GRENOBLE-INP ENSIMAG

INGÉNIERIE DES SYSTÈMES D'INFORMATION

2A

GRENOBLE-INP ENSIMAG

INFORMATION SYSTEMS ENGINEERING

2A

ÉCOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE ET DE MATHÉMATIQUES APPLIQUÉES DE GRENOBLE

PROJET GL

DOCUMENTATION DE CONCEPTION

Réalisé par :

DIAKITE Alpha Ousmane
DREUILLE Nathan
FLISS Yassine
KOSSI Yao David
OUDRHIRI IDRISSI Safwane

CLIENT :

FREIRE MARCO

Année académique : 2025–2026

Table des matières

Etape A du compilateur : Lexing et Parsing

1 Analyse lexicale

1.1 Objectif de l'analyse lexicale

L'analyse lexicale transforme un programme source Deca en une suite de *tokens* destinée à l'analyse syntaxique. Elle identifie les mots-clés, identificateurs, littéraux, opérateurs et ponctuations, et élimine les éléments non significatifs (espaces et commentaires) via l'utilisation de l'action `-> skip`.

L'implémentation repose sur **ANTLR4** via une grammaire de lexer (`DecaLexer.g4`) qui étend la classe fournie `AbstractDecaLexer`. Cette classe est notamment utilisée pour la gestion des `#include`.

1.2 Tokens implémentés

Mots-clés

Les mots-clés définis par la spécification Deca sont implémentés comme des règles lexicales dédiées afin d'éviter toute ambiguïté avec les identificateurs. Ils sont placés **avant** la règle `IDENT` afin qu'ils soient reconnus prioritairement et ne soient pas interprétés comme des identificateurs.

Exemples :

— Contrôle : `if`, `else`, `while`, `return`
— Objet : `class`, `extends`, `new`, `protected`
— Constantes : `true`, `false`, `null`, `this`
— Entrées/sorties : `print`, `println`, `printx`, `printlnx`, `readInt`, `readFloat`
— Autres : `instanceof`, `asm`

Identificateurs

Les identificateurs respectent la forme suivante :

— commencent par une lettre, `_` ou `$`
— peuvent contenir lettres, chiffres, `_` ou `$`

```
IDENT : (LETTER | '$' | '_') (LETTER | DIGIT | '$' | '_')*;
```

Littéraux entiers

Les entiers respectent la contrainte « sans zéros en tête » (sauf `0`) :

```
INT : '0' | [1-9] [0-9]*;
```

Littéraux flottants

Les flottants sont définis conformément à la spécification :

— flottants décimaux avec exposant optionnel et suffixe `f/F`

— flottants hexadécimaux (`0x...p...`)

La règle `FLOAT` s'appuie sur les fragments `FLOATDEC` et `FLOATHEX` afin d'être plus lisible :

```
FLOAT : FLOATDEC | FLOATHEX;
```

Chaînes de caractères

Deux formes sont supportées :

— `STRING` : chaînes sur une seule ligne

— `MULTI_LINE_STRING` : chaînes multi-lignes

Seuls les échappements ¨ et

sont autorisés. Les retours à la ligne sont interdits dans `STRING` mais autorisés dans `MULTI_LINE_STRING`.

Opérateurs et ponctuation

Tous les opérateurs et symboles requis par Deca sont implémentés :

— opérateurs arithmétiques, relationnels et logiques (`+`, `-`, `*`, `/`, `%`, `==`, `!=`, `<=`, `>=`, `&&`, `||`)

— ponctuation (`;`, `,`, `.`, `(`, `)`, `{`, `}`)

Commentaires et séparateurs

Les commentaires mono-ligne (`//`) et multi-lignes (`/* ... */`) ainsi que les séparateurs (espaces, tabulations, retours à la ligne) sont ignorés par le lexer :

```
WS : [ \t\r\n]+ -> skip;
```

1.3 Gestion de l'instruction `#include`

Principe

La directive `#include` permet d'inclure un fichier Deca dans un autre. Sa gestion est réalisée au niveau lexical afin de rendre l'inclusion transparente pour l'analyse syntaxique.

