Documentation de conception

Le compilateur Decac à votre disposition est un compilateur effectif sur un langage deca sans objet, et sur un langage essentiel avec objet, sans gérer la méthode @Equals, héritée de la superClasse Objet.

On rappelle également qu'au niveau des options de compilation, l'option -r, visant la limitation des registres banalisés disponibles, est fonctionnelle sur le langage deca sans objet, mais ne fonctionne pas lorsque l'on passe sur un langage deca avec objet.

A travers cette documentation de conception, vous avez à votre disposition une description la plus précise possible de l'organisation générale de notre implémentation. Cela se découpera en trois parties principales : d'abord une description des architectures de notre implémentation, puis la deuxième partie apporte une spécification sur le code du compilateur avant de terminer par une description des algorithmes et des structures de données utilisés.

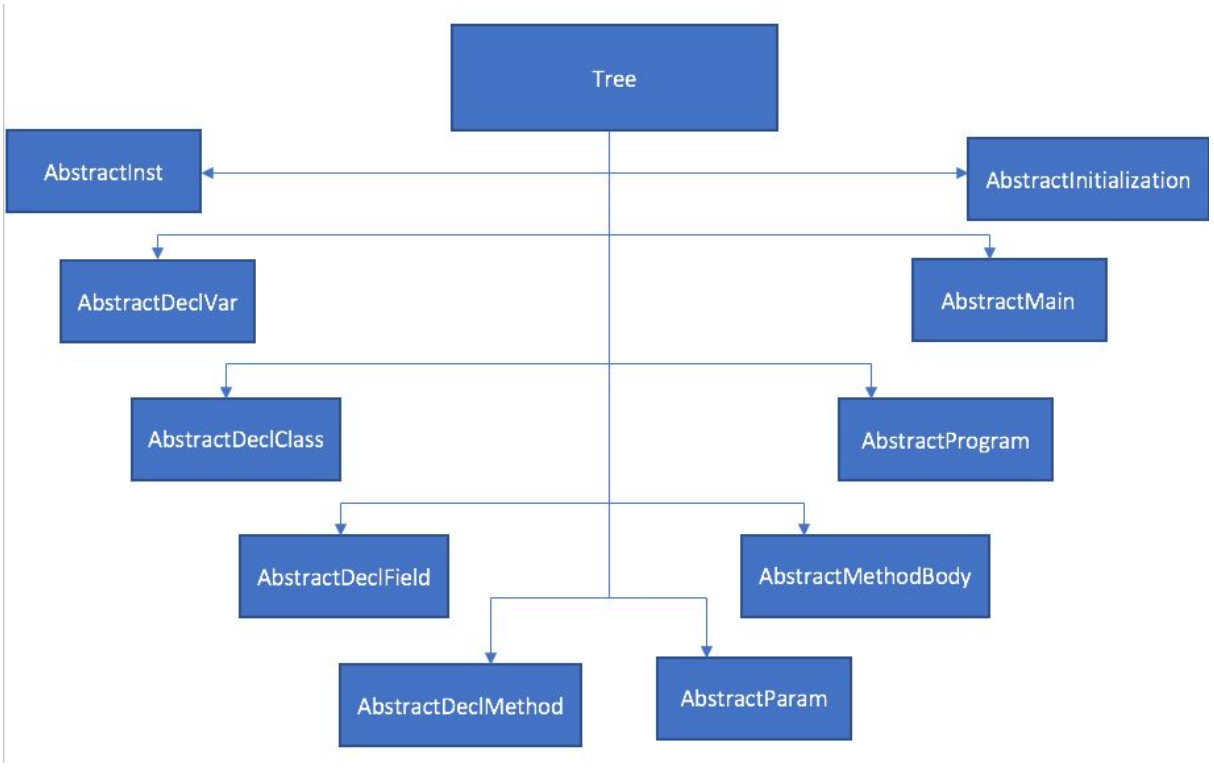
1. Les architectures du projet

La conception du projet s'est articulée autour de plusieurs portions de code. Ce qui suit est une description de cette architecture avec une liste des classes ainsi que leurs dépendances.

