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/`

Buts

- Implanter un langage
- de la classe de Javascript
- à syntaxe XML
- avec un interprète écrit en Java
- et un compilateur écrit en Java
- qui produit du C.

Buts pédagogiques

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

Cheminement

Carlos Agon

- (cours 1-4) Syntaxe, interprétation, compilation vers C (ILP1)
  - DOM, XML, RelaxNG, JUnit3, JUnit4, visiteur
- (cours 5) Fonctions, affectation, boucle (ILP2)
  - Généricité, analyse statique

Antoine Miné

- (cours 6) Exceptions (ILP3)
  - C longjmp/setjmp
- (cours 7) Lambda expressions (ILP3)
  - Définition, interprétation, compilation
- (cours 8-9) Classe, objet, appel de méthode (ILP4)
- (cours 10) Édition de liens (ILP4)

Évaluation

- Première évaluation (ILP1 et ILP2, cours 1-5) 40%
- Deuxième évaluation (ILP1 - ILP4) 60%

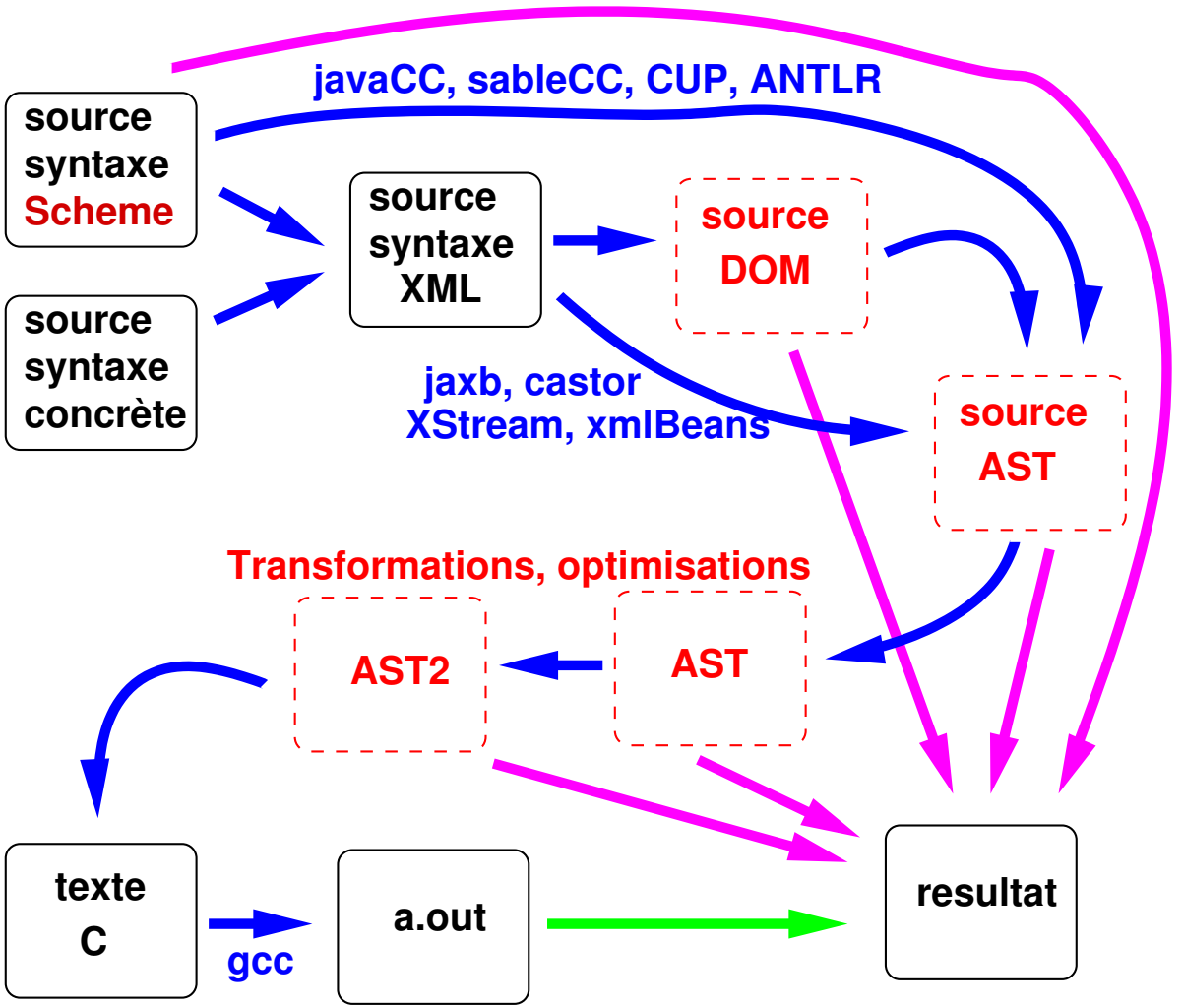
TME hebdomadaire

- (groupe 1 vendredi) Carlos Agon/Antoine Miné,
- (groupe 2 jeudi) Binh Minh Bui Xuan,
- (groupe 3 jeudi) Philippe Esling.

Plan du cours 1

- Grand schéma
- Langage ILP1
  - sémantique discursive
  - syntaxes
- XML
  - DOM
  - grammaires : Relax NG
- DOM et IAST
- AST
- Tests JUnit3 et JUnit4

Grand schéma



ILP1

- Ce qu'il y a :
