# Projet Génie Logiciel :

# Document de conception

## Groupe 7 Equipe 34

BESSET Noé
FALL Rokhaya Yvette
GILLOT Aurélie
LESPINE Marilou
ROCHE Faustine

24 janvier 2024

# Tables des matières

# Introduction

Ce document de conception vise un public de développeurs souhaitant maintenir ou faire évoluer le compilateur Deca que nous avons implémenté lors du projet Génie Logiciel. Nous décrirons l'organisation générale de l'implémentation, en passant par l'architecture, les spécifications sur le code, ainsi que la description des algorithmes et des structures de données employées.

Nous n'expliquerons pas à nouveau les aspects décrits dans le polycopié du projet, mais seulement nos implémentations supplémentaires.

La compilation d'un fichier .deca se réalise en 3 étapes qu'on nommera A, B et C : analyse syntaxique, analyse contextuelle et génération de code assembleur. Ce document comportera 3 parties détaillant les implémentations apportées à l'architecture initiale du projet.

# 1 Etape A

## a Lexer

Le lexer nous permet de reconnaître les caractères mis en paramètre et les renvoyer en une liste de token qui sera donnée au parser afin de construire l'arbre si le programme envoyé est syntaxiquement correct. Cela est fait par le biais de règles.

Les règles sont les suivantes :

- **Les mots réservés :** ce sont tous les mots qui correspondent à une instruction ou autre qui doivent être reconnus et ne peuvent pas être un nom de variable. Nous n'avons pas fait de mot "elsif" car dans le parser un elsif est reconnu en mettant à la suite un "else" puis un "if". Nous détaillerons cela plus tard dans le document. Les tests correspondants sont les tests valid et invalid nommés mots_reserves.deca.

- **Les identificateurs :** cela regroupe tous les noms de variables, de méthodes, de classes etc… Elle est réalisée en créant deux fragments qui sont les LETTER, les lettres en majuscules ou minuscules et les DIGIT qui sont les chiffres de 0 à 9. Nous testons cela dans les fichiers identificateurs.deca.

- **Les symboles spéciaux :** ce sont tous les caractères ou suite de caractère autres que les lettres ou les chiffres qui représentent quelque chose dans le langage deca. Cela peut être des caractères de comparaison comme ">" ou des parenthèses etc… Dans le langage deca, le and et le or se notent respectivement "&&" et "||". Ils sont donc répertoriés ici et non dans les mots réservés. Les fichiers de tests correspondants s'appellent symboles_speciaux.deca.

- **Les littéraux entiers :** les INT. La règle est créée grâce aux fragments DIGIT (créé précédemment) et POSITIVE_DIGIT qui sont les chiffres de 1 à 9. Cela permet d'exclure les int avec des 0 devant. Les tests litt_entiers.deca permettent de s'en assurer.

- **Les littéraux flottants :** ils peuvent être sous forme décimale ou hexadécimale. Il est aussi possible d'en trouver avec un exposant. La forme hexadécimale permet de trouver des chiffres sous forme de lettre minuscule ou majuscule mais n'est reconnaissable seulement si il y a l'expression "x0" ou "X0" au début. Nous testons cela avec les tests litt_flottants.deca. Cela nous donne la règle des FLOAT.

- **Les chaînes de caractère :** Nous définissons ici 3 règles. La première est les fin de ligne EOL qui regroupent la fin de ligne "\n" et le retour chariot "\r". Il est bien spécifié dans le lexer qu'il ne faut pas les prendre en compte. La seconde utilise un

fragment définissant les caractères de la chaîne soit tout sauf les fins de ligne, le "\" et les guillemets. Cela permet ainsi de créer la règle STRING qui met entre guillemets des STRING ou le "\\" ou le guillemet précédé d'un \. La dernière règle est celle du MULTI_LINE_STRING qui rajoute en plus le retour à la ligne. Cela est testé dans le fichier string.deca.

- **Les commentaires :** La première règle s'appelle COMMENT et reconnaît toutes les suites de caractères entourées de /*...*/. La seconde MONO_LIGN_COMMENT fonctionne sur le même principe mais il faut que la suite soit entourée des caractères // et EOL. Comme pour la règle EOL, il est spécifié qu'il ne faut pas les prendre en compte. Les fichiers commentaires.deca testent ces règles.

- **Les séparateurs :** regroupent les EOL, les caractères espace et les tab. Ils ne sont pas non plus pris en compte et sont testés avec les fichiers separateurs.deca.

