Code ILP1

Christian.Queinnec@lip6.fr

27 août 2013

Table des matières

LISEZ.MOI

```
Ces fichiers sont diffusés pour l'enseignement ILP (Implantation
d'un langage de programmation) dispensé depuis l'automne 2004 à l'UPMC
(Université Pierre et Marie Curie). Ces fichiers sont diffusés selon
les termes de la GPL (Gnu Public Licence). Pour les transparents du
cours, la bande son et les autres documents associés, consulter le
site http://www-master.ufr-info-p6.jussieu.fr/site-annuel-courant/ilp

Adressez-moi toutes vos remarques concernant ce cours ou ces fichiers
avec un courriel dont le titre contient la chaîne « ILP » ou, mieux,
postez-les sur le forum associé au cours.

Quelques détails sur ILP
========================

Quelques répertoires contiennent de sommaires documentations dans des
fichiers *.readme ou LISEZ.MOI. C'est le cas pour

     Compiler/C/C.readme
     Compiler/Java/jars/JARS.readme

Les Makefile sont également d'importantes sources d'information.

Greffon ILP
===========

Depuis 2006, ILP a introduit un nouveau greffon à incorporer à Eclipse
(au moins 3.2), facilitant quelques opérations. Ce greffon introduit
quelques opérations contextuelles dans l'explorateur de projet. Voici,
par type de fichiers, les opérations principales que l'on peut
obtenir (dans le sous-menu ILP):

 Sur un fichier .rnc (une grammaire compacte d'ILP)
      - conversion en .rng
 Sur un fichier .rng (une grammaire (en XML) pour ILP)
      - positionnement comme grammaire ILP par défaut
 Sur un fichier .xml (un programme ILP)
      - validation vis-à-vis d'une grammaire ILP choisie dynamiquement
      - validation vis-à-vis de la grammaire ILP par défaut
    (il existe maintenant sous Eclipse 3.3, un menu contextuel "Validate"
     qui ne vérifie que la conformité syntaxique vis-à-vis d'XML).
 Sur un répertoire (de grammaires .rng)
      - positionnement comme répertoire des grammaires par défaut

La grammaire par défaut et le répertoire des grammaires par défaut ne
sont pas conservés entre deux sessions.

Le greffon est susceptible d'évoluer, abonnez-vous au site de mise à jour
en http://www.master.info.upmc.fr/2012/Ext/queinnec/ILP/
ce qui va aussi servir à l'installer. Dans le Menu Help, cherchez
Software Updates, Find and Install, Search for new features to install,
new Remote Site. Remplissez le formulaire en indiquant qu'ILP est à l'url
http://www.master.info.upmc.fr/2012/Ext/queinnec/ILP/, OK
puis Finish. Eclipse cherche alors le greffon...

Autres greffons
===============

D'autres greffons sont utiles (ou intéressants) mais on peut s'en
passer, je tenterai toutefois de les faire inclure dans l'installation
à l'ARI. Ce sont:

  Checkstyle
     http://eclipse-cs.sourceforge.net/
  Jdepend pour eclipse
     http://andrei.gmxhome.de/eclipse/
  PMD pour Eclipse
     http://pmd.sf.net/eclipse
  FindBugs
     http://findbugs.cs.umd.edu/eclipse

Depuis Eclipse Helios, on peut installer ces greffons grâce au menu
Help puis Eclipse Market. Chercher alors le greffon par son nom et
l'installer.

Emacs
=====

Le paquetage nxml est réputé. Il se trouve en
  http://www.thaiopensource.com/download/nxml-mode-20041004.tar.gz
```



```
mais une copie est dans le répertoire ELISP/ ainsi qu'un mode
pour éditer des schémas RelaxNG facilement. On peut aussi demander à
Eclipse de plutôt lancer Emacs sur ces fichiers.

Divers
======

Depuis la 3.2, Eclipse dispose d'un éditeur structurel de XML (suffixe
.xml) ainsi qu'un éditeur de grammaires XMLSchema (suffixe .xsd).

Mon Eclipse est réglé, depuis cette année, sur UTF-8, La plupart des
fichiers *.java sont dans ce mode mais d'autres sont restés en Latin1
(iso-8859-1 ou -15). C'est un mal qui affecte le monde entier et qui
ne sera surmonté qu'avec le temps.
```

Grammars/Makefile

```
work :  create.rng.files \
                validate.xml.files
clean :: cleanMakefile
        -rm -f grammar*.rng

# Regenerer toutes les grammaires possibles.
all :
        for g in *.rnc ; do make $${g%c}g ; done

# 2007sep06: trang ne fonctionne pas avec le java d'Ubuntu (gij)! De
# plus, le code de trang n'est pas generique (a ne pas recompiler donc).

JAVA            =       java
TRANG           =       ../Java/jars/trang.jar
JING            =       ../Java/jars/jing.jar
#CODING         =       iso-8859-15
CODING          =       utf-8

.SUFFIXES: .rnc .rng .xsd .dtd
.rnc.rng :
        ${JAVA} -jar ${TRANG} \
                -i encoding=${CODING} \
                -o encoding=${CODING} \
                 $*.rnc $*.rng
.rnc.xsd :
        ${JAVA} -jar ${TRANG} \
                -i encoding=${CODING} \
                -o encoding=${CODING} \
                 $*.rnc $*.xsd
.rnc.dtd :
        ${JAVA} -jar ${TRANG} \
                -i encoding=${CODING} \
                -o encoding=${CODING} \
                 $*.rnc $*.dtd

# NOTA: Valider un document XML d.xml avec un schéma f.rng et jing ainsi:
#       ${JAVA} -jar ${JING} f.rng d.xml

GRAMMARS        =       \
                        grammar1.rnc \
                        grammar2.rnc \
                        grammar3.rnc \
                        grammar4.rnc \
                        grammar5.rnc \
                        grammar6.rnc

# Créer les équivalents XML des schémas RelaxNG compacts:

create.rng.files : ${GRAMMARS:.rnc=.rng}

# Creer les équivalents XSD des schémas RelaxNG compacts:

create.xsd.files : ${GRAMMARS:.rnc=.xsd}

# Les grammaires s'incluent ce qui cree des dependances:
grammar2.rng : grammar1.rng
grammar3.rng : grammar2.rng
grammar4.rng : grammar3.rng
grammar5.rng : grammar4.rng
grammar6.rng : grammar4.rng

# Valider les exemples de programmes qui sont en Samples/
# Verifier que les grammaires sont bien incluses: tout ce que reconnait
```

```
# grammar1 doit etre reconnu par grammar2, etc.

validate.xml.files : ${GRAMMARS:.rnc=.rng}
        for i in 6 5 4 3 2 1 ; do \
        for p in Samples/?*-[1-$$i].xml ; do \
        echo Validating $$p with grammar$$i ; \
        ${JAVA} -jar ${JING} grammar$$i.rng $$p ; \
        done ; done
```

Grammars/grammar1.rnc

```
# Première version du langage étudié: ILP1 pour « Innocent Langage
# Parachuté. » Il sera complété dans les cours qui suivent.

start = programme1

programme1 = element programme1 {
   instructions
}

instructions = instruction +

# Ce langage est un langage d'instructions, assez réduit pour
# l'instant. Voici les instructions possibles:

instruction =
   alternative
 | sequence
 | blocUnaire
 | expression

# Le si-alors-sinon. L'alternant est facultatif car c'est un langage
# d'instructions.

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

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

sequence = element sequence {
   instructions
}

# Un bloc local unaire. C'est, pour l'instant, la seule construction
# permettant d'introduire une variable localement. Elle sera bientôt remplacée
# par une construction permettant d'introduire plusieurs variables
# locales en meme temps.

blocUnaire = element blocUnaire {
   variable,
   element valeur { expression },
   element corps  { instructions }
}

# Comme en C, une expression est une instruction dont la valeur est
# ignorée. Il n'y a pas d'expression parenthésée car ce n'est qu'une
# fioriture syntaxique. Les expressions sont:

expression =
   constante
 | variable
 | operation
 | invocationPrimitive

# Une variable n'est caractérisée que par son nom. Les variables dont
# les noms comportent la séquence ilp ou ILP sont réservés et ne
# peuvent être utilisés par les programmeurs.
# FUTUR: restreindre plus les noms de variables aux seuls caracteres normaux!

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

# L'invocation d'une fonction primitive. Une fonction primitive est
# procurée par l'implantation et ne peut (usuellement) être définie
```



```
# par l'utilisateur. Les fonctions primitives sont, pour être
# utilisables, prédéfinies. Une fonction primitive n'est caractérisée
# que par son nom (éventuellement masquable).

invocationPrimitive = element invocationPrimitive {
   attribute fonction { xsd:Name },
   expression *
}

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

operation =
   operationUnaire
 | operationBinaire

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

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

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

Java/jars/JARS.readme

```
Ce répertoire contient des archives .jar contenant les classes et ressources
utiles pour les bibliothèques telles que jing ou trang. Les paquets .tgz
ou .zip contiennent les distributions originales et notamment la documentation
des interfaces (API).

Les bibliothèques vraiment importantes sont
      jing.jar
      trang.jar
      xmlunit-1.3.jar
      jcommander-1.27
      jdepend-2.9.1

Les bibliothèques Junit (3 et 4) à utiliser sont celles fournies par Eclipse.

JCommander est une bibliothèque pour analyser les options de la ligne
de commander. Elle n'est utilisée que pour des tests. C'est aussi le
cas our la bibliothèque JDepend qui ne sert qu'aux tests.

Les documentations (Javadoc) de jing et trang sont dans le .zip de
même nom. Il faut indiquer à Eclipse où elles sont pour l'aide
contextuelle.

Les autres bibliothèques sont les suivantes
     hansel     une bibliothèque pour mesurer les taux de couverture de test
     bcel       une bibliothèque d'instrumentation de code octet utilisée
                par hansel
     easymock   une bibliothèque pour aider aux tests
     guice      pour de l'injection de dépendances.
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IAST.java

```java
package fr.upmc.ilp.ilp1.interfaces;

/** Interface des arbres de syntaxe abstraite.
 *
 * Les arbres de syntaxe abstraite (AST) ne sont utilisés que via
 * cette interface ou, plus exactement, via une de ses
 * sous-interfaces. Cette interface ressemble un peu à celle du DOM
 * (Document Object Model) sauf qu'elle est beaucoup plus légère: elle
 * est plus typée mais elle ne permet que de descendre dans les AST.
 *
 * La raison d'être de cette interface est qu'elle sert de point de
 * rencontre entre les analyseurs syntaxiques qui doivent produire des
 * IAST qui seront ainsi manipulables par les premières passes de
 * l'interprète ou du compilateur.
 */

public interface IAST {

  /** Décrit l'AST sous forme d'une chaîne imprimable.
   *
   * En fait, c'était pour mettre quelque chose (ce qui n'est pas
   * obligatoire) ! Cette méthode vient avec toute implantation
   * puisque l'objet implantant hérite d'Object qui définit
   * toString()! Par contre, la mentionner ici rend sa définition
   * explicite obligatoire dans les classes qui implantent IAST.
   */
  String toString ();

  // FUTURE ajoute-t-on toXML() pour sérialiser les IAST ?
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTalternative.java

```java
package fr.upmc.ilp.ilp1.interfaces;

import fr.upmc.ilp.annotation.OrNull;

/** Décrit une alternative (si-alors-sinon).
 *
 * Une alternative comporte une condition (booléenne), une conséquence
 * (une expression quelconque) ainsi, éventuellement, qu'un alternant
 * (une expression aussi). La méthode isTernary() permet de distinguer
 * entre ces deux cas.
 */

public interface IASTalternative extends IAST {

  /** Renvoie la condition. */
  IAST getCondition ();

  /** Renvoie la conséquence. */
  IAST getConsequent ();

  /** Renvoie l'alternant si présent ou null.  */
  @OrNull IAST getAlternant ();

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

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTbinaryOperation.java

```java
package fr.upmc.ilp.ilp1.interfaces;

/** Interface décrivant les opérations binaires.
 *
 * Cette interface hérite de celle des opérations qui permet l'accès
 * à l'opérateur et à la liste des opérandes.
 */

public interface IASTbinaryOperation extends IASToperation {

  /** renvoie l'opérande de gauche. */
  IAST getLeftOperand ();

  /** renvoie l'opérande de droite. */
  IAST getRightOperand ();
}
```


Java/src/fr/upmc/ilp/ilp1/interfaces/IASTboolean.java

```java
package fr.upmc.ilp.ilp1.interfaces;

/** Decrit la citation d'un booleen. */

public interface IASTboolean extends IASTconstant {
        /** Renvoie le booleen cite. */
        boolean getValue ();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTconstant.java

```java
package fr.upmc.ilp.ilp1.interfaces;

/** Decrit une constante litterale. Cette interface n'existe que pour
 * etre raffinee en des sous-interfaces plus specialisees. */

public interface IASTconstant extends IAST {}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTfloat.java

```java
package fr.upmc.ilp.ilp1.interfaces;

import java.math.BigDecimal;

/** Citation d'un flottant. */

public interface IASTfloat extends IASTconstant {
        /** Renvoie le flottant cité comme constante. */
        BigDecimal getValue ();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTinteger.java

```java
package fr.upmc.ilp.ilp1.interfaces;

import java.math.BigInteger;

/** Citation d'un entier. */

public interface IASTinteger extends IASTconstant {
        /** Renvoie l'entier cite comme constante. */
        BigInteger getValue ();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTinvocation.java

```java
package fr.upmc.ilp.ilp1.interfaces;

import fr.upmc.ilp.annotation.OrNull;

/** Décrit une invocation de fonction. La fonction peut être calculée
 * ou pas, c'est néanmoins un AST. Si le nom n'est pas calculée, c'est
 * une référence à un nom de variable (car, en JavaScript, les
 * fonctions sont dans le même espace de nom).
 */

public interface IASTinvocation extends IAST {

  /** Renvoie la fonction invoquée. */
  IAST getFunction ();
```

```
    /** Renvoie les arguments de l'invocation sous forme d'une liste. */
    IAST[] getArguments ();

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

    /** Renvoie le i-ème argument de l'invocation ou null. */
    @OrNull IAST getArgument (int i);
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASToperation.java

```
package fr.upmc.ilp.ilp1.interfaces;

/** Interface décrivant les opérations.
 *
 * Une opération implique un opérateur et un ou plusieurs opérandes.
 * Les opérations peuvent être unaires ou binaires cf. les
 * sous-interfaces appropriées.
 */
public interface IASToperation extends IAST {

    /** Renvoie le nom de l'opérateur concerné par l'opération. */
    String getOperatorName ();

    /** Renvoie l'arité de l'opérateur concerné par l'opération. L'arité
     * est toujours de 1 pour une IASTunaryOperation et de 2 pour une
     * IASTbinaryOperation. */
    int getArity ();

    /** Renvoie les opérandes d'une opération.
     *
     * NOTA: cette méthode est générale, les méthodes d'accès aux
     * opérandes des opérations unaires ou binaires sont, probablement,
     * plus efficaces.
     *
     * NOTA2: normalement le nombre d'operandes doit être égal à l'arité
     * de l'opérateur (mais il se peut que certains opérateurs soit n-aires)
     * auquel cas, que l'arité soit un simple entier n'est pas une bonne idée!
     */
    IAST[] getOperands ();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTprogram.java

```
package fr.upmc.ilp.ilp1.interfaces;

public interface IASTprogram extends IAST {
    IASTsequence getBody ();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTsequence.java

```
package fr.upmc.ilp.ilp1.interfaces;

import fr.upmc.ilp.annotation.OrNull;

/** Décrit une séquence d'instructions. On reprend le même
 * style d'interface que IASTinvocation. */

public interface IASTsequence extends IAST {

    /** Renvoie la séquence des instructions contenues. */
    IAST[] getInstructions ();

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

    /** Renvoie la i-ème instruction ou null. */
    @OrNull IAST getInstruction (int i);

    // FUTURE ? ajouter getAllButLastInstructions() ???
}
```


Java/src/fr/upmc/ilp/ilp1/interfaces/IASTstring.java

```
package fr.upmc.ilp.ilp1.interfaces;

/** Citation d'une chaine de caracteres.*/

public interface IASTstring extends IASTconstant {
        /** Renvoie la chaine de caracteres citee. */
        String getValue ();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTunaryBlock.java

```
package fr.upmc.ilp.ilp1.interfaces;

/** Décrit un bloc local ne liant qu'une unique variable. */

public interface IASTunaryBlock extends IAST {

    /** Renvoie la variable liée localement. */
    IASTvariable getVariable();

    /** Renvoie l'expression initialisant la variable locale. */
    IAST getInitialization();

    /** Renvoie la sequence d'instructions présentes dans le corps du
     * bloc local. */
    IASTsequence getBody();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTunaryOperation.java

```
package fr.upmc.ilp.ilp1.interfaces;

/** Interface décrivant les opérations unaires. */

public interface IASTunaryOperation extends IASToperation {
        /** renvoie l'unique operande. */
        IAST getOperand ();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IASTvariable.java

```
package fr.upmc.ilp.ilp1.interfaces;

/**
 * Cette interface décrit une variable mais seulement une variable. D'autres
 * interfaces existent pour lire ou écrire des variables. Usuellement, les
 * variables n'apparaissent que dans des lieurs (fonctions, methodes ou blocs
 * locaux).
 */

public interface IASTvariable extends IAST {
    /** Renvoie le nom de la variable. */
    String getName();
}
```

Java/src/fr/upmc/ilp/ilp1/interfaces/IProcess.java

```
package fr.upmc.ilp.ilp1.interfaces;

import org.w3c.dom.Document;

import fr.upmc.ilp.tool.IContent;
import fr.upmc.ilp.tool.IFinder;

/** Cette interface décrit un java.bean pour l'évaluation d'un
 * programme ILP.
 */
```

```
public interface IProcess {

    // Rendre le processus plus verbeux:
    void setVerbose (boolean verbose);

    // Indiquer où trouver l'utilitaire qui sait chercher les fichiers utiles:
    IFinder getFinder ();
    void setFinder (IFinder finder);

    // Cycle de vie

    /** Initialiser le processus d'évaluation avec un contenu (le plus
     * souvent un fichier) c'est-à-dire le texte du programme. Si une
     * exception survient lors de l'initialisation, elle sera stockée
     * en getInitializationFailure(). Si tout se passe bien, le texte du
     * programme à considérer sera accessible en getProgramText(). */
    void initialize (IContent ic);
    boolean isInitialized ();
    Throwable getInitializationFailure ();
    String getProgramText ();

    /** Préparer le processus d'évaluation jusqu'à choisir entre
     * interprétation ou compilation. Toutes les phases d'analyse communes
     * sont effectuées ici. Si une exception survient lors de la préparation,
     * elle sera stockée en getPreparationFailure(). Si tout se passe bien
     * l'AST obtenu sera accessible via getIAST(). */
    void prepare ();
    boolean isPrepared ();
    Throwable getPreparationFailure ();
    Document getDocument ();
    void setGrammar(java.io.File rngFile);
    java.io.File getGrammar();
    IAST getIAST();
    void setIAST(IAST iast);
    // getParser()
    // setParser(IParser iparser)

    /** Évaluer par interprétation. Si une exception survient, elle sera
     * accessible en getInterpretationFailure(). Si une valeur est obtenue,
     * elle sera accessible en getInterpretationValue(), ce qui est écrit
     * par les primitives print et newline est accumulé et accessible par
     * getInterpretationPrinting(). */
    void interpret ();
    boolean isInterpreted ();
    Throwable getInterpretationFailure ();
    Object getInterpretationValue ();
    String getInterpretationPrinting ();

    /** Compiler vers un fichier. Si une exception survient, elle sera
     * accessible avec getCompilationFailure(). Si la compilation se passe
     * bien, le code produit sera accessible via getCompiledProgram(). La
     * compilation vers C est aidée par un patron (setCTemplateFile) et
     * par un script (seCompileThenRunScript) */
    void compile ();
    boolean isCompiled ();
    Throwable getCompilationFailure ();
    String getCompiledProgram ();
    void setCFile(java.io.File cFile);
    void setCompileThenRunScript (java.io.File scriptFile);
    // File getCompiledFile();

    /** Exécuter le fichier compilé. Si un problème survient, il sera
     * stocké en getExecutionFailure(). Sinon, ce qu'imprime le programme
     * compilé est accessible via getExecutionPrinting(). */
    void runCompiled ();
    boolean isExecuted ();
    Throwable getExecutionFailure ();
    String getExecutionPrinting ();
}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/AST.java

```
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

public abstract class AST implements IAST {

  /** Décrit l'AST en XML (surtout utile pour la mise au point).
   *
   * NOTA: cette signature conduit à une implantation naïve
   * déraisonnablement coûteuse! Il vaudrait mieux se trimballer un
   * unique StringBuffer dans lequel concaténer les fragments XML.
   */

  public abstract String toXML ();

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTException.java

```
package fr.upmc.ilp.ilp1.fromxml;

/** Une exception comme les autres mais qu'utilisent préférentiellement
 * les classes du paquetage fromxml.
 */

public class ASTException extends Exception {

        static final long serialVersionUID = +1234567890001000L;

        public ASTException (Throwable cause) {
                super(cause);
        }

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

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTParser.java

```
package fr.upmc.ilp.ilp1.fromxml;
import java.util.List;
import java.util.Vector;

