Master d'informatique 2016-2017
Spécialité STL
« Implantation de langages »
DLP – 4I501
épisode ILP2

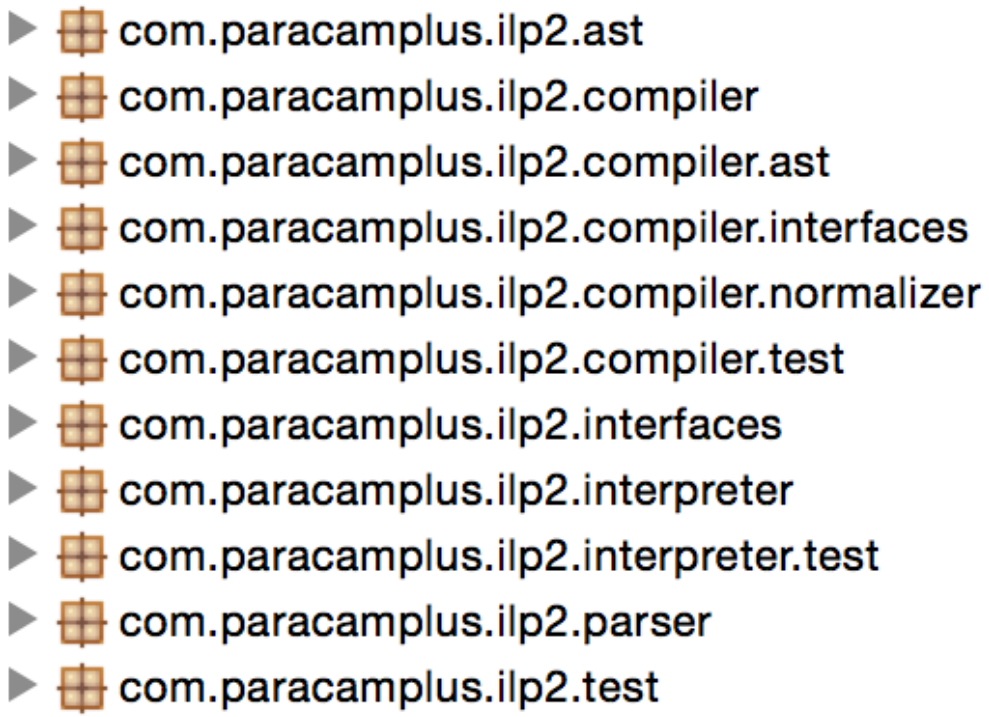
Buts

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

Plan du cours 5

- Présentation d'ILP2
- Syntaxe
- Sémantique (par l'interpretation)
- Génération de C (compilation)
- XML et RelaxNG