- **L'inclusion de fichiers :** La règle INCLUDE utilise un fragment FILENAME qui permet de reconnaître le contenu d'un fichier. Cette règle utilise la méthode doInclude pour utiliser le texte du fichier reconnu. Les tests fichier.deca permettent de s'en assurer.

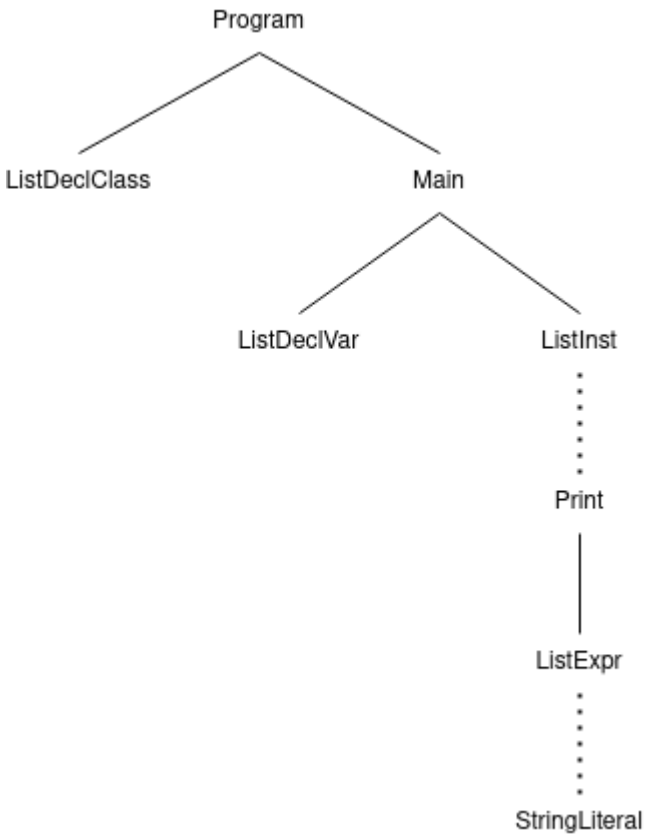
## b Parser

Nous avons divisé l'implémentation du parser en 4 sprints avec un langage de plus en plus complet à chaque fois.
Nous allons détailler l'élaboration du parser selon les différentes étapes.

### I. L'étape *Hello World !*

Le premier objectif est un parser qui crée un arbre syntaxique correct pour un programme affichant simplement le message "Hello World !".

Voici ci dessous l'arbre syntaxique souhaité :

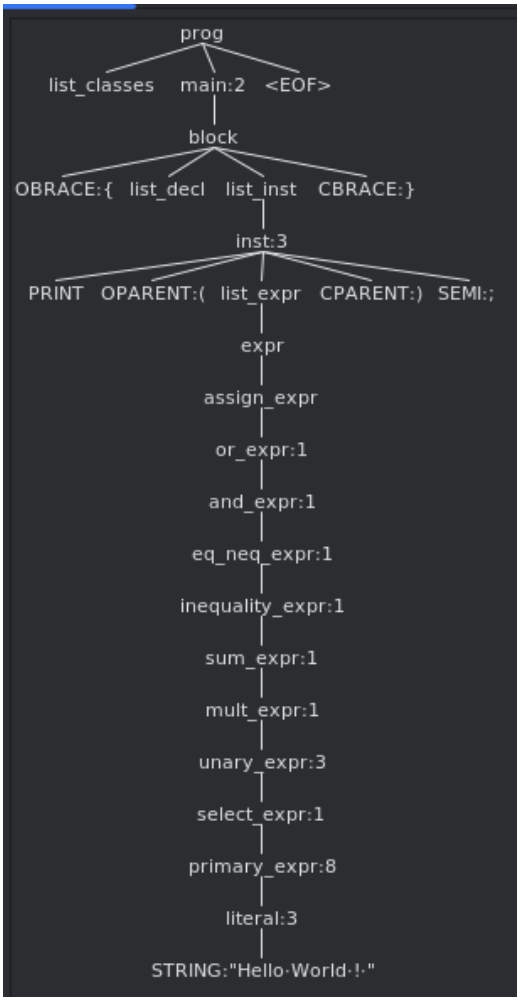



Les règles complétées sont les suivantes :

