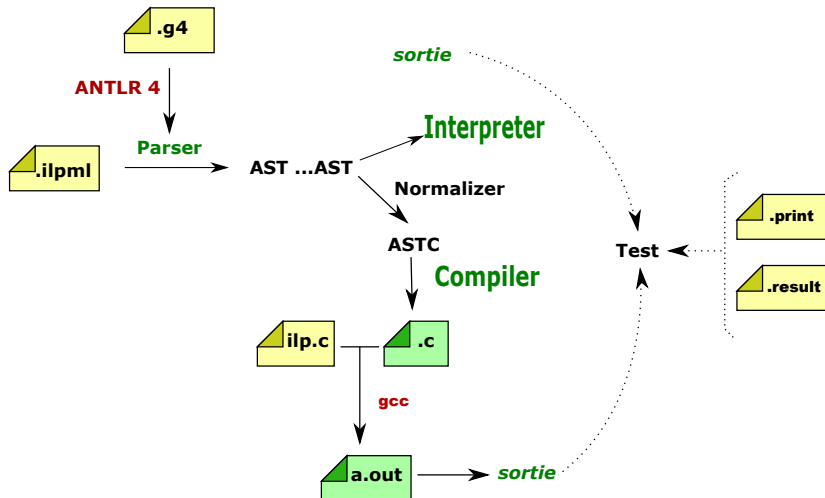


Sorbonne Université
Master Informatique 2020-2021
Spécialité STL
Développement des langages de programmation
DLP – MU4IN501

Carlos Agon
agonc@ircam.fr

Grand schéma



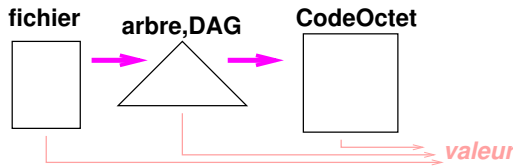
Plan du cours 2

- Interprétation
- Représentation des concepts
- Bibliothèque d'exécution
- Fabriques et visiteurs

Interprétation

Analyser la représentation du programme pour en calculer la valeur et l'effet.

Un large spectre de techniques :



- interprétation pure sur chaîne de caractères : lent
- interprétation d'arbre (ou DAG) : rapide, traçable
- interprétation de code-octet : rapide, compact, portable

Puzzles sémantiques

Les programmes suivants sont-ils légaux ? sensés ? Que font-ils ?

```
let x = print in 3;
```

```
let x = print in x(3);
```

```
let print = 3 in print(print);
```

```
if true then 1 else 2;
```

```
if 1 then 2 else 3;
```

```
if 0 then 1 else 2;
```

```
if "" then 1 else 2;
```

Concepts présents dans ILP1

- Les structures de contrôle : alternative, séquence, bloc local
- les opérateurs : `+`, `-`, etc.
- des variables prédéfinies : `pi`
- les fonctions primitives : `print`, `newline`
- instruction, expression, variable, opération, invocation
- les valeurs : entiers, flottants, chaînes, booléens.

Tous ces concepts existent en Java.

Hypothèses

L'interprète est écrit en Java 8.

- ❶ Il prend un IAST,
- ❷ calcule sa valeur,
- ❸ exécute son effet.

Il ne se soucie donc pas des problèmes syntaxiques (d'ILP1) mais uniquement des problèmes sémantiques.

Représentation des valeurs

On s'appuie sur Java :

- Les booléens par des `Boolean`
- Les entiers seront représentés par des `BigInteger`
- Les flottants par des `Double`
- Les chaînes par des `String`

En définitive, une valeur d'ILP1 sera un `Object` Java.
D'autres choix sont bien sûr possibles.

Le cas des nombres

La grammaire d'ILP1 permet le programme suivant (en syntaxe C) :

```
{ i = 1234567890123456789012345678901234567890;  
  f = 1.234567890123456789012345e-234567890123;  
  ...
```

Une restriction d'implantation est que les flottants sont limités aux valeurs que prennent les `double` en revanche les entiers sont scrupuleusement respectés.

