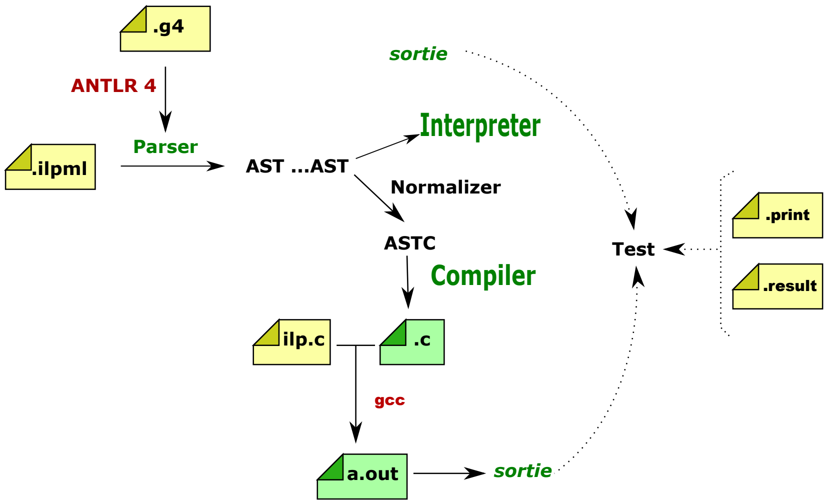
Master Informatique 2016-2017
Spécialité STL
Développement des langages de programmation
DLP – 4I501

Carlos Agon
agonc@ircam.fr

Plan du cours 4

- Génération de code
- Récapitulation

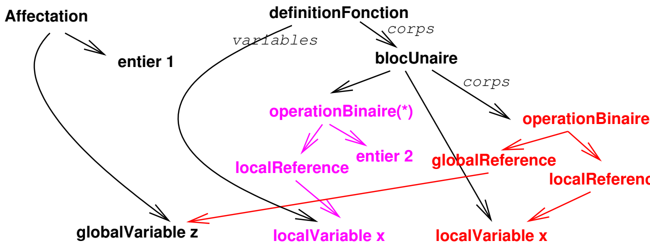
Grand schéma



Normalisation

Partage physique des objets représentant les variables.
Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.

```
z = 1;
function f(x) {
  let x = 2*x
  in z+x
}
```



L'identification des variables :

- améliore la comparaison (et notamment la vitesse de l'interprète)
- réalise l'alpha-conversion (l'adresse est le nom).

Prévention des conflits de noms

- Deux références à une même variable (locale ou globale) sont représentées par le même objet en mémoire.
- Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.
- Les séquences d'une seule expression sont normalisées à cette seule expression.

Comparaison

Comparaison physique plutôt que structurelle :

```
// depuis LexicalEnvironment
  public Object lookup (IVariable otherVariable)
    throws EvaluationException {
    if ( variable == otherVariable ) {
      return value;
    } else {
      return next.lookup(otherVariable);
    }
  }
```

Le visiteur normalizer

```
public class Normalizer implements
 IASTvisitor
 <IASTexpression, INormalizationEnvironment, CompilationException> {

    public Normalizer (INormalizationFactory factory) {
        this.factory = factory;
        this.globalVariables = new HashSet<>();
    }
    protected final INormalizationFactory factory;
    protected final Set<IASTvariable> globalVariables;


    public IASTCprogram transform(IASTprogram program)
            throws CompilationException {
    INormalizationEnvironment env = NormalizationEnvironment.EMPTY;

    IASTexpression body = program.getBody();
    IASTexpression newbody = body.accept(this, env);
    return factory.newProgram(newbody);
    }
```

```
public IASTexpression
visit(IASTboolean iast, INormalizationEnvironment env)
  throws CompilationException {
        return iast;
}

public IASTvariable
visit(IASTvariable iast, INormalizationEnvironment env)
  throws CompilationException {
  try {
    return env.renaming(iast); // look for a local variable
  } catch (NoSuchLocalVariableException exc) {
  for ( IASTvariable gv : globalVariables ) {
    if ( iast.getName().equals(gv.getName()) ) {
      return gv;
    }
  }
  IASTvariable gv = factory.newGlobalVariable(iast.getName());
  globalVariables.add(gv);
  return gv;
  }
 }
```

```java
public IASTexpression
 visit(IASTblock iast, INormalizationEnvironment env)
      throws CompilationException {

  INormalizationEnvironment newenv = env;
  IASTbinding[] bindings = iast.getBindings();
  IASTCblock.IASTCbinding[] newbindings =
          new IASTCblock.IASTCbinding[bindings.length];
  for ( int i=0 ; i<bindings.length ; i++ ) {
      IASTbinding binding = bindings[i];
      IASTexpression expr = binding.getInitialisation();
      IASTexpression newexpr = expr.accept(this, env);
      IASTvariable variable = binding.getVariable();
      IASTvariable newvariable =
              factory.newLocalVariable(variable.getName());
      newenv = newenv.extend(variable, newvariable);
      newbindings[i] =
            factory.newBinding(newvariable, newexpr);
   }
   IASTexpression newbody =
        iast.getBody().accept(this, newenv);
   return factory.newBlock(newbindings, newbody);
  }
```

