Code ILP4

Christian.Queinnec@lip6.fr

27 août 2013

Table des matières

2 — Grammars/grammar4.rnc
4 — Java/src/fr/upmc/ilp/ilp4/Process.java
6 — Java/src/fr/upmc/ilp/ilp4/XMLProcess.java
6 — Java/src/fr/upmc/ilp/ilp4/test/ProcessTest.java
7 — Java/src/fr/upmc/ilp/ilp4/test/WholeTestSuite.java
7 — Java/src/fr/upmc/ilp/ilp4/interfaces/AbstractExplicitVisitor.java
9 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4.java
9 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4Factory.java
11 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4alternative.java
11 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4assignment.java
11 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4binaryOperation.java
11 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4boolean.java
11 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4computedInvocation.java
12 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4constant.java
12 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4delegable.java
12 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4expression.java
13 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4float.java
13 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4functionDefinition.java
13 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalAssignment.java
13 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalFunctionVariable.java
13 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalInvocation.java
14 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalVariable.java
14 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4instruction.java
14 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4integer.java
14 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4internal.java
15 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4invocation.java
15 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4localAssignment.java
15 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4localBlock.java
15 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4localVariable.java
15 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4operation.java
15 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4primitiveInvocation.java
16 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4program.java
16 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4reference.java
16 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4sequence.java
16 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4string.java
17 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4try.java
17 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4unaryBlock.java
17 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4unaryOperation.java
17 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4variable.java
17 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4visitable.java
18 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4visitor.java
18 — Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4while.java
18 — Java/src/fr/upmc/ilp/ilp4/interfaces/INormalizeGlobalEnvironment.java
19 — Java/src/fr/upmc/ilp/ilp4/interfaces/INormalizeLexicalEnvironment.java
19 — Java/src/fr/upmc/ilp/ilp4/interfaces/IParser.java
19 — Java/src/fr/upmc/ilp/ilp4/ast/CEAST.java
25 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTFactory.java

29 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTParser.java
31 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTalternative.java
33 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTassignment.java
35 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTbinaryOperation.java
36 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTboolean.java
36 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTcomputedInvocation.java
38 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTconstant.java
38 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTdelegableExpression.java
39 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTdelegableInstruction.java
40 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTexpression.java
41 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTfloat.java
41 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTfunctionDefinition.java
44 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalAssignment.java
45 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalFunctionVariable.java
46 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalInvocation.java
48 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalVariable.java
50 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTinstruction.java
50 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTinteger.java
50 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTinvocation.java
52 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTlocalAssignment.java
53 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTlocalBlock.java
55 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTlocalVariable.java
57 — Java/src/fr/upmc/ilp/ilp4/ast/CEASToperation.java
57 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTprimitiveInvocation.java
59 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTprogram.java
63 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTreference.java
64 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTsequence.java
65 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTstring.java
66 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTtry.java
67 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTunaryBlock.java
69 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTunaryOperation.java
71 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTvariable.java
72 — Java/src/fr/upmc/ilp/ilp4/ast/CEASTwhile.java
74 — Java/src/fr/upmc/ilp/ilp4/ast/FindingInvokedFunctionsException.java
74 — Java/src/fr/upmc/ilp/ilp4/ast/GlobalCollector.java
75 — Java/src/fr/upmc/ilp/ilp4/ast/InliningException.java
75 — Java/src/fr/upmc/ilp/ilp4/ast/NormalizeException.java
75 — Java/src/fr/upmc/ilp/ilp4/ast/NormalizeGlobalEnvironment.java
76 — Java/src/fr/upmc/ilp/ilp4/ast/NormalizeLexicalEnvironment.java
77 — Java/src/fr/upmc/ilp/ilp4/ast/XMLwriter.java
84 — Java/src/fr/upmc/ilp/ilp4/runtime/CommonPlus.java
84 — Java/src/fr/upmc/ilp/ilp4/runtime/LexicalEnvironment.java
86 — Java/src/fr/upmc/ilp/ilp4/runtime/PrintStuff.java
86 — Java/src/fr/upmc/ilp/ilp4/cgen/AssignDestination.java

Grammars/grammar4.rnc

```
# Quatrième version du langage étudié: ILP4 pour « Incongru Langage
# Poilant ». Il sera complété dans les cours qui suivent. La grande
# nouveauté est que (à la différence de C, de Java, de JavaScript,
# etc.) toute instruction est maintenant aussi une expression. Cela
# permet de s'affranchir des différences de syntaxe entre les deux
# types d'alternatives (if-else et ?:) mais autorise le bloc local (ce
# que n'autorise pas C ni JavaScript). Le grand avantage est que cela
# simplifie le code de compilation et permet de parler plus simplement
# d'intégration de fonctions (inlining).

start =
   programme4
 | programme3
 | programme2
 | programme1

# Un programme4 est composé de définitions de fonctions globales
# suivies d'expressions les mettant en ?uvre.

programme4 = element programme4 {
   definitionEtExpressions
}
programme3 = element programme3 {
   definitionEtExpressions
}
programme2 = element programme2 {
   definitionEtExpressions
}
programme1 = element programme1 {
   expression
}

definitionEtExpressions =
   definitionFonction *,
   expression +

# Définition d'une fonction avec son nom, ses variables (éventuellement
# aucune) et un corps qui est une séquence d'expressions.

definitionFonction = element definitionFonction {
   attribute nom     { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   element variables { variable * },
   element corps     { expression + }
}

# Les expressions possibles:

expression =
   alternative
 | sequence
 | blocUnaire
 | blocLocal
 | boucle
 | try
 | affectation
 | invocation
 | constante
 | variable
 | operation
 | invocationPrimitive

# Le si-alors-sinon. L'alternant est facultatif.

alternative = element alternative {
   element condition   { expression },
   element consequence { expression + },
   element alternant   { expression + } ?
}

# La séquence qui permet de regrouper plusieurs expressions en une seule.
# Il est obligatoire qu'il y ait au moins une expression dans la séquence.

sequence = element sequence {
   expression +
}

# Le bloc local unaire. Il est conservé pour garder les tests associés.
# Mais on pourrait s'en passer au profit du blocLocal plus général.
```

```
blocUnaire = element blocUnaire {
   variable,
   element valeur { expression },
   element corps  { expression + }
}

# Un bloc local qui introduit un nombre quelconque (éventuellement nul)
# de variables locales associées à une valeur initiale (calculée avec
# une expression).

blocLocal = element blocLocal {
   element liaisons {
      element liaison {
         variable,
         element initialisation {
            expression
         }
      } *
   },
   element corps { expression + }
}

# La boucle tant-que n'a de sens que parce que l'on dispose maintenant
# de l'affectation.

boucle = element boucle {
   element condition { expression },
   element corps     { expression + }
}

# L'affectation prend une variable en cible et une expression comme
# valeur. L'affectation est une expression.

affectation = element affectation {
   attribute nom  { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   element valeur { expression }
}

# L'invocation d'une fonction définie.

invocation = element invocation {
   element fonction { expression },
   element arguments { expression * }
}


# Cette définition permet une clause catch ou une clause finally ou
# encore ces deux clauses à la fois.

try = element try {
   element corps { expression + },
   (  catch
    | finally
    | ( catch, finally )
   )
}

catch = element catch {
   attribute exception { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   expression +
}

finally = element finally {
   expression +
}

# Une variable n'est caractérisée que par son nom. Les variables dont
# les noms comportent la séquence ilp ou ILP sont réservés et ne
# peuvent être utilisés par les programmeurs.

variable = element variable {
   attribute nom { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   empty
}

# L'invocation d'une fonction primitive. Une fonction primitive est
# procurée par l'implantation et ne peut (usuellement) être définie
# par l'utilisateur. Les fonctions primitives sont, pour être
# utilisables, prédéfinies. Une fonction primitive n'est caractérisée
# que par son nom (éventuellement masquable).

invocationPrimitive = element invocationPrimitive {
   attribute fonction { xsd:Name - ( xsd:Name { pattern = "(ilp|ILP)" } ) },
   expression *
}

# Les operations sont en fait des sortes d'invocations a des fonctions
# primitives sauf que ces fonctions sont implantées par le matériel
# par des instructions particulières. On ne distingue que les
# opérations unaires et binaires (les plus usuelles):

operation =
   operationUnaire
 | operationBinaire

operationUnaire = element operationUnaire {
   attribute operateur { "-" | "!" },
   element operande { expression }
}
operationBinaire = element operationBinaire {
   element operandeGauche { expression },
   attribute operateur {
      "+" | "-" | "*" | "/" | "%" |                      # arithmétiques
      "|" | "&" | "^" |                                  # booléens
      "<" | "<=" | "==" | ">=" | ">" | "<>" | "!="        # comparaisons
   },
   element operandeDroit { expression }
}

# Les constantes sont les données qui peuvent apparaître dans les
# programmes sous forme textuelle (ou littérale comme l'on dit
# souvent). Ici l'on trouve toutes les constantes usuelles à part les
# caractères:

constante =
   element entier    {
    attribute valeur { xsd:integer },
    empty }
 | element flottant  {
    attribute valeur { xsd:float },
    empty }
 | element chaine    { text }
 | element booleen   {
    attribute valeur { "true" | "false" },
    empty }
```

Java/src/fr/upmc/ilp/ilp4/Process.java

```java
package fr.upmc.ilp.ilp4;

import java.io.IOException;

import org.w3c.dom.Document;

import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp2.runtime.ConstantsStuff;
import fr.upmc.ilp.ilp3.ThrowPrimitive;
import fr.upmc.ilp.ilp4.ast.CEAST;
import fr.upmc.ilp.ilp4.ast.CEASTFactory;
import fr.upmc.ilp.ilp4.ast.CEASTParser;
import fr.upmc.ilp.ilp4.ast.NormalizeException;
import fr.upmc.ilp.ilp4.ast.NormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.ast.NormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4program;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.runtime.CommonPlus;
import fr.upmc.ilp.ilp4.runtime.LexicalEnvironment;
import fr.upmc.ilp.ilp4.runtime.PrintStuff;
import fr.upmc.ilp.tool.IFinder;

/** Cette classe pr?cise comment est trait? un programme d'ILP4. */

public class Process extends fr.upmc.ilp.ilp3.Process {

    /** Un constructeur utilisant toutes les valeurs par defaut possibles.
     * @throws IOException */

    public Process (IFinder finder) throws IOException {
        super(finder); // pour m?moire!
        setGrammar(getFinder().findFile("grammar4.rng"));
        IAST4Factory factory = new CEASTFactory();
        setFactory(factory);
        setParser(new CEASTParser(factory));
    }

    /** Profitons de la covariance! */
    @Override
    public IAST4program getCEAST () {
        return CEAST.narrowToIAST4program(super.getCEAST());
    }
    @Override
    public IAST4Factory getFactory () {
        return CEAST.narrowToIAST4Factory(super.getFactory());
    }

    /** Initialisation: @see fr.upmc.ilp.tool.AbstractProcess. */

    /** Pr?paration. On analyse syntaxiquement le texte du programme,
     * on effectue quelques analyses et on l'am?ne ? un ?tat o? il
     * pourra ?tre interpr?t? ou compil?. Toutes les analyses communes
     * ? ces deux fins sont partag?es ici.
     */
    @Override
    public void prepare() {
        try {
            final Document d = getDocument(this.rngFile);
            setCEAST(getParser().parse(d));

            // Toutes les analyses statiques
            setCEAST(performNormalization());
            getCEAST().computeInvokedFunctions();
            getCEAST().inline(getFactory());
            getCEAST().computeGlobalVariables();

            this.prepared = true;

        } catch (Throwable e) {
            this.preparationFailure = e;
        }
    }

    /** Normalization */

    public IAST4program performNormalization()
    throws NormalizeException {
        IAST4Factory factory = getFactory();
        final INormalizeLexicalEnvironment normlexenv =
            new NormalizeLexicalEnvironment.EmptyNormalizeLexicalEnvironment();
        final INormalizeGlobalEnvironment normcommon =
            new NormalizeGlobalEnvironment();
        normcommon.addPrimitive(factory.newGlobalVariable("print"));
        normcommon.addPrimitive(factory.newGlobalVariable("newline"));
        normcommon.addPrimitive(factory.newGlobalVariable("throw"));
        return getCEAST().normalize(normlexenv, normcommon, factory);
    }

    /** Interpretation */

    @Override
    public void interpret() {
        try {
            assert this.prepared;
            final ICommon intcommon = new CommonPlus();
            intcommon.bindPrimitive("throw", ThrowPrimitive.create());
            final ILexicalEnvironment intlexenv =
                LexicalEnvironment.EmptyLexicalEnvironment.create();
            final PrintStuff intps = new PrintStuff();
            intps.extendWithPrintPrimitives(intcommon);
            final ConstantsStuff csps = new ConstantsStuff();
            csps.extendWithPredefinedConstants(intcommon);

            this.result = getCEAST().eval(intlexenv, intcommon);
            this.printing = intps.getPrintedOutput().trim();

            this.interpreted = true;

        } catch (Throwable e) {
            this.interpretationFailure = e;
        }
    }

    /** Compilation vers C (h?rit?e) */

    /** Ex?cution du programme compil? (h?rit?e) */
}
```

Java/src/fr/upmc/ilp/ilp4/XMLProcess.java

```
package fr.upmc.ilp.ilp4;

import java.io.IOException;

import org.w3c.dom.Document;

import fr.upmc.ilp.ilp4.ast.XMLwriter;
import fr.upmc.ilp.tool.FileTool;
import fr.upmc.ilp.tool.IFinder;

/**  Cette classe traite un programme ILP4 en imprimant l'AST en XML apres
 * chaque phase. Ca peut etre utile pour tracer la forme de l'arbre ou pour
 * rendre persistant un tel AST.
 * */

public class XMLProcess extends Process {

    public XMLProcess (IFinder finder, String basename) throws IOException {
        super(finder);
        this.basename = basename;
    }
    protected String basename;

    @Override
    public void prepare() {
        try {
            final Document d = getDocument(this.rngFile);
            setCEAST(getParser().parse(d));

            XMLwriter xmlWriter = new XMLwriter();
            FileTool.stuffFile(basename + "-A.xml", xmlWriter.process(getCEAST()));
            // Les analyses statiques
            setCEAST(performNormalization());
            FileTool.stuffFile(basename + "-B.xml", xmlWriter.process(getCEAST()));
            getCEAST().computeInvokedFunctions();
            FileTool.stuffFile(basename + "-C.xml", xmlWriter.process(getCEAST()));
            getCEAST().inline(getFactory());
            FileTool.stuffFile(basename + "-D.xml", xmlWriter.process(getCEAST()));
            getCEAST().computeGlobalVariables();
            FileTool.stuffFile(basename + "-E.xml", xmlWriter.process(getCEAST()));

            this.prepared = true;

        } catch (Throwable e) {
            this.preparationFailure = e;
        }
    }
}
```

Java/src/fr/upmc/ilp/ilp4/test/ProcessTest.java

```
package fr.upmc.ilp.ilp4.test;

import java.io.IOException;
import java.util.Collection;

import org.junit.Before;
import org.junit.runner.RunWith;

import fr.upmc.ilp.ilp1.test.AbstractProcessTest;
import fr.upmc.ilp.ilp4.Process;
import fr.upmc.ilp.tool.File;
import fr.upmc.ilp.tool.Parameterized;
import fr.upmc.ilp.tool.Parameterized.Parameters;
```

```
@RunWith(value=Parameterized.class)
public class ProcessTest extends fr.upmc.ilp.ilp3.test.ProcessTest {

    /** Le constructeur du test sur un fichier. */

    public ProcessTest (final File file) {
        super(file);
    }

    @Before
    @Override
    public void setUp () throws IOException {
        this.setProcess(new Process(finder));
        getProcess().setFinder(finder);
    }

    @Parameters
    public static Collection<File[]> data() throws Exception {
        initializeFromOptions();
        AbstractProcessTest.staticSetUp(samplesDir, "u\\d+-[1-4]");
        // Pour un (ou plusieurs) test(s) en particulier:
        //AbstractProcessTest.staticSetUp(samplesDir, "u59-2");
        return AbstractProcessTest.collectData();
    }
}
```

Java/src/fr/upmc/ilp/ilp4/test/WholeTestSuite.java

```
package fr.upmc.ilp.ilp4.test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

/** Regroupement de classes de tests pour le paquetage ilp4. */

@RunWith(value=Suite.class)
@SuiteClasses(value={
        // Tous les fichiers de tests un par un:
        fr.upmc.ilp.ilp4.test.DelegationTest.class,
        fr.upmc.ilp.ilp4.test.RawInterpretationTest.class,
        fr.upmc.ilp.ilp4.test.NormalizeTest.class,
        fr.upmc.ilp.ilp4.test.InliningTest.class,
        fr.upmc.ilp.ilp4.test.NonInlinedEvaluationTest.class,
        fr.upmc.ilp.ilp4.test.XMLwriterTest.class,
        fr.upmc.ilp.ilp4.test.ProcessTest.class
})
public class WholeTestSuite {}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/AbstractExplicitVisitor.java

```
package fr.upmc.ilp.ilp4.interfaces;


/**
 * Ce visiteur abstrait parcourt toutes les expressions
 * d'un AST. On aurait pu aussi le definir en utilisant les capacites
 * reflexives de Java (a faire en AbstractReflectiveVisitor). Ce visiteur
 * prend en entrée (Data) ce qu'il rend en sortie (Data): c'est souvent
 * utile mais les sous-classes peuvent très bien ignorer l'un ou l'autre.
 *
 * Attention, ce visiteur ne visite pas les variables.
 */

public abstract class AbstractExplicitVisitor<Data, Exc extends Throwable>
implements IAST4visitor<Data, Data, Exc> {

    public Data visit (IAST4alternative iast, Data data) throws Exc {
        iast.getCondition().accept(this, data);
        iast.getConsequent().accept(this, data);
        if ( null != iast.getAlternant() ) {
            iast.getAlternant().accept(this, data);
        }
        return data;
    }
    public Data visit (IAST4assignment iast, Data data) throws Exc {
        iast.getValue().accept(this, data);
        return data;
    }
   public Data visit (IAST4localAssignment iast, Data data) throws Exc {
        iast.getValue().accept(this, data);
        return data;
    }
    public Data visit (IAST4globalAssignment iast, Data data) throws Exc {
        iast.getValue().accept(this, data);
        return data;
    }

    public Data visit (IAST4constant iast, Data data) throws Exc {
        // pas de sous-expression!
        return data;
    }

    public Data visit (IAST4invocation iast, Data data) throws Exc {
        iast.getFunction().accept(this, data);
        for ( IAST4expression arg : iast.getArguments() ) {
            arg.accept(this, data);
        }
        return data;
    }
    public Data visit (IAST4computedInvocation iast, Data data) throws Exc {
        iast.getFunction().accept(this, data);
        for ( IAST4expression arg : iast.getArguments() ) {
            arg.accept(this, data);
        }
        return data;
    }
    public Data visit (IAST4globalInvocation iast, Data data) throws Exc {
        for ( IAST4expression arg : iast.getArguments() ) {
            arg.accept(this, data);
        }
        return data;
    }
    public Data visit (IAST4primitiveInvocation iast, Data data) throws Exc {
        for ( IAST4expression arg : iast.getArguments() ) {
            arg.accept(this, data);
        }
        return data;
    }

    public Data visit (IAST4functionDefinition iast, Data data) throws Exc {
        iast.getBody().accept(this, data);
        return data;
    }

    public Data visit (IAST4reference iast, Data data) throws Exc {
        return data;
    }

    public Data visit (IAST4localBlock iast, Data data) throws Exc {
        for ( IAST4expression init : iast.getInitializations() ) {
            init.accept(this, data);
        }
        iast.getBody().accept(this, data);
        return data;
    }

    public Data visit (IAST4unaryBlock iast, Data data) throws Exc {
        iast.getInitialization().accept(this, data);
        iast.getBody().accept(this, data);
        return data;
    }

    public Data visit (IAST4binaryOperation iast, Data data) throws Exc {
        iast.getLeftOperand().accept(this, data);
        iast.getRightOperand().accept(this, data);
        return data;
    }

    public Data visit (IAST4unaryOperation iast, Data data) throws Exc {
        iast.getOperand().accept(this, data);
        return data;
    }
```