Environnement

- En tout point, l'**environnement** est l'ensemble des noms utilisables en ce point.
- Le bloc local introduit une variable locale.
- Des variables globales existent également qui nomment les fonctions (primitives) prédéfinies : `print`, `newline` ou bien la constante `pi`.
- On distingue donc l'environnement **global** de l'environnement **local** (ou **lexical** = une zone de code (la portée) où on a le droit d'utiliser la variable)

Interprétation

L'interprétation est donc un processus calculant une valeur et réalisant un effet à partir :

- ❶ d'un code (expression ou instruction)
- ❷ et d'un environnement.

Classiquement on définit une méthode `eval` sur les AST

```
valeur = code.eval(environnement);
```

L'effet est un « effet secondaire » sur le flux de sortie.

Bibliothèque d'exécution

- L'environnement contient des fonctions qui s'appuient sur du code qui doit être présent pour que l'interprète fonctionne (gestion de la mémoire, des environnements, des canaux d'entrée/sortie, etc.). Ce code forme la **bibliothèque d'exécution**. Pour l'interprète d'ILP1, elle est écrite en Java.
- La bibliothèque d'exécution (ou *runtime*) de Java est écrite en Java et en C et comporte la gestion de la mémoire, des tâches, des entités graphiques, etc. ainsi que l'interprète de code-octet.
- Est **primitif** ce qui ne peut être défini dans le langage.
- Est **prédéfini** ce qui est présent avant toute exécution.

La bibliothèque permet d'exécuter :

`sin(2 π); beep;`

Pas de *runtime* en C

Environnement

ILP1 a deux espaces de noms :

- l'environnement des variables (extensibles avec `let`)
- l'environnement global (immuable)

L'**environnement** est formé de ces deux espaces de noms.

Interprète en Java

- On sépare environnement lexical et global.
- Deux environnements globaux :
 - pour les opérateurs
 - pour les fonctions prédéfinies et les constantes
- Des exceptions peuvent surgir !
- On souhaite se réserver le droit de changer d'implantation d'environnements.

Interprète en Java

```
1 public Object eval (
2     IAST iast,
3     ILexicalEnvironment lexenv,
4     IGlobalVariableEnvironment globalVariableEnvironment,
5     IOperatorEnvironment operatorEnvironment )
6     throws EvaluationException
7 {
8     try {
9         return ...
10    } catch (Exception exc) {
11        return ...
12    }
13 }
```

ILexicalEnvironment

```

1 public interface ILexicalEnvironment
2 extends
3 IEnvironment<IASTvariable, Object, EvaluationException> {
4
5 ILexicalEnvironment extend(IASTvariable variable, Object value);
6 ILexicalEnvironment getNext() throws EvaluationException;
7 }

```

Hérite de la classe IEnvironment

```

1 public interface IEnvironment<K,V,T extends Throwable> {
2 /** is the key present in the environment ? */
3 boolean isPresent(K key);
4 IEnvironment<K,V,T> extend(K key, V value);
5 K getKey() throws T;
6 V getValue(K key) throws T;
7 void update(K key, V value) throws T;
8 // Low level interface:
9 boolean isEmpty();
10 IEnvironment<K,V,T> getNext() throws T;
11 }

```


IGlobalVariableEnvironment

```
1 public interface IGlobalVariableEnvironment {  
2  
3     Object getGlobalVariableValue (String variableName);  
4  
5     void addGlobalVariableValue  
6         (String variableName, Object value);  
7  
8     void addGlobalVariableValue (IPrimitive primitive);  
9  
10    void updateGlobalVariableValue  
11        (String variableName, Object value);  
12 }
```

Ressource: [ilp1/interpreter/interfaces/IGlobalVariableEnvironment.java](#)

IOperatorEnvironment

```

1 import com.paracamplus.ilp1.interfaces.IASToperator;
2
3 public interface IOperatorEnvironment {
4     IOperator getUnaryOperator (IASToperator operator)
5         throws EvaluationException;
6     IOperator getBinaryOperator (IASToperator operator)
7         throws EvaluationException;
8     void addOperator (IOperator operator)
9         throws EvaluationException;
10 }

