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

Carlos Agon
agonc@ircam.fr

Librement inspiré du cours ILP de Christian Queinnec.

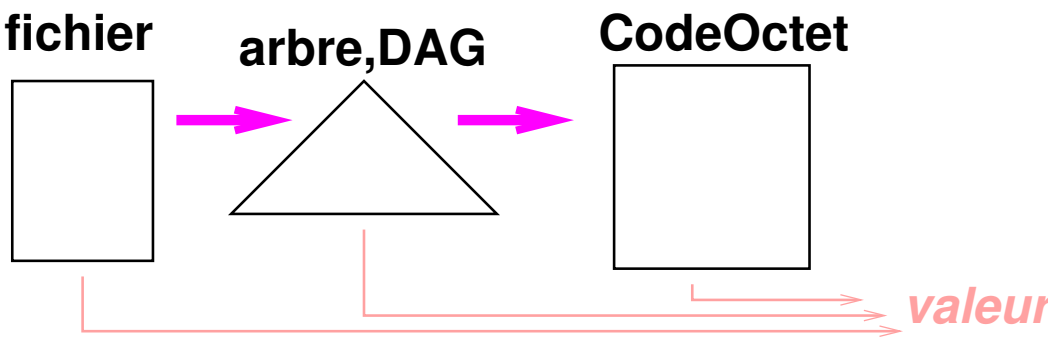
Slides et mp3 sur le site :
`www-master.ufr-info-p6.jussieu.fr/2007/Ext/queinnec/ILP/`

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

Une machine abstraite (super simple)

Le langage :

e:= N | e + e | e - e

Le jeu d'instructions de la machine :

CONST(N)	empiler l'entier N
ADD	dépiler deux entiers, empiler leur somme
SUB	dépiler deux entiers, empiler leur différence

Schéma de compilation :

C[N] = CONST(N)

C[a1 + a2] = C(a1); C[a2]; ADD

C[a1 - a2] = C(a1); C[a2]; SUB

Exemple :

C[3 – 1 + 2] = CONST(3); CONST(1) ; CONST(2);ADD ; SUB

Une machine abstraite arithmétique

Composants de la machine :

① Un pointeur de code
② Une pile

Transactions de la machine :

Etat avant		Etat après	
Code	Pile	Code	Pile
CONST(n);c	s	c	n.s
ADD;c	n2.n1;s	c	(n1 + n2).s
SUB;c	n2.n1;s	c	(n1 - n2).s

Evaluation

Etat initial code = C[exp] et pile = ε

Etat final code = ε et pile = v. ε v le résultat

Code	Pile
`CONST(3) ; CONST(1) ; CONST(2) ; ADD ; SUB`	ε
`CONST(1) ; CONST(2) ; ADD ; SUB`	3.ε
`CONST(2) ; ADD ; SUB`	1.3.ε
`ADD ; SUB`	2.1.3ε
`SUB`	3.3.ε
ε	0.ε

Exécution du code par interprétation

Interprète écrit en C ou assembler.