```

    public Data visit (IAST4program iast, Data data) throws Exc {
        for ( IAST4functionDefinition fun : iast.getFunctionDefinitions() ) {
            fun.accept(this, data);
        }
        iast.getBody().accept(this, data);
        return data;
    }

    public Data visit (IAST4sequence iast, Data data) throws Exc {
        for ( IAST4expression expr : iast.getInstructions() ) {
            expr.accept(this, data);
        }
        return data;
    }

    public Data visit (IAST4try iast, Data data) throws Exc {
        iast.getBody().accept(this, data);
        if ( null != iast.getCatcher() ) {
            iast.getCatcher().accept(this, data);
        }
        if ( null != iast.getFinallyer() ) {
            iast.getFinallyer().accept(this, data);
        }
        return data;
    }

    public Data visit (IAST4while iast, Data data) throws Exc {
        iast.getCondition().accept(this, data);
        iast.getBody().accept(this, data);
        return data;
    }
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4.java

```
package fr.upmc.ilp.ilp4.interfaces;

import java.util.Set;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp4.ast.FindingInvokedFunctionsException;
import fr.upmc.ilp.ilp4.ast.InliningException;

/** L'interface generaliste des AST à la fois interprétables et compilables.
 */

public interface IAST4
extends IAST4visitable, IAST2<CEASTparseException> {

    /** Calculer le graphe d'appel c'est-à-dire pour chaque expression,
     * les fonctions globales qu'elle invoque. */

    void computeInvokedFunctions ()
    throws FindingInvokedFunctionsException;

    /** Renvoyer l'ensemble des fonctions globales invoquées (et
     * précédemment calculées). */

    Set<IAST4globalFunctionVariable> getInvokedFunctions ();

    /** Intégrer les fonctions non récursives. */

    void inline (IAST4Factory factory) throws InliningException;

    /** Accepter le passage d'un visiteur. */

    <Data, Result, Exc extends Throwable> Result accept (
            IAST4visitor<Data, Result, Exc> visitor,
            Data data) throws Exc;

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4Factory.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp3.IAST3Factory;

public interface IAST4Factory
extends IAST3Factory<CEASTparseException>
{
    IAST4expression newVoidExpression ();

    IAST4program newProgram(IAST4functionDefinition[] defs,
                            IAST4expression body);

    IAST4sequence newSequence (IAST4expression[] asts);

    IAST4globalFunctionVariable newGlobalFunctionVariable (
            String name );
     IAST4functionDefinition newFunctionDefinition (
            IAST4globalFunctionVariable global,
            IAST4variable[] variables,
            IAST4expression body );

    // On raffine ces signatures:

    IAST4try newTry (
            IAST4expression body,
            IAST4variable caughtExceptionVariable,
            IAST4expression catcher,
            IAST4expression finallyer);

    IAST4alternative newAlternative(
            IAST4expression condition,
            IAST4expression consequent);

    IAST4alternative newAlternative(
            IAST4expression condition,
            IAST4expression consequent,
            IAST4expression alternant);

    IAST4variable newVariable(String name);
    IAST4localVariable newLocalVariable(String name);
    IAST4globalVariable newGlobalVariable(String name);

    IAST4reference newReference(IAST4variable variable);

    IAST4assignment newAssignment(
            IAST4variable variable,
            IAST4expression value);
    IAST4localAssignment newLocalAssignment(
            IAST4localVariable variable,
            IAST4expression value);
    IAST4globalAssignment newGlobalAssignment(
            IAST4globalVariable variable,
            IAST4expression value);

    IAST4invocation newInvocation(
            IAST4expression function,
            IAST4expression[] arguments);
    IAST4computedInvocation newComputedInvocation(
            IAST4expression function,
            IAST4expression[] arguments);
    IAST4globalInvocation newGlobalInvocation(
            IAST4globalFunctionVariable function,
            IAST4expression[] arguments);
    IAST4primitiveInvocation newPrimitiveInvocation(
            IAST4globalVariable function,
            IAST4expression[] arguments);

    IAST4unaryOperation newUnaryOperation(
            String operatorName,
            IAST4expression operand);

    IAST4binaryOperation newBinaryOperation(
            String operatorName,
            IAST4expression leftOperand,
            IAST4expression rightOperand);

    IAST4integer newIntegerConstant(String value);
    IAST4float newFloatConstant(String value);
    IAST4string newStringConstant(String value);
    IAST4boolean newBooleanConstant(String value);
```

```
    IAST4localBlock newLocalBlock(
            IAST4variable[] variables,
            IAST4expression[] initializations,
            IAST4expression body);

    IAST4unaryBlock newUnaryBlock(
            IAST4variable variable,
            IAST4expression initialization,
            IAST4expression body);

    IAST4while newWhile(
            IAST4expression condition,
            IAST4expression body);
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4alternative.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2alternative;

public interface IAST4alternative
extends IAST4instruction, IAST2alternative<CEASTparseException> {

    IAST4expression getCondition ();
    IAST4expression getConsequent ();
    IAST4expression getAlternant ();

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4assignment.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2assignment;

public interface IAST4assignment
extends IAST4expression, IAST2assignment<CEASTparseException> {
    IAST4variable   getVariable ();
    IAST4expression getValue ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4binaryOperation.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2binaryOperation;

public interface IAST4binaryOperation
extends IAST4operation, IAST2binaryOperation<CEASTparseException> {
    IAST4expression getLeftOperand ();
    IAST4expression getRightOperand ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4boolean.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2boolean;

public interface IAST4boolean
extends IAST4constant, IAST2boolean<CEASTparseException> {
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4computedInvocation.java

```java
package fr.upmc.ilp.ilp4.interfaces;

public interface IAST4computedInvocation
extends IAST4invocation {
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4constant.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2constant;

public interface IAST4constant
extends IAST4expression, IAST2constant<CEASTparseException> {
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4delegable.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2;

public interface IAST4delegable extends IAST4 {

    /** Rendre l'objet d'ILP2 a qui ILP4 delegue le stockage des donnees
     * et l'interpretation. Son type sera en general bien plus precis. */
    IAST2<CEASTparseException> getDelegate ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4expression.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.ast.NormalizeException;

public interface IAST4expression
extends IAST4, IAST4visitable, IAST2expression<CEASTparseException> {

    IAST4expression normalize (INormalizeLexicalEnvironment lexenv,
                               INormalizeGlobalEnvironment common,
                               IAST4Factory factory )
    throws NormalizeException;

    /** Compilation d'une instruction ou expression. Production de code
     * C par ajout à un tampon, dans un environnement lexical et un
     * environnement global. Le résultat est produit avec une certaine
     * destination. */

    void compile (StringBuffer buffer,
                  ICgenLexicalEnvironment lexenv,
                  ICgenEnvironment common,
                  IDestination destination )
    throws CgenerationException;

    //NOTE: Methodes heritees d'ILP2 nuisibles en ILP4. On se
    // contente de les marquer comme obsoletes.

    @Deprecated
    void compileExpression (
            StringBuffer buffer,
            ICgenLexicalEnvironment lexenv,
            ICgenEnvironment common,
            IDestination destination )
    throws CgenerationException;

    @Deprecated
```

```java
    void compileExpression (
            StringBuffer buffer,
            ICgenLexicalEnvironment lexenv,
            ICgenEnvironment common )
    throws CgenerationException;
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4float.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2float;

public interface IAST4float
extends IAST4constant, IAST2float<CEASTparseException> {
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4functionDefinition.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import java.util.Set;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp4.ast.NormalizeException;

public interface IAST4functionDefinition
extends IAST4internal, IAST2functionDefinition<CEASTparseException> {
    IAST4globalFunctionVariable getDefinedVariable ();
    IAST4variable[]             getVariables ();
    // Version mieux typee de la precedente:
    IAST4localVariable[]        getLocalVariables ();
    IAST4expression             getBody ();
    boolean                     isRecursive ();
    Set<IAST4globalFunctionVariable> getInvokedFunctions ();
    void setInvokedFunctions(Set<IAST4globalFunctionVariable> funvars);

    IAST4functionDefinition normalize(
            INormalizeLexicalEnvironment lexenv,
            INormalizeGlobalEnvironment common,
            IAST4Factory factory )
    throws NormalizeException;
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalAssignment.java

```java
package fr.upmc.ilp.ilp4.interfaces;

public interface IAST4globalAssignment extends IAST4assignment {
    IAST4globalVariable   getVariable ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalFunctionVariable.java

```java
package fr.upmc.ilp.ilp4.interfaces;

public interface IAST4globalFunctionVariable
extends IAST4globalVariable {

    /** Recuperer la definition de la fonction ainsi nommee. */
    IAST4functionDefinition getFunctionDefinition ();

    /** Associer une definition de fonction. */
    void setFunctionDefinition (IAST4functionDefinition functionDefinition);

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalInvocation.java

```
package fr.upmc.ilp.ilp4.interfaces;

public interface IAST4globalInvocation
extends IAST4invocation {
    IAST4globalFunctionVariable getFunctionGlobalVariable ();
    IAST4expression             getInlined ();
    void  setInlined(IAST4expression inlined);
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4globalVariable.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;

public interface IAST4globalVariable
extends IAST4variable {

    /** Engendrer une déclaration globale en C pour cette variable. */
    void compileGlobalDeclaration(
            StringBuffer buffer,
            ICgenLexicalEnvironment lexenv,
            ICgenEnvironment common )
        throws CgenerationException;

}

// end of IAST4globalVariable
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4instruction.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;

public interface IAST4instruction
extends IAST4expression, IAST2instruction<CEASTparseException> {
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4integer.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2integer;

public interface IAST4integer
extends IAST4constant, IAST2integer<CEASTparseException> {

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4internal.java

```
package fr.upmc.ilp.ilp4.interfaces;

import java.util.Set;

/** Cette interface definit quelques methodes utilitaires non
 * destinees a un usage externe. */

public interface IAST4internal extends IAST4 {

    /** Indiquer que d'autres fonctions sont invoquees. Renvoie vrai lorsque
     * de nouvelles fonctions ont ete ajoutees qui n'etaient pas encore
     * presentes (comme la methode Set.addAll()) */

    boolean addInvokedFunctions (Set<IAST4globalFunctionVariable> others);

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4invocation.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2invocation;

public interface IAST4invocation
extends IAST4expression, IAST2invocation<CEASTparseException> {
    IAST4expression   getFunction ();
    IAST4expression[] getArguments ();
    IAST4expression getArgument (int i);
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4localAssignment.java

```
package fr.upmc.ilp.ilp4.interfaces;

public interface IAST4localAssignment extends IAST4assignment {
    IAST4localVariable   getVariable ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4localBlock.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2localBlock;

public interface IAST4localBlock
extends IAST4instruction, IAST2localBlock<CEASTparseException> {
    IAST4variable[]       getVariables ();
    // La version mieux typee de la precedente:
    IAST4localVariable[]  getLocalVariables ();
    IAST4expression[]     getInitializations ();
    IAST4expression       getBody ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4localVariable.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;

public interface IAST4localVariable
extends IAST4variable {

    /** Engendrer une déclaration locale en C pour cette variable. */
    void compileDeclaration(
            StringBuffer buffer,
            ICgenLexicalEnvironment lexenv,
            ICgenEnvironment common )
        throws CgenerationException;
}

//end of IAST4localeVariable
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4operation.java

```
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2operation;

public interface IAST4operation
extends IAST4expression, IAST2operation<CEASTparseException> {
    IAST2operation<CEASTparseException> getDelegate ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4primitiveInvocation.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2primitiveInvocation;

public interface IAST4primitiveInvocation
extends IAST4invocation,
        IAST2primitiveInvocation<CEASTparseException> {
    IAST4globalVariable getFunctionGlobalVariable ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4program.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;
import fr.upmc.ilp.ilp4.ast.NormalizeException;

public interface IAST4program
extends IAST4, IAST2program<CEASTparseException> {
    IAST4functionDefinition[] getFunctionDefinitions ();
    IAST4expression           getBody ();
    IAST4globalVariable[]     getGlobalVariables ();

    // analyses statiques
    void computeGlobalVariables();
    void setGlobalVariables (IAST4globalVariable[] globals);
    IAST4program normalize (
            INormalizeLexicalEnvironment lexenv,
            INormalizeGlobalEnvironment common,
            IAST4Factory factory )
      throws NormalizeException;
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4reference.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2reference;

public interface IAST4reference
extends IAST4expression, IAST2reference<CEASTparseException> {
    /** Retourne la variable lue */
    IAST4variable getVariable();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4sequence.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2sequence;

public interface IAST4sequence
extends IAST4instruction, IAST2sequence<CEASTparseException> {
    IAST4expression   getInstruction (int i) throws CEASTparseException;
    IAST4expression[] getInstructions ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4string.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2string;

public interface IAST4string
extends IAST4constant, IAST2string<CEASTparseException> {
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4try.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp3.IAST3try;

public interface IAST4try
extends IAST4instruction, IAST3try<CEASTparseException> {
    IAST4expression getBody ();
    IAST4variable   getCaughtExceptionVariable ();
    IAST4expression getCatcher ();
    IAST4expression getFinallyer ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4unaryBlock.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryBlock;

public interface IAST4unaryBlock
extends IAST4instruction, IAST2unaryBlock<CEASTparseException> {
    IAST4variable       getVariable ();
    // Version mieux typee de la precedente:
    IAST4localVariable  getLocalVariable ();
    IAST4expression     getInitialization ();
    IAST4expression     getBody ();

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4unaryOperation.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryOperation;

public interface IAST4unaryOperation
extends IAST4operation, IAST2unaryOperation<CEASTparseException> {
    IAST4expression getOperand ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4variable.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.ast.NormalizeException;

/** L'interface décrivant les divers types de variables. Parmi elles
 * se trouvent les variables globales, prédéfinies et locales. */

public interface IAST4variable
extends IAST4visitable, IAST2variable {

    /** Valeur d'une variable. */

    Object eval (ILexicalEnvironment lexenv, ICommon common)
        throws EvaluationException;

    /** Normaliser l'AST et notamment alpha-convertir les variables. */

    IAST4variable normalize (INormalizeLexicalEnvironment lexenv,
                             INormalizeGlobalEnvironment common,
                             IAST4Factory factory )
        throws NormalizeException;
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4visitable.java

```java
package fr.upmc.ilp.ilp4.interfaces;

/** Un noeud d'AST est visitable s'il offre cette methode. Cette methode
 * ne procure qu'un rebond typé vers le visiteur.
 * Cf. fr.upmc.ilp.ilp4.interfaces.IAST4visitor */

public interface IAST4visitable {

    /** Ce visiteur peut prendre des donnees additionnelles dans la
     * variable data et retourne une valeur qui peut eventuellement
     * etre exploitee. */

    <Data, Result, Exc extends Throwable> Result accept (
            IAST4visitor<Data, Result, Exc> visitor,
            Data data) throws Exc;

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4visitor.java

```java
package fr.upmc.ilp.ilp4.interfaces;

/** Un visiteur d'IAST procure une methode pour traiter chaque type
 * de noeud present dans un IAST4.
 *
 *  NOTE: Autant que faire se peut, on utilise des interfaces mais toutes les
 *  classes ne sont pas cachables derriere des IAST* (while par
 *  exemple) et certaines classes raffinent une meme interface (les
 *  variables et les constantes par exemple).
 */

public interface IAST4visitor<Data, Result, Exc extends Throwable> {
    // visiter les expressions:
    Result visit (IAST4alternative iast, Data data) throws Exc;
    Result visit (IAST4assignment iast, Data data) throws Exc;
    Result visit (IAST4localAssignment iast, Data data) throws Exc;
    Result visit (IAST4globalAssignment iast, Data data) throws Exc;
    Result visit (IAST4constant iast, Data data) throws Exc;
    Result visit (IAST4invocation iast, Data data) throws Exc;
    Result visit (IAST4globalInvocation iast, Data data) throws Exc;
    Result visit (IAST4computedInvocation iast, Data data) throws Exc;
    Result visit (IAST4primitiveInvocation iast, Data data) throws Exc;
    Result visit (IAST4functionDefinition iast, Data data) throws Exc;
    Result visit (IAST4localBlock iast, Data data) throws Exc;
    Result visit (IAST4reference iast, Data data) throws Exc;
    Result visit (IAST4unaryBlock iast, Data data) throws Exc;
    Result visit (IAST4binaryOperation iast, Data data) throws Exc;
    Result visit (IAST4unaryOperation iast, Data data) throws Exc;
    Result visit (IAST4program iast, Data data) throws Exc;
    Result visit (IAST4sequence iast, Data data) throws Exc;
    Result visit (IAST4try iast, Data data) throws Exc;
    Result visit (IAST4while iast, Data data) throws Exc;
    // et aussi les variables referencees:
    Result visit (IAST4variable iast, Data data) throws Exc;

}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IAST4while.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2while;

public interface IAST4while
extends IAST4instruction, IAST2while<CEASTparseException> {
    IAST4expression getCondition ();
    IAST4expression getBody ();
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/INormalizeGlobalEnvironment.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.annotation.OrNull;

/** Normaliser les variables globales veut dire utiliser un unique
 * objet pour toutes les références à une variable globale. Il est
 * ainsi possible de partager simplement de l'information sur cette
 * variable globale depuis tous les endroits où elle est référencée.
 */
public interface INormalizeGlobalEnvironment {

    /** Étendre l'environnement global avec une nouvelle variable. */
    void add(IAST4globalVariable variable);

    /** Vérifie qu'une variable est présente dans l'environnement
     * global. Si elle est présente, elle est renvoyée en résultat
     * autrement null est renvoyé. */
    @OrNull IAST4globalVariable isPresent (IAST4variable variable);

    /** Ajouter une primitive à l'environnement global. */
    void addPrimitive (IAST4globalVariable variable);

    /** Vérifie qu'une variable correspond au nom d'une primitive. Si
     * elle est présente, elle est renvoyée en résultat autrement null
     * est renvoyé. */
    @OrNull IAST4globalVariable isPrimitive (IAST4variable variable);
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/INormalizeLexicalEnvironment.java

```java
package fr.upmc.ilp.ilp4.interfaces;

public interface INormalizeLexicalEnvironment {

    /** Étend l'environnement avec une nouvelle variable. */
    INormalizeLexicalEnvironment extend(IAST4variable variable);

    /** Vérifie qu'une variable est présente dans le seul environnement
     * lexical. Si elle est présente, elle est renvoyée en résultat
     * autrement null est renvoyé. */
    IAST4variable isPresent (IAST4variable variable);
}
```

Java/src/fr/upmc/ilp/ilp4/interfaces/IParser.java

```java
package fr.upmc.ilp.ilp4.interfaces;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;

public interface IParser
extends fr.upmc.ilp.ilp3.IParser<CEASTparseException> {
    // On raffine la signature de la fabrique:
    IAST4Factory getFactory ();
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEAST.java

```java
package fr.upmc.ilp.ilp4.ast;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.HashSet;
import java.util.Set;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4instruction;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4program;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

/** Code commun à tous les noeuds de l'AST d'ILP4. */

public abstract class CEAST
extends fr.upmc.ilp.ilp2.ast.CEAST
implements IAST4 {

    protected CEAST () {
        this.invokedFunctions = new HashSet<>();
    }

    public abstract Object eval (ILexicalEnvironment lexenv,
                                 ICommon common)
    throws EvaluationException;

    /** Normaliser l'AST. Par défaut, le n?ud est normalisé. La
     * normalisation porte surtout sur les variables, la ternarisation
     * des alternatives, la réduction des séquences triviales. */

    public IAST4 normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory )
    throws NormalizeException {
        return this;
    }

    /**
     * Pratique en Eclipse! Ainsi, dans la perspective de mise au point,
     * la valeur d'un CEASTprogram s'affichera de maniere plus lisible. Il
     * egalement possible de positionner (menu contextuel: edit detail
     * formatter) sur la variable Process.ceast qu'on veut la voir s'afficher
     * avec: "return new XMLwriter().process(this);". Cette meme astuce doit
     * fonctionner avec toute instance d'IAST4.
     */
    @Override
    public String toString () {
        try {
            if ( xmlwriter == null ) {
               xmlwriter = new XMLwriter();
            }
            return xmlwriter.process(this);
        } catch (Throwable t) {
            return super.toString();
        }
    }
    private static XMLwriter xmlwriter;

    /** Calculer le graphe d'appel c'est-à-dire pour chaque expression,
     * les fonctions globales qu'elle invoque. Par défaut, l'expression
     * n'invoque aucune fonction globale. */

    public void computeInvokedFunctions ()
    throws FindingInvokedFunctionsException {
        final Class<? extends CEAST> clazz = this.getClass();
        for ( final Method m : clazz.getMethods() ) {
            // FUTURE: mettre en cache cette recherche serait plus efficace!
            final ILPexpression ee = m.getAnnotation(ILPexpression.class);
            handleAnnotation(ee, m);
            final ILPvariable ev = m.getAnnotation(ILPvariable.class);
            handleAnnotation (ev, m);
        }
    }
    private Set<IAST4globalFunctionVariable> invokedFunctions;
    // NOTE: un tel champ par instance est dispendieux!
```

```
    public void setInvokedFunctions (Set<IAST4globalFunctionVariable> funvars) {
        this.invokedFunctions = funvars;
    }

    public void handleAnnotation (ILPexpression e, Method m)
    throws FindingInvokedFunctionsException {
        try {
            if ( e != null ) {
                if ( e.isArray() ) {
                    final Object[] results = (Object[])
                    m.invoke(this, EMPTY_ARGUMENT_ARRAY);
                    for ( Object result : results ) {
                        if ( e.neverNull() || result != null ) {
                            final IAST4expression component =
                                CEAST.narrowToIAST4expression(result);
                            this.findAndAdjoinToInvokedFunctions(component);
                        }
                    }
                } else {
                    final Object result =
                        m.invoke(this, EMPTY_ARGUMENT_ARRAY);
                    if ( e.neverNull() || result != null ) {
                        final IAST4expression component =
                            CEAST.narrowToIAST4expression(result);
                        this.findAndAdjoinToInvokedFunctions(component);
                    }
                }
            }
        } catch (IllegalArgumentException exc) {
            throw new FindingInvokedFunctionsException(exc);
        } catch (IllegalAccessException exc) {
            throw new FindingInvokedFunctionsException(exc);
        } catch (InvocationTargetException exc) {
            throw new FindingInvokedFunctionsException(exc);
        }
    }

    public void handleAnnotation (ILPvariable e, Method m)
    throws FindingInvokedFunctionsException {
        try {
            if ( e != null ) {
                if ( e.isArray() ) {
                    final Object[] results = (Object[])
                    m.invoke(this, EMPTY_ARGUMENT_ARRAY);
                    for ( Object result : results ) {
                        if ( e.neverNull() || result != null ) {
                            final IAST4variable component =
                                CEAST.narrowToIAST4variable(result);
                            this.findAndAdjoinToInvokedFunctions(component);
                        }
                    }
                } else {
                    final Object result =
                        m.invoke(this, EMPTY_ARGUMENT_ARRAY);
                    if ( e.neverNull() || result != null ) {
                        final IAST4variable component =
                            CEAST.narrowToIAST4variable(result);
                        this.findAndAdjoinToInvokedFunctions(component);
                    }
                }
            }
        } catch (IllegalArgumentException exc) {
            throw new FindingInvokedFunctionsException(exc);
        } catch (IllegalAccessException exc) {
            throw new FindingInvokedFunctionsException(exc);
        } catch (InvocationTargetException exc) {
            throw new FindingInvokedFunctionsException(exc);
        }
    }

    /** Une methode utilitaire pour fusionner des ensembles de fonctions
     * globales provenant des sous-arbres de l'expression courante. */

    protected void findAndAdjoinToInvokedFunctions (
            final IAST4expression e)
    throws FindingInvokedFunctionsException {
```

```java
        e.computeInvokedFunctions();
        this.invokedFunctions.addAll(e.getInvokedFunctions());
    }

    protected void findAndAdjoinToInvokedFunctions (
            final IAST4variable e)
    throws FindingInvokedFunctionsException {
        if ( e instanceof IAST4globalFunctionVariable ) {
            IAST4globalFunctionVariable gfv = (IAST4globalFunctionVariable) e;
            this.invokedFunctions.add(gfv);
        }
    }

