Grenoble INP - Ensimag

Projet Génie Logiciel

DOCUMENT DE CONCEPTION

Architecture et Choix Techniques

ÉQUIPE GL56

Ayoub – Syntaxe & Arbre Abstrait
Hamza – Architecture Génération de Code
Mouad – Optimisations & Génération de Code
Chaima – Vérification Contextuelle
Abdellah – Extensions Mathématiques

Janvier 2026 – Version 1.0

Table des matières

1 ARCHITECTURE ET ORGANISATION DES CLASSES

L'architecture du compilateur est organisée autour d'un noyau central qui orchestre les différentes phases (Lexer, Parser, Vérification, Génération). Les données sont représentées par un Arbre Syntaxique Abstrait (AST) dont les nœuds sont typés.

1.1 Le Chef d'Orchestre : DecacCompiler

La classe `DecacCompiler` est le point d'entrée unique du compilateur.

— Elle contient l'état global de la compilation (options, table des symboles).

— Elle déclenche séquentiellement les étapes A, B et C.

— Elle lie les classes entre elles : par exemple, lors de la génération de code, chaque nœud de l'arbre reçoit une référence au `DecacCompiler` pour accéder aux gestionnaires de mémoire et de registres.

1.2 L'Arbre Syntaxique : Partie Sans Objet

Cette partie constitue le cœur du langage (variables, boucles, calculs). Toutes les classes héritent de la classe abstraite `Tree`. Nous avons structuré l'héritage pour regrouper les comportements communs et éviter la duplication de code.

`AbstractProgram` et `Program`
Représente la racine. Il contient une liste de déclarations de variables globales et le bloc d'instructions principal (`Main`).

`AbstractInst`
Regroupe toutes les instructions de contrôle (`While`, `IfThenElse`) et les opérations d'entrées-sorties (`Print`, `Println`).

`AbstractExpr` et Opérations
Les expressions sont divisées en familles pour partager la logique de vérification et de génération de code :

— `AbstractOpArith` (`Plus`, `Minus`...) : Partagent la logique de propagation des types (`int` op `float` → `float`).

— `AbstractOpBool` (`And`, `Or`) : Partagent la logique des sauts conditionnels pour l'évaluation paresseuse.

— `AbstractOpCmp` (`Equals`, `Lower`...) : Gèrent les comparaisons et la génération des instructions `CMP`.