  - constantes (entière, flottante, chaîne de caractères, booléens)
  - alternative, séquence
  - variable, bloc local n-aire
  - invocation de fonctions
  - opérateurs unaires ou binaires
- ce qu'il n'y a pas (encore !) (liste non exhaustive) :
  - pas d'affectation
  - pas de boucle
  - pas de définition de fonction
  - pas d'exception
  - pas des lambda
  - pas de classe

Les composants d'un langage

- Des principes de calcul et d'architecture calcul : impératif, fonctionnel, logique, etc. architecture : modularité, polymorphisme, etc.
- Lexique + syntaxe
- Un système de types (pas tous)
- Une sémantique
- Un environnement de programmation compilateurs, débogueurs, manuels,
- Une communauté d'utilisateurs

Syntaxe Python-Caml

```
let x = 111 in
  let y = true in
    if ( y == -3.14 )
    then print "O" + "K"
    else print (x * 6.0)
    endif
  newline
```

Syntaxe Scheme

```
(let ((x 111))
  (let ((y #t))
    (if (= y -3.14)
        (print (plus "O" "K"))
        (print (* x 6.0)) ) )
  (newline) )
```

Syntaxe C/Java

```
{ int x = 111;
  { boolean y = true;
    if ( y == -3.14 ) {
        print("O" + "K");
    } else {
        print(x * 6.0);
    }
  }
  newline();
}
```

Définition d'un langage

Définitions :