```

Ressource: [ilp1/interpreter/interfaces/IOperatorEnvironment.java](#)

Un opérateur n'est pas un « **citoyen de première classe** », il ne peut qu'être appliqué. On ne peut pas écrire `let x = + in ...`.

```

1 package com.paracamplus.ilp1.interpreter.interfaces;
2
3 public interface IOperator {
4     String getName();
5     int getArity();
6     Object apply(Object ... argument) throws EvaluationException;
7 }

```

Opérateurs

Les codes de bien des opérateurs se ressemblent à quelques variations syntaxiques près : il faut factoriser !

Pour ce faire, on utilise un macro-générateur (un bon exemple est PHP <http://www.php.net/>).

```
texte ----MacroGénérateur----> texte.java
```

Des patrons définissent les différents opérateurs de la bibliothèque d'exécution :

Patron des comparateurs arithmétiques

```
1 private Object operatorLessThan
2     (final String opName, final Object a, final Object b)
3     throws EvaluationException {
4     checkNotNull(opName, 1, a);
5     checkNotNull(opName, 2, b);
6     if ( a instanceof BigInteger ) {
7         final BigInteger bi1 = (BigInteger) a;
8         if ( b instanceof BigInteger ) {
9             final BigInteger bi2 = (BigInteger) b;
10            return Boolean.valueOf(bi1.compareTo(bi2) < 0);
11        } else if ( b instanceof Double ) {
12            final double bd1 = bi1.doubleValue();
13            final double bd2 = ((Double) b).doubleValue();
14            return Boolean.valueOf(bd1 < bd2);
15        } else {
16            return signalWrongType(opName, 2, b, "number");
17        }
18    } else if ( a instanceof Double ) {
19        ...
    }
```

Fonctions génériques

ILP1 n'est pas typé statiquement.

ILP1 est typé dynamiquement : chaque valeur a un type (pour l'instant booléen, entier, flottant, chaîne).

Un opérateur arithmétique peut donc être appliqué à :

argument1	argument2	résultat
entier	entier	entier
entier	flottant	flottant
flottant	entier	flottant
flottant	flottant	flottant
<i>autre</i>	<i>autre</i>	Erreur !

Méthode binaire, **contagion flottante !**

Évaluation

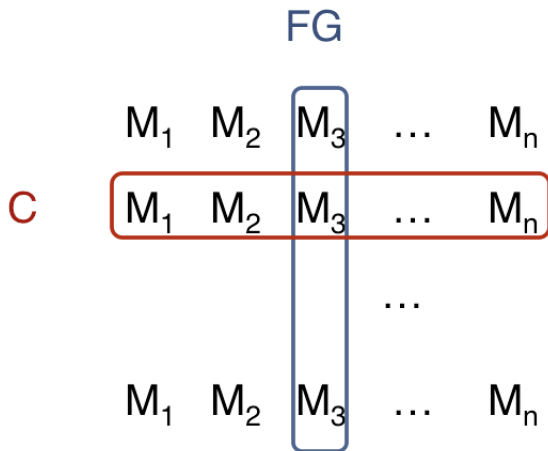
- Évaluation des structures de contrôle
- Évaluation des constantes, des variables
- Évaluation des invocations, des opérations

Problème !

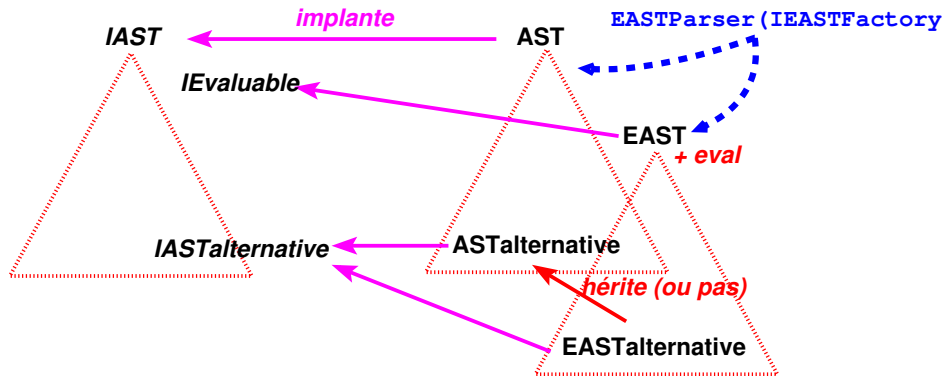
Comment installer la méthode `eval` ?

