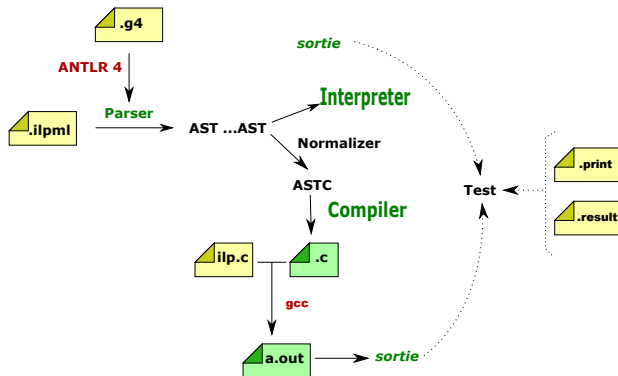


Master d'informatique 2020-2021
Spécialité STL
« Implantation de langages »
DLP – 4I501
épisode ILP2

Grand schéma














Buts

- ILP2 = ILP1 +
 - fonctions globales
 - boucle
 - affectation
 - variables globales
- Analyse statique

Plan du cours 5

- Présentation d'ILP2
- Syntaxe
- Sémantique (par l'interprétation)
- Génération de C (compilation)
- Préparation du partiel

Nouveaux packages

- ▶  com.paracamplus.ilp2.ast
- ▶  com.paracamplus.ilp2.compiler
- ▶  com.paracamplus.ilp2.compiler.ast
- ▶  com.paracamplus.ilp2.compiler.interfaces
- ▶  com.paracamplus.ilp2.compiler.normalizer
- ▶  com.paracamplus.ilp2.compiler.test
- ▶  com.paracamplus.ilp2.interfaces
- ▶  com.paracamplus.ilp2.interpreter
- ▶  com.paracamplus.ilp2.interpreter.test
- ▶  com.paracamplus.ilp2.parser
- ▶  com.paracamplus.ilp2.test

Adjonctions

ILP2 = ILP1 + définition de fonctions globales + boucle while + affectation + variables globales.

```
function deuxfois(x)
```

```
    2 * x;
```

```
function fact( n)
```

```
    if n = 1 then 1 else n * fact (n-1);
```

```
let x = 1 and y = "foo" in
```

```
    while (x < 100) do
```

```
        (
```

```
            x := deuxfois (fact(x));
```

```
            y := deuxfois (y);
```

```
        )
```

```
    y;
```

Mais encore

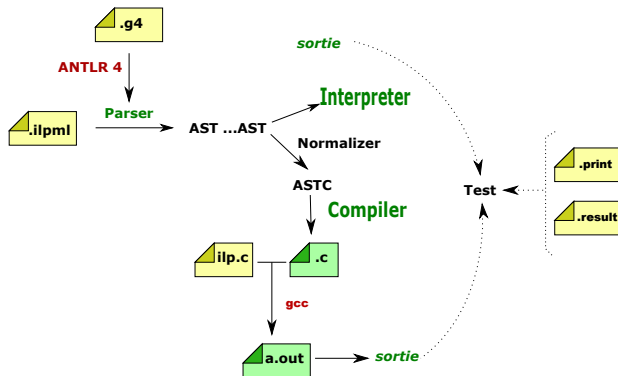
```
function deuxfois(x)
  2 * x;
```

```
function apply(f, x)
  f(x);
```

```
function second (one, two)
  two;
```

```
apply(deuxfois, 3000) - 7;
let y = 11 in
  deuxfois(second ((y = y + 1), y));
let f = deuxfois in
  g = f;
g(3000) - 5;
```

Grand schéma



Grammaire

```
grammar ILPMLgrammar2;

// Import de la grammaire a enrichir
import ILPMLgrammar1;

// Redefinition des programmes
prog returns [com.paracampus.ilp2.interfaces.IASTprogram node]
    : (defs+=globalFunDef ';'?) * (exprs+=expr ';'?) * EOF
    ;

// Fonction globale
globalFunDef returns [com.paracampus.ilp2.interfaces.IASTfunctionDefinition node]
    : 'function' name=IDENT '(' vars+=IDENT? (',' vars+=IDENT)* ')'
      body=expr
    ;

// Expressions enrichies
expr returns [com.paracampus.ilp1.interfaces.IASTexpression node]
// expressions de la grammaire precedente
    | fun=expr '(' args+=expr? (',' args+=expr)* ')' # Invocation
    ...

// ajouts

// affectation de variable
    | var=IDENT '=' val=expr # VariableAssign

// boucle
    | 'while' condition=expr 'do' body=expr # Loop
    ;
```