Nouveaux packages



Adjonctions

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

```
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

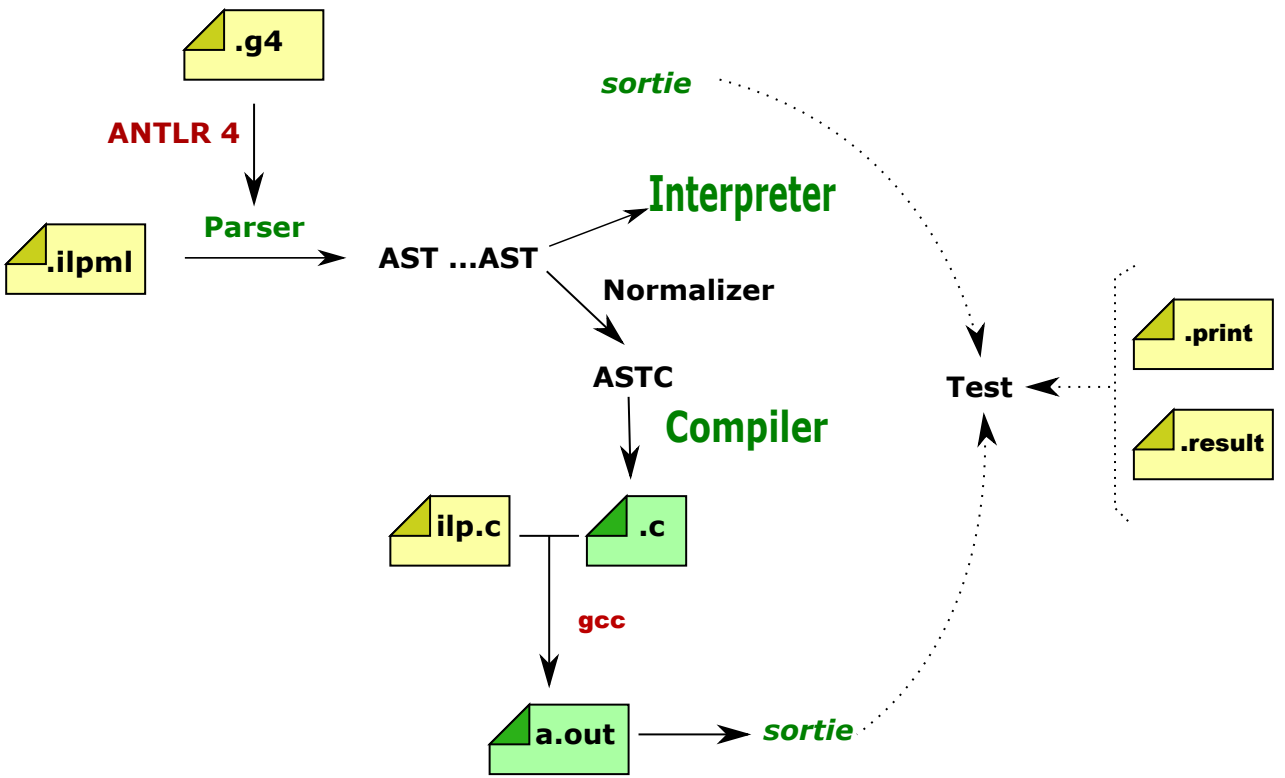
```
function deuxfois(x)
         2 * x;

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

// Expressions enrichies
expr returns [com.paracamplus.ilp1.interfaces.IASTexpression node]
// expressions de la grammaire precedente
        ...

// 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.paracamplus.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\‘eres -> 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\'emarage 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  public class ASTassignment extends ASTexpression
2  implements IASTassignment, IASTvisitable {
3
4      public ASTassignment (IASTvariable variable, IASTexpression expression) {
5          this.variable = variable;
6          this.expression = expression;
7      }
8      private final IASTvariable variable;
9      private final IASTexpression expression;
10
11     public IASTvariable getVariable() {
12         return variable;
13     }
14
15     public IASTexpression getExpression() {
16         return expression;
17     }
18
19     public <Result, Data, Anomaly extends Throwable>
20     Result accept(IASTvisitor<Result, Data, Anomaly> visitor, Data data)
21             throws Anomaly {
22         return visitor.visit(this, data);
23     }
```

Nouvelles interfaces : IASTvisitor et IASTvisitable

```
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  package com.paracamplus.ilp2.interfaces;
2  public interface IASTvisitable {
3    <Result, Data, Anomaly extends Throwable>
4
5  Result
6  accept(IASTvisitor<Result, Data, Anomaly> visitor,
7                     Data data) throws Anomaly;
8  }
```

Affectation : accept

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

The type ASTassignment must implement the inherited abstract method IASTvisitable.accept(IASTvisitor<Result,Data,Anomaly>, Data)

```
1 public <Result, Data, Anomaly extends Throwable>
2 Result
3 accept(com.paracamplus.ilp1.interfaces.IASTvisitor
4       <Result, Data, Anomaly> visitor,
5       Data data) throws Anomaly {
6 return ((IASTvisitor <Result, Data, Anomaly>) visitor).
7       visit(this, data);
8 }
```

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
  done;
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.paracamplus.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,
2              ILexicalEnvironment lexenv)
3               throws EvaluationException {
4
5  Invocable fun =
6  new Function(iast.getVariables(), iast.getBody(),
7                       new EmptyLexicalEnvironment());
8
9  getGlobalVariableEnvironment()
10            .addGlobalVariableValue(iast.getName(), fun);
11
12 return fun;
13 }
```