Compilation

Le compilateur doit avoir connaissance des environnements en jeu. Il est initialement créé avec un environnement global :

Ressource: com.paracamplus.ilp1.compiler.compiler

```
public class Compiler
implements
IASTCvisitor<Void, Compiler.Context, CompilationException> {

public Compiler (IOperatorEnvironment ioe,
         IGlobalVariableEnvironment igve ) {
           this.operatorEnvironment = ioe;
           this.globalVariableEnvironment = igve;
      }
protected final
   IOperatorEnvironment operatorEnvironment;
protected final
   IGlobalVariableEnvironment globalVariableEnvironment;
```

Environnement global

- Compiler les appels aux primitives,
- Compiler les appels aux opérateurs,
- Vérifier l’existence, l’arité.

Environnement global pour les primitives

```java
public interface IGlobalVariableEnvironment {
    void addGlobalVariableValue (String variableName, String cName);
    void addGlobalFunctionValue (IPrimitive primitive);
    boolean isPrimitive(IASTvariable variable);
    IPrimitive getPrimitiveDescription(IASTvariable variable);
    String getCName (IASTvariable variable);
}
```

```java
public class GlobalVariableEnvironment
implements IGlobalVariableEnvironment {

    public GlobalVariableEnvironment () {
        this.globalVariableEnvironment = new HashMap<>();
        this.globalFunctionEnvironment = new HashMap<>();
    }
    private final Map<String, String> globalVariableEnvironment;
    private final Map<String, IPrimitive> globalFunctionEnvironment;

    public void addGlobalVariableValue(String variableName, String cName) {
        globalVariableEnvironment.put(variableName, cName);
    }

    public void addGlobalFunctionValue(IPrimitive primitive) {
        globalFunctionEnvironment.put(primitive.getName(), primitive);
    }
```

Primitives

```java
public class Primitive implements IPrimitive {

    public Primitive(String name, String cName, int arity) {
        this.name = name;
        this.cName = cName;
        this.arity = arity;
    }
    private final String name;
    private final String cName;
    private final int arity;

    public String getName() {
        return name;
    }

    public String getCName() {
        return cName;
    }

    public int getArity () {
        return arity;
    }
}
```

Initialisation de GlobalVariableEnvironment

Ressource: com.paracamplus.ilp1.compiler.compiler.GlobalVariableStuff

```
public class GlobalVariableStuff {

public static void fillGlobalVariables
    (IGlobalVariableEnvironment env) {
 env.addGlobalVariableValue("pi", "ILP_PI");

 env.addGlobalFunctionValue(
    new Primitive("print", "ILP_print", 1));

 env.addGlobalFunctionValue(
   new Primitive("newline", "ILP_newline", 0));

    }
}
```

Environnement global pour les opérateurs

```java
public interface IOperatorEnvironment {
    String getUnaryOperator (IASToperator operator)
            throws CompilationException;
    String getBinaryOperator (IASToperator operator)
            throws CompilationException;
    void addUnaryOperator (String operator, String cOperator)
            throws CompilationException;
    void addBinaryOperator (String operator, String cOperator)
            throws CompilationException;
}
```

```java
public class OperatorEnvironment implements IOperatorEnvironment {

    public OperatorEnvironment () {
        this.unaryOperatorEnvironment = new HashMap<>();
        this.binaryOperatorEnvironment = new HashMap<>();
    }
    private final Map<String, String> unaryOperatorEnvironment;
    private final Map<String, String> binaryOperatorEnvironment;

    ...
}
```

