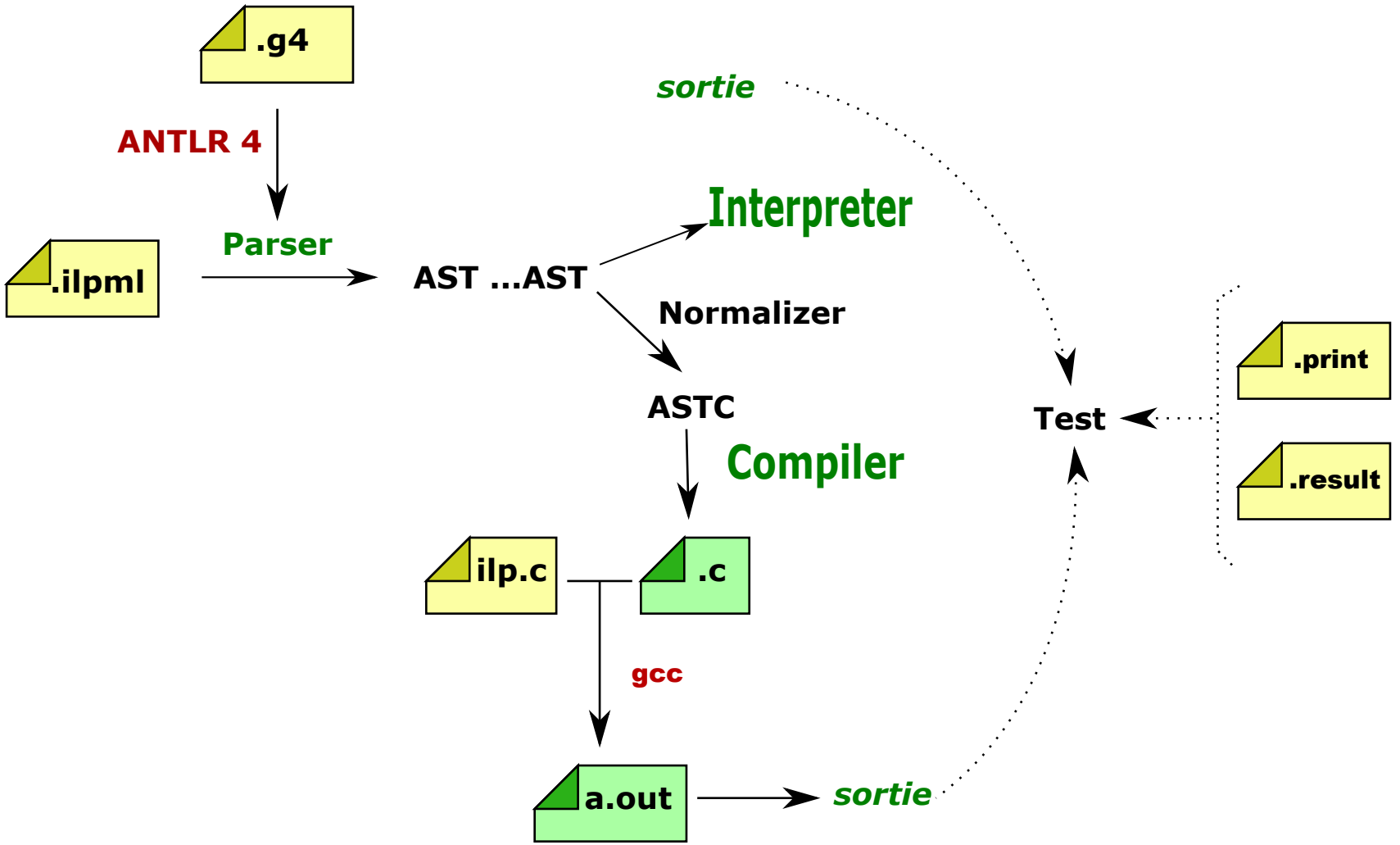
Sorbonne Université
Master Informatique 2020-2021
Spécialité STL
Développement des langages de programmation
DLP – 4I501
Cours 4

