Compilateur Deca

Groupe 35

Tony Buthod-Garçon, Thomas Clastres, Thami Benjelloun,

Lucas Bertrand, Nizar Bel Hadj Salah

Analyse du code

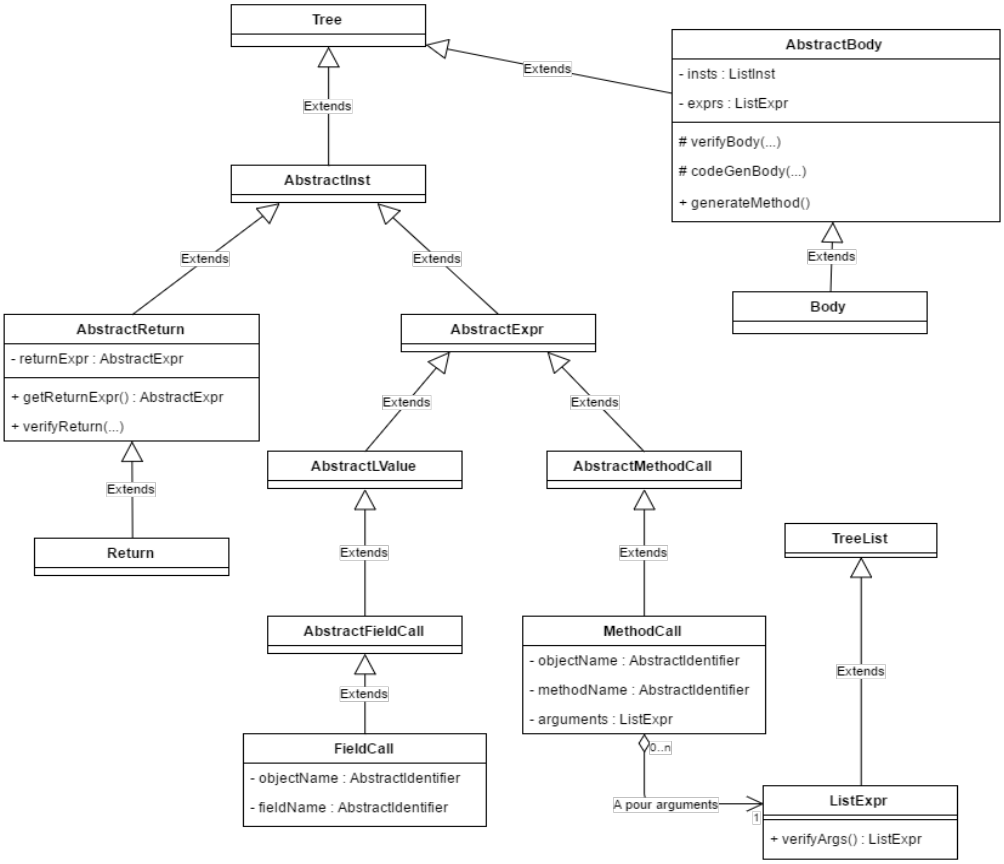
Présentation générale.

