École Nationale Supérieure d'Informatique et Mathématiques Appliquées de Grenoble

Documentation de conception

Equipe 03

Benhachem Youssef
Bouhout Ilyass
Garcia Reyes Conrado Ivan
Lemrabet Soufiane
Klou Anas

Grenoble - France
25 Janvier 2021

1-Introduction :

Le schéma UML suivant représente l'ensemble d'architecture de notre compilateur en montrant les différentes classes et leur dépendances . Pour chaque entité, l'alphabet décrit le type de cette dernière : A pour abstract , C pour classe , E pour enum , I pour interface .Pour les éléments de chaque entité (méthodes ou attributs) , on désigne par un cercle que l'élément est public et par carreau qu'il privé .Les différentes dépendances entres les différentes entitées ont été modélisé par des flèches en convention UML : une flèche en pointillé modélise que la sous entité IMPLEMENT l'entité mère , alors qu'une simple flèche désigne que la sous entité EXTENDS l'entité mère .

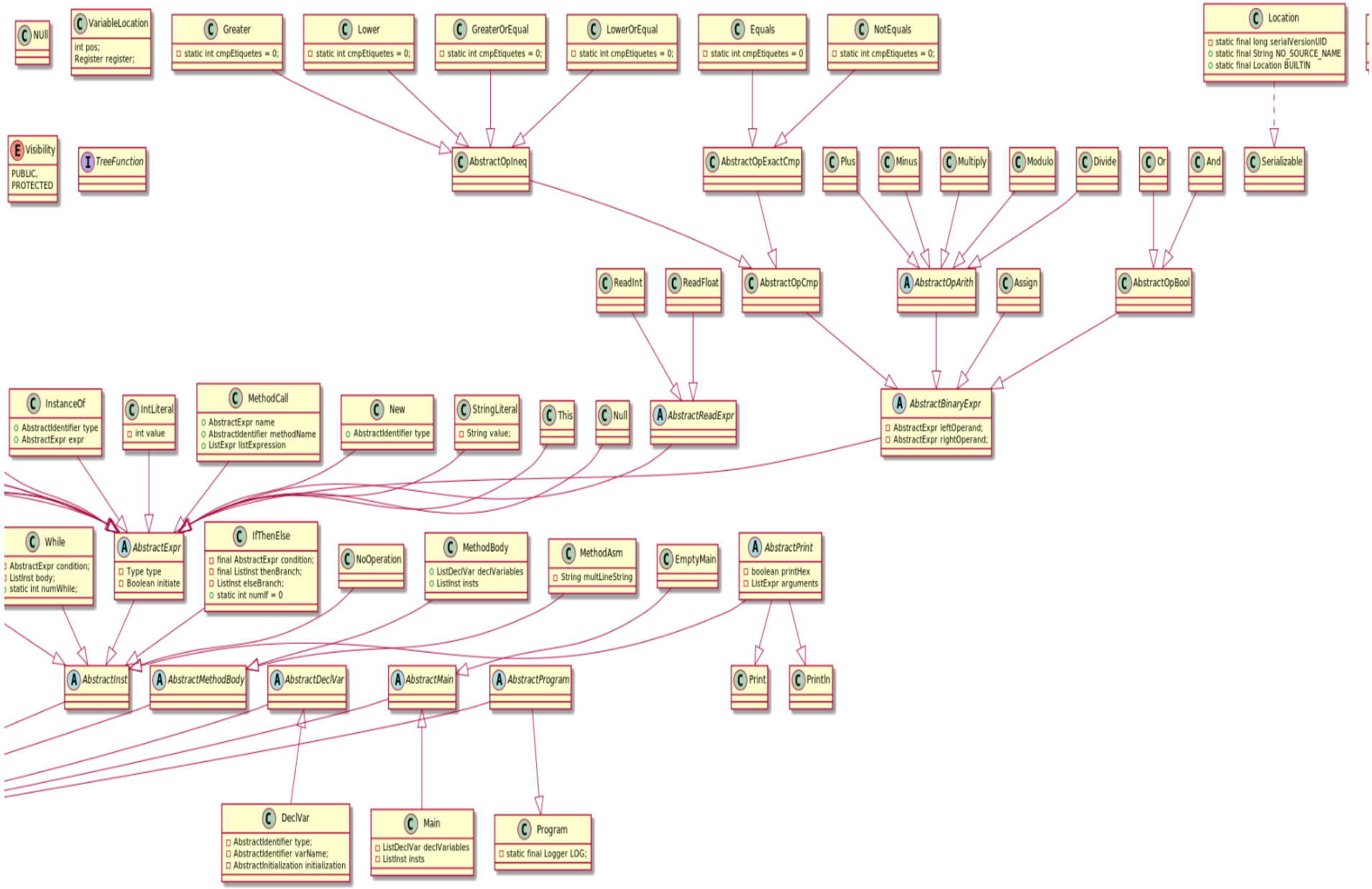
2-Conception et Implémentation de l'analyse syntaxique et lexical :