1. **prog :** création du nouveau nœud Program.
2. **list_classes :** ici, il n'y a pas de déclaration de classe, donc on crée un noeud ListDeclClass et on ne va pas plus loin.
3. **main :** création d'un nouveau nœud Main et stocke sa localisation sur le premier jeton.
4. **block :** vérifie si c'est de la forme *{code}*.
5. **list_decl :** crée un noeud ListDeclVar(), comme celle-ci est vide, on ne va pas plus loin.
6. **list_inst :** crée un noeud ListIns et ajoute les instructions de la liste dans l'arbre.
7. **inst :** vérifie qu'on est bien de la forme *print(list_expr)*. Selon le print demandé entre print, println, printx et printlnx, on crée un nœud Print ou Println en ajustant les arguments selon le besoin. Puis, on localise le nœud formé sur le token PRINT, PRINTLN, PRINTX ou PRINTLNX.
8. **list_expr :** crée un noeud ListExpr et ajoute à cette liste l'ensemble des expressions qui sont contenues dans le print.
9. **expr :** attribue la location au premier caractère de l'expression et fait rentrer dans la règle assign_expr.
10. **assign_expr :** donne une erreur si l'expression n'est pas une LValue. Sinon, fait rentrer dans la règle or_expr.

11. **or_expr :** fait rentrer dans la règle and_expr.
12. **and_expr :** fait rentrer dans la règle eq_neq_expr.
13. **eq_neq_expr :** fait rentrer dans la règle inequality_expr.
14. **inequality_expr :** fait rentrer dans la règle sum_expr.
15. **sum_expr :** fait rentrer dans la règle mult_expr.
16. **mult_expr :** fait rentrer dans la règle unary_expr.
17. **unary_expr :** fait rentrer dans la règle select_expr.
18. **select_expr :** fait rentrer dans la règle primary_expr.
19. **primary_expr :** fait rentrer dans la règle literal.
20. **literal :** reconnaît la chaîne de caractère, crée un nouveau noeud StringLiteral avec comme argument le contenu de la chaîne et met la location du noeud au $STRING.

Voici l'arbre obtenu grâce au parser :

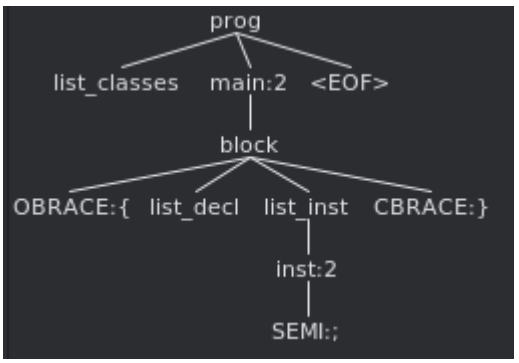


## II. L'étape sans objet :

Pour cette étape, nous avons besoin d'implémenter dans le parser toutes les instructions, les littéraux et les déclarations de variables à l'exception de l'instruction *instance of* et *return* et les littéraux *null* et *this*.

Les instructions complétées sont les suivantes :

1. **inst :** pour le SEMI, on crée un nouveau noeud NoOperation et on localise le noeud au niveau du token SEMI. Le test semi.deca nous donne cet arbre :




   Si on souhaite faire un if, then ou else, on fait rentrer dans la règle if_then_else.
   Pour le while, on crée un nouveau noeud While où on met comme argument la condition et le corps du while. On fait enfin un set location sur le WHILE.
   Pour le return, idem avec l'expression que l'on souhaite return.

2. **if_then_else** : On complète la règle de telle façon que dans le cas d'un if … else ou dans le cas où il y aurait un ou plusieurs if else, on crée dans tous les cas une nouvelle liste d'instructions newElse que l'on va mettre en argument dans la création d'un noeud IfThenElse. On crée cela afin que dans tous les cas ce soit de la même forme. On gère bien les localisations à chaque cas. Voici l'arbre rendu par le test ifthenelse-simple.deca.

3. **assign_expr :** on crée un nouveau noeud Assign.