Initialisation de OperatorEnvironment

Ressource: com.paracamplus.ilp1.compiler.compiler.OperatorStuff

```
public class OperatorStuff {

    public static void fillUnaryOperators (IOperatorEnvironment env)
            throws CompilationException {
        env.addUnaryOperator("-", "ILP_Opposite");
        env.addUnaryOperator("!", "ILP_Not");
    }

    public static void fillBinaryOperators (IOperatorEnvironment env)
            throws CompilationException {
        env.addBinaryOperator("+", "ILP_Plus");
        env.addBinaryOperator("*", "ILP_Times");
        env.addBinaryOperator("/", "ILP_Divide");
        env.addBinaryOperator("-", "ILP_Minus");
        ...
    }
}
```

Compilation

```
public class Compiler
implements
IASTCvisitor<Void, Compiler.Context, CompilationException> {

public Compiler (IOperatorEnvironment ioe,
                    IGlobalVariableEnvironment igve ) {
        this.operatorEnvironment = ioe;
        this.globalVariableEnvironment = igve;
    }

protected Writer out;

public String compile(IASTprogram program)
            throws CompilationException {

        IASTCprogram newprogram = normalize(program);
        ...
        Context context = new Context(NoDestination.NO_DESTINATION);
        StringWriter sw = new StringWriter();
        out = new BufferedWriter(sw);
        visit(newprogram, context);
        out.flush();
       ...
        return sw.toString();
    }
```

IASTCVisitor

```

import com.paracamplus.ilp1.interfaces.IASTvisitor;

public interface
 IASTCvisitor<Result, Data, Anomaly extends Throwable>
 extends IASTvisitor<Result, Data, Anomaly> {

Result visit(IASTCglobalVariable iast, Data data)
  throws Anomaly;
Result visit(IASTClocalVariable iast, Data data)
  throws Anomaly;
Result visit(IASTCprimitiveInvocation iast, Data data)
  throws Anomaly;
Result visit(IASTCvariable iast, Data data)
  throws Anomaly;
Result visit(IASTCcomputedInvocation iast, Data data)
  throws Anomaly;

}
```

Nouvelles interfaces pour l'AST

```
public interface IASTCvisitable extends IASTvisitable {
  <Result, Data, Anomaly extends Throwable>
  Result accept(IASTCvisitor<Result, Data, Anomaly> visitor,
  Data data) throws Anomaly;
}

public abstract interface IASTCvariable
extends IASTvariable, IASTCvisitable {
  boolean isMutable();
  void setMutable();
}

public interface IASTCglobalVariable extends IASTCvariable {
...
}

public interface IASTClocalVariable extends IASTCvariable {
...
}
```

Nouvelles implementations

```
public class ASTCprogram extends ASTprogram
implements IASTCprogram {

public ASTCprogram (IASTexpression expression) {
        super(expression);
        this.globalVariables = new HashSet<>();
}
protected Set<IASTCglobalVariable> globalVariables;

public Set<IASTCglobalVariable> getGlobalVariables() {
        return globalVariables;
}

public void setGlobalVariables
            (Set<IASTCglobalVariable> gvs) {
        globalVariables = gvs;
}
}
```

Qui fait l'instance du ASTCprogram ? La classe Parser ?

Compilation

```
public class Compiler
implements
IASTCvisitor<Void, Compiler.Context, CompilationException> {

public Compiler (IOperatorEnvironment ioe,
                 IGlobalVariableEnvironment igve ) {
        this.operatorEnvironment = ioe;
        this.globalVariableEnvironment = igve;
    }

protected Writer out;

public String compile(IASTprogram program)
            throws CompilationException {

        IASTCprogram newprogram = normalize(program);
        ...
        Context context = new Context(NoDestination.NO_DESTINATION);
        visit(newprogram, context);
        out.flush();
       ...
        return sw.toString();
    }
```

Context

```
    public static class Context {
        public Context (IDestination destination) {
            this.destination = destination;
        }
        public IDestination destination;
        public static AtomicInteger counter = new AtomicInteger(0);

        public IASTvariable newTemporaryVariable () {
            int i = counter.incrementAndGet();
            return new ASTvariable("ilptmp" + i);
        }

        public Context redirect (IDestination d) {
            if ( d == destination ) {
                return this;
            } else {
                return new Context(d);
            }
        }
    }
```

Destination

Toute expression doit rendre un résultat.
Toute fonction doit rendre la main avec `return`.
La **destination** indique que faire de la valeur d'une expression ou d'une instruction.
Notations pour ILP1 :