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

/** Le but de cette classe est de transformer un document XML en un
 * AST conforme à fr.upmc.ilp.interfaces.IAST.  */

public class ASTParser {

  /** Constructeur.  */

  public ASTParser () {}

  /** Transformer un document (ou un n?ud) DOM en un AST (et donc un
   * IAST).
   *
   * C'est une grande méthode monolithique qui analyse le DOM (une
   * structure de données arborescente correspondant au document XML
   * initial) en un AST où chaque n?ud est typé suivant sa catégorie
   * syntaxique. Il est difficile d'avoir un style objet sur ce type
   * de code qui correspond à une construction. Nous verrons une
   * nouvelle organisation de ce code bientôt.
   *
   * @throws ASTException en cas de problème.
   * NOTA: comme le document d'entrée est supposé être valide pour le
   * schéma RelaxNG approprié, de nombreux risques d'erreur sont ainsi
   * éliminés et ne sont donc pas explicitement testés. Par exemple,
   * le fait qu'un fils « condition » apparaît toujours sous «
   * alternative, » que l'attribut « valeur » est présent dans «
   * entier, » etc. Tous ces faits sont assumés corrects.
   */

  public AST parse (Node n)
    throws ASTException {
    switch ( n.getNodeType() ) {

    case Node.DOCUMENT_NODE: {
      Document d = (Document) n;
```

```
    return this.parse(d.getDocumentElement());
  }

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

    switch (name) {
    case "programme1" : {
        return new ASTprogram(this.parseList(nl));
    }
    case "alternative" : {
      AST cond = findThenParseChild(nl, "condition");
      AST conseq = findThenParseChildAsInstructions(nl, "consequence");
      try {
        AST alt = findThenParseChildAsInstructions(nl, "alternant");
        return new ASTalternative(cond, conseq, alt);
      } catch (ASTException exc) {
        return new ASTalternative(cond, conseq);
      }
    }
    case "sequence": {
      return new ASTsequence(this.parseList(nl));
    }
    case "blocUnaire": {
      ASTvariable var = (ASTvariable) findThenParseChild(nl, "variable");
      AST init = findThenParseChild(nl, "valeur");
      ASTsequence body = (ASTsequence) findThenParseChild(nl, "corps");
      return new ASTunaryBlock(var, init, body);
    }
    case "variable": {
      // La variable sera, suivant les contextes, encapsulée
      // dans une référence.
      String nick = e.getAttribute("nom");
      return new ASTvariable(nick);
    }
    case "invocationPrimitive": {
      String op = e.getAttribute("fonction");
      List<AST> largs = parseList(nl);
      return new ASTinvocationPrimitive(op, largs.toArray(new AST[0]));
    }
    case "operationUnaire": {
      String op = e.getAttribute("operateur");
      AST rand = findThenParseChild(nl, "operande");
      return new ASTunaryOperation(op, rand);
    }
    case "operationBinaire": {
      String op = e.getAttribute("operateur");
      AST gauche = findThenParseChild(nl, "operandeGauche");
      AST droite = findThenParseChild(nl, "operandeDroit");
      return new ASTbinaryOperation(op, gauche, droite);
    }
    case "entier": {
      return new ASTinteger(e.getAttribute("valeur"));
    }
    case "flottant": {
      return new ASTfloat(e.getAttribute("valeur"));
    }
    case "chaine": {
      String text = e.getTextContent();
      return new ASTstring(text);
    }
    case "booleen": {
      return new ASTboolean(e.getAttribute("valeur"));
    }
    // Une série d'éléments devant être analysés comme des expressions:
    case "operandeGauche":
    case "operandeDroit":
    case "operande":
    case "condition":
    case "consequence":
    case "alternant":
    case "valeur": {
      return this.parseUniqueChild(nl);
    }
    // Une série d'éléments devant être analysée comme une séquence:
    case "corps": {
      return new ASTsequence(this.parseList(nl));
    }
    default: {
```



```
        String msg = "Unknown element name: " + name;
        throw new ASTException(msg);
      }
    }
    }

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

  /** Analyser une séquence d'éléments pour en faire un ASTlist
   * c'est-à-dire une séquence d'AST.
   */

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

      case Node.ELEMENT_NODE: {
        AST p = this.parse(nd);
        result.add(p);
        continue LOOP;
      }

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

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

  protected AST findThenParseChild (NodeList nl, String childName)
    throws ASTException {
    int n = nl.getLength();
    for ( int i = 0 ; i<n ; i++ ) {
      Node nd = nl.item(i);
      switch ( nd.getNodeType() ) {

      case Node.ELEMENT_NODE: {
        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:
      }
      }
    }
    String msg = "No such child element " + childName;
    throw new ASTException(msg);
  }

  /** Trouver un élément d'après son nom et analyser son contenu pour
   * en faire un ASTsequence.
   *
   * @throws ASTException
   *   lorsqu'un tel élément n'est pas trouvé.
   */

  protected ASTsequence findThenParseChildAsInstructions (
          NodeList nl, String childName)
```

```
    throws ASTException {
    int n = nl.getLength();
    for ( int i = 0 ; i<n ; i++ ) {
      Node nd = nl.item(i);
      switch ( nd.getNodeType() ) {

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

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

  /** Analyser une suite comportant un unique élément.
   *
   * @throws ASTException
   *   lorsque la suite ne contient pas exactement un seul élément.
   */

  protected AST parseUniqueChild (NodeList nl)
  throws ASTException {
      AST result = null;
      int n = nl.getLength();
      for ( int i = 0 ; i<n ; i++ ) {
          Node nd = nl.item(i);
          switch ( nd.getNodeType() ) {

          case Node.ELEMENT_NODE: {
              Element e = (Element) nd;
              if ( result == null ) {
                  result = this.parse(e);
              } else {
                  String msg = "Non unique child";
                  throw new ASTException(msg);
              }
              break;
          }

          default: {
              // On ignore tout ce qui n'est pas élément XML:
          }
          }
      }
      if ( result == null ) {
          throw new ASTException("No child at all");
      }
      return result;
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTalternative.java

```
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.interfaces.IAST;
import fr.upmc.ilp.ilp1.interfaces.IASTalternative;

/** La représentation des alternatives (binaires ou ternaires). */

public class ASTalternative extends AST
implements IASTalternative {

  public ASTalternative (AST condition,
                         AST consequence,
                         @OrNull AST alternant ) {
    this.condition   = condition;
    this.consequence = consequence;
    this.alternant   = alternant;
  }

  // NOTA: Masquer l'implantation de l'alternative binaire afin
  // d'éviter la propagation de null.

  public ASTalternative (AST condition, AST consequence) {
    this(condition, consequence, null);
  }

  private final AST condition;
  private final AST consequence;
  private final AST alternant;

  public IAST getCondition () {
    return this.condition;
  }

  public IAST getConsequent () {
    return this.consequence;
  }

  public @OrNull IAST getAlternant () {
    return this.alternant;
  }

  /** Vérifie que l'alternative est ternaire c'est-à-dire qu'elle a un
      véritable alternant. */

  public boolean isTernary () {
    return this.alternant != null;
  }

  @Override
  public String toXML () {
    StringBuffer sb = new StringBuffer();
    sb.append("<alternative><condition>");
    sb.append(condition.toXML());
    sb.append("</condition><consequence>");
    sb.append(consequence.toXML());
    sb.append("</consequence>");
    if ( isTernary() ) {
      sb.append("<alternant>");
      sb.append(alternant.toXML());
      sb.append("</alternant>");
    }
    sb.append("</alternative>");
    return sb.toString();
  }
}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTbinaryOperation.java

```
package fr.upmc.ilp.ilp1.fromxml;

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

import fr.upmc.ilp.ilp1.interfaces.IAST;
import fr.upmc.ilp.ilp1.interfaces.IASTbinaryOperation;

/** Opérations binaires. */

public class ASTbinaryOperation extends ASToperation
implements IASTbinaryOperation {

    public ASTbinaryOperation(String operateur, AST operandeGauche,
            AST operandeDroit) {
        super(operateur, 2);
        this.operandeGauche = operandeGauche;
        this.operandeDroit = operandeDroit;
    }
    private final AST operandeGauche;
    private final AST operandeDroit;

    public IAST getLeftOperand() {
```

```
        return this.operandeGauche;
    }

    public IAST getRightOperand() {
        return this.operandeDroit;
    }

    public IAST[] getOperands() {
        // On calcule paresseusement car ce n'est pas une méthode usuelle:
        if (operands == null) {
            List<AST> loperands = new Vector<>();
            loperands.add(operandeGauche);
            loperands.add(operandeDroit);
            operands = loperands.toArray(new AST[0]);
        }
        return operands;
    }

    // NOTA: remarquer que ce mode paresseux interdit de qualifier ce
    // champ de « final » ce qui n'est pas sûr!
    private IAST[] operands;

    @Override
    public String toXML() {
        StringBuffer sb = new StringBuffer();
        // Comme signalé par Olivier.Tran@etu.upmc.fr, il faudrait coder
        // getOperatorName() pour utiliser les entités &lt; &gt; etc.
        sb.append("<operationBinaire operateur='" + getOperatorName() + "'>");
        sb.append("<operandeGauche>");
        sb.append(operandeGauche.toXML());
        sb.append("</operandeGauche><operandeDroit>");
        sb.append(operandeDroit.toXML());
        sb.append("</operandeDroit></operationBinaire>");
        return sb.toString();
    }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTboolean.java

```
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

/** Une constante booleenne. */

public class ASTboolean extends AST
implements IASTboolean {

  public ASTboolean (String valeur) {
    this.valeur = "true".equals(valeur);
  }
  private final boolean valeur;

  public boolean getValue () {
    return valeur;
  }

  @Override
  public String toXML () {
    return "<booleen valeur='" + valeur + "'/>";
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTfloat.java

```
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

import java.math.BigDecimal;

/** Une constante flottante. */

public class ASTfloat extends AST
implements IASTfloat {
```



```
  public ASTfloat (String valeur) {
    this.valeur = Double.parseDouble(valeur);
    this.bigfloat = new BigDecimal(valeur);
  }
  private final double valeur;
  private final BigDecimal bigfloat;

  public BigDecimal getValue () {
    return bigfloat;
  }

  @Override
  public String toXML () {
    return "<flottant valeur='" + valeur + "'/>";
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTinteger.java

```
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

import java.math.BigInteger;

/** Une constante entière. */

public class ASTinteger extends AST
implements IASTinteger {

  public ASTinteger (String valeur) {
    this.valeur = Integer.parseInt(valeur);
    this.bigint = new BigInteger(valeur);
  }
  private final int valeur;
  private final BigInteger bigint;

  public BigInteger getValue () {
    return bigint;
  }

  @Override
  public String toXML () {
    return "<entier valeur='" + valeur + "'/>";
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTinvocation.java

```
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.IASTinvocation;

/** Une invocation mentionne une fonction et des arguments. */

public class ASTinvocation extends AST
implements IASTinvocation {

  public ASTinvocation (AST fonction, AST[] arguments) {
    this.fonction = fonction;
    this.argument = arguments;
  }
  private final AST   fonction;
  private final AST[] argument;

  public AST getFunction () {
    return this.fonction;
  }

  public AST[] getArguments () {
    return this.argument;
  }

  public int getArgumentsLength () {
```

```
    return this.argument.length;
  }

  public AST getArgument (int i) {
    return this.argument[i];
  }

  @Override
  public String toXML () {
    StringBuffer sb = new StringBuffer();
    sb.append("<invocation><function>");
    sb.append(fonction.toXML());
    sb.append("</function>");
    //sb.append("<arguments>");
    for ( AST arg : this.argument ) {
        sb.append(arg.toXML());
    }
    // sb.append("</arguments>");
    sb.append("</invocation>");
    return sb.toString();
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTinvocationPrimitive.java

```
package fr.upmc.ilp.ilp1.fromxml;

/** Le cas particulier, parmi les invocations de fonctions,
 * des primitives.
 */

public class ASTinvocationPrimitive
  extends ASTinvocation {

  public ASTinvocationPrimitive (String fonction, AST[] arguments) {
    super(new ASTvariable(fonction), arguments);
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASToperation.java

```
package fr.upmc.ilp.ilp1.fromxml;

/** La classe abstraite des opérations.
 * Il y en a deux sortes: les unaires et les binaires.
 */

public abstract class ASToperation extends AST
{
  protected ASToperation (String operateur, int arity) {
    this.operateur = operateur;
    this.arity = arity;
  }
  private final String operateur;
  private final int arity;

  public String getOperatorName () {
    return this.operateur;
  }

  public int getArity () {
    return this.arity;
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTprogram.java


```
package fr.upmc.ilp.ilp1.fromxml;

import java.util.List;

import fr.upmc.ilp.ilp1.interfaces.IASTprogram;
import fr.upmc.ilp.ilp1.interfaces.IASTsequence;

public class ASTprogram extends AST implements IASTprogram {

    public ASTprogram (List<AST> body) {
        this.body = body;
    }
    protected List<AST> body;

    public IASTsequence getBody() {
        return new ASTsequence(this.body);
    }

    @Override
    public String toXML() {
        StringBuffer sb = new StringBuffer();
        sb.append("<programme1>");
        // Juste pour simplifier les tests dans ASTParserTest:
        if ( this.body.size() == 1 ) {
            sb.append(this.body.get(0).toXML());
        } else {
            sb.append(new ASTsequence(this.body).toXML());
        }
        sb.append("</programme1>");
        return sb.toString();
    }
}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTsequence.java

```
package fr.upmc.ilp.ilp1.fromxml;

import java.util.List;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.interfaces.IAST;
import fr.upmc.ilp.ilp1.interfaces.IASTsequence;

public class ASTsequence extends AST
implements IASTsequence {

    public ASTsequence(List<AST> instructions) {
        this.instructions = instructions;
    }
    private final List<AST> instructions;

    public IAST[] getInstructions() {
        return this.instructions.toArray(new IAST[0]);
    }

    // NOTA: si i est hors limite, cette methode ne ramene pas null comme le
    // dit l'interface mais signale une exception IndexOutOfRange.
    public @OrNull IAST getInstruction(int i) {
        return this.instructions.get(i);
    }

    public int getInstructionsLength() {
        return this.instructions.size();
    }

    @Override
    public String toXML() {
        StringBuffer sb = new StringBuffer("<sequence>");
        for (AST instruction : this.instructions) {
            sb.append(instruction.toXML());
        }
        sb.append("</sequence>");
        return sb.toString();
    }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTstring.java

```java
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

/** Une constante chaine de caracteres. */

public class ASTstring extends AST
implements IASTstring {

  public ASTstring (String valeur) {
    this.valeur = valeur;
  }
  private final String valeur;

  public String getValue () {
    return valeur;
  }

  @Override
  public String toXML () {
    return "<chaine>" + valeur + "</chaine>";
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTunaryBlock.java

```java
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

/** Description d'un bloc local pourvu d'une seule variable locale, de
 * l'expression initialisant cette variable locale et d'un corps qui
 * est une séquence d'instructions.
 */

public class ASTunaryBlock extends AST
implements IASTunaryBlock {

  public ASTunaryBlock (ASTvariable variable,
                        AST         initialization,
                        ASTsequence body)
  {
    this.variable       = variable;
    this.initialization = initialization;
    this.body           = body;
  }

  private final ASTvariable variable;
  private final AST         initialization;
  private final ASTsequence body;

  public IASTvariable getVariable () {
    return this.variable;
  }
  public IAST getInitialization () {
    return this.initialization;
  }
  public IASTsequence getBody () {
    return this.body;
  }

  @Override
  public String toXML () {
    StringBuffer sb = new StringBuffer();
    sb.append("<blocUnaire>");
    sb.append(variable.toXML());
    sb.append("<valeur>");
    sb.append(initialization.toXML());
    sb.append("</valeur><corps>");
    sb.append(body.toXML());
    sb.append("</corps></blocUnaire>");
    return sb.toString();
  }

}
```


Java/src/fr/upmc/ilp/ilp1/fromxml/ASTunaryOperation.java

```java
package fr.upmc.ilp.ilp1.fromxml;

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

import fr.upmc.ilp.ilp1.interfaces.IAST;
import fr.upmc.ilp.ilp1.interfaces.IASTunaryOperation;

/** Opérations unaires. */

public class ASTunaryOperation extends ASToperation
implements IASTunaryOperation {

    public ASTunaryOperation(String operateur, AST operand) {
        super(operateur, 1);
        this.operand = operand;
    }
    private final AST operand;

    public IAST getOperand() {
        return this.operand;
    }

    public IAST[] getOperands() {
        // On calcule paresseusement car ce n'est pas une méthode usuelle:
        if (operands == null) {
            List<AST> loperands = new Vector<>();
            loperands.add(operand);
            operands = loperands.toArray(new AST[0]);
        }
        return operands;
    }
    private AST[] operands = null;

    @Override
    public String toXML() {
        StringBuffer sb = new StringBuffer();
        sb.append("<operationUnaire operateur='" + getOperatorName() + "'>");
        sb.append("<operande>");
        sb.append(operand.toXML());
        sb.append("</operande></operationUnaire>");
        return sb.toString();
    }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/ASTvariable.java

```java
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

/** Description d'une variable. */

public class ASTvariable
  extends AST
  implements IASTvariable {

  public ASTvariable (String name) {
    this.name = name;
  }
  private final String name;

  public String getName () {
    return name;
  }

  @Override
  public String toXML () {
    return "<variable nom='" + name + "'/>";
  }

}
```

Java/src/fr/upmc/ilp/ilp1/fromxml/Main.java

```java
package fr.upmc.ilp.ilp1.fromxml;
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import com.thaiopensource.validate.*;

/** Lit puis évalue un programme. Le langage utilisé est ILP1 défini
 * par grammar1.rnc
 *
 * Usage: java fr.upmc.ilp.fromxml.Main grammaire.rng programme.xml
 */

public class Main {

  public static void main (String[] argument)
    throws ASTException {
    Main m = new Main(argument);
    m.run();
  }

  public Main (String[] argument)
    throws ASTException {
    if ( argument.length < 2 ) {
      throw new ASTException("Usage: Main grammaire.rng programme.xml");
    }
    this.rngfile = new File(argument[0]);
    this.xmlfile = new File(argument[1]);
    if ( !this.rngfile.exists() ) {
      throw new ASTException("Fichier .rng introuvable.");
    }
    if ( !this.xmlfile.exists() ) {
      throw new ASTException("Fichier .xml introuvable.");
    }
  }

  private File rngfile;
  private File xmlfile;

  public void run ()
    throws ASTException {

    try {

      // (1) validation vis-à-vis de RNG:
      // MOCHE: c'est redondant avec (2) car le programme est relu encore une
      // fois avec SAX. Les phases 1 et 2 pourraient s'effectuer ensemble.
      ValidationDriver vd = new ValidationDriver();
      InputSource isg = ValidationDriver.fileInputSource(
            rngfile.getAbsolutePath());
      vd.loadSchema(isg);
      InputSource isp = ValidationDriver.fileInputSource(
            this.xmlfile.getAbsolutePath());
      if ( ! vd.validate(isp) ) {
        throw new ASTException("programme XML non valide");
      }

      // (2) convertir le fichier XML en DOM:
      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document d = db.parse(this.xmlfile);

      // (3) conversion vers un AST donc un IAST:
      ASTParser ap = new ASTParser();
      AST ast = ap.parse(d);

      // (3bis) Impression en XML:
      System.out.println(ast.toXML());

    } catch (ASTException e) {
      throw e;
    } catch (Throwable cause) {
      throw new ASTException(cause);
    }
  }

}
```


Java/src/fr/upmc/ilp/ilp1/runtime/AbstractInvokableImpl.java

```java
package fr.upmc.ilp.ilp1.runtime;

/** Une classe abstraite de fonction qui peut servir de base à des
 * implantations particulières. Il suffit, si la fonction a moins de
 * quatre arguments, de définir la méthode invoke appropriée.
 * @see fr.upmc.ilp.ilp1.runtime.PrintStuff
 */

public abstract class AbstractInvokableImpl
  implements Invokable {

  private static final String WRONG_ARITY =
    "Wrong arity";

  /** Une fonction invoquée avec un nombre quelconque d'arguments. Les
   * petites arités sont renvoyées sur les méthodes appropriées. */

  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 zéro-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/ilp1/runtime/Common.java

```java
package fr.upmc.ilp.ilp1.runtime;
import java.math.BigInteger;

/** Cette interface définit les caractéristiques globales d'un
 * interprète du langage ILP1. On y trouve, notamment, la définition
 * des opérateurs du langage.
 */

public class Common implements ICommon {
```

```
public Common () {}

/** Les opérateurs unaires.
 *
 * Comme il n'y a que deux tels opérateurs, leur définition est
 * intégrée dans cette méthode. */

public Object applyOperator (String opName, Object operand)
  throws EvaluationException {
  checkNotNull(opName, 1, operand);

  if ( "-".equals(opName) ) {
    if ( operand instanceof BigInteger ) {
      BigInteger bi = (BigInteger) operand;
      return bi.negate();
    } else if ( operand instanceof Double ) {
      double bd = ((Double) operand).doubleValue();
      return new Double(-bd);
    } else {
      return signalWrongType(opName, 1, operand, "number");
    }

  } else if ( "!".equals(opName) ) {
    if ( operand instanceof Boolean ) {
      Boolean b = (Boolean) operand;
      return Boolean.valueOf( ! b.booleanValue() );
    } else {
      return signalWrongType(opName, 1, operand, "boolean");
    }

  } else {
    String msg = "Unknown unary operator: " + opName;
    throw new EvaluationException(msg);
  }
}

/** Les opérateurs binaires.
 *
 * Cette méthode n'est qu'un grand aiguillage. */

public Object applyOperator (String opName,
                             Object leftOperand,
                             Object rightOperand)
  throws EvaluationException
{
  if ( "+".equals(opName) ) {
    return operatorPlus(opName, leftOperand, rightOperand);
  } else if ( "-".equals(opName) ) {
    return operatorMinus(opName, leftOperand, rightOperand);

    // continuer                                          TEMP

  } else {
    String msg = "Unknown binary operator: " + opName;
    throw new EvaluationException(msg);
  }
}

// Vérifications diverses:

private void checkNotNull (String opName, int rank, Object o)
  throws EvaluationException {
  if ( o == null ) {
    String msg = opName + ": Argument " + rank + " is null!\n"
      + "Value is: " + o;
    throw new EvaluationException(msg);
  }
}

private Object signalWrongType (String opName, int rank, Object o,
                                String expectedType)
  throws EvaluationException {
 String msg = opName + ": Argument " + rank + " is not "
   + expectedType + "\n" + "Value is: " + o;
  throw new EvaluationException(msg);
}

// {{{ Les opérateurs binaires:
```



```
/** Le traitement de l'addition. C'est compliqué car il y a de
 * nombreuses conversions possibles. */

private Object operatorPlus (String opName, Object a, Object b)
  throws EvaluationException {
  checkNotNull(opName, 1, a);
  checkNotNull(opName, 2, b);
  if ( a instanceof BigInteger ) {
    BigInteger bi1 = (BigInteger) a;
    if ( b instanceof BigInteger ) {
      BigInteger bi2 = (BigInteger) b;
      return bi1.add(bi2);
    } else if ( b instanceof Double ) {
      double bd1 = bi1.doubleValue();
      double bd2 = ((Double) b).doubleValue();
      return new Double(bd1 + bd2);
    } else {
      return signalWrongType(opName, 2, b, "number");
    }
  } else if ( a instanceof Double ) {
    Double bd1 = (Double) a;
    if ( b instanceof Double ) {
      Double bd2 = (Double) b;
      return new Double(bd1.doubleValue() + bd2.doubleValue());
    } else if ( b instanceof BigInteger ) {
      BigInteger bi2 = (BigInteger) b;
      double bd2 = bi2.doubleValue();
      return new Double(bd1.doubleValue() + bd2);
    } else {
      return signalWrongType(opName, 2, b, "number");
    }
  } else {
    return signalWrongType(opName, 1, a, "number");
  }
}

/** Le traitement de la soustraction. Ca ressemble à l'addition. */

private Object operatorMinus (String opName, Object a, Object b)
  throws EvaluationException {
  checkNotNull(opName, 1, a);
  checkNotNull(opName, 2, b);
  if ( a instanceof BigInteger ) {
    BigInteger bi1 = (BigInteger) a;
    if ( b instanceof BigInteger ) {
      BigInteger bi2 = (BigInteger) b;
      return bi1.subtract(bi2);
    } else if ( b instanceof Double ) {
      double bd1 = bi1.doubleValue();
      double bd2 = ((Double) b).doubleValue();
      return new Double(bd1 - bd2);
    } else {
      return signalWrongType(opName, 2, b, "number");
    }
  } else if ( a instanceof Double ) {
    Double bd1 = (Double) a;
    if ( b instanceof Double ) {
      Double bd2 = (Double) b;
      return new Double(bd1.doubleValue() - bd2.doubleValue());
    } else if ( b instanceof BigInteger ) {
      BigInteger bi2 = (BigInteger) b;
      double bd2 = bi2.doubleValue();
      return new Double(bd1.doubleValue() - bd2);
    } else {
      return signalWrongType(opName, 2, b, "number");
    }
  } else {
    return signalWrongType(opName, 1, a, "number");
  }
}

// }}}

}
```

Java/src/fr/upmc/ilp/ilp1/runtime/CommonPlus.java

```
//
// ATTENTION: NE PAS MODIFIER! CETTE CLASSE A ÉTÉ ENGENDRÉE AUTOMATIQUEMENT
// A PARTIR DE BinOp.php (voir build.xml pour les détails).