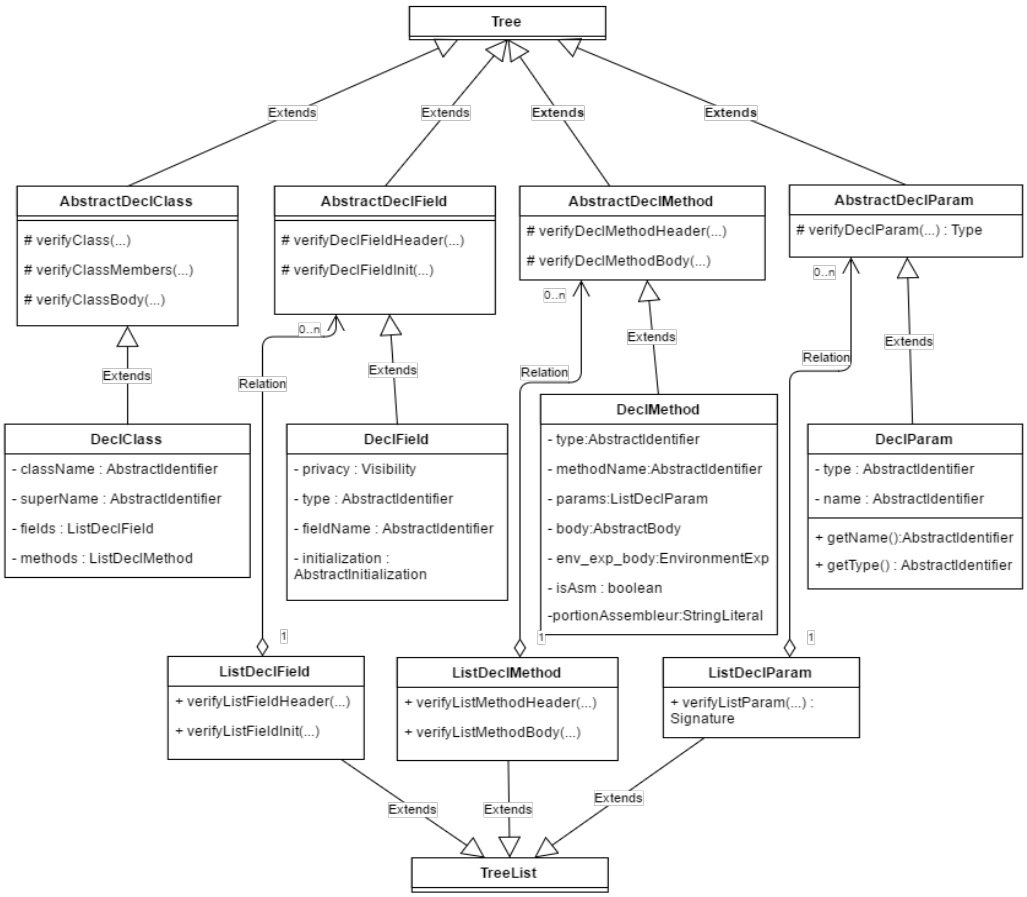
Le lexer et le parser permettent de générer un arbre à partir du programme. De la manière dont il est construit, la structure de l'arbre essaye d'être la plus générale possible et la plus souple possible. Ainsi, il est tout le temps créé des classes abstraites, à partir desquelles hérite une ou plusieurs sous-classes. Même si une classe abstraite n'était pas nécessaire dans certains cas (par exemple pour AbstractIdentifier dont la seule sous-classe est Identifier), la création d'une classe abstraite permet d'ajouter facilement de nouvelles fonctionnalités sans modifier la structure déjà existante.

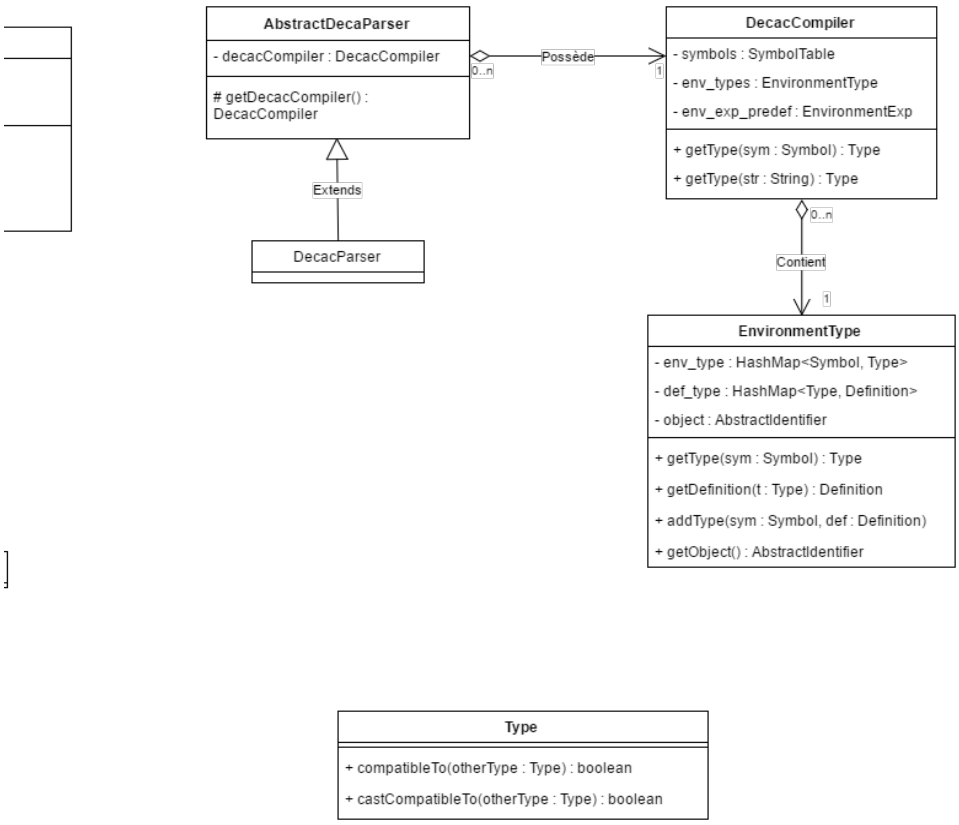
La notion d'héritage est aussi amplement utilisée dans la structure : il était possible de factoriser du code pour certaines classes, mais au contraire certains aspects étaient très spécifiques à une classe. Par exemple, les opérations binaires possède toutes une opérande droite et gauche, mais le symbole de l'opération est différent pour chacune.