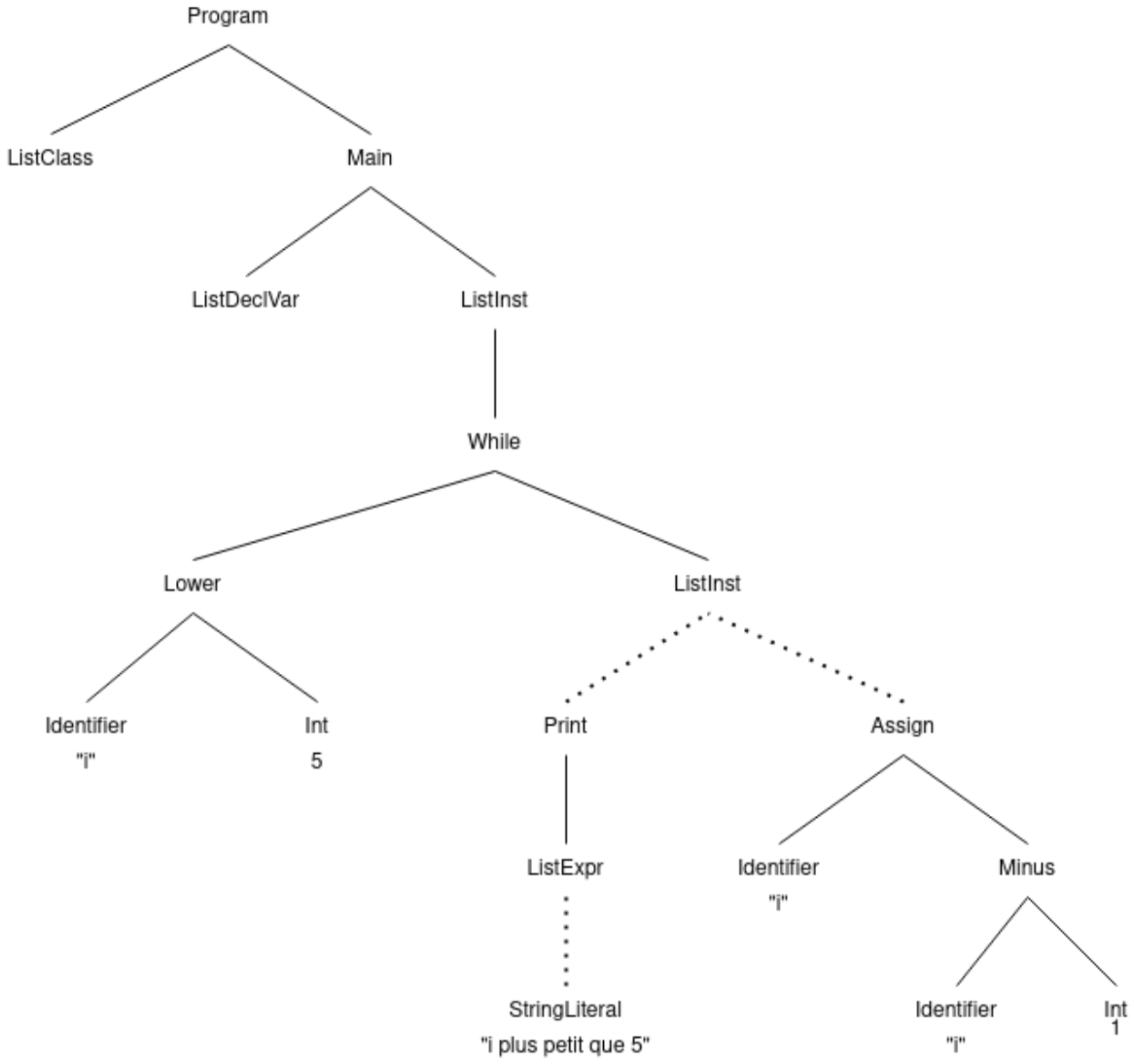
4. **or_expr :** idem que assign, on crée un nouveau noeud Or.

5. **and_expr :** idem que assign et or, on crée un nouveau noeud And.

6. **equ_neq_expr :** si on tombe sur un test d'égalité, on crée un noeud Equals, si on tombe sur un test d'inégalité, on crée un noeud NotEquals. Les expressions à gauche peuvent être à nouveau des eq_neq_expr, alors que les expressions à droite sont des inequality_expr.

Voici un exemple d'arbre qui utilise un while, un print, et un lower avec le programme suivant :

```
{
        while (i<5) {
                print("i plus petit que 5");
                i = i - 1;
                }
}
```




7. **inequality_expr :** cette règle concerne les inégalités <, <=, > et >=. Dans chaque cas, on crée un noeud correspondant (Lower, LowerOrEqual, Greater ou GreaterOrEqual). Comme dans la règle précédente, l'expression de gauche est une inequality_expr et celle de droite est une sum_expr.

8. **sum_expr :** on traite ici les + et les -, avec comme expression de droite une mult_expr. On crée le noeud Plus ou Minus.

9. **mult_expr :** idem que pour sum_expr mais avec les noeuds Multiply, Divide et Modulo. L'expression de droite est une unary_expr.

