Master Informatique 2015-2016
Spécialité STL
Développement des langages de programmation
DLP – 4I501

Carlos Agon
agonc@ircam.fr

Plan du cours 4

- Génération de code
- Récapitulation
- Techniques Java

Principes de compilation

- Les variables ILP sont compilées en variables C
- Les expressions ILP sont compilées en expressions/instructions C

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;
```

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 classes dans 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 {
...
}
```

Mais aussi de redéfinitions

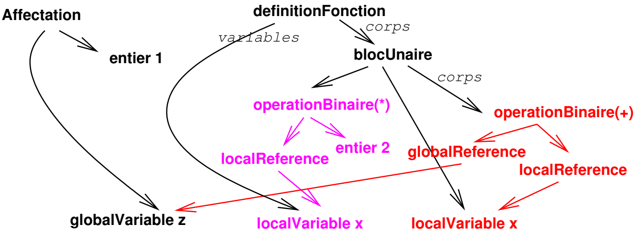
```
public class ASTCprogram extends ASTprogram
implements IASTCprogram {

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 ?

Normalisation

On fait une analyse statique : la 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).

Le visiteur normalizer

```java
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;
  }
 }
```

```
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);
  }
```

Environnement global

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

Environnement global pour les primitives

```
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);
}

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

```
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));

 env.addGlobalFunctionValue(
   new Primitive("throw", "ILP_throw", 1));
    }
}
```

Environnement global pour les opérateurs

```
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;
}

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 (IOperatorEnvironmen
            throws CompilationException {
        env.addUnaryOperator("-", "ILP_Opposite");
        env.addUnaryOperator("!", "ILP_Not");
    }

    public static void fillBinaryOperators (IOperatorEnvironme
            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);
        visit(newprogram, context);
        out.flush();
       ...
        return sw.toString();
    }
```

Context

```java
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 de l'alternative

alternative = (condition, consequence, alternant)

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

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

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

```
<sequence>
<string>Un, </string>
<string>Deux, </string>
<string>Trois, </string>
</sequence>
</program>
```

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

Compilation du bloc unaire II

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

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

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

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;
}
```

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

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

```
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, ...)

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

$$\begin{array}{l}
\texttt{d fonctionCorrespondante(} \\
\qquad\qquad \overrightarrow{argument1}\ \texttt{,} \\
\qquad\qquad \overrightarrow{argument2}\ \texttt{,} \\
\qquad\qquad \texttt{...)}
\end{array}$$

Compilation d'une opération

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

$$\overrightarrow{\mathit{opération}}^{\mathtt{d}}$$

$$\begin{array}{l} \mathtt{d\ fonctionCorrespondante(} \\ \qquad\qquad \overrightarrow{\mathit{opérandeGauche}}\ \mathtt{,} \\ \qquad\qquad \overrightarrow{\mathit{opérandeDroit}}\ \ \mathtt{)} \end{array}$$

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.getOpera
    emit(context.destination.compile());
    emit(cName);
    emit("(");
    emit(tmp1.getMangledName());
    emit(", ");
    emit(tmp2.getMangledName());
    emit(");\n");
    emit("} \n");
    return null;
}
```

Exemple

(begin (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;
}
```

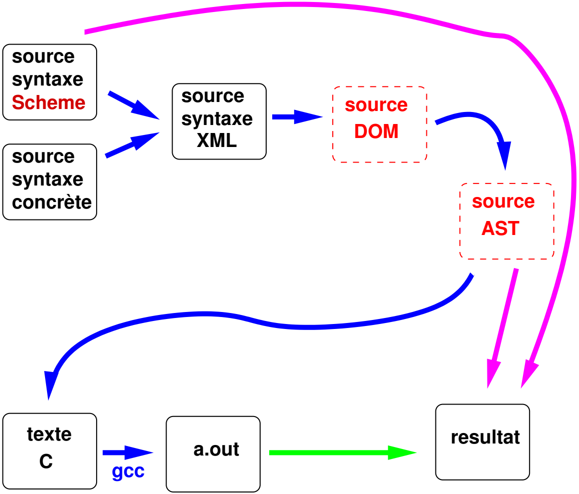
Test : processFile

Ressource: com.paracamplus.ilp1.compiler.test.CompilerTest

