PHELMA GRENOBLE INP
ENSIMAG GRENOBLE INP

Conception du compilateur Deca

DIJS Thomas FALGAYRAC Loic HO-SUN Jules
NOIRY Sylvain WANG Caroline
groupe GL 54

Phelma SEOC/ Ensimag ISI-MMIS
January 28, 2022

Table des matières

1 Architecture

Le compilateur deca se compose d'un dossier ***main*** possédant fichiers et dossiers dont l'arborescence est illustrée dans la figure 1. Ces fichiers implémentent 4 étapes majeures du decac : une analyse syntaxique (en rouge), une analyse contextuelle (en vert), suivies d'une génération de code (en bleu) couplée avec une optimisation de celle-ci (en violet). Afin de faciliter la lecture des fichiers du compilateur, nous allons ci-dessous effectuer une description exhaustive de l'ensemble des fichier du dossier ***main***.

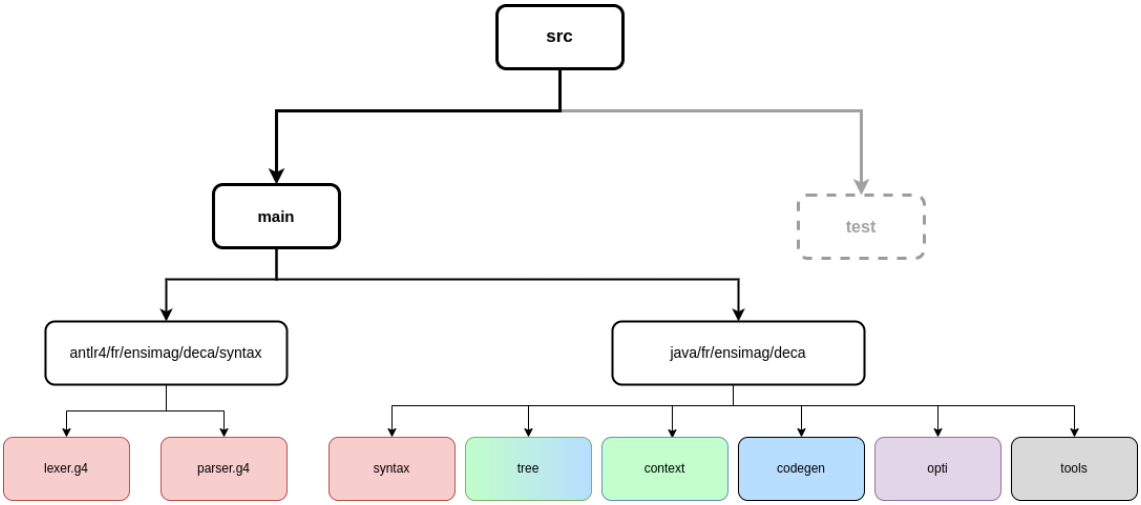


Figure 1: Architecture du dossier src

1.1 Syntax

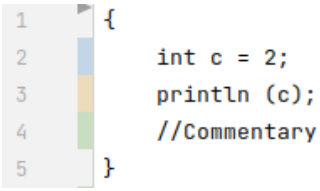
1.1.1 DecaLexer.g4

Le fichier DecaLexer.g4 est en charge de la définition du Lexer, ou analyseur lexical. Il a pour rôle de découper le code selon tous les mots détectés pour en créer des "lexèmes". Parmi ces lexèmes on retrouve :

- les mots réservés au langage ("while", "print", "class"...)
- les symboles spéciaux tels que les opérateurs, les parenthèses, les accolades...

- les commentaires, les chaînes de caractères, les types INT et FLOAT, les identificateurs et les "include"

Voici un exemple d'interprétation du code par le lexer :



```
{
    int c = 2;
    println (c);
    //Commentary
}
```

Figure 2: Exemple de code

```
OBRACE: [@0,0:0='{',<35>,1:0]
IDENT: [@1,6:8='int',<22>,2:4]
IDENT: [@2,10:10='c',<22>,2:8]
EQUALS: [@3,12:12='=',<25>,2:10]
INT: [@4,14:14='2',<45>,2:12]
SEMI: [@5,15:15=';',<38>,2:13]
PRINTLN: [@6,21:27='println',<14>,3:4]
OPARENT: [@7,29:29='(',<33>,3:12]
IDENT: [@8,30:30='c',<22>,3:13]
CPARENT: [@9,31:31=')',<34>,3:14]
SEMI: [@10,32:32=';',<38>,3:15]
CBRACE: [@11,80:80='}',<36>,5:0]
```

Figure 3: Code vu par le lexer

On remarque sur cet exemple que le lexer ne prend pas en compte les commentaires, les espaces, les retours chariots et les "fins de ligne" : c'est la fonction `skip()` qui permet de ne pas prendre en compte les caractères qui n'ont pas de valeur au sens du code.