    /** Renvoyer l'ensemble des fonctions globales invoquées (qui doivent avoir
     * ete précédemment calculées). */

    public Set<IAST4globalFunctionVariable> getInvokedFunctions () {
        return this.invokedFunctions;
    }

    /** Indiquer qu'une fonction est invoquee. */

    public void addInvokedFunction (
            final IAST4globalFunctionVariable variable) {
        this.invokedFunctions.add(variable);
    }

    /** Indiquer que d'autres fonctions sont invoquees. Renvoie vrai lorsque
     * de nouvelles fonctions ont ete ajoutees qui n'etaient pas encore
     * presentes (comme la methode Set.addAll()) */

    public boolean addInvokedFunctions (
            final Set<IAST4globalFunctionVariable> others) {
        return this.invokedFunctions.addAll(others);
    }

    /** Intégrer les fonctions non récursives. Cette implantation use de
     * réflexivité. */

    public void inline (IAST4Factory factory)
    throws InliningException {
        final Class<? extends CEAST> clazz = this.getClass();
        for ( Method m : clazz.getMethods() ) {
            try {
                final ILPexpression e = m.getAnnotation(ILPexpression.class);
                if ( e != null ) {
                    if ( e.isArray() ) {
                        final Object[] results = (Object[])
                            m.invoke(this, EMPTY_ARGUMENT_ARRAY);
                        for ( Object result : results ) {
                            if ( e.neverNull() || result != null ) {
                                final IAST4expression component =
                                    CEAST.narrowToIAST4expression(result);
                                component.inline(factory);
                            }
                        }
                    } else {
                        final Object result =
                            m.invoke(this, EMPTY_ARGUMENT_ARRAY);
                        if ( e.neverNull() || result != null ) {
                            final IAST4expression component =
                                CEAST.narrowToIAST4expression(result);
                            component.inline(factory);
                        }
                    }
                }
            } catch (IllegalArgumentException e) {
                throw new InliningException(e);
            } catch (IllegalAccessException e) {
                throw new InliningException(e);
            } catch (InvocationTargetException e) {
                throw new InliningException(e);
            }
        }
    }
    private static final Object[] EMPTY_ARGUMENT_ARRAY = new Object[0];

```

```java
    /** Les rétrécisseurs spécialisés. */

      public static IAST4 narrowToIAST4 (Object o) {
        if ( o instanceof IAST4) {
          return (IAST4) o;
        } else {
          final String msg = "Cannot cast into IAST4: " + o;
          throw new ClassCastException(msg);
        }
      }

    public static IAST4variable narrowToIAST4variable (Object o) {
      if ( o instanceof IAST4variable ) {
        return (IAST4variable) o;
      } else {
        final String msg = "Cannot cast into IAST4variable: " + o;
        throw new ClassCastException(msg);
      }
    }

    public static IAST4variable[] narrowToIAST4variableArray (Object o) {
      if ( o instanceof IAST4variable[] ) {
        return (IAST4variable[]) o;
      } else if ( o instanceof IAST2variable[] ) {
        IAST2variable[] v = (IAST2variable[]) o;
        IAST4variable[] result = new IAST4variable[v.length];
        for ( int i=0 ; i<v.length ; i++ ) {
            result[i] = CEAST.narrowToIAST4variable(v[i]);
        }
        return result;
      } else {
        final String msg = "Cannot cast into IAST4variable[]: " + o;
        throw new ClassCastException(msg);
      }
    }

    public static IAST4globalVariable narrowToIAST4globalVariable (Object o) {
      if ( o instanceof IAST4globalVariable ) {
        return (IAST4globalVariable) o;
      } else {
        final String msg = "Cannot cast into IAST4globalVariable: " + o;
        throw new ClassCastException(msg);
      }
    }

    public static IAST4globalVariable[] narrowToIAST4globalVariableArray (Object o) {
      if ( o instanceof IAST4globalVariable[] ) {
        return (IAST4globalVariable[]) o;
      } else if ( o instanceof IAST2variable[] ) {
          IAST2variable[] v = (IAST2variable[]) o;
          IAST4globalVariable[] result = new IAST4globalVariable[v.length];
          for ( int i=0 ; i<v.length ; i++ ) {
              result[i] = CEAST.narrowToIAST4globalVariable(v[i]);
          }
          return result;
      } else {
        final String msg = "Cannot cast into IAST4globalVariable: " + o;
        throw new ClassCastException(msg);
      }
    }

    public static IAST4localVariable narrowToIAST4localVariable (Object o) {
      if ( o instanceof IAST4localVariable ) {
        return (IAST4localVariable) o;
      } else {
        final String msg = "Cannot cast into IAST4localVariable: " + o;
        throw new ClassCastException(msg);
      }
    }

    public static IAST4localVariable[] narrowToIAST4localVariableArray (Object o) {
      if ( o instanceof IAST4localVariable[] ) {
        return (IAST4localVariable[]) o;
      } else if ( o instanceof IAST2variable[] ) {
          IAST2variable[] v = (IAST2variable[]) o;
          IAST4localVariable[] result = new IAST4localVariable[v.length];
          for ( int i=0 ; i<v.length ; i++ ) {
              result[i] = CEAST.narrowToIAST4localVariable(v[i]);
    }
    return result;
  } else {
   final String msg = "Cannot cast into IAST4localVariable: " + o;
   throw new ClassCastException(msg);
  }
 }

 public static IAST4globalFunctionVariable
  narrowToIAST4globalFunctionVariable (Object o) {
  if ( o instanceof IAST4globalFunctionVariable ) {
   return (IAST4globalFunctionVariable) o;
  } else {
   final String msg = "Cannot cast into IAST4globalFunctionVariable: " + o;
   throw new ClassCastException(msg);
  }
 }

 public static IAST4expression narrowToIAST4expression (Object o) {
  if ( o instanceof IAST4expression ) {
   return (IAST4expression) o;
  } else {
   final String msg = "Cannot cast into IAST4expression: " + o;
   throw new ClassCastException(msg);
  }
 }

 public static IAST4expression[] narrowToIAST4expressionArray (Object o) {
  if ( o instanceof IAST4expression[] ) {
   return (IAST4expression[]) o;
  } else if ( o instanceof IAST2expression<?>[] ) {
    IAST2expression<?>[] v = (IAST2expression[]) o;
    IAST4expression[] result = new IAST4expression[v.length];
    for ( int i=0 ; i<v.length ; i++ ) {
      result[i] = CEAST.narrowToIAST4expression(v[i]);
    }
    return result;
  } else if ( o instanceof IAST2instruction[] ) {
   IAST2instruction<?>[] v = (IAST2instruction[]) o;
   IAST4expression[] result = new IAST4expression[v.length];
   for ( int i=0 ; i<v.length ; i++ ) {
     result[i] = CEAST.narrowToIAST4expression(v[i]);
   }
   return result;
  } else {
   final String msg = "Cannot cast into IAST4expression[]: " + o;
   throw new ClassCastException(msg);
  }
 }

 public static IAST4instruction narrowToIAST4instruction (Object o) {
  if ( o instanceof IAST4instruction ) {
   return (IAST4instruction) o;
  } else {
   final String msg = "Cannot cast into IAST4instruction: " + o;
   throw new ClassCastException(msg);
  }
 }

 public static IAST4functionDefinition narrowToIAST4functionDefinition (Object o) {
   if ( o instanceof IAST4functionDefinition ) {
    return (IAST4functionDefinition) o;
   } else {
    final String msg = "Cannot cast into IAST4functionDefinition: " + o;
    throw new ClassCastException(msg);
   }
  }

 public static IAST4functionDefinition[] narrowToIAST4functionDefinitionArray (Object o) {
   if ( o instanceof IAST4functionDefinition[] ) {
    return (IAST4functionDefinition[]) o;
   } else if ( o instanceof IAST2functionDefinition<?>[] ) {
     IAST2functionDefinition<?>[] v = (IAST2functionDefinition[]) o;
     IAST4functionDefinition[] result = new IAST4functionDefinition[v.length];
     for ( int i=0 ; i<v.length ; i++ ) {
       result[i] = CEAST.narrowToIAST4functionDefinition(v[i]);
     }
     return result;
```

```
   } else {
    final String msg = "Cannot cast into IAST4functionDefinition[]: " + o;
    throw new ClassCastException(msg);
   }
  }

 public static IAST4Factory narrowToIAST4Factory (Object o) {
   if ( o instanceof IAST4Factory ) {
    return (IAST4Factory) o;
   } else {
    final String msg = "Cannot cast into IAST4Factory: " + o;
    throw new ClassCastException(msg);
   }
  }

 public static IAST4program narrowToIAST4program(Object o) {
   if ( o instanceof IAST4program ) {
    return (IAST4program) o;
   } else {
    final String msg = "Cannot cast into IAST4program: " + o;
    throw new ClassCastException(msg);
   }
  }
 public static fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment
  narrowToLexicalEnvironment2(Object o) {
   if ( o instanceof fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment ) {
     return (fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment) o;
    } else {
     final String msg = "Cannot cast into ilp2.interfaces.ILexicalEnvironment: " + o;
     throw new ClassCastException(msg);
    }
 }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTFactory.java

```
package fr.upmc.ilp.ilp4.ast;

import java.util.List;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4alternative;
import fr.upmc.ilp.ilp4.interfaces.IAST4assignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4binaryOperation;
import fr.upmc.ilp.ilp4.interfaces.IAST4boolean;
import fr.upmc.ilp.ilp4.interfaces.IAST4computedInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4float;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4integer;
import fr.upmc.ilp.ilp4.interfaces.IAST4invocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4localAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4localBlock;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4primitiveInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4program;
import fr.upmc.ilp.ilp4.interfaces.IAST4reference;
import fr.upmc.ilp.ilp4.interfaces.IAST4sequence;
import fr.upmc.ilp.ilp4.interfaces.IAST4string;
import fr.upmc.ilp.ilp4.interfaces.IAST4try;
import fr.upmc.ilp.ilp4.interfaces.IAST4unaryBlock;
import fr.upmc.ilp.ilp4.interfaces.IAST4unaryOperation;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4while;

/**
 * Une implantation de la fabrique pour ILP4. Comme l'on doit implanter
 * IAST4Factory, on doit aussi implanter des methodes prenant des IAST2*
 * que l'on redirige, apres verification, vers les methodes d'ilp4.
 */

public class CEASTFactory
implements IAST4Factory {

    public IAST4program newProgram(
            IAST2functionDefinition<CEASTparseException>[] defs,
            IAST2instruction<CEASTparseException> body) {
        return this.newProgram(
                CEAST.narrowToIAST4functionDefinitionArray(defs),
                CEAST.narrowToIAST4expression(body));
    }
    public IAST4program newProgram(
            IAST4functionDefinition[] defs,
            IAST4expression body) {
        return new CEASTprogram(defs, body);
    }

    public IAST4variable newVariable(String name) {
        return new CEASTvariable(name);
    }
    public IAST4globalFunctionVariable newGlobalFunctionVariable(
            String name ) {
        return new CEASTglobalFunctionVariable(name);
    }
    public IAST4localVariable newLocalVariable(String name) {
        return new CEASTlocalVariable(name);
    }
    public IAST4globalVariable newGlobalVariable(String name) {
        return new CEASTglobalVariable(name);
    }

    public IAST4assignment newAssignment(IAST4variable variable,
                                         IAST4expression value) {
        return new CEASTassignment(variable, value);
    }
    public IAST4localAssignment newLocalAssignment(
            IAST4localVariable variable,
            IAST4expression value) {
        return new CEASTlocalAssignment(variable, value);
    }
    public IAST4globalAssignment newGlobalAssignment(
            IAST4globalVariable variable,
            IAST4expression value) {
        return new CEASTglobalAssignment(variable, value);
    }
    public IAST4assignment newAssignment(
            IAST2variable variable, IAST2expression<CEASTparseException> value) {
        return new CEASTassignment(
                CEAST.narrowToIAST4variable(variable),
                CEAST.narrowToIAST4expression(value) );
    }

    public IAST4expression newVoidExpression () {
        return new CEASTboolean("false");
    }

    public IAST4alternative newAlternative(IAST4expression condition,
                                           IAST4expression consequence) {
        IAST4expression something = newVoidExpression();
        return new CEASTalternative(condition, consequence, something);
    }
    public IAST4alternative newAlternative(
            IAST4expression condition,
            IAST4expression consequence,
            IAST4expression alternant) {
        return new CEASTalternative(condition, consequence, alternant);
    }
    public IAST4alternative newAlternative(
            IAST2expression<CEASTparseException> condition,
            IAST2instruction<CEASTparseException> consequent) {
        return new CEASTalternative(
                CEAST.narrowToIAST4expression(condition),
                CEAST.narrowToIAST4expression(consequent) );
    }
    public IAST4alternative newAlternative(
            IAST2expression<CEASTparseException> condition,
            IAST2instruction<CEASTparseException> consequent,
            IAST2instruction<CEASTparseException> alternant) {
        return new CEASTalternative(
```

```
                CEAST.narrowToIAST4expression(condition),
                CEAST.narrowToIAST4expression(consequent),
                CEAST.narrowToIAST4expression(alternant) );
    }

    public IAST4unaryOperation newUnaryOperation(
            String operatorName,
            IAST4expression operand) {
        return new CEASTunaryOperation(operatorName, operand);
    }
    public IAST4unaryOperation newUnaryOperation(
            String operatorName, IAST2expression<CEASTparseException> operand) {
        return new CEASTunaryOperation(
                operatorName,
                CEAST.narrowToIAST4expression(operand));
    }

    public IAST4binaryOperation newBinaryOperation(
            String operatorName,
            IAST4expression leftOperand,
            IAST4expression rightOperand) {
        return new CEASTbinaryOperation(operatorName, leftOperand, rightOperand);
    }
    public IAST4binaryOperation newBinaryOperation(
            String operatorName,
            IAST2expression<CEASTparseException> leftOperand,
            IAST2expression<CEASTparseException> rightOperand) {
        return new CEASTbinaryOperation(
                operatorName,
                CEAST.narrowToIAST4expression(leftOperand),
                CEAST.narrowToIAST4expression(rightOperand) );
    }

    public IAST4boolean newBooleanConstant(String value) {
        return new CEASTboolean(value);
    }

    public IAST4float newFloatConstant(String value) {
        return new CEASTfloat(value);
    }

    public IAST4integer newIntegerConstant(String value) {
        return new CEASTinteger(value);
    }

    public IAST4string newStringConstant(String value) {
        return new CEASTstring(value);
    }

    public IAST4functionDefinition newFunctionDefinition(
            IAST4globalFunctionVariable global,
            IAST4variable[] variables,
            IAST4expression body) {
        return new CEASTfunctionDefinition(global, variables, body);
    }
    public IAST4functionDefinition newFunctionDefinition(
            String functionName,
            IAST2variable[] variables,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTfunctionDefinition(
                this.newGlobalFunctionVariable(functionName),
                CEAST.narrowToIAST4variableArray(variables),
                CEAST.narrowToIAST4expression(body) );
    }

    public IAST4invocation newInvocation(IAST4expression function,
                                         IAST4expression[] arguments) {
        return new CEASTinvocation(function, arguments);
    }
    public IAST4computedInvocation newComputedInvocation(
            IAST4expression function,
            IAST4expression[] arguments) {
        return new CEASTcomputedInvocation(function, arguments);
    }
    public IAST4globalInvocation newGlobalInvocation(
            IAST4globalFunctionVariable function,
            IAST4expression[] arguments) {
        return new CEASTglobalInvocation(function, arguments);
    }
    public IAST4invocation newInvocation(
```

```java
            IAST2expression<CEASTparseException> function,
            IAST2expression<CEASTparseException>[] arguments) {
        return new CEASTinvocation(
                CEAST.narrowToIAST4expression(function),
                CEAST.narrowToIAST4expressionArray(arguments) );
    }

    public IAST4localBlock newLocalBlock(IAST4variable[] variables,
                                         IAST4expression[] initializations,
                                         IAST4expression body ) {
        return new CEASTlocalBlock(variables, initializations, body);
    }
    public IAST4localBlock newLocalBlock(
            IAST2variable[] variables,
            IAST2expression<CEASTparseException>[] initializations,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTlocalBlock(
                CEAST.narrowToIAST4variableArray(variables),
                CEAST.narrowToIAST4expressionArray(initializations),
                CEAST.narrowToIAST4expression(body) );
    }

    public IAST4unaryBlock newUnaryBlock(
            IAST4variable variable,
            IAST4expression initialization,
            IAST4expression body) {
        return new CEASTunaryBlock(variable, initialization, body);
    }
    public IAST4unaryBlock newUnaryBlock(
            IAST2variable variable,
            IAST2expression<CEASTparseException> initialization,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTunaryBlock(
                CEAST.narrowToIAST4variable(variable),
                CEAST.narrowToIAST4expression(initialization),
                CEAST.narrowToIAST4expression(body) );
    }

    public IAST4reference newReference(IAST4variable variable) {
        return new CEASTreference(variable);
    }
    public IAST4reference newReference(
            IAST2variable variable) {
        return new CEASTreference(
                CEAST.narrowToIAST4variable(variable) );
    }

    public IAST4sequence newSequence(IAST4expression[] asts) {
        return new CEASTsequence(asts);
    }
    public IAST4sequence newSequence(
            List<IAST2instruction<CEASTparseException>> asts) {
        return new CEASTsequence(
                asts.toArray(new IAST4expression[0]) );
    }

    public IAST4try newTry(IAST4expression body,
                           IAST4variable caughtExceptionVariable,
                           IAST4expression catcher,
                           IAST4expression finallyer) {
        return new CEASTtry(body, caughtExceptionVariable, catcher, finallyer);
    }
    public IAST4try newTry(
                IAST2instruction<CEASTparseException> body,
                IAST2variable caughtExceptionVariable,
                IAST2instruction<CEASTparseException> catcher,
                IAST2instruction<CEASTparseException> finallyer) {
        return new CEASTtry(
                CEAST.narrowToIAST4expression(body),
                (caughtExceptionVariable == null) ? null
                 : CEAST.narrowToIAST4variable(caughtExceptionVariable),
                (catcher == null) ? null
                 : CEAST.narrowToIAST4expression(catcher),
                (finallyer == null) ? null
                 : CEAST.narrowToIAST4expression(finallyer) );
    }

    public IAST4while newWhile(IAST4expression condition,
                               IAST4expression body) {
        return new CEASTwhile(condition, body);
```

```java
    }
    public IAST4while newWhile(
            IAST2expression<CEASTparseException> condition,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTwhile(
                CEAST.narrowToIAST4expression(condition),
                CEAST.narrowToIAST4expression(body) );
    }

    public IAST4primitiveInvocation newPrimitiveInvocation(
            IAST4globalVariable gv,
            IAST4expression[] arguments) {
        return new CEASTprimitiveInvocation(gv, arguments);
    }
    public IAST4primitiveInvocation newPrimitiveInvocation(
            String primitiveName,
            IAST2expression<CEASTparseException>[] arguments) {
        return new CEASTprimitiveInvocation(
                this.newGlobalVariable(primitiveName),
                CEAST.narrowToIAST4expressionArray(arguments));
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTParser.java

```java
package fr.upmc.ilp.ilp4.ast;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.HashMap;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;

import fr.upmc.ilp.ilp2.ast.AbstractParser;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp4.interfaces.IAST4;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4program;
import fr.upmc.ilp.ilp4.interfaces.IAST4sequence;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Transformer un document XML en un CEAST. */

public class CEASTParser extends AbstractParser
implements IParser {

    @Override
    public IAST4Factory getFactory () {
        return (IAST4Factory) super.getFactory();
    }

    public CEASTParser (IAST4Factory factory) {
        super(factory);
        this.parsers = new HashMap<>();
        addParser("alternative",         CEASTalternative.class);
        addParser("sequence",            CEASTsequence.class);
        addParser("boucle",              CEASTwhile.class);
        addParser("affectation",         CEASTassignment.class);
        addParser("definitionFonction",  CEASTfunctionDefinition.class);
        addParser("blocUnaire",          CEASTunaryBlock.class);
        addParser("blocLocal",           CEASTlocalBlock.class);
        addParser("variable",            CEASTreference.class);
        addParser("invocationPrimitive", CEASTprimitiveInvocation.class);
        addParser("invocation",          CEASTinvocation.class);
        addParser("operationUnaire",     CEASTunaryOperation.class);
        addParser("operationBinaire",    CEASTbinaryOperation.class);
        addParser("entier",              CEASTinteger.class);
        addParser("flottant",            CEASTfloat.class);
        addParser("booleen",             CEASTboolean.class);
        addParser("chaine",              CEASTstring.class);
        addParser("try",                 CEASTtry.class);
    }
    private final HashMap<String, Method> parsers;

    /** Ajout d'une caracteristique a ILP. Lorsque l'element XML nommé
```

```
     * name est lu, method(e, parser) sera invoquee. */

    public void addParser (String name, Method method) {
        this.parsers.put(name, method);
    }

    /** Ajout d'une caracteristique a ILP. Lorsque l'element XML nommé
     * name est lu, la methode clazz.parse(e, parser) sera invoquee. */

    public void addParser (String name, Class<?> clazz) {
      try {
        // Ne fonctionne plus avec ILP6: faut ruser!
        //final Method method = clazz.getMethod("parse",
        //   new Class[]{ Element.class, IParser.class } );
        for ( Method m : clazz.getMethods() ) {
            if ( ! "parse".equals(m.getName()) ) {
                continue;
            }
            if ( ! Modifier.isStatic(m.getModifiers()) ) {
                continue;
            }
            //if ( ! IAST2.class.isAssignableFrom(m.getReturnType())) {
            //    continue;
            //}
            Class<?>[] parameterTypes = m.getParameterTypes();
            if ( parameterTypes.length != 2 ) {
                continue;
            }
            if ( ! Element.class.isAssignableFrom(parameterTypes[0]) ) {
                continue;
            }
            if ( ! IParser.class.isAssignableFrom(parameterTypes[1]) ) {
                continue;
            }
            addParser(name, m);
            return;
        }
        if ( Object.class == clazz ) {
            final String msg = "Cannot find suitable parse() method!";
            throw new RuntimeException(msg);
        } else {
            addParser(name, clazz.getSuperclass());
        }
      } catch (SecurityException e1) {
        final String msg = "Cannot access parse() method!";
        throw new RuntimeException(msg);
      }
    }

  /** Convertir un noeud DOM en un noeud AST. */

    public IAST4program parse (final Document d)
    throws CEASTparseException {
        final Element e = d.getDocumentElement();
        return CEASTprogram.parse(e, this);
    }

    public IAST4 parse (final Node n)
      throws CEASTparseException {
      switch ( n.getNodeType() ) {
      case Node.ELEMENT_NODE: {
        final Element e = (Element) n;
        final String name = e.getTagName();

        if ( parsers.containsKey(name) ) {
            final Method method = parsers.get(name);
            try {
              Object result = method.invoke(null, new Object[]{e, this});
              return CEAST.narrowToIAST4(result);
            } catch (IllegalArgumentException exc) {
              throw new CEASTparseException(exc);
            } catch (IllegalAccessException exc) {
              throw new CEASTparseException(exc);
            } catch (InvocationTargetException exc) {
                Throwable t = exc.getTargetException();
                if ( t instanceof CEASTparseException ) {
                    throw (CEASTparseException) t;
                } else {
```

```
                    throw new CEASTparseException(exc);
                }
            }
        } else {
          final String msg = "Unknown element name: " + name;
          throw new CEASTparseException(msg);
        }
      }

      default: {
        final String msg = "Unknown node type: " + n.getNodeName();
        throw new CEASTparseException(msg);
      }
      }

    }

    // NOTE: meme code qu'en ILP2 sauf que les CEASTsequence sont celles d'ILP4.