/* ****************************************************************
 * ILP -- Implantation d'un langage de programmation.
 * Copyright (C) 2004-2005 <Christian.Queinnec@lip6.fr>
 * $Id: CommonPlus.java 1224 2012-08-27 20:07:18Z queinnec $
 * GPL version>=2
 * ****************************************************************/

package fr.upmc.ilp.ilp1.runtime;

import java.math.BigInteger;

/** Cette classe (engendrée par PHP) implante les caractéristiques
 * générales d'un interprète du langage ILP1. On y trouve, notamment,
 * la définition des opérateurs du langage.
 */

public class CommonPlus implements ICommon {

    public CommonPlus () {}

    // Vérifications diverses:

    private void checkNotNull (final String opName,
                               final int rank,
                               final Object o)
        throws EvaluationException {
        if ( o == null ) {
            final String msg = opName + ": Argument " + rank + " is null!\n"
                + "Value is: " + o;
            throw new EvaluationException(msg);
        }
    }

    private Object signalWrongType (final String opName,
                                    final int rank,
                                    final Object o,
                                    final String expectedType)
        throws EvaluationException {
        final String msg = opName + ": Argument " + rank + " is not "
            + expectedType + "\n" + "Value is: " + o;
        throw new EvaluationException(msg);
    }

    /** Les opérateurs unaires.
     *
     * Comme il n'y a que deux tels opérateurs, leur définition est
     * intégrée directement dans cette méthode plutôt que d'être
     * macro-générée. */

    public Object applyOperator (final String opName, final Object operand)
        throws EvaluationException {
        checkNotNull(opName, 1, operand);

        if ( "-".equals(opName) ) {
            if ( operand instanceof BigInteger ) {
                final BigInteger bi = (BigInteger) operand;
                return bi.negate();
            } else if ( operand instanceof Double ) {
                // Profitons du déballage (unboxing) automatique:
                // final double bd = ((Double) operand).doubleValue();
                final double bd = (Double) operand;
                return new Double(-bd);
            } else {
                return signalWrongType(opName, 1, operand, "number");
            }

        } else if ( "!".equals(opName) ) {
            if ( operand instanceof Boolean ) {
                final Boolean b = (Boolean) operand;
                return Boolean.valueOf( ! b.booleanValue() );
            } else {
                return signalWrongType(opName, 1, operand, "boolean");
            }
```



```
        } else {
            final String msg = "Unknown unary operator: " + opName;
            throw new EvaluationException(msg);
        }
    }

    /** Les opérateurs binaires.
     *
     * Cette méthode n'est qu'un grand aiguillage. */

    public Object applyOperator (final String opName,
                                 final Object leftOperand,
                                 final Object rightOperand)
        throws EvaluationException
        {
            // Les opérateurs:

        if ( "+".equals(opName) ) {
            return operatorPlus(opName, leftOperand, rightOperand);
        } else
        if ( "-".equals(opName) ) {
            return operatorMinus(opName, leftOperand, rightOperand);
        } else
        if ( "*".equals(opName) ) {
            return operatorMultiply(opName, leftOperand, rightOperand);
        } else
        if ( "/".equals(opName) ) {
            return operatorQuotient(opName, leftOperand, rightOperand);
        } else
        if ( "%".equals(opName) ) {
            return operatorModulo(opName, leftOperand, rightOperand);
        } else

          // Les comparateurs:

        if ( "<".equals(opName) ) {
            return operatorLT(opName, leftOperand, rightOperand);
        } else
        if ( "<=".equals(opName) ) {
            return operatorLE(opName, leftOperand, rightOperand);
        } else
        if ( "==".equals(opName) ) {
            return operatorEQ(opName, leftOperand, rightOperand);
        } else
        if ( ">=".equals(opName) ) {
            return operatorGE(opName, leftOperand, rightOperand);
        } else
        if ( ">".equals(opName) ) {
            return operatorGT(opName, leftOperand, rightOperand);
        } else
        if ( "!=".equals(opName) ) {
            return operatorNEQ(opName, leftOperand, rightOperand);
        } else
        // L'operateur != est aussi connu sous l'alias <>
        // Cet oubli a été signalé par Cristian.Loiza_Soto@etu.upmc.fr:
        if ( "<>".equals(opName) ) {
            return operatorNEQ(opName, leftOperand, rightOperand);
        } else

      // Les opérateurs booléens:
      if ( "&".equals(opName) ) {
          if ( leftOperand == Boolean.FALSE ) {
              return Boolean.FALSE;
          } else {
              return rightOperand;
          }
      } else
      if ( "|".equals(opName) ) {
          if ( leftOperand != Boolean.FALSE ) {
              return leftOperand;
          } else {
              return rightOperand;
          }
      } else
      if ( "^".equals(opName) ) {
          boolean left = (leftOperand != Boolean.FALSE);
          boolean right = (rightOperand != Boolean.FALSE);
          return (Boolean) (left != right);
      } else
```

```
         // NOTA: il serait plus astucieux de ranger les branches de
         // ces alternatives par ordre d'usage décroissant!
    {
         final String msg = "Unknown binary operator: " + opName;
         throw new EvaluationException(msg);
    }
}

// Le cas de l'opérateur binaire + est plus compliqué car il permet
// aussi la concaténation de chaînes de caractères. Il sera donc
// traité à part et donc écrit à la main. C'est tout comme pour
// modulo qui ne s'applique qu'à des entiers et non à des flottants.
// Tous les autres opérateurs binaires seront macro-générés.

/** Définition de l'opérateur binaire + */

private Object operatorPlus (final String opName,
                             final Object a,
                             final Object b)
     throws EvaluationException {
     checkNotNull(opName, 1, a);
     checkNotNull(opName, 2, b);
     if ( a instanceof BigInteger ) {
          final BigInteger bi1 = (BigInteger) a;
          if ( b instanceof BigInteger ) {
               final BigInteger bi2 = (BigInteger) b;
               return bi1.add(bi2);
          } else if ( b instanceof Double ) {
               final double bd1 = bi1.doubleValue();
               //final double bd2 = ((Double) b).doubleValue();
               final double bd2 = (Double) b;
               return new Double(bd1 + bd2);
          } else {
               return signalWrongType(opName, 2, b, "number");
          }
     } else if ( a instanceof Double ) {
          final Double bd1 = (Double) a;
          if ( b instanceof Double ) {
               final Double bd2 = (Double) b;
               //return new Double(bd1.doubleValue() + bd2.doubleValue());
               return new Double(bd1 + bd2);
          } else if ( b instanceof BigInteger ) {
               final BigInteger bi2 = (BigInteger) b;
               final double bd2 = bi2.doubleValue();
               return new Double(bd1.doubleValue() + bd2);
          } else {
               return signalWrongType(opName, 2, b, "number");
          }
     } else if ( a instanceof String ) {
          final String sa = (String) a;
          if ( b instanceof String ) {
               final String sb = (String) b;
               return sa + sb;
          } else {
               return signalWrongType(opName, 2, b, "string");
          }
     } else {
          return signalWrongType(opName, 1, a, "number");
     }
}

/** Définition de l'opérateur binaire modulo % */

private Object operatorModulo (final String opName,
                               final Object a,
                               final Object b)
     throws EvaluationException {
     checkNotNull(opName, 1, a);
     checkNotNull(opName, 2, b);
     if ( a instanceof BigInteger ) {
          final BigInteger bi1 = (BigInteger) a;
          if ( b instanceof BigInteger ) {
               final BigInteger bi2 = (BigInteger) b;
               return bi1.mod(bi2);
          } else {
               return signalWrongType(opName, 2, b, "integer");
          }
```



```
     } else {
          return signalWrongType(opName, 1, a, "integer");
     }
}

// Les opérateurs binaires (engendrés par PHP)

     /** Définition de l'opérateur binaire -              à l'aide de BigInteger.subtract.
*/

     private Object operatorMinus (final String opName, final Object a, final Object b)
     throws EvaluationException {
          checkNotNull(opName, 1, a);
          checkNotNull(opName, 2, b);
          if ( a instanceof BigInteger ) {
               final BigInteger bi1 = (BigInteger) a;
               if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return bi1.subtract(bi2);
               } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return new Double(bd1 - bd2);
               } else {
                    return signalWrongType(opName, 2, b, "number");
               }
          } else if ( a instanceof Double ) {
               final Double bd1 = (Double) a;
               if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    //return new Double(bd1.doubleValue() - bd2.doubleValue());
                    return new Double(bd1 - bd2);
               } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    final double bd2 = bi2.doubleValue();
                    //return new Double(bd1.doubleValue() - bd2);
                    return new Double(bd1 - bd2);
               } else {
                    return signalWrongType(opName, 2, b, "number");
               }
          } else {
               return signalWrongType(opName, 1, a, "number");
          }
     }

     /** Définition de l'opérateur binaire *              à l'aide de BigInteger.multiply.
*/

     private Object operatorMultiply (final String opName, final Object a, final Object b)
     throws EvaluationException {
          checkNotNull(opName, 1, a);
          checkNotNull(opName, 2, b);
          if ( a instanceof BigInteger ) {
               final BigInteger bi1 = (BigInteger) a;
               if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return bi1.multiply(bi2);
               } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return new Double(bd1 * bd2);
               } else {
                    return signalWrongType(opName, 2, b, "number");
               }
          } else if ( a instanceof Double ) {
               final Double bd1 = (Double) a;
               if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    //return new Double(bd1.doubleValue() * bd2.doubleValue());
                    return new Double(bd1 * bd2);
               } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    final double bd2 = bi2.doubleValue();
                    //return new Double(bd1.doubleValue() * bd2);
                    return new Double(bd1 * bd2);
```

```
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else {
                return signalWrongType(opName, 1, a, "number");
            }
        }

        /** Définition de l'opérateur binaire /              à l'aide de BigInteger.divide.
    */

        private Object operatorQuotient (final String opName, final Object a, final Object b)
        throws EvaluationException {
            checkNotNull(opName, 1, a);
            checkNotNull(opName, 2, b);
            if ( a instanceof BigInteger ) {
                final BigInteger bi1 = (BigInteger) a;
                if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return bi1.divide(bi2);
                } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return new Double(bd1 / bd2);
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else if ( a instanceof Double ) {
                final Double bd1 = (Double) a;
                if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    //return new Double(bd1.doubleValue() / bd2.doubleValue());
                    return new Double(bd1 / bd2);
                } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    final double bd2 = bi2.doubleValue();
                    //return new Double(bd1.doubleValue() / bd2);
                    return new Double(bd1 / bd2);
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else {
                return signalWrongType(opName, 1, a, "number");
            }
        }

    // Les comparateurs binaires (engendrés par PHP)

        /** Définition de l'opérateur binaire LT              à l'aide de BigInteger.compareTo().
    */

        private Object operatorLT (final String opName, final Object a, final Object b)
            throws EvaluationException {
            checkNotNull(opName, 1, a);
            checkNotNull(opName, 2, b);
            if ( a instanceof BigInteger ) {
                final BigInteger bi1 = (BigInteger) a;
                if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return Boolean.valueOf(bi1.compareTo(bi2) < 0);
                } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return Boolean.valueOf(bd1 < bd2);
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else if ( a instanceof Double ) {
                final Double bd1 = (Double) a;
                if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    //return Boolean.valueOf(bd1.doubleValue() < bd2.doubleValue());
                    return Boolean.valueOf(bd1 < bd2);
                } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
```



```
                    final double bd2 = bi2.doubleValue();
                    return Boolean.valueOf(bd1.doubleValue() < bd2);
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else {
                return signalWrongType(opName, 1, a, "number");
            }
        }

        /** Définition de l'opérateur binaire LE              à l'aide de BigInteger.compareTo().
    */

        private Object operatorLE (final String opName, final Object a, final Object b)
            throws EvaluationException {
            checkNotNull(opName, 1, a);
            checkNotNull(opName, 2, b);
            if ( a instanceof BigInteger ) {
                final BigInteger bi1 = (BigInteger) a;
                if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return Boolean.valueOf(bi1.compareTo(bi2) <= 0);
                } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return Boolean.valueOf(bd1 <= bd2);
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else if ( a instanceof Double ) {
                final Double bd1 = (Double) a;
                if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    //return Boolean.valueOf(bd1.doubleValue() <= bd2.doubleValue());
                    return Boolean.valueOf(bd1 <= bd2);
                } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    final double bd2 = bi2.doubleValue();
                    return Boolean.valueOf(bd1.doubleValue() <= bd2);
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else {
                return signalWrongType(opName, 1, a, "number");
            }
        }

        /** Définition de l'opérateur binaire EQ              à l'aide de BigInteger.compareTo().
    */

        private Object operatorEQ (final String opName, final Object a, final Object b)
            throws EvaluationException {
            checkNotNull(opName, 1, a);
            checkNotNull(opName, 2, b);
            if ( a instanceof BigInteger ) {
                final BigInteger bi1 = (BigInteger) a;
                if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return Boolean.valueOf(bi1.compareTo(bi2) == 0);
                } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return Boolean.valueOf(bd1 == bd2);
                } else {
                    return signalWrongType(opName, 2, b, "number");
                }
            } else if ( a instanceof Double ) {
                final Double bd1 = (Double) a;
                if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    return Boolean.valueOf(bd1.doubleValue() == bd2.doubleValue());
                } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    final double bd2 = bi2.doubleValue();
                    return Boolean.valueOf(bd1.doubleValue() == bd2);
                } else {
```

```
                    return signalWrongType(opName, 2, b, "number");
               }
          } else {
               return signalWrongType(opName, 1, a, "number");
          }
     }

     /** Définition de l'opérateur binaire GE              à l'aide de BigInteger.compareTo().
*/

     private Object operatorGE (final String opName, final Object a, final Object b)
          throws EvaluationException {
          checkNotNull(opName, 1, a);
          checkNotNull(opName, 2, b);
          if ( a instanceof BigInteger ) {
               final BigInteger bi1 = (BigInteger) a;
               if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return Boolean.valueOf(bi1.compareTo(bi2) >= 0);
               } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return Boolean.valueOf(bd1 >= bd2);
               } else {
                    return signalWrongType(opName, 2, b, "number");
               }
          } else if ( a instanceof Double ) {
               final Double bd1 = (Double) a;
               if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    //return Boolean.valueOf(bd1.doubleValue() >= bd2.doubleValue());
                    return Boolean.valueOf(bd1 >= bd2);
               } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    final double bd2 = bi2.doubleValue();
                    return Boolean.valueOf(bd1.doubleValue() >= bd2);
               } else {
                    return signalWrongType(opName, 2, b, "number");
               }
          } else {
               return signalWrongType(opName, 1, a, "number");
          }
     }

     /** Définition de l'opérateur binaire GT              à l'aide de BigInteger.compareTo().
*/

     private Object operatorGT (final String opName, final Object a, final Object b)
          throws EvaluationException {
          checkNotNull(opName, 1, a);
          checkNotNull(opName, 2, b);
          if ( a instanceof BigInteger ) {
               final BigInteger bi1 = (BigInteger) a;
               if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    return Boolean.valueOf(bi1.compareTo(bi2) > 0);
               } else if ( b instanceof Double ) {
                    final double bd1 = bi1.doubleValue();
                    //final double bd2 = ((Double) b).doubleValue();
                    final double bd2 = (Double) b;
                    return Boolean.valueOf(bd1 > bd2);
               } else {
                    return signalWrongType(opName, 2, b, "number");
               }
          } else if ( a instanceof Double ) {
               final Double bd1 = (Double) a;
               if ( b instanceof Double ) {
                    final Double bd2 = (Double) b;
                    //return Boolean.valueOf(bd1.doubleValue() > bd2.doubleValue());
                    return Boolean.valueOf(bd1 > bd2);
               } else if ( b instanceof BigInteger ) {
                    final BigInteger bi2 = (BigInteger) b;
                    final double bd2 = bi2.doubleValue();
                    return Boolean.valueOf(bd1.doubleValue() > bd2);
               } else {
                    return signalWrongType(opName, 2, b, "number");
               }
```



```
               } else {
                    return signalWrongType(opName, 1, a, "number");
               }
          }

          /** Définition de l'opérateur binaire NEQ             à l'aide de BigInteger.compareTo().
*/

          private Object operatorNEQ (final String opName, final Object a, final Object b)
               throws EvaluationException {
               checkNotNull(opName, 1, a);
               checkNotNull(opName, 2, b);
               if ( a instanceof BigInteger ) {
                    final BigInteger bi1 = (BigInteger) a;
                    if ( b instanceof BigInteger ) {
                         final BigInteger bi2 = (BigInteger) b;
                         return Boolean.valueOf(bi1.compareTo(bi2) != 0);
                    } else if ( b instanceof Double ) {
                         final double bd1 = bi1.doubleValue();
                         //final double bd2 = ((Double) b).doubleValue();
                         final double bd2 = (Double) b;
                         return Boolean.valueOf(bd1 != bd2);
                    } else {
                         return signalWrongType(opName, 2, b, "number");
                    }
               } else if ( a instanceof Double ) {
                    final Double bd1 = (Double) a;
                    if ( b instanceof Double ) {
                         final Double bd2 = (Double) b;
                         return Boolean.valueOf(bd1.doubleValue() != bd2.doubleValue());
                    } else if ( b instanceof BigInteger ) {
                         final BigInteger bi2 = (BigInteger) b;
                         final double bd2 = bi2.doubleValue();
                         return Boolean.valueOf(bd1.doubleValue() != bd2);
                    } else {
                         return signalWrongType(opName, 2, b, "number");
                    }
               } else {
                    return signalWrongType(opName, 1, a, "number");
               }
          }

}

// fin de génération de CommonPlus.java
```

Java/src/fr/upmc/ilp/ilp1/runtime/ConstantsStuff.java

```
package fr.upmc.ilp.ilp1.runtime;

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

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

public class ConstantsStuff {

    public ConstantsStuff () {}

    /** Étendre un environnement lexical pour l'evaluation pour y
     * installer la constante Pi. */

    public ILexicalEnvironment extendWithPredefinedConstants (
            final ILexicalEnvironment lexenv ) {
      final ILexicalEnvironment lexenv1 = lexenv.extend(
             new IASTvariable() {
               public String getName () {
                 return "pi";
               }
             },
             new Double(3.14159265358979323846264 3) );
      return lexenv1;
    }

    /* NOTA: ce serait bien de regrouper cette methode avec la precedente
     * mais cela induirait une dependance du paquetage runtime (pour
```

```
     * l'evaluation (cf. cours2)) envers la compilation (cf. cours3) ce qui
     * est inutile.
     *
     * Étendre un environnement lexical pour la compilation pour y
     * installer la constante Pi.

    public static ICgenLexicalEnvironment extendWithPredefinedConstants (
            final ICgenLexicalEnvironment lexenv){
        final ICgenLexicalEnvironment lexenv1 = lexenv.extend(
                new IASTvariable () {
                    public String getName () {
                        return "pi";
                    }
                }, "ILP_PI");
        return lexenv1;
    }

    */

}
```

Java/src/fr/upmc/ilp/ilp1/runtime/EmptyLexicalEnvironment.java

```
package fr.upmc.ilp.ilp1.runtime;
import fr.upmc.ilp.ilp1.interfaces.*;

/** Une définition de l'environnement vide. */

public class EmptyLexicalEnvironment
  implements ILexicalEnvironment {

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

  /** Creer un environnement lexical vide.
   * L'environnement vide ne contient rien et signale
   * systématiquement une erreur si l'on cherche la valeur d'une
   * variable.*/

  public static EmptyLexicalEnvironment create () {
    return EmptyLexicalEnvironment.THE_EMPTY_LEXICAL_ENVIRONMENT;
  }

  /** Chercher la valeur d'une variable dans un environnement lexical.
   *
   * @param variable la variable dont la valeur est cherchee
   * @throws EvaluationException si la variable n'a pas de valeur
   */

  public Object lookup (IASTvariable variable)
      throws EvaluationException {
      String msg = "Variable sans valeur: "
        + variable.getName();
      throw new EvaluationException(msg);
    }