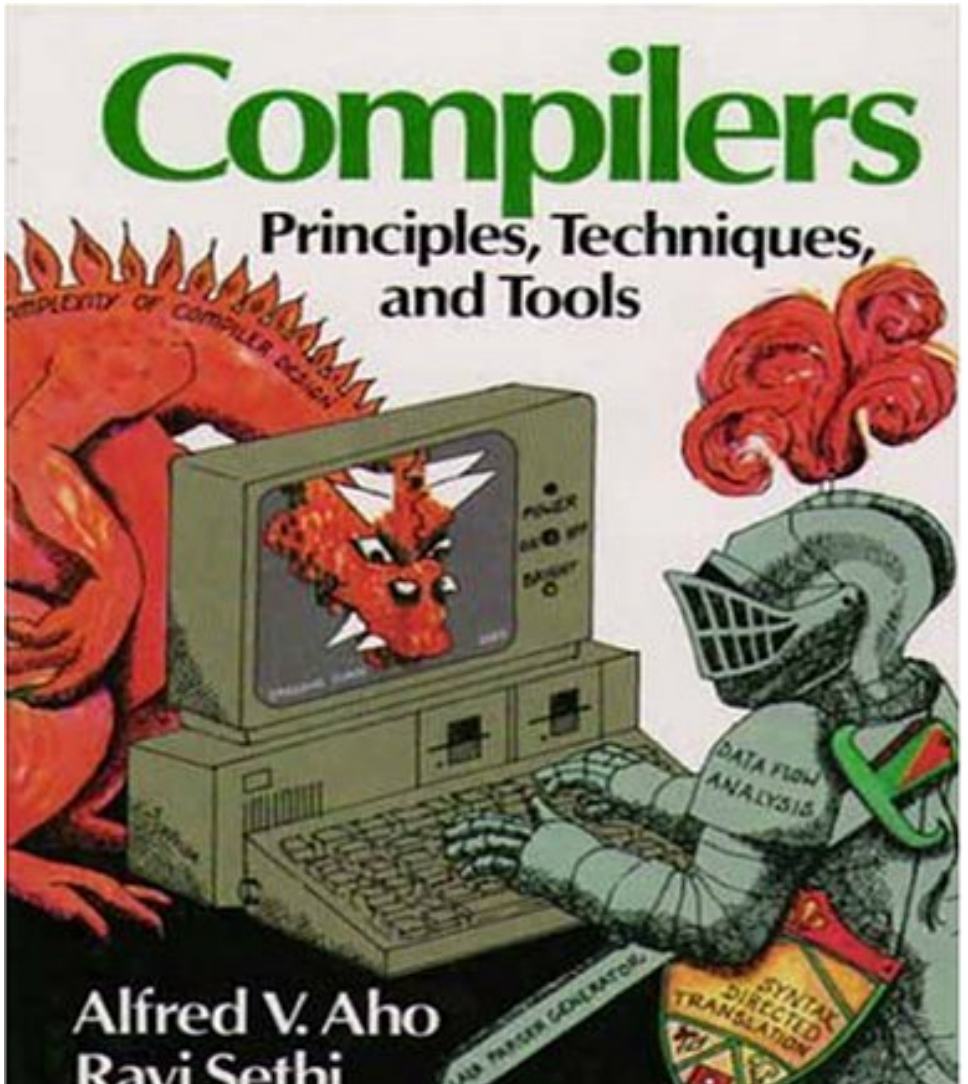
- Symbole : signe primitif (lettre, chiffre, point, ligne, ?)
- Alphabet : ensemble fini de symboles (Σ)
- Chaîne : séquence finie de symboles (la chaîne vide ϵ)
- Langages formel : ensemble de chaînes définies sur un alphabet Σ

Problèmes :