    /** Trouver un sous-noeud donné et convertir ses fils en une séquence
     * d'instructions. */
    @Override
    public IAST4sequence findThenParseChildAsSequence (
             final NodeList nl, final String childName)
    throws CEASTparseException {
        return getFactory().newSequence(
                findThenParseChildAsList(nl, childName)
                    .toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY) );
    }

    @Override
    public IAST4sequence findThenParseChildAsSequence (
             final Node n, final String childName)
    throws CEASTparseException {
        return getFactory().newSequence(
                findThenParseChildAsList(n.getChildNodes(), childName)
                    .toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY) );
    }

    @Override
    public IAST4sequence parseChildrenAsSequence (final NodeList nl)
    throws CEASTparseException {
        return getFactory().newSequence(
                parseList(nl)
                .toArray(CEASTinstruction.EMPTY_EXPRESSION_ARRAY) );
    }

}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTalternative.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2alternative;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4alternative;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** L'alternative: son interprétation et sa compilation. */

public class CEASTalternative
extends CEASTdelegableInstruction
implements IAST4alternative {
```

```java
  public CEASTalternative (final IAST4expression condition,
                           final IAST4expression consequence,
                           final IAST4expression alternant ) {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTalternative(
              condition, consequence, alternant );
  }
  public CEASTalternative (final IAST4expression condition,
                           final IAST4expression consequence ) {
      this(condition,
           consequence,
           CEASTexpression.voidExpression() );
  }
  private fr.upmc.ilp.ilp2.ast.CEASTalternative delegate;

  @Override
  public IAST2alternative<CEASTparseException> getDelegate () {
      return this.delegate;
  }

  public static IAST2alternative<CEASTparseException> parse (
          final Element e, final IParser parser)
  throws CEASTparseException {
      return fr.upmc.ilp.ilp2.ast.CEASTalternative.parse(e, parser);
  }

  @ILPexpression
  public IAST4expression getCondition () {
    return CEAST.narrowToIAST4expression(getDelegate().getCondition());
  }
  @ILPexpression
  public IAST4expression getConsequent () {
    return CEAST.narrowToIAST4expression(getDelegate().getConsequent());
  }
  @ILPexpression
  public IAST4expression getAlternant () {
      try {
          return CEAST.narrowToIAST4expression(getDelegate().getAlternant());
      } catch (CEASTparseException e) {
          assert false : "Should not occur!";
          throw new RuntimeException(e);
      }
  }
  public boolean isTernary () {
      return true;
  }

  @Override
  public void compile (final StringBuffer buffer,
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common,
                       final IDestination destination)
    throws CgenerationException {
      final IAST4variable tmp = CEASTlocalVariable.generateVariable();
      buffer.append("{ ");
      tmp.compileDeclaration(buffer, lexenv, common);
      getCondition().compile(buffer, lexenv, common, new AssignDestination(tmp));
      buffer.append(";\n if ( ILP_isEquivalentToTrue( ");
      buffer.append(tmp.getMangledName());
      buffer.append(" ) ) {\n");
      getConsequent().compile(buffer, lexenv, common, destination);
      buffer.append(";\n } else {\n");
      getAlternant().compile(buffer, lexenv, common, destination);
      buffer.append(";\n }\n}");
  }

@Override
public IAST4expression normalize (
        final INormalizeLexicalEnvironment lexenv,
        final INormalizeGlobalEnvironment common,
        final IAST4Factory factory )
  throws NormalizeException {
    return factory.newAlternative(
            getCondition().normalize(lexenv, common, factory),
            getConsequent().normalize(lexenv, common, factory),
            getAlternant().normalize(lexenv, common, factory) );
}
```

```java
  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      findAndAdjoinToInvokedFunctions(getCondition());
      findAndAdjoinToInvokedFunctions(getConsequent());
      findAndAdjoinToInvokedFunctions(getAlternant());
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
      getCondition().inline();
      getConsequent().inline();
      getAlternant().inline();
  }
  */

  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTassignment.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2assignment;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4assignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Affectation à des variables. Cette classe sert a representer une
 * affectation. La phase de normalisation la classifiera en affectation
 * locale ou globale.
 */

public class CEASTassignment
  extends CEASTdelegableExpression
  implements IAST4assignment {

  public CEASTassignment (final IAST4variable variable,
                          final IAST4expression value) {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTassignment(
                  variable, value );
  }
  private fr.upmc.ilp.ilp2.ast.CEASTassignment delegate;

  @Override
  public IAST2assignment<CEASTparseException> getDelegate () {
      return this.delegate;
  }

  @ILPvariable
  public IAST4variable getVariable () {
    return CEAST.narrowToIAST4variable(getDelegate().getVariable());
```

```
}
@ILPexpression
public IAST4expression getValue() {
  return CEAST.narrowToIAST4expression(getDelegate().getValue());
}

public static IAST2assignment<CEASTparseException> parse (
        final Element e, final IParser parser)
throws CEASTparseException {
    return fr.upmc.ilp.ilp2.ast.CEASTassignment.parse(e, parser);
}

/**
 * Normaliser l'affectation en l'une de ses sous-classes suivant
 * la nature de la variable affectee.  */
@Override
public IAST4expression normalize (
        final INormalizeLexicalEnvironment lexenv,
        final INormalizeGlobalEnvironment common,
        final IAST4Factory factory )
  throws NormalizeException {
  IAST4variable variable_ =
      getVariable().normalize(lexenv, common, factory);
  final IAST4expression value_ =
      getValue().normalize(lexenv, common, factory);

  if ( variable_ instanceof IAST4globalVariable ) {
    final IAST4globalVariable gv = (IAST4globalVariable) variable_;
    return factory.newGlobalAssignment(gv, value_);

  } else if ( variable_ instanceof IAST4localVariable ) {
      final IAST4localVariable lv = (IAST4localVariable) variable_;
    return factory.newLocalAssignment(lv, value_);

  } else {
      final String msg = "Should never occur!";
      assert false : msg;
      throw new NormalizeException(msg);
  }
}

/* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
@Override
public void findInvokedFunctions () {
    findAndAdjoinToInvokedFunctions(getValue());
}

/* Obsolète de par CEAST.inline() qui use de réflexivité.
@Override
public void inline () {
    getValue().inline();
}
*/

@Override
public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common,
                     final IDestination destination)
throws CgenerationException {
  getValue().compile(buffer, lexenv, common,
         new AssignDestination(getVariable()) );
  buffer.append(";\n");
  buffer.append(getVariable().getMangledName());
  buffer.append(";\n");
}

@Override
public Object eval (final ILexicalEnvironment lexenv, final ICommon common)
throws EvaluationException {
    final String msg = "Should never occur!";
    assert false : msg;
    throw new EvaluationException(msg);
}

public <Data, Result, Exc extends Throwable> Result
  accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
    return visitor.visit(this, data);
```

```
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTbinaryOperation.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2binaryOperation;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4binaryOperation;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Opérateurs binaires. */

public class CEASTbinaryOperation
  extends CEASToperation
  implements IAST4binaryOperation {

  public CEASTbinaryOperation (final String operatorName,
                               final IAST4expression left,
                               final IAST4expression right) {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTbinaryOperation(
                  operatorName, left, right);
  }
  private fr.upmc.ilp.ilp2.ast.CEASTbinaryOperation delegate;

  @Override
  public IAST2binaryOperation<CEASTparseException> getDelegate () {
      return this.delegate;
  }

  public static IAST2binaryOperation<CEASTparseException> parse (
          final Element e, final IParser parser)
  throws CEASTparseException {
      return fr.upmc.ilp.ilp2.ast.CEASTbinaryOperation.parse(e, parser);
  }

  @ILPexpression
  public IAST4expression getLeftOperand () {
    return CEAST.narrowToIAST4expression(this.delegate.getLeftOperand());
  }
  @ILPexpression
  public IAST4expression getRightOperand () {
    return CEAST.narrowToIAST4expression(getDelegate().getRightOperand());
  }
  public IAST4expression[] getOperands () {
      IAST2expression<CEASTparseException>[] operands =
          getDelegate().getOperands();
      return CEAST.narrowToIAST4expressionArray(operands);
  }

  @Override
  public IAST4expression normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common,
          final IAST4Factory factory )
    throws NormalizeException {
      return factory.newBinaryOperation(
              getOperatorName(),
              getLeftOperand().normalize(lexenv, common, factory),
              getRightOperand().normalize(lexenv, common, factory) );
  }

  @Override
  public void compile (final StringBuffer buffer,
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common,
                       final IDestination destination )
  throws CgenerationException {
      final IAST4variable right = CEASTlocalVariable.generateVariable();
      final IAST4variable left  = CEASTlocalVariable.generateVariable();
      buffer.append("{\n");
      ICgenLexicalEnvironment bodyLexenv = lexenv.extend(left).extend(right);
      right.compileDeclaration(buffer, lexenv, common);
      left.compileDeclaration(buffer, lexenv, common);
      getLeftOperand().compile(buffer, bodyLexenv, common,
              new AssignDestination(left) );
      buffer.append(";\n");
      getRightOperand().compile(buffer, bodyLexenv, common,
              new AssignDestination(right) );
      buffer.append(";\n");
      destination.compile(buffer, bodyLexenv, common);
      buffer.append(common.compileOperator2(getOperatorName()));
      buffer.append("(");
      buffer.append(left.getMangledName());
      buffer.append(", ");
      buffer.append(right.getMangledName());
      buffer.append(");}\n");
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      findAndAdjoinToInvokedFunctions(getLeftOperand());
      findAndAdjoinToInvokedFunctions(getRightOperand());
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
    getLeftOperand().inline();
    getRightOperand().inline();
  }
  */

  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTboolean.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2boolean;
import fr.upmc.ilp.ilp4.interfaces.IAST4boolean;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Les constantes booleennes.*/

public class CEASTboolean
extends CEASTconstant implements IAST4boolean {

    public CEASTboolean (String value) {
        super(new fr.upmc.ilp.ilp2.ast.CEASTboolean(value));
    }

    public static IAST2boolean<CEASTparseException> parse (
            final Element e, final IParser parser)
    throws CEASTparseException {
        return fr.upmc.ilp.ilp2.ast.CEASTboolean.parse(e, parser);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTcomputedInvocation.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4computedInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

public class CEASTcomputedInvocation
extends CEASTinvocation implements IAST4computedInvocation {

    protected CEASTcomputedInvocation (final IAST4expression function,
                                       final IAST4expression[] argument) {
        super(function, argument);
    }

    @Override
    public IAST4expression normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory )
      throws NormalizeException {
      final IAST4expression function_ =
          getFunction().normalize(lexenv, common, factory);
      final IAST4expression[] arguments = getArguments();
      IAST4expression[] argument_ = new IAST4expression[arguments.length];
      for ( int i = 0 ; i<arguments.length ; i++ ) {
          argument_[i] = arguments[i].normalize(lexenv, common, factory);
      }
      return factory.newComputedInvocation(function_, argument_);
    }

    /** On ne peut rien dire avant l'execution. */
    @Override
    public void checkArity () {
        return;
    }

    @Override
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final IDestination destination )
    throws CgenerationException {
        IAST4localVariable tmpfun = CEASTlocalVariable.generateVariable();
        IAST4expression[] args = getArguments();
        IAST4localVariable[] tmps = new IAST4localVariable[args.length];
        ICgenLexicalEnvironment bodyLexenv = lexenv;
        bodyLexenv = bodyLexenv.extend(tmpfun);
        buffer.append("{\n ILP_Primitive ");
        buffer.append(tmpfun.getMangledName());
        buffer.append(";\n");
        for ( int i=0; i<args.length ; i++ ) {
            tmps[i] = CEASTlocalVariable.generateVariable();
            tmps[i].compileDeclaration(buffer, lexenv, common);
            bodyLexenv = bodyLexenv.extend(tmps[i]);
        }
        getFunction().compile(buffer, bodyLexenv, common,
                new AssignDestination(tmpfun) );
        buffer.append(";\n");
        for ( int i=0 ; i<args.length ; i++ ) {
            args[i].compile(buffer, bodyLexenv, common,
                    new AssignDestination(tmps[i]) );
            buffer.append(";\n");
        }
        destination.compile(buffer, bodyLexenv, common);
        buffer.append(bodyLexenv.compile(tmpfun));
        buffer.append("(");
        for ( int i=0 ; i<(args.length-1) ; i++ ) {
            buffer.append(tmps[i].getMangledName());
            buffer.append(", ");
        }
        if ( args.length > 0 ) {
            buffer.append(tmps[args.length-1].getMangledName());
        }
        buffer.append(");\n}\n");
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTconstant.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2constant;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4constant;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;

/** Les constantes et leur interprétation. */

public abstract class CEASTconstant
extends CEASTdelegableExpression
implements IAST4constant {

    public CEASTconstant (final IAST2constant<CEASTparseException> delegate) {
        this.delegate = delegate;
    }
    protected IAST2constant<CEASTparseException> delegate;

    @Override
    public IAST2constant<CEASTparseException> getDelegate () {
        return this.delegate;
    }

    public Object getValue () {
        return getDelegate().getValue();
    }
    public String getDescription () {
        return getDelegate().getDescription();
    }

    /** Toutes les constantes valent leur propre valeur. */
    @Override
    public Object eval (final ILexicalEnvironment lexenv, final ICommon common)
    {
        return getDelegate().getValue();
    }

    public <Data, Result, Exc extends Throwable> Result
      accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
        return visitor.visit(this, data);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTdelegableExpression.java

```
package fr.upmc.ilp.ilp4.ast;

import java.util.Set;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4delegable;

public abstract class CEASTdelegableExpression
extends CEASTexpression
```

```
implements IAST4delegable {

    /** Profiter de la contravariance pour raffiner le type du delegue.
     * Attention! les expressions d'ilp4 ne sont malheureusement pas toutes
     * des expressions d'ilp2! Les expressions d'ilp4 contiennent les
     * instructions d'ilp4 qui sont deleguees a des instruction d'ilp2 qui
     * ne sont pas des expressions d'ilp2. */

    public abstract IAST2instruction<CEASTparseException> getDelegate ();

    /** Evaluer un AST */

    @Override
    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common)
    throws EvaluationException {
        return getDelegate().eval(lexenv, common);
    }

    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final IDestination destination)
    throws CgenerationException {
        final IAST2instruction<CEASTparseException> delegate = getDelegate();
        delegate.compileInstruction(buffer, lexenv, common, destination);
    }

    /** Par compatibilité avec le délégué. */
    @Override
    public void compileInstruction (final StringBuffer buffer,
                                    final ICgenLexicalEnvironment lexenv,
                                    final ICgenEnvironment common,
                                    final IDestination destination)
    throws CgenerationException {
        this.compile(buffer, lexenv, common, destination);
        buffer.append(";\n");
    }

    /** Calculer l'ensemble des variables globales de cette instruction.
     * Par défaut, il n'y a pas de variable globale.
     *
     * NOTE: Set&lt;IAST2variable&gt; n'est pas Set&lt;IAST4variable&gt; */
    @Override
    public void findGlobalVariables (final Set<IAST2variable> globalvars,
                                     final ICgenLexicalEnvironment lexenv ) {
        getDelegate().findGlobalVariables(globalvars, lexenv);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTdelegableInstruction.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4instruction;

public abstract class CEASTdelegableInstruction
extends CEASTdelegableExpression
implements IAST4instruction {

    //@Override
    //public abstract IAST2instruction getDelegate ();
    // NOTE: pas possible du point de vue typage du fait de l'inversion de
    // positionnement entre expression et instruction.

    @Override
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final IDestination destination)
    throws CgenerationException {
        final IAST2instruction<CEASTparseException> delegate =
            (IAST2instruction<CEASTparseException>) getDelegate();
        delegate.compileInstruction(buffer, lexenv, common, destination);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTexpression.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.cgen.NoDestination;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

/** La classe abstraite des expressions. */

public abstract class CEASTexpression
extends CEAST
implements IAST4expression {

    /** Renvoyer une expression vide qui ne fait rien d'interessant. */

    public static IAST4expression voidExpression() {
        return new CEASTboolean("false");
    }

    /** Une constante utile pour les conversions entre liste et tableau. */

    public static final IAST4expression[] EMPTY_EXPRESSION_ARRAY =
            new IAST4expression[0];

    /** Rendre la version integrée de l'expression. */

    public IAST4expression getInlined () {
        return this;
    }

    @Override
    public IAST4expression normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory )
    throws NormalizeException {
        return this;
    }

    public void compileExpression (final StringBuffer buffer,
                                   final ICgenLexicalEnvironment lexenv,
                                   final ICgenEnvironment common)
    throws CgenerationException {
        this.compile(buffer, lexenv, common, NoDestination.create());
    }

    @Override
    @Deprecated
    public void compileInstruction (final StringBuffer buffer,
                                    final ICgenLexicalEnvironment lexenv,
                                    final ICgenEnvironment common,
                                    final IDestination destination)
    throws CgenerationException {
        this.compile(buffer, lexenv, common, destination);
    }

    @Override
    @Deprecated
    public void compileExpression (final StringBuffer buffer,
                                   final ICgenLexicalEnvironment lexenv,
                                   final ICgenEnvironment common,
                                   final IDestination destination)
    throws CgenerationException {
```

```
        this.compile(buffer, lexenv, common, destination);
    }

}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTfloat.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2float;
import fr.upmc.ilp.ilp4.interfaces.IAST4float;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Les constantes flottantes. */

public class CEASTfloat
extends CEASTconstant implements IAST4float {

    public CEASTfloat(final String value) {
        super(new fr.upmc.ilp.ilp2.ast.CEASTfloat(value));
    }

    public static IAST2float<CEASTparseException> parse (
            final Element e, final IParser parser)
    throws CEASTparseException {
        return fr.upmc.ilp.ilp2.ast.CEASTfloat.parse(e, parser);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTfunctionDefinition.java

```
package fr.upmc.ilp.ilp4.ast;

import java.util.Set;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.cgen.ReturnDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4delegable;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Définition d'une fonction globale. */

public class CEASTfunctionDefinition
extends CEAST
implements IAST4functionDefinition, IAST4delegable {

  public CEASTfunctionDefinition (final IAST4globalFunctionVariable global,
                                  final IAST4variable[] variable,
                                  final IAST4expression body ) {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTfunctionDefinition(
                  global.getName(), variable, body );
```

```
    this.global = global;
    this.global.setFunctionDefinition(this);
}
private IAST4globalFunctionVariable global;
private fr.upmc.ilp.ilp2.ast.CEASTfunctionDefinition delegate;

public fr.upmc.ilp.ilp2.ast.CEASTfunctionDefinition getDelegate () {
    return this.delegate;
}

public IAST4globalFunctionVariable getDefinedVariable () {
  return this.global;
}
public String getFunctionName (){
  return this.delegate.getFunctionName();
}
public String getMangledFunctionName () {
    return this.delegate.getMangledFunctionName();
}
@ILPvariable(isArray=true)
public IAST4localVariable[] getLocalVariables () {
    return CEAST.narrowToIAST4localVariableArray(this.delegate.getVariables());
}
public IAST4variable[] getVariables () {
    return CEAST.narrowToIAST4variableArray(this.delegate.getVariables());
}
@ILPexpression
public IAST4expression getBody () {
    return CEAST.narrowToIAST4expression(this.delegate.getBody());
}

public static IAST4functionDefinition parse(
        final Element e, final IParser parser)
throws CEASTparseException {
    IAST2functionDefinition<CEASTparseException> delegate =
        fr.upmc.ilp.ilp2.ast.CEASTfunctionDefinition.parse(e, parser);
    IAST4Factory factory = parser.getFactory();
    IAST4globalFunctionVariable gfv =
        factory.newGlobalFunctionVariable(delegate.getFunctionName());
    return factory.newFunctionDefinition(
            gfv,
            CEAST.narrowToIAST4variableArray(delegate.getVariables()),
            CEAST.narrowToIAST4expression(delegate.getBody()) );
}

@Override
public Object eval (final ILexicalEnvironment lexenv,
                   final ICommon common)
throws EvaluationException {
    return getDelegate().eval(lexenv, common);
}

/** Émettre le prototype de la fonction globale. Cela permettra
 * d'assurer la récursion mutuelles des fonctions globales. */

public void compileHeader (final StringBuffer buffer,
                           final ICgenLexicalEnvironment lexenv,
                           final ICgenEnvironment common)
  throws CgenerationException {
  buffer.append("static ILP_Object ");
  buffer.append(getDefinedVariable().getMangledName());
  compileVariableList(buffer);
  buffer.append(";\n");
}

/** Étendre un environnement lexical de compilation avec les
 * variables de la fonction. */

public ICgenLexicalEnvironment extendWithFunctionVariables (
        final ICgenLexicalEnvironment lexenv )
{
  ICgenLexicalEnvironment newlexenv = lexenv;
  final IAST4variable[] variables = getVariables();
  for ( int i = 0 ; i<variables.length ; i++ ) {
    newlexenv = newlexenv.extend(variables[i]);
  }
  return newlexenv;
}
```

```
public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common )
  throws CgenerationException {
  // Émettre en commentaire les fonctions appelées:
    if ( getInvokedFunctions().size() > 0 ) {
        buffer.append("/* Fonctions globales invoquées: ");
        for ( IAST4globalFunctionVariable gv : getInvokedFunctions() ) {
            buffer.append(gv.getMangledName());
            buffer.append(" ");
        }
        buffer.append(" */\n");
    }
  if ( this.isRecursive() ) {
    buffer.append("/* Cette fonction est récursive. */\n");
  }
  // Émettre la définition de la fonction:
  buffer.append("\nILP_Object\n");
  buffer.append(getDefinedVariable().getMangledName());
  compileVariableList(buffer);
  buffer.append("\n{\n");
  final ICgenLexicalEnvironment bodyLexenv =
    this.extendWithFunctionVariables(lexenv);
  getBody().compile(buffer, bodyLexenv, common, ReturnDestination.create());
  buffer.append(";\n}");
}

public void compileVariableList (final StringBuffer buffer)
  throws CgenerationException {
  buffer.append(" (");
  final IAST4variable[] variables = getVariables();
  for ( int i = 0 ; i<variables.length-1 ; i++ ) {
    buffer.append("    ILP_Object ");
    buffer.append(variables[i].getMangledName());
    buffer.append(",\n");
  }
  if ( variables.length > 0 ) {
    buffer.append("    ILP_Object ");
    buffer.append(variables[variables.length-1].getMangledName());
  }
  buffer.append(" ) ");
}
// heriter aupres du delegue ???