Carlos Agon
agonc@ircam.fr

Grand schéma



Plan du cours 4

- Analyse statique (Normalisation)
- Génération de code
- Récapitulation

Une transformation : la normalization (renommage des variables)

Si on se limite aux variables locales immuables d'ILP1 la détermination de portée semble facile, mais :

- ILP2 ajoutera les variables globales utilisateur, l'affectation et la déclaration de fonctions utilisateur
- ILP3 ajoutera les fonctions de première classe

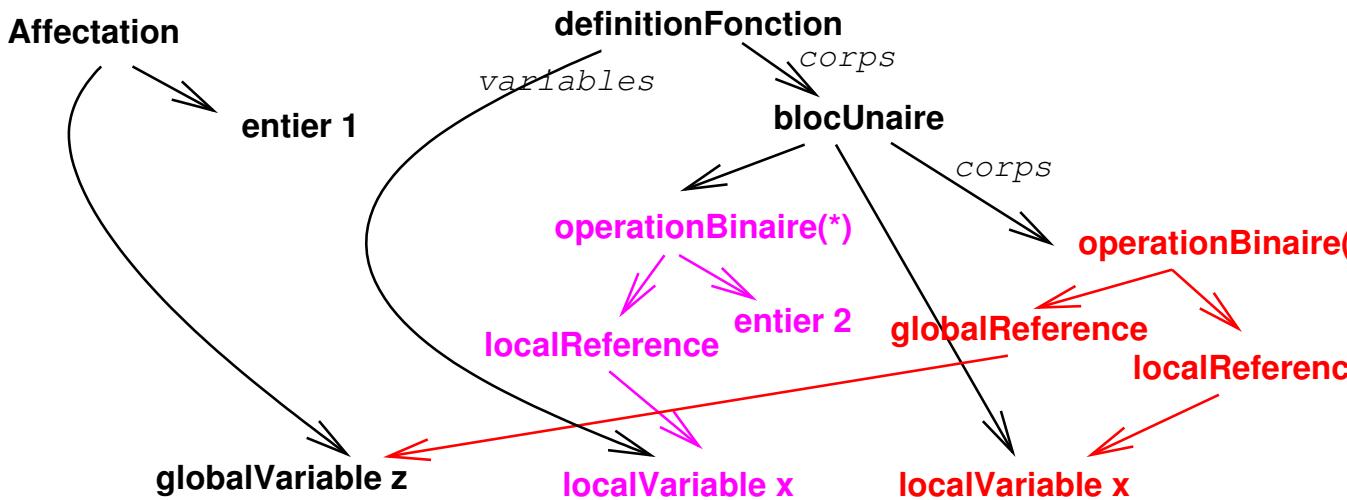
```
g = 12;
function f(z)
    (g = g + 1; g + z);
let x = g + 1 in f(x)
```

```
x = 1;
function f(y)
  let z = 2 in
  lambda (t) x + y + z  + t
```