Analyseur

Une nouvelle classe *Parser* qui hérite de la classe *Parser* d'ILP1.

```
1 public class ILPMLParser
2 extends com.paracampus.ilp1.parser.ilpml.ILPMLParser {
3
4     public ILPMLParser(IASTfactory factory) {
5         super(factory);
6     }
7
8     @Override
9     public IASTprogram getProgram() throws ParseException {
10    try {
11        ANTLRInputStream in = new ANTLRInputStream(input.getText());
12        // flux de caract\ères -> analyseur lexical
13        ILPMLgrammar2Lexer lexer = new ILPMLgrammar2Lexer(in);
14        // analyseur lexical -> flux de tokens
15        CommonTokenStream tokens = new CommonTokenStream(lexer);
16        // flux tokens -> analyseur syntaxique
17        ILPMLgrammar2Parser parser = new ILPMLgrammar2Parser(tokens);
18        // d\émarage de l'analyse syntaxique
19        ILPMLgrammar2Parser.ProgContext tree = parser.prog();
20        // parcours de l'arbre syntaxique et appels du Listener
21        ParseTreeWalker walker = new ParseTreeWalker();
22        ILPMLListener extractor = new ILPMLListener((IASTfactory)factory);
23        walker.walk(extractor, tree);
24        return tree.node;
25    } catch (Exception e) {
26        throw new ParseException(e);
27    }
28 }
```

Analyseur

Une nouvelle classe *ILPMLListener* qui implemente *ILPMLgrammar2Listener*.

```
1 public class ILPMLListener implements ILPMLgrammar2Listener {
2     protected IASTfactory factory;
3     ...
4     @Override
5     public void exitProg(ProgContext ctx) {
6         List<IASTfunctionDefinition> f = new ArrayList<>();
7         for (GlobalFunDefContext d : ctx.defs) {
8             IASTdeclaration x = d.node;
9             f.add((IASTfunctionDefinition)x);
10        }
11        IASTexpression e = factory.newSequence(toExpressions(ctx.exprs));
12        ctx.node = factory.newProgram(
13            f.toArray(new IASTfunctionDefinition[0]),
14            e);
15    }
16
17    @Override
18    public void exitGlobalFunDef(GlobalFunDefContext ctx) {
19        ctx.node = factory.newFunctionDefinition(
20            factory.newVariable(ctx.name.getText()),
21            toVariables(ctx.vars, false),
22            ctx.body.node);
23    }
```

Analyseur

Une nouvelle classe *ILPMLListener* qui implemente *ILPMLgrammar2Listener*.

```
1 public class ILPMLListener implements ILPMLgrammar2Listener {  
2     protected IASTfactory factory;  
3     ...  
4  
5  
6     @Override  
7     public void exitVariableAssign(VariableAssignContext ctx) {  
8         ctx.node = factory.newAssignment(  
9             factory.newVariable(ctx.var.getText()),  
10            ctx.val.node);  
11    }  
12  
13    @Override  
14    public void exitLoop(LoopContext ctx) {  
15        ctx.node = factory.newLoop(ctx.condition.node, ctx.body.node);  
16    }
```

Une nouvelle fabrique

L'analyseur prend une fabrique à sa construction.

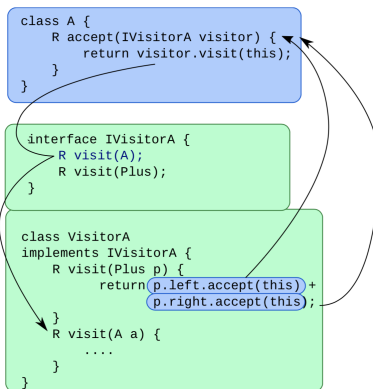
```
1 public class ASTfactory
2 extends com.paracamplus.ilp1.ast.ASTfactory implements IParserFactory{
3
4     public IASTprogram newProgram(IASTfunctionDefinition[] functions,
5                                     IASTexpression expression) {
6         return new ASTprogram(functions, expression);
7     }
8
9
10    public IASTassignment newAssignment(IASTvariable variable,
11                                        IASTexpression value) {
12        return new ASTassignment(variable, value);
13    }
14
15
16    public IASTloop newLoop(IASTexpression condition, IASTexpression body) {
17        return new ASTloop(condition, body);
18    }
19
20    public IASTfunctionDefinition newFunctionDefinition(
21        IASTvariable functionVariable,
22        IASTvariable[] variables,
23        IASTexpression body) {
24        return new ASTfunctionDefinition(functionVariable, variables, body);
25    }
26
27 }
```