Règle lexicale

Une règle dédiée reconnaît la directive `#include`. Le token est ignoré (`-> skip`) et une action Java déclenche l'inclusion en appelant `doInclude(getText())`.

```
INCLUDE
  : '#include' ' '* '"' FILENAME '"'
    { doInclude(getText()); }
    -> skip;
```

On utilise ici la méthode `doInclude`, déjà implémentée dans la classe `AbstractDecaLexer`.

Mécanisme d'inclusion

La méthode `doInclude`, fournie par `AbstractDecaLexer`, réalise les opérations suivantes :

- recherche du fichier à inclure (répertoire courant puis bibliothèque standard)
- changement temporaire du flux d'entrée du lexer
- restauration automatique du flux précédent à la fin du fichier inclus
- détection des inclusions circulaires
- gestion des erreurs (fichier introuvable ou illisible)

2 Analyse syntaxique (Parser)

2.1 Objectif de l'analyse syntaxique

L'analyse syntaxique (parsing) consomme la suite de tokens produite par le lexer et vérifie que le programme respecte la grammaire du langage Deca. En parallèle, elle construit un **AST** (*Abstract Syntax Tree*) en instanciant explicitement des nœuds (classes du paquetage `fr.ensimag.deca.tree`) dans les actions Java des règles ANTLR.

Le parser est généré avec **ANTLR4** à partir de la grammaire `DecaParser.g4`. Il étend la classe `AbstractDecaParser`, qui centralise des méthodes utilitaires (gestion d'erreurs, localisation, etc.).

2.2 Structure générale du programme

Le point d'entrée de la grammaire est la règle `prog`. Elle reconnaît :

- une liste (éventuellement vide) de déclarations de classes,
- le bloc principal (`main`),

— puis le token `EOF`.

L'AST racine construit est un objet `Program` contenant la liste des classes et le main.

2.3 Main et blocs

La règle `main` accepte soit un bloc principal, soit l'absence de main (epsilon), auquel cas un `EmptyMain` est créé.

Un bloc est délimité par des accolades et contient **d'abord** une liste de déclarations de variables, puis une liste d'instructions. Cette séparation impose donc l'ordre « déclarations avant instructions » au sein d'un bloc, ce qui explique qu'une déclaration par exemple (`int x;`) placée après une instruction (`println(...);`) est rejetée par la grammaire.

2.4 Déclarations de variables

Les déclarations suivent la forme :

`type decl_var( , decl_var )* ;`

Chaque variable peut être initialisée par une expression via `=` (`EQUALS`). Le parser construit un nœud `DeclVar` contenant :

— le type (`AbstractIdentifier`),

— l'identificateur (`Identifier`),

— l'initialisation : `Initialization(expr)` si présente, sinon `NoInitialization`.

2.5 Instructions

Les instructions supportées à ce stade comprennent :

— une expression suivie de `;` (instruction-expr),

— l'instruction vide `;` (création d'un `NoOperation`),

— `print`, `println`, `printx`, `printlnx`,

— `if / else if* / else?` (construction de `IfThenElse`),

— `while` (construction de `While`),

— `return expr ;` (construction de `Return`).

La règle `if_then_else` gère les chaînes de `else if` en construisant une structure imbriquée de nœuds `IfThenElse`. Concrètement, chaque `else if` est encodé comme une branche `else` contenant une liste d'instructions dont le premier élément est un nouveau `IfThenElse`, ce qui permet d'obtenir une représentation uniforme dans l'AST.

2.6 Expressions et gestion des précédences

Les expressions sont structurées en plusieurs niveaux afin de respecter les précédences et associativités :

`assign` → `or` → `and` → (`==`/`!=`) → (`<`/`<=`/`>`/`>=`/`instanceof`) → (`+`/`-`) → (`*`/`/`/`%`) → `unaire`

→ `sélection/appel` → `primaire`.

L'affectation est gérée dans `assign_expr` et est **récursive à droite** (*right-associative*) pour permettre des expressions du type `a = b = c`. Une contrainte est ajoutée : l'expression à gauche doit être un `LValue`. Si ce n'est pas le cas, le parser déclenche une erreur `InvalidLValue`. Cela garantit que seules des expressions assignables (identificateurs, sélections, etc.) apparaissent à gauche du `=`.