```
public void processFile () throws Throwable {

System.err.println("Testing " + file.getAbsolutePath() + " ...");
assertTrue(file.exists());
Input input = new InputFromFile(file);
parser.setInput(input);
File rngFile = new File(rngFileName);
parser.setGrammar(rngFile);
IASTprogram program = parser.getProgram();
IOperatorEnvironment ioe = new OperatorEnvironment();
OperatorStuff.fillUnaryOperators(ioe);
OperatorStuff.fillBinaryOperators(ioe);
IGlobalVariableEnvironment gve =
                                new GlobalVariableEnvironment();
GlobalVariableStuff.fillGlobalVariables(gve);
Compiler compiler = new Compiler(ioe, gve);
String compiled = compiler.compile(program);
File cFile = changeSuffix(file, "c");
```

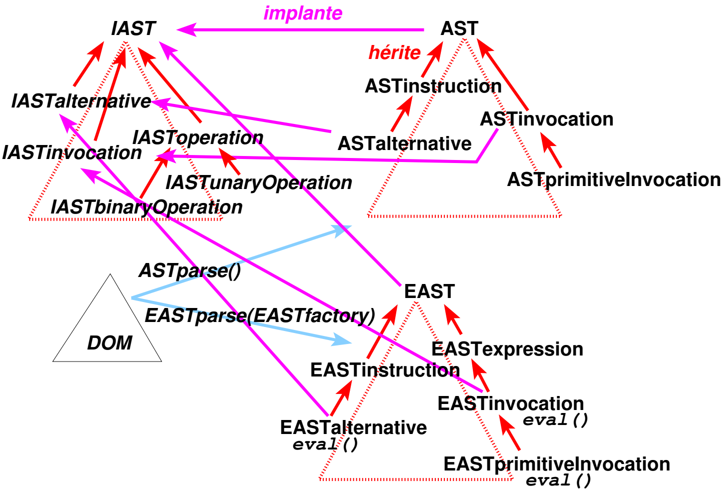
```
String compileProgram = "bash C/compileThenRun.sh +gc "
            + cFile.getAbsolutePath();
ProgramCaller pc = new ProgramCaller(compileProgram);
pc.setVerbose();
pc.run();
assertEquals("Comparing return code", 0, pc.getExitValue()
String executionPrinting = pc.getStdout().trim();
checkPrintingAndResult(executionPrinting);
}
```

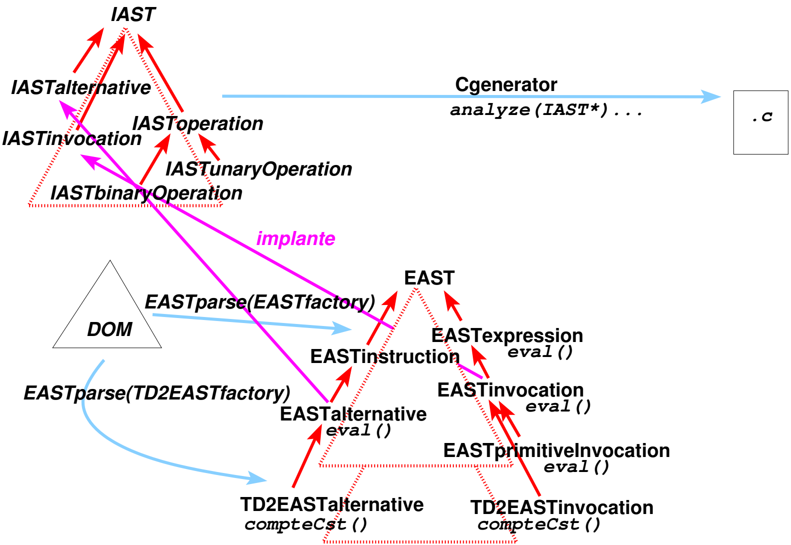
Grandes masses



Récapitulation

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

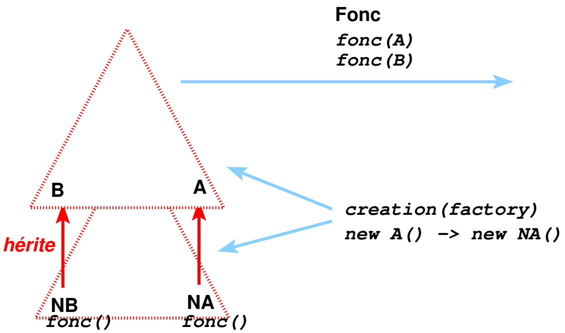

implante
IAST
AST
hérite
ASTinstruction
IASTalternative
ASTinvocation
IASToperation
ASTalternative
IASTinvocation
IASTunaryOperation
ASTprimitiveInvocation
IASTbinaryOperation
ASTparse()
EAST
EASTparse(EASTfactory)
DOM
EASTexpression
EASTinstruction
EASTinvocation
eval()
EASTalternative
eval()
EASTprimitiveInvocation
eval()

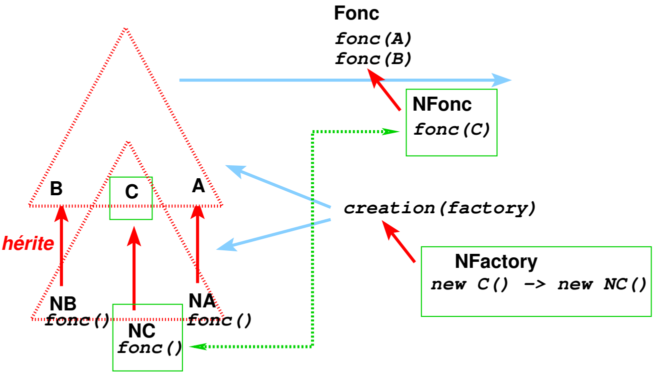

IAST
IASTalternative
IASToperation
IASTinvocation
IASTunaryOperation
IASTbinaryOperation
Cgenerator
analyze(IAST*)...
.c
implante
EAST
EASTparse(EASTfactory)
DOM
EASTexpression
eval()
EASTinstruction
EASTparse(TD2EASTfactory)
EASTinvocation
eval()
EASTalternative
eval()
EASTprimitiveInvocation
eval()
TD2EASTalternative
compteCst()
TD2EASTinvocation
compteCst()

Extensions

Deux sortes d'évolution :

- introduction de nouveaux noeuds d'AST (NA, NB)
- introduction de nouvelles fonctionnalités (*fonc*)




Fonc
fonc(A)
fonc(B)
NFonc
fonc(C)
B
C
A
hérite
creation(factory)
NFactory
new C() -> new NC()
NB
fonc()
NA
fonc()
NC
fonc()

Un compromis : le visiteurs