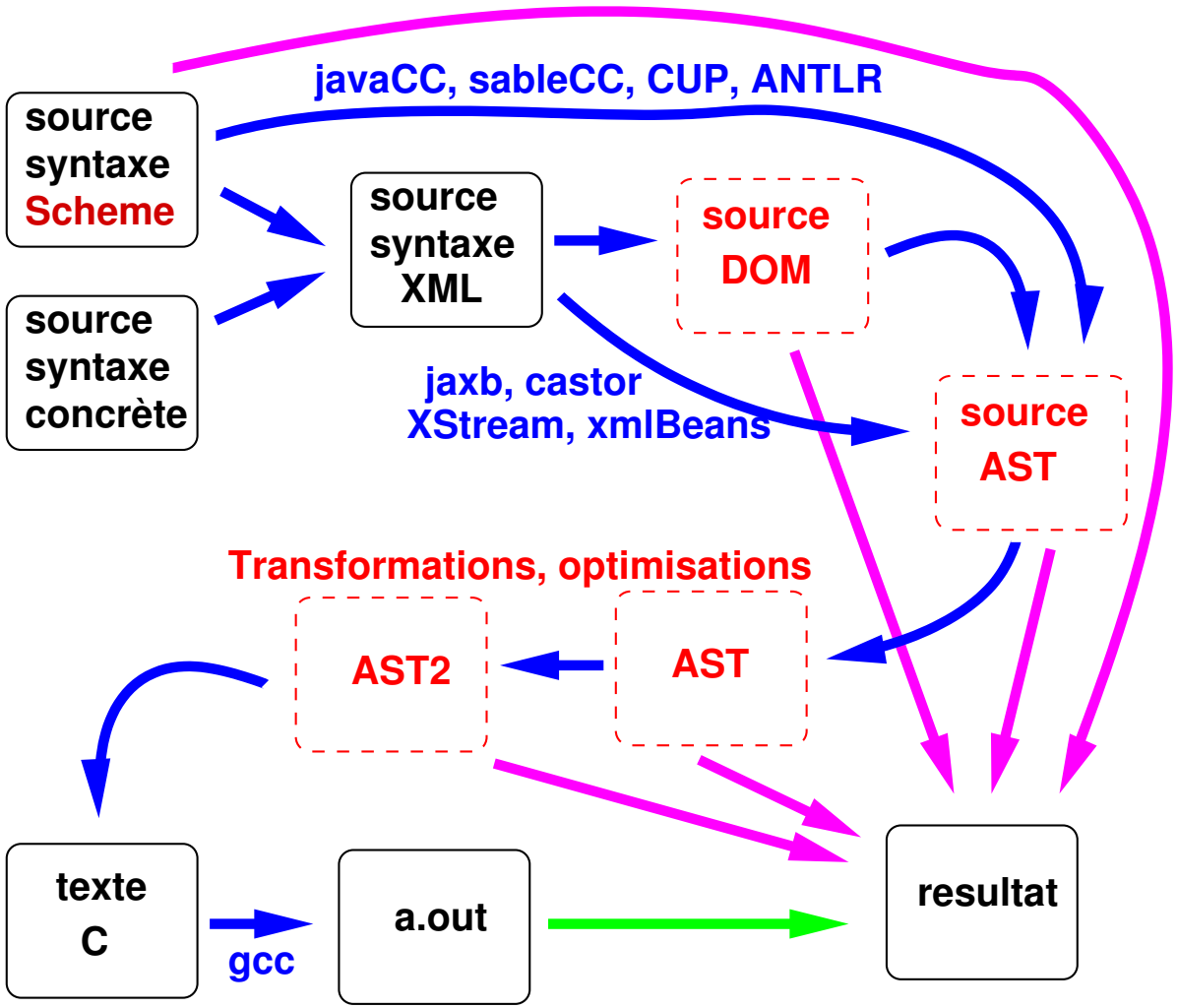
```
int interpreter(int * code)
{
int * s = bottom_of_stack;
while (1) {
 switch (*code++) {
  case CONST: *s++ = *code++; break;
  case ADD: s[-2] = s[-2] + s[-1]; s--; break;
  case SUB: s[-2] = s[-2] - s[-1]; s--; break;
  case EPSILON: return s[-1];
  }
 }
}
```

Exécution du code par expansion

Plus vite encore, convertir les instructions abstraites en séquences de code machine.

CONST(i)	--->	pushl $i
ADD	--->	popl %eax addl 0(%esp), %eax
SUB	--->	popl %eax subl 0(%esp), %eax
EPSILON	--->	popl %eax ret

Grand schéma



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.

1. Il prend un IAST,
2. calcule sa valeur,
3. 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 = 12345678901234567890123456789012345678890;
  f = 1.23456789012345678901234e-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 :

1. d'un code (expression ou instruction)
2. 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'executer :
$\sin(2\pi)$; 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

```
public Object eval (
IAST iast,
ILexicalEnvironment lexenv,
IGlobalVariableEnvironment globalVariableEnvironment,
IOperatorEnvironment operatorEnvironment )
            throws EvaluationException
{
        try {
            return ...
        } catch (Exception exc) {
            return ...
        }
   }
```

ILexicalEnvironment

```
public interface ILexicalEnvironment
extends
  IEnvironment<IASTvariable, Object, EvaluationException> {

ILexicalEnvironment extend(IASTvariable variable, Object value);
ILexicalEnvironment getNext() throws EvaluationException;
}
```

Hérite de la classe `IEnvironment`

```
public interface IEnvironment<K,V,T extends Throwable> {
        /** is the key present in the environment ? */
        boolean isPresent(K key);
        IEnvironment<K,V,T> extend(K key, V value);
        K getKey() throws T;
        V getValue(K key) throws T;
        void update(K key, V value) throws T;
        // Low level interface:
        boolean isEmpty();
        IEnvironment<K,V,T> getNext() throws T;
}
```