2.7 Sélection et appel de méthode

La règle `select_expr` permet de reconnaître :

— `e.i` : sélection d'un champ (nœud `Selection`),

— `e.i(args)` : appel de méthode sur un objet (nœud `MethodCall`).

On supporte également les appels sans receveur explicite (`m(args)`) via `primary_expr`, construisant alors un `MethodCall` avec un receveur `null`.

2.8 Littéraux, identificateurs et types

La règle `literal` construit les nœuds d'AST associés : `IntLiteral`, `FloatLiteral`, `StringLiteral`, `BooleanLiteral`, `Null`, `This`. La règle `ident` crée un `Identifier` à partir de la table des symboles du compilateur : `compiler.createSymbol($IDENT.text)`. Dans cette étape du projet, la règle `type` réutilise `ident` : un type est donc représenté comme un identificateur.

2.9 Gestion des classes (POO)

Le parser supporte une liste (éventuellement vide) de déclarations de classes. Une classe est reconnue sous la forme :

```
class <name> <extends?> { <body> }
```

L'AST correspondant est un `DeclClass` contenant :

— le nom de la classe,

— la super-classe éventuelle (ou `null` si absence de `extends`),
— la liste de champs (`ListDeclField`),
— la liste de méthodes (`ListDeclMethod`).

La visibilité est gérée par la règle `visibility` : par défaut, elle vaut `PUBLIC` (epsilon), sinon elle peut être explicitement `PUBLIC` ou `PROTECTED`. Les méthodes sont définies soit par un bloc, soit par une forme `asm(...)` prenant une `STRING` ou une `MULTI_LINE_STRING` via la règle `multi_line_string`.

2.10 Construction de l'AST et localisation

Les nœuds de l'AST ne sont pas générés automatiquement par ANTLR : ils sont **instanciés explicitement** dans les actions Java des règles du parser (`new Program(...)` etc.). La localisation (fichier/ligne/colonne) est propagée grâce à `setLocation(node, token)` afin de produire des diagnostics précis lors des phases suivantes (analyse contextuelle, génération de code). Des assertions (`assert`) permettent également de vérifier que les sous-arbres attendus sont non nuls avant de construire un nœud parent.

2.11 Interaction avec la gestion des `#include`

La directive `#include` est traitée au niveau lexical, comme nous l'avons deja evoqué auparavant dans le rapport(changement temporaire de `CharStream` dans `AbstractDecaLexer`). Le parser reçoit donc un flux de tokens correspondant au fichier principal « déplié » avec ses fichiers inclus, sans nécessiter de règle syntaxique dédiée à `#include`.

2.12 Extensibilité et simplicité d'évolution de la grammaire

Nous avons fait en sorte de synthetiser notre code pour faciliter l'ajout de nouvelles constructions syntaxiques. L'organisation des règles (séparation `decl`/`inst`, expressions en niveaux de précédence, règles dédiées pour les structures de contrôle) permet d'ajouter une nouvelle instruction très simplemment.

Par exemple, l'ajout futur de structures comme `do ... while` ou `for` (non présentes actuellement) se ferait en :

1. ajoutant les tokens/mots-clés nécessaires dans le lexer,
2. ajoutant une règle dédiée (ex. `do_while`, `for_loop`) ou une alternative dans `inst`,
3. instanciant le nœud d'AST correspondant (ex. `DoWhile`, `For`) en réutilisant les structures existantes (`ListInst`, `AbstractExpr`, etc.).

Cette approche nous permettrait donc d'apporter facilement des évolutions au langage sans craindre des gros "bug" du compilateur.

3 Étape B du compilateur : Vérifications contextuelles

3.1 Organisation générale et structures de base

Dans cette documentation, l'architecture du compilateur `Deca` est décrite à travers des diagrammes UML représentant les dépendances entre les principales classes. Ces diagrammes permettent de visualiser l'organisation globale sans entrer dans les détails d'implémentation, volontairement laissés à la documentation `Javadoc`.

Nous commençons par présenter les structures fondamentales sur lesquelles repose la vérification contextuelle.