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
2   public Object visit(IASTinvocation iast, ILexicalEnvironment lexenv)
3           throws EvaluationException {
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    }
```

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.

Test d'interpretation

Ressource: com.paracamplus.ilp2.interpreter.test.InterpreterTest

```
1  import com.paracamplus.ilp2.ast.ASTfactory;
2  import com.paracamplus.ilp2.interpreter.Interpreter;
3  import com.paracamplus.ilp2.parser.ilpml.ILPMLParser;
4
5  @RunWith(Parameterized.class)
6  public class InterpreterTest extends com.paracamplus.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'interpr\'ete
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 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 final IOperatorEnvironment
12   operatorEnvironment;
13 protected final IGlobalVariableEnvironment
14   globalVariableEnvironment;
```

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.

1. il faut collecter les variables globales
2. 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     ...
```

```
/* 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;
        }

}
}

}
```

Collecte des variables globales

Toute variable non locale est globale.
Parcours récursif de la grammaire.

- GV(sequence(i1, i2, ...) = GV(i1) $\cup$ GV(i2) $\cup$ ...
- GV(alternative(c,it,if)) = GV(c) $\cup$ GV(it) $\cup$ GV(if)
- GV(boucle(c,s)) = GV(c) $\cup$ GV(s)
- GV(affectation(n,v)) = { n } $\cup$ GV(v)
- GV(constante) = $\emptyset$
- GV(variable) = { variable }
- GV(definitionFonction(n,(v1,v2,...),c) = GV(c) - { v1, v2, ... }
- GV(blocUnaire(v,e,c)) = GV(e) $\cup$ (GV(c) - { v })

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    }
```

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 }
```

```
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  }
```

Fonctions : compilation

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

$\longrightarrow$ *fonctionGlobale*

```
// Declaration
static ILP_Object nom (
        ILP_Object variable, ... );
...
// Definition
ILP_Object nom (
        ILP_Object variable,
        ...
) {
    →return
     corps
}
```

Compilation des fonctions

Pour le prototype

```
protected void emitPrototype(IASTCfunctionDefinition iast, Context context)
          throws CompilationException {
       emit("ILP_Object "); emit(iast.getCName());emit("(\n");
       IASTvariable[] variables = iast.getVariables();
       for ( int i=0 ; i< variables.length ; i++ ) {
           IASTvariable variable = variables[i];
           emit(",\n    ILP_Object ");
           emit(variable.getMangledName());
       }
       emit(");\n");
   }
```

Pour la définition

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

   // Idem que pour le prototype
 emit(") {\n");
       Context c = context.redirect(ReturnDestination.RETURN_DESTINATION);
       iast.getBody().accept(this, c);
       emit("}\n");
       return null;
   }
```

Affectation : compilation

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

$$\overset{\longrightarrow \texttt{d}}{affectation}$$

$$\texttt{d}\ (\ \overset{\longrightarrow \texttt{variable}}{valeur}\ )$$

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 : compilation

Il y a un équivalent en C que l'on emploie !
boucle = (condition, corps)

$\overrightarrow{boucle}^{d}$

```
while ( ILP_isEquivalentToTrue( condition ) ) {
   (void)corps ;
}
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 : exemple

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

```
ILP_Object      ilptmp141;
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;
        }

}
```