Ainsi, nous avons implémenté de nouvelles classes en s'inspirant du code déjà existant, en essayant de créer une structure souple si une personne souhaite enrichir notre compilateur, et en essayant de factoriser au maximum le code pour éviter des opérations redondantes. Ainsi, dans la section suivante, nous allons présenter l'architecture des nouvelles classes créées, avant de voir plus en détaille leurs utilitées par la suite.

Architecture des dépendances (classes ajoutées).




Tree
Extends
Extends
Extends
Extends
AbstractDeclClass
verifyClass(...)
verifyClassMembers(...)
verifyClassBody(...)
AbstractDeclField
verifyDeclFieldHeader(...)
verifyDeclFieldInit(...)
AbstractDeclMethod
verifyDeclMethodHeader(...)
verifyDeclMethodBody(...)
AbstractDeclParam
verifyDeclParam(...) : Type
Extends
0..n
Relation
Extends
0..n
Relation
Extends
0..n
Relation
Extends
DeclClass
- className : AbstractIdentifier
- superName : AbstractIdentifier
- fields : ListDeclField
- methods : ListDeclMethod
DeclField
- privacy : Visibility
- type : AbstractIdentifier
- fieldName : AbstractIdentifier
- initialization : AbstractInitialization
DeclMethod
- type:AbstractIdentifier
- methodName:AbstractIdentifier
- params:ListDeclParam
- body:AbstractBody
- env_exp_body:EnvironmentExp
- isAsm : boolean
-portionAssembleur:StringLiteral
DeclParam
- type : AbstractIdentifier
- name : AbstractIdentifier
+ getName():AbstractIdentifier
+ getType() : AbstractIdentifier
1
1
1
ListDeclField
+ verifyListFieldHeader(...)
+ verifyListFieldInit(...)
ListDeclMethod
+ verifyListMethodHeader(...)
+ verifyListMethodBody(...)
ListDeclParam
+ verifyListParam(...) : Signature
Extends
Extends
Extends
TreeList



Présentation de classes ajoutées.

EnvironmentType.

Au cours de l'implémentation du compilateur, nous avons trouvé nécessaire d'implémenter un environnement pour stocker l'ensemble des types avec leurs définitions. Ainsi, nous avons créé cette classe, qui contient une table de hashage de symboles associés à un type, puis une table de hashage de types associés à leur définition.

Cette environnement se devait d'être accessible depuis de

DecacCompiler puisque celui ci était amené à vérifier contextuellement que le symbol correspond bien à un type. Nous avons alors créé un attribut env_type dans le DecacCompiler. De plus, certains types sont définis de base dans le langage Deca comme int, float, boolean et Object. Ainsi, le constructeur de EnvironmentType créer cet environnement par défaut. Nous avons aussi ajouté un attribut object dans le EnvironmentType, qui est un AbstractIdentifier, permettant de récupérer facilement la super classe Object à travers la méthode getObject() dans le DecacCompiler..

Type.

Pour factoriser le code, nous avons créé 2 nouvelles méthodes dans la classe Type :

- boolean compatibleTo(Type otherType) : cette méthode compare 2 types et renvoie true s'il est possible d'affecter otherType à this, false sinon.
- boolean castCompatibleTo(Type otherType) : cette méthode compare 2 types et renvoie true s'il est possible de caster this en otherType, false sinon.

Ces méthodes sont utilisées lors de l'étape de vérification contextuelle.

- AbstractReturn / Return :

Cette classe hérite de AbstractInst. La classe return possède un seul attribut : returnExpr. Elle possède aussi une méthode verifyReturn appelée par verifyInst, qui s'ocupe de vérifier l'expression avec verifyExpr et de vérifier que le type de l'expression est bien compatible avec le type de retour.

AbstractMethodCall / MethodCall.

Cette classe représente l'appelle d'une méthode. Elle possède en attribut :

- objectName : AbstractIdentifier → accès vers l'object à partir duquel on appelle la méthode. Si une méthode est appelée sans objet devant, alors objectName vaut null.
- methodName : AbstractIdentifier → accès vers le nom de la méthode.
- arguments : ListExpr → liste d'expressions passés en argument à la méthode.