- ❶ il est interdit de modifier une interface comme `IAST`
- ❷ on ne peut modifier le code du cours précédent `Parser`

Organisation des méthodes



Solution 1 : une méthode eval par IAST



Fabrique : interface

Une **fabrique** permet de maîtriser explicitement le processus d'instanciation.

```
1 public interface IParserFactory {  
2     IASTprogram newProgram(  
3         IASTexpression expression);  
4  
5     IASTexpression newSequence(IASTexpression[] asts);  
6  
7     IASTexpression newAlternative(  
8         IASTexpression condition,  
9         IASTexpression consequence,  
10        IASTexpression alternant);  
11  
12    IASTvariable newVariable(String name);  
13  
14    IASTexpression newBinaryOperation(  
15        IASToperator operator,  
16        IASTexpression leftOperand,  
17        IASTexpression rightOperand);  
18  
19    ...
```

Fabrique : implantation

```
1 public class ASTfactory implements IParserFactory {
2
3     public IASTprogram newProgram
4         (IASTexpression expression) {
5         return new ASTprogram(expression);
6     }
7
8     public IASTsequence newSequence
9         (IASTexpression[] asts) {
10        return new ASTsequence(asts);
11    }
12
13    public IASTalternative newAlternative
14        (IASTexpression condition,
15         IASTexpression consequence,
16         IASTexpression alternant) {
17        return
18            new ASTalternative(condition, consequence, alternant);
19    }
20    ...
}
```

Emploi de la fabrique

```
1 public class Parser extends AbstractExtensibleParser {
2
3 public Parser(IParserFactory factory) {
4     super(factory);
5     ...
6 }
7 public IParserFactory getFactory() {
8     return factory;
9 }
10 protected final IParserFactory factory;
11
12 public IASTExpression alternative (Element e) throws ParseException {
13     IAST iastc = findThenParseChildContent(e, "condition");
14     IASTExpression condition = narrowToIASTExpression(iastc);
15     IASTExpression[] iaste =
16         findThenParseChildAsExpressions(e, "consequence");
17     IASTExpression consequence = getFactory().newSequence(iaste);
18     try {
19         IASTExpression[] iasta =
20             findThenParseChildAsExpressions(e, "alternant");
21         IASTExpression alternant = getFactory().newSequence(iasta);
22         return getFactory().newAlternative(
23             condition, consequence, alternant);
24     } catch (ParseException exc) {
25         return getFactory().newAlternative(
26             condition, consequence, null);
27     }
28 }
```

Solution 2 : une classe Interpréteur avec n des méthodes eval

```
1 public class Interpreter {
2
3     public Interpreter (Environment env) {
4         this.environment = env;
5     }
6     protected final Environment env;
7
8     public String Interprete (IAST iast,
9                             Environment env)
10
11     if ( iast instanceof IASTconstant ) {
12         if ( iast instanceof IASTboolean ) {
13             eval((IASTboolean) iast, env);
14         } ...
15     } else {
16         final String msg = "Unknown type of constant: " + iast;
17         throw new EvalException(msg);
18     }
19 } else if ( iast instanceof IASTalternative ) {
20     eval((IASTalternative) iast, env);
21 } else if ( iast instanceof IASToperation ) {
22     if ( iast instanceof IASTunaryoperation ) {
23         eval((IASTunaryoperation) iast, env);
24     } else {
25         final String msg = "Unknown type of operation: " + iast;
26         throw new EvalException(msg);
27     }
28 } else if ( iast instanceof IASTsequence ) {
29     eval((IASTsequence) iast, env);
30 } ...
31 }
32 }
```

```
1  protected void eval (final IASTalternative iast,  
2      Environment env)  
3      ...  
4  }  
5  
6  protected void eval (final IASTbinaryOperation iast,  
7      Environment env)  
8      ...  
9  }  
10  
11 protected void eval (final IASTinvocation iast,  
12     Environment env)  
13     ...  
14 }  
15  
16 protected void eval (final IASTsequence iast,  
17     Environment env)  
18     ...  
19 }  
20 ...
```