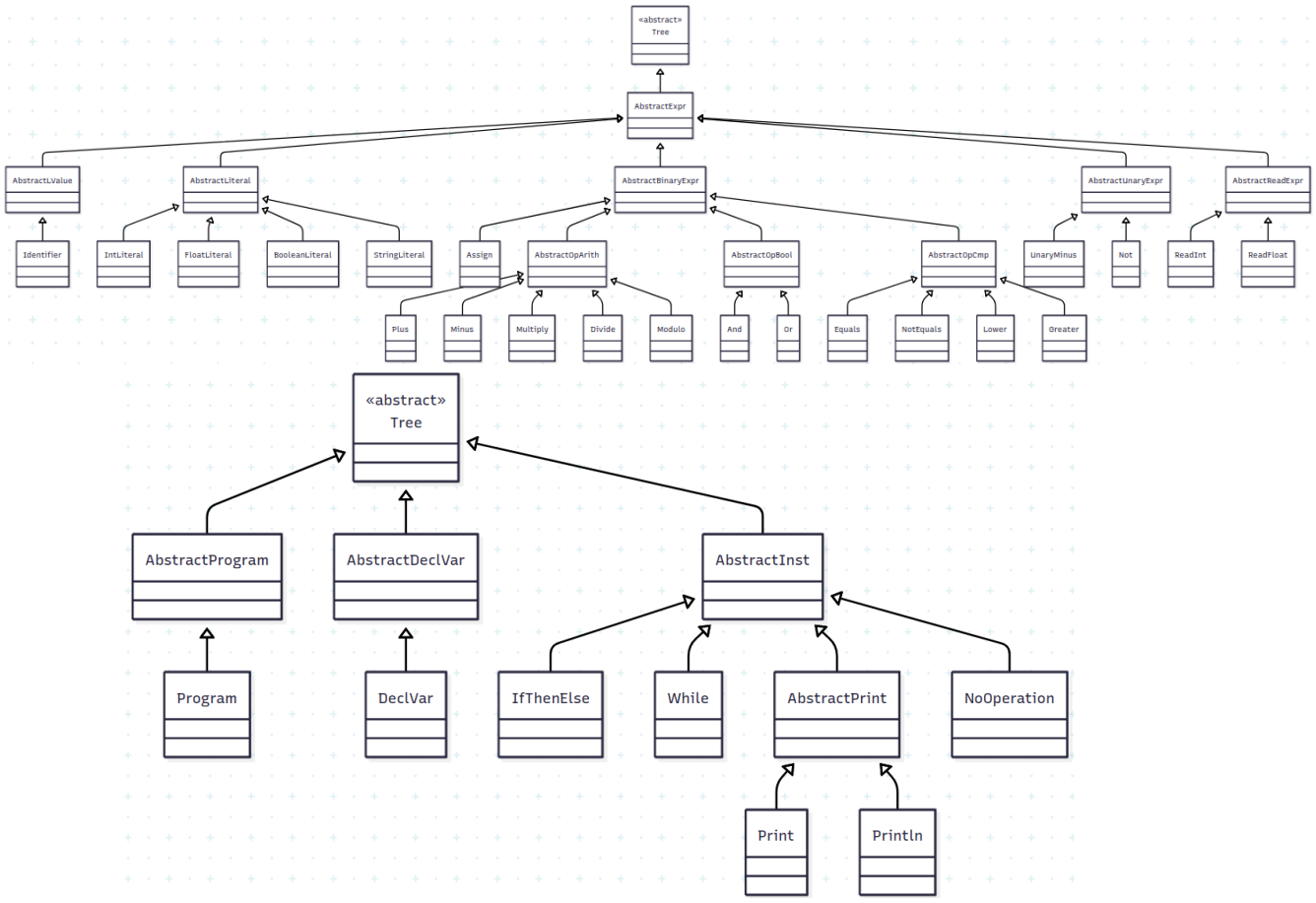

FIGURE 1 – Arbre de dépendance des classes : Partie Sans Objet

1.3 L'Arbre Syntaxique : Partie Objet

Pour supporter la programmation orientée objet, nous avons étendu l'arbre existant avec de nouveaux nœuds spécifiques aux déclarations de classes et aux interactions entre objets.

`AbstractDeclClass` et `DeclClass`

Gère la déclaration d'une classe. Elle contient des listes de `DeclField` (champs) et `DeclMethod` (méthodes).

`AbstractMethodBody`

C'est ici que réside une spécificité de notre implémentation. Cette classe abstraite possède deux enfants :

- `MethodBody` : Le corps standard d'une méthode Deca (déclarations locales + instructions).
- **`MethodBodyAsm`** : Une extension permettant de stocker du code assembleur brut (String). Cela permet au compilateur de traiter l'assembleur inline comme n'importe quel autre corps de méthode lors de l'analyse.

`Expressions Objet`

De nouvelles expressions ont été ajoutées pour manipuler le tas et la table des méthodes :

- `New` : déclarer une nouvelle variable.
- `MethodCall` : Appel de méthode.

— `Selection` : Accès à un champ d'un objet (`obj.field`).

— `Cast` et `InstanceOf` : Vérification dynamique de types.

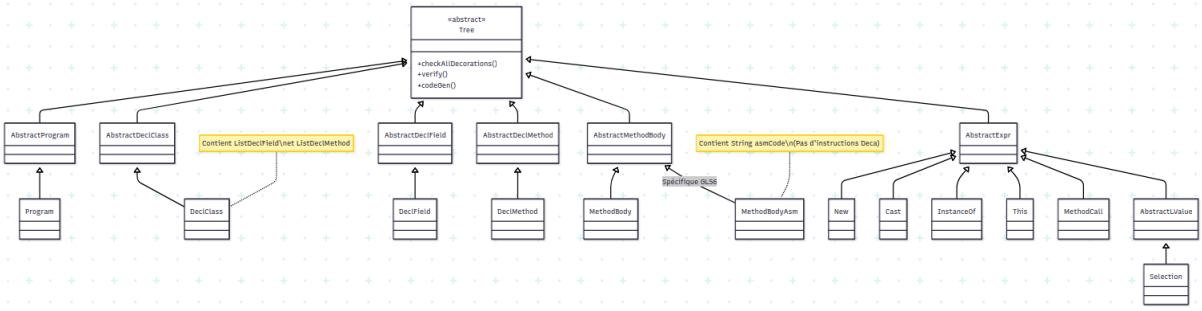

FIGURE 2 – Arbre de dépendance des classes : Partie Objet et Extensions

1.4 Classes Utilitaires et Gestionnaires

Ces classes ne font pas partie de l'arbre mais sont indispensables à son traitement.

— **`EnvironmentExp`** : Structure de données (Pile de Dictionnaires) utilisée lors de la passe de vérification contextuelle pour associer les identifiants à leurs définitions (Type, Emplacement).

— **`RegisterHandler` (Spécifique GL56)** : Classe responsable de l'allocation des registres physiques (`R2-R15`) lors de la génération de code.

— **`MemoryManager`** : Centralise le calcul des adresses. Elle gère l'avancement des pointeurs de pile (`LB` pour les variables locales) et globaux (`GB` pour les tables des méthodes et variables globales).

2 SPÉCIFICATIONS INTERNES ET CHOIX FONCTIONNELS

Cette section détaille les spécifications que nous avons définies nous-mêmes pour combler les zones d'ombre du sujet ou pour faciliter l'implémentation de fonctionnalités avancées.