10. **unary_expr :** cette règle traite les cas où l'on a soit un moins devant une expression soit une négation. Elle crée les noeuds UnaryMinus et Not.

11. **primary_expr :** on traite ici plusieurs cas différents. Tout d'abord le cas où il y a des parenthèses autour d'une expression. On ignore alors ces parenthèses. Il y a aussi le cas des readInt ou readFloat, où on crée un nouveau noeud correspondant. Enfin, le cas où l'expression est un littéral et fait rentrer dans la règle correspondante.

12. **literal :** Un noeud est créé un fonction du littéral rencontré, si c'est un int, un float ou un booléen. Pour les int et les float, il a fallu prévoir une erreur dans le cas où on rencontre des valeurs trop grandes et donc non codables sur 32 bits, l'overflow.

13. **ident :** Enfin, ident crée un symbole contenant le texte qui est dans le ident rencontré et créé un nouveau noeud Identifier.

## III. L'étape avec objet :

Pour l'étape du langage avec objet, il eut fallu créer dans le parser de quoi déclarer des variables et implémenter tout ce qui concerne la création des classes. Dans le cadre du projet, c'est surtout dans ce sprint que nous devions créer des nouvelles classes java, lesquelles étaient déjà fournies pour les étapes précédentes.

1. **déclarations de variables :** on crée d'abord la liste de déclarations de variables et on y ajoute les variables (dans les règles list_decl, decl_var_set et list_decl_var en créant les noeuds correspondants). Ensuite, la règle decl_var créé la déclaration de variable selon si elle se fait avec une initialisation ou non. On reconnaît ainsi avec le parser un ident (le nom de la variable) avec ou non = *expr* que l'on traite ensuite comme une expression (voir l'étape sans objet).
   On complète aussi la règle select_expr qui reconnait le cas où on aurait *expr.att* ou *expr.Method(args)*. Dans le premier cas, on crée un noeud Dot, dans le second on crée un noeud MethodCall.
   De même, on complète la règle primary_expr en ajoutant le noeud New. Dans le cas de deca, on ne traite pas les constructeurs qui prennent des arguments.
   Pour le type, il est traité comme un ident et est simplement ajouté à l'arbre.
   Dans la règle literal, nous devons créer des noeuds pour le THIS et le NULL.

2. **création de classes :** le parser implémenté reconnaît une liste de classes à créer. Pour chaque classe, il reconnaît son nom, si elle est une extension d'une autre classe et son corps. Il crée un noeud ClassBody qui reconnaît et sépare les champs et les méthodes de cette classe. Dans le cas où la classe est fille d'une autre classe, il crée un Identifier pour le nom de cette superclasse. Sinon, il fait de même mais avec la classe Object.
   Dans le cas des champs, le parser gère la visibilité du champ, son type et la liste de déclaration de champs. La visibilité n'est pas gérée par une classe java mais par un enum, qui contient les champs PUBLIC et PROTECTED. En deca, si le champ est public par défaut et protected si cela est précisé. Decl_field est implémenté de la même manière que decl_var.
   Pour les méthodes, il existe celles écrites en deca, et celles écrites en langage assembleur, on a alors deux types de noeud pour le corps de la méthode , MethodBody et MethodAsmBody. Pour les méthodes écrites en assembleur, on traite le code comme une chaîne de caractères ou une chaîne de caractères sur plusieurs lignes dans la règle multi_line_string.
   Dans le cas où la méthode contient des paramètres, on déclare chaque paramètre.

# IV. L'étape du langage essentiel :

Pour cette étape, il nous a fallu créer dans la règle inequality_expr le noeud InstanceOf qui prends en arguments une inequality_expr à gauche et un type à droite. Enfin, nous avons comme primary_expr le cas du cast qui reconnaît un expression de la forme *(type) (expr)*. Nous créons un nœud correspondant avec comme argument le type et l'expr.

Grâce à ces 4 sprint, nous avons un parser et un lexer fonctionnels afin de compiler le langage deca.

# 2 Etape B

Lors de la compilation, le fichier est d'abord syntaxiquement vérifié grâce aux Lexer et Parser implémentés. Si le programme deca est syntaxiquement correct, alors un arbre de syntaxe abstraite est généré.

Cet arbre est ensuite "décoré" lors de l'étape B, qui est l'étape de vérification contextuelle : si le programme est mal typé ou mal supporté au sens du contexte, alors une erreur sera levée, qui décrira exactement le problème trouvé dans le programme erroné.