$\overset{\longrightarrow}{\textit{expression}}$	laisser la valeur en place
$\overset{\longrightarrow\texttt{return}}{\textit{expression}}$	sortir de la fonction avec la valeur
$\overset{\longrightarrow\texttt{(x =)}}{\textit{expression}}$	assigner la valeur à la variable x

Destination

```
public class NoDestination implements IDestination {
  public static final NoDestination NO_DESTINATION =
            new NoDestination();
  private NoDestination () {}
  public String compile() {
            return "";
```

```
public class AssignDestination implements IDestination {
  public AssignDestination (IASTvariable variable) {
    this.variable = variable;
  private final IASTvariable variable;
  public String compile() {
    return variable.getMangledName() + " = ";
```

```
public class ReturnDestination implements IDestination {
  private ReturnDestination () {}
  public static final ReturnDestination RETURN_DESTINATION =
             new  ReturnDestination();
  public String compile() {
          return "return ";
```

Génération de code

On est prêt pour la génération de code, mais ... pas besoin d'un environnement lexicale ?

```
   public Void visit(IASTCprogram iast, Context context) throws CompilationException {
        emit(cProgramPrefix);
        emit(cBodyPrefix);
        Context cr = context.redirect(ReturnDestination.RETURN_DESTINATION);
        iast.getBody().accept(this, cr);
        emit(cBodySuffix);
        emit(cProgramSuffix);
        return null;
  }
```

```
    protected String cProgramPrefix = ""
            + "#include <stdio.h> \n"
            + "#include <stdlib.h> \n"
            + "#include \"ilp.h\" \n\n";
    protected String cBodyPrefix = "\n"
            + "ILP_Object ilp_program () \n"
            + "{ \n";
    protected String cBodySuffix = "\n"
            + "} \n";
    protected String cProgramSuffix = "\n"
           "int main (int argc, char *argv[]) \n"
            + "{ \n"
            + "   ILP_print(ilp_program()); \n"
            + "   ILP_newline(); \n"
            + "   return EXIT_SUCCESS; \n"
            + "} \n";
```

Habillage du code

```
#include <stdio.h>
#include <stdlib.h>
#include "ilp.h"

ILP_Object
ilp_program()
{
...
}

int
main(int argc, char *argv[])
{
  ILP_START_GC;
  ILP_print(ilp_program());
  ILP_newline();
  return EXIT_SUCCESS;
}
```

Grandes règles

- les variables ILP sont compilées en variables C
- les expressions ILP sont compilées en expressions C ou en instructions C dépendant du context

Compilation d'une constante

$$\overset{\longrightarrow\mathtt{d}}{\mathit{constante}}$$

```
d   ILP_Integer2ILP(constanteEntière)
                       /* ou CgenerationException */
d   ILP_Float2ILP(constanteFlottante)
d   ILP_TRUE
d   ILP_FALSE
d   ILP_String2ILP("constanteChaînePlusProtection")
```

Compilation d'un Integer

```
public Void visit(IASTinteger iast, Context context)
            throws CompilationException {

        emit(context.destination.compile());
        emit("ILP_Integer2ILP(");
        emit(iast.getValue().toString());
        emit("); \n");
        return null;
    }
```

Compilation d'une variable

$$\overrightarrow{\mathit{variable}}^{\,\mathtt{d}}$$

```
d variable      /* ou CgenerationException */
```

Attention aussi une conversion (*mangling*) est parfois nécessaire !

Compilation d'une invocation

On utilise la force du langage C. La bibliothèque d'exécution comprend également les implantations des fonctions prédéfinies `print` et `newline` (respectivement `ILP_print` et `ILP_newline`).

invocation = (fonction, argument1, ...)

$$\overset{\longrightarrow \mathtt{d}}{\textit{invocation}}$$

```
d fonctionCorrespondante(
              →
              argument1 ,
              →
              argument2 ,
              ... )
```

Compilation d'une opération

À chaque opérateur d'ILP1 correspond une fonction dans la bibliothèque d'exécution.
operation = (opérateur, opérandeGauche, opérandeDroit)

$$\overrightarrow{op\acute{e}ration}^{\mathtt{d}}$$

```
d fonctionCorrespondante(
```
$$\overrightarrow{op\acute{e}randeGauche},$$
$$\overrightarrow{op\acute{e}randeDroit})$$

Ainsi, `+` correspond à `ILP_Plus`, `-` correspond à `ILP_Minus`, etc.