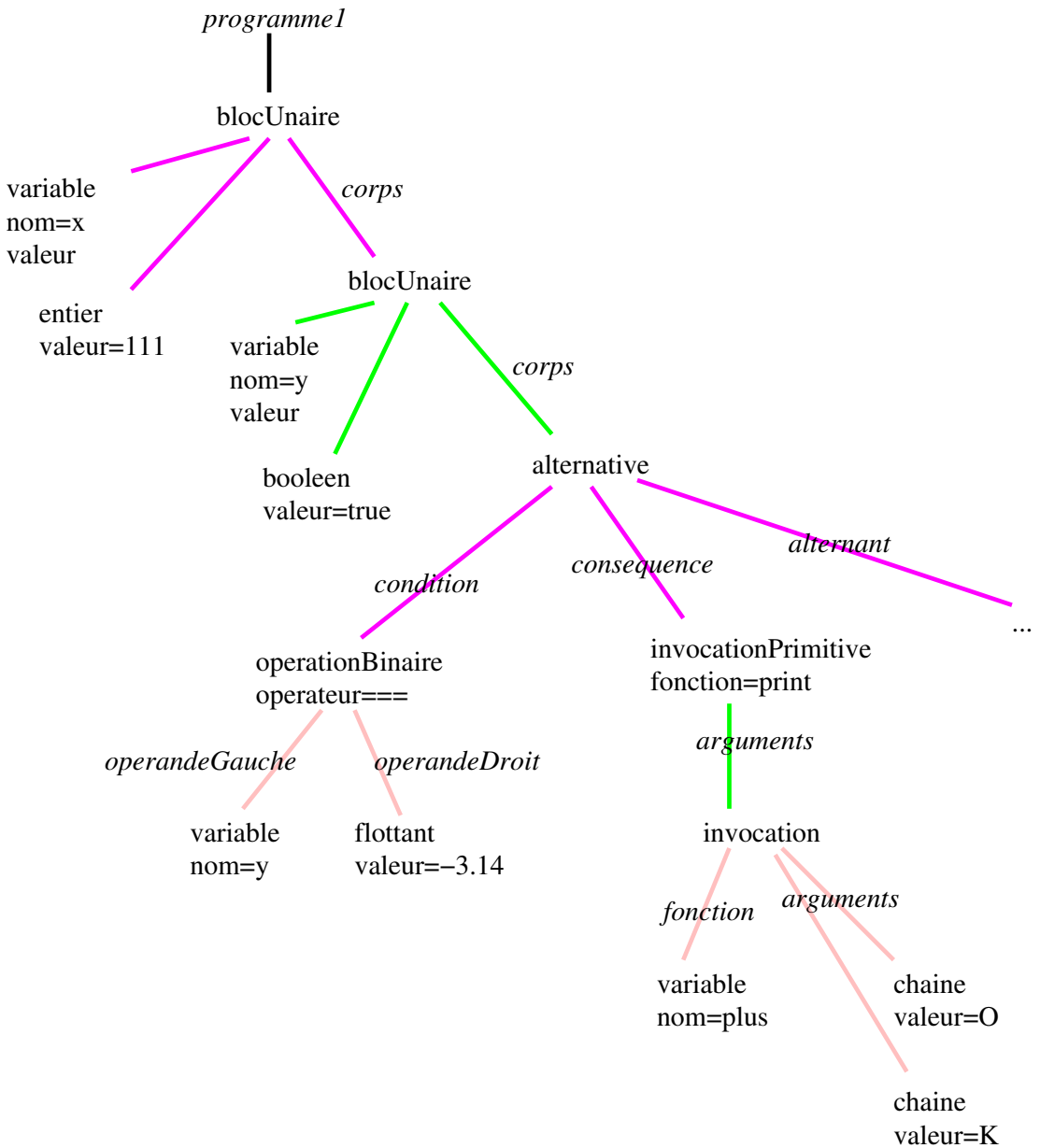
- Comment définir un langage en intension ?
- Comment calculer si une chaîne ω appartient ou non à un langage L ?

The dragon book

C'est important,mais, on ne s'y intéressera pas ! On partira donc d'une syntaxe XML.



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>
```

Rudiments d'XML

Un langage normalisé pour représenter des arbres.

```
<?xml version="1.0"
      encoding='ISO-8859-1'
      standalone="yes"
?>
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
<ul><li> un   </li>
    <!-- attention: -->
    <li rien="du tout"/>
</ul>
```

Attention à UTF-8, ISO-8859-1 (latin1) ou ISO-8859-15 (latin9).

Terminologie

- Un **élément** débute par le < de la balise ouvrante et se termine avec le > de la balise fermante correspondante.
  Un élément contient au moins une balise mais peut contenir d'autres éléments, du texte, des commentaires (et des références à des entités éventuellement des instructions de mise en œuvre).
- Une **balise** (pour *tag*) débute par un < et s'achève au premier > qui suit. Une balise possède un nom et, possiblement, des attributs.
  Les noms des balises sont structurés par espaces de noms (par exemple `xml:namespace` ou `rdf:RDF`).

Caractères bizarres en XML

Entités prédéfinies ou sections particulières (échappements) :

```
&amp; &lt; &gt; &apos; &quot;