Nouvelles classes AST

ASTassignment, ASTloop, ASTprogram, ASTfunctionDefinition.

```
1 import com.paracampus.ilp2.interfaces.IASTvisitor;  
2 import com.paracampus.ilp1.interfaces.IASTvisitable;  
3  
4 public class ASTassignment extends ASTexpression  
5 implements IASTassignment, IASTvisitable {  
6  
7     public ASTassignment (IASTvariable variable, IASTexpression expression) {  
8         this.variable = variable;  
9         this.expression = expression;  
10    }  
11    private final IASTvariable variable;  
12    private final IASTexpression expression;  
13  
14    public IASTvariable getVariable() {  
15        return variable;  
16    }  
17  
18    public IASTexpression getExpression() {  
19        return expression;  
20    }  
21  
22    public <Result, Data, Anomaly extends Throwable> Result accept(  
23        com.paracampus.ilp1.interfaces.IASTvisitor<Result, Data, Anomaly> visitor,  
24        Data data) throws Anomaly {  
25        return ((IASTvisitor <Result, Data, Anomaly>) visitor).visit(this, data);  
26    }  
}
```

Visiteurs : principe



Visiteurs : extension

```
class A {
    R accept(IVisitorA visitor) {
        return visitor.visit(this);
    }
}
```

```
interface IVisitorA {
    R visit(A);
    R visit(Plus);
}
```

```
class VisitorA
implements IVisitorA {
    R visit(Plus p) {
        return p.left.accept(this) +
               p.right.accept(this);
    }
    R visit(A a) {
        ....
    }
}
```

```
class B {
    R accept(IVisitorA visitor) {
        return ((IVisitorAB)visitor).visit(this);
    }
}
```

```
interface IVisitorAB
extends IVisitorA {
    R visit(B);
}
```

```
class VisitorAB
extends VisitorA
implements IVisitorAB {
    R visit(B b) {
        ....
    }
}
```



Nouvelle interface : IASTvisitor

```
1 package com.paracamplus.ilp2.interfaces;
2 public interface
3 IASTvisitor<Result, Data, Anomaly extends Throwable>
4 extends
5 com.paracamplus.ilp1.interfaces.IASTvisitor
6     <Result, Data, Anomaly>{
7
8 Result visit(IASTassignment iast, Data data)
9     throws Anomaly;
10 Result visit(IASTloop iast, Data data)
11     throws Anomaly;
12 }
```

```
1 import com.paracamplus.ilp2.interfaces.IASTvisitor;
2 import com.paracamplus.ilp1.interfaces.IASTvisitable;
3
4 public class ASTassignment extends ASTexpression
5 implements IASTassignment, IASTvisitable {
6
7 ...
8     public <Result, Data, Anomaly extends Throwable> Result accept(
9         com.paracamplus.ilp1.interfaces.IASTvisitor<Result, Data, Anomaly> visitor,
10         Data data) throws Anomaly {
11         return ((IASTvisitor <Result, Data, Anomaly>) visitor).visit(this, data);
12     }
```

Sémantique discursive (l'interprète)

Boucle comme en C (sans sortie prématurée)

Affectation comme en C (expression) sauf que (comme en JavaScript)
l'affectation sur une variable non locale crée la variable globale
correspondante

```
let n = 1 in
  while n < 100 do
    f = 2 * n;
print (f)
```

Fonctions globales en récursion mutuelle (comme en JavaScript, pas comme en C ou Pascal)

```
function pair (n) {  
    if ( n == 0 ) {  
        true  
    } else {  
        impair(n-1)  
    }  
}  
  
function impair (n) {  
    if ( n == 0 ) {  
        false  
    } else {  
        pair(n-1)  
    }  
}
```

Interprétation

```
1 public class Interpreter
2 extends com.paracampus.ilp1.interpreter.Interpreter
3 implements
4 IASTvisitor<Object, ILexicalEnvironment, EvaluationException> {
5
6 public Interpreter(IGlobalVariableEnvironment globalVariableEnvironment,
7     IOperatorEnvironment operatorEnvironment) {
8     super(globalVariableEnvironment, operatorEnvironment);
9 }
10
11 public Object visit(IASTprogram iast, ILexicalEnvironment lexenv)
12     throws EvaluationException {
13     for ( IASTfunctionDefinition fd : iast.getFunctionDefinitions() ) {
14         Object f = this.visit(fd, lexenv);
15         String v = fd.getName();
16         getGlobalVariableEnvironment().addGlobalVariableValue(v, f);
17     }
18     try {
19         return iast.getBody().accept(this, lexenv);
20     } catch (Exception exc) {
21         return exc;
22     }
23 }
```

Interpretation : définition de fonction

```
1 public Invocable visit(IASTfunctionDefinition iast, ILexicalEnvironment lex  
2     throws EvaluationException {  
3     Invocable fun = new Function(iast.getVariables(),  
4                                 iast.getBody(),  
5                                 new EmptyLexicalEnvironment());  
6     return fun;  
7 }  
8 }
```