1.1.2 DecaParser.g4

Le fichier DecaParser.g4 est en charge de la définition du Parser, ou analyseur syntaxique. Il a pour rôle de reconnaître les "phrases" du langage. Il utilise la suite de lexèmes envoyés par le lexer et détermine quelle règle syntaxique s'applique à la phrase analysée. Le parser va répéter la même opération sur toutes les "phrases" pour pouvoir construire un arbre abstrait. Voici l'arbre abstrait correspondant au code figure 2 :

```
`> [1, 0] Program
   +> ListDeclClass [List with 0 elements]
   `> [1, 0] Main
      +> ListDeclVar [List with 1 elements]
      |  []> [2, 8] DeclVar
      |      +> [2, 4] Identifier (int)
      |      +> [2, 8] Identifier (c)
      |      `> [2, 8] Initialization
      |         `> [2, 12] Int (2)
      `> [3, 4] ListInst [List with 1 elements]
         []> [3, 4] Println
             `> ListExpr [List with 1 elements]
                []> [3, 13] Identifier (c)
```

Figure 4: Arbre généré par le parser

Cet arbre est ensuite destiné à être décoré et vérifié par l'analyse contextuelle.

1.2 Tree

1.2.1 AbstractInst

La classe AbstractInst a pour rôle de gérer les instructions.

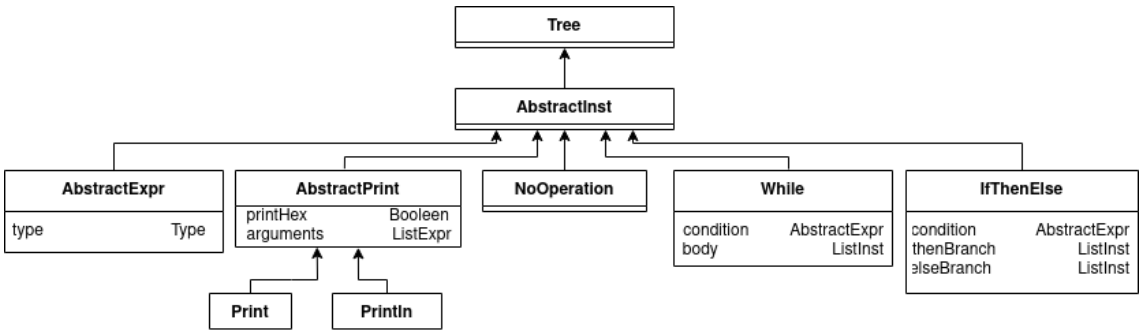


Figure 5: Hiérarchie des instructions

1.2.1.1 AbstractExpr

AbstractExpr est une classe représentant les expressions du langage. On distingue les expressions des classes (cf. figure 6) et les expressions des opérateurs

(cf. figure 7).

A - Les expressions

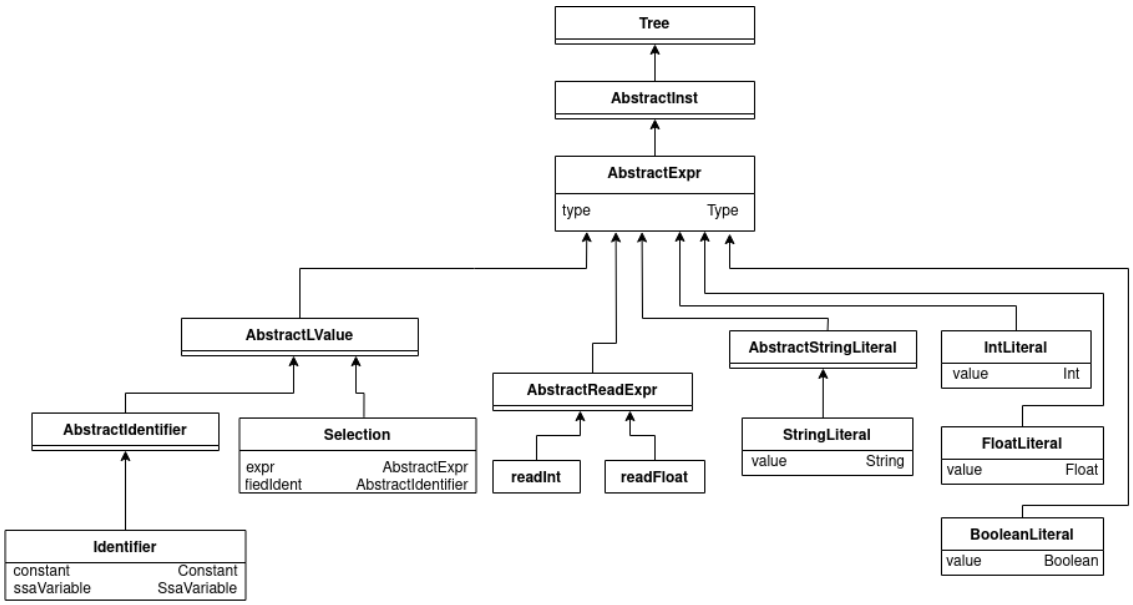


Figure 6: Hiérarchie des expressions

A.1 - AbstractLValue

Cette classe correspond à l'expression à gauche d'une affectation, elle peut être soit un **AbstractIdentifier** soit une **Sélection**. Dans notre compilateur, **AbstractIdentifier** possède une seule classe fille qui est **Identifier**.