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

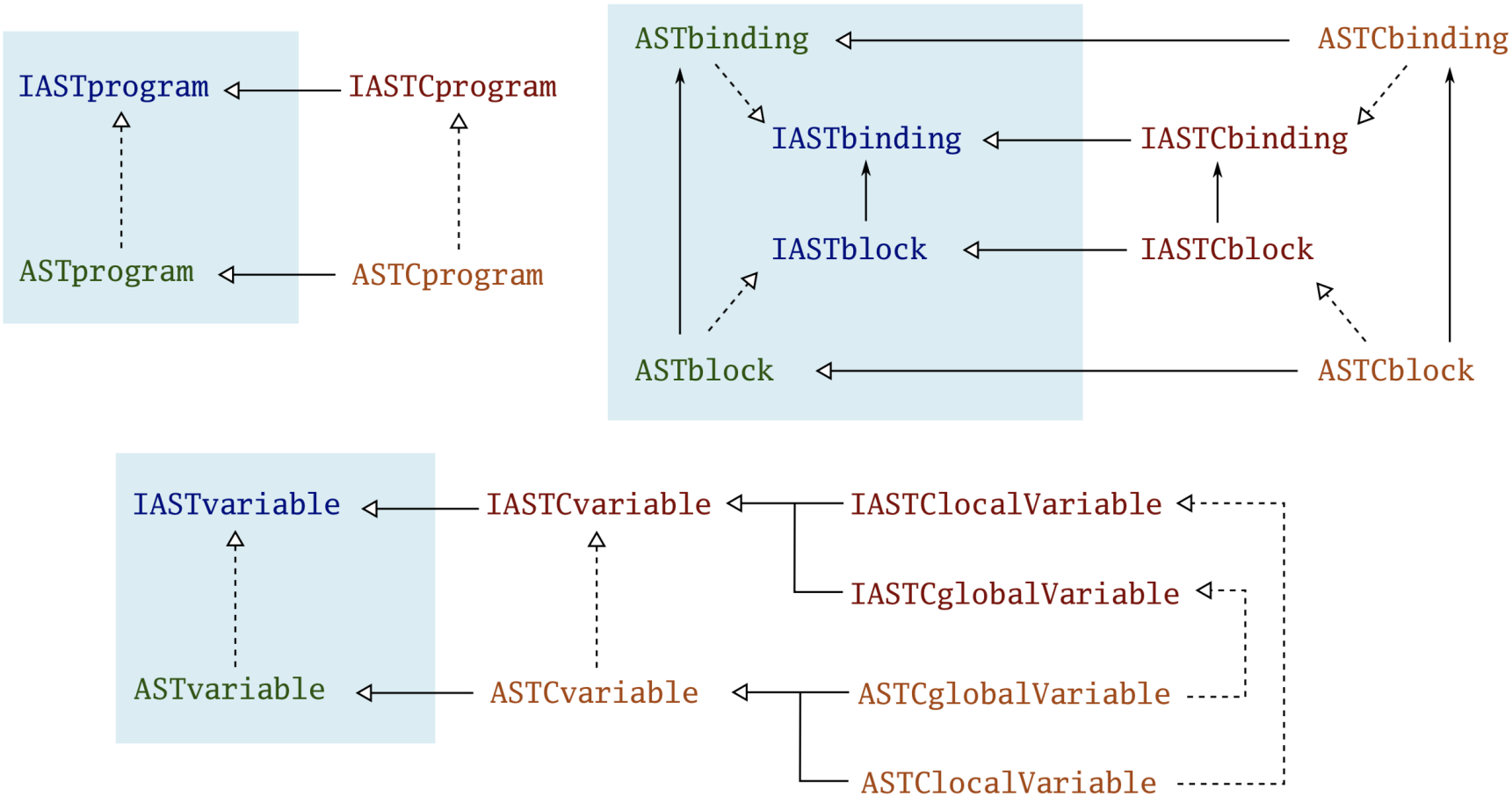
- Deux références à une même variable (locale ou globale) sont représentées par le même objet en mémoire.
- Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.
- Les séquences d'une seule expression sont normalisées à cette seule expression.

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);
    }
  }
```

Architecture des implantations : ASTC



Héritage de classes d'AST, et implantation d'interfaces IASTC héritant d'IAST.

seule l'implantation des méthodes ajoutées entre l'IAST et l'IASTC est nécessaire !

Le visiteur normalizer

```
1  public class Normalizer implements
2   IASTvisitor
3   <IASTexpression, INormalizationEnvironment, CompilationException> {
4
5      public Normalizer (INormalizationFactory factory) {
6          this.factory = factory;
7          this.globalVariables = new HashSet<>();
8      }
9      protected final INormalizationFactory factory;
10     protected final Set<IASTvariable> globalVariables;
11
12
13     public IASTCprogram transform(IASTprogram program)
14             throws CompilationException {
15     INormalizationEnvironment env = NormalizationEnvironment.EMPTY;
16
17     IASTexpression body = program.getBody();
18     IASTexpression newbody = body.accept(this, env);
19     return factory.newProgram(newbody);
20     }
```