Repose sur un nouvel objet de la bibliothèque d'exécution.

```
1 public class Function implements IFunction {
2
3 public Function (IASTvariable[] variables,
4                 IASTexpression body,
5                 ILexicalEnvironment lexenv) {
6     this.variables = variables;
7     this.body = body;
8     this.lexenv = lexenv;
9 }
10
11 public int getArity() {
12     return variables.length;
13 }
14 ...
15 public Object apply(Interpreter interpreter, Object[] argument)
16     throws EvaluationException {
17     if ( argument.length != getArity() ) {
18         String msg = "Wrong arity";
19         throw new EvaluationException(msg);
20     }
21
22     ILexicalEnvironment lexenv2 = getClosedEnvironment();
23     IASTvariable[] variables = getVariables();
24     for ( int i=0 ; i<argument.length ; i++ ) {
25         lexenv2 = lexenv2.extend(variables[i], argument[i]);
26     }
27     return getBody().accept(interpreter, lexenv2);
28 }
29 }
```

Interpretation d'une invocation

```
1 public Object visit(IASTInvocation iast, ILexicalEnvironment lexenv)
2     throws EvaluationException {
3     //attention iast.getFunction() return une expression
4     Object function = iast.getFunction().accept(this, lexenv);
5     if ( function instanceof Invocable ) {
6         Invocable f = (Invocable)function;
7         List<Object> args = new Vector<Object>();
8         for ( IASTExpression arg : iast.getArguments() ) {
9             Object value = arg.accept(this, lexenv);
10             args.add(value);
11         }
12         return f.apply(this, args.toArray());
13     } else {
14         String msg = "Cannot apply " + function;
15         throw new EvaluationException(msg);
16     }
17 }
18 }
```

Interpretation d'une boucle

```
1 public Object visit(IASTloop iast,  
2     ILexicalEnvironment lexenv)  
3     throws EvaluationException {  
4  
5     while ( true ) {  
6         Object condition=iast.getCondition().accept(this, lexenv);  
7         if ( condition instanceof Boolean ) {  
8             Boolean c = (Boolean) condition;  
9             if ( ! c ) {  
10                 break;  
11             }  
12         }  
13         iast.getBody().accept(this, lexenv);  
14     }  
15     return Boolean.FALSE;  
16 }
```


Interpretation d'une affectation

```
1 public Object visit(IASTassignment iast,
2     ILexicalEnvironment lexenv)
3     throws EvaluationException {
4
5     IASTvariable variable = iast.getVariable();
6     Object value = iast.getExpression().accept(this, lexenv);
7     try {
8         lexenv.update(variable, value);
9     } catch (EvaluationException exc) {
10         getGlobalVariableEnvironment()
11             .updateGlobalVariableValue(variable.getName(), value);
12     }
13     return value;
14 }
```

Les variables sont maintenant modifiables. Les interfaces des environnements d'interprétation doivent donc procurer cette nouvelle fonctionnalité. Si une variable n'existe pas elle est ajoutée aux variable globales.

Test d'interprétation

Ressource: `com.paracampus.ilp2.interpreter.test.InterpreterTest`

```
1 import com.paracampus.ilp2.ast.ASTfactory;
2 import com.paracampus.ilp2.interpreter.Interpreter;
3 import com.paracampus.ilp2.parser.ilpml.ILPMLParser;
4
5 @RunWith(Parameterized.class)
6 public class InterpreterTest extends com.paracampus.ilp1.interpreter.test.InterpreterTest {
7
8     protected static String[] samplesDirName = { "SamplesILP2", "SamplesILP1" };
9     public InterpreterTest(final File file) {
10         super(file);
11     }
12
13     public void configureRunner(InterpreterRunner run) throws EvaluationException {
14         // configuration du parseur
15         IASTfactory factory = new ASTfactory();
16         run.setILPMLParser(new ILPMLParser(factory));
17
18         // configuration de l'interpréteur
19         IGlobalVariableEnvironment gve = new GlobalVariableEnvironment();
20         GlobalVariableStuff.fillGlobalVariables(gve, stdout);
21         IOperatorEnvironment oe = new OperatorEnvironment();
22         OperatorStuff.fillUnaryOperators(oe);
23         OperatorStuff.fillBinaryOperators(oe);
24         Interpreter interpreter = new Interpreter(gve, oe);
25         run.setInterpreter(interpreter);
26     }
27
28     @Parameters(name = "{0}")
29     public static Collection<File[]> data() throws Exception {
30         return InterpreterRunner.getFileList(samplesDirName, pattern);
31     }
32 }
```

Compilation

```
1 public String compile(IASTprogram program)
2     throws CompilationException {
3
4     IASTCprogram newprogram = normalize(program);
5     newprogram = optimizer.transform(newprogram);
6
7     GlobalVariableCollector gvc = new GlobalVariableCollector();
8     Set<IASTCglobalVariable> gvs = gvc.analyze(newprogram);
9     newprogram.setGlobalVariables(gvs);
10    ...
11    try {
12        out = new BufferedWriter(sw);
13        visit(newprogram, context);
14        out.flush();
15    } catch (IOException exc) {
16        ...
17    }
```

AST normalisé (ASTC) : rappels et news

Le compilateur commence par transformer l'AST en ASTC.