/** Normaliser une fonction globale revient principalement à
 * normaliser son corps. */

@Override
public IAST4functionDefinition normalize (
        final INormalizeLexicalEnvironment lexenv,
        final INormalizeGlobalEnvironment common,
        final IAST4Factory factory )
  throws NormalizeException {
    final IAST4globalFunctionVariable gfv =
        CEAST.narrowToIAST4globalFunctionVariable(
                getDefinedVariable().normalize(lexenv, common, factory));
    INormalizeLexicalEnvironment bodyLexenv = lexenv;
    final IAST4variable[] variables = getVariables();
    final IAST4variable[] variables_ = new IAST4variable[variables.length];
    for ( int i = 0 ; i<variables.length ; i++ ) {
        variables_[i] = factory.newLocalVariable(variables[i].getName());
        bodyLexenv = bodyLexenv.extend(variables_[i]);
    }
    final IAST4expression body_ =
        getBody().normalize(bodyLexenv, common, factory);
    final IAST4functionDefinition fd =
        factory.newFunctionDefinition(gfv, variables_, body_);
    return fd;
}

/** Déterminer si la fonction est récursive. Cette méthode ne
 * fonctionne qu'après avoir calculé le graphe des appels. */

public boolean isRecursive () {
  for ( IAST4variable gv : getInvokedFunctions() ) {
      if ( gv == getDefinedVariable() ) {
          return true;
          }
    }
    return false;
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      findAndAdjoinToInvokedFunctions(getBody());
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
    getBody().inline();
  }
  */

  @Override
  public void findGlobalVariables (final Set<IAST2variable> globalvars,
          final ICgenLexicalEnvironment lexenv ) {
      final ICgenLexicalEnvironment newlexenv =
              this.extendWithFunctionVariables(lexenv);
      getBody().findGlobalVariables(globalvars, newlexenv);
  }

  public <Data, Result, Exc extends Throwable> Result
  accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
    return visitor.visit(this, data);
}
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalAssignment.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4assignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

/** Affectations à des variables globales.
 * Cette classe est nouvelle dans ILP4. */

public class CEASTglobalAssignment
extends CEASTassignment
implements IAST4globalAssignment {

    public CEASTglobalAssignment (final IAST4globalVariable variable,
                                  final IAST4expression value) {
        super(variable, value);
    }

    @Override
    @ILPvariable
    public IAST4globalVariable getVariable () {
      return CEAST.narrowToIAST4globalVariable(getDelegate().getVariable());
    }

    /** Interprétation. */

    @Override
    public Object eval (final ILexicalEnvironment lexenv, final ICommon common)
    throws EvaluationException {
        final Object newValue = getValue().eval(lexenv, common);
        common.updateGlobal(getVariable().getName(), newValue);
        return newValue;
    }

    @Override
```

```
    public IAST4assignment normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory)
    throws NormalizeException {
        return factory.newGlobalAssignment(
                CEAST.narrowToIAST4globalVariable(
                        getVariable().normalize(lexenv, common, factory) ),
                getValue().normalize(lexenv,common, factory) );
    }

    @Override
    public <Data, Result, Exc extends Throwable> Result
      accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
        return visitor.visit(this, data);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalFunctionVariable.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;


/** Les variables globales nommant les fonctions globales forment une
 * classe à part car elles mènent à la définition de la fonction
 * qu'elles nomment. */
public class CEASTglobalFunctionVariable
extends CEASTglobalVariable
implements IAST4globalFunctionVariable {

    public CEASTglobalFunctionVariable (final String name) {
        super(name);
    }
    public CEASTglobalFunctionVariable (
            final String name,
            final IAST4functionDefinition functionDefinition ) {
        super(name);
        this.function = functionDefinition;
    }
    private IAST4functionDefinition function;

    /** Obtenir la définition de la fonction ainsi nommée. */
    public IAST4functionDefinition getFunctionDefinition () {
        assert(this.function != null);
        return this.function;
    }

    /** Modifier la définition de la fonction ainsi nommée. */
    public void setFunctionDefinition (IAST4functionDefinition function) {
        this.function = function;
    }

    /** Génération de variables de fonctions globales utilitaires. Leur
     * nom en C débute par le préfixe "ilp" afin de ne pas perturber les
     * variables du langage ILP.*/
    public static IAST4globalFunctionVariable generateGlobalFunctionVariable(
            final IAST4Factory factory) {
        return factory.newGlobalFunctionVariable("ilpFUNCTION");
    }

    /** Génération d'une déclaration globale d'une variable globale
     * nommant une fonction globale. */
    @Override
    public void compileGlobalDeclaration (
            final StringBuffer buffer,
            final ICgenLexicalEnvironment lexenv,
            final ICgenEnvironment common )
    throws CgenerationException {
        getFunctionDefinition().compileHeader(buffer, lexenv, common);
    }

    /** Compilation d'une variable en C pour en obtenir sa valeur.
     * Comme elle est de type ILP_Primitive, on recoltera un avertissement
     * de la part du typeur car, en ISO C, on ne peut convertir (ni dans un
     * sens, ni dans l'autre) un pointeur sur une fonction et un pointeur
     * sur une donnee. Cf. http://www.lysator.liu.se/c/rat/c2.html */
    @Override
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final IDestination destination)
      throws CgenerationException {
        super.compile(buffer, lexenv, common, destination);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalInvocation.java

```
package fr.upmc.ilp.ilp4.ast;

import java.util.Set;

import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

/** Les invocations aux fonctions globales. C'est une classe technique
 * introduite par la normalisation. */

public class CEASTglobalInvocation
extends CEASTinvocation
implements IAST4globalInvocation {

  public CEASTglobalInvocation (final IAST4globalFunctionVariable function,
                                final IAST4expression[] argument) {
      super(new CEASTreference(function), argument);
      this.function = function;
  }
  private final IAST4globalFunctionVariable function;

  @ILPvariable
  public IAST4globalFunctionVariable getFunctionGlobalVariable () {
    return this.function;
  }

  /** Interprétation prenant en compte l'éventuelle intégration. */

  @Override
  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    if ( this.inlined == null ) {
        return super.eval(lexenv, common);
    } else {
      return this.inlined.eval(lexenv, common);
    }
  }

  @Override
  public void compile (final StringBuffer buffer,
```

```
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common,
                       final IDestination destination)
  throws CgenerationException {
  if ( this.inlined == null ) {
      // L'invocation n'a pas été intégrée.
      compileInvocation(buffer, lexenv, common, destination);
  } else {
    // L'invocation a été intégrée.
    buffer.append("/* Appel intégré à ");
    buffer.append(getFunctionGlobalVariable().getMangledName());
    buffer.append(" */");
    this.inlined.compile(buffer, lexenv, common, destination);
  }
}

@Override
public IAST4expression normalize (
        final INormalizeLexicalEnvironment lexenv,
        final INormalizeGlobalEnvironment common,
        final IAST4Factory factory )
  throws NormalizeException {
  // On vérifie au passage l'arité:
  checkArity();
  final IAST4globalFunctionVariable function_ =
      CEAST.narrowToIAST4globalFunctionVariable(
              getFunctionGlobalVariable().normalize(lexenv, common, factory));
  final IAST4expression[] arguments = getArguments();
  IAST4expression[] argument_ = new IAST4expression[arguments.length];
  for ( int i = 0 ; i<arguments.length ; i++ ) {
    argument_[i] = arguments[i].normalize(lexenv, common, factory);
  }
  return factory.newGlobalInvocation(function_, argument_);
}

/** Verifier que l'invocation a la bonne arite vis-a-vis de la definition
 * de la fonction globale.
 */
@Override
public void checkArity ()
throws NormalizeException {
    final IAST4functionDefinition fd =
        getFunctionGlobalVariable().getFunctionDefinition();
    final int arity = fd.getVariables().length;
    if ( arity != getArguments().length ) {
        final String msg = "arity error";
        throw new NormalizeException(msg);
    }
}

public void compileInvocation (final StringBuffer buffer,
                               final ICgenLexicalEnvironment lexenv,
                               final ICgenEnvironment common,
                               final IDestination destination )
throws CgenerationException {
    IAST4expression[] args = getArguments();
    IAST4localVariable[] tmps = new IAST4localVariable[args.length];
    ICgenLexicalEnvironment bodyLexenv = lexenv;
    buffer.append("{ ");
    for ( int i=0; i<args.length ; i++ ) {
        tmps[i] = CEASTlocalVariable.generateVariable();
        tmps[i].compileDeclaration(buffer, lexenv, common);
        bodyLexenv = bodyLexenv.extend(tmps[i]);
    }
    for ( int i=0 ; i<args.length ; i++ ) {
        args[i].compile(buffer, bodyLexenv, common,
                new AssignDestination(tmps[i]) );
        buffer.append(";\n");
    }
    destination.compile(buffer, bodyLexenv, common);
    buffer.append(getFunctionGlobalVariable().getMangledName());
    buffer.append("(");
    for ( int i=0 ; i<(args.length-1) ; i++ ) {
        buffer.append(tmps[i].getMangledName());
        buffer.append(", ");
    }
    if ( args.length > 0 ) {
        buffer.append(tmps[args.length-1].getMangledName());
        }
        buffer.append(");\n}\n");
    }

    @Override
    public void computeInvokedFunctions ()
    throws FindingInvokedFunctionsException {
        addInvokedFunction(getFunctionGlobalVariable());
        super.computeInvokedFunctions();
    }

    /** Intégration de la fonction globale invoquée (si non
     * récursive) et si non déjà intégrée. */

    @Override
    public void inline (IAST4Factory factory) throws InliningException {
      if ( this.getInlined() != null ) {
          return;
      } else {
          // On analyse les arguments!
          for ( IAST4expression arg : getArguments() ) {
              arg.inline(factory);
          }
          final IAST4functionDefinition function =
              getFunctionGlobalVariable().getFunctionDefinition();
          if ( function.isRecursive() ) {
              // On n'intègre pas les fonctions récursives!
              return;
          } else {
              // La fonction a toutes les qualités requises, on l'intègre!
              this.setInlined(factory.newLocalBlock(
                      function.getVariables(),
                      getArguments(),
                      function.getBody()));
              // inlined.inline(); // deja fait quand function fut analysée.
              return;
          }
      }
    }

    /** Rendre la version integree de l'expression. */
    @Override
    public @OrNull IAST4expression getInlined () {
        return this.inlined;
    }
    public void setInlined(IAST4expression inlined) {
        this.inlined = inlined;
    }
    // Seules les invocations a des fonctions globales sont integrees.
    private IAST4expression inlined = null;

    /** Suivre l'expression integree pour determiner les variables globales. */

    @Override
    public void findGlobalVariables (final Set<IAST2variable> globalvars,
                                     final ICgenLexicalEnvironment lexenv ) {
      if ( this.inlined != null ){
          this.inlined.findGlobalVariables(globalvars, lexenv);
      } else {
          super.findGlobalVariables(globalvars, lexenv);
      }
    }

    @Override
    public <Data, Result, Exc extends Throwable> Result
      accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
        return visitor.visit(this, data);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTglobalVariable.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
```

```
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

/** Les variables globales.  */

public class CEASTglobalVariable
extends CEASTvariable
implements IAST4globalVariable {

  public CEASTglobalVariable (final String name) {
      super(name);
  }

  /** Génération de variables globales utilitaires. Leur nom en C
   * débute par le préfixe "ilp" afin de ne pas perturber les
   * variables du langage ILP.*/

  public static synchronized CEASTglobalVariable generateGlobalVariable () {
    return new CEASTglobalVariable("ilpGLOBAL");
  }

  /** Une fois normalisee, une variable reste normalisee. */

  @Override
  public IAST4globalVariable normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common,
          final IAST4Factory factory )
  throws NormalizeException {
      final IAST4globalVariable gv = common.isPresent(this);
      if ( gv != null ) {
          return gv;
      } else {
          // l'incorporer comme variable globale:
          common.add(this);
          return this;
      }
  }

  /** Interprétation d'une référence à une variable globale. */

  @Override
  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    return common.globalLookup(this);
  }

  /** Compilation d'une variable en C pour en obtenir sa valeur. */
  public void compile (final StringBuffer buffer,
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common,
                       final IDestination destination)
    throws CgenerationException {
    destination.compile(buffer, lexenv, common);
    buffer.append(getMangledName());
  }

  /** Tentative de génération d'une déclaration locale pour une
   * variable globale. */
  //NOTE: cette methode ne devrait jamais etre invoquee. Modifier les
  // interfaces IGlobalVariable pour que ce soit le cas.
  @Override
  public void compileDeclaration (final StringBuffer buffer,
                                  final ICgenLexicalEnvironment lexenv,
                                  final ICgenEnvironment common)
    throws CgenerationException {
    final String msg = "Cannot locally declare global variable: " + getName();
    throw new CgenerationException(msg);
  }

  /** Génération d'une déclaration globale d'une variable globale. */

  public void compileGlobalDeclaration (final StringBuffer buffer,
                                        final ICgenLexicalEnvironment lexenv,
                                        final ICgenEnvironment common)
    throws CgenerationException {
    buffer.append("static ILP_Object ");
    buffer.append(getMangledName());
    buffer.append(";\n");
  }

  @Override
  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTinstruction.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp4.interfaces.IAST4instruction;

/**
 * Cette classe n'est là que pour les sous-classes n'ayant pas
 * besoin de délégation ce qui n'est pas le cas des AST d'ILP4.
 */
public abstract class CEASTinstruction
extends CEASTexpression
implements IAST4instruction {}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTinteger.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2integer;
import fr.upmc.ilp.ilp4.interfaces.IAST4integer;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Les constantes entieres. */

public class CEASTinteger
extends CEASTconstant implements IAST4integer  {

    public CEASTinteger (final String value) {
        super(new fr.upmc.ilp.ilp2.ast.CEASTinteger(value));
    }

    public static IAST2integer<CEASTparseException> parse (
            final Element e, final IParser parser)
    throws CEASTparseException {
        return fr.upmc.ilp.ilp2.ast.CEASTinteger.parse(e, parser);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTinvocation.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2invocation;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
```

```
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4invocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4reference;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** C'est une classe technique dont la normalisation mene a
 * l'un des cas particuliers reconnus.
 */

public class CEASTinvocation
extends CEASTdelegableExpression
implements IAST4invocation {

    public CEASTinvocation (final IAST4expression function,
                            final IAST4expression[] argument) {
        this.delegate =
            new fr.upmc.ilp.ilp2.ast.CEASTinvocation(function, argument);
    }
    protected fr.upmc.ilp.ilp2.ast.CEASTinvocation delegate;

    @Override
    public fr.upmc.ilp.ilp2.ast.CEASTinvocation getDelegate () {
        return this.delegate;
    }

    public static IAST2invocation<CEASTparseException> parse (
            final Element e, final IParser parser)
    throws CEASTparseException {
        return fr.upmc.ilp.ilp2.ast.CEASTinvocation.parse(e, parser);
    }

    @ILPexpression(isArray=true)
    public IAST4expression[] getArguments() {
      return CEAST.narrowToIAST4expressionArray(this.delegate.getArguments());
    }
    public IAST4expression getArgument (int i) {
      return CEAST.narrowToIAST4expression(this.delegate.getArgument(i));
    }
    public int getArgumentsLength () {
        return this.delegate.getArgumentsLength();
    }
    @ILPexpression
    public IAST4expression getFunction() {
        return CEAST.narrowToIAST4expression(this.delegate.getFunction());
    }

    /** Normaliser une invocation. Si la fonction invoquée est globale,
     * le résultat de la normalisation sera une instance de
     * CEASTglobalInvocation et la variable nommant la fonction sera une
     * CEASTglobalFunctionVariable. */

    @Override
    public IAST4expression normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory )
    throws NormalizeException {
        final IAST4expression function_ =
            getFunction().normalize(lexenv, common, factory);
        /* Les arguments seront normalises dans les sous-classes. */
        // Discrimination
        if ( function_ instanceof IAST4reference ) {
            IAST4variable var = ((IAST4reference) function_).getVariable();
            if ( var instanceof IAST4globalFunctionVariable ) {
                final IAST4globalFunctionVariable gv =
                    CEAST.narrowToIAST4globalFunctionVariable(var);
                final IAST4invocation result =
                    factory.newGlobalInvocation(gv, getArguments());
                return result.normalize(lexenv, common, factory);

            } else {
                final IAST4invocation result =
                    factory.newComputedInvocation(function_, getArguments());
                return result.normalize(lexenv, common, factory);
            }
        } else {
            final IAST4invocation result =
                factory.newComputedInvocation(function_, getArguments());
            return result.normalize(lexenv, common, factory);
        }
    }
    // NOTE: pas de super.normalize() dans les sous-classes, ca bouclerait.
    // Utiliser normalizeInvocation() sur fonction ?

    public void checkArity () throws NormalizeException {
        final String msg = "Should not be called!";
        throw new NormalizeException(msg);
    }

    @Override
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final IDestination destination )
    throws CgenerationException {
        final String msg = "Should not compile a vanilla invocation!";
        throw new CgenerationException(msg);
    }

    /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
    @Override
    public void findInvokedFunctions () {
        findAndAdjoinToInvokedFunctions(getFunction());
        for ( IAST4expression arg : getArguments() ) {
            findAndAdjoinToInvokedFunctions(arg);
        }
    }

    /* Obsolète de par CEAST.inline() qui use de réflexivité.
    @Override
    public void inline () {
        getFunction().inline();
        for ( IAST4expression arg : getArguments() ) {
            arg.inline();
        }
    }
    */

    public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTlocalAssignment.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4localAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

/** Affectation à des variables locales.
 * Cette classe est nouvelle dans ILP4.  */

public class CEASTlocalAssignment
extends CEASTassignment
implements IAST4localAssignment {

    public CEASTlocalAssignment (final IAST4localVariable variable,
                                 final IAST4expression value) {
        super(variable, value);
    }
```

```
    @Override
    @ILPvariable
    public IAST4localVariable getVariable () {
      return CEAST.narrowToIAST4localVariable(getDelegate().getVariable());
    }

    /** Interprétation. */

    @Override
    public Object eval (final ILexicalEnvironment lexenv, final ICommon common)
    throws EvaluationException {
        final Object newValue = getValue().eval(lexenv, common);
        lexenv.update(getVariable(), newValue);
        return newValue;
    }

    @Override
    public IAST4localAssignment normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory)
    throws NormalizeException {
        return factory.newLocalAssignment(
          CEAST.narrowToIAST4localVariable(
                  getVariable().normalize(lexenv, common, factory)),
          getValue().normalize(lexenv,common, factory) );
    }

    @Override
    public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
        return visitor.visit(this, data);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTlocalBlock.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2localBlock;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4localBlock;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Bloc local. */

public class CEASTlocalBlock
extends CEASTdelegableInstruction
implements IAST4localBlock {

    /** Création d'un bloc local à partir de ses constituants
     * normalisés. On suppose ici que les tailles des deux vecteurs sont
     * bien égales. */

    public CEASTlocalBlock (
            final IAST4variable[] variable,
            final IAST4expression[] initialization,
            final IAST4expression body ) {
        assert(variable.length == initialization.length);
        this.delegate =
            new fr.upmc.ilp.ilp2.ast.CEASTlocalBlock(
                    variable, initialization, body );
    }
    private fr.upmc.ilp.ilp2.ast.CEASTlocalBlock delegate;

    @Override
    public fr.upmc.ilp.ilp2.ast.CEASTlocalBlock getDelegate () {
        return this.delegate;
    }

    // ou IAST4localBlock ???
    public static IAST2localBlock<CEASTparseException> parse (
            final Element e, final IParser parser)
    throws CEASTparseException {
      return fr.upmc.ilp.ilp2.ast.CEASTlocalBlock.parse(e, parser);
    }

    @ILPvariable(isArray=true)
    public IAST4variable[] getVariables () {
      return CEAST.narrowToIAST4variableArray(this.delegate.getVariables());
    }
    public IAST4localVariable[] getLocalVariables () {
        return CEAST.narrowToIAST4localVariableArray(this.delegate.getVariables());
      }
    @ILPexpression(isArray=true)
    public IAST4expression[] getInitializations () {
      return CEAST.narrowToIAST4expressionArray(this.delegate
                                               .getInitializations());
    }
    @ILPexpression
    public IAST4expression getBody() {
      return CEAST.narrowToIAST4expression(this.delegate.getBody());
    }

    @Override
    public IAST4expression normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory )
    throws NormalizeException {
        if ( getVariables().length == 0 ) {
            // On simplifie le bloc à son corps:
            return getBody().normalize(lexenv, common, factory);
        }

        IAST4expression[] initializations = getInitializations();
        IAST4expression[] initialization_ =
            new IAST4expression[initializations.length];
        for ( int i = 0 ; i<initializations.length ; i++ ) {
            initialization_[i] =
                initializations[i].normalize(lexenv, common, factory);
        }
        // Normalisation du corps:
        INormalizeLexicalEnvironment bodyLexenv = lexenv;
        IAST4variable[] variables = getVariables();
        IAST4variable[] variables_ = new IAST4variable[variables.length];
        for ( int i = 0 ; i<variables.length ; i++ ) {
            final IAST4variable var = variables[i];
            variables_[i] = factory.newLocalVariable(var.getName());
            bodyLexenv = bodyLexenv.extend(variables_[i]);
        }
        final IAST4expression body_ =
            getBody().normalize(bodyLexenv, common, factory);
        return factory.newLocalBlock(variables_, initialization_, body_);
    }

    @Override
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final IDestination destination )
    throws CgenerationException {
        final IAST4localVariable[] vars = getLocalVariables();
        final IAST4expression[] inits = getInitializations();
        IAST4localVariable[] temp = new IAST4localVariable[vars.length];
        ICgenLexicalEnvironment templexenv = lexenv;

        buffer.append("{\n");
```

```
        for (int i = 0; i < vars.length; i++) {
            temp[i] = CEASTlocalVariable.generateVariable();
            templexenv = templexenv.extend(temp[i]);
            temp[i].compileDeclaration(buffer, templexenv, common);
        }

        for (int i = 0; i < vars.length; i++) {
            inits[i].compile(buffer, lexenv, common,
                    new AssignDestination(temp[i]) );
            buffer.append(";\n");
        }

        buffer.append("{\n");
        ICgenLexicalEnvironment bodylexenv = templexenv;
        for (int i = 0; i < vars.length; i++) {
            bodylexenv = bodylexenv.extend(vars[i]);
            vars[i].compileDeclaration(buffer, bodylexenv, common);
        }

        for (int i = 0; i < vars.length; i++) {
            buffer.append(vars[i].getMangledName());
            buffer.append(" = ");
            buffer.append(temp[i].getMangledName());
            buffer.append(";\n");
        }

        getBody().compile(buffer, bodylexenv, common, destination);
        buffer.append(";}\n}\n");
    }

    /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
    @Override
    public void findInvokedFunctions () {
        for ( IAST4expression init : getInitializations() ) {
            findAndAdjoinToInvokedFunctions(init);
        }
        findAndAdjoinToInvokedFunctions(getBody());
    }

    /* Obsolète de par CEAST.inline() qui use de réflexivité.
    @Override
    public void inline () {
        for ( IAST4expression init : this.getInitializations() ) {
            init.inline();
        }
        getBody().inline();
    }
    */

    public <Data, Result, Exc extends Throwable> Result
      accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
        return visitor.visit(this, data);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTlocalVariable.java