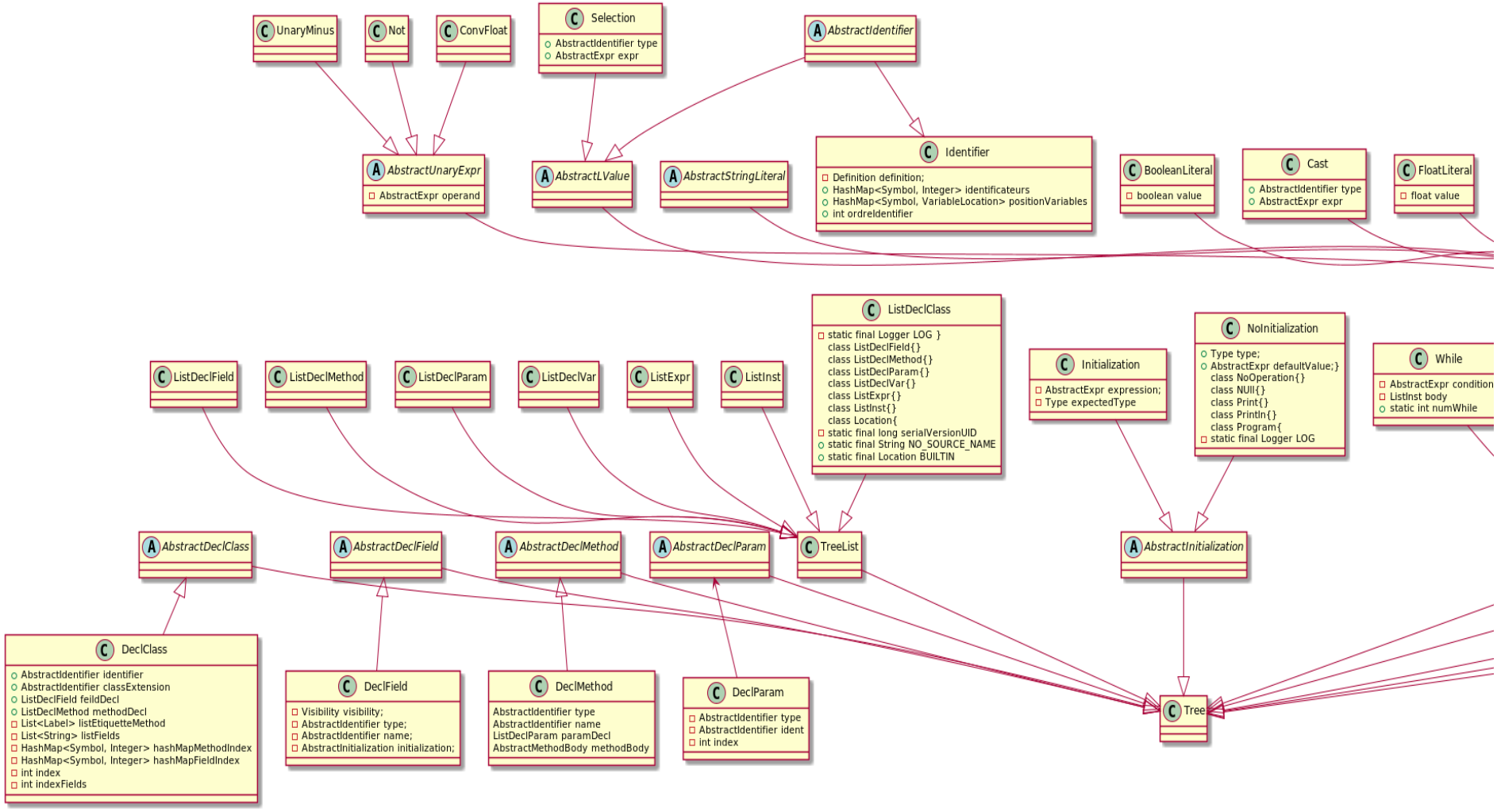
L'implémentation de l'analyse syntaxique et lexical nous a mené à créer quelques classes : AbstractDeclFeild, AbstarctDeclMethod, AbstractDeclParam.DeclFeild, DeclParam,DeclMethod qui héritent de leurs Abstract . D'autres classes ont été ajoutées : ListDeclMethod, ListDeclParam et ListDeclFeild qui contiennent une liste de méthodes, de paramètres ou de fields .