Toujours les même deux buts, avec quelques nouveautés pour ILP2 :

Classification :

- distinction du type de portée :
 - variable locale : `ASTClocalVariable`
 - variable globale : `ASTCglobalVariable`
 - fonction globale : `IASTCglobalFunctionVariable`

il faut distinguer une fonction, d'une variable globale contenant une fonction

e.g., dans `let f = double`, `f` est une `IASTCglobalVariable`

`double` est une `IASTCglobalFunctionVariable`

- distinction du type d'appel :
 - direct, par nom : `ASTCglobalInvocation`
 - indirect, par variable : `ASTCcomputedInvocation`

Partage et identification :

- chaque `ASTCvariable` identifie de manière unique une variable
- toutes les utilisations de la même variable partagent le même nœud

Il nous faut une nouvelle classe ASTCprogram

```
1 public class ASTCprogram
2 extends
3 com.paracamplus.ilp1.compiler.ast.ASTCprogram
4 implements
5 com.paracamplus.ilp2.compiler.interfaces.IASTCprogram {
6
7 public ASTCprogram (IASTCfunctionDefinition[] functions,
8                     IASTexpression expression) {
9     super(expression);
10    this.functions = Arrays.asList(functions);
11 }
12
13 protected List<IASTfunctionDefinition> functions;
14
15 protected Set<IASTCglobalVariable> globalVariables;
16
17 }
```

Normalisation de l'AST pour l'invocation

Normalizer.java (ILP2)

```
@Override public IASTExpression visit(
    IASTInvocation iast,
    INormalizationEnvironment env) throws CompilationException {
    IASTExpression funexpr = iast.getFunction().accept(this, env);
    IASTExpression[] arguments = iast.getArguments();
    IASTExpression[] args = new IASTExpression[arguments.length];
    for ( int i=0 ; i<arguments.length ; i++ ) {
        IASTExpression argument = arguments[i];
        IASTExpression arg = argument.accept(this, env);
        args[i] = arg;
    }
    if ( funexpr instanceof IASTCglobalVariable ) {
        IASTCglobalVariable f = (IASTCglobalVariable) funexpr;
        return ((INormalizationFactory)factory).newGlobalInvocation(f, args);
    } else
        return ((INormalizationFactory)factory).newComputedInvocation(funexpr, args);
}
```

Transformation d'un appel `IASTInvocation`

- transformation de la référence à la fonction
- transformation des arguments
- création d'un nœud ASTC selon le type de référence de fonction

`newGlobalInvocation` ou `newComputedInvocation`

en réalité, c'est une optimisation ; `newComputedInvocation` pourrait être utilisé partout

Variables globales

```
let x = 1 in  
(  
  g = 59;  
  g;  
)
```

Variables globales

L'affectation sur une variable non locale réclame, en C, que l'on ait déclaré au préalable cette variable globale.

- ❶ il faut collecter les variables globales
- ❷ pour chacune d'entre elles, il faut l'allouer et l'initialiser.

Première analyse statique : collecte des variables globales. Réalisation : par arpentage de l'AST (un visiteur).

Variables globales (suite)

```
1 public String compile(IASTprogram program)
2     throws CompilationException {
3
4     IASTCprogram newprogram = normalize(program);
5     newprogram = optimizer.transform(newprogram);
6
7     GlobalVariableCollector gvc = new GlobalVariableCollector();
8     Set<IASTCglobalVariable> gvs = gvc.analyze(newprogram);
9     newprogram.setGlobalVariables(gvs);
10    ...
11    try {
12        out = new BufferedWriter(sw);
13        visit(newprogram, context);
14        out.flush();
15    } catch (IOException exc) {
16        ...
17    }
```

```
/* Global variables */
ILP_Object      g;

ILP_Object
ilp_program()
{
{
ILP_Object      ilptmp209;
ilptmp209 = ILP_Integer2ILP(1);

{
    ILP_Object      x1 = ilptmp209;
    {
        ILP_Object      ilptmp210;
        {
            ILP_Object      ilptmp211;
            ilptmp211 = ILP_Integer2ILP(59);
            ilptmp210 = (g = ilptmp211);
        }
        ilptmp210 = g;
        return ilptmp210;
    }

}

}

}
```