Test de compilation

Ressource: com.paracamplus.ilp2.compiler.test.CompilerTest

```
import com.paracamplus.ilp2.ast.ASTfactory;
import com.paracamplus.ilp2.compiler.Compiler;
import com.paracamplus.ilp2.parser.ilpml.ILPMLParser;

@RunWith(Parameterized.class)
public class CompilerTest extends com.paracamplus.ilp1.compiler.test.CompilerTest {

    protected static String[] samplesDirName = { "SamplesILP2", "SamplesILP1" };

    public CompilerTest(final File file) {
      super(file);
    }

    @Override
    public void configureRunner(CompilerRunner run) throws CompilationException {
      // configuration du parseur
        IASTfactory factory = new ASTfactory();
        run.setILPMLParser(new ILPMLParser(factory));

        // configuration du compilateur
        IOperatorEnvironment ioe = new OperatorEnvironment();
        OperatorStuff.fillUnaryOperators(ioe);
        OperatorStuff.fillBinaryOperators(ioe);
        ...
        Compiler compiler = new Compiler(ioe, gve);
        compiler.setOptimizer(new IdentityOptimizer());
        run.setCompiler(compiler);
    }

    @Parameters(name = "{0}")
    public static Collection<File[]> data() throws Exception {
      return CompilerRunner.getFileList(samplesDirName, pattern);
    }
}
```

Buts pédagogiques

- Exercice de lecture de code
- Usage élaboré de Java 8
- Non modification des codes précédemment donnés
- Réflexions sur la conception de langages

Définition d'un langage

Définitions :

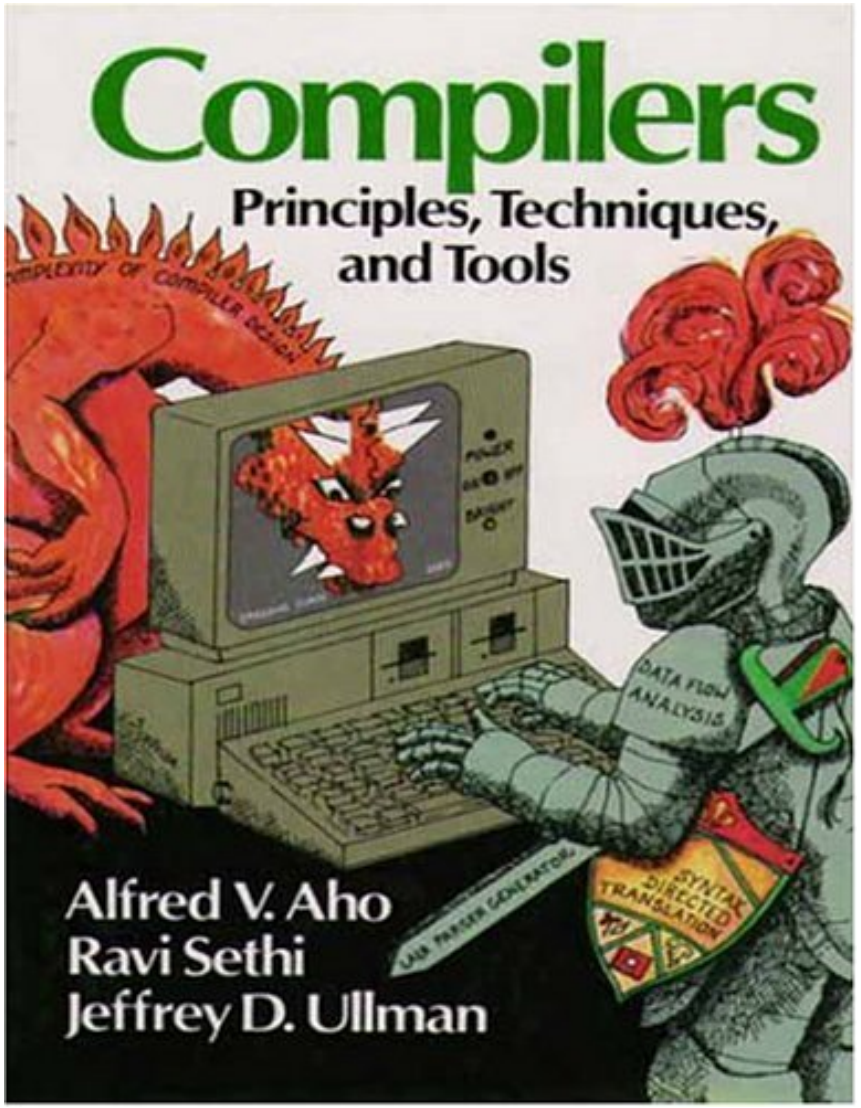
- Symbole : signe primitif (lettre, chiffre, point, ligne, ?)
- Alphabet : ensemble fini de symboles (Σ)
- Chaîne : séquence finie de symboles (la chaîne vide ϵ)
- Langages formel : ensemble de chaînes définies sur un alphabet Σ