Solution adoptée : un visiteur

```
1 public interface IASTvisitor
2 <Result, Data, Anomaly extends Throwable> {
3     Result visit(IASTalternative iast, Data data)
4         throws Anomaly;
5     Result visit(IASTbinaryOperation iast, Data data)
6         throws Anomaly;
7     Result visit(IASTblock iast, Data data)
8         throws Anomaly;
9     ...
10    Result visit(IASTinvocation iast, Data data)
11        throws Anomaly;
12    Result visit(IASToperator iast, Data data)
13        throws Anomaly;
14 }
```

```
1 public interface IASTvisitable {
2     <Result, Data, Anomaly extends Throwable>
3     Result accept(IASTvisitor<Result, Data, Anomaly> visitor,
4                   Data data) throws Anomaly;
5 }
```

IAST visitables

```
1 public abstract interface IASTexpression extends IAST, IASTvisitable {  
2 }
```

Par exemple

```
1  
2 public class ASTsequence extends ASTexpression implements IASTsequence {  
3     public ASTsequence (IASTexpression[] expressions) {  
4         this.expressions = expressions;  
5     }  
6     protected IASTexpression[] expressions;  
7  
8     @Override  
9     public IASTexpression[] getExpressions() {  
10         return this.expressions;  
11     }  
12  
13     @Override  
14     public <Result, Data, Anomaly extends Throwable>  
15         Result accept(IASTvisitor<Result, Data, Anomaly> visitor, Data data)  
16             throws Anomaly {  
17         return visitor.visit(this, data);  
18     }  
19 }
```


Parcours de la structure

Le parcours est réalisé par le visiteur, exemple d'un collecteur de variables globales :

```
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> analyze(IASTprogram program)
12        throws CompilationException {
13        result = program.getBody().accept(this, result);
14        return result;
15    }
```

```
1
2  @Override
3  public Set<IASTCglobalVariable> visit(
4      IASTsequence iast,
5      Set<IASTCglobalVariable> result)
6      throws CompilationException {
7      for ( IASTexpression expr : iast.getExpressions() ) {
8          result = expr.accept(this, result);
9      }
10     return result;
11 }
12
13
14 @Override
15 public Set<IASTCglobalVariable> visit(
16     IASTCglobalVariable iast,
17     Set<IASTCglobalVariable> result)
18     throws CompilationException {
19     result.add(iast);
20     return result;
21 }
22
23 @Override
24 public Set<IASTCglobalVariable> visit(
25     IASTClocalVariable iast,
26     Set<IASTCglobalVariable> result)
27     throws CompilationException {
28     return result;
29 }
```

Avantages des visiteurs

- + preserve les classes,
- + code similaire au fonctionnel, mais il n'y a plus besoin d'écrire le code de discrimination,
- + une seule méthode accept pour une famille des visiteurs,
 - une duplication de la méthode accept,
 - l'héritage des visiteur peut devenir compliqué, on verra...

Interpreter

```
1 package com.paracamplus.ilp1.interpreter;
2
3 public class Interpreter
4 implements IASTvisitor<Object, ILexicalEnvironment, EvaluationException> {
5
6     public Interpreter (IGlobalVariableEnvironment globalVariableEnvironment,
7                         IOperatorEnvironment operatorEnvironment ) {
8         this.globalVariableEnvironment = globalVariableEnvironment;
9         this.operatorEnvironment = operatorEnvironment;
10    }
11    protected IGlobalVariableEnvironment globalVariableEnvironment;
12    protected IOperatorEnvironment operatorEnvironment;
13
14    public IOperatorEnvironment getOperatorEnvironment() {
15        return operatorEnvironment;
16    }
17
18    public IGlobalVariableEnvironment getGlobalVariableEnvironment() {
19        return globalVariableEnvironment;
20    }
21
22    //
23
24    public Object visit(IASTprogram iast, ILexicalEnvironment lexenv)
25        throws EvaluationException {
26        try {
27            return iast.getBody().accept(this, lexenv);
28        } catch (Exception exc) {
29            return exc;
30        }
31    }
```