- Tree

Les classes utilisées pour la mise en place de l'analyse syntaxique de l'étape A héritent toutes de la classe Tree. Ce sont toutes des classes abstraites qui héritent de la classe Tree dont de nombreuses classes héritent à leur tour afin de faire un code le plus factorisé possible.

Le diagramme présent ci-dessous présente l'ensembles des classes qui héritent de la classe Tree.



Les classe héritant de AbstractDeclMethod, AbstractDeclField, AbstractParam et AbstractMethodBody étant décrites dans la partie, on ne s'y attardera pas dans cette partie.

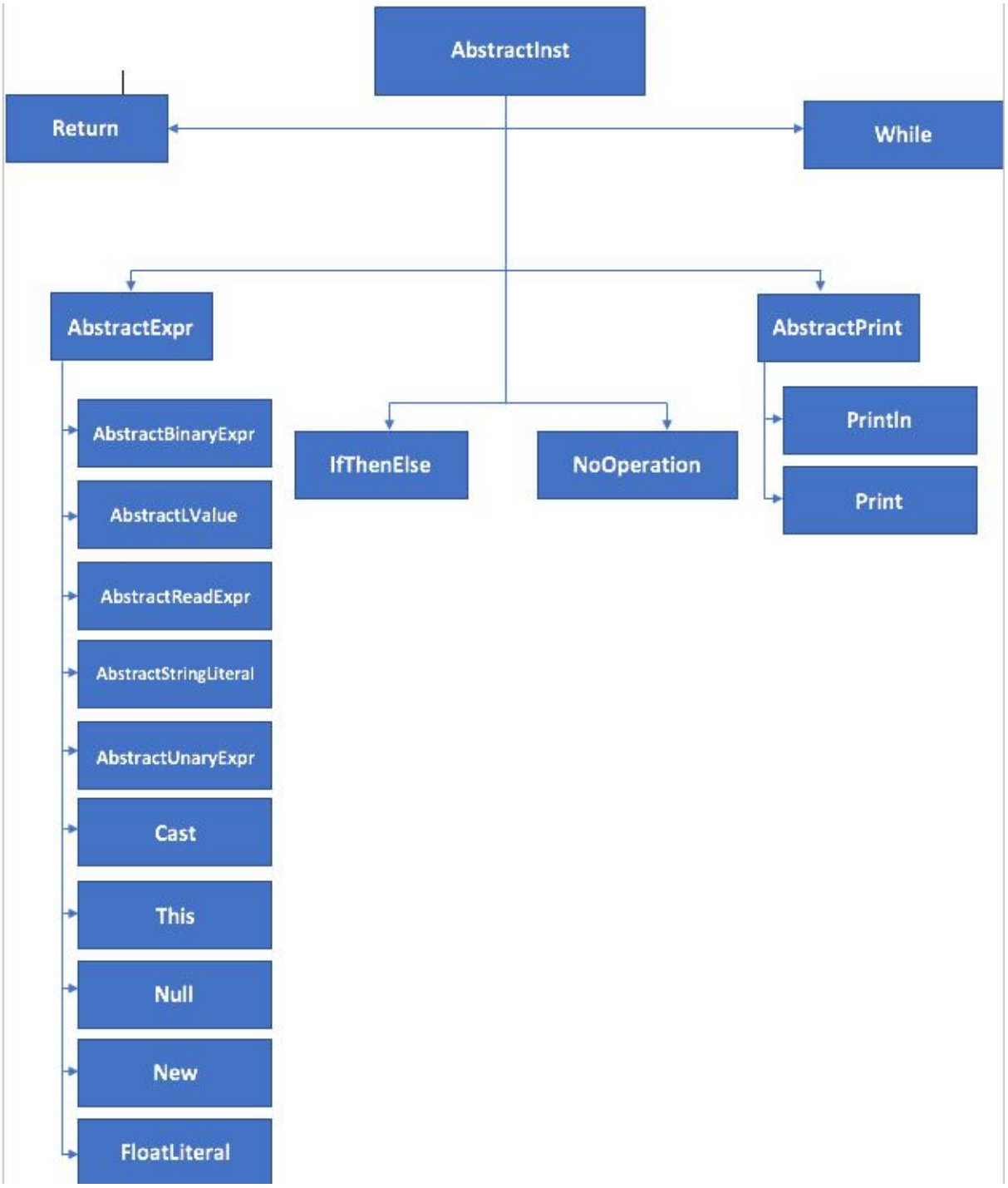
Par souci de clarté et de concision, on ne fera que citer les classes non représentées dans les deux diagrammes.