```
1  public IASTexpression
2  visit(IASTboolean iast, INormalizationEnvironment env)
3    throws CompilationException {
4          return iast;
5  }
6
7  public IASTvariable
8  visit(IASTvariable iast, INormalizationEnvironment env)
9    throws CompilationException {
10   try {
11     return env.renaming(iast); // look for a local variable
12   } catch (NoSuchLocalVariableException exc) {
13   for ( IASTvariable gv : globalVariables ) {
14     if ( iast.getName().equals(gv.getName()) ) {
15       return gv;
16     }
17   }
18   IASTvariable gv = factory.newGlobalVariable(iast.getName());
19   globalVariables.add(gv);
20   return gv;
21   }
22  }
```

```
1  public IASTexpression
2   visit(IASTblock iast, INormalizationEnvironment env)
3        throws CompilationException {
4
5    INormalizationEnvironment newenv = env;
6    IASTbinding[] bindings = iast.getBindings();
7    IASTCblock.IASTCbinding[] newbindings =
8           new IASTCblock.IASTCbinding[bindings.length];
9    for ( int i=0 ; i<bindings.length ; i++ ) {
10       IASTbinding binding = bindings[i];
11       IASTexpression expr = binding.getInitialisation();
12       IASTexpression newexpr = expr.accept(this, env);
13       IASTvariable variable = binding.getVariable();
14       IASTvariable newvariable =
15              factory.newLocalVariable(variable.getName());
16       newenv = newenv.extend(variable, newvariable);
17       newbindings[i] =
18             factory.newBinding(newvariable, newexpr);
19    }
20    IASTexpression newbody =
21         iast.getBody().accept(this, newenv);
22    return factory.newBlock(newbindings, newbody);
23   }
```

Compilation

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

Ressource: com.paracamplus.ilp1.compiler.compiler

```
1  public class Compiler
2  implements
3  IASTCvisitor<Void, Compiler.Context, CompilationException> {
4
5  public Compiler (IOperatorEnvironment ioe,
6        IGlobalVariableEnvironment igve ) {
7          this.operatorEnvironment = ioe;
8          this.globalVariableEnvironment = igve;
9      }
10 protected final
11   IOperatorEnvironment operatorEnvironment;
12 protected final
13   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

```
1  public class Primitive implements IPrimitive {
2
3      public Primitive(String name, String cName, int arity) {
4          this.name = name;
5          this.cName = cName;
6          this.arity = arity;
7      }
8      private final String name;
9      private final String cName;
10     private final int arity;
11
12     public String getName() {
13         return name;
14     }
15
16     public String getCName() {
17         return cName;
18     }
19
20     public int getArity () {
21         return arity;
22     }
23 }
```

Initialisation de GlobalVariableEnvironment

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

```
1  public class GlobalVariableStuff {
2
3  public static void fillGlobalVariables
4      (IGlobalVariableEnvironment env) {
5   env.addGlobalVariableValue("pi", "ILP_PI");
6
7   env.addGlobalFunctionValue(
8      new Primitive("print", "ILP_print", 1));
9
10  env.addGlobalFunctionValue(
11    new Primitive("newline", "ILP_newline", 0));
12
13     }
14 }
```

Environnement global pour les opérateurs

```
1  public interface IOperatorEnvironment {
2      String getUnaryOperator (IASToperator operator)
3              throws CompilationException;
4      String getBinaryOperator (IASToperator operator)
5              throws CompilationException;
6      void addUnaryOperator (String operator, String cOperator)
7              throws CompilationException;
8      void addBinaryOperator (String operator, String cOperator)
9              throws CompilationException;
10 }
```

```
1  public class OperatorEnvironment implements IOperatorEnvironment {
2
3      public OperatorEnvironment () {
4          this.unaryOperatorEnvironment = new HashMap<>();
5          this.binaryOperatorEnvironment = new HashMap<>();
6      }
7      private final Map<String, String> unaryOperatorEnvironment;
8      private final Map<String, String> binaryOperatorEnvironment;
9
10     ...
11 }
```

