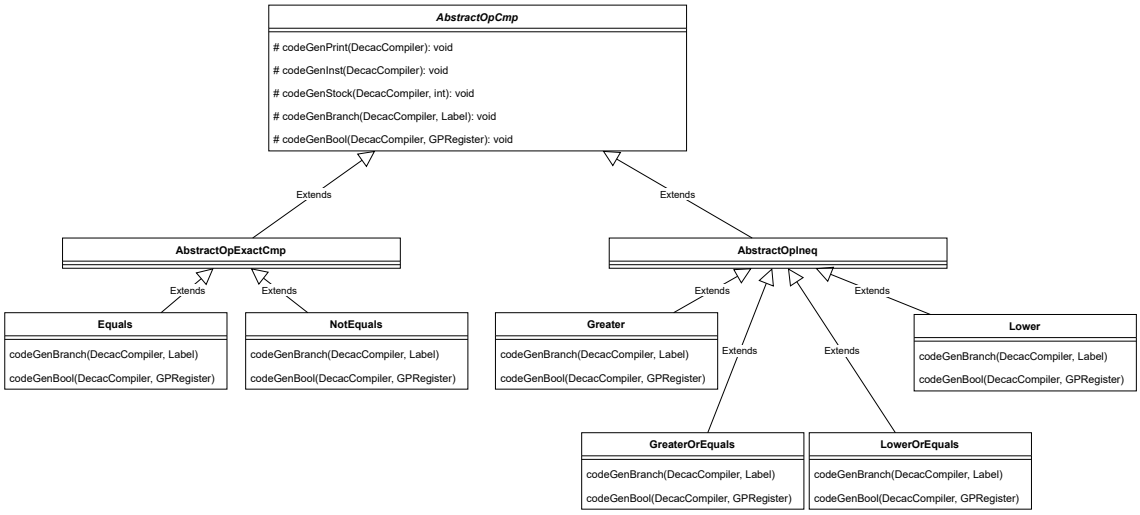

FIGURE 7 – Architecture simplifiée de la gestion des booléens

On distingue dans le code ces deux aspects via les methodes `codeGenBranch` et `codeGenBool`.
`codeGenBranch` est la méthode appelé pour gérer le branchement lors d'instruction conditionnels. Elle appele la méthode `codeGenInst` puis lui rajoute l'opération de branchement `Bcc`.
`codeGenBool` calcule le booléen de l'instruction et stocke le résultat (1 ou 0) dans le registre argument. On aurait pu "fusionner" ces deux méthodes en une seule, en transformant la génération du booléen en une instruction conditionnelle (donc généré par `codeGenBranch`) assignant la valeur associé en fonction du résultat. Cependant, l'instruction assembleur fournie `Scc` est bien plus efficace, on a donc choisi cette méthode.

3.6 Ordre de génération

On génère le code dans le sens suivant :
— Création de la table des méthodes (appel de `ListDeclClass.codeGenListDeclClass`)

— Génération du code principal (appel de `AbstractMain.codeGenMain`)
— Gestion des codes d'erreurs (appel de `ErrorManager.codeGenError`)
— Initialisation des champs des classes (appel de `ListDeclClass.codeGenListInitClass`)
— Génération du code des méthodes (appel de `ListDeclClass.codeGenListMethodes`)

3.7 Étiquetage

Un des problèmes de la génération de code est l'étiquetage des branchements. Nous avons résolu ce problème en créant un identifiant unique à l'instance du `DecacCompiler` à chacun des éléments suivants :
— Classes
— Methodes
— While
— IfThenElse
— Not
— And
Ainsi, à chaque fois qu'un de ces objets va devoir créer des étiquettes, on récupère un identifiant qu'on lui associe. Ainsi, le premier objet déclaré va avoir pour étiquette associé `object_name.1`, et sa première méthode déclaré aura l'étiquette `object_name.1.method_name.2`. L'avantage de cette méthode est qu'il est impossible d'avoir deux étiquettes identiques (on pourrait avoir recouvrement sans les points, par exemple un objet Class1 auquel serait associé l'identifiant 1 recouvrerait l'objet Class d'identifiant 11). On évite aussi les interférences entre les autres étiquettes (aucun problème pour nommer une classe IfThenElse, les identifiants sont forcément différents).