Code ILP2 et ILP3

Christian.Queinnec@lip6.fr

27 août 2013

Table des matières

Grammars/grammar2.rnc

```
# Deuxième version du langage étudié: ILP2 pour « Insipide Langage
# Prévu » Il sera complété dans les cours qui suivent.

# On étend la grammaire précédente pour accepter des programmes
# enrichis (définis par la balise programme2) et l'on modifie la
# notion d'instruction pour accepter la présence des blocs locaux.

include "grammar1.rnc"
start |= programme2
instruction |= blocLocal
instruction |= boucle
expression |= affectation
expression |= invocation

# Un programme est une suite de définitions de fonctions suivie
# d'instructions sauf que les instructions comportent également des
# blocs locaux (étendant la notion de bloc local unaire précédente),
# et la boucle tant-que.

programme2 = element programme2 {
   definitionFonction *,
   instructions
}

# Définition d'une fonction avec son nom, ses variables (éventuellement
# aucune) et un corps qui est une séquence d'instructions.
# FUTUR: restreindre les noms de fonctions a ceux des variables!

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

# 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 { instructions }
}

# 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     { instructions }
}

# L'affectation prend une variable en cible et une expression comme
# valeur. L'affectation est une expression.
# FUTUR: restreindre les noms de variables!

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 * }
}
```

Grammars/grammar3.rnc


```
# Troisième version du langage étudié: ILP3 pour « Incomplet Langage
# Poétique » Il sera complété dans les cours qui suivent.

# On ajoute une instruction try-catch-finally. Les exceptions sont
# signalées par une primitive nommée throw ce qui permet de ne rien
# ajouter de plus à cette grammaire.

include "grammar2.rnc"
start |= programme3
instruction |= try

# Un programme3 est semblable à un programme2:

programme3 = element programme3 {
   definitionFonction *,
   instructions
}

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

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

# FUTUR: restreindre les noms comme pour les variables!

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

finally = element finally {
   instructions
}
```

C/ilpError.c

```
/** Ce fichier constitue la bibliothèque d'exécution d'ILP. Il
 * implante ilpBasicError.h avec des fonctions qui signalent
 * maintenant des exceptions rattrapables par ILP.
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "ilp.h"
#include "ilpBasicError.h"
#include "ilpException.h"

char *ilpError_Id = "$Id: ilpError.c 929 2010-08-20 18:00:59Z queinnec $";

/** Allocation statique d'une instance d'exception. Sa définition fait
 * qu'elle peut être prise pour un ILP_Object. Par contre sa grosse
 * taille n'influence pas la taille des ILP_Object. */

enum ILP_ExceptionKind {
     ILP_EXCEPTION_KIND     = 0xab060ba
};

#define ILP_EXCEPTION_BUFFER_LENGTH    1000
#define ILP_EXCEPTION_CULPRIT_LENGTH     10

struct ILP_exception {
     enum ILP_ExceptionKind    _kind;
     union {
          struct asException {
               char message[ILP_EXCEPTION_BUFFER_LENGTH];
               ILP_Object culprit[ILP_EXCEPTION_CULPRIT_LENGTH];
          } asException;
     }                         _content;
};
static struct ILP_exception ILP_the_exception =  {
     ILP_EXCEPTION_KIND, { { "", { NULL } } }
```

```
};

/**
 * Signalement d'une exception
 */

ILP_Object
ILP_error (char *message)
{
     snprintf(ILP_the_exception._content.asException.message,
              ILP_EXCEPTION_BUFFER_LENGTH,
              "Error: %s\n",
              message);
     fprintf(stderr, "%s", ILP_the_exception._content.asException.message);
     ILP_the_exception._content.asException.culprit[0] = NULL;
     return ILP_throw((ILP_Object) &ILP_the_exception);
}

/** Une fonction pour signaler qu'un argument n'est pas du type attendu. */

ILP_Object
ILP_domain_error (char *message, ILP_Object o)
{
     snprintf(ILP_the_exception._content.asException.message,
              ILP_EXCEPTION_BUFFER_LENGTH,
              "Domain error: %s\nCulprit: 0x%p\n",
              message, (void*) o);
     fprintf(stderr, "%s", ILP_the_exception._content.asException.message);
     ILP_the_exception._content.asException.culprit[0] = o;
     ILP_the_exception._content.asException.culprit[1] = NULL;
     return ILP_throw((ILP_Object) &ILP_the_exception);
}

/* end of ilpError.c */
```

C/ilpException.c

```
/** Ce fichier constitue un complément à la bibliothèque d'exécution
 * d'ILP notamment les primitives manipulant les exceptions. */

#include <stdlib.h>
#include <stdio.h>
#include "ilpException.h"

/** Ces variables globales contiennent:
 *  -- le rattrapeur d'erreur courant
 *  -- l'exception courante (lorsque signalée)
 */

static struct ILP_catcher ILP_the_original_catcher = {
     NULL
};
struct ILP_catcher *ILP_current_catcher = &ILP_the_original_catcher;

ILP_Object ILP_current_exception = NULL;

/** Signaler une exception. */

ILP_Object
ILP_throw (ILP_Object exception)
{
     ILP_current_exception = exception;
     if ( ILP_current_catcher == &ILP_the_original_catcher ) {
          ILP_die("No current catcher!");
     };
     longjmp(ILP_current_catcher->_jmp_buf, 1);
     /** UNREACHABLE */
     return NULL;
}

/** Chaîner le nouveau rattrapeur courant avec l'ancien. */

void
ILP_establish_catcher (struct ILP_catcher *new_catcher)
{
     new_catcher->previous = ILP_current_catcher;
     ILP_current_catcher = new_catcher;
}
```



```
/** Remettre en place un rattrapeur. */

void
ILP_reset_catcher (struct ILP_catcher *catcher)
{
     ILP_current_catcher = catcher;
}

/* end of ilpException.c */
```

C/ilpException.h

```
#ifndef ILP_EXCEPTION_H
#define ILP_EXCEPTION_H

/** Les entêtes des ressources manipulant les exceptions.
 * Voici comment utiliser cette bibliothèque:
 *
 * À tout instant, ILP_current_catcher contient le rattrapeur
 * d'exceptions. Lorsqu'une exception est signalée par ILP_throw, elle
 * est envoyée au rattrapeur courant. Pour établir un nouveau
 * rattrapeur d'erreur, utilisez le motif suivant:
 *
 *        { struct ILP_catcher *current_catcher = ILP_current_catcher;
 *          struct ILP_catcher new_catcher;
 *          if ( 0 == setjmp(new_catcher->_jmpbuf) ) {
 *              ILP_establish_catcher(&new_catcher);
 *              ...
 *              ILP_current_exception = NULL;
 *          };
 *          ILP_reset_catcher(current_catcher);
 *          if ( NULL != ILP_current_exception ) {
 *              if ( 0 == setjmp(new_catcher->_jmpbuf) ) {
 *                  ILP_establish_catcher(&new_catcher);
 *                  ... = ILP_current_exception;
 *                  ILP_current_exception = NULL;
 *                  ...
 *              };
 *          };
 *          ILP_reset_catcher(current_catcher);
 *          ...
 *          if ( NULL != ILP_current_exception ) {
 *             ILP_throw(ILP_current_exception);
 *          };
 *        }
 */

#include <setjmp.h>
#include "ilp.h"

struct ILP_catcher {
     struct ILP_catcher *previous;
     jmp_buf _jmp_buf;
};

extern struct ILP_catcher *ILP_current_catcher;
extern ILP_Object ILP_current_exception;
extern ILP_Object ILP_throw (ILP_Object exception);
extern void ILP_establish_catcher (struct ILP_catcher *new_catcher);
extern void ILP_reset_catcher (struct ILP_catcher *catcher);

#endif /* ILP_EXCEPTION_H */

/* end of ilpException.h */
```

Java/src/fr/upmc/ilp/ilp2/package.html

```
<body><p>

Ce paquetage contient l'interprète, le compilateur et la bibliothèque
d'exécution de l'interprète d'ILP2. Ce paquetage ne dépend plus
d'aucun autre paquetage.

</p></body>
```

Java/src/fr/upmc/ilp/ilp2/Process.java

```java
package fr.upmc.ilp.ilp2;

import java.io.IOException;

import org.w3c.dom.Document;

import fr.upmc.ilp.ilp1.AbstractProcess;
import fr.upmc.ilp.ilp2.ast.CEASTFactory;
import fr.upmc.ilp.ilp2.ast.CEASTParser;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.cgen.CgenEnvironment;
import fr.upmc.ilp.ilp2.cgen.CgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IAST2Factory;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;
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.ilp2.interfaces.IParser;
import fr.upmc.ilp.ilp2.runtime.CommonPlus;
import fr.upmc.ilp.ilp2.runtime.ConstantsStuff;
import fr.upmc.ilp.ilp2.runtime.LexicalEnvironment;
import fr.upmc.ilp.ilp2.runtime.PrintStuff;
import fr.upmc.ilp.tool.FileTool;
import fr.upmc.ilp.tool.IFinder;
import fr.upmc.ilp.tool.ProgramCaller;

/** Cette classe précise comment est traité un programme d'ILP2.
 */

public class Process extends AbstractProcess {

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

    public Process (IFinder finder) throws IOException {
        super(finder);
        setGrammar(getFinder().findFile("grammar2.rng"));
        IAST2Factory<CEASTparseException> factory = new CEASTFactory();
        setFactory(factory);
        setParser(new CEASTParser(factory));
    }

    public IParser<CEASTparseException> getParser () {
        return this.parser;
    }
    private IParser<CEASTparseException> parser;
    public void setParser(IParser<CEASTparseException> parser) {
        this.parser = parser;
    }

    public IAST2Factory<CEASTparseException> getFactory () {
        return this.factory;
    }
    private IAST2Factory<CEASTparseException> factory;
    public void setFactory(IAST2Factory<CEASTparseException> factory) {
        this.factory = factory;
    }

    public IAST2program<CEASTparseException> getCEAST () {
        return this.ceast;
    }
    protected IAST2program<CEASTparseException> ceast;
    public void setCEAST (IAST2program<CEASTparseException> ceast) {
        this.ceast = ceast;
    }

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

    /** Préparation. On analyse syntaxiquement le texte du programme,
     * on effectue quelques analyses et on l'amène à un état où il
     * pourra être interprété ou compilé. Toutes les analyses communes
     * à ces deux fins sont partagées ici.
     */

    public void prepare() {
        try {
            assert this.rngFile != null;
            final Document d = getDocument(this.rngFile);
            setCEAST(getParser().parse(d));
```



```java
            this.prepared = true;

        } catch (Throwable e) {
            this.preparationFailure = e;
            if ( this.verbose ) {
                System.err.println(e);
            }
        }
    }

    /** Interpretation */

    public void interpret() {
        try {
            assert this.prepared;
            final ICommon intcommon = new CommonPlus();
            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;
            if ( this.verbose ) {
                System.err.println(e);
            }
        }
    }

    /** Compilation vers C. */

    public void compile() {
        try {
            assert this.prepared;
            final ICgenEnvironment common = new CgenEnvironment();
            final ICgenLexicalEnvironment lexenv =
                CgenLexicalEnvironment.CgenEmptyLexicalEnvironment.create();
            this.ccode = getCEAST().compile(lexenv, common);

            this.compiled = true;

        } catch (Throwable e) {
            this.compilationFailure = e;
            if ( this.verbose ) {
                System.err.println(e);
            }
        }
    }

    /** Exécution du programme compilé: */

    public void runCompiled() {
        try {
            assert this.compiled;
            assert this.cFile != null;
            assert this.compileThenRunScript != null;
            FileTool.stuffFile(this.cFile, ccode);

            // Optionnel: mettre en forme le programme:
            String indentProgram = "indent " + this.cFile.getAbsolutePath();
            ProgramCaller pcindent = new ProgramCaller(indentProgram);
            pcindent.run();

            // et le compiler:
            String program = "bash "
                + this.compileThenRunScript.getAbsolutePath() + " "
                + this.cFile.getAbsolutePath();
            ProgramCaller pc = new ProgramCaller(program);
            pc.setVerbose();
            pc.run();
```

```
            this.executionPrinting = pc.getStdout().trim();

            this.executed = ( pc.getExitValue() == 0 );

        } catch (Throwable e) {
            this.executionFailure = e;
            if ( this.verbose ) {
                System.err.println(e);
            }
        }
    }
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2.java

```
package fr.upmc.ilp.ilp2.interfaces;

import java.util.Set;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;

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

public interface IAST2<Exc extends Exception> {

  /** Interprétation dans un certain environnement lexical et global. */

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

  /** Calculer l'ensemble des variables libres. */

  void findGlobalVariables (Set<IAST2variable> globalvars,
                            ICgenLexicalEnvironment lexenv );
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2Factory.java

```
package fr.upmc.ilp.ilp2.interfaces;

import java.util.List;

public interface IAST2Factory<Exc extends Exception> {

    /* Créer un programme */
    IAST2program<Exc> newProgram(
            IAST2functionDefinition<Exc>[] functions,
            IAST2instruction<Exc> instruction);

    /** Créer une séquence d'AST. */
    IAST2sequence<Exc> newSequence(List<IAST2instruction<Exc>> asts);

    /** Créer une alternative binaire. */
    IAST2alternative<Exc> newAlternative(
            IAST2expression<Exc> condition,
            IAST2instruction<Exc> consequent);

    /** Créer une alternative ternaire. */
    IAST2alternative<Exc> newAlternative(
            IAST2expression<Exc> condition,
            IAST2instruction<Exc> consequent,
            IAST2instruction<Exc> alternant);

    /** Créer une variable. */
    IAST2variable newVariable(String name);

    /** Créer une reference à une variable. */
    IAST2reference<Exc> newReference(IAST2variable variable);

    /** Créer une invocation (un appel à une fonction). */
    IAST2invocation<Exc> newInvocation(
            IAST2expression<Exc> function,
            IAST2expression<Exc>[] arguments);

    /** Créer une invocation (un appel à une fonction predefinie). */
    IAST2primitiveInvocation<Exc> newPrimitiveInvocation(
```



```
            String primitiveName,
            IAST2expression<Exc>[] arguments);

    /** Créer une opération unaire. */
    IAST2unaryOperation<Exc> newUnaryOperation(
            String operatorName,
            IAST2expression<Exc> operand);

    /** Créer une opération binaire. */
    IAST2binaryOperation<Exc> newBinaryOperation(
            String operatorName,
            IAST2expression<Exc> leftOperand,
            IAST2expression<Exc> rightOperand);

    /** Créer une constante littérale entière. */
    IAST2integer<Exc> newIntegerConstant(String value);

    /** Créer une constante littérale flottante. */
    IAST2float<Exc> newFloatConstant(String value);

    /** Créer une constante littérale chaîne de caractères. */
    IAST2string<Exc> newStringConstant(String value);

    /** Créer une constante littérale booléenne. */
    IAST2boolean<Exc> newBooleanConstant(String value);

    /** Créer une affectation. */
    IAST2assignment<Exc> newAssignment(
            IAST2variable variable,
            IAST2expression<Exc> value);

    /** Créer un bloc local n-aire. */
    IAST2localBlock<Exc> newLocalBlock(
            IAST2variable[] variables,
            IAST2expression<Exc>[] initializations,
            IAST2instruction<Exc> body);

    /** Par compatibilite, creer un bloc local unaire. */
    IAST2unaryBlock<Exc> newUnaryBlock(
            IAST2variable variable,
            IAST2expression<Exc> initialization,
            IAST2instruction<Exc> body);

    /** Créer une boucle tant-que. */
    IAST2while<Exc> newWhile(
            IAST2expression<Exc> condition,
            IAST2instruction<Exc> body);

    /** Créer une definition de fonctions. */
    IAST2functionDefinition<Exc> newFunctionDefinition(
            String functionName,
            IAST2variable[] variables,
            IAST2instruction<Exc> body);
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2alternative.java

```
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2alternative<Exc extends Exception>
extends IAST2instruction<Exc> {

      /** Renvoie la condition. */
      IAST2expression<Exc> getCondition ();

      /** Renvoie la conséquence. */
      IAST2instruction<Exc> getConsequent ();

      /** Renvoie l'alternant si présent.
       *
       * @throws Exc lorsqu'un tel argument n'existe pas.
       */
      IAST2instruction<Exc> getAlternant () throws Exc;

      /** Indique si l'alternative est ternaire (qu'elle a un alternant). */
      boolean isTernary ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2assignment.java

```java
package fr.upmc.ilp.ilp2.interfaces;

/** L"interface equivalente n'existait pas en ILP1 */

public interface IAST2assignment<Exc extends Exception>
extends IAST2expression<Exc> {

    IAST2expression<Exc> getValue ();
    IAST2variable getVariable ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2binaryOperation.java

```java
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2binaryOperation<Exc extends Exception>
extends IAST2operation<Exc> {

        /** renvoie l'op?rande de gauche. */
    IAST2expression<Exc> getLeftOperand ();

    /** renvoie l'op?rande de droite. */
    IAST2expression<Exc> getRightOperand ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2boolean.java

```java
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2boolean<Exc extends Exception>
extends IAST2constant<Exc> {
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2constant.java

```java
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2constant<Exc extends Exception>
extends IAST2expression<Exc> {

    /** La description precise et sans perte de la constante. */
    String getDescription();