Le visitor *GlobalVariableCollector*

```
1 public class GlobalVariableCollector
2 implements IASTCvisitor<Set<IASTCglobalVariable>,
3                     Set<IASTCglobalVariable>,
4                     CompilationException> {
5
6     public GlobalVariableCollector () {
7         this.result = new HashSet<>();
8     }
9     protected Set<IASTCglobalVariable> result;
10
11     public Set<IASTCglobalVariable>
12         analyze(IASTprogram program)
13             throws CompilationException {
14         result = program.getBody().accept(this, result);
15         return result;
16     }
```

Le visitor *GlobalVariableCollector*

Grande partie du travail a été déjà fait par les visiteur *Normalize*

```
1  public Set<IASTCglobalVariable> visit(  
2      IASTCglobalVariable iast,  
3      Set<IASTCglobalVariable> result)  
4      throws CompilationException {  
5      result.add(iast);  
6      return result;  
7  }  
8  
9  public Set<IASTCglobalVariable> visit(  
10     IASTClocalVariable iast,  
11     Set<IASTCglobalVariable> result)  
12     throws CompilationException {  
13     return result;  
14 }  
15  
16  
17 public Set<IASTCglobalVariable> visit(  
18     IASTalternative iast,  
19     Set<IASTCglobalVariable> result)  
20     throws CompilationException {  
21     result = iast.getCondition().accept(this, result);  
22     result = iast.getConsequence().accept(this, result);  
23     result = iast.getAlternant().accept(this, result);  
24     return result;  
25 }
```

```
1 public Void visit(IASTCprogram iast, Context context)
2     throws CompilationException {
3     emit(cProgramPrefix);
4
5     emit(cGlobalVariablesPrefix);
6     for ( IASTCglobalVariable gv : iast.getGlobalVariables() ) {
7         emit("ILP_Object ");
8         emit(gv.getMangledName());
9         emit(";\n");
10    }
11
12    emit(cPrototypesPrefix);
13    Context c = context.redirect(NoDestination.NO_DESTINATION);
14    for ( IASTfunctionDefinition ifd : iast.getFunctionDefinitions() ) {
15        this.emitPrototype(ifd, c);
16    }
17
18    emit(cFunctionsPrefix);
19    for ( IASTfunctionDefinition ifd : iast.getFunctionDefinitions() ) {
20        this.visit(ifd, c);
21        emitClosure(ifd, c);
22    }
23
24    emit(cBodyPrefix);
25    Context cr = context.redirect(ReturnDestination.RETURN_DESTINATION);
26    iast.getBody().accept(this, cr);
27    return null;
28 }
```

Affectation : schéma de compilation

Là encore, on utilise les ressources de C.
affectation = (variable, valeur)

\xrightarrow{d}
affectation

d ($\xrightarrow{\text{variable}}$
valeur)

Affectation : génération de code

```
1 private Void visitNonLocalAssignment
2   (IASTAssignment iast, Context context)
3   throws CompilationException {
4   IASTvariable tmp1 = context.newTemporaryVariable();
5   emit("{ \n");
6   emit("  ILP_Object " + tmp1.getMangledName() + "; \n");
7   Context c1 = context.redirect(new AssignDestination(tmp1));
8   iast.getExpression().accept(this, c1);
9   emit(context.destination.compile());
10  emit("(");
11  emit(iast.getVariable().getMangledName());
12  emit(" = ");
13  emit(tmp1.getMangledName());
14  emit("); \n} \n");
15  return null;
16 }
```

Boucle : schéma de compilation

Il y a un équivalent en C que l'on emploie !

boucle = (condition, corps)

\xrightarrow{d}
boucle

$\xrightarrow{\quad}$
while (*ILP_isEquivalentToTrue*($\xrightarrow{\quad}$ *condition*)) {
 $\xrightarrow{\quad}$ *corps* ;
}

\xrightarrow{d}
nImporteQuoi ;

Compilation de la boucle

L'implantation :

```
1 public Void visit(IASTloop iast, Context context)
2     throws CompilationException {
3     emit("while ( 1 ) { \n");
4     IASTvariable tmp = context.newTemporaryVariable();
5     emit("    ILP_Object " + tmp.getMangledName() + "; \n");
6     Context c = context.redirect(new AssignDestination(tmp));
7     iast.getCondition().accept(this, c);
8     emit("    if ( ILP_isEquivalentToTrue(");
9     emit(tmp.getMangledName());
10    emit(") ) {\n");
11    Context cb = context.redirect(VoidDestination.VOID_DESTINATION);
12    iast.getBody().accept(this, cb);
13    emit("\n} else { \n");
14    emit("    break; \n");
15    emit("\n}\n\n");
16    whatever.accept(this, context);
17    return null;
18 }
```

Boucle : schéma de compilation 2

boucle = (condition, corps)

\xrightarrow{d}
while cond do body

{

while (1) {

ILP Object tmp;