```
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

/** Les variables: leur interprétation et compilation. Attention:
 * cette classe est raffinée en plusieurs sous-classes de comportement
 * légérement différents: les variables globales et prédéfinies. */

public class CEASTlocalVariable
```

```java
extends CEASTvariable
implements IAST4localVariable {

/** Créer une variable avec un certain nom. Le nom peut être modifié
 * afin de se conformer à C, il faut donc toujours demander le nom
 * de la variable plutôt que de le supposer.  */

public CEASTlocalVariable (final String name) {
  super(name);
}

/** Génération de variables temporaires. Leur nom en C débute par le
 * préfixe "ilp" afin de ne pas perturber les variables du langage
 * ILP. */

public static synchronized CEASTlocalVariable generateVariable () {
  return new CEASTlocalVariable("ilpLOCAL");
}

/** Une fois normalisee, une variable reste normalisee. */

@Override
public IAST4variable normalize (
        final INormalizeLexicalEnvironment lexenv,
        final INormalizeGlobalEnvironment common,
        final IAST4Factory factory )
throws NormalizeException {
    final IAST4variable lv = lexenv.isPresent(this);
    if ( lv != null ) {
        return lv;
    } else {
        final IAST4globalVariable global =
                factory.newGlobalVariable(getName());
        common.add(global);
        return global;
    }
}

/** Interprétation d'une référence à une variable locale. */

@Override
public Object eval (final ILexicalEnvironment lexenv,
                    final ICommon common)
  throws EvaluationException {
  return lexenv.lookup(this);
}

/** Compilation en C d'une référence à une variable locale. */

public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common,
                     final IDestination destination)
  throws CgenerationException {
  destination.compile(buffer, lexenv, common);
  buffer.append(getMangledName());
}

/** Génération d'une déclaration introduisant une variable locale. */
@Override
public void compileDeclaration (final StringBuffer buffer,
                                final ICgenLexicalEnvironment lexenv,
                                final ICgenEnvironment common)
  throws CgenerationException {
  buffer.append("ILP_Object ");
  buffer.append(getMangledName());
  buffer.append(";\n");
}

/** Tentative de génération d'une déclaration globale. Une variable locale
 * n'est pas globale. */

public void compileGlobalDeclaration (final StringBuffer buffer,
                                      final ICgenLexicalEnvironment lexenv,
                                      final ICgenEnvironment common)
  throws CgenerationException {
  final String msg = "Non global variable " + getName();
  throw new CgenerationException(msg);
```

```java
  }

  @Override
  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASToperation.java

```java
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2operation;
import fr.upmc.ilp.ilp4.interfaces.IAST4operation;

/** Les opérations (unaires ou binaires). */

public abstract class CEASToperation
extends CEASTdelegableExpression
implements IAST4operation {

    public CEASToperation () {}

    //NOTE: une methode abstraite heritee raffinant le type.
    @Override
    public abstract IAST2operation<CEASTparseException> getDelegate ();

    public String getOperatorName () {
        return getDelegate().getOperatorName();
    }
    public int getArity () {
        return getDelegate().getArity();
    }

}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTprimitiveInvocation.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2invocation;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4invocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4primitiveInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Les invocations de primitives. */

public class CEASTprimitiveInvocation
  extends CEASTinvocation
  implements IAST4primitiveInvocation {

    public CEASTprimitiveInvocation (final IAST4globalVariable function,
                                     final IAST4expression[] argument) {
        super(new CEASTreference(function), argument);
        this.primitive = function;
        this.delegate = new fr.upmc.ilp.ilp2.ast.CEASTprimitiveInvocation(
                             getPrimitiveName(), argument );
  }
  private final IAST4globalVariable primitive;

  public String getPrimitiveName() {
      return getFunctionGlobalVariable().getName();
  }

  @ILPvariable
  public IAST4globalVariable getFunctionGlobalVariable () {
      return this.primitive;
  }

  @Override
  public IAST4expression getFunction() {
      final String msg = "Internal problem! Must never be invoked!";
      throw new RuntimeException(msg);
  }

  public static IAST2invocation<CEASTparseException> parse (
            final Element e, final IParser parser)
  throws CEASTparseException {
    return fr.upmc.ilp.ilp2.ast.CEASTprimitiveInvocation.parse(e, parser);
  }

@Override
public IAST4invocation normalize (
        final INormalizeLexicalEnvironment lexenv,
        final INormalizeGlobalEnvironment common,
        final IAST4Factory factory )
  throws NormalizeException {
  final IAST4globalVariable function_ =
      CEAST.narrowToIAST4globalVariable(
              getFunctionGlobalVariable().normalize(lexenv, common, factory) );
  final IAST4expression[] arguments = getArguments();
  IAST4expression[] argument_ = new IAST4expression[arguments.length];
  for ( int i = 0 ; i<arguments.length ; i++ ) {
    argument_[i] =
      arguments[i].normalize(lexenv, common, factory);
  }
  // On vérifie au passage l'arité:
  checkArity();
  return factory.newPrimitiveInvocation(function_, argument_);
}

/** Verifier que l'invocation a la bonne arité vis-à-vis de la définition
 * de la fonction globale.  */
@Override
public void checkArity ()
throws NormalizeException {
    // TODO
}

@Override
public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common,
                     final IDestination destination )
throws CgenerationException {
    IAST4expression[] args = getArguments();
    IAST4localVariable[] tmps = new IAST4localVariable[args.length];
    ICgenLexicalEnvironment bodyLexenv = lexenv;
    buffer.append("{\n");
    for ( int i=0; i<args.length ; i++ ) {
        tmps[i] = CEASTlocalVariable.generateVariable();
        tmps[i].compileDeclaration(buffer, lexenv, common);
        bodyLexenv = bodyLexenv.extend(tmps[i]);
    }
    for ( int i=0 ; i<args.length ; i++ ) {
        args[i].compile(buffer, bodyLexenv, common,
                new AssignDestination(tmps[i]) );
        buffer.append(";\n");
    }
    destination.compile(buffer, bodyLexenv, common);
    buffer.append(common.compilePrimitive(getPrimitiveName()));
    buffer.append("(");
    for ( int i=0 ; i<(args.length-1) ; i++ ) {
        buffer.append(tmps[i].getMangledName());
        buffer.append(", ");
```

```java
      }
      if ( args.length > 0 ) {
          buffer.append(tmps[args.length-1].getMangledName());
      }
      buffer.append(");\n}\n");
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
      for ( IAST4expression arg : getArguments() ) {
          arg.inline();
      }
  }
  */

  @Override
  public <Data, Result, Exc extends Throwable> Result
  accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
    return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTprogram.java

```java
package fr.upmc.ilp.ilp4.ast;

import java.util.List;
import java.util.Set;
import java.util.Vector;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4program;
import fr.upmc.ilp.ilp4.interfaces.IAST4sequence;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** La classe d'un programme composé de fonctions globales et
 * d'instructions. Ce n'est pas une expression ni une instruction mais
 * un programme. */

public class CEASTprogram
extends CEAST implements IAST4program {

    public CEASTprogram (final IAST4functionDefinition[] definitions,
                         final IAST4expression body) {
        this.delegate = new fr.upmc.ilp.ilp3.CEASTprogram(definitions, body);
    }
    protected fr.upmc.ilp.ilp3.CEASTprogram delegate;

    public fr.upmc.ilp.ilp3.CEASTprogram getDelegate () {
        return this.delegate;
    }

    public static IAST4program parse (final Element e, final IParser parser)
```

```
            throws CEASTparseException {
        List<IAST2<CEASTparseException>> itemsAsList =
            parser.parseList(e.getChildNodes());
        IAST4[] items = itemsAsList.toArray(new IAST4[0]);
        final List<IAST4functionDefinition> definitions = new Vector<>();
        final List<IAST4expression> instructions = new Vector<>();
        for ( IAST4 item : items ) {
            if ( item instanceof IAST4functionDefinition ) {
                definitions.add((IAST4functionDefinition) item);
            } else if ( item instanceof IAST4expression ) {
                instructions.add((IAST4expression) item);
            } else {
                final String msg = "Should never occur!";
                assert false : msg;
                throw new CEASTparseException(msg);
            }
        }
        IAST4functionDefinition[] defs =
            definitions.toArray(new IAST4functionDefinition[0]);
        IAST4Factory factory = parser.getFactory();
        IAST4sequence body = factory.newSequence(
                instructions.toArray(new IAST4expression[0]));
        return factory.newProgram(defs, body);
    }

    @ILPexpression
    public IAST4expression getBody () {
      return CEAST.narrowToIAST4expression(this.getDelegate().getBody());
    }
    @ILPexpression(isArray=true)
    public IAST4functionDefinition[] getFunctionDefinitions () {
        IAST2functionDefinition<CEASTparseException>[] fds =
            this.getDelegate().getFunctionDefinitions();
        IAST4functionDefinition[] result =
            new IAST4functionDefinition[fds.length];
        System.arraycopy(fds, 0, result, 0, fds.length);
        return result;
    }

    @Override
    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common)
    throws EvaluationException {
        return getDelegate().eval(lexenv, common);
    }

  /** Compiler un programme tout entier. */
  public void compile (final StringBuffer buffer,
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common )
    throws CgenerationException {
      buffer.append("#include <stdio.h>\n");
      buffer.append("#include <stdlib.h>\n");
      buffer.append("\n");
      buffer.append("#include \"ilp.h\"\n");
      buffer.append("#include \"ilpException.h\"\n");
      buffer.append("\n");
      // Declarer les variables globales:
      buffer.append("/* Variables ou prototypes globaux: */\n");
      for ( IAST4globalVariable var : getGlobalVariables() ) {
          var.compileGlobalDeclaration(buffer, lexenv, common);
          if ( ! common.isPresent(var.getName()) ) {
            common.bindGlobal(var);
          }
      }
      IAST4functionDefinition[] definitions = getFunctionDefinitions();
      for ( IAST4functionDefinition fun : definitions ) {
          // On pourrait ne pas compiler les fonctions non recursives car
          // si elles sont integrees, elles ne sont plus invoquees!
          fun.compileHeader(buffer, lexenv, common);
      }
      // Puis le code des fonctions globales:
      buffer.append("\n/* Fonctions globales: */\n");
      for ( IAST4functionDefinition fun : definitions ) {
          fun.compile(buffer, lexenv, common);
      }
      buffer.append("\n");
      buffer.append("static ILP_Object ilp_caught_program () {\n");
      buffer.append("  struct ILP_catcher* current_catcher = ILP_current_catcher;\n");
```

```
      buffer.append("  struct ILP_catcher new_catcher;\n");
      buffer.append("\n");
      buffer.append("  if ( 0 == setjmp(new_catcher._jmp_buf) ) {\n");
      buffer.append("    ILP_establish_catcher(&new_catcher);\n");
      buffer.append("    return ilp_program();\n");
      buffer.append("  };\n");
      buffer.append("  /* Une exception est survenue. */\n");
      buffer.append("  return ILP_current_exception;\n");
      buffer.append("}\n");
      buffer.append("\n");
      buffer.append("int main (int argc, char *argv[]) {\n");
      buffer.append("  ILP_print(ilp_caught_program());\n");
      buffer.append("  ILP_newline();\n");
      buffer.append("  return EXIT_SUCCESS;\n");
      buffer.append("}\n\n");
      buffer.append("\n/* fin */\n");
  }

  /** Le nom de la fonction C correspondant au programme. */
  public static String PROGRAM = "ilp_program";

  /** Compiler une instruction en une chaîne de caractères. C'est une
   * méthode permettant de compiler confortablement tout un programme
   * et d'obtenir le résultat sous forme d'une chaîne qu'il suffira
   * d'écrire dans un fichier pour le compiler avec un compilateur
   * C. */

  public String compile (final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common )
    throws CgenerationException {
    final StringBuffer buffer = new StringBuffer(4095);
    this.compile(buffer, lexenv, common);
    return buffer.toString();
  }

  /** Normaliser un programme dans un environnement lexical et global
   * particuliers. */

  @Override
  public IAST4program normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common,
          final IAST4Factory factory )
    throws NormalizeException {
      // Introduire d'abord toutes les variables globales nommant les
      // fonctions globales et les associer ensemble:
      IAST4functionDefinition[] definitions = getFunctionDefinitions();
      for ( int i = 0 ; i<definitions.length ; i++ ) {
          IAST4globalFunctionVariable gfv =
              factory.newGlobalFunctionVariable(
                      definitions[i].getDefinedVariable().getName());
          gfv.setFunctionDefinition(definitions[i]);
          common.add(gfv);
      }
      // On normalise toutes les definitions
      final IAST4functionDefinition[] definitions_ =
          new IAST4functionDefinition[definitions.length + 1];
      for ( int i = 0 ; i<definitions.length ; i++ ) {
          definitions_[i] = definitions[i].normalize(lexenv, common, factory);
      }
      // Empaqueter le code hors fonction en une fonction globale:
      final IAST4expression body_=
          getBody().normalize(lexenv, common, factory);
      final IAST4globalFunctionVariable program =
          new CEASTglobalFunctionVariable(PROGRAM);
      common.add(program);
      final IAST4functionDefinition bodyAsFunction =
          factory.newFunctionDefinition(program, new IAST4variable[0], body_);
      program.setFunctionDefinition(bodyAsFunction);
      definitions_[definitions.length] =
          bodyAsFunction.normalize(lexenv, common, factory);
      final IAST4expression body__ =
          factory.newGlobalInvocation(program, new CEASTexpression[0]);
      // Finalisation
      IAST4program program_ = factory.newProgram(definitions_, body__);
      return program_;
  }

  /** On calcule pour chaque fonction globale, les fonctions qu'elle
   * invoque puis on calcule la fermeture transitive. L'ensemble des
   * fonctions invoquées du programme n'est constitué que des seules
   * fonctions invoquées par son corps. */

  @Override
  public void computeInvokedFunctions ()
  throws FindingInvokedFunctionsException {
      IAST4functionDefinition[] definitions = getFunctionDefinitions();
      for ( int i = 0 ; i<definitions.length ; i++ ) {
          definitions[i].computeInvokedFunctions();
      }
      boolean shouldContinue = true;
      while ( shouldContinue ) {
          shouldContinue = false;
          for ( int i = 0 ; i<definitions.length ; i++ ) {
              final IAST4functionDefinition currentFunction = definitions[i];
              for ( IAST4globalFunctionVariable gv :
                      currentFunction.getInvokedFunctions()
                          .toArray(IAST4GFV_EMPTY_ARRAY) ) {
                  // currentFunction invoque gv donc elle invoque
                  // (indirectement) les fonctions qu'invoque gv.
                  final IAST4functionDefinition other =
                    gv.getFunctionDefinition();
                  // | et non || comme remarqué par <Jeremie.Lumbroso@etu.upmc.fr>
                  shouldContinue |= currentFunction
                      .addInvokedFunctions(other.getInvokedFunctions());
                  // NOTA: la precedente methode change la collection que l'on
                  // est en train d'inspecter ce qui pose des problemes
                  // d'acces simultanes a cette collection d'ou l'emploi d'un
                  // toArray() plus haut.
              }
          }
      }
      // Savoir ce qu'invoque le programme est de peu d'utilite!
      findAndAdjoinToInvokedFunctions(getBody());
  }
  public static final IAST4globalFunctionVariable[] IAST4GFV_EMPTY_ARRAY =
      new IAST4globalFunctionVariable[0];

  /** Integration de toutes les fonctions non recursives. */

  @Override
  public void inline (IAST4Factory factory) throws InliningException {
      IAST4functionDefinition[] definitions = getFunctionDefinitions();
      for ( IAST4functionDefinition fd : definitions ) {
          fd.inline(factory);
      }
      getBody().inline(factory);
  }

  /** Recensement des variables globales. */
  // Nouvelle version avec visiteur:
  public void computeGlobalVariables () {
      globals = GlobalCollector.getGlobalVariables(this);
  }
  // Ancienne version dépréciée:
  @Deprecated
  public void computeGlobalVariables (final ICgenLexicalEnvironment lexenv) {
      // Cette methode est heritee mais son argument ne sert plus a rien car
      // on a change de mode de calcul des variables globales.
      computeGlobalVariables();
  }
  public IAST4globalVariable[] getGlobalVariables () {
      return this.globals;
  }
  private IAST4globalVariable[] globals = new IAST4globalVariable[0];
  public void setGlobalVariables (IAST4globalVariable[] globals) {
      this.globals = globals;
  }

  @Override
  public void findGlobalVariables (final Set<IAST2variable> globalvars,
          final ICgenLexicalEnvironment lexenv ) {
      throw new RuntimeException("Should not occurr!");
  }