Initialisation de OperatorEnvironment

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

```
1  public class OperatorStuff {
2
3      public static void fillUnaryOperators (IOperatorEnvironment env)
4              throws CompilationException {
5          env.addUnaryOperator("-", "ILP_Opposite");
6          env.addUnaryOperator("!", "ILP_Not");
7      }
8
9      public static void fillBinaryOperators (IOperatorEnvironment env)
10             throws CompilationException {
11         env.addBinaryOperator("+", "ILP_Plus");
12         env.addBinaryOperator("*", "ILP_Times");
13         env.addBinaryOperator("/", "ILP_Divide");
14         env.addBinaryOperator("-", "ILP_Minus");
15         ...
16     }
17 }
```

Compilation

```
1  public class Compiler
2  implements
3  IASTCvisitor<Void, Compiler.Context, CompilationException> {
4
5  public Compiler (IOperatorEnvironment ioe,
6                   IGlobalVariableEnvironment igve ) {
7          this.operatorEnvironment = ioe;
8          this.globalVariableEnvironment = igve;
9      }
10
11 protected Writer out;
12
13 public String compile(IASTprogram program)
14             throws CompilationException {
15
16         IASTCprogram newprogram = normalize(program);
17         ...
18         Context context = new Context(NoDestination.NO_DESTINATION);
19         StringWriter sw = new StringWriter();
20         out = new BufferedWriter(sw);
21         visit(newprogram, context);
22         out.flush();
23        ...
24         return sw.toString();
25     }
```

IASTCVisitor

```
1
2  import com.paracamplus.ilp1.interfaces.IASTvisitor;
3
4  public interface
5   IASTCvisitor<Result, Data, Anomaly extends Throwable>
6   extends IASTvisitor<Result, Data, Anomaly> {
7
8  Result visit(IASTCglobalVariable iast, Data data)
9    throws Anomaly;
10 Result visit(IASTClocalVariable iast, Data data)
11   throws Anomaly;
12 Result visit(IASTCprimitiveInvocation iast, Data data)
13   throws Anomaly;
14 Result visit(IASTCvariable iast, Data data)
15   throws Anomaly;
16 Result visit(IASTCcomputedInvocation iast, Data data)
17   throws Anomaly;
18
19 }
```

Nouvelles interfaces pour l'AST

```
1  public interface IASTCvisitable extends IASTvisitable {
2    <Result, Data, Anomaly extends Throwable>
3    Result accept(IASTCvisitor<Result, Data, Anomaly> visitor,
4    Data data) throws Anomaly;
5  }
6
7  public abstract interface IASTCvariable
8  extends IASTvariable, IASTCvisitable {
9    boolean isMutable();
10   void setMutable();
11 }
12
13 public interface IASTCglobalVariable extends IASTCvariable {
14 ...
15 }
16
17 public interface IASTClocalVariable extends IASTCvariable {
18 ...
19 }
```

Nouvelles implementations

```
1  public class ASTCprogram extends ASTprogram
2  implements IASTCprogram {
3
4  public ASTCprogram (IASTexpression expression) {
5          super(expression);
6          this.globalVariables = new HashSet<>();
7  }
8  protected Set<IASTCglobalVariable> globalVariables;
9
10 public Set<IASTCglobalVariable> getGlobalVariables() {
11         return globalVariables;
12 }
13
14 public void setGlobalVariables
15             (Set<IASTCglobalVariable> gvs) {
16         globalVariables = gvs;
17 }
18 }
```