Identifier est une classe correspondant à un identificateur ; son constructeur prend deux arguments :

- `constant` : constante remplaçant la variable dans le cas de la propagation de constante (voir doc Extension).
- `ssaVariable` : variable "Single Static Assignment" (SSA) associé à l'identificateur en particulier par le processeur SSA (voir doc Extension).

Selection est un classe correspondant à la sélection d'un champ, utile dans un langage "avec-objet"), son constructeur prend deux arguments :

- `expr` : expression en partie gauche de la sélection.
- `fieldIdent` : Identificateur en partie gauche de la sélection.

A.2 - AbstractReadExpr

Cette classe abstraite a deux classe filles **ReadInt** et **ReadFloat** permettant de lire des valeurs que l'utilisateur entre dans la console :

- **ReadInt** : permet de lire des entiers.
- **ReadFloat** : permet de lire des flottants.

A.3 - Les classes de type "Literal"

Ces classes permet de définir des variables d'un certain type :

- **StringLiteral** hérite de la classe **AbstractStringLiteral** et correspond à une chaîne de caractère.
- **IntLiteral** correspond à un entier.
- **FloatLiteral** correspond à un flottant.
- **BooleanLiteral** correspond à un booléen.

B - Les opérateurs

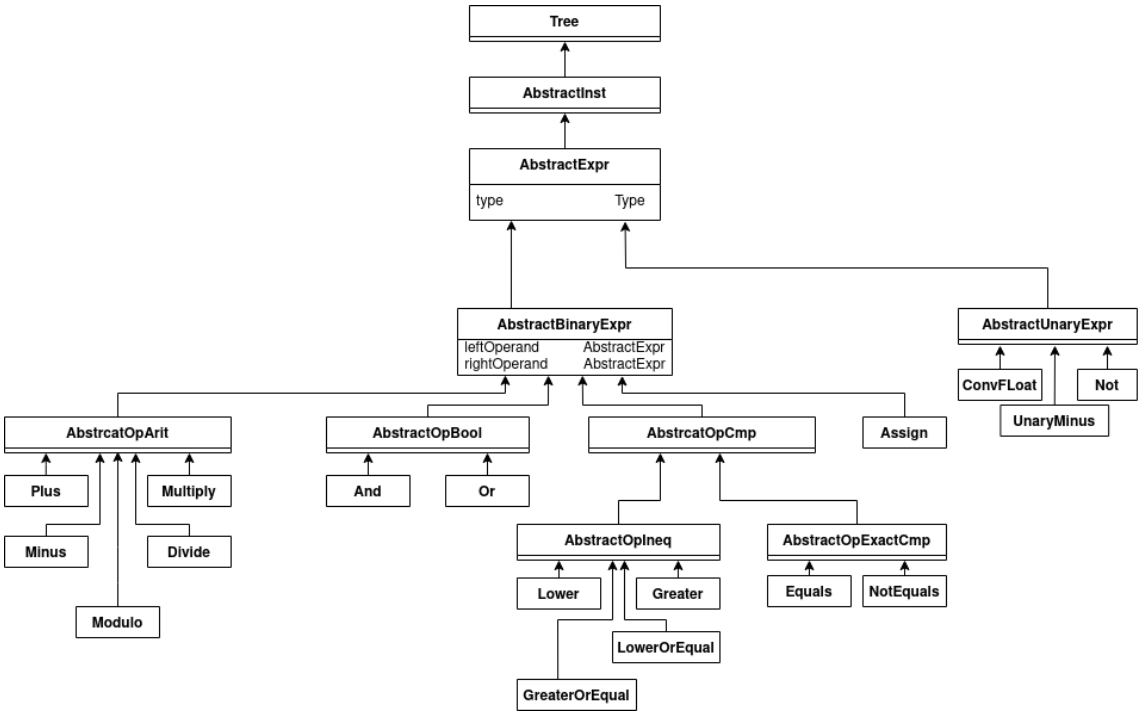


Figure 7: Hiérarchie des opérateurs

B.1 - AbstractUnaryExpr

Cette classe représente l'ensemble des expressions ayant une seule opérande. Elle a plusieurs classes filles : **UnaryMinus**, **Not**, **ConvFloat**.

- **UnaryMinus** : permet d'avoir l'opposé de l'opérande.
- **Not** : permet d'avoir l'inverse de l'opérande booléenne.
- **ConvFloat** : permet de convertir un entier en flottant.