Décrivons maintenant l'organisation de cette étape de vérification contextuelle. En annexe 1, on retrouve un arbre représentant les classes utiles à la vérification contextuelle d'un bloc (Main et corps des méthodes). L'annexe 2 schématise les hérédités des fichiers qui vérifient les classes.

Le vérification contextuelle se fait par 3 passes consécutives sur le programme. La première passe consiste à vérifier uniquement le nom des classes et la hiérarchie des classes. Lors de la deuxième passe, on vérifie les déclarations des champs et la signature des méthodes. Les méthodes pouvant être mutuellement récursives, on vérifie le corps des méthodes au cours d'une troisième passe, ainsi que le bloc Main du programme.

Les méthodes appelées par l'étape B sont de la forme verify…(), et correspondent aux règles de la syntaxe contextuelle. On les retrouve dans les fichiers .java portant le nom du non-terminal associée à cette règle. Chacune vérifie les conditions de sur les attributs héritées, et réalise les affectations des attributs synthétisés.

## Passe 1

La passe 1 consiste simplement à ajouter les classes à l'environnement des types, leur définition étant complétée à la passe 2.

Pour chaque classe de ListDeclClass, on commence par générer un environnement qui correspond à une liste chaînée de dictionnaires (identificateur, définition). Un identifiant peut être un nom de variable, de méthodes, de type, etc…, et chaque environnement pointe sur un environnement "parent" qui représente la portée englobante ou le "super-bloc". Cela permet de prendre en compte la portée des blocs imbriqués, par exemple, dans le contexte des classes et des superclasses.

Avant d'ajouter le nouveau type, on vérifie que la superclasse est déjà définie dans l'environnement des types. Si ce n'est pas le cas, une erreur contextuelle sera levée. Dans le cas contraire, nous appelons la méthode declareClass() ajoutée dans EnvironmentType, qui vérifie que le type ne soit pas déjà déclaré.

## Passe 2

Lors de la seconde passe, nous ajoutons les champs et méthodes aux définitions de chaque classe. Ceux-ci seront ajoutés à l'environnement members de la définition de la classe.

Dans la méthode verifyCassMembers(), nous appelons la méthode verifyClassBody() de la classe ClassBody, qui appelle elle-même les méthodes de vérification de ListDeclField et ListDeclMethod. Tout au long de la passe 2, nous incrémenter les champs NumberOfFields et NumberOfMethods de la définition de la classe.

Nous commençons par vérifier la déclaration des champs dans DeclField (qui possède les attributs visibilité, typeField, varName et initialization). On vérifie que son type est valide et différent de void (grâce à verifyType()). Ensuite, on vérifie que les superclasses ne contiennent pas déjà de champs du même nom, auquel cas une erreur sera levée. Ainsi, nous déclarons une nouvelle définition de champ (FieldDefinition) en précisant la classe contenant le champ, sa visibilité, ainsi que son indice. Cet indice est défini lors de l'étape B et utilisé par l'étape C. La gestion de ces indices est détaillée dans le polycopié du projet, nous n'en parlerons pas en détail dans ce document.

Ensuite, la vérification des déclarations de méthodes consiste à définir leur signature. Ceci est réalisé dans la méthode verifyDeclMethod() de la classe DeclMethod. On commence par vérifier le type de retour de la fonction. Ensuite, nous appelons verifyListDeclParam() afin de vérifier le type des paramètres et donc de créer la signature de la méthode. Si tous les types sont valides, alors on vérifie la cohérence de la méthode par rapport aux méthodes des superclasses : si une méthode est déjà définie dans une superclasse, alors celle-ci doit avoir la même signature, et son type de retour doit être un sous-type de la méthode héritée. Ainsi, de la même manière que pour les champs, nous pouvons déclarer une nouvelle définition de méthode (MethodDefinition) contenant son type, sa signature et son indice.

Pour finir la seconde passe, nous empilons l'environnement de la classe courante à celui de sa superclasse, grâce à la nouvelle méthode empilement() de la classe EnvironmentExp.

## Passe 3

La dernière passe consiste à vérifier les déclarations de variables et les instructions du bloc Main et du corps des méthodes.

Pour cela, il nous est fourni la classe ListDeclVar qui hérite d'une classe TreeList grâce à laquelle nous pouvons récupérer la liste des variables déclarées.

La vérification de ces déclarations se fait dans la classe DeclVar (qui possède les attributs type, varName et initialization), en commençant par la vérification du type. Pour cela, dans notre classe Identifier (identifiant), on va vérifier si la définition du type existe bien dans l'environnement des types, et si c'est le cas on lui attribue sa définition et son type, ce qui va permettre de décorer l'arbre syntaxique.