Problèmes :

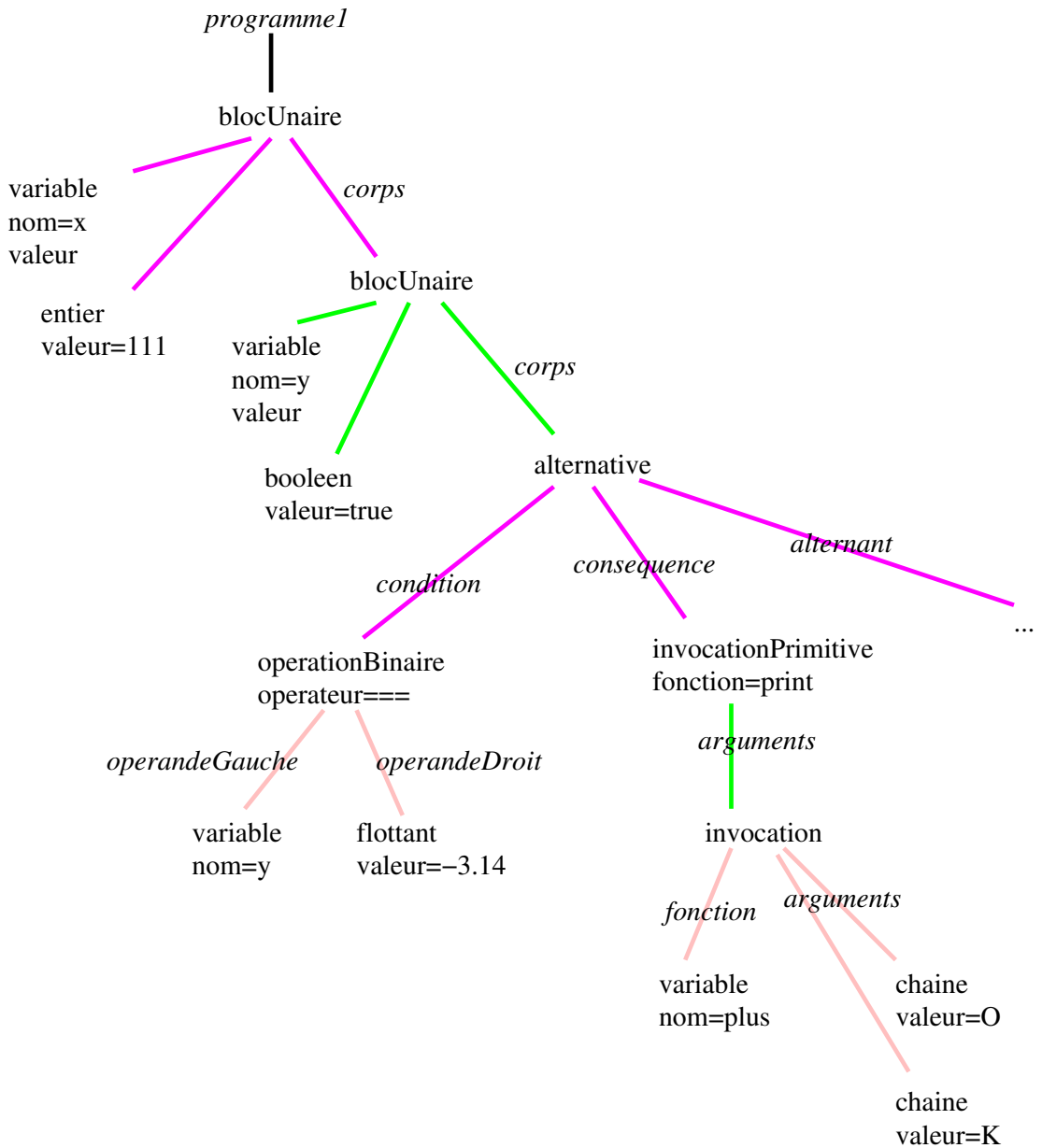
- Comment définir un langage en intension ?
- Comment calculer si une chaîne ω appartient ou non à un langage L ?