$\xrightarrow{(tmp=)}$
cond

if (ILP isEquivalentToTrue(tmp)) {

$\xrightarrow{\quad}$
body

} else { break; }

}

\xrightarrow{d}
false

}

Boucle : exemple

```
let x1 = 50 in
(
  while (< x1 52) do
    x1 =x1 + 1;
  x1
)
```

```
ILP_Object      iltmp141;
ilptmp141 = ILP_Integer2ILP(50);
{
    ILP_Object      x1 = ilptmp141;
    {
        ILP_Object      ilptmp142;
        while (1) {
            ILP_Object      ilptmp143;
            {
                ILP_Object      ilptmp144;
                ILP_Object      ilptmp145;
                ilptmp144 = x1;
                ilptmp145 = ILP_Integer2ILP(52);
                ilptmp143 = ILP_LessThan(ilptmp144, ilptmp145);
            }
            if (ILP_isEquivalentToTrue(ilptmp143)) {
                {
                    ILP_Object      ilptmp146;
                    {
                        ILP_Object      ilptmp147;
                        ILP_Object      ilptmp148;
                        ilptmp147 = x1;
                        ilptmp148 = ILP_Integer2ILP(1);
                        ilptmp146 = ILP_Plus(ilptmp147, ilptmp148);
                    }
                    (void)(x1 = ilptmp146);
                }
            } else {
                break;
            }
        }
        ilptmp142 = ILP_FALSE;
        ilptmp142 = x1;
        return ilptmp142;
    }
}
```

Fonctions : schéma de compilation

fonctionGlobale = (nom, variables..., corps)

$\xrightarrow{\quad}$
fonctionGlobale

// Declaration

```
static ILP_Object nom (  
    ILP_Object variable, ... );
```

...

// Definition

```
ILP_Object nom (  
    ILP_Object variable,
```

...

```
) {
```

$\xrightarrow{\quad}$ **return**
corps

```
}
```

Compilation des fonctions

Pour le prototype

```
1 protected void emitPrototype(IASTCfunctionDefinition iast, Context context)
2     throws CompilationException {
3     emit("ILP_Object "); emit(iast.getCName()); emit("\n");
4     IASTvariable[] variables = iast.getVariables();
5     for ( int i=0 ; i< variables.length ; i++ ) {
6         IASTvariable variable = variables[i];
7         emit(",\n    ILP_Object ");
8         emit(variable.getMangledName());
9     }
10    emit(");\n");
11 }
```

Pour la définition

```
1 public Void visit(IASTCfunctionDefinition iast, Context context)
2     throws CompilationException {
3
4     // Idem que pour le prototype
5     emit(") {\n");
6     Context c = context.redirect(ReturnDestination.RETURN_DESTINATION);
7     iast.getBody().accept(this, c);
8     emit("}\n");
9     return null;
10 }
```

Compilateur : invocation directe et indirecte de fonctions

Important pour le partiel!!!!

appel direct (ILP)

```
function double(x) (2 * x);
double(27)
```

appel indirect (ILP)

```
function double(x) (2 * x);
let f = double in f(3) - 8
```

appel direct (C généré)

```
ILP_Object ilp__double
(ILP_Closure ilp_useless,
 ILP_Object x1)
{
  ILP_Object ilptmp2267;
  ilptmp2267 = ILP_Integer2ILP (2);
  return ILP_Times (ilptmp2267, x1);
}

ILP_Object ilp_program ()
{
  return ilp__double (
    NULL,
    ILP_Integer2ILP (27));
}
```

appel indirect (C généré)

```
struct ILP_Closure double_closure_object = {
  &ILP_object_Closure_class,
  {{ilp__double, 1, {NULL}}}}
};

ILP_Object ilp_program ()
{
  ILP_Object f2 = &double_closure_object;
  {
    ILP_Object ilptmp2412 =
      ILP_invoke (f2, 1, ILP_Integer2ILP(3));
    return ILP_Minus (ilptmp2412,
      ILP_Integer2ILP(8));
  }
}
```

Difficulté : appeler une fonction référencée par une variable

Schéma de compilation fonction (suite)

fonctionGlobale = (nom, variables..., corps)

$\xrightarrow{\quad}$
fonctionGlobale

// Declaration

```
static ILP_Object ilp__nom (
    ILP_Object variable, ... );
```

...