Une déclaration de variable ou de champ peut contenir une initialisation ( int a = 1; ) ou pas ( int a; ), d'où l'existence des classes Initialization et NoInitialization qui hérite de la classe abstraite AbstractInitialization. On va donc appliquer la méthode verifyInitialization() sur l'attribut initialization, afin de vérifier l'expression du côté droit (VerifyRValue()) correspondant à l'affectation si elle existe, ou ne rien faire si elle n'existe pas.

Plusieurs classes héritent (directement ou non) de la classe AbstractExpr (une expression), et possèdent donc une méthode verifyExpr() afin de vérifier contextuellement l'expression. La classe Assign, qui représente une affectation, est une de ces classes, et donc cette méthode sera appelée lors de la vérification de l'expression du côté droit (qui est une affectation). C'est alors qu'on vérifiera dans verifyExpr() de la classe Assign l'expression à gauche de l'affectation et celle à droite de l'affectation, qui peut elle-même être une affectation par exemple. Une fois que les 2 opérandes sont vérifiées et que leurs types ont été définis, on teste si les opérandes sont "compatibles", c'est-à-dire qu'il est possible d'assigner opérande droite de type1 à opérande gauche de type2. Pour le langage sans objet, on s'assurera seulement que type1 = type2, ou que type1 est un float et type2 est un int, et pour le langage avec classes, on s'assurera que le type_class2 est un sous-type de type_class1. Si ce n'est pas le cas, alors une erreur contextuelle sera levée qui informera l'utilisateur qu'il y a une incompatibilité dans l'affectation.

Au-delà des affectations, dans les expressions à vérifier se trouvent également les opérations binaires (classe AbstractBinaryExpr), dont les opérations arithmétiques, booléennes ou de comparaison. On aura également la possibilité de vérifier l'expression d'une opération arithmétique (class AbstractOpArith) : de la même façon que pour le Assign, on récupère le type des opérandes droite et gauche dont on vérifie les expressions, puis on vérifie que le type du résultat de l'opération est en accord avec le type des opérandes. Si ce n'est pas le cas, une erreur est levée pour types invalides dans l'opération arithmétique.
Si l'opération effectuée est entre un entier et un flottant, alors l'entier sera converti en flottant à l'aide de la classe ConvFloat et des méthodes de setter et getter des opérandes.

Revenons un peu en arrière. Nous avons vérifié la liste des déclarations de variables, il faut maintenant vérifier la liste des instructions du programme.

Dans les instructions se trouvent notamment les expressions (exemple du Assign), mais aussi le Print, le Return, le While.. qui sont des classes dans lesquelles se trouvent les méthodes verifyInst(). La classe While possède les attributs condition et body. La condition étant forcément une seule expression (AbstractExpr), on la vérifie à l'aide de verifyInst() qui appelle verifyExpr() sur la condition, et on utilise la méthode verifyListInst() sur le body pour vérifier les expressions de chaque instruction à l'intérieur du while. Pour cela, nous avons une classe ListInst qui fonctionne de la même manière que ListDeclVar, c'est-à-dire qui hérite d'une classe TreeList<AbstractInst> permettant d'itérer sur les instructions de la liste. Pour chaque instruction, on vérifiera donc son expression à l'aide de verifyExpr().

La classe IfThenEsle fonctionne de la même manière que la classe While, mais elle contient un attribut supplémentaire qui correspond aux instructions contenues dans Else. Nous appelons donc verifyListInst() sur les instructions du Then et du Else.

Lorsqu'un programme .deca appelle une fonction du groupe print, un ou plusieurs paramètres lui sont donnés. Ces paramètres sont des expressions, que nous vérifierons alors avec verifyExpr(). Nous appelons cette méthode dans verifyInst() de la classe AbstractPrint. Nous vérifions que l'expression est bien un type imprimable (String, Int ou Float).

La classe Return a pour attribut l'expression qui doit être retournée. On doit donc vérifier que le type de cette expression est compatible avec le type de retour de la méthode, toujours avec verifyType().

Dans le corps d'une méthode ou dans le bloc Main, nous avons la possibilité d'appeler une méthode d'une classe créée précédemment. La vérification de cette instruction consiste alors à contrôler l'existence de cette méthode, et vérifier qu'elle soit appelée avec des paramètres compatibles avec sa signature. Ces vérifications sont réalisées dans verifyExpr() de MethodCall.