B.2 - AbstractBinaryExpr

Cette classe représente l'ensemble des expressions ayant deux opérandes, une de gauche et une de droite. Elle a plusieurs classes filles : les **AbstractOpArith**, les **AbstractOpBool**, les **AbstractOpCmp** et **Assign**.

AbstractOpArith regroupe les opération arithmétiques :

- **Plus** : permet d'additionner les deux opérandes.
- **Minus** : permet de soustraire l'opérande de droite à l'opérande de gauche.
- **Multiply** : permet de multiplier les deux opérandes.
- **Divide** : permet de diviser l'opérande gauche par l'opérande droite.
- **Modulo** : permet d'avoir le reste de la division euclidienne de l'opérande gauche par l'opérande droite.

AbstractOpBool regroupe les opérations booléennes :

- **Or** : permet de réaliser l'opération booléen "ou" entre les deux opérandes.
- **And** : permet de réaliser l'opération booléen "et" entre les deux opérandes.

AbstractOpCmp regroupe les opération de comparaison suivantes :

- **AbstractOpExactCmp** :
  - **Equals** : égalité
  - **NotEquals** : différence
- **AbstractOpIneq**
  - **Lower** : plus petit que
  - **Greater** : plus grand que
  - **LowerOrEqual** : plus petit ou égal à
  - **GreaterOrEqual** : plus grand ou égal à

Assign permet l'assignation d'une valeur à un identificateur.

1.2.1.2 AbstractPrint

AbstractPrint est une classe qui gère l'affichage d'expressions. Son constructeur prend deux arguments :

- `printHex` : booléen qui indique si l'expression doit être affichée en hexadécimal ou non.
- `arguments` : liste des arguments/expression à afficher

Cette classe abstraite a deux classes filles **Print** et **Println** :

- **Print** : Cette classe permet l'affichage d'une expression sans saut de ligne
- **Println** : Cette classe permet l'affichage d'une expression avec un retour à la ligne à la fin de l'affichage.

1.2.1.3 NoOperation

NoOperation est la classe qui traite une instruction vide.

1.2.1.4 While

While est la classe qui traite l'instruction "while". Son constructeur prend deux arguments :

- `condition` : la condition sur la boucle
- `body` : le corps de la boucle (les instruction à suivre si on passe la condition)

1.2.1.5 IfThenElse

IfThenElse est la classe qui traite l'instruction "if then else". Son constructeur prend trois arguments :

- `conditon` : la condition sur la branche "if"
- `thenBranch` : le corps de la branche "then" (les instructions à suivre si on passe la condition)
- `elseBranch` : le corps de la branche "else" (les instructions à suivre si on ne passe pas la condition).

1.2.2 AbstractInitialization

1.2.2.1 Initialization

Cette classe permet d'initialiser un champ ou une variable. Son constructeur prend un seul argument :

- `expr` : l'expression affectée au champ ou à la variable en question.

1.2.2.2 NoInitialization

Cette classe permet de définir un champ ou une variable sans l'initialiser. Elle n'a donc pas de constructeur.

1.2.3 Lancement des vérifications

1.2.3.1 AbstractProgram

AbstractProgram a une seule classe fille, **Program**.

Program est une classe correspondant à l'ensemble d'un programme. Son constructeur prend deux arguments :

- `classes` : La liste des classes déclarées dans le programme
- `main` : le corps du programme principal

C'est cette classe qui va lancer les différentes passes de vérification contextuelles.

1.2.3.2 AbstractMain

AbstractMain a deux classes filles qui sont **Main** et **EmptyMain**.

Main est une classe correspondant au corps d'un programme. Son constructeur prend deux arguments :

- `declVariables` : La liste des variables déclarées dans le main
- `insts` : la liste des instructions qui sont écrites dans le programme

EmptyMain est une classe correspondant à un corps de programme vide. Par exemple, dans un fichier.deca qui sert uniquement à définir une classe, un EmptyMain sera appelé. Il n'y a donc pas de constructeur pour cette classe.

1.2.3.3 AbstractDeclVar

Cette classe abstraite a une seule classe fille **DeclVar**.

- **DeclVar** permet de déclarer une variable dans le main ou dans le corps d'une classe. Son constructeur prend trois arguments :
  - `type` : le type de la variable
  - `varName` : le nom de la variable
  - `initialization` : la valeur à laquelle la variable est initialisée

1.2.4 Implémentation des objets

Voici un diagramme de classe de l'implémentassions des objets.