  /** On peut étendre l'environnement vide.
   *
   * Malheureusement, cela crée une dépendance avec la classe des
   * environnements non vides et çà c'est moche!                    MOCHE.
   */
  public ILexicalEnvironment extend (IASTvariable variable, Object value) {
    return new LexicalEnvironment(variable, value, this);
  }

}
```

Java/src/fr/upmc/ilp/ilp1/runtime/EvaluationException.java

```
package fr.upmc.ilp.ilp1.runtime;

public class EvaluationException extends Exception {
```



```
        static final long serialVersionUID = +12345678900003000L;

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

        public EvaluationException (Throwable cause) {
                super(cause);
        }

}
```

Java/src/fr/upmc/ilp/ilp1/runtime/ICommon.java

```
package fr.upmc.ilp.ilp1.runtime;

/**
 * Cette interface définit les caractéristiques globales d'un interprète du
 * langage ILP1. On y trouve, notamment, la définition des opérateurs du
 * langage.
 */

public interface ICommon {

    /** Appliquer un opérateur unaire sur un opérande. */
    Object applyOperator(String opName, Object operand)
            throws EvaluationException;

    /** Appliquer un opérateur binaire sur deux opérandes. */
    Object applyOperator(String opName, Object leftOperand, Object rightOperand)
            throws EvaluationException;

}
```

Java/src/fr/upmc/ilp/ilp1/runtime/ILexicalEnvironment.java

```
package fr.upmc.ilp.ilp1.runtime;
import fr.upmc.ilp.ilp1.interfaces.*;

/** 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 {

  /** Renvoie la valeur d'une variable si présente dans
   * l'environnement.
   *
   * @throws EvaluationException si la variable est absente.
   */
  Object lookup (IASTvariable variable)
    throws EvaluationException;

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

}
```

Java/src/fr/upmc/ilp/ilp1/runtime/Invokable.java

```
package fr.upmc.ilp.ilp1.runtime;

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

/** L'interface des fonctions. La plupart des fonctions ont moins de 4
 * arguments aussi des méthodes d'arité diverses sont-elles procurées
 * pour invoquer une fonction avec 0, 1, 2 ou 3 arguments.
 */

public interface Invokable {
```

```
11
    /** Une fonction invoquée avec un nombre quelconque d'arguments. */

    Object invoke (Object[] arguments)
      throws EvaluationException;
16
    /** Invocation d'une fonction zéro-aire (ou niladique) */

    Object invoke ()
      throws EvaluationException;
21
    /** Invocation d'une fonction unaire (ou monadique) */

    Object invoke (Object argument1)
      throws EvaluationException;
26
    /** Invocation d'une fonction binaire (ou dyadique) */

    Object invoke (Object argument1, Object argument2)
      throws EvaluationException;
31
    /** Invocation d'une fonction ternaire */

    Object invoke (Object argument1, Object argument2, Object argument3)
      throws EvaluationException;
36
}
```

Java/src/fr/upmc/ilp/ilp1/runtime/LexicalEnvironment.java

```
package fr.upmc.ilp.ilp1.runtime;
2 import fr.upmc.ilp.ilp1.interfaces.*;

/** Cette implantation d'environnement est très naïve: c'est une
 * simple liste chaînée (mais comme nous n'avons pour l'instant que
 * des blocs unaires cela suffit!).
7 */

public class LexicalEnvironment
implements ILexicalEnvironment {

12   public LexicalEnvironment (final IASTvariable variable,
                                final Object value,
                                final ILexicalEnvironment next )
    {
      this.variableName = variable.getName();
17    this.value = value;
      this.next = next;
    }
    protected final String variableName;
    protected volatile Object value;
22  protected final ILexicalEnvironment next;

    /** Renvoie la valeur d'une variable si présente dans
     * l'environnement.
     */
27  public Object lookup (final IASTvariable variable)
      throws EvaluationException {
      if ( variableName.equals(variable.getName()) ) {
        return value;
      } else {
32      return next.lookup(variable);
      }
    }

    /** On peut étendre tout environnement. */
37  public ILexicalEnvironment extend (final IASTvariable variable,
                                       final Object value) {
      return new LexicalEnvironment(variable, value, this);
    }
}
```

Java/src/fr/upmc/ilp/ilp1/runtime/PrintStuff.java


```
package fr.upmc.ilp.ilp1.runtime;

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

import fr.upmc.ilp.ilp1.interfaces.IASTvariable;
8
/** 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.
13 */

public class PrintStuff {

    private Writer output;
18
    public PrintStuff () {
        this(new StringWriter());
    }
    public PrintStuff (Writer writer) {
23      this.output = writer;
    }

    /** Renvoyer les caractères imprimés. */

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

33  /** étendre un environnement lexical pour l'evaluation pour y installer
     * les primitives print() et newline(). */

    public ILexicalEnvironment
    extendWithPrintPrimitives (final ILexicalEnvironment lexenv) {
38      final ILexicalEnvironment lexenv1 = lexenv.extend(
                new IASTvariable() {
                    public String getName () {
                        return "print";
                    }
43              }, new PrintPrimitive());
        final ILexicalEnvironment lexenv2 = lexenv1.extend(
                new IASTvariable() {
                    public String getName () {
                        return "newline";
48                  }
                }, new NewlinePrimitive());
        return lexenv2;
    }

53  // Cf. NOTA de ConstantsStuff pour l'extension de l'environnement de
    // compilation.

    /** Cette classe implante la fonction print() qui permet d'imprimer
     * une valeur. Elle ne se soucie pas du succès de l'opération! */
58
    protected class PrintPrimitive extends AbstractInvokableImpl {
        public PrintPrimitive () {}
        // La fonction print() est unaire:
        @Override
63      public Object invoke (Object value) {
            try {
                output.append(value.toString());
            } catch (IOException e) {}
            return Boolean.FALSE;
68      }
    }

    /** Cette classe implante la fonction newline() qui permet de passer
     * à la ligne. Elle ne se soucie pas du succès de l'opération! */
73
    protected class NewlinePrimitive extends AbstractInvokableImpl {
        public NewlinePrimitive () {}
        // La fonction newline() est zéro-aire:
        @Override
78      public Object invoke () {
            try {
                output.append("\n");
```

```
            } catch (IOException e) {}
            return Boolean.FALSE;
        }
    }
}
```

Java/src/fr/upmc/ilp/ilp1/eval/EAST.java

```
package fr.upmc.ilp.ilp1.eval;
import fr.upmc.ilp.ilp1.interfaces.*;
import fr.upmc.ilp.ilp1.runtime.*;

public abstract class EAST
implements IAST, IASTEvaluable {

  /** La méthode qui évalue un EAST et retourne sa valeur. Attention:
   * les valeurs sont des objets JAVA (POJO comme l'on dit). */

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

  /** Un programme qui calcule n'importe quoi. C'est, par exemple,
   * utilisé pour les alternatives binaires.
   *
   * NOTA: une fois que ce n'importe-quoi est déterminé, ici, il ne
   * change plus!
   */

  public static EAST voidConstant () {
    return THE_VOID_CONSTANT;
  }
  private static final EAST THE_VOID_CONSTANT;
  static {
    THE_VOID_CONSTANT = new EASTboolean("false");
  }

  /** Rendre une valeur quelconque. Ici c'est la valeur de
   * voidConstant(lexenv, common) qui a été choisie.
   */
  public static Object voidConstantValue () {
      return Boolean.FALSE;
  }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTConstant.java

```
package fr.upmc.ilp.ilp1.eval;
import fr.upmc.ilp.ilp1.interfaces.IASTconstant;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

/** La classe abstraite des constantes. Elles partagent un même
 * comportement à savoir rendre leur propre valeur (un objet Java).
 *
 * Attention, même si le code de la méthode eval serait le même dans
 * les sous-classes de EASTConstant, le type de valeur est varié et ne
 * peut être , sans précaution, partagé! Noter le abstract de la
 * classe, le protected du constructeur. Noter le protected final sur
 * valueAsObject (explication en EASTentier).
 */

public abstract class EASTConstant extends EAST implements IASTconstant {

  protected EASTConstant (Object value) {
    this.valueAsObject = value;
  }
  protected final Object valueAsObject;

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

  @Override
  public Object eval (ILexicalEnvironment lexenv, ICommon common) {
    return valueAsObject;
  }
}
```


Java/src/fr/upmc/ilp/ilp1/eval/EASTException.java

```
package fr.upmc.ilp.ilp1.eval;

/**
 * Une exception comme les autres mais qu'utilisent préférentiellement les
 * classes du paquetage eval.
 */

public class EASTException extends Exception {

    static final long serialVersionUID = +12345678900020000L;

    public EASTException(Throwable cause) {
        super(cause);
    }

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

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTFactory.java

```
package fr.upmc.ilp.ilp1.eval;
import java.util.List;

/** Une fabrique pour fabriquer des EAST. */

public class EASTFactory implements IEASTFactory<EASTException> {

  public EASTprogram newProgram (List<EAST> asts) {
      return new EASTprogram(asts);
  }

  /** Créer une séquence d'AST. */
  public EASTsequence newSequence (List<EAST> asts) {
    return new EASTsequence(asts);
  }

  /** Créer une alternative binaire. */
  public EASTalternative newAlternative (
          EAST condition,
          EAST consequent) {
    return new EASTalternative(condition, consequent);
  }

  /** Créer une alternative ternaire. */
  public EASTalternative newAlternative (
          EAST condition,
          EAST consequent,
          EAST alternant) {
    return new EASTalternative(condition, consequent, alternant);
  }

  /** Créer un bloc local unaire (avec une seule variable locale). */
  public EASTunaryBlock newUnaryBlock (
          EASTvariable variable,
          EAST initialisation,
          EASTsequence body) {
    return new EASTunaryBlock(variable, initialisation, body);
  }

  /** Créer une variable. */
  public EASTvariable newVariable (String name) {
    return new EASTvariable(name);
  }

  /** Créer une invocation (un appel à une fonction). */
  public EASTinvocation newInvocation (String name, List<EAST> asts) {
    return new EASTinvocationPrimitive(name, asts);
  }

  /** Créer une opération unaire. */
  public EASTunaryOperation newUnaryOperation (String operatorName,
                                               EAST operand) {
```

```
    return new EASTunaryOperation(operatorName, operand);
  }

  /** Créer une opération binaire. */
  public EASTbinaryOperation newBinaryOperation (String operatorName,
                                                 EAST leftOperand,
                                                 EAST rightOperand) {
    return new EASTbinaryOperation(operatorName, leftOperand, rightOperand);
  }

  /** Créer une constante littérale entière. */
  public EASTinteger newIntegerConstant (String value) {
    return new EASTinteger(value);
  }

  /** Créer une constante littérale flottante. */
  public EASTfloat newFloatConstant (String value) {
    return new EASTfloat(value);
  }

  /** Créer une constante littérale chaîne de caractères. */
  public EASTstring newStringConstant (String value) {
    return new EASTstring(value);
  }

  /** Créer une constante littérale booléenne. */
  public EASTboolean newBooleanConstant (String value) {
    return new EASTboolean(value);
  }

  /** Signaler un problème avec un message. */
  public EAST throwParseException (String message)
    throws EASTException {
    throw new EASTException(message);
  }

  /** Signaler un problème avec une exception. */
  public EAST throwParseException (Throwable cause)
    throws EASTException {
    throw new EASTException(cause);
  }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTParser.java

```
package fr.upmc.ilp.ilp1.eval;

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

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

/** Le but de cette classe est de transformer un document XML en un
 * AST avec évaluation conforme à fr.upmc.ilp.interfaces.IAST.
 *
 * Cet analyseur améliore ASTParser en ce sens qu'il est paramétré par
 * une fabrique de construction d'AST. À part sa construction, cet
 * analyseur s'utilise tout comme un ASTParser.
 */

public class EASTParser {

    /** Constructeur d'analyseur de document (ou noeud) DOM en un AST
     * dont les constructeurs particuliers sont fournis par une
     * fabrique. */

    public EASTParser (final IEASTFactory<EASTException> factory) {
        this.factory = factory;
    }

    protected final IEASTFactory<EASTException> factory;

    /** Analyseur de DOM en AST. Les noeuds de l'AST sont créés par la
     * fabrique spécifiée à la construction de l'EASTParser. Même les
     * exceptions sont signalées par cette même fabrique.
     */

    public EAST parse (final Node n)
    throws EASTException {
        try {
            switch ( n.getNodeType() ) {

            case Node.DOCUMENT_NODE: {
                final Document d = (Document) n;
                return this.parse(d.getDocumentElement());
            }

            case Node.ELEMENT_NODE: {
                final Element e = (Element) n;
                final NodeList nl = e.getChildNodes();
                final String name = e.getTagName();
                switch (name) {
                case "programme1": {
                    return factory.newProgram(parseList(nl));
                }
                case "alternative": {
                    final EAST cond = findThenParseChild(nl, "condition");
                    final EAST conseq =
                        findThenParseChildAsInstructions(nl, "consequence");
                    try {
                        final EAST alt =
                            findThenParseChildAsInstructions(nl, "alternant");
                        return factory.newAlternative(cond, conseq, alt);
                    } catch (EASTException exc) {
                        return factory.newAlternative(cond, conseq);
                    }
                }
                case "sequence": {
                    return factory.newSequence(this.parseList(nl));
                }
                case "blocUnaire": {
                    final EASTvariable var =
                        (EASTvariable) findThenParseChild(nl, "variable");
                    final EAST init = findThenParseChild(nl, "valeur");
                    final EASTsequence body =
                        (EASTsequence) findThenParseChild(nl, "corps");
                    return factory.newUnaryBlock(var, init, body);
                }
                case "variable": {
                    final String nick = e.getAttribute("nom");
                    return factory.newVariable(nick);
                }
                case "invocationPrimitive": {
                    final String op = e.getAttribute("fonction");
                    final List<EAST> args = parseList(nl);
                    return factory.newInvocation(op, args);
                }
                case "operationUnaire": {
                    final String op = e.getAttribute("operateur");
                    final EAST rand = findThenParseChild(nl, "operande");
                    return factory.newUnaryOperation(op, rand);
                }
                case "operationBinaire": {
                    final String op = e.getAttribute("operateur");
                    final EAST gauche = findThenParseChild(nl, "operandeGauche");
                    final EAST droite = findThenParseChild(nl, "operandeDroit");
                    return factory.newBinaryOperation(op, gauche, droite);
                }
                case "entier": {
                    return factory.newIntegerConstant(e.getAttribute("valeur"));
                }
                case "flottant": {
                    return factory.newFloatConstant(e.getAttribute("valeur"));
                }
                case "chaine": {
                    final String text = n.getTextContent();
                    //final String text = this.extractText(e);
                    return factory.newStringConstant(text);
                }
                case "booleen": {
                    return factory.newBooleanConstant(e.getAttribute("valeur"));
                }
                // Une série d'éléments devant être analysés comme des expressions:
```

```
                case "operandeGauche":
                case "operandeDroit":
                case "operande":
                case "condition":
                case "consequence":
                case "alternant":
                case "valeur": {
                    return this.parseUniqueChild(nl);
                }
                // Une série d'éléments devant être analysée comme une séquence:
                case "corps": {
                    return factory.newSequence(this.parseList(nl));
                }
                default: {
                    final String msg = "Unknown element name: " + name;
                    return factory.throwParseException(msg);
                }
            }
            }

            default: {
                final String msg = "Unknown node type: " + n.getNodeName();
                return factory.throwParseException(msg);
            }
            }
        } catch (final EASTException e) {
            throw e;
        } catch (final Exception e) {
            throw new EASTException(e);
        }
    }

    /** Analyser une séquence d'éléments pour en faire un ASTlist
     * c'est-à-dire une séquence d'AST.
     */

    public List<EAST> parseList (final NodeList nl)
    throws EASTException {
        final List<EAST> 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 EAST p = this.parse(nd);
                    result.add(p);
                    continue LOOP;
                }

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

    /** Trouver un élément d'après son nom et l'analyser pour en faire
     * un AST.  */

    public EAST findThenParseChild (final NodeList nl, final String childName)
    throws EASTException {
        EAST result = null;
        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()) ) {
                    if ( result == null ) {
                        result = this.parse(e);
                    } else {
                        final String msg = "Non unique child with name " + childName;
                        return factory.throwParseException(msg);
```



```
                    }
                }
                break;
            }

            default: {
                // On ignore tout ce qui n'est pas élément XML:
            }
            }
        }
        if ( result == null ) {
            final String msg = "No such child element " + childName;
            return factory.throwParseException(msg);
        }
        return result;
    }

    /** Trouver un élément d'après son nom et analyser son contenu pour
     * en faire un ASTsequence.
     *
     * @throws ILPException
     *   lorsqu'un tel élément n'est pas trouvé.
     */

    public EAST findThenParseChildAsInstructions (
            final NodeList nl,
            final String childName)
    throws EASTException {
        int n = nl.getLength();
        for ( int i = 0 ; i<n ; i++ ) {
            Node nd = nl.item(i);
            switch ( nd.getNodeType() ) {

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

            default: {
                // On ignore tout ce qui n'est pas élément XML:
            }
            }
        }
        String msg = "No such child element " + childName;
        return factory.throwParseException(msg);
    }

    /** Analyser une suite comportant un unique élément.  */

    public EAST parseUniqueChild (final NodeList nl)
    throws EASTException {
        EAST result = null;
        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 ( result == null ) {
                    result = this.parse(e);
                } else {
                    final String msg = "Non unique child";
                    return factory.throwParseException(msg);
                }
                break;
            }

            default: {
                // On ignore tout ce qui n'est pas élément XML:
            }
            }
        }
        if ( result == null ) {
            factory.throwParseException("No child at all");
```

```
        }
        return result;
    }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTalternative.java

```
package fr.upmc.ilp.ilp1.eval;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.interfaces.IASTalternative;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

public class EASTalternative extends EAST
implements IASTalternative {

  public EASTalternative(EAST condition, EAST consequence, EAST alternant) {
    this.condition = condition;
    this.consequence = consequence;
    this.alternant = alternant;
  }

  // NOTA: Masquer l'implantation de l'alternative binaire afin
  // d'éviter la propagation de null.

  public EASTalternative(EAST condition, EAST consequence) {
     this(condition, consequence, null);
  }

  protected EAST condition;
  protected EAST consequence;
  protected EAST alternant;

  public EAST getCondition() {
    return this.condition;
  }

  public EAST getConsequent() {
    return this.consequence;
  }

  /** Quand il n'y a pas d'alternant, on rend n'importe quoi. Ce
   * n'importe-quoi est engendré par un utilitaire partagé au niveau d'AST. */

  @OrNull public EAST getAlternant() {
    return this.alternant;
  }

  public boolean isTernary() {
    return alternant != null;
  }

  @Override
    public Object eval(ILexicalEnvironment lexenv, ICommon common)
    throws EvaluationException {
    Object bool = condition.eval(lexenv, common);
    if (Boolean.FALSE == bool) {
      if (isTernary()) {
        return alternant.eval(lexenv, common);
      } else {
        // Et pas EAST.voidConstant() comme signalé par
        // <bourgerie.quentin@gmail.com>
        return EAST.voidConstantValue();
      }
    } else {
      return consequence.eval(lexenv, common);
    }
  }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTbinaryOperation.java


```
package fr.upmc.ilp.ilp1.eval;

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

import fr.upmc.ilp.ilp1.interfaces.IASTbinaryOperation;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

/** Les operations binaires. */

public class EASTbinaryOperation extends EASToperation
implements IASTbinaryOperation {

  public EASTbinaryOperation (String operateur,
                              EAST operandeGauche,
                              EAST operandeDroit)
  {
    super(operateur, 2);
    this.operandeGauche = operandeGauche;
    this.operandeDroit = operandeDroit;
  }

  protected EAST operandeGauche;
  protected EAST operandeDroit;

  public EAST getLeftOperand () {
    return this.operandeGauche;
  }
  public EAST getRightOperand () {
    return this.operandeDroit;
  }

  public EAST[] getOperands () {
    // On calcule paresseusement car ce n'est pas une methode usuelle:
    if ( operands == null ) {
        List<EAST> loperands = new Vector<>();
        loperands.add(operandeGauche);
        loperands.add(operandeDroit);
        operands = loperands.toArray(new EAST[0]);
    }
    return operands;
  }
  // NOTA: remarquer que ce mode paresseux interdit de qualifier ce
  // champ de « final » ce qui n'est pas sûr!
  private EAST[] operands;

  @Override
  public Object eval (ILexicalEnvironment lexenv, ICommon common)
    throws EvaluationException {
    Object r1 = operandeGauche.eval(lexenv, common);
    Object r2 = operandeDroit.eval(lexenv, common);
    return common.applyOperator(this.getOperatorName(), r1, r2);
  }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTboolean.java

```
package fr.upmc.ilp.ilp1.eval;
import fr.upmc.ilp.ilp1.interfaces.IASTboolean;

/** Une constante booleenne. */

public class EASTboolean extends EASTConstant implements IASTboolean {

  /** Nota: Encore un cas ennuyeux où l'appel à super() n'est pas
   * trivial. Il y a quelques années, le vérifieur de code-octet avait
   * tendance à refuser ce genre de programmation! */

  public EASTboolean (String valeur) {
    super(  ("true".equals(valeur))
            ? Boolean.TRUE : Boolean.FALSE );
    this.valeur = "true".equals(valeur);
  }
  private final boolean valeur;
```

```
  public boolean getValue () {
    return valeur;
  }
}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTfloat.java

```
package fr.upmc.ilp.ilp1.eval;
import java.math.BigDecimal;

import fr.upmc.ilp.ilp1.interfaces.IASTfloat;

/** Les constantes flottantes. */

public class EASTfloat extends EASTConstant implements IASTfloat {

  /** Constructeur.
   *
   * Ici, le double est pris comme une chaîne puis manipulé comme un
   * Double (en guise d'objet Java) pour l'évaluation. En revanche
   * pour les IAST, il est manipulé comme un BigDecimal pour ne pas
   * subir de dégradation de précision. L'implantation perd donc de la
   * précision par rapport à la syntaxe!
   */

  public EASTfloat (String valeur) {
    super(new Double(valeur));
    this.bigfloat = new BigDecimal(valeur);
  }
  private final BigDecimal bigfloat;

  public BigDecimal getValue () {
    return bigfloat;
  }
}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTinteger.java