L'environnement des types est centralisé dans la classe `DecacCompiler`, via un attribut public de type `EnvironmentType`. Cet environnement encapsule une `HashMap<Symbol, TypeDefinition>` initialisée avec les types prédéfinis du langage (types primitifs, type `null`, classe `Object`). L'ajout de la méthode prédéfinie `equals` a été réalisé à ce niveau afin de garantir sa visibilité globale et sa cohérence avec le reste du système de types. Ce choix permet un accès direct et uniforme aux types tout au long des différentes passes de vérification.

L'environnement des expressions est implémenté par la classe `EnvironmentExp`, qui repose également sur une `HashMap<Symbol, ExpDefinition>`. Cette structure garantit :

— l'unicité des identificateurs dans un environnement donné ;

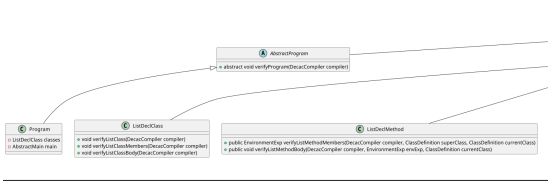
— un temps de recherche optimal, nécessaire lors de la vérification contextuelle.

Afin de modéliser correctement les portées imbriquées, `EnvironmentExp` fournit des mécanismes d'empilement et de fusion contrôlée des environnements (`stack` et `add`), permettant de représenter les blocs, méthodes et classes de manière hiérarchique.

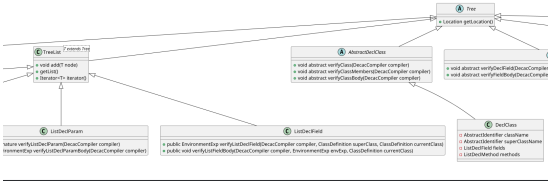
3.2 Extension objet et séparation des passes de vérification

Afin de faciliter la compréhension de l'architecture globale du compilateur, nous avons choisi de représenter la hiérarchie des classes à l'aide d'un diagramme UML présenté dans la figure 1 plutôt que par une description textuelle exhaustive. Ce choix permet de visualiser plus clairement les relations d'héritage et les dépendances entre les différentes entités, tout en offrant une vue d'ensemble cohérente de l'organisation du code. Toutefois, en raison de la taille et de la densité de cette hiérarchie, le diagramme a été scindé en trois parties distinctes afin de garantir une lisibilité suffisante. Ces trois sous-figures constituent ainsi une représentation complémentaire et continue de la structure globale présentée dans cette section.

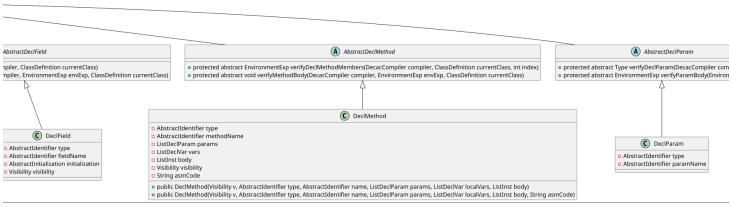
L'extension objet du compilateur repose sur l'introduction des hiérarchies abstraites `AbstractDeclClass`, `AbstractDeclField`, `AbstractDeclMethod` et



(a) Partie 1



(b) Partie 2



(c) Partie 3

FIGURE 1 – Diagramme UML de la hiérarchie des classes

`AbstractDeclParam`, ainsi que sur les classes de listes associées. Ces structures ont été essentielles pour organiser la vérification contextuelle des programmes orientés objet et pour factoriser le traitement commun des déclarations lors des différentes passes.

Les trois passes de l'étape B sont lancées successivement depuis le nœud `Program` et propagées récursivement vers `ListDeclClass` à l'aide de trois méthodes distinctes. Ce choix permet de séparer clairement les responsabilités de chaque passe et de contrôler précisément l'ordre dans lequel les informations sont construites et utilisées.

Pour les méthodes et leurs paramètres, la passe 2 est dédiée à la vérification de la validité des déclarations et des signatures (types de retour, paramètres, redéfinitions). La passe 3 crée ensuite l'environnement local associé à chaque méthode et procède à la vérification de son corps, en s'appuyant sur les informations validées précédemment.