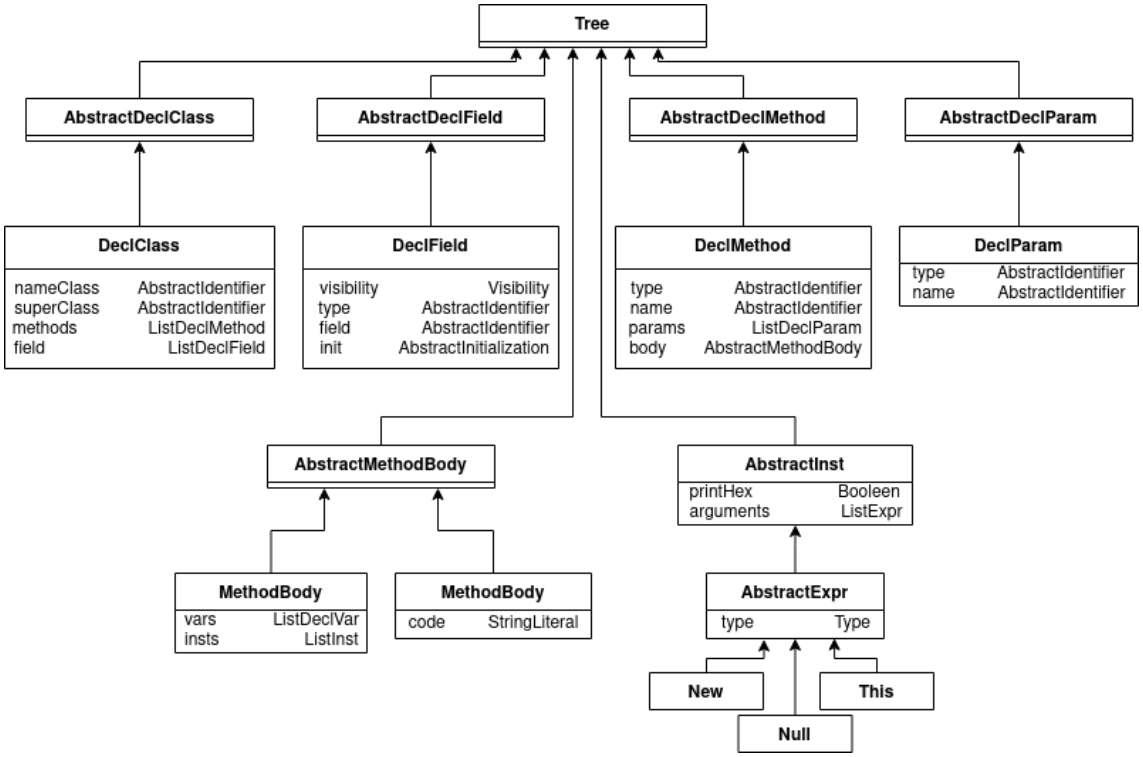


Figure 8: Hiérarchie des classes pour les objets

1.2.4.1 New

Cette classe permet de créer un nouvel Objet. Son constructeur prend un seul argument :

- `type` : type de l'objet à créer.

1.2.4.2 Null

Cette classe permet de spécifier qu'un objet n'a pas de valeur. Il est déréférencé. La classe n'a donc pas de constructeur.

1.2.4.3 This

Cette classe permet de déclarer des champs dans une classe. Son constructeur prend un seul argument :

- `implicit` : booléen qui indique si "this" est présent de manière implicite ou non.

1.2.4.4 Visibility

Visibility est un type énuméré. Cette classe répertorie la visibilité possible pour les variables déclarées. Seules les visibilités "public" et "protected" ont été implémentées.

1.2.4.5 AbstractDeclClass

Cette classe abstraite a une seule classe fille **DeclClass**.

- **DeclClass** permet de déclarer une classe. Son constructeur prend quatre arguments :
  - `nameClass` : nom de la classe
  - `superClass` : nom de la classe mère
  - `methods` : liste des méthodes présentes dans la classe
  - `field` : liste des champs présents dans la classe

1.2.4.6 AbstractDeclCField

Cette classe abstraite a une seule classe fille **DeclField**.

- **DeclField** permet de déclarer un champ dans une méthode. Son constructeur prend quatre arguments :
  - `visibility` : visibilité du champ ("public" ou "protected")
  - `type` : type du champ
  - `field` : nom du champ
  - `init` : valeur à laquelle on initialise le champ

1.2.4.7 AbstractDeclMethod

Cette classe abstraite a une seule classe fille **DeclMethod**.

- **DeclMethod** permet de déclarer une méthode dans une classe. Son constructeur prend quatre arguments :
  - `type` : visibilité du champ ("public" ou "protected")
  - `name` : nom de la méthode
  - `params` : nom des attributs utiles à l'appel de la méthode
  - `body` : le corps de la méthode

1.2.4.8 AbstractMethodBody

Cette classe abstraite a deux classes fille **MethodBody** et **MethodAsmBody**.

- **MethodBody** permet de déclarer le corps d'une méthode. Son constructeur prend deux arguments :
  - `insts` : liste des instructions déclarées dans le corps de méthode
  - `vars` : liste des variables déclarées dans le corps de méthode
- **MethodAsmBody** permet de déclarer le corps d'une méthode assembleur:
  - `code` : chaine de caractère représentant l'instruction en code assembleur devant être effectuée.