```
package fr.upmc.ilp.ilp1.eval;
import java.math.BigInteger;

import fr.upmc.ilp.ilp1.interfaces.IASTinteger;

/** Les constantes entières littérales. Elles sont représentées par
 * des BigInteger ce qui permet de ne pas rendre syntaxiquement faux
 * un programme qui mentionnerait des entiers plus grand que ce que
 * sait faire la machine. En outre, cela permet de donner une
 * arithmétique complète à cette implantation.
 */

public class EASTinteger extends EASTConstant implements IASTinteger {

  /** Constructeur.
   *
   * NOTA: Pour initialiser le champ valueAsObject, on fait le bon
   * appel à super() mais comme un BigInteger est un gros objet que
   * l'on ne tient pas à dupliquer, on l'utilise tel qu'il apparaît
   * dans EASTConstant. Conclusion: valueAsObject ne peut plus être
   * privée à EASTConstant et voilà comment les « protected »
   * croissent dans les codes!
   */

  public EASTinteger (String valeur) {
    super(new BigInteger(valeur));
  }

  public BigInteger getValue () {
    return (BigInteger) this.valueAsObject;
  }
}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTinvocation.java


```
package fr.upmc.ilp.ilp1.eval;

import java.util.List;

import fr.upmc.ilp.ilp1.interfaces.IASTinvocation;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;
import fr.upmc.ilp.ilp1.runtime.Invokable;

/** La classe abstraite des invocations en general. */

public abstract class EASTinvocation extends EAST
implements IASTinvocation {

  protected EASTinvocation (final EAST fonction, final List<EAST> arguments) {
    this.fonction = fonction;
    this.argument = arguments.toArray(new EAST[0]);
  }
  protected EAST     fonction;
  protected EAST[]   argument;

  public EAST getFunction () {
    return this.fonction;
  }

  public EAST[] getArguments () {
    return this.argument;
  }

  public int getArgumentsLength () {
    return this.argument.length;
  }

  public EAST getArgument (final int i) {
    return this.argument[i];
  }

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

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTinvocationPrimitive.java

```
package fr.upmc.ilp.ilp1.eval;

import java.util.List;

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

/** L'invocation des primitives. */

public class EASTinvocationPrimitive
extends EASTinvocation {

  public EASTinvocationPrimitive (final String fonction,
                                  final List<EAST> arguments) {
    super(new EASTvariable(fonction), arguments);
  }
```

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

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASToperation.java

```
package fr.upmc.ilp.ilp1.eval;

import fr.upmc.ilp.ilp1.interfaces.IASToperation;

/** La classe abstraite des operations.
 *
 * NOTA: Notez que la méthode getOperands() manque alors qu'elle est
 * présente dans l'interface (IASToperation) que cette classe
 * implante! Savez-vous pourquoi ?
 */

public abstract class EASToperation extends EAST
implements IASToperation
{

  protected EASToperation (final String operateur, final int arity) {
    this.operateur = operateur;
    this.arity = arity;
  }
  private final String operateur;
  private final int arity;

  // et pourquoi les final ici ?
  public final String getOperatorName () {
    return this.operateur;
  }

  public final int getArity () {
    return this.arity;
  }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTprogram.java

```
package fr.upmc.ilp.ilp1.eval;

import java.util.List;

import fr.upmc.ilp.ilp1.interfaces.IASTprogram;
import fr.upmc.ilp.ilp1.interfaces.IASTsequence;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

public class EASTprogram extends EAST implements IASTprogram {
    public EASTprogram (List<EAST> instructions) {
        this.body = new EASTsequence(instructions);
    }
    protected EASTsequence body;

    public IASTsequence getBody() {
        return body;
    }

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

Java/src/fr/upmc/ilp/ilp1/eval/EASTsequence.java


```
package fr.upmc.ilp.ilp1.eval;

import java.util.List;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.interfaces.IASTsequence;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

/** Les sequences d'instructions. Les grammaires imposent normalement
 * qu'il y a au moins une instruction dans une sequence.  */

public class EASTsequence extends EAST
implements IASTsequence {

    public EASTsequence (List<EAST> instructions) {
        this.instruction = instructions.toArray(new EAST[0]);
    }
    protected EAST[] instruction;

    public EAST[] getInstructions () {
        return this.instruction;
    }
    public @OrNull EAST getInstruction (int i) {
        return this.instruction[i];
    }
    public int getInstructionsLength () {
        return this.instruction.length;
    }

    /**
     * L'évaluation d'une séquence passe par celle, ordonnée, de toutes
     * les instructions qu'elle contient.
     *
     * NOTA: inutile de se compliquer la vie, Java ne supporte pas la
     * récursion terminale.
     * NOTA2: Le cas de la séquence vide est prévu.
     */

    @Override
    public Object eval (ILexicalEnvironment lexenv, ICommon common)
    throws EvaluationException {
        Object last = EAST.voidConstantValue();
        for ( int i = 0 ; i < instruction.length ; i++ ) {
            last = instruction[i].eval(lexenv, common);
        }
        return last;
    }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTstring.java

```
package fr.upmc.ilp.ilp1.eval;
import fr.upmc.ilp.ilp1.interfaces.IASTstring;

/** Les constantes de type chaine de caracteres. */

public class EASTstring extends EASTConstant implements IASTstring {

  public EASTstring (String valeur) {
    super(valeur);
    this.valeur = valeur;
  }
  private final String valeur;

  public String getValue () {
    return valeur;
  }
}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTunaryBlock.java

```java
package fr.upmc.ilp.ilp1.eval;

import fr.upmc.ilp.ilp1.interfaces.IASTunaryBlock;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

/** Les blocs locaux à une seule variable. */

public class EASTunaryBlock extends EAST implements IASTunaryBlock {

  public EASTunaryBlock (EASTvariable variable,
                         EAST initialization,
                         EASTsequence body)
  {
    this.variable       = variable;
    this.initialization = initialization;
    this.body           = body;
  }

  protected EASTvariable variable;
  protected EAST         initialization;
  protected EASTsequence body;

  public EASTvariable getVariable () {
    return this.variable;
  }
  public EAST getInitialization () {
    return this.initialization;
  }
  public EASTsequence getBody () {
    return this.body;
  }

  @Override
  public Object eval (ILexicalEnvironment lexenv, ICommon common)
    throws EvaluationException {
    ILexicalEnvironment newlexenv =
      lexenv.extend(variable, initialization.eval(lexenv, common));
    return body.eval(newlexenv, common);
  }

}
```

Java/src/fr/upmc/ilp/ilp1/eval/EASTunaryOperation.java

```java
package fr.upmc.ilp.ilp1.eval;

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

import fr.upmc.ilp.ilp1.interfaces.IASTunaryOperation;
import fr.upmc.ilp.ilp1.runtime.EvaluationException;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;

/** Les operations unaires. */

public class EASTunaryOperation extends EASToperation
implements IASTunaryOperation {

  public EASTunaryOperation (String operateur, EAST operand) {
    super(operateur, 1);
    this.operand = operand;
  }
  protected EAST operand;

  public EAST getOperand () {
    return this.operand;
  }

  public EAST[] getOperands () {
    // On calcule paresseusement car ce n'est pas une methode usuelle:
    if ( operands == null ) {
        List<EAST> loperands = new Vector<>();
        loperands.add(operand);
        operands = loperands.toArray(new EAST[0]);
    }
    return operands;
  }
  private EAST[] operands;

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

}
```


Java/src/fr/upmc/ilp/ilp1/eval/EASTvariable.java

```java
package fr.upmc.ilp.ilp1.eval;

import fr.upmc.ilp.ilp1.interfaces.*;
import fr.upmc.ilp.ilp1.runtime.*;

/** Une variable. */

public class EASTvariable extends EAST implements IASTvariable {

  public EASTvariable (String name) {
    this.name = name;
  }
  private final String name;

  public String getName () {
    return name;
  }

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

Java/src/fr/upmc/ilp/ilp1/eval/IASTEvaluable.java

```java
package fr.upmc.ilp.ilp1.eval;

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

public interface IASTEvaluable {

    /** La méthode qui évalue un IAST et retourne sa valeur. Attention:
     * les valeurs sont des objets JAVA (POJO comme l'on dit). */

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

}
```

Java/src/fr/upmc/ilp/ilp1/eval/IEASTFactory.java

```java
package fr.upmc.ilp.ilp1.eval;
import java.util.List;

/** Cette interface décrit une fabrique d'AST conforme aux IAST. Elle
 * est notamment utilisée par le convertisseur générique de DOM en
 * IAST (eval/EASTParser). Le resultat appartient a la famille des EAST.
 */

public interface IEASTFactory<Exc extends Exception> {

  /** créer un nouveau programme. */
  EASTprogram newProgram(List<EAST> asts);
```

```
    /** Créer une séquence d'AST. */
    EASTsequence newSequence (List<EAST> asts);

    /** Créer une alternative binaire. */
    EASTalternative newAlternative (EAST condition,
                                    EAST consequent);

    /** Créer une alternative ternaire. */
    EASTalternative newAlternative (EAST condition,
                                    EAST consequent,
                                    EAST alternant);

    /** Créer un bloc local unaire (avec une seule variable locale). */
    EASTunaryBlock newUnaryBlock (EASTvariable variable,
                                  EAST initialisation,
                                  EASTsequence body);

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

    /** Créer une invocation (un appel à une fonction). */
    EASTinvocation newInvocation (String name, List<EAST> asts);

    /** Créer une opération unaire. */
    EASTunaryOperation newUnaryOperation (String operatorName,
                                          EAST operand);
    /** Créer une opération binaire. */
    EASTbinaryOperation newBinaryOperation (String operatorName,
                                            EAST leftOperand,
                                            EAST rightOperand);

    /** Créer une constante littérale entière. */
    EASTinteger newIntegerConstant (String value);

    /** Créer une constante littérale flottante. */
    EASTfloat newFloatConstant (String value);

    /** Créer une constante littérale chaîne de caractères. */
    EASTstring newStringConstant (String value);

    /** Créer une constante littérale booléenne. */
    EASTboolean newBooleanConstant (String value);

    /** Signaler un problème avec un message. */
    EAST throwParseException (String message) throws Exc;

    /** Signaler un problème avec une exception. */
    EAST throwParseException (Throwable cause) throws Exc;


    /* NOTA: on ne peut exprimer en Java que ces deux dernières méthodes
     * signalent toujours une exception et ne renvoient donc jamais une
     * valeur. Comme on ne peut écrire ni:
     *          factory.throwParseException("blabla");
     *  [car alors la méthode englobante manque d'un return]
     *                                  ni:
     *          return factory.throwParseException("blabla");
     *  [car on ne peut renvoyer void]
     * je change le type de retour pour EAST. */
}
```

Java/src/fr/upmc/ilp/ilp1/cgen/CgenEnvironment.java

```
package fr.upmc.ilp.ilp1.cgen;

import java.util.*;

import fr.upmc.ilp.ilp1.interfaces.*;

/** La représentation de l'environnement des opérateurs prédéfinis. Il
 * définit comment les compiler. C'est un peu l'analogue de
 * runtime/Common pour le paquetage cgen. */

public class CgenEnvironment
  implements ICgenEnvironment {

  private static final Map<String,String> MAP_OP1 = new HashMap<>();
  private static final Map<String,String> MAP_OP2 = new HashMap<>();

  static {
    // Binaires:
    MAP_OP2.put("+", "Plus");
    MAP_OP2.put("-", "Minus");
    MAP_OP2.put("*", "Times");
    MAP_OP2.put("/", "Divide");
    MAP_OP2.put("%", "Modulo");
    MAP_OP2.put("<", "LessThan");
    MAP_OP2.put("<=", "LessThanOrEqual");
    MAP_OP2.put("==", "Equal");
    MAP_OP2.put(">=", "GreaterThanOrEqual");
    MAP_OP2.put(">", "GreaterThan");
    MAP_OP2.put("!=", "NotEqual");
    // Unaires:
    MAP_OP1.put("-", "Opposite");
    MAP_OP1.put("!", "Not");
  }

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

  @Override
  public String  compileOperator1 (final String opName)
    throws CgenerationException {
    return compileOperator(opName, MAP_OP1);
  }

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

  @Override
  public String compileOperator2 (final String opName)
    throws CgenerationException {
    return compileOperator(opName, MAP_OP2);
  }

  /** Méthode interne pour trouver le nom en C d'un opérateur. */

  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 operator: " + opName;
      throw new CgenerationException(msg);
    }
  }

  /** Compiler une référence à une variable. */

  @Override
  public IASTvariable generateVariable () {
    CgenEnvironment.counter++;
    return CgenEnvironment.createVariable("TEMP" + CgenEnvironment.counter);
  }
  private transient static int counter = 0;

  private static IASTvariable createVariable (final String name) {
    return new IASTvariable () {
        @Override
        public String getName () {
          return name;
        }
      };
  }

  /** Enrichir un environnement lexical avec les primitives
   * d'impression (print et newline). */

  @Override
  public ICgenLexicalEnvironment extendWithPrintPrimitives (
          final ICgenLexicalEnvironment lexenv) {
    final ICgenLexicalEnvironment lexenv2 =
      lexenv.extend(CgenEnvironment.createVariable("print"),
                    "ILP_print");
    final ICgenLexicalEnvironment lexenv3 =
      lexenv2.extend(CgenEnvironment.createVariable("newline"),
                     "ILP_newline");
    return lexenv3;
  }
```

```
  @Override
public ICgenLexicalEnvironment extendWithPredefinedConstants (
          final ICgenLexicalEnvironment lexenv) {
        final ICgenLexicalEnvironment lexenv2 =
          lexenv.extend(CgenEnvironment.createVariable("pi"),
                        "ILP_PI");
        return lexenv2;
      }

}
```

Java/src/fr/upmc/ilp/ilp1/cgen/CgenLexicalEnvironment.java

```
package fr.upmc.ilp.ilp1.cgen;

import fr.upmc.ilp.ilp1.interfaces.*;

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

public class CgenLexicalEnvironment
  implements ICgenLexicalEnvironment {

  public CgenLexicalEnvironment (final IASTvariable variable,
                                 final String compiledName,
                                 final ICgenLexicalEnvironment next) {
    this.variableName = variable.getName();
    this.compiledName = compiledName;
    this.next = next;
  }
  private final String variableName;
  private final String compiledName;
  private final ICgenLexicalEnvironment next;

  @Override
public String compile (final IASTvariable variable)
    throws CgenerationException {
    if ( variableName.equals(variable.getName()) ) {
      return compiledName;
    } else {
      return next.compile(variable);
    }
  }

  @Override
public ICgenLexicalEnvironment extend (final IASTvariable variable,
                                       final String compiledName) {
    return new CgenLexicalEnvironment(variable, compiledName, this);
  }

  @Override
public ICgenLexicalEnvironment extend (final IASTvariable variable) {
    return new CgenLexicalEnvironment(variable, variable.getName(), this);
  }

  // Classe interne:
  public static class Empty
  extends CgenLexicalEnvironment {

      // La technique du singleton:
      private Empty () {
          super(new IASTvariable() {
              @Override
            public String getName() { return null; }
          }, null, null);
      }
      private static final Empty
        EMPTY_LEXICAL_ENVIRONMENT;
      static {
        EMPTY_LEXICAL_ENVIRONMENT = new Empty();
      }

      public static ICgenLexicalEnvironment create () {
        return Empty.EMPTY_LEXICAL_ENVIRONMENT;
      }
```



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

}
```

Java/src/fr/upmc/ilp/ilp1/cgen/CgenerationException.java

```
package fr.upmc.ilp.ilp1.cgen;

/** Les exceptions préférentiellement signalées par la conversion en C. */

public class CgenerationException extends Exception {

    static final long serialVersionUID = +12345678900040000L;

  public CgenerationException (Throwable cause) {
    super(cause);
  }

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

}
```

Java/src/fr/upmc/ilp/ilp1/cgen/Cgenerator.java

```
package fr.upmc.ilp.ilp1.cgen;

import java.math.BigInteger;

import fr.upmc.ilp.ilp1.interfaces.IAST;
import fr.upmc.ilp.ilp1.interfaces.IASTalternative;
import fr.upmc.ilp.ilp1.interfaces.IASTbinaryOperation;
import fr.upmc.ilp.ilp1.interfaces.IASTboolean;
import fr.upmc.ilp.ilp1.interfaces.IASTconstant;
import fr.upmc.ilp.ilp1.interfaces.IASTfloat;
import fr.upmc.ilp.ilp1.interfaces.IASTinteger;
import fr.upmc.ilp.ilp1.interfaces.IASTinvocation;
import fr.upmc.ilp.ilp1.interfaces.IASToperation;
import fr.upmc.ilp.ilp1.interfaces.IASTprogram;
import fr.upmc.ilp.ilp1.interfaces.IASTsequence;
import fr.upmc.ilp.ilp1.interfaces.IASTstring;
import fr.upmc.ilp.ilp1.interfaces.IASTunaryBlock;
import fr.upmc.ilp.ilp1.interfaces.IASTunaryOperation;
import fr.upmc.ilp.ilp1.interfaces.IASTvariable;

/** La génération vers C. */

public class Cgenerator {

  public Cgenerator (final ICgenEnvironment common) {
    this.common = common;
  }
  protected final ICgenEnvironment common;

  /** Convertir un AST en une chaîne de caractères C. */

  public String compile (final IAST iast,
                         final ICgenLexicalEnvironment lexenv,
                         final String destination)
    throws CgenerationException {
    this.buffer = new StringBuffer(1024);
    buffer.append("/* Fichier compilé vers C */\n");
    analyze(iast, lexenv, this.common, destination);
    final String result = this.buffer.toString();
    return result;
  }
  protected transient StringBuffer buffer;
```

```
  /** Convertir une expression en C. */

  protected void analyzeExpression (final IAST iast,
                                    final ICgenLexicalEnvironment lexenv,
                                    final ICgenEnvironment common)
    throws CgenerationException {
    analyze(iast, lexenv, common, "");
  }

  /** Convertir une instruction en C.
   *
   * La destination est soit "" (ou "(void)" pour indiquer que la
   * valeur obtenue est inutile, soit "return" pour indiquer que c'est
   * la valeur finale d'une fonction ou encore "variable =" pour
   * ranger la valeur obtenue dans une variable.
   */

  protected void analyzeInstruction (final IAST iast,
                                     final ICgenLexicalEnvironment lexenv,
                                     final ICgenEnvironment common,
                                     final String destination)
    throws CgenerationException {
    analyze(iast, lexenv, common, destination);
  }

  /** Cette méthode analyse la nature de l'AST à traiter et détermine
   * la bonne méthode à appliquer. C'est un envoi de message simulé à
   * la main, l'ordre de discrimination n'est pas le plus efficace,
   * mais utilise les marqueurs d'interface pour regrouper certains
   * tests. Le code spécifique aux divers cas se trouve dans les
   * méthodes generate() surchargées.
   */

  protected void analyze (final IAST iast,
                          final ICgenLexicalEnvironment lexenv,
                          final ICgenEnvironment common,
                          final String destination)
    throws CgenerationException {
    if ( iast instanceof IASTconstant ) {
      if ( iast instanceof IASTboolean ) {
        generate((IASTboolean) iast, lexenv, common, destination);
      } else if ( iast instanceof IASTfloat ) {
        generate((IASTfloat) iast, lexenv, common, destination);
      } else if ( iast instanceof IASTinteger ) {
        generate((IASTinteger) iast, lexenv, common, destination);
      } else if ( iast instanceof IASTstring ) {
        generate((IASTstring) iast, lexenv, common, destination);
      } else {
        final String msg = "Unknown type of constant: " + iast;
        throw new CgenerationException(msg);
      }
    } else if ( iast instanceof IASTalternative ) {
      generate((IASTalternative) iast, lexenv, common, destination);
    } else if ( iast instanceof IASTinvocation ) {
      generate((IASTinvocation) iast, lexenv, common, destination);
    } else if ( iast instanceof IASToperation ) {
      if ( iast instanceof IASTunaryOperation ) {
        generate((IASTunaryOperation) iast, lexenv, common, destination);
      } else if ( iast instanceof IASTbinaryOperation ) {
        generate((IASTbinaryOperation) iast, lexenv, common, destination);
      } else {
        final String msg = "Unknown type of operation: " + iast;
        throw new CgenerationException(msg);
    }
    } else if ( iast instanceof IASTsequence ) {
      generate((IASTsequence) iast, lexenv, common, destination);
    } else if ( iast instanceof IASTunaryBlock ) {
      generate((IASTunaryBlock) iast, lexenv, common, destination);
    } else if ( iast instanceof IASTvariable ) {
      generate((IASTvariable) iast, lexenv, common, destination);
    } else if ( iast instanceof IASTprogram ) {
        generate((IASTprogram) iast, lexenv, common);
    } else {
      final String msg = "Unknown type of AST: " + iast;
      throw new CgenerationException(msg);
    }
  }
```



```
  /** Toutes les méthodes qui suivent remplissent le tampon courant
   * (buffer). Il y a une méthode generate par catégorie
   * syntaxique. Attention, la methode pour un programme ne prend
   * pas de destination. */

  protected void generate (final IASTprogram iast,
                           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("\n");
      buffer.append("ILP_Object ilp_program ()\n");
      analyzeInstruction(iast.getBody(), lexenv, common, "return");
      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");
      buffer.append("/* fin */\n");
  }

  protected void generate (final IASTalternative iast,
                           final ICgenLexicalEnvironment lexenv,
                           final ICgenEnvironment common,
                           final String destination)
    throws CgenerationException {
    buffer.append(" if ( ILP_isEquivalentToTrue( ");
    analyzeExpression(iast.getCondition(), lexenv, common);
    buffer.append(" ) ) {\n");
    analyzeInstruction(iast.getConsequent(), lexenv, common, destination);
    buffer.append(";\n}");
    if ( iast.isTernary() ) {
      buffer.append(" else {\n");
      try {
        analyzeInstruction(iast.getAlternant(), lexenv, common, destination);
        buffer.append(";");
      } catch (Exception e) {
          final String msg = "Should never occur!";
          assert false : msg;
          throw new CgenerationException(msg);
      }
      buffer.append("\n}");
    } else {
      buffer.append(" else {\n");
      buffer.append(destination);
      buffer.append(" ILP_FALSE;");
      buffer.append("\n}");
    }
  }

  protected void generate (final IASTbinaryOperation iast,
                           final ICgenLexicalEnvironment lexenv,
                           final ICgenEnvironment common,
                           final String destination)
    throws CgenerationException {
    buffer.append(destination);
    buffer.append(" ");
    buffer.append(common.compileOperator2(iast.getOperatorName()));
    buffer.append("(");
    analyzeExpression(iast.getLeftOperand(), lexenv, common);
    buffer.append(", ");
    analyzeExpression(iast.getRightOperand(), lexenv, common);
    buffer.append(") ");
  }

  protected void generate (final IASTboolean iast,
                           final ICgenLexicalEnvironment lexenv,
                           final ICgenEnvironment common,
                           final String destination)
    throws CgenerationException {
    buffer.append(destination);
    if ( iast.getValue() ) {
      buffer.append(" ILP_TRUE ");
    } else {
      buffer.append(" ILP_FALSE ");
```