- **Initialization** et **NoInitialization** sont les deux classes qui héritent de AbstractInitialization.
- **Main** et **EmptyMain** sont les deux classes héritant de AbstractMain.
- **Program** est la classe qui hérite de AbstractProgram.
- **DeclClass** est la classe qui hérite de DeclClass.
- **DeclVar** est la classe qui hérite de AbstractDeclVar.

On va alors s'intéresser de près à la classe AbstractInst dont héritent une grande partie des classes. Les classes héritant de AbstractExpr ont également été représentées. Elles ne seront pas détaillées dans un diagramme à leur tour mais une description brève en sera faite si besoin.

- AbstractBinaryExpr regroupe toutes les opérations binaires : comparaison, égalité, arithmétique…
- AbstractLValue s'étend en principalement en AbstractIdentifier qui est utilisée pour les différents noms dans les classes

- AbstractUnaryExpr comporte toutes les opérations unaires telles que le *Not* ou encore le moins unaire.



2. Spécifications sur le code du compilateur autres que celles fournies

Nous allons présenter les architectures qui ont été ajoutées en fonction des différentes étapes du compilateur.

- Etape A

Au niveau de l'étape A, le travail réalisé au niveau de l'architecture des classes a été en grande partie une création de classes respectant l'architecture que l'on pouvait comprendre de la syntaxe abstraite présente dans la documentation fournie.

En effet, pour le langage deca sans objet, aucune classe n'avait besoin d'être rajoutée, étant donné que tout le squelette des classes était déjà présent. En revanche, en ce qui concerne le langage deca avec objet, il a fallu rajouter des classes, conformément à la syntaxe abstraite du langage Deca fournie dans la documentation du projet. Nous allons donc faire une liste exhaustive des classes qui ont dû être ajoutées.

Tout d'abord, pour permettre la déclaration d'une classe, il fallait créer les attributs, notés **field** et les **méthodes**. Ces dernières nécessitaient à leur tour la création des **paramètres** de la méthode ainsi que son corps, que l'on a appelé **body** pour respecter les notations de la grammaire.

Pour éviter que la description des méthodes des nombreuses classes qui ont été créées ne soit trop redondante, on décrira comme méthodes habituelles des classes abstraites héritant de la classe Tree les méthodes @verify{nom_de la classe} et @codeGen{nom_de_la_classe}.

- ❖ Les attributs d'une classe

Pour implémenter les attributs d'une classe, l'objectif de notre implémentation a été de respecter le plus possible l'esprit du squelette du code fourni au début du projet. Similaire à la déclaration d'une variable, la déclaration d'un attribut a nécessité la création de deux classes, **AbstractDeclField** et **DeclField**. La classe AbstractDeclField hérite de Tree, et déclare les méthodes habituelles des classes abstraites héritant de Tree en plus de la méthode @verifyInitialization. Ensuite, la classe DeclField hérite de la classe AbstractDeclField, et définit un attribut à travers sa visibilité, son type, son nom et s'il y a initialisation ou non. Enfin, vu qu'une classe

peut avoir un ou plusieurs attributs, on a créé la classe **ListDeclField** héritant de TreeList, qui est utilisée lors de la déclaration des attributs d'une classe.

- Les méthodes d'une classe

De même que pour les attributs d'une classe, il fallait créer les classes nécessaires pour permettre la déclaration de méthodes dans une classe. Pour la déclaration d'une méthode, il a encore fallu la création de deux classes, **AbstractDeclMethod** et **DeclMethod**. La classe AbstractDeclMethod hérite de Tree, et déclare les méthodes habituelles des classes abstraites héritant de Tree. Par la suite, la classe DeclMethod hérite de la classe AbstractDeclMethod, et définit une méthode à travers son type, son nom, ses paramètres et son corps. Ces deux derniers éléments ont nécessité la mise en place de classes appropriées qui seront décrites dans le reste du document. Enfin, vu qu'une classe peut posséder une ou plusieurs méthodes, on a créé la classe **ListDeclMethod** héritant de TreeList, qui est utilisée lors de la déclaration des méthodes d'une classe.