Alternative

```
1 public Object visit(IASAlternative iast,  
2     ILexicalEnvironment lexenv)  
3     throws EvaluationException {  
4     Object c = iast.getCondition().accept(this, lexenv);  
5     if ( c != null && c instanceof Boolean ){  
6         Boolean b = (Boolean) c;  
7         if ( b.booleanValue() ) {  
8             return iast.getConsequence().accept(this, lexenv);  
9         } else if ( iast.isTernary() ) {  
10            return iast.getAlternant().accept(this, lexenv);  
11        } else {  
12            return whatever;  
13        }  
14    } else {  
15        return iast.getConsequence().accept(this, lexenv);  
16    }  
17 }
```

Séquence

```
1 public Object visit(IASTsequence iast,  
2     ILexicalEnvironment lexenv)  
3     throws EvaluationException {  
4     IASTexpression[] expressions = iast.getExpressions();  
5     Object lastValue = null;  
6     for ( IASTexpression e : expressions ) {  
7         lastValue = e.accept(this, lexenv);  
8     }  
9     return lastValue;  
10 }
```

Block

```
1 public Object visit(IASTblock iast,  
2     ILexicalEnvironment lexenv)  
3     throws EvaluationException {  
4     ILexicalEnvironment lexenv2 = lexenv;  
5     for ( IASTbinding binding : iast.getBindings() ) {  
6         Object initialisation =  
7             binding.getInitialisation().accept(this, lexenv);  
8         lexenv2 = lexenv2.extend(binding.getVariable(),  
9             initialisation);  
10    }  
11    return iast.getBody().accept(this, lexenv2);  
12 }
```

Constante

Toutes les constantes ont une valeur décrite par une chaîne.

```
1 public class ASTinteger extends ASTconstant implements IASTinteger {
2
3     public ASTinteger (String description) {
4         super(description, new BigInteger(description));
5     }
6     @Override
7     public BigInteger getValue() {
8         return (BigInteger) super.getValue();
9     }
10
11     @Override
12     public <Result, Data, Anomaly extends Throwable>
13         Result accept(IASTvisitor<Result, Data, Anomaly> visitor, Data data)
14             throws Anomaly {
15         return visitor.visit(this, data);
16     }
17 }
```

```
1 public Object visit(IASTinteger iast, ILexicalEnvironment lexenv)
2     throws EvaluationException {
3     return iast.getValue();
4 }
```

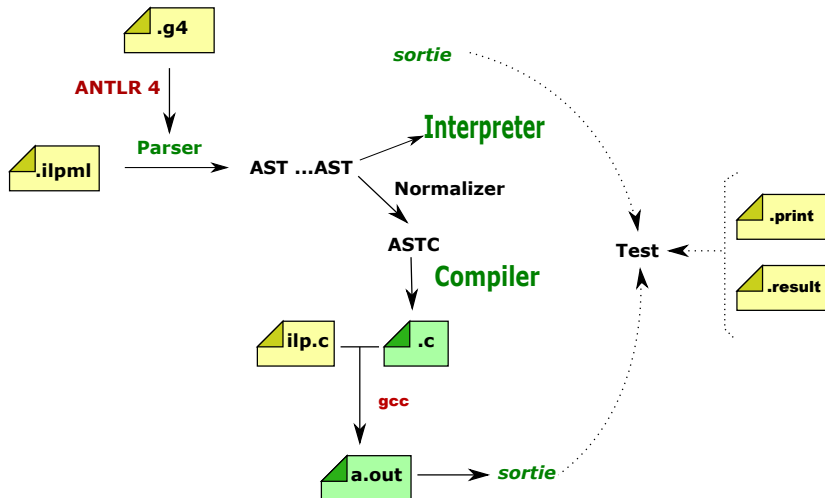

Variable

```
1 public Object visit(IASTvariable iast,  
2     ILexicalEnvironment lexenv)  
3     throws EvaluationException {  
4     try {  
5         return lexenv.getValue(iast);  
6     }  
7     catch (EvaluationException exc) {  
8         return getGlobalVariableEnvironment()  
9             .getGlobalVariableValue(iast.getName());  
10    }  
11 }
```