The dragon book

C'est important,mais on ne s'y intéressera pas trop !



Syntaxe arborescente



Syntaxe XML

```
<program>
<block>
<bindings>
<binding><variable name='x'/>
        <initialisation><float value='111'/></initialisation>
</binding>
<binding><variable name='y'/>
        <initialisation><boolean value='true'/></initialisation>
</binding>
</bindings>
<body>
<alternative>
<condition>
        <binaryOperation operator='=='>
        <leftOperand><variable name='y'</leftOperand>
        <rightOperand>
                <unaryOperation operator='-'>
                <operand> <float value='3.14'/></rightOperand> </operand>
                </unaryOperation>
        </binaryOperation>
</condition>
<consequence>
        <invocation>
        <function><variable name='print'/></function>
        <arguments>
        . . .
        </arguments>
        </invocation>
</consequence>
<alternant>
        . . .
</alternant>
```

Validation d'XML

- Un programme écrit en XML doit être *bien formé* c'est-à-dire respectueux des conventions d'XML.
- Un programme écrit en XML doit aussi être *valide* vis-à-vis d'une grammaire.

Les grammaires sont des DTD (pour *Document Type Definition*) ou maintenant des XML Schémas ou des schémas Relax NG.
Énorme intérêt pour la lecture en cas d'erreur.

RelaxNG

Relax NG est un formalisme pour spécifier des grammaires pour XML (bien plus lisible que les schémas XML (suffixe `.xsd` mais pour lesquels existe un mode dans Eclipse)).
Les grammaires Relax NG (prononcer *relaxing*) sont des documents XML (suffixe `.rng`) écrivables de façon compacte (suffixe `.rnc`) et surtout lisibles !
Une fois validé, les textes peuvent être réifiés en DOM (*Document Object Model*).

Ressource: Grammars/grammar1.rnc

Grammaire RelaxNG – ILP1

Les caractéristiques simples sont codées comme des attributs, les composants complexes (sous-arbres) sont codés comme des sous-éléments.

Ressource: Grammars/grammar1.rnc

```
start = program

program = element program {
    expression +
}

expression =
    alternative
  | sequence
  | block
  | constant
  | invocation
  | operation
  | variable
```

```
alternative = element alternative {
    element condition   { expression },
    element consequence { expression + },
    element alternant   { expression + } ?
}

sequence = element sequence {
    expression +
}

block = element block {
    element bindings {
        element binding {
            variable,
            element initialisation {
                expression
            }
        } *
    },
    element body { expression + }
}
```

```
invocation = element invocation {
    element function  { expression },
    element arguments { expression * }
}

variable = element variable {
    attribute name { xsd:Name - ( xsd:Name { pattern = "(
    empty
}

operation =
    unaryOperation
  | binaryOperation

unaryOperation = element unaryOperation {
    attribute operator { "-" | "!" },
    element operand    { expression }
}
```

```
binaryOperation = element binaryOperation {
    element leftOperand { expression },
    attribute operator {
        "+" | "-" | "*" | "/" | "%" |
        "|" | "&" | "^" |
        "<" | "<=" | "==" | ">=" | ">" | "<>" | "!="
    },
    element rightOperand { expression }
}

constant =
    element integer {
     attribute value { xsd:integer },
     empty }
  | element float {
     attribute value { xsd:float },
     empty }
  | element string { text }
  | element boolean  {
     attribute value { "true" | "false" },
     empty }
```