1.2.4.9 AbstractDeclParam

Cette classe abstraite a une seule classe fille **DeclParam**.

- **DeclParam** permet de déclarer des paramètres lors de la création d'une méthode. Son constructeur prend deux arguments :
  - `type` : le type du paramètre
  - `name` : le nom du paramètre

1.3 Context

Ce dossier contient sous catégories :

- `les environnements` : des dictionnaires qui à tout symbole associent une définition
- `les définitions` : l'ensemble des définitions possibles
- `les types` : l'ensemble des types
- `les erreurs` : renvoyées à la détection d'erreurs contextuelles.
- la signature : une liste de types

Dans le répertoire *src/main/java/fr/ensimag/deca/context/*, le compilateur réalise la vérification contextuelle de l'arbre généré dans le parser. Il est ensuite décoré avant d'être transmis au codegen.

1.3.1 Les environnements

- `EnvironmentExp` : un dictionnaire qui à tout symbole associe une définition
- `EnvironementType` : un dictionnaire qui à tout symbole associe une définition de type

1.3.2 Les définitions

- `ClassDefinition` : définition de classes
- `ExpDefinition` : définition de expressions
- `FieldDefinition` : définition de champs
- `MethodDefinition` : définition de méthodes
- `ParamDefinition` : définition de paramètres
- `TypeDefinition` : définition de types
- `VariableDefinition` : définition de variables

1.3.3 Les types

- `ClassType` : type de classes
- `BooleanType` : type de booléens
- `IntType` : type d'entiers
- `FloatType` : type de flottants
- `VoidType` : type vide
- `NullType` : type null
- `StringType` : type de chaînes de caractères

1.3.4 Les erreurs

- `ContextualError` : erreur de context
- `DoubleExceptionError` : erreur plu spécifique de double définition

1.4 Codegen

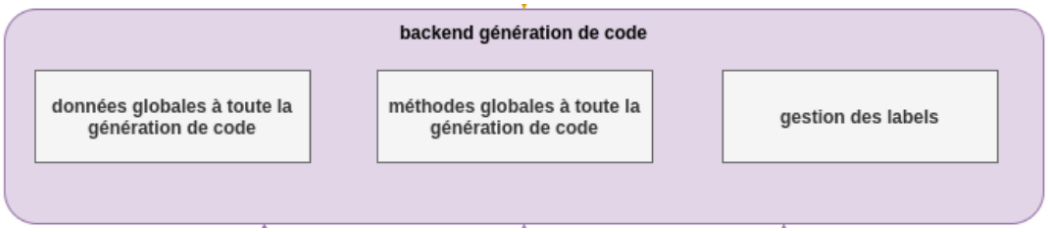
Le répertoire Codegen regroupe les classes nécessaires à la génération du code assembleur. Ces classes prennent en argument l'arbre déjà construit, vérifié et décoré.

1.4.1 Classes "Operation"

Dans le répertoire *src/main/java/fr/ensimag/codegen/*, on retrouve la même architecture que dans le dossier `Tree`, en légèrement simplifiée. En effet, il s'agit de lire la même architecture générée grâce à l'arborescence définie dans `tree`.

Ainsi, la classe non abstraite **BinaryBoolOperation** regroupe les classes abstraites **AbstractOpBool** et **AbstractOpCmp** situées dans le répertoire `tree` car la génération de code assembleur ne nécessite pas de les séparer.

1.4.2 CodeGenBackend



Pour la génération de code assembleur, nous avons décidé d'implémenter une nouvelle classe, le **CodeGenBackend**. Ce backend nous permet de gérer trois aspect de la génération de code.

1.4.2.1 Données globales à la génération de code

Le backend stocke l'ensemble des données nécessaires à la génération de code. C'est-à-dire qu'il permet de retrouver l'architecture totale de l'arbre du code à générer. C'est dans le backend que l'assembleur va chercher les opérations et les opérandes. Le backend est en entrée de toutes les classes **Operation** définies ci-dessous.

1.4.2.2 Méthodes globales à toute la génération de code

Le backend stocke également l'ensemble des méthodes nécessaires à la génération de code. On y trouve les méthodes:

- **contextManager**: qui gère de la gestion du contexte de l'arbre.
- **classManager**: qui gère l'ensemble des classes
- **errorsManager**: gère l'ensemble des erreurs d'exécution
- **startupManager**: qui ...

1.4.2.3 Gestion des labels

Enfin, le backend s'occupe de la gestion des labels. **Bon ben là je sais pas trop quoi mettre mais je fais quand même le tableau**

- **labels**
- **trueBooleanLabel**
- **falseBooleanLabel**

1.5 Optimisation

cf **documentation sur l'optimisation**.