```
public interface IASTvisitor
<Result, Data, Anomaly extends Throwable> {
    Result visit(IASTalternative iast, Data data)
        throws Anomaly;
    Result visit(IASTbinaryOperation iast, Data data)
         throws Anomaly;
    Result visit(IASTblock iast, Data data)
        throws Anomaly;
...
    Result visit(IASTinvocation iast, Data data)
         throws Anomaly;
    Result visit(IASToperator iast, Data data)
         throws Anomaly;
}

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

Raffinement, spécialisation (*override*)

```
public class C {
   public void crunch (C c) {
      // utiliser this et c
   }
}
public class Sc extends C {
   @Overriding
   public void crunch (C c) {
      // utiliser this, c et super.crunch()
   }
}
```

Surcharge (*overload*)

Facilité d'écriture !

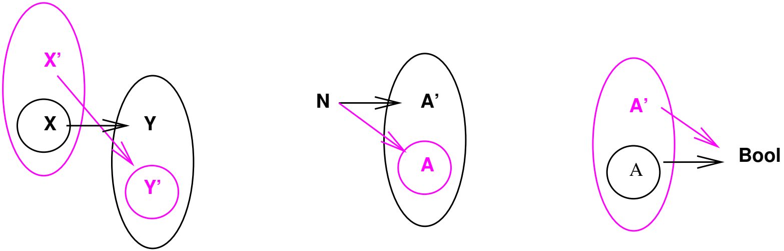
```
public class C {
   public Truc crunch(Integer i) {..}
   public Chose crunch(String d) {..}
   public Muche crunch(Object d) {..}
   void utilisation (Object o) {
      crunch(3); // boxing automatique
      if ( o instanceof Integer ) {
        crunch(o);
}
```

Peut se réécrire **statiquement** en :