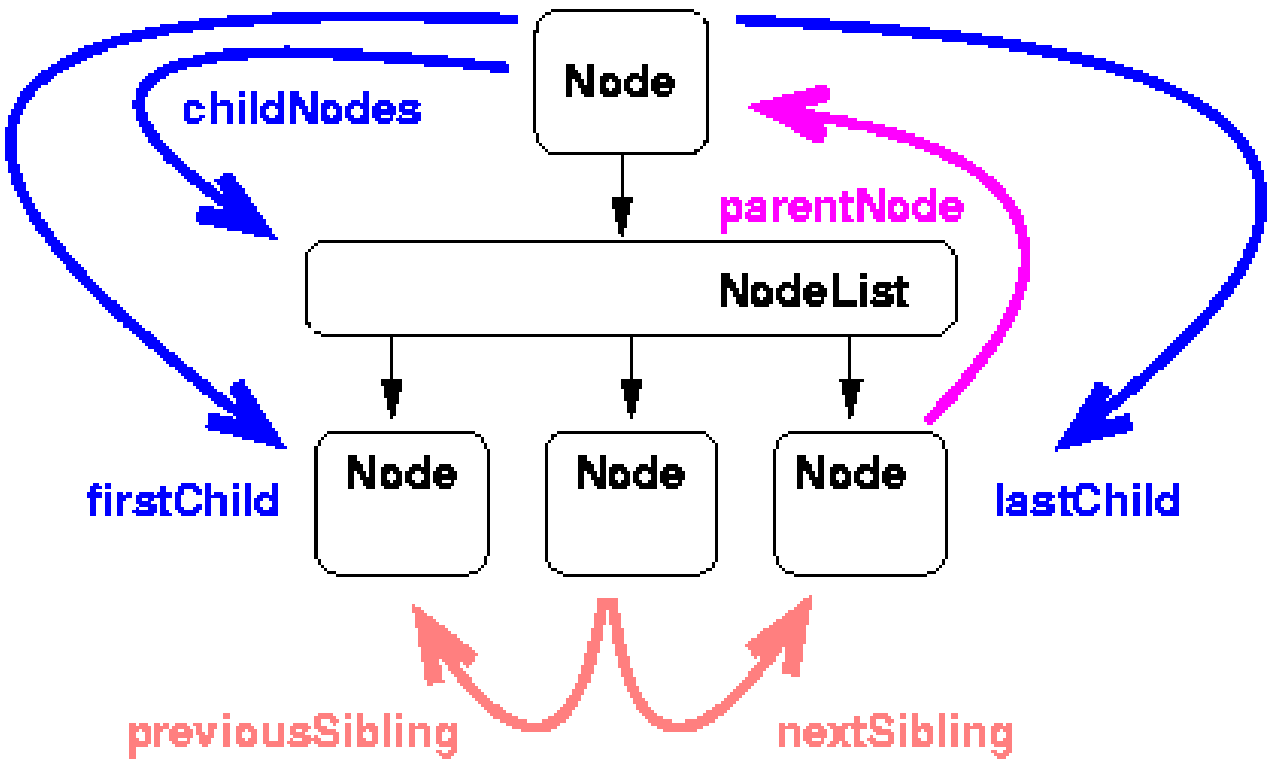
Validation

```
<program>
<block>
<bindings>
<binding><variable name='x'/>
        <initialisation><float value='111'/></initialisation>
</binding>
<binding><variable name='y'/>
        <initialisation><boolean value='true'/></initialisation>
</binding>
</bindings>
<body>
<alternative>
<condition>
        <binaryOperation operator='=='>
        <leftOperand><variable name='y'</leftOperand>
        <rightOperand>
                <unaryOperation operator='-'>
                <operand> <float value='3.14'/></rightOperand> </operand>
                </unaryOperation>
        </binaryOperation>
</condition>
<consequence>
        <invocation>
        <function><variable name='print'/></function>
        <arguments>
        . . .
```

Si le programme est validé vis-à-vis la grammaire, il sera réifié en DOM (*Document Object Model*).