IGlobalVariableEnvironment

```
public interface IGlobalVariableEnvironment {
    Object getGlobalVariableValue (String variableName);
    void addGlobalVariableValue
                (String variableName, Object value);
    void addGlobalVariableValue (IPrimitive primitive);
    void updateGlobalVariableValue
                (String variableName, Object value);
}
```

Ressource: ilp1/interpreter/interfaces/IGlobalVariableEnvironment.java

IOperatorEnvironment

```
import com.paracamplus.ilp1.interfaces.IASToperator;

public interface IOperatorEnvironment {
    IOperator getUnaryOperator (IASToperator operator)
            throws EvaluationException;
    IOperator getBinaryOperator (IASToperator operator)
            throws EvaluationException;
    void addOperator (IOperator operator)
            throws EvaluationException;
}
```

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 ...`

```
package com.paracamplus.ilp1.interpreter.interfaces;

public interface IOperator {
    String getName();
    int getArity();
    Object apply(Object ... argument) throws EvaluationException;
}
```

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 ----MacroGenerateur----> texte.java
```

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

Patron des comparateurs arithmétiques

```
private Object operatorLessThan
      (final String opName, final Object a, final Object b)
  throws EvaluationException {
  checkNotNull(opName, 1, a);
  checkNotNull(opName, 2, b);
  if ( a instanceof BigInteger ) {
    final BigInteger bi1 = (BigInteger) a;
    if ( b instanceof BigInteger ) {
      final BigInteger bi2 = (BigInteger) b;
      return Boolean.valueOf(bi1.compareTo(bi2) < 0);
    } else if ( b instanceof Double ) {
      final double bd1 = bi1.doubleValue();
      final double bd2 = ((Double) b).doubleValue();
      return Boolean.valueOf(bd1 < bd2);
    } else {
      return signalWrongType(opName, 2, b, "number");
    }
  } else if ( a instanceof Double ) {
    ...
```

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
autre	*autre*	**Erreur !**

Méthode binaire, **contagion flottante** !

Évaluation

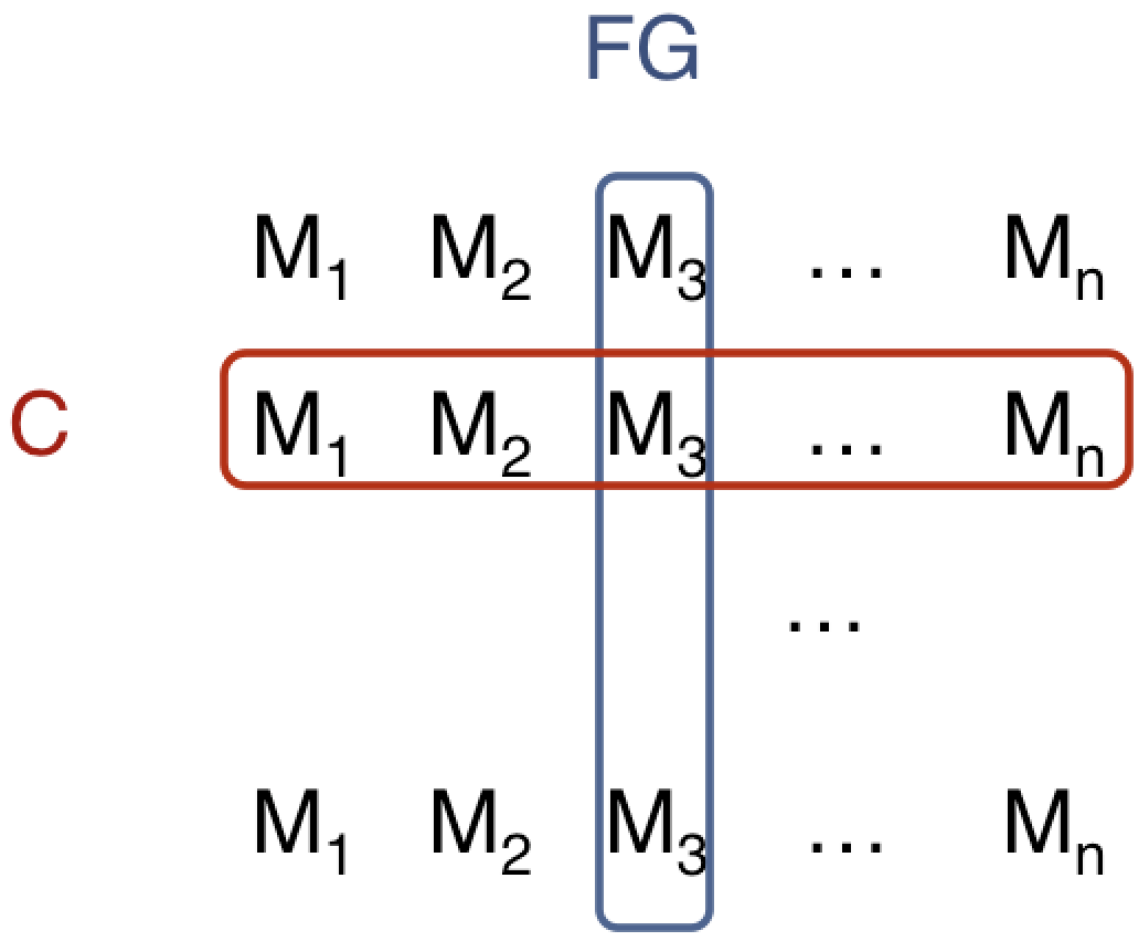
- Évaluation des structures de contrôle
- Évaluation des constantes, des variables
- Évaluation des invocations, des opérations

Problème !

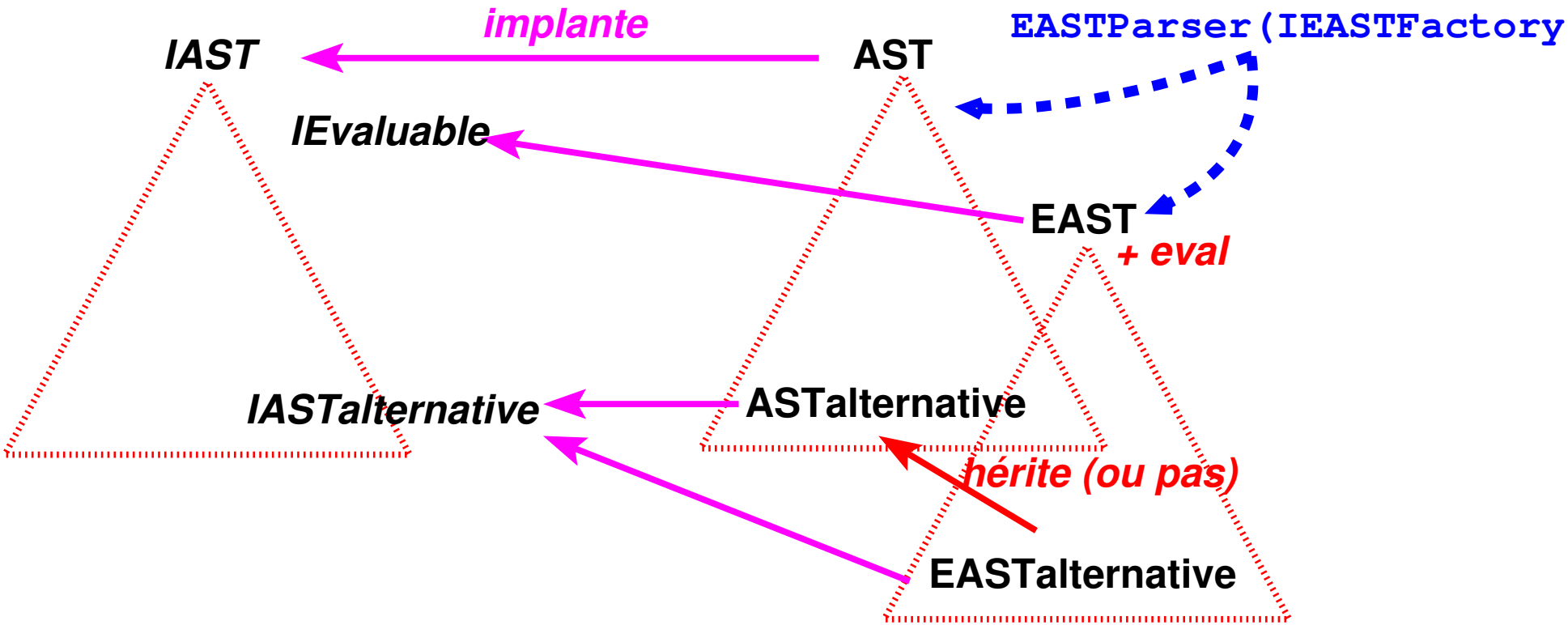
Comment installer la méthode `eval` ?

1. il est interdit de modifier une interface comme `IAST`
2. 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.