  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
```

```
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTreference.java

```
package fr.upmc.ilp.ilp4.ast;

import java.util.Set;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2reference;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4reference;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

public class CEASTreference
extends CEASTdelegableExpression
implements IAST4reference {

    public CEASTreference(IAST4variable variable) {
        this.delegate =
            new fr.upmc.ilp.ilp2.ast.CEASTreference(variable);
    }
    private final fr.upmc.ilp.ilp2.ast.CEASTreference delegate;

    @Override
    public fr.upmc.ilp.ilp2.ast.CEASTreference getDelegate () {
        return this.delegate;
    }

    @ILPvariable
    public IAST4variable getVariable() {
        return CEAST.narrowToIAST4variable(getDelegate().getVariable());
    }

    public static IAST2reference<CEASTparseException> parse (
            final Element e, final IParser parser) {
        return fr.upmc.ilp.ilp2.ast.CEASTreference.parse(e, parser);
    }

    //NOTE: Accès direct aux champs interdit à partir d'ici!

    @Override
    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common )
    throws EvaluationException {
        return getVariable().eval(lexenv, common);
    }

    @Override
    public void compileExpression (final StringBuffer buffer,
                                   final ICgenLexicalEnvironment lexenv,
                                   final ICgenEnvironment common,
                                   final IDestination destination)
    throws CgenerationException {
        destination.compile(buffer, lexenv, common);
        buffer.append(" ");
        try {
            buffer.append(lexenv.compile(getVariable()));
        } catch (CgenerationException e) {
            buffer.append(common.compileGlobal(getVariable()));
        }
        buffer.append(" ");
```

```java
    }

    /** Une variable est globale si elle n'est pas locale. */

    @Override
    public void findGlobalVariables (final Set<IAST2variable> globalvars,
                                     final ICgenLexicalEnvironment lexenv ) {
        if ( ! lexenv.isPresent(getVariable()) ) {
            globalvars.add(getVariable());
        }
    }

    @Override
    public IAST4reference normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory )
    throws NormalizeException {
        return factory.newReference(
                  getVariable().normalize(lexenv, common, factory));
    }

    public <Data, Result, Exc extends Throwable> Result
      accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
        return visitor.visit(this, data);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTsequence.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2sequence;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4sequence;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Les sequences d'instructions. */

public class CEASTsequence
extends CEASTdelegableInstruction
implements IAST4sequence {

  public CEASTsequence (final IAST4expression[] instructions) {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTsequence(instructions);
  }
  private final fr.upmc.ilp.ilp2.ast.CEASTsequence delegate;

  @Override
  public fr.upmc.ilp.ilp2.ast.CEASTsequence getDelegate () {
      return this.delegate;
  }

  @ILPexpression(isArray=true)
  public IAST4expression[] getInstructions () {
    return CEAST.narrowToIAST4expressionArray(
                     getDelegate().getInstructions() );
  }
  public int getInstructionsLength () {
      return getDelegate().getInstructions().length;
  }
  public IAST4expression getInstruction (int i) throws CEASTparseException {
      return CEAST.narrowToIAST4expression(
             getDelegate().getInstruction(i) );
  }

  public static IAST2sequence<CEASTparseException> parse (
          final Element e, final IParser parser)
```

```java
    throws CEASTparseException {
    return fr.upmc.ilp.ilp2.ast.CEASTsequence.parse(e, parser);
  }

  /** Renvoyer une séquence d'instructions réduite à une seule
   * instruction ne faisant rien. */

  public static IAST4expression voidSequence () {
      return CEASTexpression.voidExpression();
  }

  /** Si la séquence ne comporte qu'une unique expression, la
   * remplacer par cette unique expression (normalisée elle-même bien
   * sûr). */

  @Override
  public IAST4expression normalize (final INormalizeLexicalEnvironment lexenv,
                                    final INormalizeGlobalEnvironment common,
                                    final IAST4Factory factory )
    throws NormalizeException {
      IAST4expression[] instructions = getInstructions();
      IAST4expression[] instructions_ = new IAST4expression[instructions.length];
      for ( int i = 0 ; i< instructions.length ; i++ ) {
          instructions_[i] =
              instructions[i].normalize(lexenv, common, factory);
      }
      if ( instructions.length == 1 ) {
          return instructions_[0];
      } else {
          return factory.newSequence(instructions_);
      }
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      for ( IAST4expression instr : getInstructions() ) {
          findAndAdjoinToInvokedFunctions(instr);
      }
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
      for ( IAST4expression instr : getInstructions() ) {
          instr.inline();
      }
  }
  */

  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTstring.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2string;
import fr.upmc.ilp.ilp4.interfaces.IAST4string;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Les constantes chaines de caracteres. */
public class CEASTstring
extends CEASTconstant implements IAST4string {

    public CEASTstring (final String value) {
        super(new fr.upmc.ilp.ilp2.ast.CEASTstring(value));
    }

    public static IAST2string<CEASTparseException> parse (
            final Element e, final IParser parser)
    throws CEASTparseException {
        return fr.upmc.ilp.ilp2.ast.CEASTstring.parse(e, parser);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTtry.java

```
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp3.IAST3try;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4try;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

public class CEASTtry
  extends CEASTdelegableInstruction
  implements IAST4try {

  protected CEASTtry (final IAST4expression body,
                      final IAST4variable caughtExceptionVariable,
                      final IAST4expression catcher,
                      final IAST4expression finallyer) {
      this.delegate = new fr.upmc.ilp.ilp3.CEASTtry(
              body, caughtExceptionVariable, catcher, finallyer );
  }
  private fr.upmc.ilp.ilp3.CEASTtry delegate;

  @Override
  public fr.upmc.ilp.ilp3.CEASTtry getDelegate () {
      return this.delegate;
  }

  @ILPexpression
  public IAST4expression getBody () {
    return CEAST.narrowToIAST4expression(this.delegate.getBody());
  }
  @ILPvariable(neverNull=false)
  public @OrNull IAST4variable getCaughtExceptionVariable () {
      IAST2variable cev = this.delegate.getCaughtExceptionVariable();
      if ( null != cev ) {
          return CEAST.narrowToIAST4variable(cev);
      } else {
          return null;
      }
  }
  @ILPexpression(neverNull=false)
  public @OrNull IAST4expression getCatcher () {
      IAST2instruction<CEASTparseException> result = this.delegate.getCatcher();
      if ( null != result ) {
          return CEAST.narrowToIAST4expression(result);
      } else {
          return null;
      }
  }
  @ILPexpression(neverNull=false)
  public @OrNull IAST4expression getFinallyer () {
      IAST2instruction<CEASTparseException> result = this.delegate.getFinallyer();
      if ( null != result ) {
          return CEAST.narrowToIAST4expression(result);
      } else {
          return null;
      }
  }

  public static IAST3try<CEASTparseException> parse (
          final Element e, final IParser parser)
  throws CEASTparseException {
    IAST3try<CEASTparseException> delegate =
      fr.upmc.ilp.ilp3.CEASTtry.parse(e, parser);
    IAST4Factory factory = parser.getFactory();
    IAST4expression body_ =
        CEAST.narrowToIAST4expression(delegate.getBody());
    IAST4variable cev = null;
    IAST4expression catcher_ = null;
    if ( null != delegate.getCaughtExceptionVariable() ) {
        IAST2variable cev_ = delegate.getCaughtExceptionVariable();
        cev = (IAST4variable) factory.newVariable(cev_.getName());
        catcher_ = CEAST.narrowToIAST4expression(delegate.getCatcher());
    }
    IAST4expression finallyer_ = null;
    if ( null != delegate.getFinallyer() ) {
        finallyer_ = CEAST.narrowToIAST4expression(delegate.getFinallyer());
    }
    return factory.newTry(body_, cev, catcher_, finallyer_);
  }

  @Override
  public IAST4expression normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common,
          final IAST4Factory factory )
    throws NormalizeException {
    IAST4expression body_ =
        getBody().normalize(lexenv, common, factory);
    IAST4expression catcher_ = getCatcher();
    IAST4variable caughtVar_ = null;
    if ( catcher_ != null ) {
        caughtVar_ = factory.newLocalVariable(
            getCaughtExceptionVariable().getName());
        final INormalizeLexicalEnvironment catcherLexenv =
            lexenv.extend(caughtVar_);
        catcher_ = catcher_.normalize(catcherLexenv, common, factory);
    }
    IAST4expression finallyer_ = null;
    if ( null != getFinallyer() ) {
        finallyer_ = getFinallyer().normalize(lexenv, common, factory);
    }
    return factory.newTry(body_, caughtVar_, catcher_, finallyer_);
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      findAndAdjoinToInvokedFunctions(getBody());
      if ( getCatcher() != null ) {
          findAndAdjoinToInvokedFunctions(getCatcher());
      };
      if ( getFinallyer() != null ) {
          findAndAdjoinToInvokedFunctions(getFinallyer());
      };
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
      getBody().inline();
      if ( getCatcher() != null ) {
          getCatcher().inline();
      };
      if ( getFinallyer() != null ) {
          getFinallyer().inline();
      };
  }
  */

  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTunaryBlock.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.annotation.ILPvariable;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryBlock;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4unaryBlock;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Le bloc unaire: interprétation et compilation.
 *
 * Il n'hérite pas du bloc local car il n'y a pas de code commun.
 */

public class CEASTunaryBlock
  extends CEASTdelegableInstruction
  implements IAST4unaryBlock {

  public CEASTunaryBlock (final IAST4variable variable,
                          final IAST4expression initialization,
                          final IAST4expression body)
  {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTunaryBlock(
                  variable, initialization, body );
  }
  private fr.upmc.ilp.ilp2.ast.CEASTunaryBlock delegate;

  @Override
  public fr.upmc.ilp.ilp2.ast.CEASTunaryBlock getDelegate () {
      return this.delegate;
  }

  @ILPvariable
  public IAST4variable getVariable () {
    return CEAST.narrowToIAST4variable(this.delegate.getVariable());
  }
  public IAST4localVariable getLocalVariable () {
        return CEAST.narrowToIAST4localVariable(this.delegate.getVariable());
      }
  @ILPexpression
  public IAST4expression getInitialization () {
    return CEAST.narrowToIAST4expression(this.delegate.getInitialization());
  }
  @ILPexpression
  public IAST4expression getBody() {
    return CEAST.narrowToIAST4expression(this.delegate.getBody());
  }

  public static IAST2unaryBlock<CEASTparseException> parse (
          final Element e, final IParser parser)
  throws CEASTparseException {
    return fr.upmc.ilp.ilp2.ast.CEASTunaryBlock.parse(e, parser);
  }

  @Override
  public IAST4expression normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common,
          final IAST4Factory factory )
    throws NormalizeException {
      IAST4variable var = factory.newLocalVariable(getVariable().getName());
      final INormalizeLexicalEnvironment bodyLexenv = lexenv.extend(var);
      return factory.newUnaryBlock(
              var,
```

```java
              getInitialization().normalize(lexenv, common, factory),
              getBody().normalize(bodyLexenv, common, factory) );
  }

  @Override
  public void compile (final StringBuffer buffer,
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common,
                       final IDestination destination)
    throws CgenerationException {
    final IAST4variable tmp = CEASTlocalVariable.generateVariable();
    final ICgenLexicalEnvironment lexenv2 = lexenv.extend(tmp);
    final ICgenLexicalEnvironment lexenv3 =
        lexenv2.extend(getVariable());

    buffer.append("{\n");
    tmp.compileDeclaration(buffer, lexenv2, common);
    getInitialization().compile(buffer, lexenv, common,
            new AssignDestination(tmp) );
    buffer.append(";\n");

    buffer.append("{\n");
    getVariable().compileDeclaration(buffer, lexenv2, common);
    buffer.append(getLocalVariable().getMangledName());
    buffer.append(" = ");
    buffer.append(tmp.getMangledName());
    buffer.append(";\n");

    getBody().compile(buffer, lexenv3, common, destination);
    buffer.append(";}\n}\n");
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      findAndAdjoinToInvokedFunctions(getInitialization());
      findAndAdjoinToInvokedFunctions(getBody());
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
    getInitialization().inline();
    getBody().inline();
  }
  */

  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTunaryOperation.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryOperation;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4unaryOperation;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** Les opérations unaires. */
```

```java
public class CEASTunaryOperation
  extends CEASToperation
  implements IAST4unaryOperation {

  public CEASTunaryOperation (final String operatorName,
                              final IAST4expression operand)
  {
    this.delegate =
        new fr.upmc.ilp.ilp2.ast.CEASTunaryOperation(operatorName, operand);
  }
  private fr.upmc.ilp.ilp2.ast.CEASTunaryOperation delegate;

  @Override
  public fr.upmc.ilp.ilp2.ast.CEASTunaryOperation getDelegate () {
      return this.delegate;
  }

  public static IAST2unaryOperation<CEASTparseException> parse
    (final Element e, final IParser parser)
  throws CEASTparseException {
    return fr.upmc.ilp.ilp2.ast.CEASTunaryOperation.parse(e, parser);
  }

  @ILPexpression
  public IAST4expression getOperand () {
    return CEAST.narrowToIAST4expression(getDelegate().getOperand());
  }
  public IAST4expression[] getOperands () {
      IAST2expression<CEASTparseException>[] operands =
          getDelegate().getOperands();
      return CEAST.narrowToIAST4expressionArray(operands);
  }

  @Override
  public IAST4expression normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common,
          final IAST4Factory factory )
    throws NormalizeException {
      return factory.newUnaryOperation(
              getOperatorName(),
              getOperand().normalize(lexenv, common, factory) );
  }

  @Override
  public void compile (final StringBuffer buffer,
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common,
                       final IDestination destination )
  throws CgenerationException {
      IAST4localVariable tmp = CEASTlocalVariable.generateVariable();
      buffer.append("{\n");
      tmp.compileDeclaration(buffer, lexenv, common);
      ICgenLexicalEnvironment bodyLexenv = lexenv;
      bodyLexenv = bodyLexenv.extend(tmp);
      getOperand().compile(buffer, bodyLexenv, common,
              new AssignDestination(tmp) );
      buffer.append(";\n");
      destination.compile(buffer, bodyLexenv, common);
      buffer.append(common.compileOperator1(getOperatorName()));
      buffer.append("(");
      buffer.append(tmp.getMangledName());
      buffer.append(");}\n");
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      findAndAdjoinToInvokedFunctions(getOperand());
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
    getOperand().inline();
  }
  */
```

```java
  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTvariable.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;
import fr.upmc.ilp.tool.CStuff;

public class CEASTvariable
implements IAST4variable {

    public CEASTvariable (final String name) {
        this.name = name;
        synchronized (this) {
            if (    name.startsWith("ilp_")
                 || name.startsWith("ILP_") ) {
                this.mangledName = name;
            } else {
                counter++;
                this.mangledName = CStuff.mangle(this.name)
                    + "_" + counter;
            }
        }
    }
    final String name;
    final String mangledName;
    private static int counter = 0;

    public CEASTvariable (final String name, final String mangledName) {
        this.name = name;
        this.mangledName = mangledName;
    }

    public String getName () {
        return this.name;
    }
    public String getMangledName () {
        return this.mangledName;
    }

    @Override
    public boolean equals (final Object that) {
        if ( that instanceof IAST4variable ) {
            IAST4variable iv = (IAST4variable) that;
            return this.name.equals(iv.getName());
        }
        return false;
    }
    @Override
    public int hashCode() {
        return this.name.hashCode();
    }

    public static IAST2variable parse (final Element e, final IParser parser)
    throws CEASTparseException {
        return parser.getFactory().newVariable(e.getAttribute("nom"));
    }

    /** Determiner la nature plus precise de la variable. */

    public IAST4variable normalize (
            final INormalizeLexicalEnvironment lexenv,
            final INormalizeGlobalEnvironment common,
            final IAST4Factory factory )
    throws NormalizeException {
        // Les variables locales ont precedence:
        final IAST4variable lv = lexenv.isPresent(this);
        if ( lv != null ) {
            return lv;
        }
        // puis les globales:
        final IAST4globalVariable gv = common.isPresent(this);
        if ( gv != null ) {
            return gv;
        }
        // Sinon c'est une globale:
        final IAST4globalVariable global = factory.newGlobalVariable(getName());
        common.add(global);
        return global;
    }
    // NOTE: les globalVariable peuvent aussi etre raffinees en
    // globalFunctionVariable.

    public Object eval (ILexicalEnvironment lexenv, ICommon common)
    throws EvaluationException {
        final String msg = "Should not evaluate vanilla variable!";
        throw new EvaluationException(msg);
    }

    public void compileDeclaration(
            StringBuffer buffer,
            ICgenLexicalEnvironment lexenv,
            ICgenEnvironment common )
    throws CgenerationException {
        final String msg = "No compileDeclaration on vanilla variable!";
        throw new CgenerationException(msg);
    }

    public <Data, Result, Exc extends Throwable> Result
      accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
        return visitor.visit(this, data);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/CEASTwhile.java

```java
package fr.upmc.ilp.ilp4.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.ILPexpression;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.cgen.VoidDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2while;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.cgen.AssignDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4Factory;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4localVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.IAST4while;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IParser;

/** La boucle tant-que: son interprétation et sa compilation.
 *
 * La principale question est: « Quelle est la valeur d'une telle
 * boucle ? »
 */
```

```java

public class CEASTwhile
  extends CEASTdelegableInstruction
  implements IAST4while {

  public CEASTwhile (final IAST4expression condition,
                     final IAST4expression body)
  {
      this.delegate =
          new fr.upmc.ilp.ilp2.ast.CEASTwhile(condition, body);
  }
  private IAST2while<CEASTparseException> delegate;

  @Override
  public IAST2while<CEASTparseException> getDelegate () {
      return this.delegate;
  }

  @ILPexpression
  public IAST4expression getCondition () {
    return CEAST.narrowToIAST4expression(getDelegate().getCondition());
  }
  @ILPexpression
  public IAST4expression getBody() {
    return CEAST.narrowToIAST4expression(getDelegate().getBody());
  }

  public static IAST2while<CEASTparseException> parse (
          final Element e, final IParser parser)
  throws CEASTparseException {
    return fr.upmc.ilp.ilp2.ast.CEASTwhile.parse(e, parser);
  }

  @Override
  public IAST4expression normalize (
          final INormalizeLexicalEnvironment lexenv,
          final INormalizeGlobalEnvironment common,
          final IAST4Factory factory )
    throws NormalizeException {
      return factory.newWhile(
          getCondition().normalize(lexenv, common, factory),
          getBody().normalize(lexenv, common, factory) );
  }

  @Override
  public void compile (final StringBuffer buffer,
                       final ICgenLexicalEnvironment lexenv,
                       final ICgenEnvironment common,
                       final IDestination destination)
  throws CgenerationException {
      IAST4localVariable tmp = CEASTlocalVariable.generateVariable();
      buffer.append(" while ( 1 ) {\n");
      tmp.compileDeclaration(buffer, lexenv, common);
      ICgenLexicalEnvironment bodyLexenv = lexenv;
      bodyLexenv = bodyLexenv.extend(tmp);
      getCondition().compile(buffer, bodyLexenv, common,
              new AssignDestination(tmp) );
      IDestination garbage = VoidDestination.create();
      buffer.append(" if ( ILP_isEquivalentToTrue(");
      buffer.append(tmp.getMangledName());
      buffer.append(") ) {\n");
      getBody().compile(buffer, bodyLexenv, common, garbage);
      buffer.append("}\n else { break; }\n}\n");
      CEASTinstruction.voidExpression()
          .compile(buffer, lexenv, common, destination);
      buffer.append(";\n");
  }

  /* Obsolete de par CEAST.findInvokedFunctions() qui use d'annotations
  @Override
  public void findInvokedFunctions () {
      findAndAdjoinToInvokedFunctions(getCondition());
      findAndAdjoinToInvokedFunctions(getBody());
  }

  /* Obsolète de par CEAST.inline() qui use de réflexivité.
  @Override
  public void inline () {
    getCondition().inline();
    getBody().inline();
  }
  */

  public <Data, Result, Exc extends Throwable> Result
    accept (IAST4visitor<Data, Result, Exc> visitor, Data data) throws Exc {
      return visitor.visit(this, data);
  }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/FindingInvokedFunctionsException.java

```java
package fr.upmc.ilp.ilp4.ast;

public class FindingInvokedFunctionsException
extends Exception {

    static final long serialVersionUID = +1234567890010000L;

    public FindingInvokedFunctionsException (String message) {
        super(message);
    }

    public FindingInvokedFunctionsException (Throwable cause) {
        super(cause);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/GlobalCollector.java

```java
package fr.upmc.ilp.ilp4.ast;

import java.util.HashSet;
import java.util.Set;

import fr.upmc.ilp.ilp4.interfaces.AbstractExplicitVisitor;
import fr.upmc.ilp.ilp4.interfaces.IAST4assignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4localAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4program;
import fr.upmc.ilp.ilp4.interfaces.IAST4reference;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;

/** Ce visiteur collecte les variables globales par visite plutot que
 * d'utiliser les methodes des delegues. Les variables globales sont
 * accumulees dans le visiteur meme. La visite est triviale puisque ce
 * visiteur ne visite que des AST normalisés où les variables globales
 * sont repérées par des instances de IAST4globalVariable.
 */

public class GlobalCollector
extends AbstractExplicitVisitor<Object, RuntimeException> {

    public GlobalCollector () {
        this.globals = new HashSet<>();
    }
    protected final Set<IAST4globalVariable> globals;

    // Le point d'entrée de ce collecteur:
    public static IAST4globalVariable[] getGlobalVariables (IAST4program iast) {
        final GlobalCollector visitor = new GlobalCollector();
        iast.accept(visitor, null);
        return visitor.globals.toArray(new IAST4globalVariable[0]);
    }

    // Les visiteurs specialises: Ici l'on visite les variables!

    @Override
    public Object visit (IAST4assignment iast, Object nothing) {
        // Visiter la variable affectee:
        iast.getVariable().accept(this, nothing);
        iast.getValue().accept(this, nothing);
        return null;
    }
```

```java
    @Override
    public Object visit (IAST4localAssignment iast, Object nothing) {
        visit((IAST4assignment) iast, nothing);
        return null;
    }
    @Override
    public Object visit (IAST4globalAssignment iast, Object nothing){
        visit((IAST4assignment) iast, nothing);
        return null;
    }

    @Override
    public Object visit (IAST4reference iast, Object nothing) {
        // Visiter la variable referencee:
        iast.getVariable().accept(this, nothing);
        return null;
    }

    @Override
    public Object visit (IAST4variable iast, Object nothing) {
        if ( iast instanceof IAST4globalVariable ) {
            globals.add((IAST4globalVariable) iast);
        }
        return null;
     }
    public Object visit (IAST4globalVariable iast, Object nothing) {
        globals.add((IAST4globalVariable) iast);
        return null;
     }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/InliningException.java

```java
package fr.upmc.ilp.ilp4.ast;

public class InliningException extends Exception {

    static final long serialVersionUID = +1234567890010000L;

    public InliningException(String message) {
        super(message);
    }

    public InliningException(Throwable cause) {
        super(cause);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/NormalizeException.java

```java
package fr.upmc.ilp.ilp4.ast;

public class NormalizeException extends Exception {
    static final long serialVersionUID = +1234567890010000L;

    public NormalizeException(String message) {
        super(message);
    }

    public NormalizeException(Throwable cause) {
        super(cause);
    }
}
```

Java/src/fr/upmc/ilp/ilp4/ast/NormalizeGlobalEnvironment.java

```java
package fr.upmc.ilp.ilp4.ast;

import java.util.HashMap;
import java.util.Map;

import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.INormalizeGlobalEnvironment;

/** Une implantation d'environnement global pour la normalisation des
 * expressions. */

public class NormalizeGlobalEnvironment
  implements INormalizeGlobalEnvironment {

  public NormalizeGlobalEnvironment () {
    this.globals = new HashMap<>();
    this.primitives = new HashMap<>();
  }
  private final Map<String,IAST4globalVariable> globals;
  private final Map<String,IAST4globalVariable> primitives;

  public void add (final IAST4globalVariable variable) {
    globals.put(variable.getName(), variable);
  }

  public IAST4globalVariable isPresent (final IAST4variable otherVariable) {
    Object gv = globals.get(otherVariable.getName());
    if ( gv != null ) {
      return (IAST4globalVariable) gv;
    } else {
      return null;
    }
  }

  public void addPrimitive (IAST4globalVariable variable) {
    primitives.put(variable.getName(), variable);
  }

  public IAST4globalVariable isPrimitive (final IAST4variable variable) {
    Object pv = primitives.get(variable.getName());
    if ( pv != null ) {
      return (CEASTglobalVariable) pv;
    } else {
      return null;
    }
  }

}
```