```
  }
}

protected void generate (final IASTfloat iast,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  buffer.append(destination);
  buffer.append(" ILP_Float2ILP(");
  buffer.append(iast.getValue());
  buffer.append(") ");
}

protected void generate (final IASTinteger iast,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  final BigInteger bi = iast.getValue();
  // && et non || comme l'a remarqué Nicolas.Bros@gmail.com
  if (    bi.compareTo(BIMIN) > 0
       && bi.compareTo(BIMAX) < 0 ) {
    buffer.append(destination);
    buffer.append(" ILP_Integer2ILP(");
    buffer.append(bi.intValue());
    buffer.append(") ");
  } else {
    final String msg = "Too large integer " + bi;
    throw new CgenerationException(msg);
  }
}
public static final BigInteger BIMIN;
public static final BigInteger BIMAX;
static {
  final Integer i = Integer.MIN_VALUE;
  BIMIN = new BigInteger(i.toString());
  final Integer j = Integer.MAX_VALUE;
  BIMAX = new BigInteger(j.toString());
}

protected void generate (final IASTinvocation iast,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  buffer.append(destination);
  buffer.append(" ");
  analyzeExpression(iast.getFunction(), lexenv, common);
  buffer.append("(");
  final int numberOfArguments = iast.getArgumentsLength();
  for ( int i=0 ; i<numberOfArguments ; i++ ) {
      IAST iast2 = iast.getArgument(i);
      analyzeExpression(iast2, lexenv, common);
      if ( i<numberOfArguments-1 ) {
          buffer.append(", ");
      }
  }
  buffer.append(")");
}

protected void generate (final IASTsequence iast,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  buffer.append("{\n");
  final IAST[] instrs = iast.getInstructions();
  for ( int i = 0 ; i < instrs.length-1 ; i ++) {
      final IAST iast2 = instrs[i];
      analyzeInstruction(iast2, lexenv, common, "(void)");
      buffer.append(";\n");
  }
  analyzeInstruction(instrs[instrs.length-1], lexenv, common, destination);
  buffer.append(";\n}\n");
}

protected void generate (final IASTstring iast,
```



```
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  buffer.append(destination);
  buffer.append(" ILP_String2ILP(\"");
  final String s = iast.getValue();
  final int n = s.length();
  for ( int i = 0 ; i<n ; i++ ) {
    char c = s.charAt(i);
    switch ( c ) {
    case '\\':
    case '"': {
      buffer.append("\\");
    }
  //$FALL-THROUGH$
  default: {
      buffer.append(c);
    }
    }
  }
  buffer.append("\") ");
}

protected void generate (final IASTunaryBlock iast,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  final IASTvariable tmp = common.generateVariable();
  final ICgenLexicalEnvironment lexenv2 =
    lexenv.extend(tmp, tmp.getName());
  final ICgenLexicalEnvironment lexenv3 =
    lexenv2.extend(iast.getVariable());

  buffer.append("{\n");
  buffer.append(" ILP_Object ");
  analyzeExpression(tmp, lexenv2, common);
  buffer.append(" = ");
  analyzeExpression(iast.getInitialization(), lexenv, common);
  buffer.append(";\n");

  buffer.append(" ILP_Object ");
  analyzeExpression(iast.getVariable(), lexenv3, common);
  buffer.append(" = ");
  analyzeExpression(tmp, lexenv2, common);
  buffer.append(";\n");

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

protected void generate (final IASTunaryOperation iast,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  buffer.append(destination);
  buffer.append(" ");
  buffer.append(common.compileOperator1(iast.getOperatorName()));
  buffer.append("(");
  analyzeExpression(iast.getOperand(), lexenv, common);
  buffer.append(") ");
}

protected void generate (final IASTvariable iast,
                         final ICgenLexicalEnvironment lexenv,
                         final ICgenEnvironment common,
                         final String destination)
  throws CgenerationException {
  buffer.append(destination);
  buffer.append(" ");
  buffer.append(lexenv.compile(iast));
  buffer.append(" ");
}

}
```

Java/src/fr/upmc/ilp/ilp1/cgen/ICgenEnvironment.java

```java
package fr.upmc.ilp.ilp1.cgen;

import fr.upmc.ilp.ilp1.interfaces.*;

/** L'interface décrivant l'environnement des opérateurs prédéfinis du
 * langage à compiler vers C. Il est l'analogue de runtime/ICommon
 * pour le paquetage cgen. */
public interface ICgenEnvironment {

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

  String compileOperator1 (String opName)
    throws CgenerationException ;

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

  String compileOperator2 (String opName)
    throws CgenerationException ;

  /** un générateur de variables temporaires. */

  IASTvariable generateVariable ();

  /** L'enrichisseur d'environnement lexical avec les primitives. */

  ICgenLexicalEnvironment
    extendWithPrintPrimitives (ICgenLexicalEnvironment lexenv);

  /** L'enrichisseur d'environnement lexical avec les constantes. */

  ICgenLexicalEnvironment
    extendWithPredefinedConstants (ICgenLexicalEnvironment lexenv);
}
```

Java/src/fr/upmc/ilp/ilp1/cgen/ICgenLexicalEnvironment.java

```java
package fr.upmc.ilp.ilp1.cgen;

import fr.upmc.ilp.ilp1.interfaces.*;

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

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

  String compile (IASTvariable variable)
    throws CgenerationException;

  /** Étend l'environnement avec une nouvelle variable et vers quel
   * nom la compiler. */

  ICgenLexicalEnvironment extend (IASTvariable variable,
                                  String compiledName );

  /** Étend l'environnement avec une nouvelle variable qui sera
   * compilée par son propre nom. */

  ICgenLexicalEnvironment extend (IASTvariable variable);
}
```

Java/src/fr/upmc/ilp/ilp1/AbstractProcess.java


```java
package fr.upmc.ilp.ilp1;

//import javax.inject.Inject; // Utiliser Guice

import java.io.IOException;
import java.io.StringReader;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;

import com.thaiopensource.validate.ValidationDriver;

import fr.upmc.ilp.annotation.OrNull;
import fr.upmc.ilp.ilp1.interfaces.IAST;
import fr.upmc.ilp.ilp1.interfaces.IProcess;
import fr.upmc.ilp.tool.File;
import fr.upmc.ilp.tool.IContent;
import fr.upmc.ilp.tool.IFinder;

/** Cette classe abstraite est une implantation de IProcess avec quelques
 * methodes probablement utiles si l'on en herite. */

public abstract class AbstractProcess
implements IProcess {

    protected AbstractProcess (IFinder finder) throws IOException {
        setFinder(finder);
        // Par defaut, on cherche les grammaires dans Grammars/
        getFinder().addPath("Grammars");
        // les programmes C dans C/
        getFinder().addPath("C");
        // Le script de compilation et d'execution est C/compileThenRun.sh
        setCompileThenRunScript(getFinder().findFile("compileThenRun.sh"));
        // et les fichiers temporaires sont chez l'utilisateur:
        final File tempDir = new File(System.getProperty("user.dir"));
        setCFile(new java.io.File(tempDir, "theProgram.c"));
    }

    public void setVerbose (boolean verbose) {
        this.verbose = verbose;
    }
    /** Ce booleen rend Process verbeux! */
    protected boolean verbose = true;

    /** Definir où chercher tous les fichiers nécessaires. */
    public void setFinder (IFinder finder) {
        this.finder = finder;
    }
    public IFinder getFinder () {
        assert(this.finder != null);
        return this.finder;
    }
    //@Inject
    private IFinder finder;

    /** Initialisation: On se contente de recuperer le texte du programme
     * dont on va s'occuper.
     */

  public void initialize (IContent ic) {
      try {
          this.programText = ic.getContent();
          this.initialized = true;
      } catch (Throwable e) {
          this.initializationFailure = e;
          if ( this.verbose ) {
              System.err.print(e);
          }
      }
  }

  public boolean isInitialized() {
      return this.initialized;
  }
```

```
    public @OrNull Throwable getInitializationFailure() {
        return this.initializationFailure;
    }

    public @OrNull String getProgramText() {
        assert this.initialized;
        return this.programText;
    }

    protected boolean initialized = false;
    protected Throwable initializationFailure = null;
    protected String programText;

    /** Preparation */

    public boolean isPrepared() {
        return this.prepared;
    }
    protected boolean prepared = false;

    public @OrNull Throwable getPreparationFailure() {
        return this.preparationFailure;
    }
    protected Throwable preparationFailure = null;

    public @OrNull IAST getIAST() {
        assert this.prepared;
        return this.iast;
    }
    public void setIAST(IAST iast) {
        this.iast = iast;
    }
    //@Inject
    protected IAST iast;

    /** Un utilitaire pour la preparation.
     * Étant donné le nom d'une grammaire, il charge la grammaire et
     * valide le texte du programme à traiter. */
    public Document getDocument (String rngBaseName)
            throws IOException, SAXException, ParserConfigurationException {
        final java.io.File rngFile = getFinder().findFile(rngBaseName);
        return getDocument(rngFile);
    }

    public Document getDocument (java.io.File rngFile)
        throws IOException, SAXException, ParserConfigurationException {
        final String rngFilePath = rngFile.getAbsolutePath();
        final InputSource isg = ValidationDriver.fileInputSource(rngFilePath);
        final ValidationDriver vd = new ValidationDriver();
        vd.loadSchema(isg);
        InputSource is =
            new org.xml.sax.InputSource(
                    new StringReader(this.programText));
        // Manquait (comme signalé par Rafael.Cerioli@gmail.com):
        if ( ! vd.validate(is) ) {
            throw new SAXException("Invalid XML program!");
        }

        final DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
        final DocumentBuilder db = dbf.newDocumentBuilder();
        // Le precedent flux est tari!
        is = new org.xml.sax.InputSource(
                    new StringReader(this.programText));
        this.document = db.parse(is);
        return this.document;
    }
    /** getDocument() sans argument retourne le DOM. */
    public Document getDocument () {
        assert this.document != null;
        return this.document;
    }
    private Document document = null;

    /** Definir la grammaire a utiliser pour comprendre le programme XML */
    public void setGrammar (java.io.File rngFile) {
```



```
        this.rngFile = rngFile;
    }
    public java.io.File getGrammar() {
        assert(this.rngFile != null);
        return this.rngFile;
    }
    /** Le fichier contenant la grammaire d'ILP a utiliser: */
    //@Inject
    protected java.io.File rngFile;

    /** Interpretation */

    public boolean isInterpreted() {
        return this.interpreted;
    }
    protected boolean interpreted = false;

    public @OrNull Throwable getInterpretationFailure() {
        return this.interpretationFailure;
    }
    protected Throwable interpretationFailure = null;

    public String getInterpretationPrinting() {
        assert(this.interpreted);
        return this.printing;
    }
    protected String printing = "";

    public Object getInterpretationValue() {
        assert(this.interpreted);
        return this.result;
    }
    protected Object result = null;

    /** Compilation vers C. */

    public boolean isCompiled() {
        return this.compiled;
    }
    protected boolean compiled = false;

    public @OrNull Throwable getCompilationFailure() {
        return this.compilationFailure;
    }
    protected Throwable compilationFailure = null;

    public String getCompiledProgram() {
        assert(this.compiled);
        return this.ccode;
    }
    protected String ccode;

    /** Exécution du programme compilé: */

    /** Set the name of the C file to generate. */
    public void setCFile (java.io.File cFile) {
        this.cFile = cFile;
    }
    /** Le nom du fichier C qui sera engendré: */
    //@Inject
    protected java.io.File cFile;
    /** Le nom du script compilant et executant le programme C engendré: */
    //@Inject
    protected java.io.File compileThenRunScript;
    public void setCompileThenRunScript (java.io.File scriptFile) {
        this.compileThenRunScript = scriptFile;
    }

    public @OrNull Throwable getExecutionFailure() {
        return this.executionFailure;
    }
    protected Throwable executionFailure = null;

    public String getExecutionPrinting() {
        assert(this.executed);
        return this.executionPrinting;
    }
    protected String executionPrinting;
```

```
    public boolean isExecuted() {
        return this.executed;
    }
    protected boolean executed = false;
}
```

Java/src/fr/upmc/ilp/ilp1/Process.java

```
package fr.upmc.ilp.ilp1;

import java.io.IOException;

import org.w3c.dom.Document;

import fr.upmc.ilp.ilp1.cgen.CgenEnvironment;
import fr.upmc.ilp.ilp1.cgen.CgenLexicalEnvironment.Empty;
import fr.upmc.ilp.ilp1.cgen.Cgenerator;
import fr.upmc.ilp.ilp1.cgen.ICgenEnvironment;
import fr.upmc.ilp.ilp1.cgen.ICgenLexicalEnvironment;
import fr.upmc.ilp.ilp1.eval.EAST;
import fr.upmc.ilp.ilp1.eval.EASTFactory;
import fr.upmc.ilp.ilp1.eval.EASTParser;
import fr.upmc.ilp.ilp1.runtime.CommonPlus;
import fr.upmc.ilp.ilp1.runtime.ConstantsStuff;
import fr.upmc.ilp.ilp1.runtime.EmptyLexicalEnvironment;
import fr.upmc.ilp.ilp1.runtime.ICommon;
import fr.upmc.ilp.ilp1.runtime.ILexicalEnvironment;
import fr.upmc.ilp.ilp1.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'ILP1. */

public class Process extends AbstractProcess {

    public Process (IFinder finder) throws IOException {
        super(finder);
        // Initialiser ici les caractéristiques non liées aux tests et,
        // éventuellement indiquer où chercher ces fichiers:
        setGrammar(getFinder().findFile("grammar1.rng"));
    }

    /** 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.
     */

    @Override
    public void prepare() {
        try {
            assert this.rngFile != null;
            final Document d = getDocument(this.rngFile);

            final EASTFactory factory = new EASTFactory();
            final EASTParser parser = new EASTParser(factory);
            this.east = parser.parse(d);
            setIAST(this.east);

            this.prepared = true;

        } catch (Throwable e) {
            this.preparationFailure = e;
            if ( this.verbose ) {
                System.err.println(e);
            }
        }
    }
    protected EAST east; // Hack car IAST insuffisant pour eval().

    /** Interprétation */

    @Override
    public void interpret() {
```



```
        try {
            assert this.prepared;
            final ICommon intcommon = new CommonPlus();
            ILexicalEnvironment intlexenv = EmptyLexicalEnvironment.create();
            final PrintStuff intps = new PrintStuff();
            intlexenv = intps.extendWithPrintPrimitives(intlexenv);
            final ConstantsStuff intcs = new ConstantsStuff();
            intlexenv = intcs.extendWithPredefinedConstants(intlexenv);

            this.result = this.east.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. */

    @Override
    public void compile() {
        try {
            assert this.prepared;
            final ICgenEnvironment common = new CgenEnvironment();
            final Cgenerator compiler = new Cgenerator(common);
            ICgenLexicalEnvironment lexenv = Empty.create();
            lexenv = common.extendWithPrintPrimitives(lexenv);
            lexenv = common.extendWithPredefinedConstants(lexenv);
            this.ccode = compiler.compile(this.east, lexenv, "return");

            this.compiled = true;

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

    /** Exécution du programme compilé: */

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

            // 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/tool/AbstractEnvironment.java

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

/**
 * Une classe utilitaire pour les environnements: c'est une fonction
 * des variables vers les valeurs representee par une liste chainee.
 *
 * @param <V> la classe des variables
 * @param <W> la classe des valeurs
 */
public abstract class AbstractEnvironment<V, W> {

    public AbstractEnvironment(V variable, W value, AbstractEnvironment<V, W> next) {
        this.variable = variable;
        this.value = value;
        this.next = next;
    }
    private final V variable;
    private W value;
    private final AbstractEnvironment<V, W> next;

    public V getVariable () {
        return this.variable;
    }
    public W getValue () {
        return this.value;
    }
    public AbstractEnvironment<V, W> getNextEnvironment () {
        return this.next;
    }

    public W setValue (W newValue) {
        W oldValue = this.value;
        this.value = newValue;
        return oldValue;
    }

    /** Renvoie la valeur d'une variable si prÃ©sente dans
     * l'environnement. */
    public W lookup (V variable) {
        // NOTA: la comparaison s'effectue par equals():
        if (this.variable.equals(variable)) {
            return this.value;
        } else {
            return this.next.lookup(variable);
        }
    }

}

// fin d'AbstractEnvironment.java
```

Java/src/fr/upmc/ilp/tool/CStuff.java

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

/** Conversion de suites de lettres du monde ILP vers le monde C. */

public class CStuff {

  /** Transformer un identificateur ILP en un identificateur C. */

  public static String mangle (final String s) {
    assert(s != null);
    final StringBuffer sb = new StringBuffer(s.length());
    for ( int i=0 ; i<s.length() ; i++ ) {
      final char c = s.charAt(i);
      if (   ( 'a' <= c && c <= 'z' )
          || ( 'A' <= c && c <= 'Z' )
          || ( '0' <= c && c <= '9' )
          || ( '_' == c ) ) {
        sb.append(c);
      } else {
          sb.append("_");
          sb.append(Integer.toHexString(c));
      }
    }
    return sb.toString();
  }

  /** Transformer une chaine ILP en une chaine C. */

  public static String protect (String value) {
      final int n = value.length();
      final StringBuffer result = new StringBuffer(n);
      for ( int i = 0 ; i<n ; i++ ) {
        char c = value.charAt(i);
        switch ( c ) {
        case '\\':
        case '"': {
          result.append("\\");
        }
        //$FALL-THROUGH$
        default: {
          result.append(c);
        }
        }
      }
      return result.toString();
  }

}
```


Java/src/fr/upmc/ilp/tool/File.java

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

import java.io.*;

public class File extends java.io.File
implements IContent {

    static final long serialVersionUID = +1234567890002100OL;

    public File (String filename) {
        super(filename);
    }

    public File (java.io.File file) {
        super(file.getAbsolutePath());
    }

    /** Calculer le nom de base (sans repertoire ni suffixe). */

    public String getBaseName () {
        String name = getName();
        int dotIndex = name.lastIndexOf('.');
        if (dotIndex >= 0) {
            return name.substring(0, dotIndex);
        } else {
            return name;
        }
    }

    /** Calculer le nom complet du fichier hors son suffixe. */

    public String getNameWithoutSuffix () {
        return getParent() + File.separator + getBaseName();
    }

    /** Fournir le contenu d'un fichier sous la forme d'une chaine de
     * caracteres. */

    public String getContent() {
      try {
        return FileTool.slurpFile(this);
      } catch (IOException e) {
        throw new RuntimeException(e);
      }
    }

    /** Cette methode statique est utile pour recuperer le contenu d'une
```

```
     * instance de java.io.File. */

    public static String getContent (File file) {
        try {
          return FileTool.slurpFile(file);
        } catch (IOException e) {
          throw new RuntimeException(e);
        }
    }

}
```

Java/src/fr/upmc/ilp/tool/FileTool.java

```
package fr.upmc.ilp.tool;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;

/** Utilitaire pour lire ou écrire des fichiers entiers. */

public final class FileTool {

    public FileTool () {
        super();
    }

    /** Convertir un fichier en une chaîne de caracteres.
     * @return le contenu du fichier lu
     * @param  file le fichier a lire
     */

    public static String slurpFile (final File file)
            throws IOException {
        final int length = (int) file.length();
        final char[] buffer = new char[length];
        try (final FileReader fr = new FileReader(file)) {
            int offset = 0;
            while ( offset < length ) {
                int read = fr.read(buffer, offset, length-offset);
                offset += read;
            }
        }
        return new String(buffer);
    }

    public static String slurpFile (final Reader fr)
    throws IOException {
        final StringBuffer sb = new StringBuffer();
        final BufferedReader br = new BufferedReader(fr);
        final char[] buffer = new char[4096];
        while ( true ) {
            int count = br.read(buffer);
            if (count < 0) {
                return sb.toString();
            }
            sb.append(buffer, 0, count);
        }
    }

    /** Convertir un fichier en une chaîne de caracteres.
     * @return le contenu du fichier lu
     * @param filename le nom du fichier a lire
     */

    public static String slurpFile (final String filename)
    throws IOException {
        return FileTool.slurpFile(new File(filename));
    }

    /** Lire un fichier correspondant au résultat attendu d'un programme ILP.
     * Cette methode est utile pour tester ILP car elle retire les
     * separateurs (blancs et retours a la ligne) superflus ce qui permet
```



```
     * de compenser les differences d'impression entre Scheme, Java et C. */

    public static String readExpectedResult (String basefilename)
      throws IOException {
      File resultFile = new File(basefilename + ".result");
      return FileTool.slurpFile(resultFile).trim();
    }

    public static String readExpectedResult (fr.upmc.ilp.tool.File file)
      throws IOException {
      File resultFile = new File(file.getNameWithoutSuffix() + ".result");
      return FileTool.slurpFile(resultFile).trim();
    }

    // TODO enlever les trim() en trop dans les CompilerTest

    /** Lire le fichier correspondant aux impressions attendues d'un
     * programme ILP. Cette methode est utile pour tester ILP car elle
     * retire les separateurs (blancs et retours a la ligne) superflus
     * ce qui permet de compenser les differences d'impression entre Scheme,
     * Java et C. */

    public static String readExpectedPrinting (String basefilename)
      throws IOException {
      File resultFile = new File(basefilename + ".print");
      return FileTool.slurpFile(resultFile).trim();
    }

    public static String readExpectedPrinting (fr.upmc.ilp.tool.File file)
      throws IOException {
      File resultFile = new File(file.getNameWithoutSuffix() + ".print");
      return FileTool.slurpFile(resultFile).trim();
    }