AbstractFieldCall / FieldCall.

Cette classe représente l'appelle d'un attribut d'un objet. Elle possède 2 attributs, objectName et fieldName, tous deux de type AbstractIdentifier. Contrairement à methodCall, objectName ne peut pas être null. On peut cependant appeler des attributs sans préciser de this devant, l'attribut sera alors simplement interprété comme un Identifier, et le chaînage des différents environnements permet de retrouver la définition de cette identifieur.

AbstractDeclField / DeclField.

Cette classe ressemble à la déclaration d'une variable : elle possède en attribut un type (type : AbstractIdentifier), un nom (fieldName : AbstractIdentifier), une initialisation (initialization : AbstractInitialization), mais possède en plus un attribut indiquant sa visibilité (privacy : Visibility).

Concernant les méthodes de cette classe, verifyDeclFieldHeader et verifyDeclFieldInit seront décrites plus loin dans la passe 2 et 3.

AbstractDeclMethod / DeclMethod.

Cette classe possède comme attribut un type (type : AbstractIdentifier), un nom (methodName : AbstractIdentifier), mais aussi une liste de paramètres (params : ListDeclParam), un corps de

méthode (body : AbstractBody), un environnement de variables (env_exp_body : EnvironmentExp), et pour prendre en compte si la méthode est ASM, cette classe possède un boolean isAsm, et un StringLiteral portionAssembleur, représentant le code assembleur à générer. Concernant l'environnement env_exp_body, celui ci possède pour parent l'environnement de la classe mère, et contient initialement l'ensemble des paramètres de la méthode. Ce choix est expliqué dans la section suivante "Choix des environnements d'expressions".

Concernant les méthodes de cette classe pour la vérification contextuelle, elles sont détaillées dans la passe 2 et 3.

AbstractDeclParam / DeclParam.

Cette classe possède comme attribut un type (type : AbstractIdentifier) et un nom(name : AbstractIdentifier). Elle est utilisée pour la déclaration de paramètre lors d'une déclaration de méthode.

Le méthodes de cette classe pour la vérification contextuelle sont détaillées dans la passe 2 et 3.

ListDeclField / ListDeclMethod / ListDeclParam.

Ces classes représentent simplement des listes de différentes classes et s'inspirent de ListDeclClass. Elles possèdent une ou 2 méthodes pour la vérification contextuelle pour la passe 2 ou la passe 3, qui sont détaillées dans ces parties.

Chaînage des environnements d'expressions.

Lors de la compilation d'un programme en deca, si nous cherchons à accéder à la valeur d'une variable dans une méthode d'une

classe, cette variable peut être définie à plusieurs niveaux. Elle peut être définie dans la méthode, soit à travers une liste de déclaration de variables locales, soit à travers les paramètres de la méthode. Mais cette variable peut aussi être un attribut de la classe dans laquelle la méthode se trouve, ou bien un attribut d'une classe mère. On voit alors la nécessité de créer différents niveaux d'environnement et de les chaîner entre eux. Ainsi, dans cette partie, nous allons voir comment ont été chaîné les environnements d'expressions.

Tout d'abord, un environnement d'expressions est représenté par la classe EnvironmentExp, et contient 2 attributs:

parentEnvironment : EnvironmentExp → Environnement parent.

variables : HashMap<Symbol, ExpDefinition> → Ensemble des variables ou méthodes définies.

Ainsi, lorsqu'on cherche une variable dans un environnement, on regarde si la table de hashage contient le symbole recherché, et si ce n'est pas le cas, répète cette opération sur les environnements parents. Avec cette structure de données, nous définissons un environnement initialement vide pour le programme principale. Celui-ci est potentiellement rempli par la liste des déclarations de variables. Concernant les classes, chaque classe possède dans sa définition un environnement d'expressions, qui contient l'ensemble des attributs et méthodes déclarés dans cette classe. Son environnement parent est celui de sa classe mère, ce qui lui permet d'accéder à l'ensemble des attributs et méthodes déclarés dans une de ses classes mère. Par défaut, la classe Object est créée, et possède dans son environnement une seule méthode "equals". Ainsi, toute classe hérite de cette méthode, et il est donc possible d'accéder à la définition de cette méthode en parcourant l'ensemble des environnements parents.

A chaque déclaration de méthode, un environnement d'expressions lui est associé, avec pour environnement parent celui de la classe dans laquelle la méthode est définie. Cette environnement contient initialement l'ensemble des paramètres de la méthode, et est potentiellement complété par d'autres variables définies dans la méthode. De plus, à chaque objet créé, il lui est associé l'environnement de sa classe. Ce chaînage permet ainsi de différencier "x" déclaré en paramètre d'une méthode et "x" déclaré comme attribut de la classe avec la clé "this". Il permet de déclarer des variables de même nom à différent niveaux, sans perdre la définition de l'ancienne,

tout en rendant la variable définie dans un environnement parent presque invisible.