    /** Une valeur Java representant une constante ILP sans garantie
     * d'exactitude. Utile pour l'interpretation! */
    Object getValue ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2expression.java

```java
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;

public interface IAST2expression<Exc extends Exception>
extends IAST2instruction<Exc> {

    /** Compilation d'une 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 compileExpression (StringBuffer buffer,
                            ICgenLexicalEnvironment lexenv,
                            ICgenEnvironment common,
                            IDestination destination)
      throws CgenerationException;

    /** Compilation d'une expression. Production de code C par ajout à
     * un tampon, dans un environnement lexical et un environnement
     * global. Le résultat est produit en ligne sans destination. C'est
     * donc un raccourci pour utiliser la methode precedente.
     */

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


Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2float.java

```java
package fr.upmc.ilp.ilp2.interfaces;


public interface IAST2float<Exc extends Exception>
extends IAST2constant<Exc> {
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2functionDefinition.java

```java
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;

public interface IAST2functionDefinition<Exc extends Exception>
extends IAST2<Exc> {

    String getFunctionName ();
    String getMangledFunctionName ();

    IAST2variable[] getVariables ();

    IAST2instruction<Exc> getBody ();

    void compileHeader (StringBuffer buffer,
                        ICgenLexicalEnvironment lexenv,
                        ICgenEnvironment common )
    throws CgenerationException;

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

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2instruction.java

```java
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;

public interface IAST2instruction<Exc extends Exception>
extends IAST2<Exc> {

    /** Compilation d'une instruction. 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 compileInstruction (StringBuffer buffer,
                             ICgenLexicalEnvironment lexenv,
                             ICgenEnvironment common,
                             IDestination destination)
      throws CgenerationException;
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2integer.java

```java
package fr.upmc.ilp.ilp2.interfaces;


public interface IAST2integer<Exc extends Exception>
extends IAST2constant<Exc> {
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2invocation.java

```
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2invocation<Exc extends Exception>
extends IAST2expression<Exc> {

    /** Renvoie la fonction invoquée. */
    IAST2expression<Exc> getFunction ();

    /** Renvoie les arguments de l'invocation sous forme d'une liste. */
    IAST2expression<Exc>[] getArguments ();

    /** Renvoie le nombre d'arguments de l'invocation. */
    int getArgumentsLength ();

    /** Renvoie le i-ème argument de l'invocation. */
    IAST2expression<Exc> getArgument (int i);
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2localBlock.java

```
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2localBlock<Exc extends Exception>
extends IAST2instruction<Exc> {

    IAST2variable[] getVariables ();

    IAST2expression<Exc>[] getInitializations ();

    IAST2instruction<Exc> getBody ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2operation.java

```
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2operation<Exc extends Exception>
extends IAST2expression<Exc> {

    /** Renvoie le nom de l'op?rateur concern? par l'op?ration. */
    String getOperatorName ();

    /** Renvoie l'arit? de l'op?rateur concern? par l'op?ration. L'arit?
     * est toujours de 1 pour IAST2unaryOperation et de 2 pour une
     * IAST2binaryOperation. */
    int getArity ();

    /** Renvoie les op?randes d'une op?ration. */
    IAST2expression<Exc>[] getOperands ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2primitiveInvocation.java

```
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2primitiveInvocation<Exc extends Exception>
extends IAST2invocation<Exc> {
    String getPrimitiveName ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2program.java

```
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;

public interface IAST2program<Exc extends Exception>
extends IAST2<Exc> {
```



```
    IAST2instruction<Exc> getBody ();

    IAST2functionDefinition<Exc>[] getFunctionDefinitions ();

    IAST2variable[] getGlobalVariables ();

    void computeGlobalVariables (ICgenLexicalEnvironment lexenv);

    String compile (ICgenLexicalEnvironment lexenv,
                    ICgenEnvironment common )
        throws CgenerationException;
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2reference.java

```
package fr.upmc.ilp.ilp2.interfaces;

/** Cette interface d?crit une r?f?rence en lecture ? une variable
 * car on distingue maintenant lecture et r?f?rence ? une variable. */

public interface IAST2reference<Exc extends Exception>
extends IAST2expression<Exc> {

    /** Retourne la variable lue */
    IAST2variable getVariable();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2sequence.java

```
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2sequence<Exc extends Exception>
extends IAST2instruction<Exc> {

    /** Renvoie la s?quence des instructions contenues. */
    IAST2instruction<Exc>[] getInstructions ();

    /** Renvoie le nombre d'instructions de la sequence. */
    int getInstructionsLength ();

    /** Renvoie la i-eme instruction.
     *
     * @throws Exc lorsqu'un tel argument n'existe pas. */
    IAST2instruction<Exc> getInstruction (int i) throws Exc;
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2string.java

```
package fr.upmc.ilp.ilp2.interfaces;


public interface IAST2string<Exc extends Exception>
extends IAST2constant<Exc> {
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2unaryBlock.java

```
package fr.upmc.ilp.ilp2.interfaces;


public interface IAST2unaryBlock<Exc extends Exception>
extends IAST2instruction<Exc> {

    IAST2variable getVariable ();

    IAST2expression<Exc> getInitialization ();

    IAST2instruction<Exc> getBody ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2unaryOperation.java

```java
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2unaryOperation<Exc extends Exception>
extends IAST2operation<Exc> {

    IAST2expression<Exc> getOperand ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2variable.java

```java
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.interfaces.IASTvariable;

public interface IAST2variable
extends IASTvariable {

    String getMangledName ();

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

Java/src/fr/upmc/ilp/ilp2/interfaces/IAST2while.java

```java
package fr.upmc.ilp.ilp2.interfaces;

public interface IAST2while<Exc extends Exception>
extends IAST2instruction<Exc> {

    IAST2expression<Exc> getCondition ();
    IAST2instruction<Exc> getBody ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/ICgenEnvironment.java

```java
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;

/** L'interface decrivant l'environnement des operateurs pr?d?finis du
 * langage a compiler vers C. Il est l'analogue de runtime/ICommon
 * pour le paquetage cgen. */

public interface ICgenEnvironment {

  /** Comment convertir un operateur unaire en C. */

  String compileOperator1 (String opName)
    throws CgenerationException ;

  /** Comment convertir un operateur binaire en C. */

  String compileOperator2 (String opName)
    throws CgenerationException ;

  /** Comment convertir le nom d'une primitive en C. */

  String compilePrimitive (String primitiveName)
    throws CgenerationException;

  /** Comment compiler une variable globale en C. */

  String compileGlobal (IAST2variable variable)
    throws CgenerationException;

  /** Enregistrer une nouvelle variable globale. */

  void bindGlobal (IAST2variable var);

  /** Enregister une nouvelle primitive. */
```



```java
  void bindPrimitive (final String primitiveName);

  //@Deprecated
  //void bindGlobal (String varName);

  /** Verifier si une variable globale est presente ? */

  boolean isPresent (String variableName);
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/ICgenLexicalEnvironment.java

```java
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;

/** L'interface décrivant l'environnement lexical de compilation vers
 * C. Il est l'analogue de runtime/ILexicalEnvironment pour le
 * paquetage cgen. */

public interface ICgenLexicalEnvironment
extends IEnvironment<IAST2variable> {

  // Soyons covariant:
  ICgenLexicalEnvironment getNext ();
  ICgenLexicalEnvironment shrink (IAST2variable variable);

  /** Renvoie le code compilé d'accès à cette variable.
   *
   * @throws CgenerationException si la variable est absente.
   */

  String compile (IAST2variable variable)
    throws CgenerationException;

  /** Étend l'environnement avec une nouvelle variable et vers quel
   * nom la compiler. */
  ICgenLexicalEnvironment extend (IAST2variable variable);
  ICgenLexicalEnvironment extend (IAST2variable variable, String cname);
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/ICommon.java

```java
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;

/** Cette interface definit les caracteristiques globales d'un interprete du
 * langage ILP2. On y trouve, notamment, la definition des operateurs du
 * langage. */

public interface ICommon
extends fr.upmc.ilp.ilp1.runtime.ICommon {

  /** Determiner la valeur d'une primitive. */
  Object primitiveLookup (String primitiveName)
    throws EvaluationException;

  /** Associer une valeur a une primitive. */
  void bindPrimitive (String primitiveName, Object value)
    throws EvaluationException;

  /** Determiner la valeur d'une variable globale. */
  Object globalLookup (IAST2variable variable)
    throws EvaluationException;

  /** Associer une valeur a une variable globale. */
  void updateGlobal (String variableName, Object value)
    throws EvaluationException;

  /** Determiner si une variable globale est presente. */
  boolean isPresent (String variableName);
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IDestination.java

```
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;

/** Une interface pour décrire la destination d'une compilation.
 * Actuellement, il y a trois destinations possibles c'est-à-dire
 * trois implantations de cette interface: (void), return ou
 * l'affectation à une variable.  */

public interface IDestination {

    /** Émettre une destination dans le tampon de sortie. */
    void compile (final StringBuffer buffer,
                  final ICgenLexicalEnvironment lexenv,
                  final ICgenEnvironment common ) throws CgenerationException ;
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IEnvironment.java

```
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.annotation.OrNull;

/** Beaucoup d'environnements associent des variables a des valeurs.
 * Cet environnement contient les fonctions primordiales qu'on
 * attend d'eux. Cet environnement n'est fondamentalement qu'une
 * simple liste chainée. */

public interface IEnvironment<V> {

    /** Retourne le premier maillon concernant la variable (s'il existe) */
    @OrNull IEnvironment<V> shrink (V variable);

    /** Vérifie qu'une variable est présente dans l'environnement. */
    boolean isPresent (V variable);
    // isPresent s'implante facilement avec shrink!

    // Introspection des environnements

    /** L'environnement est-il vide ? */
    boolean isEmpty ();

    /** Si l'environnement n'est pas vide, rendre le maillon suivant. */
    IEnvironment<V> getNext ();

    /** Quelle est la variable concernée par le maillon courant. */
    V getVariable ();
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/ILexicalEnvironment.java

```
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;

/** Cette interface définit un environnement lexical pour une
 * évaluation. Un environnement est une structure de données présente
 * à l'exécution et contenant une suite de couples (on dit « liaison »)
 * (variable, valeur) de cette variable. */

public interface ILexicalEnvironment
extends IEnvironment<IAST2variable> {

    /** Un peu de covariance. */
    ILexicalEnvironment getNext ();
    @OrNull ILexicalEnvironment shrink (IAST2variable variable);

    Object lookup (IAST2variable variable)
    throws EvaluationException;

    /** Met à jour l'environnement d'interprétation en associant une
     * nouvelle valeur à une variable.
     *
     * @throws EvaluationException si la variable est absente.
     */

    void update (IAST2variable variable, Object value)
    throws EvaluationException;

    /** Étend l'environnement avec un nouveau couple variable-valeur. */
    ILexicalEnvironment extend (IAST2variable variable, Object value);
}
```


Java/src/fr/upmc/ilp/ilp2/interfaces/IParser.java

```
package fr.upmc.ilp.ilp2.interfaces;

import java.util.List;

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

/**
 * L'interface des analyseurs syntaxiques devant analyser un DOM et le
 * convertir en un AST. Toutes ces methodes peuvent signaler des exceptions
 * provenant de la conversion.
 */

public interface IParser<Exc extends Exception> {

    /** Rendre la fabrique associee a cet analyseur. */

    IAST2Factory<Exc> getFactory ();

    /** Analyser un noeud de DOM et le convertir en un AST. */

    IAST2program<Exc> parse (Document n) throws Exc;
    IAST2<Exc> parse (Node n) throws Exc;

    /** Analyser une suite de noeuds en une suite d'AST. */

    List<IAST2<Exc>> parseList (NodeList nl) throws Exc;
    // FUTUR: changer plutot pour IAST2[] ?

    /** Trouver un sous-noeud de nom donné et le convertir en un AST.
     * On peut fournir un noeud ou une suite de noeuds. */

    IAST2<Exc> findThenParseChild (NodeList nl, String childName)
    throws Exc;
    IAST2<Exc> findThenParseChild (Node n, String childName)
    throws Exc;

    /** Trouver un sous-noeud donné et convertir ses fils en une suite d'AST.
     * On peut fournir un noeud ou une suite de noeuds. */

    List<IAST2<Exc>> findThenParseChildAsList (NodeList nl, String childName)
    throws Exc;
    List<IAST2<Exc>> findThenParseChildAsList (Node n, String childName)
    throws Exc;

    /** Trouver un sous-noeud donné et convertir ses fils en une séquence
     * d'instructions.
     * On peut fournir un noeud ou une suite de noeuds. */

    IAST2sequence<Exc> findThenParseChildAsSequence (NodeList nl, String childName)
    throws Exc;
    IAST2sequence<Exc> findThenParseChildAsSequence (Node n, String childName)
    throws Exc;

    /** Faire une sequence d'une suite de noeuds. */

    IAST2sequence<Exc> parseChildrenAsSequence (NodeList nl)
    throws Exc;

    /** Trouver un sous-noeud donné et convertir son unique fils en un AST.
     * On peut fournir un noeud ou une suite de noeuds.*/

    IAST2<Exc> findThenParseChildAsUnique (Node n, String childName)
```

```
    throws Exc;
    IAST2<Exc> findThenParseChildAsUnique (NodeList nl, String childName)
    throws Exc;
}
```

Java/src/fr/upmc/ilp/ilp2/interfaces/IUserFunction.java

```
package fr.upmc.ilp.ilp2.interfaces;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;

/** L'interface des fonctions définies par l'utilisateur. */

public interface IUserFunction {

  /** Une fonction invoquée avec un nombre quelconque d'arguments. */

  Object invoke (Object[] arguments, ICommon common)
    throws EvaluationException;

  /** Obtenir les composantes d'une fonction utilisateur. */
  IAST2variable[] getVariables ();
  IAST2instruction<CEASTparseException> getBody ();
  ILexicalEnvironment getEnvironment ();
}
```