Concernant les champs, leur déclaration est vérifiée lors de la passe 2 afin de garantir la cohérence des types et des visibilités, tandis que leur initialisation est traitée lors de la passe 3, une fois l'ensemble des définitions nécessaires disponibles.

Cette organisation en passes distinctes vise avant tout à respecter le cahier des charges du compilateur, qui impose un cadre strict quant au rôle de chaque passe de vérification. Chaque étape est ainsi limitée à des responsabilités bien définies, garantissant la conformité de l'implémentation avec la structure attendue et facilitant le raisonnement sur la vérification contextuelle.

La figure 2 représente les quelques fonctions issues de Abstractexpr et AbstractInst qu'on a du ajouter pour le bon fonctionnement de deca objet et conformément au cahier de charge.

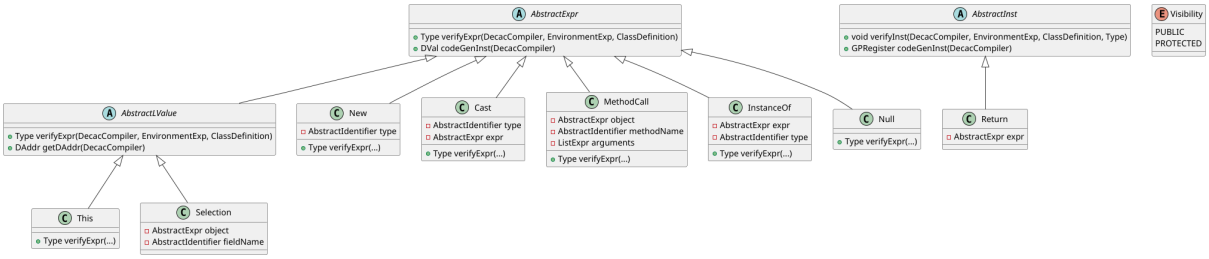


FIGURE 2 – Diagramme de classes du package `tree`

4 Etape C du compilateur : Génération du code ASM

La phase de génération de code (Gencode) traduit l'arbre syntaxique abstrait décoré en instructions pour la machine virtuelle IMA. Cette étape repose principalement sur les classes situées dans le package `fr.ensimag.deca.tree` pour la structure du langage et `fr.ensimag.deca.codegen` pour la gestion des ressources machine.

Nous avons privilégié une approche modulaire où le `DecacCompiler` centralise l'état global (options, drapeaux d'erreurs, gestionnaires), tandis que chaque nœud de l'arbre est responsable de sa propre génération de code.

4.1 Le Gestionnaire de Registres

La classe `fr.ensimag.deca.codegen.GestionnaireRegistre` est un composant central de l'étape de génération de code. Elle a pour responsabilité d'abstraire les contraintes physiques de la machine cible (IMA) en fournissant une interface d'allocation de registres virtuellement illimitée aux nœuds de l'arbre syntaxique.

Elle gère l'ensemble des registres banalisés disponibles, de R_2 à R_{MAX} (généralement R_{15}), et implémente la stratégie de "Spill" (débordement sur pile) ainsi qu'un mode de simulation pour le calcul de la taille de la pile.

Stratégie d'Allocation et de Spill

L'algorithme d'allocation implémenté dans la méthode `utiliserRegistre` suit une logique gloutonne linéaire avec gestion de saturation :

1. **Cas Nominal (Registres disponibles) :** Tant que le `compteurRegistre` est strictement inférieur à R_{MAX}, la méthode retourne le registre correspondant à l'index courant et incrémente le compteur. Cela garantit une utilisation optimale des registres rapides.
2. **Cas de Saturation (Spill) :** Lorsque tous les registres physiques sont occupés (`compteurRegistre` $\geq R_{MAX}$), le gestionnaire bascule en mode débordement :
   — Il incrémente le compteur `nbRegStokes`.

— Il génère immédiatement une instruction `PUSH` du dernier registre physique ($R_{MAX} - 1$) sur la pile.
— Il retourne ce même registre ($R_{MAX} - 1$) pour qu'il soit réutilisé par l'opération courante.