```
public interface IParserFactory {
    IASTprogram newProgram(
            IASTexpression expression);

    IASTexpression newSequence(IASTexpression[] asts);

    IASTexpression newAlternative(
            IASTexpression condition,
            IASTexpression consequence,
            IASTexpression alternant);

    IASTvariable newVariable(String name);

    IASTexpression newBinaryOperation(
            IASTOperator operator,
            IASTexpression leftOperand,
            IASTexpression rightOperand);

    ...
```

Fabrique : implantation

```
public class ASTfactory implements IParserFactory {

        public IASTprogram newProgram
                (IASTexpression expression) {
        return new ASTprogram(expression);
    }

        public IASTsequence newSequence
                (IASTexpression[] asts) {
        return new ASTsequence(asts);
    }

        public IASTalternative newAlternative
                (IASTexpression condition,
                  IASTexpression consequence,
                  IASTexpression alternant) {
        return
           new ASTalternative(condition, consequence, alternant);
    }
    ...
```

Emploi de la fabrique

```
public class Parser extends AbstractExtensibleParser {

public Parser(IParserFactory factory) {
                super(factory);
        ...
        }
public IParserFactory getFactory() {
                return factory;
        }
protected final IParserFactory factory;

public IASTexpression alternative (Element e) throws ParseException {
        IAST iastc = findThenParseChildContent(e, "condition");
        IASTexpression condition = narrowToIASTexpression(iastc);
        IASTexpression[] iaste =
                findThenParseChildAsExpressions(e, "consequence");
        IASTexpression consequence = getFactory().newSequence(iaste);
        try {
            IASTexpression[] iasta =
                    findThenParseChildAsExpressions(e, "alternant");
            IASTexpression alternant = getFactory().newSequence(iasta);
            return getFactory().newAlternative(
                    condition, consequence, alternant);
        } catch (ParseException exc) {
            return getFactory().newAlternative(
                    condition, consequence, null);
        }
    }
```

Solution 2 : une classe Interpréteur avec *n* des méthodes eval

```
public class Interpreter {

  public Interpreter (Environment env) {
    this.environment = env;
  }
  protected final Environment env;

  public String Interprete (IAST iast,
                            Environment env)

    if ( iast instanceof IASTconstant ) {
      if ( iast instanceof IASTboolean ) {
        eval((IASTboolean) iast, env);
     } ...
      } else {
        final String msg = "Unknown type of constant: " + iast;
        throw new EvalException(msg);
      }
    } else if ( iast instanceof IASTalternative ) {
      eval((IASTalternative) iast, env);
    } else if ( iast instanceof IASToperation ) {
      if ( iast instanceof IASTunaryOperation ) {
        eval((IASTunaryOperation) iast, env);
      } else {
        final String msg = "Unknown type of operation: " + iast;
        throw new EvalException(msg);
    }
    } else if ( iast instanceof IASTsequence ) {
      eval((IASTsequence) iast, env);
    } ...
    }
  }
```

```
protected void eval (final IASTalternative iast,
      Environment env)
  ...
}

protected void eval (final IASTbinaryOperation iast,
      Environment env)
  ...
}

protected void eval (final IASTinvocation iast,
      Environment env)
  ...
}

protected void eval (final IASTsequence iast,
       Environment env)
 ...
}
...
```

Solution adoptée : un visiteur

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

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

IAST visitables

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

Par exemple

```
public class ASTsequence extends ASTexpression implements IASTsequence {
    public ASTsequence (IASTexpression[] expressions) {
        this.expressions = expressions;
    }
    protected IASTexpression[] expressions;

    @Override
        public IASTexpression[] getExpressions() {
        return this.expressions;
    }

    @Override
        public <Result, Data, Anomaly extends Throwable>
    Result accept(IASTvisitor<Result, Data, Anomaly> visitor, Data data)
            throws Anomaly {
        return visitor.visit(this, data);
    }
}
```

Parcours de la structure

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

```
public class GlobalVariableCollector
implements IASTCvisitor<Set<IASTCglobalVariable>,
                        Set<IASTCglobalVariable>,
                        CompilationException> {

    public GlobalVariableCollector () {
        this.result = new HashSet<>();
    }
    protected Set<IASTCglobalVariable> result;

    public Set<IASTCglobalVariable> analyze(IASTprogram program)
            throws CompilationException {
        result = program.getBody().accept(this, result);
        return result;
    }
```