Pour l'appel des méthodes , les classes MethodCall , MethodBody et MethodAsm permettent d'assurer cette fonctionnalité .

L'accès à des attributs d'un objet est géré par la classe Sélection qui hérite de AbstractLValue.

Les classes This, Null , Return et instanceOf ont été créées lors de l'implémentation de la partie A


NULL
VariableLocation
int pos;
Register register;
Greater
static int cmpEtiquetes = 0;
Lower
static int cmpEtiquetes = 0;
GreaterOrEqual
static int cmpEtiquetes = 0;
LowerOrEqual
static int cmpEtiquetes = 0;
Equals
static int cmpEtiquetes = 0
NotEquals
static int cmpEtiquetes = 0;
Location
static final long serialVersionUID
static final String NO_SOURCE_NAME
static final Location BUILTIN
Visibility
PUBLIC,
PROTECTED
TreeFunction
AbstractOpIneq
AbstractOpExactCmp
Plus
Minus
Multiply
Modulo
Divide
Or
And
Serializable
ReadInt
ReadFloat
AbstractOpCmp
AbstractOpArith
Assign
AbstractOpBool
InstanceOf
AbstractIdentifier type
AbstractExpr expr
IntLiteral
int value
MethodCall
AbstractExpr name
AbstractIdentifier methodName
ListExpr listExpression
New
AbstractIdentifier type
StringLiteral
String value;
This
Null
AbstractReadExpr
AbstractBinaryExpr
AbstractExpr leftOperand;
AbstractExpr rightOperand;
While
AbstractExpr condition;
ListInst body;
static int numWhile;
AbstractExpr
Type type
Boolean initiate
IfThenElse
final AbstractExpr condition;
final ListInst thenBranch;
ListInst elseBranch;
static int numIf = 0
NoOperation
MethodBody
ListDeclVar declVariables
ListInst insts
MethodAsm
String multLineString
EmptyMain
AbstractPrint
boolean printHex
ListExpr arguments
AbstractInst
AbstractMethodBody
AbstractDeclVar
AbstractMain
AbstractProgram
Print
Println
DeclVar
AbstractIdentifier type;
AbstractIdentifier varName;
AbstractInitialization initialization
Main
ListDeclVar declVariables
ListInst insts
Program
static final Logger LOG;


UnaryMinus
Not
ConvFloat
Selection
AbstractIdentifier type
AbstractExpr expr
AbstractIdentifier
AbstractUnaryExpr
AbstractExpr operand
AbstractLValue
AbstractStringLiteral
Identifier
Definition definition;
HashMap<Symbol, Integer> identificateurs
HashMap<Symbol, VariableLocation> positionVariables
int ordreIdentifier
BooleanLiteral
boolean value
Cast
AbstractIdentifier type
AbstractExpr expr
FloatLiteral
float value
ListDeclClass
static final Logger LOG }
class ListDeclField{}
class ListDeclMethod{}
class ListDeclParam{}
class ListDeclVar{}
class ListExpr{}
class ListInst{}
class Location{
static final long serialVersionUID
static final String NO_SOURCE_NAME
static final Location BUILTIN
Initialization
AbstractExpr expression;
Type expectedType
NoInitialization
Type type;
AbstractExpr defaultValue;}
class NoOperation{}
class NUll{}
class Print{}
class PrintIn{}
class Program{
static final Logger LOG
While
AbstractExpr condition
ListInst body
static int numWhile
ListDeclField
ListDeclMethod
ListDeclParam
ListDeclVar
ListExpr
ListInst
AbstractDeclClass
AbstractDeclField
AbstractDeclMethod
AbstractDeclParam
TreeList
AbstractInitialization
DeclClass
AbstractIdentifier identifier
AbstractIdentifier classExtension
ListDeclField feildDecl
ListDeclMethod methodDecl
List<Label> listEtiquetteMethod
List<String> listFields
HashMap<Symbol, Integer> hashMapMethodIndex
HashMap<Symbol, Integer> hashMapFieldIndex
int index
int indexFields
DeclField
Visibility visibility;
AbstractIdentifier type;
AbstractIdentifier name;
AbstractInitialization initialization;
DeclMethod
AbstractIdentifier type
AbstractIdentifier name
ListDeclParam paramDecl
AbstractMethodBody methodBody
DeclParam
AbstractIdentifier type
AbstractIdentifier ident
int index
Tree