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

Compilation

```
1  public class Compiler
2  implements
3  IASTCvisitor<Void, Compiler.Context, CompilationException> {
4
5  public Compiler (IOperatorEnvironment ioe,
6                   IGlobalVariableEnvironment igve ) {
7          this.operatorEnvironment = ioe;
8          this.globalVariableEnvironment = igve;
9      }
10
11 protected Writer out;
12
13 public String compile(IASTprogram program)
14             throws CompilationException {
15
16         IASTCprogram newprogram = normalize(program);
17         ...
18         Context context = new Context(NoDestination.NO_DESTINATION);
19         visit(newprogram, context);
20         out.flush();
21        ...
22         return sw.toString();
23     }
```

Context

```
1   public static class Context {
2       public Context (IDestination destination) {
3           this.destination = destination;
4       }
5       public IDestination destination;
6       public static AtomicInteger counter = new AtomicInteger(0);
7
8       public IASTvariable newTemporaryVariable () {
9           int i = counter.incrementAndGet();
10          return new ASTvariable("ilptmp" + i);
11      }
12
13      public Context redirect (IDestination d) {
14          if ( d == destination ) {
15              return this;
16          } else {
17              return new Context(d);
18          }
19      }
20  }
```

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 :

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

Exemples :

- $\overrightarrow{\textit{2}}$ → x = ILP_Integer2ILP(2)
- $\overrightarrow{\texttt{return}}$ $\textit{2}$ → return ILP_Integer2ILP(2)
- $\overrightarrow{\textit{2}}$ → ILP_Integer2ILP(2)

Destination

```
1 public class NoDestination implements IDestination {
2   public static final NoDestination NO_DESTINATION =
3             new NoDestination();
4   private NoDestination () {}
5   public String compile() {
6             return "";
```

```
1 public class AssignDestination implements IDestination {
2   public AssignDestination (IASTvariable variable) {
3     this.variable = variable;
4   private final IASTvariable variable;
5   public String compile() {
6     return variable.getMangledName() + " = ";
```

```
1 public class ReturnDestination implements IDestination {
2   private ReturnDestination () {}
3   public static final ReturnDestination RETURN_DESTINATION =
4               new  ReturnDestination();
5   public String compile() {
6           return "return ";
```

Génération de code

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

```
1   public Void visit(IASTCprogram iast, Context context) throws CompilationException {
2       emit(cProgramPrefix);
3       emit(cBodyPrefix);
4       Context cr = context.redirect(ReturnDestination.RETURN_DESTINATION);
5       iast.getBody().accept(this, cr);
6       emit(cBodySuffix);
7       emit(cProgramSuffix);
8       return null;
9   }
```

```
1   protected String cProgramPrefix = ""
2           + "#include <stdio.h> \n"
3           + "#include <stdlib.h> \n"
4           + "#include \"ilp.h\" \n\n";
5   protected String cBodyPrefix = "\n"
6           + "ILP_Object ilp_program () \n"
7           + "{ \n";
8   protected String cBodySuffix = "\n"
9           + "} \n";
10  protected String cProgramSuffix = "\n"
11         "int main (int argc, char *argv[]) \n"
12          + "{ \n"
13          + "  ILP_print(ilp_program()); \n"
14          + "  ILP_newline(); \n"
15          + "  return EXIT_SUCCESS; \n"
16          + "} \n";
```

Habillage du code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "ilp.h"
4
5  ILP_Object
6  ilp_program()
7  {
8  ...
9  }
10
11 int
12 main(int argc, char *argv[])
13 {
14   ILP_START_GC;
15   ILP_print(ilp_program());
16   ILP_newline();
17   return EXIT_SUCCESS;
18 }
```

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

Le code Java qui génère du code C est difficile à lire !
Introduction du Schema de compilation plus proche du code généré que du générateur de code (facile à déduire)

Note : en examen, pour éviter de perdre du temps, on ne demandera pas de donner le générateur de code en Java, mais uniquement le schéma de compilation, sous forme de texte ASCII.

Compilation d'une constante

$\longrightarrow$d
entier
d ILP_Integer2ILP(constanteEntière)

$\longrightarrow$d
float
d ILP_Float2ILP(constanteFlottante)

$\longrightarrow$d
boolean
d ILP_TRUE
d ILP_FALSE

$\longrightarrow$d
string
d ILP_String2ILP("constanteChaînePlusProtection")

Compilation d'un Integer