```
@Override
    public Set<IASTCglobalVariable> visit(
        IASTsequence iast,
        Set<IASTCglobalVariable> result)
                throws CompilationException {
    for ( IASTexpression expr : iast.getExpressions() ) {
        result = expr.accept(this, result);
    }
    return result;
}


@Override
    public Set<IASTCglobalVariable> visit(
        IASTCglobalVariable iast,
        Set<IASTCglobalVariable> result)
                throws CompilationException {
    result.add(iast);
    return result;
}

@Override
    public Set<IASTCglobalVariable> visit(
        IASTClocalVariable iast,
        Set<IASTCglobalVariable> result)
                throws CompilationException {
    return result;
}
```

Advantages des visiteurs

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

Interpreter

```
package com.paracamplus.ilp1.interpreter;

public class Interpreter
implements IASTvisitor<Object, ILexicalEnvironment, EvaluationException> {

    public Interpreter (IGlobalVariableEnvironment globalVariableEnvironment,
                        IOperatorEnvironment operatorEnvironment ) {
        this.globalVariableEnvironment = globalVariableEnvironment;
        this.operatorEnvironment = operatorEnvironment;
    }
    protected IGlobalVariableEnvironment globalVariableEnvironment;
    protected IOperatorEnvironment operatorEnvironment;

    public IOperatorEnvironment getOperatorEnvironment() {
        return operatorEnvironment;
    }

    public IGlobalVariableEnvironment getGlobalVariableEnvironment() {
        return globalVariableEnvironment;
    }

    //

    public Object visit(IASTprogram iast, ILexicalEnvironment lexenv)
            throws EvaluationException {
        try {
            return iast.getBody().accept(this, lexenv);
        } catch (Exception exc) {
            return exc;
        }
    }
```

Alternative

```
public Object visit(IASTalternative iast,
                    ILexicalEnvironment lexenv)
          throws EvaluationException {
Object c = iast.getCondition().accept(this, lexenv);
if ( c != null && c instanceof Boolean ){
    Boolean b = (Boolean) c;
    if ( b.booleanValue() ) {
          return iast.getConsequence().accept(this, lexenv);
    } else if ( iast.isTernary() ) {
          return iast.getAlternant().accept(this, lexenv);
    } else {
          return whatever;
    }
} else {
          return iast.getConsequence().accept(this, lexenv);
      }
  }
```

Séquence

```
public Object visit(IASTsequence iast,
                    ILexicalEnvironment lexenv)
             throws EvaluationException {
IASTexpression[] expressions = iast.getExpressions();
Object lastValue = null;
for ( IASTexpression e : expressions ) {
             lastValue = e.accept(this, lexenv);
         }
return lastValue;
}
```

Block