// Definition

```
ILP_Object ilp__nom (
    ILP_Object variable,
    ...
```

```
) {
     $\xrightarrow{\quad}$ return
    corps
}
```

// Closure

```
struct ILP_Closure nom_closure_object =
{ &ILP_object_Closure_class, {{ilp__nom, length(variables), {NULL}}} };
}
```


Schéma de compilation function (suite)

```
1 typedef struct ILP_Closure {  
2     struct ILP_Class* _class;  
3     union {  
4         struct asClosure_ {  
5             ILP_general_function function;  
6             short arity;  
7             struct ILP_Object* closed_variables[1];  
8         } asClosure;  
9     } _content;  
10 } *ILP_Closure;
```

Bibliothèque d'exécution : invocations indirectes

[Ressource: ilp.c](#)

```
ILP_Object ILP_invoke (ILP_Object closure, int argc, ...)
{
    va_list args;
    ILP_Object result;
    ILP_general_function f = ILP_find_invokee(closure, argc);
    va_start(args, argc);
    switch ( argc ) {
        case 0: {
            result = f(closure); break;
        }
        case 1: {
            ILP_Object arg1 = va_arg(args, ILP_Object);
            result = f(closure, arg1);
            break;
        }
        case 2: {
            ILP_Object arg1 = va_arg(args, ILP_Object);
            ILP_Object arg2 = va_arg(args, ILP_Object);
            result = f(closure, arg1, arg2);
            break;
        }
        ... }
    va_end(args);
    return result; }
```

Test de compilation

Ressource: `com.paracampus.ilp2.compiler.test.CompilerTest`

```
1 import com.paracampus.ilp2.ast.ASTfactory;
2 import com.paracampus.ilp2.compiler.Compiler;
3 import com.paracampus.ilp2.parser.ilpml.ILPMLParser;
4
5 @RunWith(Parameterized.class)
6 public class CompilerTest extends com.paracampus.ilp1.compiler.test.CompilerTest {
7
8     protected static String[] samplesDirName = { "SamplesILP2", "SamplesILP1" };
9
10    public CompilerTest(final File file) {
11        super(file);
12    }
13
14    @Override
15    public void configureRunner(CompilerRunner run) throws CompilationException {
16        // configuration du parseur
17        IASTfactory factory = new ASTfactory();
18        run.setILPMLParser(new ILPMLParser(factory));
19
20        // configuration du compilateur
21        IOperatorEnvironment ioe = new OperatorEnvironment();
22        OperatorStuff.fillUnaryOperators(ioe);
23        OperatorStuff.fillBinaryOperators(ioe);
24        ...
25        Compiler compiler = new Compiler(ioe, gve);
26        compiler.setOptimizer(new IdentityOptimizer());
27        run.setCompiler(compiler);
28    }
29
30    @Parameters(name = "{0}")
31    public static Collection<File[]> data() throws Exception {
32        return CompilerRunner.getFileList(samplesDirName, pattern);
33    }
34 }
```

Schéma de compilation : accès aux variables

Schéma de compilation : IASTCglobalFunctionVariable

\xrightarrow{d}
var

$\frac{}{d \text{ (ILP_Object) \&var_closure_object;}}$

IASTClocalVariable

\xrightarrow{d}
var

$\frac{}{d \text{ var.getMangledName();}}$

IASTCglobalVariable

\xrightarrow{d}
var

$\frac{}{d \text{ globalVariableEnvironment.getCName(var);}}$

Réorientation dans `visit(IASTvariable iast, Context context)` selon le type `T` de variable : `if (iast instanceof T) visit((T)iast, context)`

- IASTClocalVariable
- IASTCglobalVariable
- IASTCglobalFunctionVariable

Schéma de compilation : appel de fonction (1/2)

Schéma de compilation : cas IASTCglobalFunctionVariable

```

       $\xrightarrow{d}$ 
      var(arg1, ..., argN)
      -----
{
  ILP_Object tmp1;
  ...
  ILP_Object tmpN;
   $\xrightarrow{(tmp1=)}$ 
    arg1
  ...
   $\xrightarrow{(tmpN=)}$ 
    argN
  d ilp_var.getMangledName() (tmp1, ..., tmpN);
}
```

Le nom de la fonction appelée est **connu statiquement**.

On génère un appel direct à la fonction C correspondante.

Schéma de compilation : appel de fonction (2/2)

Schéma de compilation : cas général

```

       $\xrightarrow{d}$ 
       $\text{expr}(\text{arg1}, \dots, \text{argN})$ 


---


{
  ILP_Object tmpF;
  ILP_Object tmp1;
  ...
  ILP_Object tmpN;
   $\xrightarrow{(tmpF=)}$ 
   $\text{expr}$ 
   $\xrightarrow{(tmp1=)}$ 
   $\text{arg1}$ 
  ...
   $\xrightarrow{(tmpN=)}$ 
   $\text{argN}$ 
   $d \quad \text{ILP\_invoke}(\text{tmpF}, \text{tmp1}, \dots, \text{tmp2});$ 
}
```

Le nom de la fonction appelée n'est pas connu statiquement.

Il est nécessaire d'évaluer une expression à l'**exécution** pour trouver la fonction, puis de l'appeler par pointeur avec `ILP_invoke`