- Les paramètres d'une méthode

Lorsque l'on déclare une méthode dans une classe, cela nécessite la déclaration de paramètres, y compris lorsque ladite méthode ne prend pas de paramètre - les paramètres sont juste vides dans ce cas. Cela a donc motivé le choix de créer des classes **AbstractParam** et **DeclParam**. La classe AbstractParam hérite de Tree et déclare les méhodes habituelles des classes abstraites héritant de Tree. Encore une fois, la classe DeclParam hérite de la classe AbstractParam, et définit un paramètre à traver son type et son nom. Enfin, vu qu'une méthode peut prendre aucun, un ou plusieurs paramètres, une classe **ListDeclParam** héritant de TreeList a été créée et est utilisée lors de la déclaration d'une méthode.

- Le corps d'une méthode

Comme spécifié dans le cahier des charges, le compilateur doit reconnaître deux types de corps de méthodes : les méthodes avec un corps sous forme de *block*, qui se présente sous la forme d'instructions en deca délimitées par des accolades et les méthodes en assembleur, dont le corps se présente sous la syntaxe "asm (<instructions en langage assembleur>);". Afin de généraliser le code de la déclaration de méthodes, une première classe abstraite **AbstractMethodBody** a été créée, étendant Tree. Cette classe hérite de la classe Tree et en hérite les méthodes habituelles définies plus haut. Logiquement, une classe **MethodBody** a été créée pour les méthodes avec un corps sous forme de *block* et une classe **MethodAsmBody** a été créée pour les méthodes en langage assembleur. Ces deux classes héritent de AbstractMethodBody. La première définit la liste de variables et

la liste d'instructions de la méthode. MethodAsmBody définit un code, qui est une chaîne de caractères contenant les instructions en langage assembleur.

- Return

Nous avons spécifié précédemment qu'une méthode, lorsqu'elle est déclarée, définit entre autres un type. Lorsque ce type n'est pas *void*, il faut que la méthode retourne quelque chose du bon type. Cette fonctionnalité a été implémentée à travers la classe **Return** qui hérite de AbstractInst. Cette classe définit une expression, ce qui est retourné.

- This

Lorsque l'on déclare une classe, on doit être capable de réaliser la sélection de champs, avec un *this* apparent ou non. Cette fonctionnalité a été ajoutée grâce à la création d'une classe **This**, étendant la classe AbstractExpr. Cette classe définit un attribut booléen, signalant la présence ou nom du mot "this" dans la sélection.

- Sélection

Ainsi dans une déclaration de classe ou lorsque l'on instancie un objet appartenant à une certaine classe, il est nécessaire de pouvoir sélectionner les champs qu'on a le droit de sélectionner. Cette fonctionnalité a nécessité la création de la classe **Selection**, étendant la classe AbstractLValue. Cette classe définit un objet et un champ sélectionné.

- L'appel de méthode

Une fois qu'une méthode est déclarée au sein d'une classe, il faut permettre aux instances de cette classe de faire appel à cette méthode. Pour répondre à ce besoin, nous avons créé la classe **MethodCall**, héritant de la classe AbstractExpr. Cette classe définit l'objet qui fait appel à la méthode, le nom de la méthode appelée ainsi que la liste de paramètres dont la méthode appelée a besoin.

- New

Une fois qu'une classe a été correctement déclarée, nous devons être en mesure d'instancier de nouveaux objets appartenant à cette même classe. C'est la classe **New** étendant la classe AbstractExpr qui permet cette fonctionnalité. Cette classe définit un nom de classe, qui est le nom de la classe à laquelle appartient l'instance.