```
public Object visit(IASTblock iast,
                    ILexicalEnvironment lexenv)
             throws EvaluationException {
ILexicalEnvironment lexenv2 = lexenv;
for ( IASTbinding binding : iast.getBindings() ) {
  Object initialisation =
    binding.getInitialisation().accept(this, lexenv);
  lexenv2 = lexenv2.extend(binding.getVariable(),
                                  initialisation);
}
return iast.getBody().accept(this, lexenv2);
}
```

Constante

Toutes les constantes ont une valeur décrite par une chaîne.

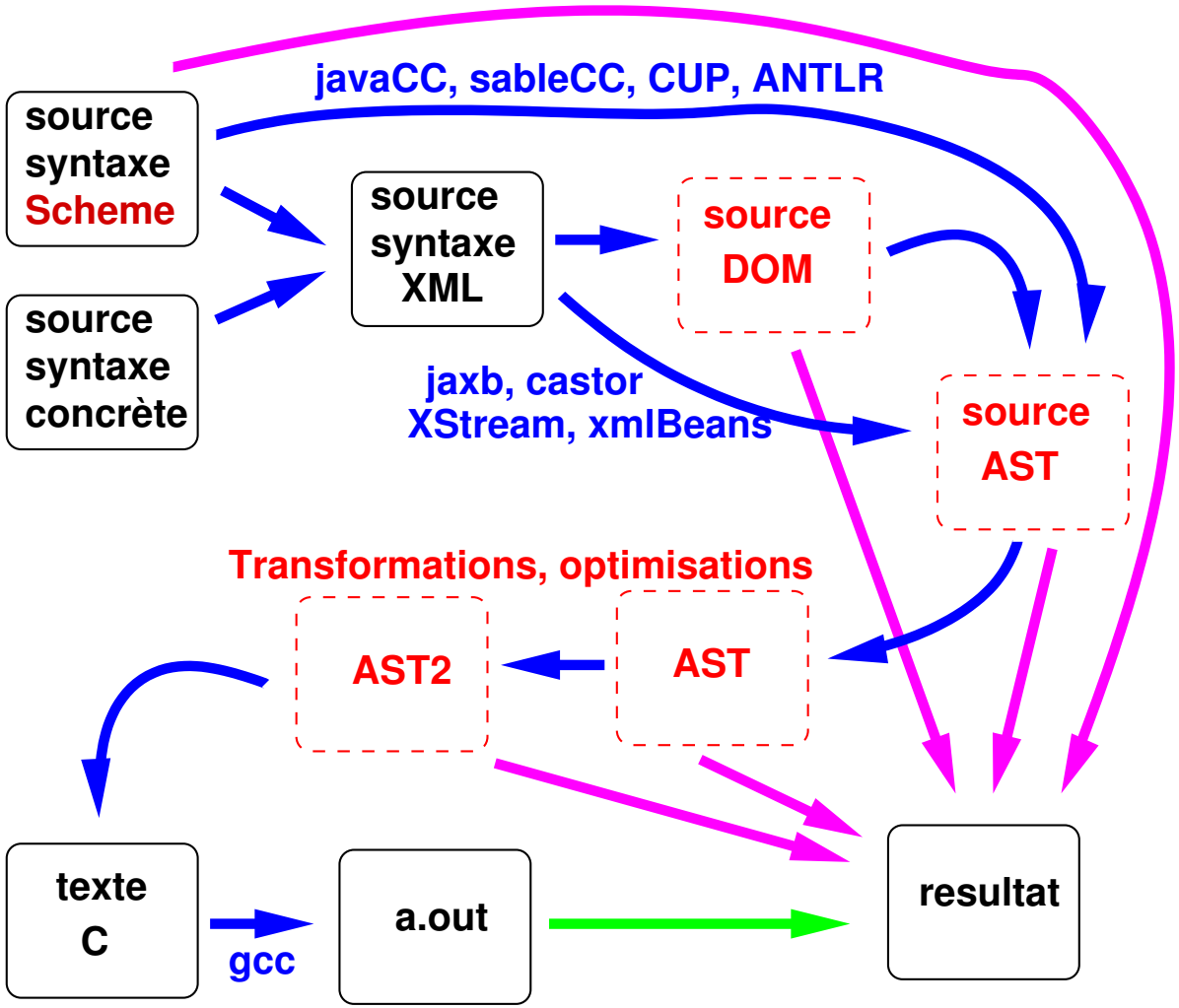
```
public class ASTinteger extends ASTconstant implements IASTinteger {

    public ASTinteger (String description) {
        super(description, new BigInteger(description));
    }
    @Override
        public BigInteger getValue() {
        return (BigInteger) super.getValue();
    }

    @Override
        public <Result, Data, Anomaly extends Throwable>
    Result accept(IASTvisitor<Result, Data, Anomaly> visitor, Data data)
            throws Anomaly {
        return visitor.visit(this, data);
    }
}

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

Variable

```
public Object visit(IASTvariable iast,
        ILexicalEnvironment lexenv)
            throws EvaluationException {
try {
        return lexenv.getValue(iast);
     }
catch (EvaluationException exc) {
        return getGlobalVariableEnvironment()
        .getGlobalVariableValue(iast.getName());
 }
}
```