Compilation d'une opération

```
public Void visit(IASTbinaryOperation iast, Context context)
  throws CompilationException {
  IASTvariable tmp1 = context.newTemporaryVariable();
  IASTvariable tmp2 = context.newTemporaryVariable();
  emit("{ \n");
  emit("  ILP_Object " + tmp1.getMangledName() + "; \n");
  emit("  ILP_Object " + tmp2.getMangledName() + "; \n");
  Context c1 = context.redirect(new AssignDestination(tmp1));
  iast.getLeftOperand().accept(this, c1);
  Context c2 = context.redirect(new AssignDestination(tmp2));
  iast.getRightOperand().accept(this, c2);
  String cName = operatorEnvironment.getBinaryOperator
                                              (iast.getOperator());
  emit(context.destination.compile());
  emit(cName);
  emit("(");
  emit(tmp1.getMangledName());
  emit(", ");
  emit(tmp2.getMangledName());
  emit(");\n");
  emit("} \n");
  return null;
}
```

Compilation de l'alternative

alternative = (condition, consequence, alternant)

$$\overrightarrow{\mathit{alternative}}^{\mathrm{d}}$$

```
if ( ILP_isEquivalentToTrue( $\overrightarrow{\mathit{condition}}$ ) ) {
   $\overrightarrow{\mathit{consequence}}^{\mathrm{d}}$ ;
} else {
   $\overrightarrow{\mathit{alternant}}^{\mathrm{d}}$ ;
}
```

Compilation de l'alternative

```
public void visit(IASTalternative iast, Context context)
            throws CompilationException {

IASTvariable tmp1 = context.newTemporaryVariable();
emit("{ \n");
emit("  ILP_Object " + tmp1.getMangledName() + "; \n");
Context c = context.redirect(new AssignDestination(tmp1));
iast.getCondition().accept(this, c);
emit("  if ( ILP_isEquivalentToTrue(");
emit(tmp1.getMangledName());
emit(" ) ) {\n");
iast.getConsequence().accept(this, context);
if ( iast.isTernary() ) {
    emit("\n  } else {\n");
    iast.getAlternant().accept(this, context);
 }
emit("\n  }\n}\n");
return null;
}
```

Compilation de la séquence

sequence = (instruction1, ... dernièreInstruction)

$$\overset{\longrightarrow \mathtt{d}}{\textit{séquence}}$$

```
{ ILP_Object temp;
    ⟶(temp =)
   instruction1 ;
    ⟶(temp =)
   instruction2 ;
   ...
              ⟶d
   dernièreInstruction ;
}
```

Compilation de la séquence

```
public Void visit(IASTsequence iast, Context context)
throws CompilationException {

IASTvariable tmp = context.newTemporaryVariable();
IASTexpression[] expressions = iast.getExpressions();
Context c = context.redirect(new AssignDestination(tmp));
emit("{ \n");
emit("  ILP_Object " + tmp.getMangledName() + "; \n");
for ( IASTexpression expr : expressions ) {
  expr.accept(this, c);
}
emit(context.destination.compile());
emit(tmp.getMangledName());
emit("; \n} \n");
return null;
}
```

Compilation de la séquence

```
(
"un"; "deux"; "trois"
)
```

```
    {
    ILP_Object  ilptmp117;
    ilptmp117 = ILP_String2ILP("Un,");
    ilptmp117 = ILP_String2ILP("Deux ");
    ilptmp117 = ILP_String2ILP("Trois,");
    return ilptmp117;
    }
```

Compilation du bloc unaire I

Comme au judo, utiliser la force du langage cible !
bloc = (variable, initialisation, corps)
corps = (instruction1, ... dernièreInstruction)

$$\overrightarrow{bloc}^{\,\mathtt{d}}$$

`{`

  `ILP_Object variable =` $\overrightarrow{initialisation}$ `;`

  `ILP_Object temp;`

  $\overrightarrow{instruction1}^{\,\mathtt{(temp\ =)}}$ `;`

  $\overrightarrow{instruction2}^{\,\mathtt{(temp\ =)}}$ `;`

  `...`

  $\overrightarrow{derni\grave{e}reInstruction}^{\,\mathtt{d}}$ `;`

`}`

Compilation du bloc unaire II

$\overrightarrow{bloc}^{\mathrm{d}}$

```
{
  ILP_Object temporaire = $\overrightarrow{initialisation}$ ;
  ILP_Object variable = temporaire;

  ILP_Object temp;
  $\overrightarrow{instruction1}^{\mathrm{(temp\ =)}}$ ;
  $\overrightarrow{instruction2}^{\mathrm{(temp\ =)}}$ ;
  ...
  $\overrightarrow{dernièreInstruction}^{\mathrm{d}}$ ;
}
```