2 Structure du compilateur

2.1 Création, vérification, décoration de l'arbre

2.1.1 Création

La création de l'arbre se fait en deux temps :

- l'analyse lexicale : cette analyse se fait grâce au lexer qui analyse le code "mot par mot".
- l'analyse syntaxique : cette analyse se fait grâce au parser qui analyse les mots du lexer "phrase par phrase" en cherchant quelle règle du parser est suivie par la liste de mots.

Le fonctionnement du lexer et du parser a été expliqué plus tôt dans le paragraphe 1.1

2.1.2 Vérification contextuelle

La vérification contextuelle consiste à regarder si le code répond bien aux critères imposés par la grammaire du langage. Celle-ci se fait en respectant une structure d'arbre dont la racine est au niveau du fichier src/main/java/fr/ensimag/deca/DecaCompiler. Deux environmentType y sont créés : un pour y mettre les types prédéfinis et l'autre pour les types prédéfinis et les classes qui seront déclarés.

La vérification suit naturellement la grammaire du langage : elle se fait par un parcours de l'arbre de la racine aux feuilles dont les fonctions verify de chaque noeud appellent celle(s) du/des noeud(s) qui sui(ven)t.

Les fonctions verify retournent un type et suivent un schéma pré-établi :

- récupération des sous-noeuds si besoin
- vérification de l'existence du type ou des types (si noeuds il y a) et des conditions qui lui sont imposées (non vide)
- vérification du nom de l'objet (ne doit pas être un nom de types prédéfinis, de classes, et de méthodes, champs, variables, paramètres dans les cas concernés.
- ajout du couple objet/définition à l'environnement qui le contient
- configuration de la définition et type du noeud en question (ce qui décore l'arbre au passage)
- retourne le type

2.2 Génération du code assembleur

Maintenant que l'arbre décoré a été généré, il faut récupérer chaque feuille de cet arbre avec sa décoration afin de générer un code assembleur. C'est la dernière étape du compilateur ; elle est primordiale puisqu'elle permet en quelque sorte de valider les étapes précédentes : c'est elle qui permet d'obtenir un code fonctionnel.

La génération de code est appelée récursivement dans l'arbre décoré par chacun des noeuds jusqu'aux feuilles ; pour générer le code, plusieurs composantes seront appelées afin de gérer spécifiquement les différents morceaux de la génération. Pour cela, une architecture en deux parties principales qui interagissent est utilisée : un "back-end" et un "front-end".

2.2.1 Instructions assembleur

Nous avons à disposition un ensemble d'instructions assembleur déjà fournies et codées. Elles se trouvent dans le répertoire *src/main/java/fr/ensimag/ima/pseudocode/instructions*. Ce sont ces instructions que nous allons appeler et utiliser pour la génération du code dans la méthode **addInstruction()**.

2.2.2 Lecture et compréhension de l'arbre décoré

Avant de générer le code assembleur, il faut d'abord lire l'arbre en entrée. Cette lecture s'effectue dans le répertoire *src/main/java/fr/ensimag/deca/codegen*. Pour cela, on utilisera une architecture similaire à celle que l'on trouve dans le répertoire *src/main/java/fr/ensimag/deca/tree*, mais en légèrement simplifiée. En effet, nous avons décidé de fusionner certaines classes qui pouvaient l'être, comme la classe **BinaryBoolOperation** qui regroupe le rôle des classes **AbstractOpBool** et **AbstractOpCmp**.

2.2.3 Appel des opérateurs en assembleur

Pour exécuter le code, nous utilisons le langage du plus bas niveau que nous connaissons : l'assembleur. Pour cela, nous avons à disposition une bibliothèque d'opérateurs en assembleur que nous devons utiliser en fonction de ce que l'utilisateur a écrit en deca.

Les appels se font dans le répertoire *src/main/java/fr/ensimag/deca/codegen*, au cours de la lecture de l'arbre. Cette méthode prend en argument une instruction assembleur et génère en sortie le code assembleur souhaité.

2.2.4 Code assembleur généré

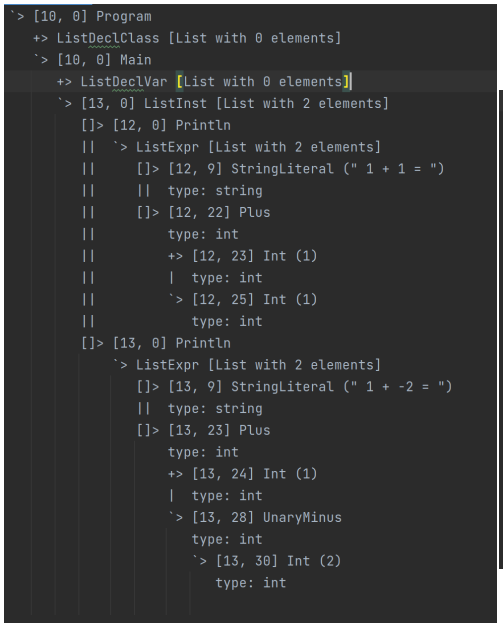
Pour montrer le code assembleur généré, nous allons traiter un exemple très simple. Nous allons traiter du code assembleur généré pour le programme très simple ci-dessous:

```
{
println ("1 + 1 = ", (1+1) );
```

```
println ("1 + -2 = ", (1+(-2)) );
}
```

2.2.4.1 Entrée du générateur de code

Les lignes de code ci-dessus, après avoir été traduites en un arbre décoré (cf. figure 9), sont traduites en code assembleur (cf. figure 10).