Java/src/fr/upmc/ilp/ilp4/ast/NormalizeLexicalEnvironment.java

```java
package fr.upmc.ilp.ilp4.ast;

import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.INormalizeLexicalEnvironment;

public class NormalizeLexicalEnvironment implements INormalizeLexicalEnvironment {
    private final IAST4variable variable;
    private final INormalizeLexicalEnvironment next;

    public NormalizeLexicalEnvironment(final IAST4variable variable,
        final INormalizeLexicalEnvironment next) {
        this.variable = variable;
        this.next = next;
    }

    public INormalizeLexicalEnvironment extend(final IAST4variable variable) {
        return new NormalizeLexicalEnvironment(variable, this);
    }

    public IAST4variable isPresent(final IAST4variable otherVariable) {
        if (variable.getName().equals(otherVariable.getName())) {
            return variable;
        } else {
            return next.isPresent(otherVariable);
        }
    }

    public static class EmptyNormalizeLexicalEnvironment
    implements INormalizeLexicalEnvironment {

        public EmptyNormalizeLexicalEnvironment() {}
```

```java
        public INormalizeLexicalEnvironment extend(final IAST4variable variable) {
            return new NormalizeLexicalEnvironment(variable, this);
        }

        public IAST4variable isPresent(final IAST4variable otherVariable) {
            return null;
        }
    }

}
```

Java/src/fr/upmc/ilp/ilp4/ast/XMLwriter.java

```java
package fr.upmc.ilp.ilp4.ast;

import java.io.StringWriter;
import java.io.Writer;
import java.util.HashMap;
import java.util.Map;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerConfigurationException;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp4.interfaces.IAST4alternative;
import fr.upmc.ilp.ilp4.interfaces.IAST4assignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4binaryOperation;
import fr.upmc.ilp.ilp4.interfaces.IAST4computedInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4constant;
import fr.upmc.ilp.ilp4.interfaces.IAST4expression;
import fr.upmc.ilp.ilp4.interfaces.IAST4functionDefinition;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalFunctionVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4globalVariable;
import fr.upmc.ilp.ilp4.interfaces.IAST4invocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4localAssignment;
import fr.upmc.ilp.ilp4.interfaces.IAST4localBlock;
import fr.upmc.ilp.ilp4.interfaces.IAST4primitiveInvocation;
import fr.upmc.ilp.ilp4.interfaces.IAST4program;
import fr.upmc.ilp.ilp4.interfaces.IAST4reference;
import fr.upmc.ilp.ilp4.interfaces.IAST4sequence;
import fr.upmc.ilp.ilp4.interfaces.IAST4try;
import fr.upmc.ilp.ilp4.interfaces.IAST4unaryBlock;
import fr.upmc.ilp.ilp4.interfaces.IAST4unaryOperation;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitable;
import fr.upmc.ilp.ilp4.interfaces.IAST4visitor;
import fr.upmc.ilp.ilp4.interfaces.IAST4while;

/** Un arpenteur d'AST le transformant en XML. Une fois instancié, il est
 * possible de l'utiliser plusieurs fois.
 */

public class XMLwriter
implements IAST4visitor<Object, Element, RuntimeException> {

    public XMLwriter ()
    throws ParserConfigurationException {
        final DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        this.documentBuilder = dbf.newDocumentBuilder();
    }
    protected final DocumentBuilder documentBuilder;
    protected Document document;
    protected Map<Object,Element> memory;

    protected synchronized int getCounter () {
        return counter++;
    }
    private int counter = 1000;

    /** Obtenir l'XML correspondant a l'AST. Cette methode ne peut etre
     * utilisee qu'apres process(). */

    public String getXML ()
    throws TransformerConfigurationException, TransformerException {
        if ( null == this.result ) {
            TransformerFactory tf = TransformerFactory.newInstance();
            tf.setAttribute("indent-number", 2);
            Transformer transformer = tf.newTransformer();
            transformer.setOutputProperty(OutputKeys.ENCODING, "ISO-8859-1");
            transformer.setOutputProperty(OutputKeys.STANDALONE, "yes");
            transformer.setOutputProperty(OutputKeys.METHOD, "xml");
            transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "yes");
            // FUTURE devrait figurer en propriete de compilation:
            transformer.setOutputProperty(OutputKeys.INDENT, "yes");
            // NOTE: xmllint -format f.xml > g.xml est aussi possible.
            // Voir aussi XmlStarlet en xmlstar.sourceforge.net
            DOMSource source = new DOMSource(this.document);
            Writer writer = new StringWriter();
            StreamResult sr = new StreamResult(writer);
            transformer.transform(source, sr);
            this.result = writer.toString();
        }
        return this.result;
    }
    protected @OrNull String result;

  /** La methode initiale pour lancer le traitement sur un programme
   * ILP entier. La methode getXML() permet de recuperer le resultat
   * de la traduction vers DOM. */

    public synchronized String process (IAST4visitable iast) {
        this.result = null;
        this.document = this.documentBuilder.newDocument();
        this.memory = new HashMap<>();
        Element lastVisitedElement = iast.accept(this, null);
        this.document.appendChild(lastVisitedElement);
        try {
            return getXML();
        } catch (TransformerConfigurationException e) {
            return null;
        } catch (TransformerException e) {
            return null;
        }
    }

    // Tous les visiteurs specialises. Ils convertissent un noeud IAST4
    // en un element DOM qu'ils retournent en valeur. La variable data
    // ne sert a rien ici.

    // Tous les visiteurs ont la meme structure:
    // Pour un iast donné, créer le noeud XML demandé,
    // chercher dans la memoire si l'iast avait déjà un noeud XML
    // associé et si oui, indiquer le partage avec un idref. Si non,
    // arpenter les composants de l'iast et les integrer au noeud XML
    // courant. Finalement, retourner le noeud XML créé.

    public Element visit (IAST4program iast, Object data) {
        final Element program = this.document.createElement(
                iast.getClass().getName() );
        final Element definitions =
            this.document.createElement("functionDefinitions");
        program.appendChild(definitions);
        for ( IAST4functionDefinition fun : iast.getFunctionDefinitions() ) {
            Element lastVisitedElement = fun.accept(this, null);
            definitions.appendChild(lastVisitedElement);
        }

        final Element globals = this.document.createElement("globalVariables");
        program.appendChild(globals);
        for ( IAST4globalVariable gv : iast.getGlobalVariables() ) {
            Element lastVisitedElement = gv.accept(this, null);
            globals.appendChild(lastVisitedElement);
        }

        final Element body = this.document.createElement("programBody");
```

```
        program.appendChild(body);
        Element lastVisitedElement = iast.getBody().accept(this, null);
        body.appendChild(lastVisitedElement);
        return program;
    }

    public Element visit (IAST4alternative iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            result.setAttribute("ternary", (iast.isTernary())?"true":"false");
            Element lastVisitedElement = iast.getCondition().accept(this, null);
            result.appendChild(lastVisitedElement);
            lastVisitedElement = iast.getConsequent().accept(this, null);
            result.appendChild(lastVisitedElement);
            lastVisitedElement = iast.getAlternant().accept(this, null);
            result.appendChild(lastVisitedElement);
        }
        return result;
    }

    public Element visit (IAST4assignment iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            Element lastVisitedElement = iast.getVariable().accept(this, null);
            result.appendChild(lastVisitedElement);
            lastVisitedElement = iast.getValue().accept(this, null);
            result.appendChild(lastVisitedElement);
        }
        return result;
    }
    public Element visit (IAST4localAssignment iast, Object data) {
        return visit((IAST4assignment) iast, data);
    }
    public Element visit (IAST4globalAssignment iast, Object data) {
        return visit((IAST4assignment) iast, data);
    }

    public Element visit (IAST4constant iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            result.setAttribute("value", iast.getValue().toString());
        }
        return result;
    }

    public Element visit (IAST4invocation iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            final Element fun = this.document.createElement("function");
            result.appendChild(fun);
            Element lastVisitedElement = iast.getFunction().accept(this, null);
            fun.appendChild(lastVisitedElement);

            final Element args = this.document.createElement("arguments");
            result.appendChild(args);
            for ( IAST4expression arg : iast.getArguments() ) {
                lastVisitedElement = arg.accept(this, null);
                args.appendChild(lastVisitedElement);
            }
        }
        return result;
    }

    public Element visit(IAST4computedInvocation iast, Object data) {
        return visit((IAST4invocation) iast, data);
    }

    public Element visit (IAST4globalInvocation iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            IAST4expression inlined = iast.getInlined();
            result.setAttribute("id", "" + getCounter());
            result.setAttribute("inlined",
                    (inlined != null) ? "true" : "false" );
            this.memory.put(iast, result);
            // Serialisation:
            if ( inlined != null ) {
                final Element inl = this.document.createElement("inlined");
                result.appendChild(inl);
                Element lastVisitedElement = inlined.accept(this, null);
                inl.appendChild(lastVisitedElement);
            }

            final Element fun = this.document.createElement("globalFunction");
            result.appendChild(fun);
            Element lastVisitedElement = iast.getFunctionGlobalVariable().accept(this, null);
            fun.appendChild(lastVisitedElement);

            final Element args = this.document.createElement("arguments");
            result.appendChild(args);
            for ( IAST4expression arg : iast.getArguments() ) {
                lastVisitedElement = arg.accept(this, null);
                args.appendChild(lastVisitedElement);
            }
        }
        return result;
    }

    public Element visit (IAST4primitiveInvocation iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            final Element fun = this.document.createElement("primitivefunction");
            result.appendChild(fun);
            Element lastVisitedElement = iast.getFunctionGlobalVariable().accept(this, null);
            fun.appendChild(lastVisitedElement);

            final Element args = this.document.createElement("arguments");
            result.appendChild(args);
            for ( IAST4expression arg : iast.getArguments() ) {
                lastVisitedElement = arg.accept(this, null);
                args.appendChild(lastVisitedElement);
            }
        }
        return result;
    }

    public Element visit (IAST4functionDefinition iast, Object data) {
        final Element result = this.document.createElement(
```

```
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            result.setAttribute("name", iast.getFunctionName());
            result.setAttribute("recursive",
                    Boolean.toString(iast.isRecursive()) );

            Element lastVisitedElement = iast.getDefinedVariable().accept(this, null);
            result.appendChild(lastVisitedElement);

            final Element funs =
                this.document.createElement("invokedFunctions");
            result.appendChild(funs);
            for ( IAST4globalFunctionVariable gfv : iast.getInvokedFunctions() ) {
                lastVisitedElement = gfv.accept(this, null);
                funs.appendChild(lastVisitedElement);
            }

            final Element vars = this.document.createElement("variables");
            result.appendChild(vars);
            for ( IAST4variable lv : iast.getVariables() ) {
                lastVisitedElement = lv.accept(this, null);
                vars.appendChild(lastVisitedElement);
            }

            final Element body = this.document.createElement("body");
            result.appendChild(body);
            lastVisitedElement = iast.getBody().accept(this, null);
            body.appendChild(lastVisitedElement);
        }
        return result;
    }

    public Element visit (IAST4reference iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            final Element var = this.document.createElement("variable");
            result.appendChild(var);
            Element lastVisitedElement = iast.getVariable().accept(this, null);
            var.appendChild(lastVisitedElement);
        }
        return result;
    }

    public Element visit (IAST4variable iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            result.setAttribute("name", iast.getName());
            result.setAttribute("mangledName", iast.getMangledName());
        }
        return result;
    }

    public Element visit (IAST4localBlock iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            final Element vars = this.document.createElement("variables");
            result.appendChild(vars);
            for ( IAST4variable lv : iast.getVariables() ) {
                Element lastVisitedElement = lv.accept(this, null);
                vars.appendChild(lastVisitedElement);
            }

            final Element inits = this.document.createElement("initialisations");
            result.appendChild(inits);
            for ( IAST4expression lvinit : iast.getInitializations() ) {
                Element lastVisitedElement = lvinit.accept(this, null);
                inits.appendChild(lastVisitedElement);
            }

            final Element body = this.document.createElement("body");
            result.appendChild(body);
            Element lastVisitedElement = iast.getBody().accept(this, null);
            body.appendChild(lastVisitedElement);
        }
        return result;
    }

    public Element visit (IAST4binaryOperation iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            result.setAttribute("operation", iast.getOperatorName());
            Element lastVisitedElement = iast.getLeftOperand().accept(this, null);
            result.appendChild(lastVisitedElement);
            lastVisitedElement = iast.getRightOperand().accept(this, null);
            result.appendChild(lastVisitedElement);
        }
        return result;
    }

    public Element visit (IAST4unaryOperation iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            result.setAttribute("operation", iast.getOperatorName());
            Element lastVisitedElement = iast.getOperand().accept(this, null);
            result.appendChild(lastVisitedElement);
        }
        return result;
    }

    public Element visit (IAST4sequence iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            for ( IAST4expression instr : iast.getInstructions() ) {
                Element lastVisitedElement = instr.accept(this, null);
                result.appendChild(lastVisitedElement);
            }
        }
        return result;
```

```
    }

    public Element visit (IAST4try iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            final Element body = this.document.createElement("body");
            result.appendChild(body);
            Element lastVisitedElement = iast.getBody().accept(this, null);
            body.appendChild(lastVisitedElement);

            final Element catcher = this.document.createElement("catcher");
            if ( null != iast.getCaughtExceptionVariable() ) {
                result.appendChild(catcher);
                final Element var =
                    this.document.createElement("caughtVariable");
                catcher.appendChild(var);
                lastVisitedElement = iast.getCaughtExceptionVariable().accept(this, null);
                var.appendChild(lastVisitedElement);

                final Element catcherBody =
                    this.document.createElement("catcherBody");
                catcher.appendChild(catcherBody);
                lastVisitedElement = iast.getCatcher().accept(this, null);
                catcher.appendChild(lastVisitedElement);
            }

            final Element finallyer = this.document.createElement("finallyer");
            if ( null != iast.getFinallyer() ) {
                result.appendChild(finallyer);
                lastVisitedElement = iast.getFinallyer().accept(this, null);
                finallyer.appendChild(lastVisitedElement);
            }
        }
        return result;
    }

    public Element visit (IAST4unaryBlock iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            final Element var = this.document.createElement("variable");
            result.appendChild(var);
            Element lastVisitedElement = iast.getVariable().accept(this, null);
            var.appendChild(lastVisitedElement);

            final Element init = this.document.createElement("initialisation");
            result.appendChild(init);
            lastVisitedElement = iast.getInitialization().accept(this, null);
            init.appendChild(lastVisitedElement);

            final Element body = this.document.createElement("body");
            result.appendChild(body);
            lastVisitedElement = iast.getBody().accept(this, null);
            body.appendChild(lastVisitedElement);
        }
        return result;
    }

    public Element visit (IAST4while iast, Object data) {
        final Element result = this.document.createElement(
                iast.getClass().getName() );
        if ( this.memory.containsKey(iast) ) {
            final Element old = this.memory.get(iast);
            result.setAttribute("idref", old.getAttribute("id"));
        } else {
            result.setAttribute("id", "" + getCounter());
            this.memory.put(iast, result);
            // Serialisation:
            final Element cond = this.document.createElement("condition");
            result.appendChild(cond);
            Element lastVisitedElement = iast.getCondition().accept(this, null);
            cond.appendChild(lastVisitedElement);

            final Element body = this.document.createElement("body");
            result.appendChild(body);
            lastVisitedElement = iast.getBody().accept(this, null);
            body.appendChild(lastVisitedElement);
        }
        return result;
    }
}
```

Java/src/fr/upmc/ilp/ilp4/runtime/CommonPlus.java

```
package fr.upmc.ilp.ilp4.runtime;

import fr.upmc.ilp.ilp2.interfaces.ICommon;

/** Environnement global d'interpretation d'ILP4. */

public class CommonPlus
extends fr.upmc.ilp.ilp2.runtime.CommonPlus
implements ICommon {

     public CommonPlus () {
         super();
     }

     @Override
     public boolean isPresent (final String variableName) {
          return (   globalMap.containsKey(variableName)
                  || primitiveMap.containsKey(variableName) );
    }

}
```

Java/src/fr/upmc/ilp/ilp4/runtime/LexicalEnvironment.java

```
package fr.upmc.ilp.ilp4.runtime;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp2.runtime.BasicEmptyEnvironment;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;

/** Cette implantation d'environnement est très naïve par contre elle
 * utilise des comparaisons d'adresses rapides plutot que des comparaisons
 * de chaine de caracteres comme dans ILP2.
 */

public class LexicalEnvironment
extends fr.upmc.ilp.ilp2.runtime.LexicalEnvironment
implements ILexicalEnvironment {

    public LexicalEnvironment (final IAST2variable variable,
                               final Object value,
                               final ILexicalEnvironment next) {
        super(variable, value, next);
    }

    /** Renvoie la valeur d'une variable si présente dans
     * l'environnement. */

    @Override
    public Object lookup (final IAST2variable otherVariable)
        throws EvaluationException {
        if ( this.variable == otherVariable ) {
            return this.value;
```

```
        } else {
            return getNext().lookup(otherVariable);
        }
    }

    /** verifie si une variable est presente dans l'environnement. */
    @Override
    public boolean isPresent (final IAST2variable otherVariable) {
        if (variable == otherVariable) {
            return true;
        } else {
            return getNext().isPresent(otherVariable);
        }
    }

    @Override
    public void update (final IAST2variable otherVariable,
                        final Object value)
        throws EvaluationException {
        if (variable == otherVariable) {
            this.value = value;
        } else {
            getNext().update(otherVariable, value);
        }
    }

    /** On peut étendre tout environnement. */
    public ILexicalEnvironment extend (final IAST4variable variable,
                                       final Object value) {
        return new LexicalEnvironment(variable, value, this);
    }

    /** Comme il y a une dependance entre  EmptyLexicalEnvironment
     * et LexicalEnvironment, on lie leurs definitions en un unique fichier.
     * */

    public static class EmptyLexicalEnvironment
    extends BasicEmptyEnvironment<IAST2variable>
    implements ILexicalEnvironment {

      // La technique du singleton:
      private EmptyLexicalEnvironment () {}
      private static final EmptyLexicalEnvironment THE_EMPTY_LEXICAL_ENVIRONMENT;
      static {
        THE_EMPTY_LEXICAL_ENVIRONMENT = new EmptyLexicalEnvironment();
      }

      public static EmptyLexicalEnvironment create () {
        return EmptyLexicalEnvironment.THE_EMPTY_LEXICAL_ENVIRONMENT;
      }

      public Object lookup (IAST2variable variable) {
          String msg = "Variable sans valeur: " + getVariable().getName();
          throw new RuntimeException(msg);
      }
      @Override
      public EmptyLexicalEnvironment getNext() {
          final String msg = "Empty environment!";
          throw new RuntimeException(msg);
      }
      @Override
      public @OrNull ILexicalEnvironment shrink(IAST2variable v) {
          return null;
      }
      /** L'environnement vide ne contient rien et signale
       * systematiquement une erreur si l'on cherche la valeur d'une
       * variable. */

      public void update (final IAST2variable variable,
                          final Object value )
        throws EvaluationException {
        final String msg = "Variable inexistante: " + variable.getName();
        throw new EvaluationException(msg);
      }

      /** On peut etendre l'environnement vide.
       *
       * Malheureusement, cela cree une dependance avec la classe des
       * environnements non vides d'ou l'inclusion de cette classe dans
      * celle des environnements non vides.
      */
     @Override
     public ILexicalEnvironment extend (final IAST2variable variable,
                                        final Object value) {
       return new LexicalEnvironment(variable, value, this);
     }
   }
}
```

Java/src/fr/upmc/ilp/ilp4/runtime/PrintStuff.java

```
package fr.upmc.ilp.ilp4.runtime;

import fr.upmc.ilp.ilp1.runtime.AbstractInvokableImpl;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;

/** Les primitives pour imprimer à savoir print et newline. En fait,
 * newline pourrait se programmer à partir de print et de la chaîne
 * contenant une fin de ligne mais comme nous n'avons pas encore de
 * fonctions, elle est utile.
 */
public class PrintStuff {

    public PrintStuff() {
        this.output = new StringBuffer();
    }
    private final StringBuffer output;

    /** On peut aussi imposer le flux de sortie.
     *
     * MOCHE: devrait plutot etre un flux qu'un tampon. */
    public PrintStuff (StringBuffer output) {
        this.output = output;
    }

    /** Renvoyer les caractères imprimés et remettre à vide le tampon
     * d'impression. */
    public synchronized String getPrintedOutput() {
        final String result = output.toString();
        output.delete(0, output.length());
        return result;
    }

    public void extendWithPrintPrimitives(final ICommon common)
        throws EvaluationException {
        common.bindPrimitive("print", new PrintPrimitive());
        common.bindPrimitive("newline", new NewlinePrimitive());
    }

    /** Cette classe implante la fonction print() qui permet d'imprimer
     * une valeur. */
    private class PrintPrimitive extends AbstractInvokableImpl {
        private PrintPrimitive() {}

        // La fonction print() est unaire:
        @Override
        public Object invoke (Object value) {
            output.append(value.toString());
            return Boolean.FALSE;
        }
    }

    /** Cette classe implante la fonction newline() qui permet de passer
     * à la ligne. */
    private class NewlinePrimitive extends AbstractInvokableImpl {
        private NewlinePrimitive() {}

        // La fonction newline() est zéro-aire:
        @Override
        public Object invoke () {
            output.append("\n");
            return Boolean.FALSE;
        }
    }
}
```

Java/src/fr/upmc/ilp/ilp4/cgen/AssignDestination.java

```
package fr.upmc.ilp.ilp4.cgen;

import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp4.interfaces.IAST4variable;

/** Destination indiquant dans quelle variable affecter un résultat.
 * Ici, on utilise le nom adapté au langage visé. */

public class AssignDestination implements IDestination {

    public AssignDestination (final IAST4variable variable) {
        this.variable = variable;
    }
    private final IAST4variable variable;

    /** Préfixe le résultat avec "variable = " pour indiquer une
     * affectation. */
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common ) {
        buffer.append(variable.getMangledName());
        buffer.append(" = ");
    }

}
```