Interface DOM

L'interface DOM (pour *Document Object Model*) lit le document XML et le convertit entièrement en un arbre (en fait un graphe modifiable).
DOM est une interface, il faut lui adjoindre une implantation et, pour XML, il faut adjoindre un analyseur syntaxique (pour *parser*)



Verification

Paquetage `org.w3c.dom.*`
Implantations : `javax.xml.parsers.*`, `org.xml.sax.*`
RelaxNG : `com.thaiopensource.validate.ValidationDriver;`

```
1  public IASTprogram getProgram() throws ParseException {
2  try {
3   final String programText = input.getText();
4   final String rngFilePath = rngFile.getAbsolutePath();
5   final InputSource isg = ValidationDriver.fileInputSource(rngFilePath);
6   final ValidationDriver vd = new ValidationDriver();
7   vd.loadSchema(isg);
8   InputSource is = new InputSource(new StringReader(programText));
9   if (  vd.validate(is) ) throw new ParseException("Invalid XML program");
10  }
11      ...
12  }
```

Conversion à DOM

```
1   public IASTprogram getProgram() throws ParseException {
2       try {
3           ...
4           final DocumentBuilderFactory dbf =
5               DocumentBuilderFactory.newInstance();
6           final DocumentBuilder db = dbf.newDocumentBuilder();
7           is = new InputSource(new StringReader(programText));
8           final Document document = db.parse(is);
9          ...
10      } catch (ParseException e) {
11          throw e;
12      } catch (Exception e) {
13          throw new ParseException(e);
14      }
15  }
```

Arpentage du DOM

- `org.w3c.dom.Document`

```
Element getDocumentElement();
```

- `org.w3c.dom.Node`

```
Node.uneCONSTANTE getNodeType();
// avec Node.DOCUMENT_NODE, Node.ELEMENT_NODE,
// Node.TEXT_NODE ...
NodeList getChildNodes();
```

- `org.w3c.dom.Element` hérite de `Node`

```
String getTagName();
String getAttribute("attributeName");
```

- `org.w3c.dom.Text` hérite de `Node`

```
String getData();
```

- `org.w3c.dom.NodeList`

```
int getLength();
Node item(int);
```

Conversion DOM vers AST

```
1   public IASTprogram getProgram()
2     throws ParseException {
3       try {
4           ... Validation
5           }
6           ... XML to DOM
7           final Document document = db.parse(is);
8          IASTprogram program = parse(document);
9           return program;
10      } catch (ParseException e) {
11          throw e;
12      } catch (Exception e) {
13          throw new ParseException(e);
14      }
15  }
```

La classe Parser

La conversion est effectuée par la méthode `parse(Node)` de la classe Parser :

```
1  package com.paracamplus.ilp1.ast;
2
3  public class Parser extends AbstractExtensibleParser {
4
5    public Parser(IParserFactory factory) {
6      super(factory);
7          addMethod("alternative", Parser.class);
8          addMethod("sequence", Parser.class);
9          addMethod("integerConstant", Parser.class, "integer");
10         addMethod("floatConstant", Parser.class, "float");
11         addMethod("stringConstant", Parser.class, "string");
12         addMethod("booleanConstant", Parser.class, "boolean");
13         addMethod("unaryOperation", Parser.class);
14         addMethod("binaryOperation", Parser.class);
15         addMethod("block", Parser.class);
16         addMethod("binding", Parser.class);
17         addMethod("variable", Parser.class);
18         addMethod("invocation", Parser.class);
19   }
```

Méthode pour l'alternative

```
1  public IASTexpression alternative (Element e) throws ParseException {
2     IAST iastc = findThenParseChildContent(e, "condition");
3     IASTexpression condition = narrowToIASTexpression(iastc);
4     IASTexpression[] iaste =
5        findThenParseChildAsExpressions(e, "consequence");
6     IASTexpression consequence = getFactory().newSequence(iaste);
7        try {
8            IASTexpression[] iasta =
9               findThenParseChildAsExpressions(e, "alternant");
10           IASTexpression alternant = getFactory().newSequence(iasta);
11            return getFactory().newAlternative(
12                    condition, consequence, alternant);
13       } catch (ParseException exc) {
14            return getFactory().newAlternative(
15                    condition, consequence, null);
16       }
17    }
```