```
1 public Void visit(IASTinteger iast, Context context)
2             throws CompilationException {
3
4         emit(context.destination.compile());
5         emit("ILP_Integer2ILP(");
6         emit(iast.getValue().toString());
7         emit("); \n");
8         return null;
9     }
```

Compilation d'une variable

$$\xrightarrow{} \texttt{d} \\ \texttt{variable}$$

```
d variable ;
```

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

```
1 @Override public Void visit(IASTClocalVariable iast, Context co
2   emit(context.destination.compile());
3   emit(iast.getMangledName());
4   return null; }
5 @Override public Void visit(IASTCglobalVariable iast, Context c
6   emit(context.destination.compile());
7   emit(globalVariableEnvironment.getCName(iast));
8   emit("; \n");
9   return null; }
```

Schéma de Compilation de l'addition

Schéma de compilation

$$\frac{\overset{\longrightarrow d}{arg1 + arg2}}{\{\ \texttt{ILP\_Object}\ tmp1;\ \texttt{ILP\_Object}\ tmp2;\ \overset{\longrightarrow (tmp1=)}{arg1}\ \overset{\longrightarrow (tmp2=)}{arg2}\ d\ \ \texttt{ILP\_Plus}(tmp1, tmp2);\ \}}$$

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`).

primitive = (arg1, ..., argn)

```
                          ⟶d
                       invocation
{
ILP_Object tmp1;
...
ILP_Object tmpn;

              ⟶(tmp1 =)
                 arg1    ,
              ...
              ⟶(tmpn =)
                 argn    ,
d primitiveC( tmp1,...tmpn )
}
```

Compilation d'une opération

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

$$\overrightarrow{\texttt{opération}}\texttt{d}$$

$$\texttt{d fonctionCorrespondante(}\overrightarrow{\textit{opérandeGauche}}\texttt{,}\ \overrightarrow{\textit{opérandeDroit}}\texttt{)}$$

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

Compilation d'une opération

```
1  public Void visit(IASTbinaryOperation iast, Context context)
2    throws CompilationException {
3    IASTvariable tmp1 = context.newTemporaryVariable();
4    IASTvariable tmp2 = context.newTemporaryVariable();
5    emit("{ \n");
6    emit("  ILP_Object " + tmp1.getMangledName() + "; \n");
7    emit("  ILP_Object " + tmp2.getMangledName() + "; \n");
8    Context c1 = context.redirect(new AssignDestination(tmp1));
9    iast.getLeftOperand().accept(this, c1);
10   Context c2 = context.redirect(new AssignDestination(tmp2));
11   iast.getRightOperand().accept(this, c2);
12   String cName = operatorEnvironment.getBinaryOperator
13                                          (iast.getOperator());
14   emit(context.destination.compile());
15   emit(cName);
16   emit("(");
17   emit(tmp1.getMangledName());
18   emit(", ");
19   emit(tmp2.getMangledName());
20   emit(");\n");
21   emit("} \n");
22   return null;
23 }
```

Compilation de l'alternative

alternative = (condition, consequence, alternant)

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

```
if ( ILP_isEquivalentToTrue( condition ) ) {
   consequence ;
} else {
   alternant ;
}
```

Compilation de l'alternative

```
1  public void visit(IASTalternative iast, Context context)
2              throws CompilationException {
3
4  IASTvariable tmp1 = context.newTemporaryVariable();
5  emit("{ \n");
6  emit("  ILP_Object " + tmp1.getMangledName() + "; \n");
7  Context c = context.redirect(new AssignDestination(tmp1));
8  iast.getCondition().accept(this, c);
9  emit("  if ( ILP_isEquivalentToTrue(");
10 emit(tmp1.getMangledName());
11 emit(" ) ) {\n");
12 iast.getConsequence().accept(this, context);
13 if ( iast.isTernary() ) {
14     emit("\n  } else {\n");
15     iast.getAlternant().accept(this, context);
16  }
17 emit("\n  }\n}\n");
18 return null;
19 }
```

Compilation de la séquence

sequence = (exp1, ... expn)