Java/src/fr/upmc/ilp/ilp2/ast/AbstractParser.java

```
package fr.upmc.ilp.ilp2.ast;

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

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

import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp2.interfaces.IAST2Factory;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2sequence;
import fr.upmc.ilp.ilp2.interfaces.IParser;

/** Une classe abstraite contenant quelques utilitaires communs aux différents
 * types d'analyseurs syntaxiques d'ILP2 et de ses extensions. */

public abstract class AbstractParser
implements IParser<CEASTparseException> {

    public AbstractParser (final IAST2Factory<CEASTparseException> factory) {
        this.factory = factory;
    }
    protected final IAST2Factory<CEASTparseException> factory;

    public IAST2Factory<CEASTparseException> getFactory () {
        return this.factory;
    }

    public IAST2<CEASTparseException> parseAsExpression (final NodeList nl)
    throws CEASTparseException {
        if ( 1 == nl.getLength() ) {
            return this.parse(nl.item(0));
        } else {
            final String msg = "Should contain a single DOM node!";
            throw new CEASTparseException(msg);
        }
    }

    /** Trouver un élément d'après son nom et l'analyser pour en faire
     * un CEAST.
     *
     * @throws CEASTparseException
     *   lorsqu'un tel élément n'est pas trouvé.
     */
```



```
    public IAST2<CEASTparseException>
    findThenParseChild (final Node n, final String childName)
    throws CEASTparseException {
        return findThenParseChild(n.getChildNodes(), childName);
    }

    public IAST2<CEASTparseException>
    findThenParseChild (final NodeList nl, final String childName)
    throws CEASTparseException {
        final int n = nl.getLength();
        for ( int i = 0 ; i<n ; i++ ) {
            final Node nd = nl.item(i);
            switch ( nd.getNodeType() ) {

            case Node.ELEMENT_NODE: {
                final Element e = (Element) nd;
                if ( childName.equals(e.getTagName()) ) {
                    return this.parse(e);
                }
                break;
            }

            default: {
                // On ignore tout ce qui n'est pas élément XML:
            }
            }
        }
        final String msg = "No such child element " + childName;
        throw new CEASTparseException(msg);
    }

    /** Trouver un élément d'après son nom et analyser son contenu pour
     * en faire une liste de CEAST.
     *
     * @throws CEASTparseException
     *   lorsqu'un tel élément n'est pas trouvé.
     */

    public List<IAST2<CEASTparseException>> findThenParseChildAsList (
            final Node n, final String childName)
    throws CEASTparseException {
        return findThenParseChildAsList(n.getChildNodes(), childName);
    }

    /** Trouver un certain élément d'après son nom et analyser son contenu pour
     * en faire une liste de CEAST.
     *
     * @throws CEASTparseException
     *   lorsqu'un tel élément n'est pas trouvé.
     */

    public List<IAST2<CEASTparseException>> findThenParseChildAsList (
            final NodeList nl, final String childName)
    throws CEASTparseException {
        final int n = nl.getLength();
        for ( int i = 0 ; i<n ; i++ ) {
            final Node nd = nl.item(i);
            switch ( nd.getNodeType() ) {

            case Node.ELEMENT_NODE: {
                final Element e = (Element) nd;
                if ( childName.equals(e.getTagName()) ) {
                    return this.parseList(e.getChildNodes());
                }
                break;
            }

            default: {
                // On ignore tout ce qui n'est pas élément XML:
            }
            }
        }
        final String msg = "No such node " + childName;
        throw new CEASTparseException(msg);
    }

    /** Trouver un sous-noeud d'un certain type et analyser son unique
```

```
     * fils comme une unique entité */

    public IAST2<CEASTparseException> findThenParseChildAsUnique (
            final Node n,
            final String childName)
            throws CEASTparseException {
        return findThenParseChildAsUnique(n.getChildNodes(), childName);
    }

    /** Trouver un certain sous-noeud d'un certain type et analyser son unique
     * fils comme une unique entité */

    public IAST2<CEASTparseException> findThenParseChildAsUnique (
            final NodeList nl,
            final String childName)
            throws CEASTparseException {
        final List<IAST2<CEASTparseException>> results =
            findThenParseChildAsList(nl, childName);
        if ( 1 == results.size() ) {
            return results.get(0);
        } else {
            final String msg = "Should be an unique DOM node!";
            throw new CEASTparseException(msg);
        }
    }

    /** Analyser une séquence d'élément pour en faire une suite de CEAST.
     * Cette fonction sert à lire les
     * instructions d'une séquence, les arguments d'une invocation, les
     * variables d'une définition de fonction, etc.
     *
     * @throws CEASTparseException
     *   lorsque l'analyse d'un sous-noeud provoque une exception.
     */

    public List<IAST2<CEASTparseException>> parseList (NodeList nl)
    throws CEASTparseException {
        final List<IAST2<CEASTparseException>> result = new Vector<>();
        final int n = nl.getLength();
        LOOP:
            for ( int i = 0 ; i<n ; i++ ) {
                final Node nd = nl.item(i);
                switch ( nd.getNodeType() ) {

                case Node.ELEMENT_NODE: {
                    final IAST2<CEASTparseException> p = this.parse(nd);
                    result.add(p);
                    continue LOOP;
                }

                default: {
                    // On ignore tout ce qui n'est pas élément XML:
                }
                }
            }
        return result;
    }

    /** Trouver un sous-noeud donné et convertir ses fils en une séquence
     * d'instructions. */

    public IAST2sequence<CEASTparseException> findThenParseChildAsSequence (
            NodeList nl, String childName)
    throws CEASTparseException {
        List<IAST2<CEASTparseException>> li =
            findThenParseChildAsList(nl, childName);
        List<IAST2instruction<CEASTparseException>> lins = new Vector<>();
        for (IAST2<CEASTparseException> i : li) {
            lins.add((IAST2instruction<CEASTparseException>) i);
        }
        return getFactory().newSequence(lins);
    }

    /** Trouver un certain sous-noeud et convertir ses fils en une séquence
     * d'instructions. */

    public IAST2sequence<CEASTparseException> findThenParseChildAsSequence (
            Node n, String childName)
    throws CEASTparseException {
```



```
        return findThenParseChildAsSequence(n.getChildNodes(), childName);
    }

    /** Réunir tous les sous-noeuds en une séquence. */

    public IAST2sequence<CEASTparseException> parseChildrenAsSequence (
            NodeList nl)
    throws CEASTparseException {
        List<IAST2<CEASTparseException>> li = parseList(nl);
        List<IAST2instruction<CEASTparseException>> lins = new Vector<>();
        for (IAST2<CEASTparseException> i : li) {
            lins.add((IAST2instruction<CEASTparseException>) i);
        }
        return getFactory().newSequence(lins);
    }

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEAST.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;

/** Code commun à CEASTInstruction, à CEASTfunctionDefinition et à
 * CEASTprogram et a toutes leurs sous-classes. */

public abstract class CEAST
  implements IAST2<CEASTparseException> {
    // Pas moyen d'imposer que les sous-classes aient toutes
    // une méthode statique parse(Element, IParser).

    /** Chercher les variables globales dans un AST. Les variables trouvées
     * sont insérées dans l'ensemble globalvars. */
    public abstract void
    findGlobalVariables (final Set<IAST2variable> globalvars,
                         final ICgenLexicalEnvironment lexenv  );
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTFactory.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.List;

import fr.upmc.ilp.ilp2.interfaces.IAST2Factory;
import fr.upmc.ilp.ilp2.interfaces.IAST2alternative;
import fr.upmc.ilp.ilp2.interfaces.IAST2assignment;
import fr.upmc.ilp.ilp2.interfaces.IAST2binaryOperation;
import fr.upmc.ilp.ilp2.interfaces.IAST2boolean;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2float;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2integer;
import fr.upmc.ilp.ilp2.interfaces.IAST2invocation;
import fr.upmc.ilp.ilp2.interfaces.IAST2localBlock;
import fr.upmc.ilp.ilp2.interfaces.IAST2primitiveInvocation;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;
import fr.upmc.ilp.ilp2.interfaces.IAST2reference;
import fr.upmc.ilp.ilp2.interfaces.IAST2sequence;
import fr.upmc.ilp.ilp2.interfaces.IAST2string;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryBlock;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryOperation;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.IAST2while;

public class CEASTFactory
implements IAST2Factory<CEASTparseException> {

    public CEASTFactory () {}
```

```
    public IAST2program<CEASTparseException> newProgram(
            IAST2functionDefinition<CEASTparseException>[] functions,
            IAST2instruction<CEASTparseException> instruction) {
        return new CEASTprogram(functions, instruction);
    }

    public IAST2alternative<CEASTparseException> newAlternative(
            IAST2expression<CEASTparseException> condition,
            IAST2instruction<CEASTparseException> consequent,
            IAST2instruction<CEASTparseException> alternant) {
        return new CEASTalternative(condition, consequent, alternant);
    }

    public IAST2alternative<CEASTparseException> newAlternative(
            IAST2expression<CEASTparseException> condition,
            IAST2instruction<CEASTparseException> consequent) {
        return new CEASTalternative(condition, consequent);
    }

    public IAST2assignment<CEASTparseException> newAssignment(
            IAST2variable variable, IAST2expression<CEASTparseException> value) {
        return new CEASTassignment(variable, value);
    }

    public IAST2binaryOperation<CEASTparseException> newBinaryOperation(
            String operatorName,
            IAST2expression<CEASTparseException> leftOperand,
            IAST2expression<CEASTparseException> rightOperand) {
        return new CEASTbinaryOperation(operatorName, leftOperand, rightOperand);
    }

    public IAST2boolean<CEASTparseException> newBooleanConstant(String value) {
        return new CEASTboolean(value);
    }

    public IAST2float<CEASTparseException> newFloatConstant(String value) {
        return new CEASTfloat(value);
    }

    public IAST2functionDefinition<CEASTparseException> newFunctionDefinition(
            String functionName,
            IAST2variable[] variables,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTfunctionDefinition(functionName, variables, body);
    }

    public IAST2integer<CEASTparseException> newIntegerConstant(String value) {
        return new CEASTinteger(value);
    }

    public IAST2invocation<CEASTparseException> newInvocation(
            IAST2expression<CEASTparseException> function,
            IAST2expression<CEASTparseException>[] arguments) {
        return new CEASTinvocation(function, arguments);
    }

    public IAST2localBlock<CEASTparseException> newLocalBlock(
            IAST2variable[] variables,
            IAST2expression<CEASTparseException>[] initializations,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTlocalBlock(variables, initializations, body);
    }

    public IAST2primitiveInvocation<CEASTparseException> newPrimitiveInvocation(
            String primitiveName,
            IAST2expression<CEASTparseException>[] arguments) {
        return new CEASTprimitiveInvocation(primitiveName, arguments);
    }

    public IAST2reference<CEASTparseException> newReference(
            IAST2variable variable) {
        return new CEASTreference(variable);
    }

    public IAST2sequence<CEASTparseException> newSequence(
            List<IAST2instruction<CEASTparseException>> asts) {
        return new CEASTsequence(asts);
    }

    public IAST2string<CEASTparseException> newStringConstant(String value) {
```



```
        return new CEASTstring(value);
    }

    public IAST2unaryBlock<CEASTparseException> newUnaryBlock(
            IAST2variable variable,
            IAST2expression<CEASTparseException> initialization,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTunaryBlock(variable, initialization, body);
    }

    public IAST2unaryOperation<CEASTparseException> newUnaryOperation(
            String operatorName, IAST2expression<CEASTparseException> operand) {
        return new CEASTunaryOperation(operatorName, operand);
    }

    public IAST2variable newVariable(String name) {
        return new CEASTvariable(name);
    }

    public IAST2while<CEASTparseException> newWhile(
            IAST2expression<CEASTparseException> condition,
            IAST2instruction<CEASTparseException> body) {
        return new CEASTwhile(condition, body);
    }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTParser.java

```
package fr.upmc.ilp.ilp2.ast;

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

import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;
import fr.upmc.ilp.ilp2.interfaces.IAST2Factory;

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

public class CEASTParser extends AbstractParser {

    public CEASTParser (IAST2Factory<CEASTparseException> factory) {
        super(factory);
    }

    public IAST2program<CEASTparseException> parse (final Document d)
    throws CEASTparseException {
        final Element e = d.getDocumentElement();
        return CEASTprogram.parse(e, this);
    }

  public IAST2<CEASTparseException> parse (final Node n)
    throws CEASTparseException {
    switch ( n.getNodeType() ) {

    case Node.ELEMENT_NODE: {
      final Element e = (Element) n;
      final String name = e.getTagName();
      //System.err.println("Parsing " + name + " ..."); // DEBUG
      switch (name) {
      case "programme1":
      case "programme2":
          return CEASTprogram.parse(e, this);
      case "alternative":
        return CEASTalternative.parse(e, this);
      case "sequence":
        return CEASTsequence.parse(e, this);
      case "boucle":
        return CEASTwhile.parse(e, this);
      case "affectation":
        return CEASTassignment.parse(e, this);
      case "definitionFonction":
        return CEASTfunctionDefinition.parse(e, this);
      case "blocUnaire":
        return CEASTunaryBlock.parse(e, this);
      case "blocLocal":
```

```
        return CEASTlocalBlock.parse(e, this);
      case "variable":
        return CEASTreference.parse(e, this);
      case "invocationPrimitive":
        return CEASTprimitiveInvocation.parse(e, this);
      case "invocation":
        return CEASTinvocation.parse(e, this);
      case "operationUnaire":
        return CEASTunaryOperation.parse(e, this);
      case "operationBinaire":
        return CEASTbinaryOperation.parse(e, this);
      case "entier":
        return CEASTinteger.parse(e, this);
      case "flottant":
        return CEASTfloat.parse(e, this);
      case "chaine":
        return CEASTstring.parse(e, this);
      case "booleen":
        return CEASTboolean.parse(e, this);
      default: {
        final String msg = "Unknown element name: " + name;
        throw new CEASTparseException(msg);
      }
      }
    }

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

Java/src/fr/upmc/ilp/ilp2/ast/CEASTalternative.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2alternative;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
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.ilp2.interfaces.IParser;

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

public class CEASTalternative
  extends CEASTinstruction
  implements IAST2alternative<CEASTparseException> {

  public CEASTalternative (
          final IAST2expression<CEASTparseException> condition,
          final IAST2instruction<CEASTparseException> consequence,
          final IAST2instruction<CEASTparseException> alternant ) {
    this.condition   = condition;
    this.consequence = consequence;
    this.alternant   = alternant;
    this.ternary     = true;
  }
  public CEASTalternative (
          final IAST2expression<CEASTparseException> condition,
          final IAST2instruction<CEASTparseException> consequence ) {
      this.condition   = condition;
      this.consequence = consequence;
```



```
      this.alternant   = CEASTinstruction.voidInstruction();
      this.ternary     = false;
  }
  private final IAST2expression<CEASTparseException> condition;
  private final IAST2instruction<CEASTparseException> consequence;
  private final IAST2instruction<CEASTparseException> alternant;
  private boolean ternary;

  public IAST2expression<CEASTparseException> getCondition () {
      return this.condition;
  }
  public IAST2instruction<CEASTparseException> getConsequent (){
      return this.consequence;
  }
  // Plus de @OrNull: impossible si emploi du bon constructeur:
  public IAST2instruction<CEASTparseException> getAlternant () {
      return this.alternant;
  }
  public boolean isTernary () {
      return this.ternary;
  }

  public static IAST2alternative<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      final NodeList nl = e.getChildNodes();
      IAST2expression<CEASTparseException> condition =
          (IAST2expression<CEASTparseException>)
          parser.findThenParseChildAsUnique(nl, "condition");
      IAST2instruction<CEASTparseException> consequence =
          (IAST2instruction<CEASTparseException>)
          parser.findThenParseChildAsSequence(nl, "consequence");
      try {
          IAST2instruction<CEASTparseException> alternant =
              (IAST2instruction<CEASTparseException>)
              parser.findThenParseChildAsSequence(nl, "alternant");
          return parser.getFactory().newAlternative(
                  condition, consequence, alternant);
      } catch (CEASTparseException exc) {
          return parser.getFactory().newAlternative(condition, consequence);
      }
  }

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

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    Object bool = getCondition().eval(lexenv, common);
    if ( bool instanceof Boolean ) {
      Boolean b = (Boolean) bool;
      if ( b.booleanValue() ) {
        return getConsequent().eval(lexenv, common);
      } else {
          return getAlternant().eval(lexenv, common);
      }
    } else {
      return getConsequent().eval(lexenv, common);
    }
  }

  public void compileInstruction (final StringBuffer buffer,
                                  final ICgenLexicalEnvironment lexenv,
                                  final ICgenEnvironment common,
                                  final IDestination destination)
    throws CgenerationException {
    buffer.append(" if ( ILP_isEquivalentToTrue( ");
    getCondition().compileExpression(buffer, lexenv, common);
    buffer.append(" ) ) {\n");
    getConsequent().compileInstruction(buffer, lexenv, common, destination);
    buffer.append(";\n} else {\n");
    getAlternant().compileInstruction(buffer, lexenv, common, destination);
    buffer.append(";\n}");
  }

  @Override
  public void
    findGlobalVariables (final Set<IAST2variable> globalvars,
```

```
                        final ICgenLexicalEnvironment lexenv  ) {
    getCondition().findGlobalVariables(globalvars, lexenv);
    getConsequent().findGlobalVariables(globalvars, lexenv);
    getAlternant().findGlobalVariables(globalvars, lexenv);
  }

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTassignment.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.cgen.AssignDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2assignment;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
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.ilp2.interfaces.IParser;

public class CEASTassignment
extends CEASTexpression
implements IAST2assignment<CEASTparseException> {

    public CEASTassignment (final IAST2variable variable,
                            final IAST2expression<CEASTparseException> value) {
        this.variable = variable;
        this.value = value;
    }
    private final IAST2variable variable;
    private final IAST2expression<CEASTparseException> value;

    public IAST2expression<CEASTparseException> getValue () {
        return this.value;
    }
    public IAST2variable getVariable () {
        return this.variable;
    }

    public static IAST2assignment<CEASTparseException> parse (
            final Element e, final IParser<CEASTparseException> parser)
    throws CEASTparseException {
        final String nick = e.getAttribute("nom");
        final IAST2variable variable = parser.getFactory().newVariable(nick);
        final IAST2expression<CEASTparseException> value =
            (IAST2expression<CEASTparseException>)
            parser.findThenParseChildAsUnique(e, "valeur");
        return parser.getFactory().newAssignment(variable, value);
    }

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

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

    public void compileExpression (final StringBuffer buffer,
                                   final ICgenLexicalEnvironment lexenv,
                                   final ICgenEnvironment common,
                                   final IDestination destination)
```



```
    throws CgenerationException {
        destination.compile(buffer, lexenv, common);
        buffer.append(" (");
        getValue().compileExpression(buffer, lexenv, common,
                new AssignDestination(getVariable()) );
        buffer.append(") ");
    }

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

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTbinaryOperation.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2binaryOperation;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
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.ilp2.interfaces.IParser;

/** Opérateurs binaires. */

public class CEASTbinaryOperation extends CEASToperation
  implements IAST2binaryOperation<CEASTparseException> {

  public CEASTbinaryOperation (
          final String operatorName,
          final IAST2expression<CEASTparseException> left,
          final IAST2expression<CEASTparseException> right)
  {
    super(operatorName, 2);
    this.left = left;
    this.right = right;
  }
  private final IAST2expression<CEASTparseException> left;
  private final IAST2expression<CEASTparseException> right;

  public IAST2expression<CEASTparseException> getLeftOperand() {
      return this.left;
  }
  public IAST2expression<CEASTparseException> getRightOperand() {
      return this.right;
  }
  public IAST2expression<CEASTparseException>[] getOperands () {
      final CEASTexpression[] result = {
              (CEASTexpression) this.left,
              (CEASTexpression) this.right,
      };
      return result;
  }

  public static IAST2binaryOperation<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      String operatorName = e.getAttribute("operateur");
      final NodeList nl = e.getChildNodes();
```

```
        IAST2expression<CEASTparseException> leftOperand =
            (IAST2expression<CEASTparseException>)
            parser.findThenParseChildAsUnique(nl, "operandeGauche");
        IAST2expression<CEASTparseException> rightOperand =
            (IAST2expression<CEASTparseException>)
            parser.findThenParseChildAsUnique(nl, "operandeDroit");
        return parser.getFactory().newBinaryOperation(
                operatorName, leftOperand, rightOperand);
    }

    //NOTE: Acces direct aux champs interdit a partir d'ici!