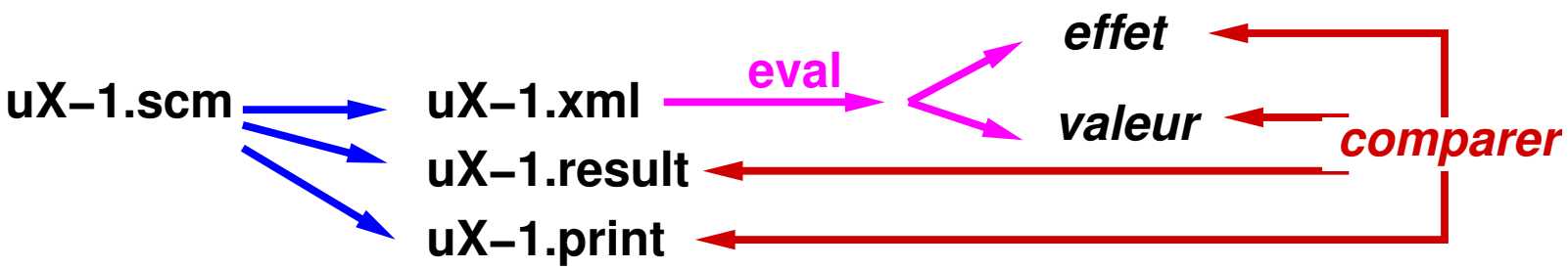
```
public Object visit(IASTinvocation iast,
                    ILexicalEnvironment lexenv)
             throws EvaluationException {
Object function =
        iast.getFunction().accept(this, lexenv);
if ( function instanceof Invocable ) {
        Invocable f = (Invocable)function;
        List<Object> args = new Vector<Object>();
        for ( IASTexpression arg : iast.getArguments() )
                Object value = arg.accept(this, lexenv);
                args.add(value);
            }
        return f.apply(this, args.toArray());
 } else {
        String msg = "Cannot apply " + function;
        throw new EvaluationException(msg);
        }
}
```

Grand schéma



Batterie de tests

Ressource: Samples/u*-1.*



Ressource: com.paracamplus.ilp1.interpreter.test.InterpreterTest.java

Tests

Tests avec JUnit3 Cf. `http://www.junit.org/`

```
package com.paracamplus.ilp1.tools.test;

import junit.framework.TestCase;

import com.paracamplus.ilp1.tools.ProgramCaller;

public class ProgramCallerTest extends TestCase {

    public void testProgramCallerInexistentVerbose () {
        final String programName = "lasdljsdfousadfl lsjd";
        ProgramCaller pc =
                          new ProgramCaller(programName);
        assertNotNull(pc);
        pc.setVerbose();
        pc.run();
        assertTrue(pc.getExitValue() != 0);
    }
```

Séquencement JUnit3

Pour une classe de tests `SomeTest` :

1. charger la classe de test `SomeTest`
2. pour chaque méthode nommée `test`X,
   1. instancier la classe de test `SomeTest`
   2. tourner `setUp()`
   3. tourner `test`X
   4. tourner `tearDown()`

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
```

Séquencement JUnit4

Pour une classe de tests `Foobar` :

1. charger la classe `Foobar`
2. tourner toutes les methodes `@BeforeClass`
3. pour chaque méthode annotée `@Test`,
   1. instancier la classe `Foobar`
   2. tourner toutes les méthodes `@Before`
   3. tourner la méthode testée
   4. tourner toutes les méthodes `@After`
4. enfin, tourner toutes les methodes `@AfterClass`

Annotations

Les annotations sont des métadonnées dans le code source

- originalement en JAVA avec Javadoc
- annotations connues : @Deprecated, @Override, ...
- annotations multi paramétrées : @Annotation(arg1="val1", arg2="val2", ...)

Utilisations des annotations :

- par le compilateur pour détecter des erreurs
- pour la documentation
- pour la génération de code
- pour la génération de fichiers

Avec les annotations le code source est parcouru mais il n'est pas modifié.

Définition d'une annotation

```
public @interface MyAnnotation {
int arg1() default 4;
String arg2();
}
```

```
@MyAnnotation(arg1=0, arg2="valeur2")
public class UneCLasse {
...
}
```

Les annotations des annotations

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface MyAnnotation {
int arg1() default 4;
String arg2();
}
```

Pour l'annotation @Retention :

- RetentionPolicy.SOURCE : dans le code source uniquement (ignorée par le compilateur)
- RetentionPolicy.CLASS : dans le code source et le bytecode (fichier .java et .class)
- RetentionPolicy.RUNTIME : dans le code source et le bytecode et pendant l'exécution par introspection

Les annotations pendant l'exécution

La pluspart des méta-objets implémentent `java.lang.reflect.AnnotatedElement` :

- `boolean isAnnotationPresent(Class<? extends Annotation>)` :
  True si le méta-objet est annoté avec le type du paramètre
- `<T extends Annotation> getAnnotation(Class<T>)` :
  renvoie l'annotation de type T ou null
- `Annotation[] getAnnotations()` :
  renvoie la liste des annotations
- `Annotation[] getDeclaredAnnotations()` :
  renvoie la liste des annotations directes (pas les heritées)

Récapitulation

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