2-Implémentation de l'analyse contextuelle :

En général, nous avons essayé d'implémenter directement ce que nous avons décrit lors de la phase de la conception.En faite, pour tout ce qui est en liaison avec l'analyse contextuelle nous avons procédé par le principe implémenter et tester, c'est à dire après l'implémentation de chaque partie on lance un seul test qui permettra de corriger les erreurs remontées ou de passer à l'étape suivante. Pour le codage de cette partie, nous avons essayé d'implémenter les différentes méthodes de vérification contextuelle spécifique à une règle de grammaire. Les méthodes qui occupent ce rôle sont les "verifyExpr" dans laquelle on compare conformité de la séquence ou l'expression retournée par le Parser avec le résultat attendu par la grammaire.

Pour la mise en oeuvre des environnements des expressions et l'environnement des types, nous avons choisi de travailler avec des HashMap qui présente des associations entre symboles(le nom du type/var/méthode) avec leurs définitions.chaque classe a son propre environnement d'expression, indépendant des autres classes inclus ses classe meres, dans lequel les attributs et les méthodes sont stockés. Le choix des hashMap est justifié par le fait que la complexité d'accès et de récupération d'un élément par sa clé est constante, et donc l'implémentation des méthodes qui font l'empilement et l'union disjointe devient plus manipulable. Il faut mentionner aussi que chaque fois qu'une classe est déclarée, on stocke son association avec sa définition dans un dictionnaire indépendant de celui des attributs et les méthodes.Donc, si nous voudrions par exemple accéder à un attribut dans une classe mère, on récupère d'abord le nom de la super classe à partir de la définition de la classe courante, qui est stockée dans l'environnement des types, et puis on accède à l'environnement de cette super classe pour chercher l'attribut en question. Nous avons évité de travailler avec des listes chaînées, car si on voulait, par exemple, accéder à une méthode qu'on connaît qu'il est dans la super classe de la super classe il faudrait passer par l'environnement la classe mère. Une autre utilité des hashMap apparaît lors de la vérification du corps d'une méthode. En effet, pour empiler la liste des paramètres de la méthode à l'environnement de la classe il suffit de faire un simple addAll pour réussir à faire un empilement avec les collection des HashMap.

Afin d'attribuer à chaque méthode un indice, nous parcourons toutes les classes mère de la classe courante et si cette méthode est déjà existante on récupère son indice et puis on lui attribue à la méthode en traitement sinon on

lui attribue le nombre de méthodes que possède cette classe en incluant celles héritées et puis on incrémente le nombre de méthodes.Mais, pour les indices des champs nous avons procédé différemment, en faite, on leurs attribue leurs ordres de déclaration dans la classe.

2-Implémentation de la génération du code :

Pour la grande majorité de cette partie , on a décidé d'adapter les algorithmes proposés que ce soit pour la génération de code des opérations arithmétiques, booléennes ou pour la génération de code de la partie Objet.

Pour l'implémentation de la partie de génération de code, nous nous sommes mis d'accord sur l'utilisation de régistre R2 pour déposer le résultat final. Ce choix vient du fait de minimiser le nombre de cycles faites par le processeur et donc de minimiser la consommation au niveau énergétique et le temps d'exécution sera bien évidemment moins petit. Mettre le résultat des expressions dans le même registre va diminuer aussi le nombre d'instruction en assembleur. Dans les méthodes ***CodeGenExpr*** pour les opérations arithmétiques le résultat est mis par défaut dans l'opérande gauche et donc on profite de cela pour augmenter l'efficacité de notre compilateur, le paramètre "int n"que prend cette méthode sert juste à nous faciliter compréhension de l'algorithme implémenté.

1 - Le Sans Objet:

1.1- Pour les opérations Arithmétiques:

Le travail de gestion de registre et implémentation en niveau de la classe `AbstractOpArith`. En effet, la méthode `CodeGenInst` stocke la valeur de l'opérande droite dans un registre `R` et celle de l'opérande gauche dans `R(n+1)`, si n est inférieure au nombre maximal de registre déterminé par l'option "-r X" ou par défaut 15, sinon on profite de la pile pour que n'utilise que les registres `R0 et Rn`. L'avantage que nous donne l'utilisation de `Rn et Rn+1` c'est que lors la récupération et le stockage des valeurs coûte 2 cycles pour `LOAD` et deux autres pour le `STORE`, mais pour la pile le `PUSH` toute seule coûte quatres cycles et le `POP` deux cycles. Pour profiter de cet avantage il est conseillé de diviser l'opération à calculer en de petites opérations.