2.1 Support de l'Assembleur Inline (MethodBodyAsm)

Spécification : Le langage Deca standard impose que le corps d'une méthode soit constitué de déclarations de variables et d'instructions. Nous avons étendu cette spécification en introduisant un nouveau type de corps de méthode : le corps "natif" en assembleur. Dans notre AST, cela se traduit par le nœud `MethodBodyAsm`, qui contient une chaîne de caractères brute représentant des instructions IMA.

Justification : Cette fonctionnalité était indispensable pour l'implémentation performante de la bibliothèque mathématique (`Math.decah`). Elle permet d'écrire des algorithmes bas niveau (comme CORDIC pour sinus/cosinus) directement en assembleur optimisé, tout en les exposant au reste du programme Deca via une signature de méthode typée. Cela évite la complexité de l'édition de liens avec des fichiers objets externes.

2.2 Gestion "Défensive" de la Pile (TSTO Global)

Spécification : Concernant la gestion de la mémoire de pile, nous avons opté pour une politique d'allocation statique par méthode. Au début de chaque méthode, le compilateur calcule la taille maximale nécessaire (`StackSize`) définie par :

$$Taille = VarsLocales + SauvegardeRegistres + MaxTemporairesExpressions$$

Une unique instruction `TSTO` (Test Stack Overflow) est générée au tout début de la méthode pour cette taille totale.

Justification : Le sujet laisse le choix de la gestion du débordement. Nous avons choisi cette approche conservatrice pour garantir la sûreté de l'exécution. Si le test `TSTO` passe à l'entrée de la méthode, nous avons la garantie mathématique qu'aucun *Stack Overflow* ne surviendra au milieu d'un calcul complexe. Cela simplifie également le calcul des offsets par rapport au registre de base (`LB`).

2.3 Convention d'Appel et Sauvegarde des Registres

Spécification : Lors d'un appel de méthode, nous suivons la convention "Callee-Save" (le appelé sauvegarde) pour les registres de travail (`R2-R15`). Cependant, le résultat de la fonction est toujours retourné dans le registre `R0`.

Justification : Le registre `R0` est implicitement utilisé par de nombreuses instructions IMA (comme retour de routine). Utiliser `R0` comme registre de retour standardise les interfaces entre méthodes et simplifie la récupération du résultat dans l'expression appelante.

3 ALGORITHMES ET STRUCTURES DE DONNÉES

Cette section décrit la mécanique interne du compilateur, notamment les algorithmes non triviaux développés pour l'étape de génération de code.

3.1 Structure de Données : Environnements Empilés (Symbol Table)

Description : Pour l'étape de vérification contextuelle (Passe B), nous n'utilisons pas une simple table de hachage, mais une structure hiérarchique récursive via la classe `EnvironmentExp`. Chaque environnement maintient :

1. Une `Map<Symbol, Definition>` pour les symboles déclarés à ce niveau.
2. Un pointeur `parent` vers l'environnement englobant.

Justification : Cette structure en "pile de dictionnaires" modélise naturellement la portée statique (lexicale) du langage Deca. L'algorithme de résolution d'un identifiant est simple et robuste : on cherche dans la map courante ; si on ne trouve pas, on délègue récursivement au parent. Cela gère automatiquement le masquage des variables (*shadowing*) : la définition la plus "proche" est toujours trouvée en premier.

3.2 Algorithme : Construction des VTables (Dispatch Dynamique)

Description : Pour gérer le polymorphisme, nous construisons pour chaque classe une Table des Méthodes Virtuelles (VTable) dans le segment de données statiques. L'algorithme de construction est le suivant :

1. Si la classe a un parent, on **copie** la VTable du parent.
2. On parcourt les méthodes de la classe courante :
   — Si la méthode existe déjà dans la table (Override), on met à jour l'étiquette (adresse) de la méthode.
   — Sinon, on **ajoute** la nouvelle méthode à la fin de la table.

Justification : Cet algorithme garantit que l'index d'une méthode donnée est constant pour toute une hiérarchie d'héritage. Cela permet de compiler un appel de méthode `o.m()` en une opération en temps constant $O(1)$ (instruction `BSR` avec offset fixe), quelle que soit la classe réelle de l'objet à l'exécution.

3.3 Allocation de Registres

Description : L'allocation des registres est gérée par la classe `RegisterHandler`.

— **Allocation :** On demande le premier registre libre entre `R2` et `R15`. La méthode `allocate()` parcourt séquentiellement les registres et retourne le premier disponible.

— **Spilling (Débordement) :** Si tous les registres sont occupés, `allocate()` retourne `null`. C'est alors le code appelant (lors de la génération de code pour les expressions) qui gère le débordement en :

1. Sauvegardant le registre courant avec `PUSH`
2. Réutilisant ce même registre pour l'opération suivante
3. Restaurant la valeur avec `POP` dans `R0`
4. Effectuant l'opération avec `R0` et le registre courant

— **Gestion de la pile :** Le `RegisterHandler` maintient un compteur de profondeur de pile (`currentStackDepth` et `maxStackDepth`) pour suivre l'utilisation de la pile durant le spilling, mais ne gère pas directement les instructions PUSH/POP.