    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common )
      throws EvaluationException {
      Object r1 = getLeftOperand().eval(lexenv, common);
      Object r2 = getRightOperand().eval(lexenv, common);
      return common.applyOperator(getOperatorName(), r1, r2);
    }

    public void compileExpression (final StringBuffer buffer,
                                   final ICgenLexicalEnvironment lexenv,
                                   final ICgenEnvironment common,
                                   final IDestination destination)
      throws CgenerationException {
      destination.compile(buffer, lexenv, common);
      buffer.append(" ");
      buffer.append(common.compileOperator2(getOperatorName()));
      buffer.append("(");
      getLeftOperand().compileExpression(buffer, lexenv, common);
      buffer.append(", ");
      getRightOperand().compileExpression(buffer, lexenv, common);
      buffer.append(") ");
    }

    @Override
    public void
      findGlobalVariables (final Set<IAST2variable> globalvars,
                           final ICgenLexicalEnvironment lexenv ) {
      getLeftOperand().findGlobalVariables(globalvars, lexenv);
      getRightOperand().findGlobalVariables(globalvars, lexenv);
    }

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTboolean.java

```
package fr.upmc.ilp.ilp2.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2boolean;
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.ilp2.interfaces.IParser;

/** Les constantes booleennes.*/

public class CEASTboolean
extends CEASTconstant
implements IAST2boolean<CEASTparseException> {

  public CEASTboolean (String value) {
    super(value, "true".equals(value));
  }

  public static IAST2boolean<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      String value = e.getAttribute("valeur");
      return parser.getFactory().newBooleanConstant(value);
  }

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

  public void compileExpression (final StringBuffer buffer,
```



```
                                 final ICgenLexicalEnvironment lexenv,
                                 final ICgenEnvironment common,
                                 final IDestination destination)
      throws CgenerationException {
      destination.compile(buffer, lexenv, common);
      if ( Boolean.FALSE != getValue() ) {
          buffer.append(" ILP_TRUE ");
      } else {
        buffer.append(" ILP_FALSE ");
      }
    }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTconstant.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

import fr.upmc.ilp.ilp2.interfaces.IAST2constant;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;

/** Les constantes et leur interprétation. */

public abstract class CEASTconstant
  extends CEASTexpression
  implements IAST2constant<CEASTparseException> {

  public CEASTconstant (final String description, final Object value) {
      this.description = description;
      this.valueAsObject = value;
  }
  private final String description;
  private final Object valueAsObject;

  public Object getValue() {
      return this.valueAsObject;
  }
  public String getDescription () {
      return this.description;
  }

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

  /** Toutes les constantes valent leur propre valeur. */

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common) {
    return getValue();
  }

  @Override
  public void
  findGlobalVariables (final Set<IAST2variable> globalvars,
                       final ICgenLexicalEnvironment lexenv ) {
      return;
  }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTexpression.java

```
package fr.upmc.ilp.ilp2.ast;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.cgen.NoDestination;
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;

/** La classe des expressions. Rappelons que les expressions sont des
 * instructions (comme en C). */
```

```
public abstract class CEASTexpression
  extends CEASTinstruction
  implements IAST2expression<CEASTparseException> {

  /** Compiler une instruction c'est ajouter un point-virgule a la
   * compilation de l'expression. */
  public void compileInstruction (final StringBuffer buffer,
                                  final ICgenLexicalEnvironment lexenv,
                                  final ICgenEnvironment common,
                                  final IDestination destination)
    throws CgenerationException {
    this.compileExpression(buffer, lexenv, common, destination);
    buffer.append("; ");
  }

  /** Compiler une expression sans destination. */
  public void compileExpression (final StringBuffer buffer,
                                 final ICgenLexicalEnvironment lexenv,
                                 final ICgenEnvironment common)
    throws CgenerationException {
    this.compileExpression(buffer, lexenv, common, NoDestination.create());
  }

  /** Une constante utile pour les conversions entre liste et tableau. */
  @SuppressWarnings("unchecked")
  public static final
  IAST2expression<CEASTparseException>[] EMPTY_EXPRESSION_ARRAY =
          new IAST2expression[0];

  /** Renvoyer une expression vide qui ne fait rien mais, comme il faut
   * rendre une valeur, on renvoie FALSE.  */
  public static IAST2expression<CEASTparseException> voidExpression () {
    return new CEASTboolean("false");
  }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTfloat.java

```
package fr.upmc.ilp.ilp2.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2float;
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.ilp2.interfaces.IParser;

/** Les constantes flottantes. */

public class CEASTfloat
extends CEASTconstant
implements IAST2float<CEASTparseException> {

  public CEASTfloat (final String value) {
    super(value, new Double(value));
  }

  public static IAST2float<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      String value = e.getAttribute("valeur");
      return parser.getFactory().newFloatConstant(value);
  }

  //NOTE: Acc?s direct aux champs interdit ? partir d'ici!

  public void compileExpression (final StringBuffer buffer,
                                 final ICgenLexicalEnvironment lexenv,
                                 final ICgenEnvironment common,
                                 final IDestination destination)
    throws CgenerationException {
    final Double value = (Double) getValue();
    final double d = value.doubleValue();
    destination.compile(buffer, lexenv, common);
    buffer.append(" ILP_Float2ILP(");
    buffer.append(d);
    buffer.append(") ");
  }
}
```


Java/src/fr/upmc/ilp/ilp2/ast/CEASTfunctionDefinition.java

```
package fr.upmc.ilp.ilp2.ast;

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

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.cgen.ReturnDestination;
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.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.ilp2.interfaces.IParser;
import fr.upmc.ilp.ilp2.runtime.UserGlobalFunction;
import fr.upmc.ilp.tool.CStuff;

public class CEASTfunctionDefinition
  extends CEAST
  implements IAST2functionDefinition<CEASTparseException> {

  public CEASTfunctionDefinition (
          final String name,
          final IAST2variable[] variables,
          final IAST2instruction<CEASTparseException> body ) {
      this.name = name;
      this.variable = variables;
      this.body = body;
  }
  private final String name;
  private final IAST2variable[] variable;
  private final IAST2instruction<CEASTparseException> body;

  public String getFunctionName () {
      return this.name;
  }
  public String getMangledFunctionName () {
      return CStuff.mangle(this.name);
  }
  public IAST2variable[] getVariables () {
      return this.variable;
  }
  public IAST2instruction<CEASTparseException> getBody () {
      return this.body;
  }

  public static IAST2functionDefinition<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      final NodeList nl = e.getChildNodes();
      IAST2variable[] variables = new IAST2variable[0];
      String name = e.getAttribute("nom");
      try {
          final XPathExpression varsPath = xPath.compile("variables/*");
          final NodeList nlVars = (NodeList)
          varsPath.evaluate(e, XPathConstants.NODESET);
          final List<IAST2variable> vars = new Vector<>();
```

```
        for ( int i=0 ; i<nlVars.getLength() ; i++ ) {
            final Element varNode = (Element) nlVars.item(i);
            final IAST2variable var =
                parser.getFactory().newVariable(varNode.getAttribute("nom"));
            vars.add(var);
        }
        variables = vars.toArray(new IAST2variable[]{});
    } catch (XPathExpressionException e1) {
        throw new CEASTparseException(e1);
    }
    IAST2instruction<CEASTparseException> body =
        (IAST2instruction<CEASTparseException>)
            parser.findThenParseChildAsSequence(nl, "corps");
    return parser.getFactory().newFunctionDefinition(
            name, variables, body );
}
private static final XPath xPath = XPathFactory.newInstance().newXPath();

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

public Object eval (final ILexicalEnvironment lexenv,
                    final ICommon common)
  throws EvaluationException {
  final Object function =
      new UserGlobalFunction(
              getFunctionName(),
              getVariables(),
              getBody());
  common.updateGlobal(getFunctionName(), function);
  return function;
}

public void compileHeader (final StringBuffer buffer,
                           final ICgenLexicalEnvironment lexenv,
                           final ICgenEnvironment common)
  throws CgenerationException {
  buffer.append("static ILP_Object ");
  buffer.append(getMangledFunctionName());
  final ICgenLexicalEnvironment lexenv2 = this.extend(lexenv);
  this.compileVariableList(buffer, lexenv2, common);
  buffer.append(";\n");
}

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

public ICgenLexicalEnvironment extend (final ICgenLexicalEnvironment lexenv)
{
  ICgenLexicalEnvironment newlexenv = lexenv;
  final IAST2variable[] vars = getVariables();
  for ( int i = 0 ; i<vars.length ; i++ ) {
    newlexenv = newlexenv.extend(vars[i]);
  }
  return newlexenv;
}

public void compile (final StringBuffer buffer,
                     final ICgenLexicalEnvironment lexenv,
                     final ICgenEnvironment common )
  throws CgenerationException {
  buffer.append("\nILP_Object\n");
  buffer.append(getMangledFunctionName());
  final ICgenLexicalEnvironment lexenv2 = this.extend(lexenv);
  compileVariableList(buffer, lexenv2, common);
  buffer.append("\n");
  getBody().compileInstruction(
          buffer, lexenv2, common, ReturnDestination.create());
  buffer.append("\n");
}

public void compileVariableList (final StringBuffer buffer,
                                 final ICgenLexicalEnvironment lexenv,
                                 final ICgenEnvironment common)
  throws CgenerationException {
```



```
  buffer.append(" (");
  final IAST2variable[] vars = getVariables();
  for ( int i = 0 ; i<vars.length-1 ; i++ ) {
    buffer.append("    ILP_Object ");
    buffer.append(vars[i].getMangledName());
    buffer.append(",\n");
  }
  if ( vars.length > 0 ) {
    buffer.append("    ILP_Object ");
    buffer.append(vars[vars.length-1].getMangledName());
  }
  buffer.append(" ) ");
 }

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTinstruction.java

```
package fr.upmc.ilp.ilp2.ast;

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

/** La classe abstraite des instructions. */

public abstract class CEASTinstruction
  extends CEAST
  implements IAST2instruction<CEASTparseException> {

    /** Renvoyer une instruction vide qui ne fait rien et, comme il faut
     * rendre une valeur, on renvoie FALSE.  */
    public static IAST2instruction<CEASTparseException> voidInstruction () {
        return CEASTexpression.voidExpression();
    }

    /** Une constante utile pour les conversions entre liste et tableau. */
    @SuppressWarnings("unchecked")
    public final static
    IAST2instruction<CEASTparseException>[] EMPTY_INSTRUCTION_ARRAY =
        new IAST2instruction[0];
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTinteger.java

```
package fr.upmc.ilp.ilp2.ast;

import java.math.BigInteger;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2integer;
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.ilp2.interfaces.IParser;

/** Les constantes entieres. */

public class CEASTinteger
extends CEASTconstant
implements IAST2integer<CEASTparseException> {

  public CEASTinteger (final String value) {
    super(value, new BigInteger(value));
  }

  /** Le constructeur analysant syntaxiquement un DOM. */

  public static IAST2integer<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      String value = e.getAttribute("valeur");
      return parser.getFactory().newIntegerConstant(value);
  }

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

```
  public void compileExpression (final StringBuffer buffer,
                                 final ICgenLexicalEnvironment lexenv,
                                 final ICgenEnvironment common,
                                 final IDestination destination)
    throws CgenerationException {
    final BigInteger value = (BigInteger) getValue();
    // && et non || comme l'a remarqué Nicolas.Bros@gmail.com
    if (    value.compareTo(BIMIN) >= 0
         && value.compareTo(BIMAX) <= 0 ) {
      destination.compile(buffer, lexenv, common);
      buffer.append(" ILP_Integer2ILP(");
      buffer.append(value.intValue());
      buffer.append(") ");
    } else {
      final String msg = "Too large integer " + value;
      throw new CgenerationException(msg);
    }
  }

  // Définir les bornes encadrant les seuls entiers représentables:
  public static final BigInteger BIMIN;
  public static final BigInteger BIMAX;
  static {
    final Integer i = new Integer(Integer.MIN_VALUE);
    BIMIN = new BigInteger(i.toString());
    final Integer j = new Integer(Integer.MAX_VALUE);
    BIMAX = new BigInteger(j.toString());
  }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTinvocation.java

```
package fr.upmc.ilp.ilp2.ast;

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

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.Invokable;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2invocation;
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.ilp2.interfaces.IParser;
import fr.upmc.ilp.ilp2.interfaces.IUserFunction;

public class CEASTinvocation
  extends CEASTexpression
  implements IAST2invocation<CEASTparseException> {

  // NOTE: les deux methodes qui suivent ne peuvent etre simultanement
  // presentes car la genericite (type erasure) de Java ne le permet pas.
  // Je retire la seconde qui pose plus de problemes de typage.
  public CEASTinvocation (final IAST2expression<CEASTparseException> function,
                          final List<IAST2expression<CEASTparseException>> arguments) {
    this(function, arguments.toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY));
  }
  //public CEASTinvocation (final CEASTexpression function,
  //                        final List<CEASTexpression> arguments) {
  //    this(function, arguments.toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY);
  //}
  public CEASTinvocation (final IAST2expression<CEASTparseException> function,
                          final IAST2expression<CEASTparseException>[] arguments) {
    this.function = function;
    this.argument = arguments;
  }
  private final IAST2expression<CEASTparseException>   function;
  private final IAST2expression<CEASTparseException>[] argument;
```



```
  public IAST2expression<CEASTparseException> getFunction () {
    return this.function;
  }
  public IAST2expression<CEASTparseException>[] getArguments () {
      return this.argument;
  }
  public IAST2expression<CEASTparseException> getArgument (int i) {
      return this.argument[i];
  }
  public int getArgumentsLength () {
      return this.argument.length;
  }

  public static IAST2invocation<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      final NodeList nl = e.getChildNodes();
      IAST2expression<CEASTparseException> function =
          (IAST2expression<CEASTparseException>)
          parser.findThenParseChildAsUnique(nl, "fonction");
      IAST2expression<CEASTparseException>[] arguments =
          parser.findThenParseChildAsList(nl, "arguments")
          .toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY);
      return parser.getFactory().newInvocation(function, arguments);
  }

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

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    final Object fn = getFunction().eval(lexenv, common);
    if ( fn instanceof IUserFunction ) {
      final IUserFunction function = (IUserFunction) fn;
      final IAST2expression<CEASTparseException>[] arguments = getArguments();
      final Object[] args = new Object[arguments.length];
      for ( int i = 0 ; i<arguments.length; i++ ) {
        args[i] = arguments[i].eval(lexenv, common);
      }
      return function.invoke(args, common);
    } else if ( fn instanceof Invokable ) {
        final Invokable function = (Invokable) fn;
        final IAST2expression<CEASTparseException>[] arguments = getArguments();
        final Object[] args = new Object[arguments.length];
        for ( int i = 0 ; i<arguments.length; i++ ) {
          args[i] = arguments[i].eval(lexenv, common);
        }
        return function.invoke(args);
    } else {
      final String msg = "Not a function: " + fn;
      throw new EvaluationException(msg);
    }
  }

  public void compileExpression (final StringBuffer buffer,
                                 final ICgenLexicalEnvironment lexenv,
                                 final ICgenEnvironment common,
                                 final IDestination destination)
    throws CgenerationException {
    destination.compile(buffer, lexenv, common);
    getFunction().compileExpression(buffer, lexenv, common);
    buffer.append("(");
    final IAST2expression<CEASTparseException>[] arguments = getArguments();
    for ( int i = 0 ; i<arguments.length-1 ; i++ ) {
      arguments[i].compileExpression(buffer, lexenv, common);
      buffer.append(", ");
    }
    if ( arguments.length> 0 ) {
      arguments[arguments.length-1].compileExpression(buffer, lexenv, common);
    }
    buffer.append(")");
  }
  // NOTE: les expressions sont compilees en expressions C,
  // les instructions en instructions C.

  @Override
```

```
    public void findGlobalVariables (final Set<IAST2variable> globalvars,
                                     final ICgenLexicalEnvironment lexenv ) {
        getFunction().findGlobalVariables(globalvars, lexenv);
        for ( IAST2expression<CEASTparseException> arg : getArguments() ) {
            arg.findGlobalVariables(globalvars, lexenv);
        }
    }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTlocalBlock.java

```
package fr.upmc.ilp.ilp2.ast;

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

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.cgen.AssignDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2localBlock;
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.ilp2.interfaces.IParser;

public class CEASTlocalBlock
extends CEASTinstruction
implements IAST2localBlock<CEASTparseException> {

    private final IAST2variable[] variable;
    private final IAST2expression<CEASTparseException>[] initialization;
    private final IAST2instruction<CEASTparseException> body;

    public CEASTlocalBlock (
            final IAST2variable[] variable,
            final IAST2expression<CEASTparseException>[] initialization,
            final IAST2instruction<CEASTparseException> body ) {
        this.variable = variable;
        this.initialization = initialization;
        this.body = body;
    }

    public IAST2variable[] getVariables () {
        return this.variable;
    }
    public IAST2expression<CEASTparseException>[] getInitializations () {
        return this.initialization;
    }
    public IAST2instruction<CEASTparseException> getBody () {
        return this.body;
    }

    public static IAST2localBlock<CEASTparseException> parse (
            final Element e, final IParser<CEASTparseException> parser)
    throws CEASTparseException {
        IAST2variable[] variables = new IAST2variable[0];
        IAST2expression<CEASTparseException>[] initializations;
        try {
            final XPathExpression bindingVarsPath =
                xPath.compile("./liaisons/liaison/variable");
            final NodeList nlVars = (NodeList)
                bindingVarsPath.evaluate(e, XPathConstants.NODESET);
```



```
            final List<IAST2variable> vars = new Vector<>();
            final List<IAST2expression<CEASTparseException>> inits =
                new Vector<>();
            for ( int i=0 ; i<nlVars.getLength() ; i++ ) {
                final Element varNode = (Element) nlVars.item(i);
                final IAST2variable var =
                    parser.getFactory().newVariable(varNode.getAttribute("nom"));
                vars.add(var);
                final IAST2expression<CEASTparseException> init =
                    (IAST2expression<CEASTparseException>)
                    parser.findThenParseChildAsUnique(
                        varNode.getParentNode(),
                        "initialisation");
                inits.add(init);

            }
            variables = vars.toArray(CEASTvariable.EMPTY_VARIABLE_ARRAY);
            initializations =
                inits.toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY);
        } catch (XPathExpressionException e1) {
            throw new CEASTparseException(e1);
        }
        IAST2instruction<CEASTparseException> body =
            (IAST2instruction<CEASTparseException>)
            parser.findThenParseChildAsSequence(e, "corps");
        return parser.getFactory()
            .newLocalBlock(variables, initializations, body);
    }
    private static final XPath xPath = XPathFactory.newInstance().newXPath();
    // NOTE: factoriser xPath plus globalement ?