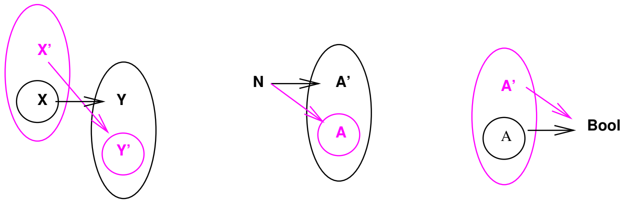
```
public class C {
   public Truc crunch_integer(Integer i) {..}
   public Chose crunch_double(String d) {..}
   public Muche crunch_object(Object d) {..}
   void utilisation (Object o) {
      crunch_integer(new Integer(3));
      if ( o instanceof Integer ) {
        crunch(o); // != crunch_integer(o)
                   // = crunch_object(o)
}
```

Contravariance

A est un sous-type de B si un $a \in A$ peut remplacer un $b \in B$ dans tous ses emplois possibles.
Une fonction $X' \to Y'$ est un sous-type de $X \to Y$ ssi $X \subset X'$ et $Y' \subset Y$.



NB : J'utilise l'inclusion ensembliste comme notation pour le sous-typage.
N est l'ensemble des entiers.



Cas des tableaux : si $A \subset A'$ alors $N \rightarrow A$ sous-type de $N \rightarrow A'$ donc `A[]` sous-type de `A’[]`.

Attention, en Java, le type d’un tableau est statique et ne dépend pas du type réel de ses éléments.

A est un sous-type de B si un $a \in A$ peut remplacer un $b \in B$ **dans tous ses emplois possibles**.

Emplois possibles : A, A.length, instanceof A[], A[i] alors

```
PointColore[] pcs = new PointColore[]{ new PointColore(
assert pcs[0] instanceof Point; // OK
ps[0] = new PointColore();      // OK
Point[] ps = (Point[]) pcs;     // OK
```

Mais pour un tableau, en fait, les emplois sont A, A.length, instanceof A[], A[i] et A[i] = v

```
Point[] ps = new Point[]{ new PointColore() };
assert ps[0] instanceof Point; // OK
ps[0] = new PointColore();      // OK

// PointColore[] pcs = (PointColore[]) ps;   // FAUX!
PointColore[] pcs = new PointColore[ps.length];
for ( int i=0 ; i<pcs.length ; i++ ) {
   pcs[i] = (PointColore) ps[i];
}
ps = (Point[]) pcs;             // OK
ps[0] = new Point(); // ArrayStoreException
```

Génériques

http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

```
public interface List<E, T extends Exception> {
  void add(E x);
  Iterator<E> iterator();
  E getOne() throws T;
}
public interface PointIterator<E extends Point>
extends Iterator<E> {
  boolean isFull();
}
// PointIterator<Color> < Iterator<Color>
// Iterator<Color> < Iterator<?>
```

Généricité

∀ E sous-type de Exception, alors la méthode frobnicate doit prendre une liste de E et rendre une liste de E :

```
public interface Igeneric<E extends Exception> {
    List<E> frobnicate (List<E> es) throws E;
    <T extends E> Collection<T> crunch(List<E> es);
}
```

∀ T sous-type de E, la méthode crunch doit prendre une liste de E et rendre une collection de T.

Généricité suite

```
public class Generic implements Igeneric<IOException> {
    public AbstractList<IOException>
        frobnicate (List<IOException> es)
    throws EOFException { ... }
    public <T extends IOException> Collection<T>
        crunch(List<IOException> es) { ... }
}
```

Sont erronées les méthodes :

```
public Collection<IOException> crunch(List<IOException> es) {
 // The method crunch(List<IOException>) of type Generic
 // has the same erasure as crunch(List<E>) of type
 // IGeneric<E> but does not override it. => The type
 // Generic must implement the inherited abstract method
 // IGeneric<IOException>.crunch(List<IOException>)
public <T extends IOException>
    Collection<IOException> crunch(List<IOException> es) {
 // The return type is incompatible with
 // IGeneric<IOException>.crunch(List<IOException>)
```

Généricité suite

La généricité en Java est implantée à l'exécution par effacement de types. La question `o instanceof List<Point>` n'a donc pas de sens. On peut cependant écrire `o instanceof List<?>`
Il n'est pas non plus possible d'écrire :

```
public void crunch(Set<Integer> si) { ...}
public void crunch(Set<String> ss) { ...}
```