Invocation

```
1
2 public Object visit(IASTinvocation iast,
3     ILexicalEnvironment lexenv)
4     throws EvaluationException {
5     Object function =
6         iast.getFunction().accept(this, lexenv);
7     if ( function instanceof Invocable ) {
8         Invocable f = (Invocable)function;
9         List<Object> args = new Vector<Object>();
10        for ( IASTexpression arg : iast.getArguments() ) {
11            Object value = arg.accept(this, lexenv);
12            args.add(value);
13        }
14        return f.apply(this, args.toArray());
15    } else {
16        String msg = "Cannot apply " + function;
17        throw new EvaluationException(msg);
18    }
19 }
```

Grand schéma



JUnit 4

Les tests ne sont plus déclarés par héritage mais par annotation (cf. aussi TestNG). Les annotations sont (sur les méthodes) :

`@BeforeClass`

`@Before`

`@Test`

`@Test(expected = Exception.class)`

`@After`

`@AfterClass`

et quelques autres comme (sur les classes) :

`@RunWith` `@SuiteClasses`

`@Parameters`

Test d'ILP : exemple de l'interprète

```
1 @RunWith(Parameterized.class)
2 public class InterpreterTest {
3
4     @Test
5     public void processFile() throws ParseException, IOException, EvaluationException {
6         InterpreterRunner run = new InterpreterRunner();
7         configureRunner(run);
8         run.testFile(file);
9         run.checkPrintingAndResult(file);
10    }
11
12    public void configureRunner(InterpreterRunner run) throws EvaluationException {
13        IASTfactory factory = new ASTfactory();
14        run.setILMPLParser(new ILMPLParser(factory));
15        IGlobalVariableEnvironment gve = new GlobalVariableEnvironment();
16        GlobalVariableStuff.fillGlobalVariables(gve, stdout);
17        IOperatorEnvironment oe = new OperatorEnvironment();
18        OperatorStuff.fillUnaryOperators(oe);
19        OperatorStuff.fillBinaryOperators(oe);
20        Interpreter interpreter = new Interpreter(gve, oe);
21        run.setInterpreter(interpreter);
22    }
23
24    @Parameters(name = "{0}")
25    public static Collection<File[]> data() throws Exception {
26        return InterpreterRunner.getFileList(samplesDirName, pattern);
27    }
28
29 }
```

Test d'ILP : exemple de l'interprète (suite)

```
1 public class InterpreterRunner {
2
3     protected Interpreter interpreter;
4
5     public void testFile(File file)
6         throws ParseException, IOException, EvaluationException {
7         System.err.println("Testing " + file.getAbsolutePath() + " ...");
8         assertTrue(file.exists());
9         IASTprogram program = parser.parse(file);
10        interpretProgram(program);
11    }
12
13    public void interpretProgram(IASTprogram program) throws EvaluationException {
14        if (interpreter == null) {
15            throw new EvaluationException("interpreter not set");
16        }
17        ILexicalEnvironment lexenv = new EmptyLexicalEnvironment();
18        result = interpreter.visit(program, lexenv);
19        printing = stdout.toString();
20        System.out.println("  Value: " + result);
21        if ( ! "".equals(printing) ) {
22            System.out.println("  Printing: " + printing);
23        }
24    }
25 }
26 }
```

Récapitulation

- interprétation,
- choix de représentation (à l'exécution) des valeurs,
- bibliothèque d'exécution,
- environnement lexical d'exécution,
- visiteurs,
- annotations.