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

    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common)
    throws EvaluationException {
        ILexicalEnvironment newlexenv = lexenv;
        final IAST2variable[] vars = getVariables();
        final IAST2expression<CEASTparseException>[] inits = getInitializations();
        for (int i = 0; i < vars.length; i++) {
            newlexenv = newlexenv.extend(vars[i], inits[i].eval(
                    lexenv, common));
        }
        return getBody().eval(newlexenv, common);
    }

    public void compileInstruction (final StringBuffer buffer,
                                    final ICgenLexicalEnvironment lexenv,
                                    final ICgenEnvironment common,
                                    final IDestination destination )
    throws CgenerationException {
        final IAST2variable[] vars = getVariables();
        final IAST2expression<CEASTparseException>[] inits = getInitializations();
        IAST2variable[] temp = new IAST2variable[vars.length];
        ICgenLexicalEnvironment templexenv = lexenv;

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

        for (int i = 0; i < vars.length; i++) {
            inits[i].compileInstruction(
                    buffer, templexenv, common,
                    new AssignDestination(temp[i]) );
        }

        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(bodylexenv.compile(vars[i]));
            buffer.append(" = ");
```

```
            buffer.append(bodylexenv.compile(temp[i]));
            buffer.append(";\n");
        }

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

    @Override
    public void findGlobalVariables (
            final Set<IAST2variable> globalvars,
            final ICgenLexicalEnvironment lexenv ) {
        ICgenLexicalEnvironment bodylexenv = lexenv;
        for ( IAST2variable var : getVariables() ) {
            bodylexenv = bodylexenv.extend(var);
        }
        for ( IAST2expression<CEASTparseException> expr : getInitializations() ) {
            expr.findGlobalVariables(globalvars, lexenv);
        }
        getBody().findGlobalVariables(globalvars, bodylexenv);
    }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASToperation.java

```
package fr.upmc.ilp.ilp2.ast;

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

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

public abstract class CEASToperation
  extends CEASTexpression
  implements IAST2operation<CEASTparseException> {

  public CEASToperation (final String operatorName, final int arity) {
    this.operatorName = operatorName;
    this.arity = arity;
  }
  private final String operatorName;
  private final int arity;

  public String getOperatorName () {
      return this.operatorName;
  }
  public int getArity () {
      return this.arity;
  }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTparseException.java

```
package fr.upmc.ilp.ilp2.ast;

/** Les exceptions signalées lors de l'analyse syntaxique. */

public class CEASTparseException
  extends Exception {

  static final long serialVersionUID = +1234567890006000L;

  public CEASTparseException (final Throwable cause) {
    super(cause);
  }

  public CEASTparseException (final String message) {
    super(message);
  }

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTprimitiveInvocation.java


```
package fr.upmc.ilp.ilp2.ast;

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

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.Invokable;
import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2invocation;
import fr.upmc.ilp.ilp2.interfaces.IAST2primitiveInvocation;
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.ilp2.interfaces.IParser;

/** Les invocations de primitives. Elles sont implantées comme des
 * invocations normales à des primitives obtenues à l'aide d'un espace
 * de nom un peu particulier afin d'être toujours disponibles.
 */

public class CEASTprimitiveInvocation
extends CEASTinvocation
implements IAST2primitiveInvocation<CEASTparseException> {

  public CEASTprimitiveInvocation (
          final String primitiveName,
          final IAST2expression<CEASTparseException>[] arguments ) {
      super(new CEASTreference(new CEASTvariable(primitiveName)),
            arguments);
      this.primitiveName = primitiveName;
  }
  private final String primitiveName;

  public String getPrimitiveName() {
      return this.primitiveName;
  }

  public static IAST2invocation<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
    throws CEASTparseException {
      String primitiveName = e.getAttribute("fonction");
      List<IAST2<CEASTparseException>> li = parser.parseList(e.getChildNodes());
      IAST2expression<CEASTparseException>[] arguments =
          li.toArray(CEASTexpression.EMPTY_EXPRESSION_ARRAY);
      return parser.getFactory()
          .newPrimitiveInvocation(primitiveName, arguments);
  }

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

  @Override
  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    final Object fn = common.primitiveLookup(getPrimitiveName());
    if ( fn instanceof Invokable ) {
      Invokable invokable = (Invokable) fn;
      final IAST2expression<CEASTparseException>[] arguments = getArguments();
      final Object[] args = new Object[arguments.length];
      for ( int i = 0 ; i<arguments.length ; i++ ) {
        args[i] = arguments[i].eval(lexenv, common);
      }
      return invokable.invoke(args);
    } else {
      final String msg = "Not a function: " + fn;
      throw new EvaluationException(msg);
    }
  }

  @Override
  public void compileExpression (final StringBuffer buffer,
                                 final ICgenLexicalEnvironment lexenv,
```

```
                                final ICgenEnvironment common,
                                final IDestination destination)
    throws CgenerationException {
        destination.compile(buffer, lexenv, common);
        //getFunction().compileExpression(buffer, lexenv, common);
        buffer.append(common.compilePrimitive(primitiveName));
        buffer.append("(");
        final IAST2expression<CEASTparseException>[] arguments = getArguments();
        for ( int i = 0 ; i<arguments.length-1 ; i++ ) {
            arguments[i].compileExpression(buffer, lexenv, common);
            buffer.append(", ");
        }
        if ( arguments.length> 0 ) {
            arguments[arguments.length-1].compileExpression(buffer, lexenv, common);
        }
        buffer.append(")");
    }

    @Override
    public void findGlobalVariables (final Set<IAST2variable> globalvars,
                                final ICgenLexicalEnvironment lexenv ) {
        for ( IAST2expression<CEASTparseException> arg : getArguments() ) {
            arg.findGlobalVariables(globalvars, lexenv);
        }
    }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTprogram.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp2.interfaces.IAST2Factory;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;
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.ilp2.interfaces.IParser;

/** La classe d'un programme composé de fonctions globales et
 * d'instructions. */

public class CEASTprogram
extends CEAST
implements IAST2program<CEASTparseException> {

    public CEASTprogram (
            final IAST2functionDefinition<CEASTparseException>[] definitions,
            final IAST2instruction<CEASTparseException> body ) {
        this.definition = definitions;
        this.body = body;
        this.globalVariables = new HashSet<>();
    }
    protected IAST2functionDefinition<CEASTparseException>[] definition;
    protected IAST2instruction<CEASTparseException> body;

    private final static void sortDefinitionsAndInstructions (
            final List<IAST2<CEASTparseException>> items,
            final List<IAST2functionDefinition<CEASTparseException>> definitions,
            final List<IAST2instruction<CEASTparseException>> instructions ) {
        for ( IAST2<CEASTparseException> item : items ) {
            if ( item instanceof IAST2functionDefinition<?> ) {
                definitions.add((IAST2functionDefinition<CEASTparseException>) item);
            } else if ( item instanceof IAST2instruction<?> ) {
```



```
                instructions.add((IAST2instruction<CEASTparseException>) item);
            } else {
                final String msg = "Should never occur!";
                assert false : msg;
                throw new RuntimeException(msg);
            }
        }
    }

    public static IAST2program<CEASTparseException> parse (
            final Element e, final IParser<CEASTparseException> parser)
    throws CEASTparseException {
        final IAST2Factory<CEASTparseException> factory = parser.getFactory();
        final List<IAST2<CEASTparseException>> items =
                parser.parseList(e.getChildNodes());
        final List<IAST2functionDefinition<CEASTparseException>> definitions =
                new ArrayList<>();
        final List<IAST2instruction<CEASTparseException>> instructions =
                new ArrayList<>();
        sortDefinitionsAndInstructions(items, definitions, instructions);
        final IAST2instruction<CEASTparseException> body =
                factory.newSequence(instructions);
        final IAST2program<CEASTparseException> program =
                factory.newProgram(
                        definitions.toArray(new CEASTfunctionDefinition[0]),
                        body );
        return program;
    }

    public IAST2instruction<CEASTparseException> getBody () {
        return this.body;
    }
    public IAST2functionDefinition<CEASTparseException>[] getFunctionDefinitions () {
        return this.definition;
    }
    public IAST2variable[] getGlobalVariables () {
        return this.globalVariables.toArray(CEASTvariable.EMPTY_VARIABLE_ARRAY);
    }
    private Set<IAST2variable> globalVariables;
    public String[] getGlobalVariableNames () {
        final Set<String> ss = new HashSet<>();
        for ( IAST2variable gv : globalVariables ) {
            ss.add(gv.getName());
        }
        return ss.toArray(new String[0]);
    }

    /** Recenser toutes les variables globales. */

    public void computeGlobalVariables (final ICgenLexicalEnvironment lexenv) {
        findGlobalVariables(this.globalVariables, lexenv);
    }

    //NOTE: Acces direct aux champs interdit a partir d'ici!

    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common)
    throws EvaluationException {
        IAST2functionDefinition<CEASTparseException>[] definitions = getFunctionDefinitions();
        for ( int i = 0 ; i<definitions.length ; i++ ) {
            definitions[i].eval(lexenv, common);
        }
        return getBody().eval(lexenv, common);
    }

    /** Compiler une instruction en une chaine de caracteres. */

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

    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 \"ilpBasicError.h\"\n");
        buffer.append("\n");
        final IAST2functionDefinition<CEASTparseException>[] definitions =
            getFunctionDefinitions();
        // Declarer les variables globales:
        buffer.append("/* Variables globales: */\n");
        for ( IAST2variable var : getGlobalVariables() ) {
            buffer.append("static ILP_Object ");
            buffer.append(var.getMangledName());
            buffer.append(" = NULL;\n");
            if ( ! common.isPresent(var.getName()) ) {
                common.bindGlobal(var);
            }
        }
        // Emettre le code des fonctions:
        buffer.append("/* Prototypes: */\n");
        for ( IAST2functionDefinition<CEASTparseException> fun : definitions ) {
            fun.compileHeader(buffer, lexenv, common);
        }
        buffer.append("/* Fonctions globales: */\n");
        for ( IAST2functionDefinition<CEASTparseException> fun : definitions ) {
            common.bindGlobal(new CEASTvariable(fun.getFunctionName()));
        }
        for ( IAST2functionDefinition<CEASTparseException> fun : definitions ) {
            fun.compile(buffer, lexenv, common);
        }
        // Emettre les instructions regroupees dans une fonction:
        buffer.append("/* Code hors fonction: */\n");
        IAST2functionDefinition<CEASTparseException> bodyAsFunction =
            new CEASTfunctionDefinition(
                    "ilp_program",
                    CEASTvariable.EMPTY_VARIABLE_ARRAY,
                    getBody() );
        //INUTILE: bodyAsFunction.compile_header(buffer, lexenv, common);
        bodyAsFunction.compile(buffer, lexenv, common);
        buffer.append("\n");
        buffer.append("int main (int argc, char *argv[]) {\n");
        buffer.append("  ILP_print(ilp_program());\n");
        buffer.append("  ILP_newline();\n");
        buffer.append("  return EXIT_SUCCESS;\n");
        buffer.append("}\n\n");
        buffer.append("/* end */\n");
    }

    @Override
    public void findGlobalVariables (
            final Set<IAST2variable> globalvars,
            final ICgenLexicalEnvironment lexenv ) {
        for ( IAST2functionDefinition<CEASTparseException> fun
                : getFunctionDefinitions() ) {
            fun.findGlobalVariables(globalvars, lexenv);
        }
        getBody().findGlobalVariables(globalvars, lexenv);
        // retirer les fonctions
        for ( IAST2functionDefinition<CEASTparseException> fun
                : getFunctionDefinitions() ) {
            globalvars.remove(new CEASTvariable(fun.getFunctionName()));
        }
    }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTreference.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

import org.w3c.dom.Element;

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

public class CEASTreference
extends CEASTexpression
implements IAST2reference<CEASTparseException> {

    public CEASTreference(IAST2variable variable) {
        this.variable = variable;
    }
    private final IAST2variable variable;

    public IAST2variable getVariable() {
        return this.variable;
    }

    public static IAST2reference<CEASTparseException> parse (
            final Element e, final IParser<CEASTparseException> parser) {
        String name = e.getAttribute("nom");
        IAST2variable variable = parser.getFactory().newVariable(name);
        return parser.getFactory().newReference(variable);
    }

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

    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common )
    throws EvaluationException {
        try {
          return lexenv.lookup(getVariable());
        } catch (Exception e) {
          return common.globalLookup(getVariable());
        }
    }

    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) {
            String v = common.compileGlobal(getVariable());
            //if ( common.isPresent(getVariable().getName())) {
            //    // FUTUR Separer les variables globales des fonctions globales
            //    buffer.append("ILP_globalIfInitialized(" + v + ")");
            //} else {
                buffer.append(v);
            //}
        }
        buffer.append(" ");
    }

    /** Une variable est globale si elle n'est ni locale, ni predefinie. */

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

Java/src/fr/upmc/ilp/ilp2/ast/CEASTsequence.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.List;
```

```java
import java.util.Set;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.cgen.VoidDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2sequence;
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.ilp2.interfaces.IParser;

/** Les sequences d'instructions. */

public class CEASTsequence
  extends CEASTinstruction
  implements IAST2sequence<CEASTparseException> {

  public CEASTsequence (final IAST2instruction<CEASTparseException>[] instruction) {
    this.instruction = instruction;
  }
  public CEASTsequence (final List<IAST2instruction<CEASTparseException>> instructions) {
    this(instructions.toArray(CEASTinstruction.EMPTY_INSTRUCTION_ARRAY));
  }
  private final IAST2instruction<CEASTparseException>[] instruction;

  public IAST2instruction<CEASTparseException>[] getInstructions () {
      return this.instruction;
  }
  public int getInstructionsLength () {
      return this.instruction.length;
  }
  public IAST2instruction<CEASTparseException> getInstruction (final int i)
    throws CEASTparseException {
      try {
        return this.instruction[i];
      } catch (Exception exc) {
          throw new CEASTparseException(exc);
      }
  }

  public static IAST2sequence<CEASTparseException> parse(
          final Element e, final IParser<CEASTparseException> parser)
    throws CEASTparseException {
      return parser.parseChildrenAsSequence(e.getChildNodes());
  }

  //NOTE: Acces direct aux champs interdit a partir d'ici!

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
      IAST2instruction<CEASTparseException>[] instructions = getInstructions();
      Object last = Boolean.FALSE;
      for ( int i = 0 ; i < instructions.length ; i++ ) {
          last = instructions[i].eval(lexenv, common);
      }
      return last;
  }

  public void compileInstruction (final StringBuffer buffer,
                                  final ICgenLexicalEnvironment lexenv,
                                  final ICgenEnvironment common,
                                  final IDestination destination)
    throws CgenerationException {
    buffer.append("{\n");
    IAST2instruction<CEASTparseException>[] instructions = getInstructions();
    for ( int i = 0 ; i<instructions.length-1 ; i++ ) {
      instructions[i].compileInstruction(
              buffer, lexenv, common, VoidDestination.create() );
    }
    final IAST2instruction<CEASTparseException> last = instructions[instructions.length-1];
    last.compileInstruction(buffer, lexenv, common, destination);
```



```java
    buffer.append("\n}\n");
  }

  @Override
  public void findGlobalVariables (
          final Set<IAST2variable> globalvars,
          final ICgenLexicalEnvironment lexenv ) {
      for ( IAST2instruction<CEASTparseException> instr : getInstructions() ) {
          instr.findGlobalVariables(globalvars, lexenv);
      }
  }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTstring.java

```java
package fr.upmc.ilp.ilp2.ast;

import org.w3c.dom.Element;

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2string;
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.ilp2.interfaces.IParser;
import fr.upmc.ilp.tool.CStuff;

/** Les constantes chaines de caracteres. */

public class CEASTstring
extends CEASTconstant
implements IAST2string<CEASTparseException> {

  public CEASTstring (String value) {
      super(value, value);
  }

  public static IAST2string<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      String value = e.getTextContent();
      return parser.getFactory().newStringConstant(value);
  }

  //NOTE: Acc?s direct aux champs interdit ? partir d'ici!