Les différentes passes et leurs méthodes de vérification associées.

Passe 1.

Au cours de la première passe, le compilateur parcourt l'ensemble des classes et s'intéresse simplement à leur déclaration sans s'intéresser à leur contenu. Elle vérifie que le nom de chaque classe et de sa super classe sont corrects. La méthode qui vérifie ceci est la méthode void verifyClass(DecacCompiler compiler). Cette méthode définie dans la classe DeclClass vérifie que le nom de la classe n'a pas déjà été déclaré auparavant, et que sa super classe est déjà déclarée. Par défaut, si aucune super classe n'est déclarée, la classe créée hérite de Object, et cette étape est effectuée dans le Parser.

Si tout est bon la classe est déclarée dans l'envionnement des types env_types dans le DecacCompiler.

Passe 2.

Au cours de la passe 2, le compilateur parcourt chaque déclaration d'attribut ou méthode de chaque classe et vérifie que leurs déclarations sont correctes. Cette étape est vérifiée par verifyClassMembers(DecacCompiler compiler) dans DeclClass. Cette méthode appelle 2 principales méthodes: verifyListFieldHeader et verifyListMethodHeader.

VerifyListFieldHeader prend en paramètre le compilateur ainsi que la définition de la classe courante. Cette méthode s'occupe de parcourir la liste des attributs et d'appeler la méthode verifyDeclFieldHeader sur chacun des attributs de la classe. verifyDeclFieldHeader s'occupe de vérifier que le type de l'attribut est correct, que son nom n'est pas déjà

utilisé pour autre chose et si tout est bon le compilateur déclare l'attribut dans la définition de la classe. L'initialisation de l'attribut est seulement vérifié lors de la passe 3.

verifyListMethodHeader procède de la même manière que verifyListFieldHeader : il parcourt la liste des méthodes et exécute verifyDeclMethodHeader. Comme verifyDeclFieldHeader, cette méthode vérifie le type de retour de la méthode, puis vérifie l'ensemble des paramètres avec verifyListParam. Cette méthode parcourt l'ensemble des paramètres de la liste et exécute la méthode verifyDeclParam (qui vérifie le type et le nom comme pour la déclaration d'un attribut). Cette méthode s'occupe aussi d'enregistrer chaque paramètre dans l'environnement des expressions de la méthode (attribut env_exp_body de la classe DeclMethod), et renvoie la signature de la méthode. Une fois la signature de la méthode récupérée, il reste à vérifier le nom de la méthode. On vérifie alors que son nom methodName n'est pas déjà utilisé. Si c'est le cas, alors on peut déclarer la méthode. Dans le cas contraire, on récupère la définition de l'élément de même nom déjà existant. Si cette définition correspond à une méthode de même signature définie dans une classe parente, et que son type de retour est un sur-type du type de retour de la méthode actuelle, dans ce cas on redéfinit la méthode de la super classe. Sinon, on renvoie une erreur contextuelle puisque la surcharge de méthodes n'est pas possible en Deca.

Passe 3.

Au cours de la passe 3, le compilateur vérifie toutes les initialisations des attributs et tous les corps des méthodes de chaque classe, puis vérifie le programme principale avec la méthode verifyMain. verifyClassBody s'occupe d'effectuer les vérifications pour une classe. Cette méthode appelle verifyListFieldInit, qui se charge de vérifier chaque initialisation de chaque attribut de la classe, et verifyListMethodBody, qui se charge de vérifier chaque corps des méthodes de la classe.

verifyListFieldInit appelle la méthode verifyDeclFieldInit sur chaque attribut de la classe, qui s'occupe à son tour de vérifier que

l’expression de l’initialisation est correcte et que son type est compatible avec le type de l’attribut.

verifyListMethodBody appelle la méthode verifyDeclMethodBody sur chaque méthode de la classe, qui s’occupe à son tour de vérifier le corps de la méthode avec verifyBody. Cette vérification fonctionne de la même manière que verifyMain, à la différence un type de retour peut être attendu. Ainsi, il nous faut vérifier qu’à chaque return, le type de retour soit compatible avec le type de la méthode passé en argument. verifyBody prend aussi en argument un environnement de variables. Cet environnement correspond correspond à l’attribut env_exp_body de DeclMethod. Ainsi, les paramètres sont déjà déclarés dans cet environnement et on peut facilement accéder aux attributs de la classe ou aux méthodes à travers l’environnement parent comme on a vu dans la partie “Chainage des environnements d’expressions”.