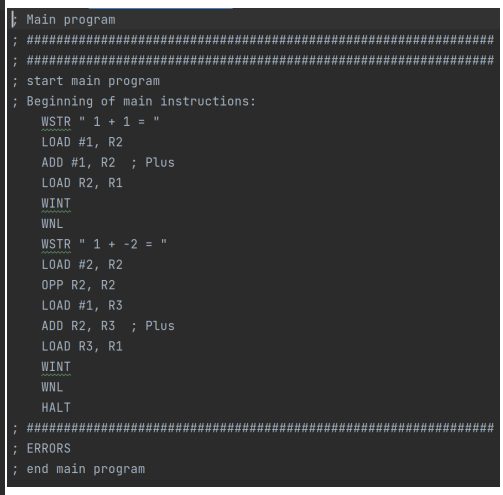


Figure 10: code assembleur généré

Figure 9: Arbre en entrée du générateur de code

Il s'agit du seul lien entre le programme écrit et le générateur de code assembleur. Cet arbre est syntaxiquement et contextuellement correct (il a déjà été vérifié par de nombreuses passes et le générateur de code assembleur ne doit pas s'occuper de ce genre d'erreurs). A la lecture de cet arbre, et grâce à l'architecture expliquée ci-dessus, il devra générer le code assembleur.

2.2.4.2 Code assembleur généré

Le générateur de code va donc lire l'arbre de manière récursive, jusqu'à ce que les branches soient vides. Il génère ainsi au fur et à mesure le code assembleur nécessaire à l'exécution du programme. Pour l'exemple proposé, le code assembleur généré correspond à la figure 10. Le travail du compilateur est maintenant presque terminé, le programme est correct, généré en assembleur et l'ordinateur peut l'exécuter. Il ne reste que la gestion des erreurs d'exécution (les exceptions d'exécution).

3 Gestion et allocation de la mémoire

3.1 Utilisation de registres

Dans l'architecture du processeur ima, nous disposons de 16 registres (qui vont de R_0 à R_{15}). Les registres R_0 et R_1 sont des registres temporaires : leurs valeurs ne sont pas préservées par les appels de méthodes. Il ne faut donc pas d'attendre à retrouver ces valeurs après un appel de méthode. Nous avons choisi de créer des registres virtuels, stockés sur la pile. L'implémentation de ces registres virtuels dans des classes à part nous permet de ne plus nous occuper de l'allocation des registres et de les appeler sans se soucier d'en avoir des disponibles.

3.2 Gestion des registres

Les registres sont gérés par la classe **Contextmanager**. Ce gestionnaire va pouvoir créer des registres virtuels ou utiliser les registres physiques si le code en a absolument besoin.

3.2.1 Les registres virtuels

Dès qu'une opération est effectuée, le gestionnaire de contexte utilise la méthode `requestNewRegister` pour créer un nouveau registre virtuel ; le registre sera vide ou stockera l'immédiat qui est passé en paramètre de la fonction.

Lorsqu'un registre virtuel n'est plus utilisé, le gestionnaire de contexte le libère à l'aide de la méthode `destroy`.

3.2.2 Les registres physiques

Pour effectuer une opération, ima a besoin d'un registre physique sur lequel effectuer cette opération. Le gestionnaire de contexte va alors appeler explicitement un registre physique avec la fonction `requestPhysicalRegister`. S'il n'y a aucun registre de libre, le contenu d'un des registres physiques va être déplacé dans la pile d'exécution ; l'opération va ainsi pouvoir être effectuée sur le registre libéré.

Comme les registres utilisés par les classes des opérations pile sont virtuels, l'utilisateur n'aura aucune action à effectuer pour gérer ce transfert.

3.2.3 Sauvegarde des registres

Lorsqu'une méthode est appelée, il est attendu que les valeurs des registres physiques utilisés soient les mêmes à la fin qu'au début de la méthode ; c'est la classe **StartUpManager** qui s'en charge. Cette classe va, au début du code de la méthode, mettre toutes les valeurs des registres concernés dans la pile, et va les récupérer à la fin du code de la méthode pour les remettre dans les registres physiques.

Avant cette sauvegarde, ima va vérifier que la taille de la pile est suffisante. Si elle ne l'est pas, le **StartupManager** va brancher vers l'erreur de dépassement de pile.