<?xml version="1.0" encoding="ISO-8859-15" standalone="y
<ul><li/>
    <li><![CDATA[ 1li>1m ?
<!@#~*}  ]]></li>
    <li> 1li&gt;1m ?
&lt;!@#~*}  </li>
</ul>
```

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.

Backus Naur Form

BNF

- méta-symboles :
  ::=, (,), |
- non terminaux :
  <nom>
- terminaux :
  «mots»

Exemple

<terme> ::= <nombre signé> | <nombre signé> <op> <terme>
<nombre signé> ::= <nombre> | «+» <nombre> | «-» <nombre>
<nombre> ::= <entier> | <nombre fractionnaire>
<nombre fractionnaire> ::= <entier> | <entiere> «/» <entiere>
<entier> ::= <chiffre> | <chiffre><entier>
<chiffre> ::= «0» | «1» | ... | «9»

Exemple

<terme> ::= <nombre signé> | <nombre signé> <op> <terme>
<nombre signé> ::= <nombre> | «+» <nombre> | «-» <nombre>
<nombre> ::= <entier> | <nombre fractionnaire>
<nombre fractionnaire> ::= <entier> | <entiere> «/» <entiere>
<entier> ::= <chiffre> | <chiffre><entier>
<chiffre> ::= «0» | «1» | ... | «9»

1 + 3 - 1/4

1 + 3 * 1/4

1 + 3 * 1/0

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*).

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 }
```

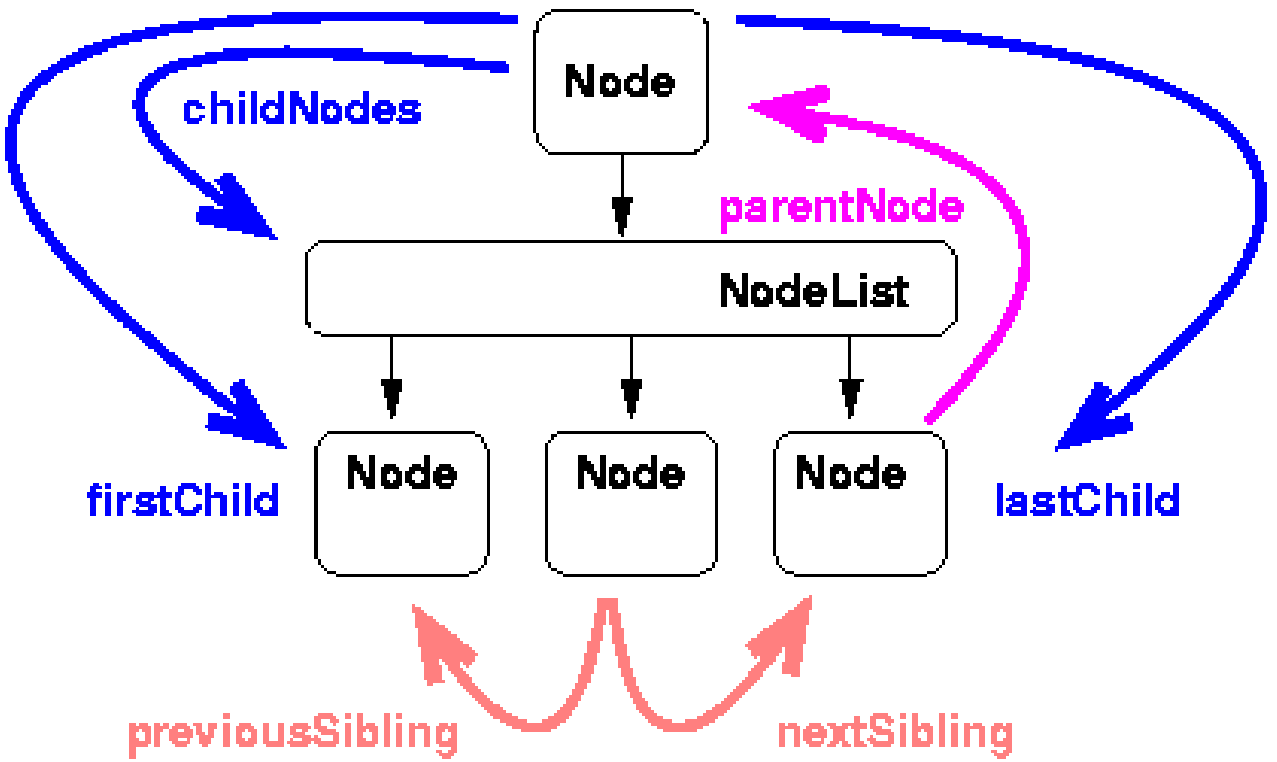
Validation

```
<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;`