Compilation du bloc unaire II

```
    public void visit(IASTblock iast, Context context) throws CompilationException {
        emit("{ \n");
        IASTbinding[] bindings = iast.getBindings();
        IASTvariable[] tmps = new IASTvariable[bindings.length];
        for ( int i=0 ; i<bindings.length ; i++ ) {
            IASTvariable tmp = context.newTemporaryVariable();
            emit("  ILP_Object " + tmp.getMangledName() + "; \n");
            tmps[i] = tmp;
        }
        for ( int i=0 ; i<bindings.length ; i++ ) {
            IASTbinding binding = bindings[i];
            IASTvariable tmp = tmps[i];
            Context c = context.redirect(new AssignDestination(tmp));
            binding.getInitialisation().accept(this, c);
        }
        emit("\n  {\n");
        for ( int i=0 ; i<bindings.length ; i++ ) {
            IASTbinding binding = bindings[i];
            IASTvariable tmp = tmps[i];
            IASTvariable variable = binding.getVariable();
            emit("    ILP_Object ");
            emit(variable.getMangledName());
            emit(" = ");
            emit(tmp.getMangledName());
            semit(";\n");
        }
        iast.getBody().accept(this, context);
        emit("\n  }\n}\n");
        return null;
    }
```

Exemple

(if true print ("invisible") ; 48)

```
#include <stdio.h>
#include <stdlib.h>
#include "ilp.h"

ILP_Object ilp_program()
{{ILP_Object  ilptmp121;
  {ILP_Object ilptmp122;
    ilptmp122 = ILP_TRUE;
    if (ILP_isEquivalentToTrue(ilptmp122)) {
      {ILP_Object ilptmp123;
      ilptmp123 = ILP_String2ILP("invisible");
      ilptmp121 = ILP_print(ilptmp123); } }
    else {ilptmp121 = ILP_FALSE;}}
  ilptmp121 = ILP_Integer2ILP(48);
return ilptmp121;}}

int  main(int argc, char *argv[])
{
  ILP_START_GC;
  ILP_print(ilp_program());
  ILP_newline();
  return EXIT_SUCCESS;
}
```

Test d'ILP : exemple du compiler

```
@RunWith(Parameterized.class)
public class CompilerTest {

protected File file;

public CompilerTest(final File file) {
  this.file = file;
}

public void configureRunner(CompilerRunner run) throws CompilationException {
  IASTfactory factory = new ASTfactory();
  run.setILPMLParser(new ILPMLParser(factory));
  IOperatorEnvironment ioe = new OperatorEnvironment();
  OperatorStuff.fillUnaryOperators(ioe);
  OperatorStuff.fillBinaryOperators(ioe);
  IGlobalVariableEnvironment gve = new GlobalVariableEnvironment();
  GlobalVariableStuff.fillGlobalVariables(gve);
  Compiler compiler = new Compiler(ioe, gve);
  compiler.setOptimizer(new IdentityOptimizer());
  run.setCompiler(compiler);
}

@Test
public void processFile() throws CompilationException, ParseException, IOException {
  CompilerRunner run = new CompilerRunner();
  configureRunner(run);
  run.checkPrintingAndResult(file, run.compileAndRun(file));
}
```

Test d'ILP : exemple du compiler (suite)

```java
public class CompilerRunner {

    public String compileAndRun(File file)
        throws ParseException, CompilationException, IOException {
        System.err.println("Testing " + file.getAbsolutePath() + " ...");
        assertTrue(file.exists());
        // lancement du parsing
        IASTprogram program = parser.parse(file);
        // lancement de la compilation vers C
        String compiled = compiler.compile(program);
        File cFile = FileTool.changeSuffix(file, "c");
        FileTool.stuffFile(cFile, compiled);
        // lancement du script de compilation et d'execution
        // runtimeScript = "C/compileThenRun.sh +gc"
        String compileProgram = "bash " + runtimeScript + " " + cFile.getAbsolutePath(
        ProgramCaller pc = new ProgramCaller(compileProgram);
        pc.setVerbose();
        pc.run();
        assertEquals("Comparing return code", 0, pc.getExitValue());
        return pc.getStdout().trim();
    }
```

Récapitulation

- statique/dynamique
- choix de représentation (à l'exécution) des valeurs
- bibliothèque d'exécution
- schema de compilation
- destination