    /** Écrire une chaîne dans un fichier.
     * @param filename le nom du fichier a ecrire
     * @param s le nouveau contenu du fichier
     */

    public static void stuffFile (final String filename, final String s)
    throws IOException {
        FileTool.stuffFile(new File(filename), s);
    }

    /** Écrire une chaîne dans un fichier.
     * @param file le fichier a ecrire
     * @param s le nouveau contenu du fichier
     */

    public static void stuffFile (final File file, final String s)
    throws IOException {
        //final FileWriter fw = new FileWriter(file);
        //try {
        //    fw.write(s, 0, s.length());
        //} finally {
        //    fw.close();
        //}
        // Nouvelle écriture pour Java7:
        try (final FileWriter fw = new FileWriter(file)) {
            fw.write(s, 0, s.length());
        }
    }
}
```

Java/src/fr/upmc/ilp/tool/Finder.java

```
package fr.upmc.ilp.tool;

/** Implantation d'IFinder, une sorte de mécanisme de PATH listant une suite
 * de répertoires où chercher des fichiers. */

import java.io.IOException;
import java.util.Vector;
import java.io.File;

public class Finder implements IFinder {
```

```
    private final boolean verbose = false; // DEBUG

    public Finder () {
        this.directories = new Vector<>();
    }
    private final Vector<File> directories;

    @Override
    public String toString () {
        StringBuffer sb = new StringBuffer();
        sb.append("Finder[");
        for ( File d : directories ) {
            sb.append(d + ",");
        }
        sb.append("]");
        return sb.toString();
    }

    public File findFile (final String baseFileName)
    throws IOException {
        File file = new File(baseFileName);
        if ( file.isAbsolute() ) {
            if ( file.exists() ) {
                if ( verbose ) {
                    System.err.println("Found " + file.getAbsolutePath());
                }
                return file;
            }
            throw new IOException("Inexistent absolute file");
        }
        for ( File dir : directories ) {
            if ( verbose ) {
                System.err.println("Searching " + dir.getAbsolutePath());
            }
            final String fileName = dir.getAbsolutePath()
                    + File.separator + baseFileName;
            file = new File(fileName);
            if ( file.exists() ) {
                if ( verbose ) {
                    System.err.println("Found " + file.getAbsolutePath());
                }
                return file;
            }
        }
        throw new IOException("Cannot find file " + baseFileName);
    }

    public void addPath (final File directory)
    throws IOException {
        if ( ! directory.isDirectory() ) {
            throw new IOException("Not a directory: "
               + directory.getAbsolutePath()
               + " !");
        }
        if ( ! directories.contains(directory) ) {
            directories.add(directory);
        }
    }

    public void addPath (String directoryName)
    throws IOException {
        directoryName = directoryName.replaceAll("/", File.separator);
        final File directory = new File(directoryName);
        addPath(directory);
    }

    public void addPossiblePath (String directoryName) {
        try {
            addPath(directoryName);
        } catch (IOException e) {
            // Ce n'est pas grave!
        }
    }

    public java.io.File[] getPaths () {
        return directories.toArray(new File[0]);
    }
```



```
    public void setPaths (final File[] paths) throws IOException {
        directories.removeAllElements();
        for ( File dir : paths ) {
            addPath(dir);
        }
    }
}
```

Java/src/fr/upmc/ilp/tool/IContent.java

```
package fr.upmc.ilp.tool;

import java.io.IOException;

/** Une interface fournissant un contenu à savoir un programme ILP.
 * Cette interface sert le plus souvent à masquer un fichier mais ce
 * pourrait être également une chaîne, une URL ou un générateur de
 * tests. */

public interface IContent {

    /** Fournir la chaîne de caractères représentant le programme ILP
     * auquel on s'intéresse.
     *
     * @return String
     */
    String getContent () throws IOException;

}
```

Java/src/fr/upmc/ilp/tool/IFinder.java

```
package fr.upmc.ilp.tool;

import java.io.File;
import java.io.IOException;

/**
 * Cet utilitaire contient une liste de répertoires où chercher un
 * fichier. On peut dynamiquement changer la liste des répertoires où
 * chercher. En cas d'anomalie, il signale des exceptions.
 *
 * D'abord, on crée un Finder, on l'enrichit avec des répertoires (un
 * par un ou en bloc) puis on recherche des fichiers avec ce Finder.
 */

public interface IFinder {

    /** Chercher un fichier par son petit nom dans les répertoires
     * associés à ce Finder. Signale une exception si l'on ne le trouve
     * nulle part. */
    public File findFile (String baseFileName) throws IOException;

    /** Ajouter un répertoire à la fin de la liste des répertoires
     * où chercher. Le répertoire doit être un vrai répertoire sinon
     * une exception est signalée. */
    public void addPath (File directory) throws IOException;
    /** Ajouter un répertoire à la fin de la liste des répertoires
     * où chercher. Le répertoire doit être un vrai répertoire sinon
     * une exception est signalée. Le répertoire est ici spécifié par
     * une chaîne de caractères en convention URL (avec des slashes). */
    public void addPath (String directory) throws IOException;
    /** AJouter un répertoire mais seulement si c'est bien un répertoire. */
    public void addPossiblePath (String directory);

    /** Renvoyer la liste des répertoires où chercher. */
    public java.io.File[] getPaths ();

    /** Imposer la liste des répertoires où chercher. Tous doivent être
     * des vrais répertoires autrement une exception est signalée. */
    public void setPaths (File[] paths) throws IOException;
}
```

Java/src/fr/upmc/ilp/tool/Parameterized.java

```java
// $Id$
package fr.upmc.ilp.tool;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import java.lang.reflect.Modifier;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.junit.runner.Runner;
import org.junit.runner.notification.RunNotifier;
import org.junit.runners.BlockJUnit4ClassRunner;
import org.junit.runners.Suite;
import org.junit.runners.model.FrameworkMethod;
import org.junit.runners.model.InitializationError;
import org.junit.runners.model.Statement;
import org.junit.runners.model.TestClass;

/** This class is a cloned version of the org.junit.runners.Parameterized
 * class. It modifies a single method, the getName method, that is used
 * by Eclipse to display the name of the JUnit4 test.
 * */

public class Parameterized extends Suite {

    @Retention(RetentionPolicy.RUNTIME)
    @Target(ElementType.METHOD)
    public static @interface Parameters {}

    private class TestClassRunnerForParameters
        extends BlockJUnit4ClassRunner {
        private final int fParameterSetNumber;

        private final List<Object[]> fParameterList;

        TestClassRunnerForParameters(Class<?> type,
                List<Object[]> parameterList, int i) throws InitializationError {
            super(type);
            fParameterList= parameterList;
            fParameterSetNumber= i;
        }

        @Override
        public Object createTest() throws Exception {
            return getTestClass().getOnlyConstructor().newInstance(
                    computeParams());
        }

        private Object[] computeParams() throws Exception {
            try {
                return fParameterList.get(fParameterSetNumber);
            } catch (ClassCastException e) {
                throw new Exception(String.format(
                        "%s.%s() must return a Collection of arrays.",
                        getTestClass().getName(), getParametersMethod(
                                getTestClass()).getName()));
            }
        }

        /* This method prints the name of the file currently processed. */
        @Override
        protected String getName() {
            File file = (File) fParameterList.get(fParameterSetNumber)[0];
            return String.format("[%s : %s]",
                    fParameterSetNumber,
                    file.getBaseName() );
        }

        @Override
        protected String testName(final FrameworkMethod method) {
            return String.format("%s[%s]", method.getName(),
                    fParameterSetNumber);
        }

        @Override
        protected void validateZeroArgConstructor(List<Throwable> errors) {
            // constructor can, nay, should have args.
```



```java
        }

        @Override
        protected Statement classBlock(RunNotifier notifier) {
            return childrenInvoker(notifier);
        }
    }

    private final ArrayList<Runner> runners= new ArrayList<>();

    /**
     * Only called reflectively. Do not use programmatically.
     */
    public Parameterized(Class<?> klass) throws Throwable {
        super(klass, Collections.<Runner>emptyList());
        List<Object[]> parametersList= getParametersList(getTestClass());
        for (int i= 0; i < parametersList.size(); i++)
            runners.add(new TestClassRunnerForParameters(getTestClass().getJavaClass(),
                    parametersList, i));
    }

    @Override
    protected List<Runner> getChildren() {
        return runners;
    }

    @SuppressWarnings("unchecked")
    private List<Object[]> getParametersList(TestClass klass)
            throws Throwable {
        return (List<Object[]>) getParametersMethod(klass).invokeExplosively(
                null);
    }

    private FrameworkMethod getParametersMethod(TestClass testClass)
            throws Exception {
        List<FrameworkMethod> methods= testClass
                .getAnnotatedMethods(Parameters.class);
        for (FrameworkMethod each : methods) {
            int modifiers= each.getMethod().getModifiers();
            if (Modifier.isStatic(modifiers) && Modifier.isPublic(modifiers))
                return each;
        }

        throw new Exception("No public static parameters method on class "
                + testClass.getName());
    }

}
```

Java/src/fr/upmc/ilp/tool/ProgramCaller.java

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

import java.io.BufferedInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.concurrent.CountDownLatch;

/** Une classe pour invoquer un programme externe (gcc par exemple) et
 * récupérer le contenu de son flux de sortie. */

public class ProgramCaller {

    /** Construire une instance de ProgramCaller qui permettra d'exécuter
     * une commande via le système d'exploitation.
     *
     * @param program
     *     la commande (programme et arguments) à exécuter.
     */

    public ProgramCaller (final String program) {
        this.program = program;
        this.stdout = new StringBuffer(1023);
        this.stderr = new StringBuffer(1023);
        this.countDownLatch = new CountDownLatch(2);
        this.running = false;
        this.verbose = 0;
    }
```

```
    private final String program;
    private static final Runtime RUNTIME = Runtime.getRuntime();
    private Process process;
    private final StringBuffer stdout;
    private final StringBuffer stderr;
    private CountDownLatch countDownLatch;
    private boolean running;
    private int verbose = 110;

    /** Verbaliser les traces d'exécution de la commande. */

    public void setVerbose () {
        this.verbose++;
    }

    private void verbalize (final String message) {
        this.verbalize(message, 0);
    }

    private void verbalize (final String message, final int level) {
        if ( this.verbose > level ) {
            System.err.println(message);
        }
    }

    /** Récupérer le flux de sortie produit par la commande.
     *
     * @return la sortie standard de la commande
     */

    public String getStdout () {
        return stdout.toString();
    }

    /** Récupérer le flux de sortie d'erreur produit par la commande.
     *
     * @return la sortie d'erreur de la commande
     */

    public String getStderr () {
        return stderr.toString();
    }

    /** Recuperer le code de retour de la commande.
     * Bloque quand celle-ci n'est pas terminee.
     *
     * @return le code de retour
     */

    public int getExitValue () {
        this.verbalize("-getExitValue-", 10);
        try {
            this.countDownLatch.await();
        } catch (InterruptedException e) {
            this.verbalize("-getExitValue Exception-", 10);
            throw new RuntimeException();  // MOCHE
        }
        return this.exitValue;
    }
    private transient int exitValue = 199;

    /** Lancer la commande et stocker ses flux de sortie (normale et
     * d'erreur) dans un tampon pour pouvoir les analyser apres-coup. */

    public void run () {
        // Au plus, une seule invocation:
        synchronized (this) {
            if ( running ) {
                return;
            } else {
                running = true;
            }
        }
        // Exécuter la commande:
        verbalize("[Running: " + program + "...");
        try {
            this.process = RUNTIME.exec(program);
        } catch (Throwable e) {
            // Exception notamment signalee quand le programme n'existe pas:
```



```
            this.stderr.append(e.getMessage());
            // On ne fait pas partir les deux taches slurpStd*():
            this.countDownLatch.countDown();
            this.countDownLatch.countDown();
            verbalize("...not started]");
            return;
        }

        // NOTA: documentation tells that, on Windows, one should read
        // asap the results or deadlocks may occur.
        slurpStdOut();
        slurpStdErr();

        try {
            this.process.waitFor();
            this.countDownLatch.await();
        } catch (InterruptedException e) {
            this.verbalize("!run Exception!", 10);
            throw new RuntimeException(); // MOCHE
        }
        this.exitValue = this.process.exitValue();
        verbalize("...finished]");
    }

    private void slurpStdOut () {
        final Thread tstdout = new Thread () {
            @Override
            public void run () {
                try (final InputStream istdout = process.getInputStream();
                     final BufferedInputStream bstdout =
                           new BufferedInputStream(istdout) ) {
                    final int size = 4096;
                    final byte[] buffer = new byte[size];
                    READ:
                        while ( true ) {
                            int count = 0;
                            try {
                                count = bstdout.read(buffer, 0, size);
                            } catch (IOException exc) {
                                continue READ;
                            }
                            if ( count > 0 ) {
                                final String s = new String(buffer, 0, count);
                                stdout.append(s);
                                ProgramCaller.this.verbalize("[stdout Reading: " + s + "]");
                            } else if ( count == -1 ) {
                                ProgramCaller.this.verbalize("[stdout Dried!]", 10);
                                ProgramCaller.this.countDownLatch.countDown();
                                return;
                            }
                        }
                } catch (IOException e) {
                    ProgramCaller.this.verbalize("[stdout problem!" + e + ']', 10);}
            }
        };
        tstdout.start();
    }

    // La meme chose sur le flux d'erreur:
    private void slurpStdErr () {
        final Thread tstderr = new Thread () {
            @Override
            public void run () {
                try (final InputStream istderr = process.getErrorStream();
                     final BufferedInputStream bstderr =
                           new BufferedInputStream(istderr) ) {
                    final int size = 4096;
                    final byte[] buffer = new byte[size];
                    READ:
                        while ( true ) {
                            int count = 0;
                            try {
                                count = bstderr.read(buffer, 0, size);
                            } catch (IOException exc) {
                                continue READ;
                            }
                            if ( count > 0 ) {
                                final String s = new String(buffer, 0, count);
```

```
                            stderr.append(s);
                            ProgramCaller.this.verbalize("[stderr Reading: " + s + "]");
                        } else if ( count == -1 ) {
                            ProgramCaller.this.verbalize("[stderr Dried!]", 10);
                            ProgramCaller.this.countDownLatch.countDown();
                            return;
                        }
                    }
                } catch (IOException e) {
                    ProgramCaller.this.verbalize("[stderr problem!" + e + ']', 10);
                }
            }
        };
        tstderr.start();
    }

}
```

Java/src/fr/upmc/ilp/annotation/ILPexpression.java

```
package fr.upmc.ilp.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/** Cette annotation, présente à l'exécution, indique qu'une méthode mène
 * à une expression ILP. Divers sous-attributs caractérisent mieux cette
 * annotation.
 *
 * Cas particuliers: <ul>
 * <li> getOperands() n'est pas annoté (ce serait redondant
 *      avec get*Operand()) </li>
 * </ul> */

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Target(ElementType.METHOD)
public @interface ILPexpression {

    /** Indique si la valeur obtenue par la methode peut être null. */

    boolean neverNull () default true;

    /** Indique si la valeur obtenue est en fait un vecteur d'expressions. */

    boolean isArray () default false;
}
```

Java/src/fr/upmc/ilp/annotation/ILPvariable.java

```
package fr.upmc.ilp.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/** Cette annotation, presente a l'execution, indique qu'une methode mene
 * a une variable ILP (ou a un tableau de variables) lue ou ecrite. Les
 * variables introduites par de nouvelles liaisons ne sont pas annotees.
 */

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Target(ElementType.METHOD)
public @interface ILPvariable {

    /** Indique si la valeur obtenue par la methode peut etre null. */
```



```
    boolean neverNull () default true;

    /** Indique si la valeur obtenue est en fait un vecteur de variables. */

    boolean isArray() default false;
}
```

Java/src/fr/upmc/ilp/annotation/OrNull.java

```
package fr.upmc.ilp.annotation;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Target;

/** Cette annotation precise que la valeur attendue peut aussi etre null.
 * Attention null instanceof AnyClass est toujours vrai!
 * Cette annotation pourrait servir a des outils d'analyse statique de code.
 */

@Documented
@Inherited
@Target({ElementType.METHOD,
         ElementType.PARAMETER,
         ElementType.FIELD})
public @interface OrNull {}
```

C/C.readme

```
Ce répertoire contient la bibliothèque d'exécution des différentes
versions d'ILP. Le Makefile contient les règles de reconstruction. La
commande 'make' sans argument devrait suffire. Si vous souhaitez
utiliser un GC (glaneur de cellule ou garbage collector), il faut
installer le gc de Boehm présent dans gc*.tgz et, pour cela,
tourner:

    make compile.gc

Si les tests du GC passent, vous pouvez utiliser l'option +gc dans
compileThenRun.sh. Cette option est utile à partir d'ILP6.
```

C/Makefile

```
# On efface les resultats de compilation pour faciliter le changement
# de machine (et d'architecture (Intel*86 vers powerpc par exemple)).
# En fait, il serait mieux de stocker les .o et .a dans un
# sous-repertoire mais comment automatiser (de facon portable) cela
# sous un OS non Unix ?

work : clean libilp.a
        -[ -f gc.a -a -d include ] || make compile.gc
# Pas de -o sur MacOSX:
        uname -s -r -v -m -o > HOSTTYPE 2>/dev/null || \
                uname -s -r -v -m > HOSTTYPE

# Effacer les resultats de compilation

clean :
        -rm -f *.o libilp.a HOSTTYPE gc-7.2/gc.a

# Les bibliotheques d'execution sont compilees en mode debug, c'est
# plus utile pour les exercices.

CC      =       gcc

# Pas de glibc pour MacOSX, on suppose que l'existence de /Library indique
# que c'est un Mac.
CFLAGS  =       -Wall -std=c99 -pedantic \
                -g \
 `if [ -d /Library ] ; then : ; else if pkg-config --exists glib-2.0 ; then pkg-config --cflags --libs g

# On compile tout ce qui est en C (sauf ilpHash.c non porte sur MacOSX)
# et sauf les template*.c qui sont bien sur incomplets.
```

```
CFILES  =       $(shell ls ilp*.c)

# On place ilpBasicError dans l'archive et non ilpError (afin de
# pouvoir en changer s'il le faut).

ARFILES =       ilp.o ilpAlloc.o ilpBasicError.o

# Ne pas oublier ranlib, c'est utile sur MacOSX.

libilp.a : ${CFILES:.c=.o}
        ar cvr libilp.a ${ARFILES}
        -ranlib libilp.a

# Autres dependances:

ilp.o : ilp.c ilp.h
ilpAlloc.o : ilpAlloc.c ilpAlloc.h ilp.h
ilpBasicError.o : ilpBasicError.c ilpBasicError.h ilp.h
ilpException.o : ilpException.c ilpException.h ilp.h
ilpError.o : ilpError.c ilpBasicError.h ilp.h
ilpObj.o : ilpObj.c ilpObj.h

# L'option +gc de compileThenRun.sh utilise le GC de Boehm qu'il faut
# aussi installer (cf. C.readme):

compile.gc : gc-7.2/gc.a
        mv gc-7.2/gc.a .
        mv gc-7.2/include .
        rm -rf gc-7.2
gc-7.2/gc.a : gc-7.2d.tgz
        [ -d gc-7.2 ] || tar xzf gc-7.2d.tgz
        cd gc-7.2/ && make -f Makefile.direct test
# QNC 2010oct: OK sur amd64 (ubuntu 9.10)
# QNC 2010oct: 6.8 ne compile pas sur MacOSX 10.6.4
# QNC 2012aug: 7.2d OK sur amd64 (Debian 6) et sur MacOSX 10.6.8

# end of Imakefile
```