verifyMain s’occupe de vérifier la liste des déclarations de variables avec verifyListDeclVariable et la liste des instanciations avec verifyListInst. verifyListDeclVariable parcourt la liste des déclarations de variables et vérifie leur type, leur nom et leur initialisation à travers la méthode verifyDeclVar. Cette méthode vérifie donc que le type est correct, que le nom de variable n’a pas déjà été déclaré dans le même environnement, que l’initialisation est correcte et que son type est compatible avec le type de la variable. Quant à verifyListInst, elle parcourt la liste des instanciations et appelle la méthode verifyInst sur chaque instanciation. Cette méthode vérifie alors que l'instanciation est correcte. Les méthodes verifyInst sont spécifique à chaque instanciation en fonction de si celle-ci est une expression, ou bien une boucle while, un return, une condition, etc. Nous ne détaillerons pas chacune de ces méthodes pour chaque classe, celle-ci sont suffisamment claires pour comprendre ce qu’elles font, il suffit de regarder dans les fichiers correspondant. On pourra également prendre pour exemple la boucle while, la méthode verifyInst vérifie que l’expression passée entre parenthèses est bien une condition avec la méthode verifyCondition, puis elle vérifie que la liste des instanciations située dans le while est correcte toujours avec la méthode verifyListInst.

Génération du code

Généralités de génération.

La classe GenCode.

Nous avons, pour la génération du code, créé une classe appelée GenCode qui joue un rôle principal pour la gestion du module de génération du code.

Cette classe s'occupe de toutes les parties redondantes dans la génération du code du programme comme la génération d'une méthode ou la création d'un objet. Elle gère aussi des fonctionnalités utilitaires comme la génération de label non utilisé, l'initialisation du programme, la terminaison du programme ou encore la gestion des variables temporaires dans les registres lors de l'évaluation d'une expression.

L'objet GenCode est unique pour la génération d'un programme, il est créé dans le constructeur de Program et est utilisé dans toutes les fonctions codegen d'une classe dérivant de AbstractInst, nous avons, pour gérer cela, modifié les arguments de toutes les fonctions codegen.

Initialisation.

Le début du code IMA est composé de 3 instructions :

- TSTO : pour tester la taille maximum de la pile
- BOV : pour rediriger vers l'erreur si la pille n'est pas assez grande
- ADDSP : pour laisser la place au variables globales et à la table des méthodes

Partant du principe que la taille de mémoire est grande de nos jour sur la plupart des machines, une valeur de sécurité est utilisée pour la place des variables globales et la taille maximale de la pile.

Terminaison.

Le code situé à la fin du programme correspond aux traitements de toutes les erreurs levée lors de l'exécution comme une division par 0. Ces traitement ont tous le même principe : Ajout du label correspondant à l'erreur, message d'erreur à l'utilisateur décrivant l'erreur en utilisant l'instruction WSTR puis enfin l'instruction ERROR pour terminer le programme avec une erreur.

Évaluation des expressions.

Évaluation générale d'une expression.

Une expression est un « morceau » de code dans le fichier deca dont l'évaluation peut être exploitée par un autre « morceau » de code. Par exemple, dans le code *method1(expr1)* l'évaluation de expr1 est exploitée par le code method1(...). Une instruction n'est pas forcement une expression, par exemple, l'instruction *print(expr2)* n'est pas exploitable par un code.

Pour pouvoir exploiter l'évaluation d'une expression, nous sommes obligés de passer par les registres. Pour contenir l'évaluation d'une expression nous avons utilisé le registre R2 universel qui est fondamental dans la génération de notre code. Ce registre contient constamment le retour (autre nom pour signifier évaluation) d'une expression. Après avoir appelé le codegenInst de l'expression, nous sommes sûr que son retour sera contenu dans le registre. Après évaluation de 1+1, 2 est contenu dans le registre R2. Nous verrons par la suite que d'autres registres peuvent être utilisés lorsqu'une évaluation nécessite de retenir des valeurs intermédiaires (par exemple l'évaluation de gauche pour une expression binaire).

Évaluation des expressions unaires.

Les expressions unaires sont très simples à évaluer, chaque expression unaire à comme code en commun d'évaluer leur unique opérande. Cela est donc fait dans le codegenInst de AbstractUnaryExpr, Chaque codegenInst des expressions appelle donc la méthode super puis réalise leur propre opération à l'aide du registre R2.

Évaluation des expressions binaires.