Il est aussi possible de récupérer la valeur d'un champ d'une classe en utilisant la sélection grâce au caractère ".". Nous vérifions cette expression dans la classe Selection qui a pour champ expr (partie gauche) et field (champ sélectionné). Nous vérifions alors que expr est de type Class, et que field est un est champ de cette classe. Nous vérifions aussi la visibilité du champ (Protected ou Public).

# 3 Etape C

L'étape C correspond à l'implémentation de toutes les méthodes dont le nom commence par codeGen mais également les classes que nous avons ajoutées dans le dossier codegen. Le rôle de chacune de ces méthodes est sous-entendu par leur nom, on se focalise donc sur l'implémentation des classes de codegen ainsi que leur utilisation.

Le premier point important sur lequel nous avons focalisé notre attention est la gestion des registres en créant la classe **GestionRegistre**. Nous associons un objet de cette classe à notre **DecacCompiler** pour garder l'information de l'état des registre tout le long de la génération de code. On utilise principalement dans cet objet les méthodes **getFreeRegister** qui retourne le registre libre de plus petit indice, **getOneFullRegister** qui retourne le dernier registre rempli, **getTwoFullRegister** qui retourne les deux dernier registres rempli et **newLiberateRegister** qui indique à l'objet que le registre passé en paramètre a été libéré. L'intérêt de cette classe réside dans la gestion de la pile lorsque l'ensemble des registres sont utilisés, les quatre fonctions précédentes vont en effet introduire des nouvelles instructions pour permettre la bonne utilisation de la pile. Si tous les registres sont remplis, il faut push le dernier registre pour le libérer quand le programme a besoin d'un registre libre. Et si de plus, la pile possède déjà des valeurs temporaires, lorsque le programme veut récupérer deux registre remplis, il faut pop la dernière valeur temporaire de la pile pour le déplacer dans un registre, et lorsque que l'on libère un registre il faut remplir ce registre par une valeur de la pile.

Un autre point important est la gestion des différents labels utilisés principalement pour les instructions conditionnelles. Une partie de ces labels est généré grâce à la classe **LabelGen** dont un objet est associé encore une fois au compilateur. Lorsqu'un programme est compilé, il faut que l'assembleur produit n'ai pas de doublons sur les labels sinon les branch ne peuvent pas fonctionner. L'objet de la classe **LabelGen** garde donc des conteurs pour savoir combiens il y a eu de certains types de labels depuis le début de la compilation, il les incrémentés à chaque fois que le reste du programme réclame ces labels et les labels retournés sont de la forme "While_12:" avec le numéro indiquant que ce while est 12ème, ce qui résout le problème de doublons.

Lors de la compilation notre compilateur doit ajouter la gestion des erreurs et il faut donc ajouter à la fin du code compilé la section associée au erreurs. Dans une optique d'éviter de générer du code mort, notre compilateur n'écrit dans la section des erreurs que les erreurs qui peuvent également arriver lors de l'exécution du code ima. On a ainsi créé une classe **CodeGenErrors** qui a un de ses objets associé au compiler, et qui possède des booléens qui indiquent quelles gestions d'erreurs ont besoin d'être implémentées. Par exemple, si le programme lit un entier entré par l'utilisateur, le booléen **isIoOverflow** passe à true et on affiche donc cette gestion d'erreur à la fin du programme.

Il est nécessaire tout au long de la compilation d'ajouter des contrôles de débordement de la pile, et c'est le rôle de la classe **GestionVariables** de compter les valeurs qui ont été ajoutées à la pile pour déterminer la valeur a mettre dans **TSTO**. Encore une fois, un objet de cette classe est associée au compiler et c'est à la fin de la génération du code du main, d'une initialisation d'objet ou d'une méthode que l'on exécute la méthode **addTSTO** qui permet de générer un contrôle de la pile en tête du bloc ainsi que l'ajout de **ADDSP** si nécessaire. Pour pouvoir ajouter en tête d'un bloc, nous avons ajouté une méthode **addInstructionsIndex** qui prend en deuxième argument, un index, et l'instruction passée en argument va aller se placer dans la liste d'instructions à l'index passé en paramètre. Aussi avec **addTSTO**, dans le cadre d'une méthode, on ajoute des push au début (et des pop à la fin) pour sauvegarder l'ensemble des registres qui sont utilisés par la méthode.

# Annexes

## Annexe 1 : Arbre des classes utilisées pour la vérification contextuelle du Main et corps des méthodes

Annexe 2 : Arbre des classes utilisées pour la vérification contextuelle des classes