  public void compileExpression (final StringBuffer buffer,
                                 final ICgenLexicalEnvironment lexenv,
                                 final ICgenEnvironment common,
                                 final IDestination destination)
    throws CgenerationException {
    String value = (String) getValue();
    destination.compile(buffer, lexenv, common);
    buffer.append(" ILP_String2ILP(\"");
    buffer.append(CStuff.protect(value));
    buffer.append("\") ");
  }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTunaryBlock.java

```java
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
```

```java
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.cgen.AssignDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryBlock;
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.ilp2.interfaces.IParser;

/** Le bloc unaire: interpretation et compilation.
 *
 * Il n'herite pas du bloc local car il n'y a aucun code en commun. On peut
 * toujours remplacer les blocs unaires par des blocs locaux n-aires: c'est
 * a l'analyseur syntaxique de le faire. J'ai maintenu les blocs unaires
 * pour pouvoir continuer a utiliser les tests qui en contiennent.
 *
 * On pourrait faire une nouvelle implantation transformant les blocs unaires
 * en blocs n-aires.
 */

public class CEASTunaryBlock
extends CEASTinstruction
implements IAST2unaryBlock<CEASTparseException> {

  public CEASTunaryBlock (final IAST2variable variable,
                          final IAST2expression<CEASTparseException> initialization,
                          final IAST2instruction<CEASTparseException> body)
  {
    this.variable = variable;
    this.initialization = initialization;
    this.body = body;
  }
  private final IAST2variable variable;
  private final IAST2expression<CEASTparseException> initialization;
  private final IAST2instruction<CEASTparseException> body;

  public IAST2variable getVariable () {
    return this.variable;
  }
  public IAST2expression<CEASTparseException> getInitialization () {
    return this.initialization;
  }
  public IAST2instruction<CEASTparseException> getBody () {
    return this.body;
  }

  public static IAST2unaryBlock<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      final NodeList nl = e.getChildNodes();
      IAST2variable variable;
      try {
          final XPathExpression varPath = xPath.compile("./variable");
          final Element nVar = (Element) varPath.evaluate(e, XPathConstants.NODE);
          variable = parser.getFactory().newVariable(nVar.getAttribute("nom"));
      } catch (XPathExpressionException e1) {
          throw new CEASTparseException(e1);
      }
      IAST2expression<CEASTparseException> initialization = (IAST2expression<CEASTparseException>)
          parser.findThenParseChildAsUnique(nl, "valeur");
      IAST2instruction<CEASTparseException> body = (IAST2instruction<CEASTparseException>)
          parser.findThenParseChildAsSequence(nl, "corps");
      return parser.getFactory()
          .newUnaryBlock(variable, initialization, body);
  }
  private static final XPath xPath = XPathFactory.newInstance().newXPath();

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

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
    ILexicalEnvironment newlexenv =
      lexenv.extend(getVariable(), getInitialization().eval(lexenv, common));
```



```java
    return getBody().eval(newlexenv, common);
  }

  public void compileInstruction (final StringBuffer buffer,
                                  final ICgenLexicalEnvironment lexenv,
                                  final ICgenEnvironment common,
                                  final IDestination destination)
    throws CgenerationException {
    final IAST2variable tmp = CEASTvariable.generateVariable();
    final ICgenLexicalEnvironment lexenv2 = lexenv.extend(tmp);
    final ICgenLexicalEnvironment lexenv3 = lexenv2.extend(getVariable());

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

    buffer.append("{\n");
    getVariable().compileDeclaration(buffer, lexenv3, common);
    buffer.append(lexenv3.compile(getVariable()));
    buffer.append(" = ");
    buffer.append(lexenv2.compile(tmp));
    buffer.append(";\n");
    getBody().compileInstruction(buffer, lexenv3, common, destination);
    buffer.append("}\n}\n");
  }

  @Override
  public void
    findGlobalVariables (final Set<IAST2variable> globalvars,
                         final ICgenLexicalEnvironment lexenv ) {
    getInitialization().findGlobalVariables(globalvars, lexenv);
    final ICgenLexicalEnvironment bodylexenv = lexenv.extend(getVariable());
    getBody().findGlobalVariables(globalvars, bodylexenv);
  }

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTunaryOperation.java

```java
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.cgen.NoDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2unaryOperation;
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.ilp2.interfaces.IParser;

/** Les operations unaires. */

public class CEASTunaryOperation
  extends CEASToperation
  implements IAST2unaryOperation<CEASTparseException> {

  public CEASTunaryOperation (
          final String operatorName,
          final IAST2expression<CEASTparseException> operand) {
    super(operatorName, 1);
    this.operand = operand;
  }
  private final IAST2expression<CEASTparseException> operand;

  public IAST2expression<CEASTparseException> getOperand() {
    return operand;
  }
```

```
    public IAST2expression<CEASTparseException>[] getOperands () {
        CEASTexpression[] result = {
                (CEASTexpression) this.operand,
        };
        return result;
    }

    public static IAST2unaryOperation<CEASTparseException> parse (
            final Element e, final IParser<CEASTparseException> parser)
      throws CEASTparseException {
        String operatorName = e.getAttribute("operateur");
        final NodeList nl = e.getChildNodes();
        IAST2expression<CEASTparseException> operand =
            (IAST2expression<CEASTparseException>)
            parser.findThenParseChildAsUnique(nl, "operande");
        return parser.getFactory().newUnaryOperation(operatorName, operand);
    }

    //NOTE: Acces direct aux champs interdit a partir d'ici!

    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common)
      throws EvaluationException {
      return common.applyOperator(getOperatorName(),
                                  getOperand().eval(lexenv, common));
    }

    public void compileExpression (final StringBuffer buffer,
                                   final ICgenLexicalEnvironment lexenv,
                                   final ICgenEnvironment common,
                                   final IDestination destination)
      throws CgenerationException {
      destination.compile(buffer, lexenv, common);
      buffer.append(" ");
      buffer.append(common.compileOperator1(getOperatorName()));
      buffer.append("(");
      getOperand().compileExpression(
              buffer, lexenv, common, NoDestination.create() );
      buffer.append(") ");
    }

    @Override
    public void findGlobalVariables (final Set<IAST2variable> globalvars,
                                     final ICgenLexicalEnvironment lexenv ) {
        getOperand().findGlobalVariables(globalvars, lexenv);
    }

}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTvariable.java

```
package fr.upmc.ilp.ilp2.ast;

import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;

/** Les variables: interprétation et compilation. */

public class CEASTvariable
implements IAST2variable {

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

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

  @Override
  public boolean equals (Object other) {
      if ( other instanceof IAST2variable ) {
          IAST2variable otherV = (IAST2variable) other;
          return otherV.getName().equals(getName());
      } else {
          return false;
      }
  }

  @Override
  public int hashCode () {
      return getName().hashCode();
  }

  /** Utile pour les conversions de Liste de variable vers tableau
   * de variables */

  public final static IAST2variable[] EMPTY_VARIABLE_ARRAY =
      new IAST2variable[]{};

  /** Génération de variables temporaires. */

  public synchronized static IAST2variable generateVariable () {
    counter++;
    return new CEASTvariable("ilpTMP_" + counter);
  }
  private static int counter = 100;

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

  public void compileDeclaration (final StringBuffer buffer,
                                  final ICgenLexicalEnvironment lexenv,
                                  final ICgenEnvironment common ) {
      buffer.append(" ILP_Object ");
      buffer.append(getMangledName());
      buffer.append(";\n");
  }
}
```

Java/src/fr/upmc/ilp/ilp2/ast/CEASTwhile.java

```
package fr.upmc.ilp.ilp2.ast;

import java.util.Set;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.cgen.VoidDestination;
import fr.upmc.ilp.ilp2.interfaces.IAST2expression;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
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.ICommon;
import fr.upmc.ilp.ilp2.interfaces.IDestination;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IParser;

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

public class CEASTwhile
  extends CEASTinstruction
  implements IAST2while<CEASTparseException> {

  public CEASTwhile (
          final IAST2expression<CEASTparseException> condition,
          final IAST2instruction<CEASTparseException> body) {
    this.condition = condition;
    this.body = body;
  }
  private final IAST2expression<CEASTparseException> condition;
  private final IAST2instruction<CEASTparseException> body;
```

```
    public IAST2expression<CEASTparseException> getCondition () {
42      return condition;
    }
    public IAST2instruction<CEASTparseException> getBody () {
        return body;
    }
47
    public static IAST2while<CEASTparseException> parse (
            final Element e, final IParser<CEASTparseException> parser)
      throws CEASTparseException {
      final NodeList nl = e.getChildNodes();
52    IAST2expression<CEASTparseException> condition =
        (IAST2expression<CEASTparseException>)
          parser.findThenParseChildAsUnique(nl, "condition");
      IAST2instruction<CEASTparseException> body =
        (IAST2instruction<CEASTparseException>)
57        parser.findThenParseChildAsSequence(nl, "corps");
      return parser.getFactory().newWhile(condition, body);
    }

    //NOTE: Acces direct aux champs interdit a partir d'ici!
62
    public Object eval (final ILexicalEnvironment lexenv,
                        final ICommon common)
      throws EvaluationException {
      while ( true ) {
67      Object bool = getCondition().eval(lexenv, common);
        if ( Boolean.FALSE == bool ) {
          break;
        }
        getBody().eval(lexenv, common);
72    }
      // Bogue ici précédemment: vue par Rafael.Cerioli@etu.upmc.fr
      return Boolean.FALSE;
    }

77  public void compileInstruction (final StringBuffer buffer,
                                    final ICgenLexicalEnvironment lexenv,
                                    final ICgenEnvironment common,
                                    final IDestination destination)
      throws CgenerationException {
82    buffer.append(" while ( ILP_isEquivalentToTrue( ");
      getCondition().compileExpression(buffer, lexenv, common);
      buffer.append(") ) { ");
      getBody().compileInstruction(
              buffer, lexenv, common, VoidDestination.create() );
87    buffer.append("}\n");
      CEASTinstruction.voidInstruction()
        .compileInstruction(buffer, lexenv, common, destination);
    }

92  @Override
    public void findGlobalVariables (final Set<IAST2variable> globalvars,
                                     final ICgenLexicalEnvironment lexenv ) {
        getCondition().findGlobalVariables(globalvars, lexenv);
        getBody().findGlobalVariables(globalvars, lexenv);
97  }
}
```

Java/src/fr/upmc/ilp/ilp2/runtime/BasicEmptyEnvironment.java

```
1  package fr.upmc.ilp.ilp2.runtime;

   import fr.upmc.ilp.annotation.OrNull;
   import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
   import fr.upmc.ilp.ilp2.interfaces.IEnvironment;
6
   /** Une simple liste vide chainee de variables (de type V).
    *
    * Attention: pas moyen de parametrer le type de l'exception
    * a signaler en cas d'environnement vide. */
11
   public class BasicEmptyEnvironment<V extends IAST2variable>
   implements IEnvironment<V> {

       protected BasicEmptyEnvironment () {}
```



```
16
       public BasicEmptyEnvironment<V> getNext () {
           final String msg = "Basic empty environment!";
           throw new RuntimeException(msg);
       }
21
       public V getVariable () {
           final String msg = "No variable in an empty environment!";
           throw new RuntimeException(msg);
       }
26
       public boolean isEmpty () {
           return true;
       }

31     public boolean isPresent (V variable) {
           return false;
       }

           public @OrNull IEnvironment<V> shrink(V variable) {
36             return null;
           }

   }
```

Java/src/fr/upmc/ilp/ilp2/runtime/BasicEnvironment.java

```
1  package fr.upmc.ilp.ilp2.runtime;

   import fr.upmc.ilp.annotation.OrNull;
   import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
   import fr.upmc.ilp.ilp2.interfaces.IEnvironment;
6
   /** Une simple liste non vide chainee de variables (de type V). */

   public class BasicEnvironment<V extends IAST2variable>
   implements IEnvironment<V> {
11
       public BasicEnvironment (V variable, IEnvironment<V> next) {
           this.variable = variable;
           this.next = next;
       }
16     protected final V variable;
       protected final IEnvironment<V> next;

       public IEnvironment<V> getNext () {
           return this.next;
21     }

       public V getVariable () {
           return this.variable;
       }
26
       public boolean isEmpty () {
           return false;
       }

31         public @OrNull IEnvironment<V> shrink (V variable) {
           if ( getVariable().equals(variable) ) {
               return this;
           } else {
               return getNext().shrink(variable);
36         }
           }

       public boolean isPresent (V variable) {
           return null != this.shrink(variable);
41     }
   }
```

Java/src/fr/upmc/ilp/ilp2/runtime/CommonPlus.java

```
   package fr.upmc.ilp.ilp2.runtime;
```

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

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICommon;

/** Environnement global d'interpretation d'ILP2. */

public class CommonPlus
extends fr.upmc.ilp.ilp1.runtime.CommonPlus
implements ICommon {

    public CommonPlus () {
        this(new HashMap<String,Object>(),
             new HashMap<String,Object>());
    }
    public CommonPlus (Map<String, Object> primitiveMap,
                       Map<String, Object> globalMap ) {
        this.primitiveMap = primitiveMap;
        this.globalMap = globalMap;
    }

    /** Ces tables associent des chaines et des objets. */
    protected final Map<String, Object> primitiveMap;
    protected final Map<String, Object> globalMap;

    /** Renvoyer la valeur d'une primitive. */

    public Object primitiveLookup (final String primitiveName)
         throws EvaluationException {
         final Object value = primitiveMap.get(primitiveName);
         if ( value != null ) {
              return value;
         } else {
              final String msg = "No such entity: " + primitiveName;
              throw new EvaluationException(msg);
         }
    }

    /** Etablir la valeur d'une primitive. */

    public void bindPrimitive (final String primitiveName,
                               final Object primitive)
         throws EvaluationException {
         final Object value = primitiveMap.get(primitiveName);
         if ( value == null ) {
              primitiveMap.put(primitiveName, primitive);
         } else {
              final String msg = "Already defined primitive: " + primitiveName;
              throw new EvaluationException(msg);
         }
    }

    /** Renvoyer la valeur d'une variable globale. */

    public Object globalLookup (final IAST2variable variable)
         throws EvaluationException {
         final String variableName = variable.getName();
         final Object value = globalMap.get(variableName);
         if ( value != null ) {
              return value;
         } else if ( globalMap.containsKey(variableName) ) {
              final String msg = "uninitialized global: " + variable.getName();
              throw new EvaluationException(msg);
         } else {
              final String msg = "No such global: " + variable.getName();
              throw new EvaluationException(msg);
         }
    }

    /** Determiner si une variable globale est presente. */

    public boolean isPresent (final String variableName) {
         return globalMap.containsKey(variableName);
    }

    /** Etablir ou modifier la valeur d'une variable globale. Et si
```



```
     * c'était une fonction globale ? */

    public void updateGlobal (final String globalVariableName,
                              final Object value) {
        globalMap.put(globalVariableName, value);
    }
}
```

Java/src/fr/upmc/ilp/ilp2/runtime/ConstantsStuff.java

```
package fr.upmc.ilp.ilp2.runtime;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.ICommon;

/** Cette classe permet d'etendre un environnement lexical avec une definition
 * de la constante Pi.
 */

public class ConstantsStuff {

    public ConstantsStuff () {}

    /** Etendre un environnement lexical pour y installer la constante Pi. */

    public void extendWithPredefinedConstants (final ICommon common)
        throws EvaluationException {
      common.updateGlobal(
              "pi",
              new Double(3.14159265358979323846264 3) );
    }
}
```

Java/src/fr/upmc/ilp/ilp2/runtime/InvokableImpl.java

```
package fr.upmc.ilp.ilp2.runtime;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.Invokable;

/** Une classe abstraite de fonction qui peut servir de base ? des
 * implantations particuli?res. Il suffit, si la fonction a moins de
 * quatre arguments, de d?finir la m?thode invoke appropri?e.
 * @see fr.upmc.ilp.ilp2.runtime.PrintStuff
 */

public abstract class InvokableImpl
  implements Invokable {

  private static final String WRONG_ARITY =
    "Wrong arity";

  /** Une fonction invoqu?e avec un nombre quelconque d'arguments. Les
   * petites arit?s sont renvoy?es sur les m?thodes appropri?es. */

  public Object invoke (final Object[] arguments)
    throws EvaluationException {
    switch (arguments.length) {
    case 0: return this.invoke();
    case 1: return this.invoke(arguments[0]);
    case 2: return this.invoke(arguments[0], arguments[1]);
    case 3: return this.invoke(arguments[0], arguments[1], arguments[2]);
    default: throw new EvaluationException(WRONG_ARITY);
    }
  }

  /** Invocation d'une fonction z?ro-aire (ou niladique) */

  public Object invoke ()
    throws EvaluationException {
    throw new EvaluationException(WRONG_ARITY);
  }

  /** Invocation d'une fonction unaire (ou monadique) */
```

```
  public Object invoke (final Object argument1)
    throws EvaluationException {
    throw new EvaluationException(WRONG_ARITY);
  }

  /** Invocation d'une fonction binaire (ou dyadique) */

  public Object invoke (final Object argument1,
                        final Object argument2)
    throws EvaluationException {
    throw new EvaluationException(WRONG_ARITY);
  }

  /** Invocation d'une fonction ternaire */

  public Object invoke (final Object argument1,
                        final Object argument2,
                        final Object argument3)
    throws EvaluationException {
    throw new EvaluationException(WRONG_ARITY);
  }

}
```

Java/src/fr/upmc/ilp/ilp2/runtime/LexicalEnvironment.java

```
package fr.upmc.ilp.ilp2.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;

/** Cette implantation d'environnement est tres naive: c'est une
 * simple liste chainee (mais comme nous n'avons pour l'instant que
 * des blocs unaires cela suffit!). En revanche, comme la classe
 * correspondant a l'environnement vide depend de celui-ci, on la
 * definit comme classe emboitee.
 */

public class LexicalEnvironment
extends BasicEnvironment<IAST2variable>
implements ILexicalEnvironment {

  public LexicalEnvironment (final IAST2variable variable,
                             final Object value,
                             final ILexicalEnvironment next )
  {
      super(variable, next);
      this.value = value;
  }
  protected Object value;