```
                          ⟶d
                    séquence

{ ILP_Object temp;
   ⟶(temp =)
      exp1      ;
    ...
   ⟶(temp =)
      expn      ;
  d temp ;
}
```

Compilation de la séquence

```
1  public Void visit(IASTsequence iast, Context context)
2  throws CompilationException {
3
4  IASTvariable tmp = context.newTemporaryVariable();
5  IASTexpression[] expressions = iast.getExpressions();
6  Context c = context.redirect(new AssignDestination(tmp));
7  emit("{ \n");
8  emit("  ILP_Object " + tmp.getMangledName() + "; \n");
9  for ( IASTexpression expr : expressions ) {
10   expr.accept(this, c);
11 }
12 emit(context.destination.compile());
13 emit(tmp.getMangledName());
14 emit("; \n} \n");
15 return null;
16 }
```

Compilation de la séquence

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

```
1       {
2       ILP_Object  ilptmp117;
3       ilptmp117 = ILP_String2ILP("Un,");
4       ilptmp117 = ILP_String2ILP("Deux ");
5       ilptmp117 = ILP_String2ILP("Trois,");
6       return ilptmp117;
7       }
```

Compilation du bloc unaire I

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

$\overrightarrow{\texttt{d}}$
`bloc`

```
{
   ILP_Object variable = initialisation→ ;
   corps→d ;
}
```

Compilation du bloc unaire II

$\overrightarrow{\text{d}}$
bloc

```
{
   ILP_Object temporaire = initialisation ;
   ILP_Object variable = temporaire;
   corps ;
}
```

Compilation du bloc unaire II

```
1  public void visit(IASTblock iast, Context context) throws CompilationException {
2      emit("{ \n");
3      IASTbinding[] bindings = iast.getBindings();
4      IASTvariable[] tmps = new IASTvariable[bindings.length];
5      for ( int i=0 ; i<bindings.length ; i++ ) {
6          IASTvariable tmp = context.newTemporaryVariable();
7          emit("  ILP_Object " + tmp.getMangledName() + "; \n");
8          tmps[i] = tmp;
9      }
10     for ( int i=0 ; i<bindings.length ; i++ ) {
11         IASTbinding binding = bindings[i];
12         IASTvariable tmp = tmps[i];
13         Context c = context.redirect(new AssignDestination(tmp));
14         binding.getInitialisation().accept(this, c);
15     }
16     emit("\n  {\n");
17     for ( int i=0 ; i<bindings.length ; i++ ) {
18         IASTbinding binding = bindings[i];
19         IASTvariable tmp = tmps[i];
20         IASTvariable variable = binding.getVariable();
21         emit("    ILP_Object ");
22         emit(variable.getMangledName());
23         emit(" = ");
24         emit(tmp.getMangledName());
25         semit(";\n");
26     }
27     iast.getBody().accept(this, context);
28     emit("\n  }\n}\n");
29     return null;
30 }
```

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;
}
```

Compilation

La compilation est un processus complexe qui nécessite :

- plusieurs étapes successives :
  - lecture et analyse syntaxique du fichier source
  - analyse et transformation de l'AST
  - génération du code C
  - appel au compilateur et éditeur de lien C
  - appel du binaire final généré
- une configuration préalable :
  - quelles sont les primitives ?
  - quel analyseur syntaxique utiliser ?
  - quelles sont les passes d'optimisation ? (optionnelles !)
- Pour nous aider ILP possède deux classes :
  - CompilerRunner : regroupe la configuration et la compilation
  - CompilerTest : exemple de mise en œuvre du compilateur et intégration avec JUnit4

Test d'ILP : exemple du compiler

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

protected static String scriptCommand = "C/compileThenRun.sh +gc";

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);
  // configuration du script de compilation et execution
  run.setRuntimeScript(scriptCommand);
}

@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