La méthode `libererRegistre` effectue l'opération inverse : elle décrémente `nbRegStokes` si des registres ont été "spillés", ou décrémente `compteurRegistre` sinon. Elle assure la cohérence de l'état interne du gestionnaire sans générer d'instruction `POP` (cette responsabilité est déléguée à l'appelant, typiquement `AbstractOpArith`, pour permettre des optimisations contextuelles).

Mode Simulation pour le Calcul du TSTO

Une fonctionnalité clé de cette classe est sa capacité à simuler une allocation sans émettre d'instructions assembleur. Ce mécanisme est indispensable pour prédire la taille de pile nécessaire (`TSTO`) avant le début de la génération du code d'une méthode.

Le gestionnaire maintient un état sauvegardé via les attributs `saveCompteurRegistre` et `saveNbRegStokes`.

— **Début de simulation :** La méthode `commencerSimulationRegistre` sauvegarde l'état actuel des compteurs.
— **Simulation :** Les méthodes `simulerAllocationRegistre` et `simulerLiberationRegistre` modifient les compteurs comme lors d'une exécution réelle, permettant de calculer le pic d'utilisation des registres et de la pile (variable `nbRegStokes`).
— **Restauration :** La méthode `resetGestionnaireRegistre` permet de revenir à l'état sauvegardé une fois la simulation terminée, garantissant que la génération de code réelle démarre avec un état cohérent.

Cette architecture permet de dissocier l'analyse des besoins en ressources de la génération effective des instructions (Passe 3), tout en réutilisant la même logique algorithmique.

Le Gestionnaire de Pile

La classe `fr.ensimag.deca.codegen.GestionnairePile` a été conçue pour suivre dynamiquement l'évolution de la pile lors de la génération de code. Elle maintient deux compteurs : `tailleCourante` et `maxTaillePile`. À chaque instruction influant sur la pile (`PUSH`, `POP`, `ADDSP`, `BSR`), les méthodes `incrementer()` ou `decrementer()` sont appelées. Le maximum atteint est mémorisé pour générer l'instruction `TSTO` adéquate au début de chaque bloc (Main, Méthode, Init).

Génération des Erreurs Runtime

Afin d'optimiser la taille du code assembleur généré, les blocs de gestion d'erreurs (Overflow, Division par zéro, Tas plein, etc.) ne sont pas dupliqués à chaque occurrence. Nous utilisons un système de drapeaux booléens dans `DecacCompiler` (ex : `getOverflowLabel()`). Lorsqu'un nœud de l'arbre a besoin d'une vérification (par exemple une division), il active le drapeau correspondant. À la fin de la méthode `codeGenProgram` dans `Program.java`, la méthode `ajouteLabelErreurDemandee` génère uniquement les étiquettes d'erreurs qui ont été sollicitées durant la compilation.

Liste des erreurs gérées :

— `stack_overflow_error` : Débordement de pile.
— `overflow_error` : Débordement arithmétique sur flottant (si option `-n` absente).
— `div_zero_error` : Division par 0 (entier ou flottant).
— `tas_plein` : Échec d'allocation mémoire (`NEW`).
— `dereferencement.null` : Accès à un membre d'un objet null.
— `cast_error`, `io_error`, `absence_return_error`, etc.

4.2 Génération des Expressions Arithmétiques

La génération des expressions repose sur un contrat fort : la méthode `codeGenInst(DecacCompiler compiler)` retourne systématiquement un objet de type `DVal` (valeur directe, registre ou littéral) contenant le résultat de l'évaluation.

Factorisation dans `AbstractOpArith`

La classe abstraite `fr.ensimag.deca.tree.AbstractOpArith` implémente une logique générique robuste pour toutes les opérations binaires (+, -, *, /) :

1. **Évaluation des opérandes :** L'opérande gauche est évalué. Si le résultat est un littéral, il est immédiatement chargé dans un registre alloué via `compiler.utiliserRegistre()`. Sinon, le registre résultat est conservé.
2. **Gestion des Conversions de Type :** Avant d'évaluer la droite, nous vérifions les types. Si une opération mixe `int` et `float`, l'instruction de conversion `FLOAT` est injectée pour l'opérande concernée (gauche ou droite).
3. **Gestion du Spill (Débordement de registres) :** Si l'évaluation de la droite a provoqué un "Spill" de l'opérande gauche (détecté par `operande1 == operande2`), nous utilisons le registre `R0` comme tampon pour dépiler la valeur de gauche et effectuer l'opération, avant de recharger le résultat dans le bon registre.
4. **Libération :** Les registres temporaires utilisés par l'opérande de droite sont libérés immédiatement après l'opération.