Tout comme les expressions unaires, l'évaluation d'une expression binaire se réalise en deux temps :

- Évaluation des deux membres de l'expression.
- Réalisation de l'opération.

Contrairement à l'expression unaire où le registre R2 peut être directement utilisé pour réaliser l'opération, une opération binaire réalise une opération sur deux résultats, donc deux registres sont nécessaires.

Encore une fois, toutes les expressions binaires ont un code en commun qu'ils appelleront à l'aide de la méthode super. Ce code en commun évalue la première expression puis stocke son résultat dans un registre temporaire puis évalue la deuxième expression et enfin garde son résultat de le registre de retour. Chaque expression binaire vont utiliser ces deux registres pour évaluer sa propre expression.

Le registre temporaire est le plus souvent le registre R3, cela ne peut cependant parfois ne pas être le cas. Certaines expressions utilisant des parenthèses nécessitent l’utilisation de plusieurs registres pour stocker les résultats temporaires. Pour cela nous avons utilisé une pile pour constamment connaître les registres déjà utilisés et ceux à utiliser, nous incrémentons donc le numéro du registre utilisé : R3 → R4 → R5 → etc. Parfois nous avons un nombre limité de registres utilisables. Pour palier à ce problème, nous utilisons l'instruction PUSH et POP lorsque le nombre maximal de registres est atteint. Cela est réalisé facilement avec notre gestionnaire de registres.

Identifier.

Les *Identifier* désignent toutes les expressions qui peuvent être représentées avec un nom (une classe, un objet, un champ, une variable locale, une variable globale). L'évaluation d'un identifieur dépend du type de celui-ci. Pour une variable, locale ou globale, ils nous suffit de mettre la valeur stockée dans son adresse (operand) dans le registre de retour R2. S'il s'agit d'un champ (non précédé de préfixe), il suffit de récupérer l'objet qui est toujours stocké dans -2(LB) à l'intérieur d'une méthode.

FieldCall.

FieldCall est l'objet généré dans l'arbre lorsque que le champ est précédé d'un préfixe (représenté par une expression retournant l'adresse d'un objet). Nous devons distinguer ce cas car l'adresse de l'objet ne sera pas forcement dans -2(LB). Pour récupérer l'adresse de l'objet il suffit d'évaluer l'expression constituant le préfixe.

MethodCall.

MethodCall est assez similaire à FieldCall, cependant à la place de directement obtenir une valeur à placer dans R2, nous devons sauter à une adresse représentant la méthode. Toutes les opérations à gérer pour la méthode (allocation des variables globales, sauvegarde des registres) sont faites après le saut pour la méthode, il nous suffit juste de mettre les paramètres de la méthode dans la pile, le reste des opérations est assez similaire à FieldCall.

Gestions des expressions flottantes.

Certaines expressions renvoient des variables flottantes. Nous devons distinguer ce cas car le codage pour un int et un flottant est

différente et ces deux valeurs peuvent être utilisées pour un affichage avec WINT et WFLOAT. Pour gérer cela nous utilisons les fonctions *setExprFloat* et *isExprFloat,* nous mettons une expression à true lorsque celle-ci utilise des opérations mettant le retour de l'évaluation en float comme par exemple une conversion en floattant avec cast ou convFloat. Une expression est remise à false lorsque l'on passe à une nouvelle instruction.

Structures If et while.

If.

Pour la génération d'une structure if nous suivons le modèle mis sur le polycopié. Plusieurs if peuvent bien sûr intervenir dans le programme deca, pour éviter de générer plusieurs label de même nom, ce qui renvoie une erreur, nous utilisons une fonction utilitaire newLabel permettant d'obtenir un nom de label unique.

While.

Il s'agit de la même chose que la structure if, sauf que la structure est différente et utilise plus de labels. Encore une fois nous utilisons la méthode newLabel.

Gestion des classes.

Génération de la table des méthodes.

La table des méthodes est réalisée juste après l'initialisation du programme. Il suffit pour chaque méthode de récupérer leur label et le stocker dans la mémoire globale (registre GB + offset) les labels ont toujours le même modèle : *nomClasse.nomMethode,* ils sont gérés à l'aide de fonctions utilitaires dans la classe GenCode.

Nous réalisons automatiquement (sans aucun élément dans l'arbre) la table de la classe Object qui ne contient qu'une seule méthode : *equals*. Comme dans l'exemple du sujet, chaque table de chaque classe remet les fonctions de la classe parente, l'index de chaque méthode de l'objet prend en compte toutes les méthodes des classes parentes.

Génération des méthodes.