C/compileThenRun.sh

```
#! /bin/bash
#* ********************************************************************
#* ILP -- Implantation d'un langage de programmation.
#* Copyright (C) 2004 <Christian.Queinnec@lip6.fr>
#* $Id: compileThenRun.sh 1299 2013-08-27 07:09:39Z queinnec $
#* GPL version>=2
#* ********************************************************************

USAGE="Usage: $0 [ +gc ] [ +v ] foo.c [baz.c -Dq=3 ...]
Compile les fichiers C, cree puis lance l'executable /tmp/test$USER

+gc est une option incluant le GC de Boehm (si utilisable)
+v montre sans l'executer la commande gcc synthetisee
foo.c est le resultat de la compilation d'ILP
bar.o sont des modules compiles a utiliser plutot que ceux de libilp.a
   [libilp.a contient les modules par defaut d'ILP1. Pour certaines
    versions d'ILP, il faut des modules plus appropries.]
hux.h est un fichier d'entete a inclure pour la compilation de foo.c
baz.c est un fichier C a compiler en meme temps que foo.c
-Dvar=val  variable cpp supplementaire a passer a gcc

Ce script est egalement sensible aux variables d'environnement:
  OTHER_LIBS   ajout de bibliotheques supplementaires
"

# Les variables contrôlant si l'on utilise le GC de Boehm:
#    Chemin relatif vers la bibliotheque statique:
LIB_GC=gc.a
#    Incorpore-t-on un GC ou pas ?
WITH_GC=false
# Montre-t-on la commande gcc synthetisee
VERBOSE=false

# Y a-t-il des options comme +gc ou +v ?
while [ $# -gt 0 ]
do
    case "$1" in
        +gc)
            WITH_GC=true
            shift
            ;;
        +v)
            VERBOSE=true
            shift
            ;;
        *)
            break
            ;;
    esac
done

CFLAGS='-Wall -Wno-unused-variable -Wno-unused-label -std=c99 -pedantic -g '
# NOTE: Il y a des tests avec des variables inutilisees, ne pas
# attirer l'attention dessus.

# C'est utile pour utiliser des tables associatives pour le tme8
UNAME_FLAGS=" -o"
if [ -n "${Apple_PubSub_Socket_Render}" ]
then
    # Sur MacOSX (il y a surement mieux comme test!)
    # Pas d'option -o sur MacOSX
    UNAME_FLAGS=''
else
    if pkg-config --exists glib-2.0 2>/dev/null
    then
        CFLAGS="$CFLAGS"`pkg-config --cflags --libs glib-2.0`
    fi
fi

COMMAND_DIR=`dirname $0`
case "$COMMAND_DIR" in
    # rendre le repertoire qui contient compileThenRun.sh absolu:
    /*)
        true
        ;;
    *)
        COMMAND_DIR=`pwd -P`/$COMMAND_DIR
        ;;
esac

# Si les .c ne sont pas compilés pour l'architecture courante, on les
# recompile à la volée.
( cd $COMMAND_DIR/
  RECOMPILE=true
  if [ -r ./HOSTTYPE ]
  then if [ "$(< ./HOSTTYPE)" = "`uname -s -r -v -m ${UNAME_FLAGS}`" ]
       then RECOMPILE=false
       fi
  fi
  if $RECOMPILE
  then
      echo "Compilation de la bibliotheque d'execution d'ILP..." >&2
      make clean
      if make work
      then :
      else
          echo "GC non compilable: option +gc non possible!" >&2
          exit 13
      fi
  fi 1>&2
)

if [ ! -r $COMMAND_DIR/libilp.a ]
then
    echo Bibliotheque introuvable: $COMMAND_DIR/libilp.a >&2
    exit 3
fi

if $WITH_GC
then
    if [ -r $COMMAND_DIR/$LIB_GC ]
    then
        CFLAGS="-DWITH_GC $CFLAGS"
    else
        echo "GC introuvable (consulter Makefile): $COMMAND_DIR/$LIB_GC" >&2
        # On continue sans GC puisque non compilé!
        WITH_GC=false
```

```
    fi
fi

# Collecter les fichiers (.o, .a .h ou .c) à incorporer et les rendre absolus:
FILES=${FILES:-}
for file
do
  case "$file" in
      /*.[coah])
          FILES="$FILES $file"
          if [ ! -r $file ]
          then
              echo Fichier introuvable: $file >&2
              exit 7
          fi
          ;;
      *.[coah])
          file=`pwd -P`/$file
          FILES="$FILES $file"
          if [ ! -r $file ]
          then
              echo Fichier introuvable: $file >&2
              exit 7
          fi
          ;;
      *)
          FILES="$FILES $file"
          ;;
  esac
done

OTHER_LIBS=${OTHER_LIBS:- -lm}
if $WITH_GC
then OTHER_LIBS="$OTHER_LIBS $COMMAND_DIR/$LIB_GC"
fi

AOUT=${TMPDIR:-/tmp}/test$USER
if [ -r $AOUT ]
then rm -f $AOUT
fi

if $VERBOSE
then
    echo gcc ${CFLAGS} -o $AOUT \
    -I. -I$COMMAND_DIR \
    $FILES $COMMAND_DIR/libilp.a $OTHER_LIBS '&&' $AOUT
fi

#echo Trying to compile $FILE ... >&2
gcc ${CFLAGS} -o $AOUT \
    -I. -I$COMMAND_DIR \
    $FILES $COMMAND_DIR/libilp.a $OTHER_LIBS && \
$AOUT

# end of compileThenRun.sh
```

C/ilp.c

```
/** Ce fichier constitue la bibliothèque d'exécution d'ILP. */
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "ilp.h"
#include "ilpAlloc.h"
#include "ilpBasicError.h"

/** Booléens.
 *
 * On alloue statiquement les deux booléens.
 * Pas possible encore en C d'allouer le flottant pi statiquement.
 * On l'alloue donc dynamiquement mais comme un singleton (cf. ilp_pi).
 */

struct ILP_Object ILP_object_true = {
     ILP_BOOLEAN_KIND,
     { ILP_BOOLEAN_TRUE_VALUE }
};

struct ILP_Object ILP_object_false = {
     ILP_BOOLEAN_KIND,
     { ILP_BOOLEAN_FALSE_VALUE }
};

/** Une fonction pour stopper abruptement l'application. */

ILP_Object
ILP_die (char *message)
{
     fputs(message, stderr);
     fputc('\n', stderr);
     fflush(stderr);
     exit(EXIT_FAILURE);
}

/** Ce n'est pas une vraie allocation mais une simple conversion. */

ILP_Object
ILP_make_boolean (int b)
{
     if ( b ) {
          return ILP_TRUE;
     } else {
          return ILP_FALSE;
     }
}

ILP_Object
ILP_make_integer (int d)
{
     ILP_Object result = ILP_AllocateInteger();
     result->_content.asInteger = d;
     return result;
}

ILP_Object
ILP_make_float (double d)
{
     ILP_Object result = ILP_AllocateFloat();
     result->_content.asFloat = d;
     return result;
}

ILP_Object
ILP_pi ()
{
     static ILP_Object object_pi = NULL;
     if ( object_pi == NULL ) {
          object_pi = ILP_make_float(ILP_PI_VALUE);
     }
     return object_pi;
}


ILP_Object
ILP_make_string (char *s)
{
     int size = strlen(s);
     ILP_Object result = ILP_AllocateString(size);
     result->_content.asString._size = size;
     memmove(result->_content.asString.asCharacter, s, size);
     return result;
}

/** String primitives */

ILP_Object
ILP_concatenate_strings (ILP_Object o1, ILP_Object o2)
{
     int size1 = o1->_content.asString._size;
     int total_size = size1 + o2->_content.asString._size;
     ILP_Object result = ILP_AllocateString(total_size);
     memmove(&(result->_content.asString.asCharacter[0]),
             o1->_content.asString.asCharacter,
             o1->_content.asString._size);
     memmove(&(result->_content.asString.asCharacter[size1]),
             o2->_content.asString.asCharacter,
```

```
            o2->_content.asString._size);
     return result;
}

/** Opérateurs unaires.
 La négation de flottant manquait comme vu par <Clement.Bossut@etu.upmc.fr> */

ILP_Object
ILP_make_opposite (ILP_Object o)
{
     if ( ILP_isInteger(o) ) {
          ILP_Object result = ILP_AllocateInteger();
          result->_content.asInteger = (- o->_content.asInteger);
          return result;
     } else if ( ILP_isFloat(o) ) {
          ILP_Object result = ILP_AllocateFloat();
          result->_content.asFloat = (- o->_content.asFloat);
          return result;
     } else {
          return ILP_domain_error("Not a number", o);
     }
}

ILP_Object
ILP_make_negation (ILP_Object o)
{
     /* Opere sur toutes les representations possibles de Vrai */
     if ( ILP_isEquivalentToTrue(o) ) {
          return ILP_FALSE;
     } else {
          return ILP_TRUE;
     }
}

/** Opérateurs binaires. */

/* DefineOperator(addition, +) est incorrect car + représente également
 * la concaténation des chaînes de caractères. */

ILP_Object
ILP_make_addition (ILP_Object o1, ILP_Object o2)
{
  if ( ILP_isInteger(o1) ) {
    if ( ILP_isInteger(o2) ) {
      ILP_Object result = ILP_AllocateInteger();
      result->_content.asInteger = o1->_content.asInteger + o2->_content.asInteger;
      return result;
    } else if ( ILP_isFloat(o2) ) {
      ILP_Object result = ILP_AllocateFloat();
      result->_content.asFloat = o1->_content.asInteger + o2->_content.asFloat;
      return result;
    } else {
      return ILP_domain_error("Not a number", o2);
    }
  } else if ( ILP_isFloat(o1) ) {
    if ( ILP_isInteger(o2) ) {
      ILP_Object result = ILP_AllocateFloat();
      result->_content.asFloat = o1->_content.asFloat + o2->_content.asInteger;
      return result;
    } else if ( ILP_isFloat(o2) ) {
      ILP_Object result = ILP_AllocateFloat();
      result->_content.asFloat = o1->_content.asFloat + o2->_content.asFloat;
      return result;
    } else {
      return ILP_domain_error("Not a number", o2);
    }

  } else if ( ILP_isString(o1) ) {
    if ( ILP_isString(o2) ) {
      return ILP_concatenate_strings(o1, o2);
    } else {
      return ILP_domain_error("Not a string", o2);
    }
  } else {
    return ILP_domain_error("Not addable", o1);
  }
}

#define DefineOperator(name,op)                                         \
ILP_Object                                                              \
```



```
ILP_make_##name (ILP_Object o1, ILP_Object o2)                          \
{                                                                       \
     if ( ILP_isInteger(o1) ) {                                         \
          if ( ILP_isInteger(o2) ) {                                    \
               ILP_Object result = ILP_AllocateInteger();               \
               result->_content.asInteger =                             \
                    o1->_content.asInteger op o2->_content.asInteger;   \
               return result;                                           \
          } else if ( ILP_isFloat(o2) ) {                               \
               ILP_Object result = ILP_AllocateFloat();                 \
               result->_content.asFloat =                               \
                    o1->_content.asInteger op o2->_content.asFloat;     \
               return result;                                           \
          } else {                                                      \
               return ILP_domain_error("Not a number", o2);             \
          }                                                             \
     } else if ( ILP_isFloat(o1) ) {                                    \
          if ( ILP_isInteger(o2) ) {                                    \
               ILP_Object result = ILP_AllocateFloat();                 \
               result->_content.asFloat =                               \
                    o1->_content.asFloat op o2->_content.asInteger;     \
               return result;                                           \
          } else if ( ILP_isFloat(o2) ) {                               \
               ILP_Object result = ILP_AllocateFloat();                 \
               result->_content.asFloat =                               \
                    o1->_content.asFloat op o2->_content.asFloat;       \
               return result;                                           \
          } else {                                                      \
               return ILP_domain_error("Not a number", o2);             \
          }                                                             \
     } else {                                                           \
          return ILP_domain_error("Not a number", o1);                  \
     }                                                                  \
}

DefineOperator(subtraction, -)
DefineOperator(multiplication, *)
DefineOperator(division, /)

/* DefineOperator(modulo, %) est incorrect car le modulo ne se prend
 * que sur de entiers. */

ILP_Object
ILP_make_modulo (ILP_Object o1, ILP_Object o2)
{
     if ( ILP_isInteger(o1) ) {
          if ( ILP_isInteger(o2) ) {
               ILP_Object result = ILP_AllocateInteger();
               result->_content.asInteger =
                    o1->_content.asInteger % o2->_content.asInteger;
               return result;
          } else {
               return ILP_domain_error("Not an integer", o2);
          }
     } else {
          return ILP_domain_error("Not an integer", o1);
     }
}

#define DefineComparator(name,op)                                       \
ILP_Object                                                              \
ILP_compare_##name (ILP_Object o1, ILP_Object o2)                       \
{                                                                       \
     if ( ILP_isInteger(o1) ) {                                         \
          if ( ILP_isInteger(o2) ) {                                    \
               return ILP_make_boolean(                                 \
                    o1->_content.asInteger op o2->_content.asInteger);  \
          } else if ( ILP_isFloat(o2) ) {                               \
               return ILP_make_boolean(                                 \
                    o1->_content.asInteger op o2->_content.asFloat);    \
          } else {                                                      \
               return ILP_domain_error("Not a number", o2);             \
          }                                                             \
     } else if ( ILP_isFloat(o1) ) {                                    \
          if ( ILP_isInteger(o2) ) {                                    \
               return ILP_make_boolean(                                 \
                    o1->_content.asFloat op o2->_content.asInteger);    \
          } else if ( ILP_isFloat(o2) ) {                               \
               return ILP_make_boolean(                                 \
                    o1->_content.asFloat op o2->_content.asFloat);      \
```

```
          } else {                                                    \
              return ILP_domain_error("Not a number", o2);            \
          }                                                           \
     } else {                                                         \
          return ILP_domain_error("Not a number", o1);                \
     }                                                                \
}

DefineComparator(less_than, <)
DefineComparator(less_than_or_equal, <=)
DefineComparator(equal, ==)
DefineComparator(greater_than, >)
DefineComparator(greater_than_or_equal, >=)
DefineComparator(not_equal, !=)

/** Primitives */

ILP_Object
ILP_newline ()
{
     fputc('\n', stdout);
     return ILP_FALSE;
}

ILP_Object
ILP_print (ILP_Object o)
{
  switch (o->_kind) {
  case ILP_INTEGER_KIND: {
    fprintf(stdout, "%d", o->_content.asInteger);
    break;
  }
  case ILP_FLOAT_KIND: {
    /* Supprimer les blancs du debut */
    char buffer[15];
    char *p = &buffer[0];
    sprintf(buffer, "%12.5g", o->_content.asFloat);
    while ( isspace(*p) ) {
      p++;
    }
    fprintf(stdout, "%s", p);
    break;
  }
  case ILP_BOOLEAN_KIND: {
    fprintf(stdout, "%s", (ILP_isTrue(o) ? "true" : "false"));
    break;
  }
  case ILP_STRING_KIND: {
    fprintf(stdout, "%s", o->_content.asString.asCharacter);
    break;
  }
  default: {
    fprintf(stdout, "<0x%x:0x%p>", o->_kind, (void*) o);
    break;
  }
  }
  return ILP_FALSE;
}

/* end of ilp.c */
```

C/ilp.h

```
#ifndef ILP_H
#define ILP_H

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

/** Definition additionnelle pour enrichir ILP_Object */

#ifndef ILP_OTHER_KINDS
#define ILP_OTHER_KINDS
#define ILP_OTHER_STRUCTS
#endif /* ILP_OTHER_KINDS */

/** Le type général des fonctions d'ILP (d'arité quelconque). */
```



```
typedef struct ILP_Object* (*ILP_general_function)();

/** Il y a cinq types de valeurs pour l'instant repérées par ces
 * constantes (elles figureront dans le champ _kind d'ILP_Object.
 */

enum ILP_Kind {
     ILP_BOOLEAN_KIND       = 0xab010ba,
     ILP_INTEGER_KIND       = 0xab020ba,
     ILP_FLOAT_KIND         = 0xab030ba,
     ILP_STRING_KIND        = 0xab040ba,
     ILP_PRIMITIVE_KIND     = 0xab050ba
     ILP_OTHER_KINDS
};

/** Toutes les valeurs manipulées ont cette forme.
 *
 * Un premier champ indique leur nature ce qui permet de décoder le
 * champ qui suit. Les chaînes de caractères sont préfixées par leur
 * longueur.

 * NOTE: asBoolean est la première possibilité ce qui est nécessaire pour
 * l'allocation statiques des booléens (cf. ilp.c). Ce ne l'était pas
 * auparavant ce qui créait une bogue pour MacOSX (ILP_TRUE valait faux!).
 */

typedef struct ILP_Object {
     enum ILP_Kind      _kind;
     union {
          unsigned char asBoolean;
          int           asInteger;
          double        asFloat;
          struct asString {
               int      _size;
               char     asCharacter[1];
          } asString;
          struct asPrimitive {
               ILP_general_function  _code;
          } asPrimitive;
          ILP_OTHER_STRUCTS
     }                  _content;
} *ILP_Object;

/** -----------------------------------------
 * Des macros pour manipuler toutes ces valeurs.
 */

/** Booléens. */

/** Il y a deux sortes de booléens et ces deux constantes les repèrent. */

enum ILP_BOOLEAN_VALUE {
     ILP_BOOLEAN_FALSE_VALUE = 0,
     ILP_BOOLEAN_TRUE_VALUE  = 1
};

#define ILP_Boolean2ILP(b) \
  ILP_make_boolean(b)

#define ILP_isBoolean(o) \
  ((o)->_kind == ILP_BOOLEAN_KIND)

#define ILP_isTrue(o) \
  (((o)->_kind == ILP_BOOLEAN_KIND) && \
   ((o)->_content.asBoolean))

#define ILP_TRUE  (&ILP_object_true)
#define ILP_FALSE (&ILP_object_false)

#define ILP_isEquivalentToTrue(o) \
  ((o) != ILP_FALSE)

#define ILP_CheckIfBoolean(o) \
  if ( ! ILP_isBoolean(o) ) { \
       ILP_domain_error("Not a boolean", o); \
  };

/** Entiers */
```

```
#define ILP_Integer2ILP(i) \
  ILP_make_integer(i)

#define ILP_AllocateInteger() \
  ILP_malloc(sizeof(struct ILP_Object), ILP_INTEGER_KIND)

#define ILP_isInteger(o) \
  ((o)->_kind == ILP_INTEGER_KIND)

#define ILP_CheckIfInteger(o) \
  if ( ! ILP_isInteger(o) ) { \
       ILP_domain_error("Not an integer", o); \
  };

/** Flottants */

#define ILP_Float2ILP(f) \
  ILP_make_float(f)

#define ILP_AllocateFloat() \
  ILP_malloc(sizeof(struct ILP_Object), ILP_FLOAT_KIND)

#define ILP_isFloat(o) \
  ((o)->_kind == ILP_FLOAT_KIND)

#define ILP_CheckIfFloat(o) \
  if ( ! ILP_isFloat(o) ) { \
       ILP_domain_error("Not a float", o); \
  };

#define ILP_PI_VALUE 3.1415926535
#define ILP_PI (ILP_pi())

/** Chaînes de caractères */

#define ILP_String2ILP(s) \
  ILP_make_string(s)

#define ILP_AllocateString(length) \
  ILP_malloc(sizeof(struct ILP_Object) \
             + (sizeof(char) * (length)), ILP_STRING_KIND)

#define ILP_isString(o) \
  ((o)->_kind == ILP_STRING_KIND)

#define ILP_CheckIfString(o) \
  if ( ! ILP_isString(o) ) { \
       ILP_domain_error("Not a string", o); \
  };

/** Opérateurs unaires */

#define ILP_Opposite(o) \
  ILP_make_opposite(o)

#define ILP_Not(o) \
  ILP_make_negation(o)

/** Opérateurs binaires */

#define ILP_Plus(o1,o2) \
  ILP_make_addition(o1, o2)

#define ILP_Minus(o1,o2) \
  ILP_make_subtraction(o1, o2)

#define ILP_Times(o1,o2) \
  ILP_make_multiplication(o1, o2)

#define ILP_Divide(o1,o2) \
  ILP_make_division(o1, o2)

#define ILP_Modulo(o1,o2) \
  ILP_make_modulo(o1, o2)

/** modulo a faire */

#define ILP_LessThan(o1,o2) \
```



```
  ILP_compare_less_than(o1,o2)

#define ILP_LessThanOrEqual(o1,o2) \
  ILP_compare_less_than_or_equal(o1,o2)

#define ILP_GreaterThan(o1,o2) \
  ILP_compare_greater_than(o1,o2)

#define ILP_GreaterThanOrEqual(o1,o2) \
  ILP_compare_greater_than_or_equal(o1,o2)

#define ILP_Equal(o1,o2) \
  ILP_compare_equal(o1,o2)

#define ILP_NotEqual(o1,o2) \
  ILP_compare_not_equal(o1,o2)

/** Primitives:
 * Les fonctions ILP_print() et ILP_newline() sont définies dans ilp.c
 */

#define ILP_globalIfInitialized(n) \
     (((n)!=NULL)?(n):(ILP_error("Uninitialized " #n " variable!")))

typedef ILP_Object (*ILP_Primitive) ();

extern struct ILP_Object ILP_object_true;
extern struct ILP_Object ILP_object_false;
extern struct ILP_Object ILP_object_pi;
extern ILP_Object ILP_die (char *message);
extern ILP_Object ILP_make_boolean (int b);
extern ILP_Object ILP_make_integer (int d);
extern ILP_Object ILP_make_float (double d);
extern ILP_Object ILP_pi ();
extern ILP_Object ILP_make_string (char *s);
extern ILP_Object ILP_make_opposite (ILP_Object o);
extern ILP_Object ILP_make_negation (ILP_Object o);
extern ILP_Object ILP_make_addition (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_concatenate_strings (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_make_subtraction (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_make_multiplication (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_make_division (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_make_modulo (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_compare_less_than (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_compare_less_than_or_equal (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_compare_equal (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_compare_greater_than (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_compare_greater_than_or_equal (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_compare_not_equal (ILP_Object o1, ILP_Object o2);
extern ILP_Object ILP_newline ();
extern ILP_Object ILP_print (ILP_Object o);

#endif /* ILP_H */

/* end of ilp.h */
```

C/ilpAlloc.c

```
/** Ce fichier constitue la bibliothèque d'exécution d'ILP. */

#include <stdlib.h>
#include "ilp.h"
#include "ilpAlloc.h"
#include "ilpBasicError.h"

char *ilpAlloc_Id = "$Id: ilpAlloc.c 768 2008-11-02 15:56:00Z queinnec $";

/** Allouer un objet d'ILP avec une taille et une étiquette. */

ILP_Object
ILP_malloc (int size, enum ILP_Kind kind)
{
     ILP_Object result = malloc(size);
     if ( result == NULL ) {
          return ILP_error("Memory exhaustion");
     };
     result->_kind = kind;
     return result;
```

```
}

/* end of ilpAlloc.c */
```

C/ilpAlloc.h

```
#ifndef ILP_ALLOC_H
#define ILP_ALLOC_H

extern ILP_Object ILP_malloc (int size, enum ILP_Kind kind);

#endif /* ILP_ALLOC_H */
/* end of ilpAlloc.h */
```

C/ilpBasicError.c

```
/** Ce fichier constitue la bibliothèque d'exécution d'ILP. */

#include <stdlib.h>
#include <stdio.h>
#include "ilp.h"
#include "ilpBasicError.h"

char *ilpBasicError_Id = "$Id: ilpBasicError.c 822 2009-10-07 08:04:40Z queinnec $";

/**
 * Signalement de problèmes.
 *
 * Cette fonction ne renvoit rien mais le typage est meilleur si on
 * l'utilise après un return (cf. exemples plus bas).
 */

ILP_Object
ILP_error (char *message)
{
     ILP_die(message);
     /** NOT REACHED */
     return NULL;
}

#define BUFFER_LENGTH 1000

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

ILP_Object
ILP_domain_error (char *message, ILP_Object o)
{
     char buffer[BUFFER_LENGTH];
     snprintf(buffer, BUFFER_LENGTH, "%s\nCulprit: 0x%p\n",
              message, (void*) o);
     ILP_die(buffer);
     /** NOT REACHED */
     return NULL;
}

/* end of ilpBasicError.c */
```

C/ilpBasicError.h

```
#ifndef ILP_BASIC_ERROR_H
#define ILP_BASIC_ERROR_H

extern ILP_Object ILP_error (char *message);
extern ILP_Object ILP_domain_error (char *message, ILP_Object o);

#endif /* ILP_BASIC_ERROR_H */
/* end of ilpBasicError.h */
```