1.2- Pour les opérations d'affectation : `Assign.java`

La classe assign qui héritent de `AbsractBinaryExpr` et à l'aide de la méthode `codeGenInst`(DecacCompiler compiler) permet de de générer le code assembleur d'une affectation en évaluant la partie droite d'une affectation à l'aide d'un appel récursif sur la partie droite, alors que le résultat d'évaluation de partie gauche est mis dans le registre `R2.`

1.3- Pour les expressions conditionnelles : `while.java, ifthenelse.java`

La génération de code pour les expressions conditionnelles est basée sur le branchement entre les différentes étiquettes `(début , body et fin)`. L'implémentation de la méthode codeGenInst fait appel à la méthode `codeGenOpBool` sur l'attribut condition qui permet d'évaluer la condition du `while` ou du `ifThenElse` et de faire un branchement conditionnel vers l'étiquette qui exécute les instruction du If/while ou bien elle saute vers l'étiquette `finwhile / else` . Pour gérer l'appel récursive de plusieurs boucles conditionnelles , on déclare une variable statique qui permet de distinguer entre les différentes étiquettes des différents appels.

1.4- Pour les opérations de comparaison :

Dans cette partie on était face à deux problèmes l'un l'appel d'une opération de comparaison comme étant une condition dans une boucle while/if , ou lors d'une instruction d'affectation ou d'évaluation de valeur :

- Pour l'appel d'une opération de comparaison , on a décidé de garder la méthode `codeGenOpBool` décrit plus haut tout en évaluant la partie droite et de mettre le résultat dans le registre R2 puis la partie gauche et de mettre le résultat de cette dernière dans le registre R3 , ainsi on compare le résultat des deux registres à l'aide de l'instruction d'assembleur CMP afin de faire un branchement vers l'étiquette qui convient .

- Pour l'appel d'une affectation , par exemple `b = true == true` , c'est la méthode `codeGenInst` qui génère le code assembleur correspondant en mettant la valeur du résultat dans le registre R2 (Dans notre exemple la fonction mettra la valeur #1 dans le registre R2) .

1.5- Pour les expressions booléennes :

Pour le codage des expressions booléennes, on a décidé de coder ces derniers par les flots de contrôle à l'aide du branchement cela permet de factoriser le code à l'aide de la méthode `codeGenOpBool` décrit plus haut . Le principe de cette approche consiste à évaluer la partie droite de l'expression et de considérer l'expression comme une suite de lignes de code comportant un ou plusieurs branchements à une certaine étiquette E

2- La Partie Objet :

La conception d'une grande majorité de cette partie se base sur les algorithmes définis dans le polycopié page 210 :

- **Pour la création de la table des méthodes :**

La création de la table des méthodes est faite dans la classe `DeclClass.java` , on utilise dans cette class deux structures de données , un `treemap` entre les labels et leurs indices , et un `hashmap` qui permet de stocker l'association entre un symbol et un integer représentant , la méthode `createLabelList` permet d'initialiser le `treemap` en associant à chaque label sous la forme "`code.NomDeLaClasse.NomDeLaMethode`" à son indice approprié qu'on obtient à l'aide de la méthode `getIndex()` de la classe `MethodField` . Pour chaque déclaration de classe , un appel à la méthode createLabelList créer la table des méthodes appropriée en parcourant les environnements des classes mères.

- **Pour les paramètres :**

Pour chaque paramètre on associe un indice à partir de -3 qui correspond à son indice dans la pile LB , et on le stocke dans un hashmap spécifique pour la pile Lb qui associe à chaque symbol son indice dans la pile LB .Pour récupérer l'indice associé à un paramètre on fait appel à la fonction `getIndexParam(getName())` .

- **Pour les variables et les champs :**

Le traitement des variables et des champs est similaire au traitement des paramètres . La méthode `getVariableAdress(getName())` permet de renvoyer la position des variables dans la pile GB .

- **Pour les méthodes qui renvoient un résultat :**

Si la méthode renvoie un résultat de type différent de void autrement que par une instruction return on obtient un message d'erreur , sinon le résultat du return est enregistré dans le registre R0.