La génération des méthodes se fait quant à elle à la fin du code IMA (juste avant la terminaison du programme). Il s'agit du corps de chacune des fonctions de chaque objet. Ces corps sont générés à l'aide de la méthode codegenListInst de la liste des instructions de la classe body. Il y a tout de même des opérations à réaliser à chaque fois :

- Incrémenter la pile
- Sauvegarder les registres
- Allocations des variables locales

Pour la sauvegarde des registres il y a aucun moyen de savoir qu'elles sont les registres utilisés comme la méthode peut être appelée à n'importe quel endroit dans le programme. Nous avons donc décidé par précaution, et pour être sûr de ne jamais avoir de problème, de sauvegarder tous les registres utilisables.

Création d'un nouvel objet.

La création d'un nouvel objet se fait par l'insertion de l'instruction NEW dans le programme. La taille à allouer est de 1 + le nombre d'attributs de l'objet. Le premier espace dans la mémoire est réservée pour contenir un pointeur vers l'index de la table des méthodes.

Nous ne gérons pas la libération d'objet avec l'instruction DELETE. Il n'y a pas de free comme dans le langage C donc cela devrait être géré avec un ramasse miettes de la même manière que Java. Cela étant très complexe à réaliser (un objet créé dans une méthode peut parfois être utilisé à l’extérieur de cette méthode), nous avons décidé de ne

pas l'implémenter.

Cast et InstanceOf.

Pour la réalisation de InstanceOf, il suffit de recupérer l'index de la table des méthodes des objets, le premier élément de cette index contient un pointeur vers l'index de la table des méthodes de la classe parente. Nous pouvons donc parcourir tous ces index et vérifier s'il correspond à la valeur de la classe recherchée. Si on arrive à null (valeur dans la table de Objet) alors cet objet n'est pas une instance de la classe.

Pour le cast entre classes cela correspond à peu près à la même chose, il n'y a en réalité aucune opération à réaliser pour la conversion, l'adresse est la même quelque soit la classe et elle appellera les mêmes méthodes. La seule opération de l'instruction est de vérifier que l'objet à convertir est une instance de la classe de conversion. Nous ne gérons pas le cas de cast d'un objet vers une classe fille, il y a des éventuels problèmes de mémoire dont la détection dépasse le cadre de ce projet.

Gestion des erreurs en runtime.

Les erreurs de runtime (qui peuvent se produire durant l'exécution du programme IMA) sont les suivantes :

- Pas de return dans une fonctions.
- Déréférencement nul
- Conversion de type impossible
- Division par zéro
- Débordement arithmétiques
- Débordement de mémoire

La génération du code qui traite ces erreurs est faite à la fin du programme à l'aide de la fonction *terminateProgram*.

Absence de return dans une fonction.

Chaque méthode possède un label permettant de connaitre la fin de la méthode. Les méthodes possédant un return ajoutent dans leur corps un code levant une erreur juste avant le label de fin de la méthode. Ainsi, s'il n'y a pas de saut vers ce label (qui apparaît lors d'un return) alors l'erreur est levée.

Déréférencement nul.

Le déréférencement nul intervient lorsque l'on veut accéder à un objet qui n'existe pas, cette erreur est facilement détectée en comparant l'objet au pointeur null lors de l'appel d'un champ ou d'une méthode.

Conversion de type impossible.

Pour les types qui ne sont pas une classe, seul deux types de conversions sont possibles : int vers float et float vers int. Pour cela la vérification est très simple à faire.

Pour la conversion d'une classe à une autre l'erreur est levée si la classe de conversion n'est pas une classe d'instance de la classe de l'objet à convertir.

Division par zéro.

La division par 0 est bien évidemment une erreur en runtime car nous pourrions avoir *1/readInt()* qui est syntaxiquement correcte. Pour cela nous appelons, de la même manière que pour un if, les instructions CMP et BEQ. Cela va de même pour l'opération modulo.

Débordement arithmétique.

La détection de ces erreurs se fait simplement, lorsque l'expression binaire est flottante, par l'insertion de l'instruction BOV à la fin de l'évaluation de l'expression.

Débordement de mémoire.

Ces cas sont bien sûr une erreur à détecter à l'exécution car il est impossible de détecter ce genre d'erreur durant l'analyse (sauf un analyseur très complexe). Pour connaître la taille maximal de la pile nous devons déjà réaliser la génération de tout le code et ensuite ajouter les instructions TSTO et BOV à la fin de la génération à l'aide des fonctions *addFirst*.

C'est la raison pour laquelle la méthode *InitProgram* est appelée dans la fonction *terminateProgram*, ce qui peut sembler au premier abord contre-intuitif.