Cas Spécifique : Division et Modulo

Les classes `Divide` et `Modulo` héritent de `AbstractOpDivMod` que nous avons ajoutée à l'architecture déjà en place. Elles intègrent une vérification de la division par zéro (sauf si l'option `-n` est active). Pour éviter des erreurs de type lors de la comparaison, nous chargeons systématiquement le diviseur dans `R0` avant d'effectuer le `CMP #0` (ou `0.0`). Cela garantit que la comparaison se fait toujours sur un registre, indépendamment de la nature de l'opérande (littéral, variable, etc.).

4.3 Génération des Structures de Contrôle et Booléens

Pour les expressions booléennes (`And`, `Or`, `Not`, comparaisons), nous n'utilisons pas le calcul arithmétique (0 ou 1) par défaut, mais le branchement conditionnel (Lazy Evaluation).

- **Méthodes `codeGenVrai` et `codeGenFaux` :** Redéfinies dans les classes de comparaison, ces méthodes génèrent des sauts (`BEQ`, `BNE`, etc.) vers des étiquettes fournies en paramètre.
- **Structures `IfThenElse` et `While` :** Elles gèrent la création des étiquettes de début, de fin et de clause `else`, déléguant la génération des sauts aux expressions booléennes contenues.

4.4 Gestion des Chaînes de Caractères (Print)

La méthode `codeGenPrint` dans `StringLiteral` traite les caractères spéciaux avant l'affichage. Une analyse de la chaîne permet de découper le contenu sur les sauts de ligne (`\n` ou saut physique) pour alterner entre les instructions `WSTR` (affichage) et `WNL` (saut de ligne assembleur), garantissant un formatage correct en sortie.

4.5 Implémentation de l'Objet

L'implémentation de la partie Objet suit une stratégie en deux passes dans `Program.java`, optimisée pour réduire la taille du code et les cycles CPU.

Passe 1 : Construction des VTables

Cette passe génère les tables virtuelles de toutes les classes. **Optimisation R0/R1 :** Nous avons remarqué que l'adresse de la classe `Object` (`null`) et l'adresse de la méthode `Object.equals` sont utilisées répétitivement.

- Nous chargeons `null` dans `R0` et l'adresse de `code.Object.equals` dans `R1` au tout début du programme.

— Lors de la construction des VTables, nous utilisons directement `STORE R0, ...` et `STORE R1, ...` sans avoir besoin de les recharger, économisant ainsi des cycles d'horloge précieux.

Passe 2 : Corps des Méthodes et Initialisation

Nous générons ensuite séquentiellement :

1. Le programme principal (`Main`).
2. Les méthodes d'initialisation des champs (`init.Classe`), gérant l'héritage par appel au `init` du parent.
3. Le corps des méthodes (`code.Classe.Methode`), incluant la protection de pile.

4.6 Simulation des Registres et Calcul du TSTO

Afin d'éviter une passe supplémentaire coûteuse lors de l'étape C pour calculer la taille de la pile (`TSTO`), nous avons intégré une simulation lors de la vérification contextuelle (juste avant la génération du code asm).

La méthode `simuleExecutionNbRegistres(DecacCompiler compiler)` parcourt l'arbre des expressions. Elle simule l'état du gestionnaire de registres :

```
if (compiler.getNbRegStokes() > stokesAvant) {
    // Un spill a eu lieu à gauche : on restaure et on incrémente le compteur
    compiler.setNbRegStokes(stokesAvant + 1);
}
```

Cette méthode retourne le nombre maximum de registres utilisés simultanément. Couplée au `GestionnairePile`, elle permet de prédire exactement l'espace nécessaire pour les sauvegardes de registres (`PUSH`) et d'écrire l'instruction `TSTO` correcte dès le début de la génération de la méthode.