❖ Limitations et possibilités pour le futur

Ce compilateur deca fonctionnant sur un langage deca essentiel, il ne gère pas les instructions *instanceof* et *cast*. Cela n'empêche pas le fait que les classes de l'architecture gérant la partie d'analyse lexicale et syntaxique de ces deux instructions ont déjà été implémentées. La classe **Instanceof**, étendant la classe AbstractExpr, définit une expression et un type. La classe **Cast**, étendant la classe AbstractExpr, définit quant à elle un type et une expression, afin de pouvoir forcer le typage de l'expression.

- Etape B

Au sein de la partie B, une grosse partie du travail, que ce soit dans la partie objet ou la partie sans objet, a été réalisé au sein des classes déjà existantes. Nous avons cependant trouvé judicieux de rajouter deux classes : **EnvType** et **ObjectType**. Pour EnvType, c'est une classe qui , assez similairement à la classe EnvExp qui a été fournie, permet d'associer le nom d'une classe à une définition (que cela soit une TypeDefinition, ClassDefinition…). ObjectType quant à elle a été utilisée pour définir la classe, le type Object, dont héritent toutes les classes en Deca.

- Etape C

En ce qui concerne l'étape de génération de code en langage assembleur, il a fallu, en plus d'implémenter les méthodes @codeGen{nom_de_classe}, créer une grande classe **Codegen**. Cette classe permet notamment d'implémenter l'option de compilation -r qui limite le nombre de registres à l'aide d'un gestionnaire des registres qui y a été implémenté ainsi que l'option de compilation -n qui supprime les tests de débordement à l'aide de méthodes qui mettent en place les différents label d'erreur.

Cette classe Codegen comporte de nombreux attributs :

- Tous les Label d'erreur (stackOverflowError qui gère les cas de pile pleine, zeroDivisionError qui gère les division par 0 …)
- Un entier id_register permettant de savoir quel est le prochain registre à utiliser lorsque l'on va utiliser l'instruction LOAD
- Un entier nb_Register_max qui sert à mettre en place l'option de compilation -n décrite ci-dessus
- Des structures permettant de stocker les Label des différentes méthodes pour gérer l'appel des méthodes ainsi que le code des méthodes dans le fichier assembleur

- Une liste initializedVariables qui stocke les adresses dans la pile des variables non initialisées, ce qui permet de gérer les erreurs qui occurrent lorsqu'il y a une opération avec des variables non initialisées
- Des entiers id_bool, id_while, id_not, id_ifthenelse permettant de savoir l'indice du prochain label de ces branchement conditionnels

3. Description des algorithmes et structures de données employés

Les seuls algorithmes notables utilisés au cours de ce projet sont les différents algorithmes utilisés pour implémenter l'extension trigonométrique, qui ont été décrits en détail dans la documentation réalisée à cet effet.

On s'intéressera donc essentiellement aux structures de données qui ont été employées.

Les structures de données utilisées, mises à part les différentes listes d'instructions, de paramètres, d'attributs etc qui héritaient de la classe TreeList, l'ont été essentiellement dans l'étape de génération de code en assembleur, dans la classe que nous avons créé, nommée Codegen.

Dans ces structures, nous pouvons trouver :

- Une ArrayList *initializedVariables* qui stocke les adresses dans la pile des variables non initialisées, ce qui permet de gérer les erreurs qui occurrent lorsqu'il y a une opération avec des variables non initialisées
- Un HashMap defClass qui associe au nom d'une classe sa définition
- Un HashMap initClass qui associe au nom d'une classe le Label qui va permettre d'en initialiser les différents attributs à la fin du code assembleur

Comme cela est cité ci-dessus, la structure de données qui a été le plus mise en avant est le dictionnaire. Ce choix a été fait car nous trouvions que c'était la structure le plus adapté à la génération de langage assembleur. En effet, étant donné que chaque Label d'initialisation était défini une et une seule mais qu'il était utilisé à chaque instruction New, il fallait pouvoir le retrouver rapidement en le reliant au nom de la classe qu'il représentait. Il en est de même pour la définition des classes. Ayant besoin des définitions tout au long du développement de la génération de code, il fallait un moyen efficace de s'y retrouver entre les différentes classes créées dans le programme Deca.