  @Override
  public ILexicalEnvironment getNext () {
      return (ILexicalEnvironment) super.getNext();
  }
  @Override
  public ILexicalEnvironment shrink (final IAST2variable v) {
      return (ILexicalEnvironment) super.shrink(v);
  }

  @Override
  public boolean isPresent (final IAST2variable variable) {
      if ( getVariable().equals(variable) ) {
          return true;
      } else {
          return getNext().isPresent(variable);
      }
  }

  public Object lookup (IAST2variable variable)
  throws EvaluationException {
      if ( getVariable().equals(variable) ) {
          return this.value;
      } else {
          return getNext().lookup(variable);
      }
  }

  public void update (final IAST2variable variable,
                      final Object value)
    throws EvaluationException {
    if ( getVariable().equals(variable) ) {
        this.value = value;
    } else {
        getNext().update(variable, value);
    }
  }

  /** On peut etendre tout environnement. */
  public ILexicalEnvironment extend (final IAST2variable 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:
    protected 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;
    }

    /** Encore une fois covariant! */
    @Override
    public EmptyLexicalEnvironment getNext() {
        final String msg = "Empty environment!";
        throw new RuntimeException(msg);
    }
    @Override
    public @OrNull ILexicalEnvironment shrink(IAST2variable v) {
        return null;
    }

    public Object lookup (IAST2variable variable) {
        String msg = "Variable sans valeur: " + getVariable().getName();
        throw new RuntimeException(msg);
    }

    /** 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.
     */
    public ILexicalEnvironment extend (final IAST2variable variable,
                                       final Object value) {
      return new LexicalEnvironment(variable, value, this);
    }
  }

}
```

Java/src/fr/upmc/ilp/ilp2/runtime/PrintStuff.java

```
package fr.upmc.ilp.ilp2.runtime;

import java.io.IOException;
import java.io.StringWriter;
import java.io.Writer;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.interfaces.ICommon;

/** 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(new StringWriter());
    }
    public PrintStuff (Writer writer) {
        this.output = writer;
    }
    private Writer output;

    /** Renvoyer les caractères imprimés et remettre à vide le tampon
     * d'impression. */

    public synchronized String getPrintedOutput () {
        final String result = output.toString();
        return result;
    }

    /** Cette classe implante la fonction print() qui permet d'imprimer
     * une valeur. */

  private class PrintPrimitive extends InvokableImpl {
    private PrintPrimitive () {}
    // La fonction print() est unaire:
    @Override
    public Object invoke (Object value) {
      try {
        output.append(value.toString());
    } catch (IOException e) {}
      return Boolean.FALSE;
    }
  }

  /** Cette classe implante la fonction newline() qui permet de passer
   * à la ligne. */

  private class NewlinePrimitive extends InvokableImpl {
    private NewlinePrimitive () {}
    // La fonction newline() est zéro-aire:
    @Override
    public Object invoke () {
        try {
            output.append("\n");
        } catch (IOException e) {}
        return Boolean.FALSE;
    }
  }

  public void extendWithPrintPrimitives (ICommon common)
      throws EvaluationException {
    common.bindPrimitive("print", new PrintPrimitive());
    common.bindPrimitive("newline", new NewlinePrimitive());
  }

}
```

Java/src/fr/upmc/ilp/ilp2/runtime/UserFunction.java

```
package fr.upmc.ilp.ilp2.runtime;
```



```
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.ICommon;
import fr.upmc.ilp.ilp2.interfaces.ILexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IUserFunction;

/* La classe des fonctions anonymes définies par l'utilisateur. */

public class UserFunction
implements IUserFunction {

  public UserFunction (final IAST2variable[] variable,
                       final IAST2instruction<CEASTparseException> body,
                       final ILexicalEnvironment lexenv) {
    this.variable = variable;
    this.body = body;
    this.lexenv = lexenv;
  }
  protected final IAST2variable[] variable;
  protected final IAST2instruction<CEASTparseException> body;
  protected final ILexicalEnvironment lexenv;

  public IAST2variable[] getVariables () {
      return this.variable;
  }
  public IAST2instruction<CEASTparseException> getBody () {
      return this.body;
  }
  public ILexicalEnvironment getEnvironment () {
      return this.lexenv;
  }

  // Interdit d'acceder directement les champs a partir d'ici!

  public Object invoke (final Object[] arguments,
                        final ICommon common)
    throws EvaluationException {
      IAST2variable[] variables = getVariables();
      if ( variables.length != arguments.length ) {
          final String msg = "Wrong arity";
          throw new EvaluationException(msg);
      }
      return unsafeInvoke(arguments, common);
  }

  protected Object unsafeInvoke (final Object[] arguments,
                                 final ICommon common)
    throws EvaluationException {
      IAST2variable[] variables = getVariables();
      ILexicalEnvironment lexenv = getEnvironment();
      for ( int i = 0 ; i<variables.length ; i++ ) {
          lexenv = lexenv.extend(variables[i], arguments[i]);
      }
      return getBody().eval(lexenv, common);
  }
}
```

Java/src/fr/upmc/ilp/ilp2/runtime/UserGlobalFunction.java

```
package fr.upmc.ilp.ilp2.runtime;

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.ICommon;

/* La classe des fonctions globales definies par l'utilisateur. */

public class UserGlobalFunction
extends UserFunction {

  public UserGlobalFunction (final String name,
                             final IAST2variable[] variable,
```

```
                            final IAST2instruction<CEASTparseException> body) {
        super(variable, body, LexicalEnvironment.EmptyLexicalEnvironment.create());
        this.name = name;
    }
    protected final String name;

    @Override
    public Object invoke (final Object[] arguments,
                          final ICommon common)
      throws EvaluationException {
        IAST2variable[] variables = getVariables();
        if ( variables.length != arguments.length ) {
            final String msg = "Wrong arity for function:" + name;
            throw new EvaluationException(msg);
        }
        return unsafeInvoke(arguments, common);
    }
}
```

Java/src/fr/upmc/ilp/ilp2/cgen/AssignDestination.java

```
package fr.upmc.ilp.ilp2.cgen;

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.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.IDestination;

/** Destination indiquant dans quelle variable affecter un r?sultat. */
public class AssignDestination
implements IDestination {

    private final IAST2variable variable;

    public AssignDestination (final IAST2variable variable) {
        this.variable = variable;
    }

    /** Préfixe le résultat avec "variable = " pour indiquer une
     * affectation.
     * @throws CgenerationException */
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common )
        throws CgenerationException {
        if ( lexenv.isPresent(variable) ) {
            buffer.append(lexenv.compile(variable));
        } else {
            buffer.append(common.compileGlobal(variable));
        }
        buffer.append(" = ");
    }
}
```

Java/src/fr/upmc/ilp/ilp2/cgen/CgenEnvironment.java

```
package fr.upmc.ilp.ilp2.cgen;

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

import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.tool.CStuff;

/** La representation de l'environnement des operateurs predefinis. Il
 * definit comment les compiler. C'est un peu l'analogue de
 * runtime/Common pour le paquetage cgen.
 *
 *  NOTE: on abandonne l'usage de donnees statiques.
 */
```



```
public class CgenEnvironment
  implements ICgenEnvironment {

  protected final Map<String, String> mapOp1 = new HashMap<>();
  protected final Map<String, String> mapOp2 = new HashMap<>();
  protected final Map<String, String> mapPrimitives = new HashMap<>();
  protected final Map<String, String> mapGlobals = new HashMap<>();

  public CgenEnvironment () {
      // Binaires:
    mapOp2.put("+", "Plus");
    mapOp2.put("-", "Minus");
    mapOp2.put("*", "Times");
    mapOp2.put("/", "Divide");
    mapOp2.put("%", "Modulo");
    mapOp2.put("<", "LessThan");
    mapOp2.put("<=", "LessThanOrEqual");
    mapOp2.put("==", "Equal");
    mapOp2.put(">=", "GreaterThanOrEqual");
    mapOp2.put(">", "GreaterThan");
    mapOp2.put("!=", "NotEqual");
    // Unaires:
    mapOp1.put("-", "Opposite");
    mapOp1.put("!", "Not");
    // Primitives
    mapPrimitives.put("print", "print");
    mapPrimitives.put("newline", "newline");
    // Predefined globals
    mapGlobals.put("pi", "ILP_PI");
  }

  /** Comment convertir un operateur unaire en C. */

  public String compileOperator1 (final String opName)
    throws CgenerationException {
    return compileOperator(opName, mapOp1);
  }

  /** Comment convertir un operateur binaire en C. */

  public String compileOperator2 (final String opName)
    throws CgenerationException {
    return compileOperator(opName, mapOp2);
  }

  /** Comment convertir une primitive en C. */

  public String compilePrimitive (final String opName)
    throws CgenerationException {
    return compileOperator(opName, mapPrimitives);
  }

  /** Methode interne pour trouver le nom en C d'un operateur.
   *
   * @throws CgenerationException si le nom est inconnu.
   */

  private String compileOperator (final String opName, Map<String, String> map)
    throws CgenerationException {
    final String cName = map.get(opName);
    if ( cName != null ) {
      return "ILP_" + cName;
    } else {
      final String msg = "No such entity: " + opName;
      throw new CgenerationException(msg);
    }
  }

  /** Compiler une variable globale. */

  public String compileGlobal (final IAST2variable variable)
    throws CgenerationException {
    final String globalName = variable.getName();
    final String cName = mapGlobals.get(globalName);
    if ( cName != null ) {
        return cName;
    } else {
        return variable.getMangledName();
    }
```

```
    }

    /** Enregistrer une nouvelle variable globale. */

    public void bindGlobal (final IAST2variable var) {
        mapGlobals.put(var.getName(), var.getMangledName());
    }

    @Deprecated
    public void bindGlobal (final String variableName) {
        mapGlobals.put(variableName, CStuff.mangle(variableName));
    }

    /** Enregistrer une nouvelle primitive. */

    public void bindPrimitive (final String primitiveName) {
        this.mapPrimitives.put(primitiveName, primitiveName);
    }
    public void bindPrimitive (final String primitiveName, final String cName) {
        this.mapPrimitives.put(primitiveName, cName);
    }

    /** La variable est-elle presente dans l'environnement global ? */

    public boolean isPresent (final String variableName) {
        return mapGlobals.containsKey(variableName);
    }

}
```

Java/src/fr/upmc/ilp/ilp2/cgen/CgenLexicalEnvironment.java

```
package fr.upmc.ilp.ilp2.cgen;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.cgen.CgenerationException;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.runtime.BasicEmptyEnvironment;
import fr.upmc.ilp.ilp2.runtime.BasicEnvironment;

/** La représentation des environnements lexicaux de compilation vers
 * C. C'est l'analogue de runtime/LexicalEnvironment pour le
 * paquetage cgen. */

public class CgenLexicalEnvironment
extends BasicEnvironment<IAST2variable>
implements ICgenLexicalEnvironment {

    public CgenLexicalEnvironment (final IAST2variable variable,
                                   final ICgenLexicalEnvironment next) {
        super(variable, next);
        this.cname = variable.getMangledName();
    }
    protected String cname;

    public CgenLexicalEnvironment (final IAST2variable variable,
                                   final String cname,
                                   final ICgenLexicalEnvironment next) {
        super(variable, next);
        this.cname = cname;
    }

    @Override
    public ICgenLexicalEnvironment getNext () {
        return (ICgenLexicalEnvironment) super.getNext();
    }

    @Override
    public @OrNull ICgenLexicalEnvironment shrink (IAST2variable variable) {
        if ( getVariable().equals(variable) ) {
            return this;
        } else {
            return getNext().shrink(variable);
        }
    }
    public String compile (final IAST2variable variable)
```



```
      throws CgenerationException {
      if ( getVariable().equals(variable) ) {
        return this.cname;
      } else {
        return getNext().compile(variable);
      }
    }

    public ICgenLexicalEnvironment extend (final IAST2variable variable) {
      return new CgenLexicalEnvironment(variable, this);
    }
    public ICgenLexicalEnvironment extend(final IAST2variable variable,
                                          final String cname) {
        return new CgenLexicalEnvironment(variable, cname, this);
    }

    /** ================================================== */
    public static class CgenEmptyLexicalEnvironment
    extends BasicEmptyEnvironment<IAST2variable>
    implements ICgenLexicalEnvironment {

      // La technique du singleton:
      protected CgenEmptyLexicalEnvironment () {}
      private static final CgenEmptyLexicalEnvironment
        THE_EMPTY_LEXICAL_ENVIRONMENT;
      static {
        THE_EMPTY_LEXICAL_ENVIRONMENT = new CgenEmptyLexicalEnvironment();
      }

      public static ICgenLexicalEnvironment create () {
        return CgenEmptyLexicalEnvironment.THE_EMPTY_LEXICAL_ENVIRONMENT;
      }

      public String compile (final IAST2variable variable)
        throws CgenerationException {
        final String msg = "Variable inaccessible: "
          + variable.getName();
        throw new CgenerationException(msg);
      }

      public ICgenLexicalEnvironment extend (final IAST2variable variable) {
        return new CgenLexicalEnvironment(variable, this);
      }
      public ICgenLexicalEnvironment extend(final IAST2variable variable,
                                            final String cname) {
          return new CgenLexicalEnvironment(variable, cname, this);
      }

      /** Aucune variable n'est présente dans l'environnement vide. */

      @Override
      public boolean isPresent (IAST2variable variable) {
        return false;
      }

      @Override
    public @OrNull CgenEmptyLexicalEnvironment shrink(IAST2variable variable) {
          return null;
      }
      @Override
      public CgenEmptyLexicalEnvironment getNext () {
          final String msg = "Really empty environment!";
          throw new RuntimeException(msg);
      }

      @Override
      public IAST2variable getVariable () {
          final String msg = "Empty environment!";
          throw new RuntimeException(msg);
      }
      @Override
      public boolean isEmpty () {
          return true;
      }
    }

}
```

Java/src/fr/upmc/ilp/ilp2/cgen/NoDestination.java

```
package fr.upmc.ilp.ilp2.cgen;

import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;

/** Destination indiquant qu'un r?sultat est inutile. */

public class NoDestination
implements IDestination {

    private static final NoDestination NO_DESTINATION =
        new NoDestination();

    private NoDestination() {}

    // Singleton
    public static NoDestination create () {
        return NO_DESTINATION;
    }

    /** Ne pr?fixe rien devant le r?sultat. Cette destination est
     * utilis?e pour compiler la d?finition des fonctions globales. */

    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common ) {
        // rien du tout!
    }
}
```

Java/src/fr/upmc/ilp/ilp2/cgen/ReturnDestination.java

```
package fr.upmc.ilp.ilp2.cgen;

import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;

/** Destination indiquant le r?sultat d'une fonction. */

public class ReturnDestination
implements IDestination {

    private static final ReturnDestination RETURN_DESTINATION =
        new ReturnDestination();

    private ReturnDestination() {}

    // Singleton
    public static ReturnDestination create () {
        return RETURN_DESTINATION;
    }

    /** Pr?fixe le r?sultat avec "return" pour indiquer son int?r?t. */
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common ) {
        buffer.append("return ");
    }
}
```

Java/src/fr/upmc/ilp/ilp2/cgen/VoidDestination.java

```
package fr.upmc.ilp.ilp2.cgen;

import fr.upmc.ilp.ilp2.interfaces.ICgenEnvironment;
import fr.upmc.ilp.ilp2.interfaces.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp2.interfaces.IDestination;

/** Destination indiquant qu'un r?sultat est inutile. */
```



```
public class VoidDestination
implements IDestination {

    private static final VoidDestination VOID_DESTINATION =
        new VoidDestination();

    private VoidDestination() {}

    // Singleton
    public static VoidDestination create() {
        return VOID_DESTINATION;
    }

    /** Pr?fixe le r?sultat avec (void) pour indiquer son inint?r?t. */
    public void compile (final StringBuffer buffer,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common) {
        buffer.append("(void) ");
    }
}
```

Java/src/fr/upmc/ilp/ilp3/package.html

```
<!-- $Id: package.html 735 2008-09-26 16:38:19Z queinnec $ -->
<body><p>

Ce paquetage contient l'interprÃ¨te, le compilateur et la bibliothÃ¨que
d'exÃ©cution de l'interprÃ¨te d'ILP3. ILP3 correspond au langage ILP2
plus le traitement d'exceptions (try-catch-finally et throw).

