**Listener: analyseur lexicale** **Parser: analyseur syntaxique**
si la **derniere** expression de **.ilpml** ne retourne pas de valeur, on met **false** dans le fichier **result**

on crée ASTC => si on a un appel au Function , Variable

**POUR LE PARTIEL:**

Tout noeud **ASTC** doit **hériter** de **AST** du noeud. + on crée un **ASTCvisitor.java**

**compiler/Normalizer.java:** (Construction d'un ASTC à partir de l'AST)
il **visite les noeuds**, pour chaque un il **visit/accepte** ses composants puis il **return** une noeud crée par **NormalizationFactory du C**

**compiler/GlobalVariableCollector.java** (**analyse statique des variables globales visite)**:
extends ilp.compiler.globalVarcollector ------- implements IASTCvisitor du partiel
-Si je visite un **IASTtri** et **IASTCtri** existe: je teste si **IASTtri** est une instance de **IASTCtri**.
-visiter les noeuds, et pour chaque un on **accepte** les **expressions** - (**result** du paramètre sera **enrichi** avec les composants qu'on a **accepté**).

**compiler/FreeVariableCollector.java (analyse des variables libres)** : si je tombe sur un **IASTC**, je visite **IAST**.
extends ilp.compiler.FreeVarcollector ------- implements IASTCvisitor du partiel
- Dans le visiteur de **IAST**, si j'ai une **expression** je l'accepte, **sinon rien**.
- return **null**;

----------→

**Recherche de la valeur d'une variable** :
• recherche en priorité dans **l'environnement lexical**
• puis dans **l'environnement global** (si non trouvé)
• puis signalement d'erreur (si toujours pas trouvé) (l'erreur est signalée par getGlobalVariableValue et propagée)

updateGLobalVariableValue – a utiliser en ILP2

## - Intoduire une nouvelle structure de données:

Compilateur:
1. Ajouter la definition a **ILP_Object** dans le fichier **ilp.h**.
2. Définition de la classe en **ilp_nomStructure.c** et
3. Création de **ilp_nomStructure.h** et déclarer la fonction de l'allocation + les fonction de **ilp_nomStructure.c**
4. Déclaration des fonction qui retournent un **ILP_Object**, comme **ilp_make_vector**(ILP_Object o1) et faire la vérification des paramètres.

5. **dans le Compilateur**:
1. introduire **Compiler.java** qui va inclure les includes vers les fichiers **.h**
2. introduire **GlobalVariableStuff.java** pour renomer la fonction crée en **ilp_nomStructure.c** en leur donnant un nom lisible par l'utilisateur comme "**makeVector**" au lieu de "**ilp_make_vector**".
3. creation du fichier **CompilerTest.java** en changeant scriptCommand + Compiler + GlobalVariableStuff

6. **dans l'interpreteur**:
    1. pour chaque fonction, créer une classe qui etendre **Primitive** qui contient le nom finale, + **ecrire la logique** dedans.
    2. définir la fonction dans le fichier **GlobalVariableStuff.java**
    3. modifier **InterpreterTest** en changeant la reference de GlobalVariableStuff

## ILP 4:

**INTERPRETER:** je visite un noeud **AST** mais quand je le traite, j'accepte ses champs et puis soit je retourne **body.accept()** ou bien un Objet d'une classe que je **définis** dans le répertoire **interpreter**.

**IClassEnvironment.java** et **ClassEnvironment.java**: stocke les classes dans une table global (**clazzes**) avant l'exécution.

Super: - Ne prend pas d'argument, il utilise ceux de la méthode courante.

**ISuperCallInformation.java** : Sauvegarde les information de la méthode parent (args et getSuperMethod)

**Verification** si un champ d'une AST est une instance d'une classe on utilise "**i instanceof ILPInstance**"

**les variables var1 à varN sont locales à expr1**: -- -- -- -- -- --

```
ILexicalEnvironment lexenv2 = lexenv;
IASTvariable[] vars = iast.getVariables();
for (int i = 0; i < vars.length; i++) {
        Object v = vars[i].accept(this, lexenv2);
        lexenv2 = lexenv2.extend(vars[i], v);
}

return iast.getConsequence().accept(this, lexenv2);
```

Fonctiondefinition qui change,

```
public class ASTcountingFunctionDefinition extends ASTfunctionDefinition implements IASTcountingFunctionDefinition

public interface IASTcountingFunctionDefinition extends IASTfunctionDefinition {
        boolean isCounting();
}
```

in Compiler:

```
public interface IASTCcountingFunctionDefinition extends IASTcountingFunctionDefinition, IASTCfunctionDefinition {

public class ASTCcountingFunctionDefinition extends ASTCfunctionDefinition implements
IASTCcountingFunctionDefinition
```

`@OrNull`: dans l'interface avant la fonction, dans l'AST : `private @OrNull final IASTexpression defaultExpr;`