</p></body>
```

Java/src/fr/upmc/ilp/ilp3/CEASTFactory.java

```
package fr.upmc.ilp.ilp3;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;

public class CEASTFactory
extends fr.upmc.ilp.ilp2.ast.CEASTFactory
implements IAST3Factory<CEASTparseException> {

    public CEASTFactory () {
        super();
    }

    @Override
    public IAST2program<CEASTparseException> newProgram(
            IAST2functionDefinition<CEASTparseException>[] functions,
            IAST2instruction<CEASTparseException> instruction) {
        return new CEASTprogram(functions, instruction);
    }

    public IAST3try<CEASTparseException> newTry(
            IAST2instruction<CEASTparseException> body,
            IAST2variable caughtExceptionVariable,
            IAST2instruction<CEASTparseException> catcher,
            IAST2instruction<CEASTparseException> finallyer) {
        return new CEASTry(body, caughtExceptionVariable, catcher, finallyer);
    }
}
```

Java/src/fr/upmc/ilp/ilp3/CEASTParser.java

```
package fr.upmc.ilp.ilp3;

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

```
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.interfaces.IAST2;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;

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

public class CEASTParser extends fr.upmc.ilp.ilp2.ast.CEASTParser
implements IParser<CEASTparseException> {

    public CEASTParser (IAST3Factory<CEASTparseException> factory) {
        super(factory);
    }

    @Override
    public IAST3Factory<CEASTparseException> getFactory () {
        return (IAST3Factory<CEASTparseException>) super.getFactory();
    }

    @Override
    public IAST2program<CEASTparseException> parse (final Document d)
    throws CEASTparseException {
        final Element e = d.getDocumentElement();
        return CEASTprogram.parse(e, this);
    }

    @Override
    public IAST2<CEASTparseException> parse (final Node n)
    throws CEASTparseException {
    switch ( n.getNodeType() ) {

    case Node.ELEMENT_NODE: {
      final Element e = (Element) n;
      final String name = e.getTagName();

      if ( "try".equals(name) ) {
          return CEASTry.parse(e, this);
      } else {
          return super.parse(n);
      }
    }

    default: {
        return super.parse(n);
    }
    }
  }
}
```

Java/src/fr/upmc/ilp/ilp3/CEASTprogram.java

```
package fr.upmc.ilp.ilp3;

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.CEASTfunctionDefinition;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.ast.CEASTvariable;
import fr.upmc.ilp.ilp2.interfaces.IAST2functionDefinition;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2program;
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.ilp2.interfaces.IParser;

/** La classe d'un programme composé de fonctions globales et
 * d'instructions. */

public class CEASTprogram
extends fr.upmc.ilp.ilp2.ast.CEASTprogram {

    public CEASTprogram (
            final IAST2functionDefinition<CEASTparseException>[] definitions,
```



```
            final IAST2instruction<CEASTparseException> body ) {
        super(definitions, body);
    }

    /** Le constructeur analysant syntaxiquement un DOM. */

    public static IAST2program<CEASTparseException> parse (
            final Element e, final IParser<CEASTparseException> parser)
    throws CEASTparseException {
        return fr.upmc.ilp.ilp2.ast.CEASTprogram.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 {
    IAST2functionDefinition<CEASTparseException>[] definitions =
        getFunctionDefinitions();
    for ( int i = 0 ; i<definitions.length ; i++ ) {
        definitions[i].eval(lexenv, common);
    }
    Object result = null;
    try {
        result = getBody().eval(lexenv, common);
    } catch ( ThrownException exc ) {
        // Deballer la valeur si c'est une valeur obtenue par
        // echappement. Cela revient a admettre qu'un programme P est
        // equivalent a try { P } catch (e) { e };
        result = exc.getThrownValue();
    }
    return result;
}

@Override
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");
    final IAST2functionDefinition<CEASTparseException>[] definitions =
            getFunctionDefinitions();
    // Declarer les variables globales:
    buffer.append("/* Variables globales: */\n");
    for ( IAST2variable var : getGlobalVariables() ) {
        buffer.append("static ILP_Object ");
        buffer.append(var.getMangledName());
        buffer.append(" = NULL;\n");
        if ( ! common.isPresent(var.getName()) ) {
            common.bindGlobal(var);
        }
    }
    // Emettre le code des fonctions:
    buffer.append("/* Prototypes: */\n");
    for ( IAST2functionDefinition<CEASTparseException> fun : definitions ) {
        fun.compileHeader(buffer, lexenv, common);
    }
    buffer.append("/* Fonctions globales: */\n");
    for ( IAST2functionDefinition<CEASTparseException> fun : definitions ) {
        common.bindGlobal(new CEASTvariable(fun.getFunctionName()));
    }
    for ( IAST2functionDefinition<CEASTparseException> fun : definitions ) {
        fun.compile(buffer, lexenv, common);
    }
    // Emettre les instructions regroupees dans une fonction:
    buffer.append("/* Code hors fonction: */\n");
    IAST2functionDefinition<CEASTparseException> bodyAsFunction =
            new CEASTfunctionDefinition(
                    "ilp_program",
                    CEASTvariable.EMPTY_VARIABLE_ARRAY,
                    getBody() );
    //INUTILE: bodyAsFunction.compile_header(buffer, lexenv, common);
    bodyAsFunction.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("/* fin */\n");
    }
}
```

Java/src/fr/upmc/ilp/ilp3/CEASTtry.java

```
package fr.upmc.ilp.ilp3;

import java.util.Set;

import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathFactory;

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

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.ast.CEASTinstruction;
import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.cgen.VoidDestination;
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.ilp3.IParser;

public class CEASTtry extends CEASTinstruction
implements IAST3try<CEASTparseException> {

  public CEASTtry (final IAST2instruction<CEASTparseException> body,
                   final IAST2variable caughtExceptionVariable,
                   final IAST2instruction<CEASTparseException> catcher,
                   final IAST2instruction<CEASTparseException> finallyer) {
    this.body = body;
    // S'il n'y a pas de rattrapeur, il n'y a pas de variable caughtException:
    this.caughtExceptionVariable = caughtExceptionVariable;
    // et le rattrapeur sera null:
    this.catcher = catcher;
    // Il y a toujours un finaliseur (eventuellement vide):
    this.finallyer = finallyer;
  }
  private IAST2instruction<CEASTparseException> body;
  private @OrNull IAST2variable    caughtExceptionVariable;
  private @OrNull IAST2instruction<CEASTparseException> catcher;
  private @OrNull IAST2instruction<CEASTparseException> finallyer;

  // Les accesseurs

  public IAST2instruction<CEASTparseException> getBody () {
      return this.body;
  }
  public @OrNull IAST2variable getCaughtExceptionVariable () {
      return this.caughtExceptionVariable;
  }
```



```
  public @OrNull IAST2instruction<CEASTparseException> getCatcher () {
      return this.catcher;
  }
  public @OrNull IAST2instruction<CEASTparseException> getFinallyer () {
      return this.finallyer;
  }

  public static IAST3try<CEASTparseException> parse (
          final Element e, final IParser<CEASTparseException> parser)
  throws CEASTparseException {
      try {
          final XPathExpression bodyPath = xPath.compile("./corps/*");
          final NodeList nlBody = (NodeList)
              bodyPath.evaluate(e, XPathConstants.NODESET);
          IAST2instruction<CEASTparseException> body =
              (IAST2instruction<CEASTparseException>)
              parser.parseChildrenAsSequence(nlBody);
          IAST2variable caughtExceptionVariable = null;
          IAST2instruction<CEASTparseException> catcher = null;
          final XPathExpression catchPath = xPath.compile("./catch");
          final Element nCatch = (Element)
              catchPath.evaluate(e, XPathConstants.NODE);
          if ( null != nCatch ) {
              caughtExceptionVariable =
                  parser.getFactory()
                    .newVariable(nCatch.getAttribute("exception"));
              catcher = (IAST2instruction<CEASTparseException>)
                  parser.parseChildrenAsSequence(nCatch.getChildNodes());
          }
          final XPathExpression finallyPath = xPath.compile("./finally");
          final Element nlFinally = (Element)
              finallyPath.evaluate(e, XPathConstants.NODE);
          IAST2instruction<CEASTparseException> finallyer = null;
          if ( null != nlFinally ) {
              finallyer = (IAST2instruction<CEASTparseException>)
                  parser.parseChildrenAsSequence(nlFinally.getChildNodes());
          }
          IAST3Factory<CEASTparseException> factory = parser.getFactory();
          return factory.newTry(body, caughtExceptionVariable,
                                catcher, finallyer);
      } catch (Exception e1) {
          throw new CEASTparseException(e1);
      }
  }
  private static final XPath xPath = XPathFactory.newInstance().newXPath();

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

  public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
    throws EvaluationException {
      Object result = Boolean.FALSE;
      try {
          // Bizarrement, je n'ai pu écrire return ci-dessous et ai dû
          // passer par la variable result !?
          result = getBody().eval(lexenv, common);
      } catch (ThrownException e) {
          if ( getCatcher() != null ) {
              final ILexicalEnvironment catcherLexenv =
                  lexenv.extend(getCaughtExceptionVariable(), e.getThrownValue());
              getCatcher().eval(catcherLexenv, common);
          } else {
              throw e;
          }
      } catch (EvaluationException e) {
          if ( getCatcher() != null ) {
              final ILexicalEnvironment catcherLexenv =
                  lexenv.extend(getCaughtExceptionVariable(), e);
              // VARIANTE: On pourrait vouloir prefixer par "result = "
              getCatcher().eval(catcherLexenv, common);
          } else {
              throw e;
          }
      } catch (RuntimeException e) {
          if ( getCatcher() != null ) {
              final ILexicalEnvironment catcherLexenv =
                  lexenv.extend(getCaughtExceptionVariable(), e);
              getCatcher().eval(catcherLexenv, common);
```

```
            } else {
                throw e;
            }
        } finally {
            if ( getFinallyer() != null ) {
                getFinallyer().eval(lexenv, common);
            }
        }
        return result;
    }

    public void compileInstruction (final StringBuffer buffer,
                                    final ICgenLexicalEnvironment lexenv,
                                    final ICgenEnvironment common,
                                    final IDestination destination)
      throws CgenerationException {
      buffer.append("{ struct ILP_catcher* current_catcher = ILP_current_catcher; \n");
      buffer.append("  struct ILP_catcher new_catcher;  \n");
      buffer.append("  if ( 0 == setjmp(new_catcher._jmp_buf) ) { \n");
      buffer.append("      ILP_establish_catcher(&new_catcher); \n");
      getBody().compileInstruction(buffer, lexenv, common, destination);
      buffer.append("      ILP_current_exception = NULL; \n");
      buffer.append("  }; \n");

      if ( getCatcher() != null ) {
        final ICgenLexicalEnvironment catcherLexenv =
          lexenv.extend(getCaughtExceptionVariable());
        buffer.append("  ILP_reset_catcher(current_catcher); \n");
        buffer.append("  if ( NULL != ILP_current_exception ) { \n");
        buffer.append("      if ( 0 == setjmp(new_catcher._jmp_buf) ) { \n");
        buffer.append("          ILP_establish_catcher(&new_catcher); \n");
        buffer.append("          { ILP_Object ");
        buffer.append(getCaughtExceptionVariable().getMangledName());
        buffer.append(" = ILP_current_exception; \n");
        buffer.append("            ILP_current_exception = NULL; \n");
        getCatcher().compileInstruction(
                buffer, catcherLexenv, common, VoidDestination.create());
        buffer.append("          } \n");
        buffer.append("      }; \n");
        buffer.append("  }; \n");
      }

      buffer.append("  ILP_reset_catcher(current_catcher); \n");
      if ( getFinallyer() != null ) {
          getFinallyer().compileInstruction(
                  buffer, lexenv, common, VoidDestination.create());
      }
      buffer.append("  if ( NULL != ILP_current_exception ) { \n");
      buffer.append("      ILP_throw(ILP_current_exception); \n");
      buffer.append("  }; \n");
      CEASTinstruction.voidInstruction().compileInstruction(
              buffer, lexenv, common, destination);
      buffer.append("}\n");
    }

    @Override
    public void findGlobalVariables (
                final Set<IAST2variable> globalvars,
                final ICgenLexicalEnvironment lexenv ) {
      getBody().findGlobalVariables(globalvars, lexenv);
      if ( getCatcher() != null ) {
        final ICgenLexicalEnvironment catcherLexenv =
          lexenv.extend(getCaughtExceptionVariable());
        getCatcher().findGlobalVariables(globalvars, catcherLexenv);
      }
      if ( getFinallyer() != null ) {
          getFinallyer().findGlobalVariables(globalvars, lexenv);
      }
    }
}
```

Java/src/fr/upmc/ilp/ilp3/IAST3Factory.java

```
package fr.upmc.ilp.ilp3;

import fr.upmc.ilp.ilp2.interfaces.IAST2Factory;
import fr.upmc.ilp.ilp2.interfaces.IAST2instruction;
import fr.upmc.ilp.ilp2.interfaces.IAST2variable;

public interface IAST3Factory<Exc extends Exception>
extends IAST2Factory<Exc> {

        IAST3try<Exc> newTry (
                IAST2instruction<Exc> body,
                IAST2variable caughtExceptionVariable,
                IAST2instruction<Exc> catcher,
                IAST2instruction<Exc> finallyer);
}
```


Java/src/fr/upmc/ilp/ilp3/IAST3try.java

```
package fr.upmc.ilp.ilp3;

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;

public interface IAST3try<Exc extends Exception>
extends IAST2instruction<Exc> {
    IAST2instruction<CEASTparseException> getBody ();
    @OrNull IAST2variable getCaughtExceptionVariable ();
    @OrNull IAST2instruction<CEASTparseException> getCatcher ();
    @OrNull IAST2instruction<CEASTparseException> getFinallyer ();
}
```

Java/src/fr/upmc/ilp/ilp3/IParser.java

```
package fr.upmc.ilp.ilp3;

public interface IParser<Exc extends Exception>
extends fr.upmc.ilp.ilp2.interfaces.IParser<Exc> {
    IAST3Factory<Exc> getFactory ();
}
```

Java/src/fr/upmc/ilp/ilp3/Process.java

```
package fr.upmc.ilp.ilp3;

import java.io.IOException;

import fr.upmc.ilp.ilp2.ast.CEASTparseException;
import fr.upmc.ilp.ilp2.cgen.CgenEnvironment;
import fr.upmc.ilp.ilp2.cgen.CgenLexicalEnvironment;
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.ilp2.runtime.CommonPlus;
import fr.upmc.ilp.ilp2.runtime.ConstantsStuff;
import fr.upmc.ilp.ilp2.runtime.LexicalEnvironment;
import fr.upmc.ilp.ilp2.runtime.PrintStuff;
import fr.upmc.ilp.tool.FileTool;
import fr.upmc.ilp.tool.IFinder;
import fr.upmc.ilp.tool.ProgramCaller;

/** Cette classe pr?cise comment est trait? un programme d'ILP3.
 *
 *  Je n'herite volontairement pas de fr.upmc.ilp.ilp2.Process afin
 *  de rassembler toutes les informations utiles en une seule classe ici.
 */

public class Process extends fr.upmc.ilp.ilp2.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("grammar3.rng"));
        IAST3Factory<CEASTparseException> factory = new CEASTFactory();
        setFactory(factory);
```

```
    setParser(new CEASTParser(factory));
}

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

/** Pr?paration. H?rit?e! */

/** 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. */

@Override
public void compile() {
    try {
        assert this.prepared;
        final ICgenEnvironment common = new CgenEnvironment();
        common.bindPrimitive("throw");
        final ICgenLexicalEnvironment lexenv =
            CgenLexicalEnvironment.CgenEmptyLexicalEnvironment.create();
        this.ccode = getCEAST().compile(lexenv, common);

        this.compiled = true;

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

/** Ex?cution du programme compil?: */

@Override
public void runCompiled() {
    try {
        assert this.compiled;
        assert this.cFile != null;
        assert this.compileThenRunScript != null;
        FileTool.stuffFile(this.cFile, this.ccode);

        // Optionnel: mettre en forme le programme:
        String indentProgram = "indent " + this.cFile.getAbsolutePath();
        ProgramCaller pcindent = new ProgramCaller(indentProgram);
        pcindent.run();

        // et le compiler:
        String program = "bash "
            + this.compileThenRunScript.getAbsolutePath() + " "
            + this.cFile.getAbsolutePath()
            + " C/ilpError.o C/ilpException.o";
        ProgramCaller pc = new ProgramCaller(program);
        pc.setVerbose();
        pc.run();
        this.executionPrinting = pc.getStdout().trim();

        this.executed = ( pc.getExitValue() == 0 );
```



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

Java/src/fr/upmc/ilp/ilp3/ThrowPrimitive.java

```
package fr.upmc.ilp.ilp3;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp2.runtime.InvokableImpl;

/** Cette classe implante la fonction throw() qui permet de signaler
 * une exception (n'importe quelle valeur d'ILP3). */

public class ThrowPrimitive
  extends InvokableImpl {

  private ThrowPrimitive () {}
  private static final ThrowPrimitive THE_INSTANCE;
  static {
    THE_INSTANCE = new ThrowPrimitive();
  }
  public static ThrowPrimitive create () {
    return THE_INSTANCE;
  }

  /** La valeur à signaler est enveloppée dans une exception et
   * signalée à Java. */

  @Override
  public Object invoke (final Object exception)
    throws EvaluationException {
    if ( exception instanceof EvaluationException ) {
      final EvaluationException exc = (EvaluationException) exception;
      throw exc;
    } else if ( exception instanceof RuntimeException ) {
      final RuntimeException exc = (RuntimeException) exception;
      throw exc;
    } else {
      throw new ThrownException(exception);
    }
  }
}
```

Java/src/fr/upmc/ilp/ilp3/ThrownException.java

```
package fr.upmc.ilp.ilp3;

import fr.upmc.ilp.ilp1.runtime.EvaluationException;

/** La classe des valeurs signalées comme exception. */

public class ThrownException
extends EvaluationException {

  static final long serialVersionUID = +200711241641L;

  public ThrownException (final Object value) {
    super("Thrown value");
    this.value = value;
  }
  private final Object value;

  public Object getThrownValue () {
    return value;
  }

  @Override
  public String toString () {
    return "Thrown value: " + value;
  }
}
```