```
public IASTprogram getProgram() throws ParseException {
try {
 final String programText = input.getText();
 final String rngFilePath = rngFile.getAbsolutePath();
 final InputSource isg = ValidationDriver.fileInputSource(rngFilePath);
 final ValidationDriver vd = new ValidationDriver();
 vd.loadSchema(isg);
 InputSource is = new InputSource(new StringReader(programText));
 if ( ! vd.validate(is) ) {
     throw new ParseException("Invalid XML program!");
 }
     ...
 }
```

Conversion à DOM

```
public IASTprogram getProgram() throws ParseException {
    try {
        ...
        final DocumentBuilderFactory dbf =
            DocumentBuilderFactory.newInstance();
        final DocumentBuilder db = dbf.newDocumentBuilder();
        // the previous value of is is totally drained!
        is = new InputSource(new StringReader(programText));
        final Document document = db.parse(is);
       ...
    } catch (ParseException e) {
        throw e;
    } catch (Exception e) {
        throw new ParseException(e);
    }
}
```

Les interfaces du DOM

- org.w3c.dom.Document
- org.w3c.dom.Node
  - org.w3c.dom.Element
  - org.w3c.dom.CharacterData
- org.w3c.dom.NodeList

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);
```

Variations autour du DOM

- attention au découpage des textes, aux blancs superflus,

```
<a b='c'
></a>
```

```
<a b='c'>
</a>
```

- attention à la présence de commentaires, instructions de traitement

```
<a>
 <b> text1
 <!-- comment -->
text2
 </b>
</a>
```

Du bon emploi de XML

XML favorise l'auto-description avec des balises signifiantes
Ne pas employer d'attributs là où une certaine extensibilité est envisagée ou là où ne peut être contenue l'information souhaitée.
XML cherche à minimiser les efforts d'analyse (d'où l'intérêt des conteneurs)

```
<point>
  <x>1</x>
  <y>2</y>
</point>
```

```
<point x='1' y='2'/>
```

```
<ligne>
  <characteristics
    color='blue' />
  <points>
    <point>..</point>
    <point>..</point>
  </points>
</ligne>
```

```
<ligne>
 <point>..</point>
 <characteristics ../>
 <point>..</point>
 ..
</ligne>
```

Sémantique discursive

- Langage non typé statiquement : les variables n'ont pas de type
- Langage sûr, typé dynamiquement : toute valeur a un type (donc de la classe de Scheme, Javascript, Smalltalk)
- Langage à instruction (séquence, alternative, bloc unaire)
- Toute expression est une instruction
- Les expressions sont des constantes, des variables, des opérations ou des appels de fonctions (des invocations).

Détails sémantiques

Opérateurs unaires : `-` (opposé) et `!` (négation)
Opérateurs binaires :

- arithmétiques : `+` (sur nombres et chaînes), `-`, `*`, `/`, `%` (sur entiers),
- comparateurs arithmétiques : `<`, `<=`, `>`, `>=`,
- comparateurs généraux : `==`, `<>`, `!=` (autre graphie),
- booléens : `|`, `&`, `^`.

Variables globales prédéfinies : fonctions primitives : `print` et `newline` (leur résultat est indéfini) ou constantes comme `pi`.
Le nom d'une variable ne peut débuter par `ilp` ou `ILP`.
L'alternative est binaire ou ternaire (l'alternant est facultatif).
La séquence contient au moins un terme.

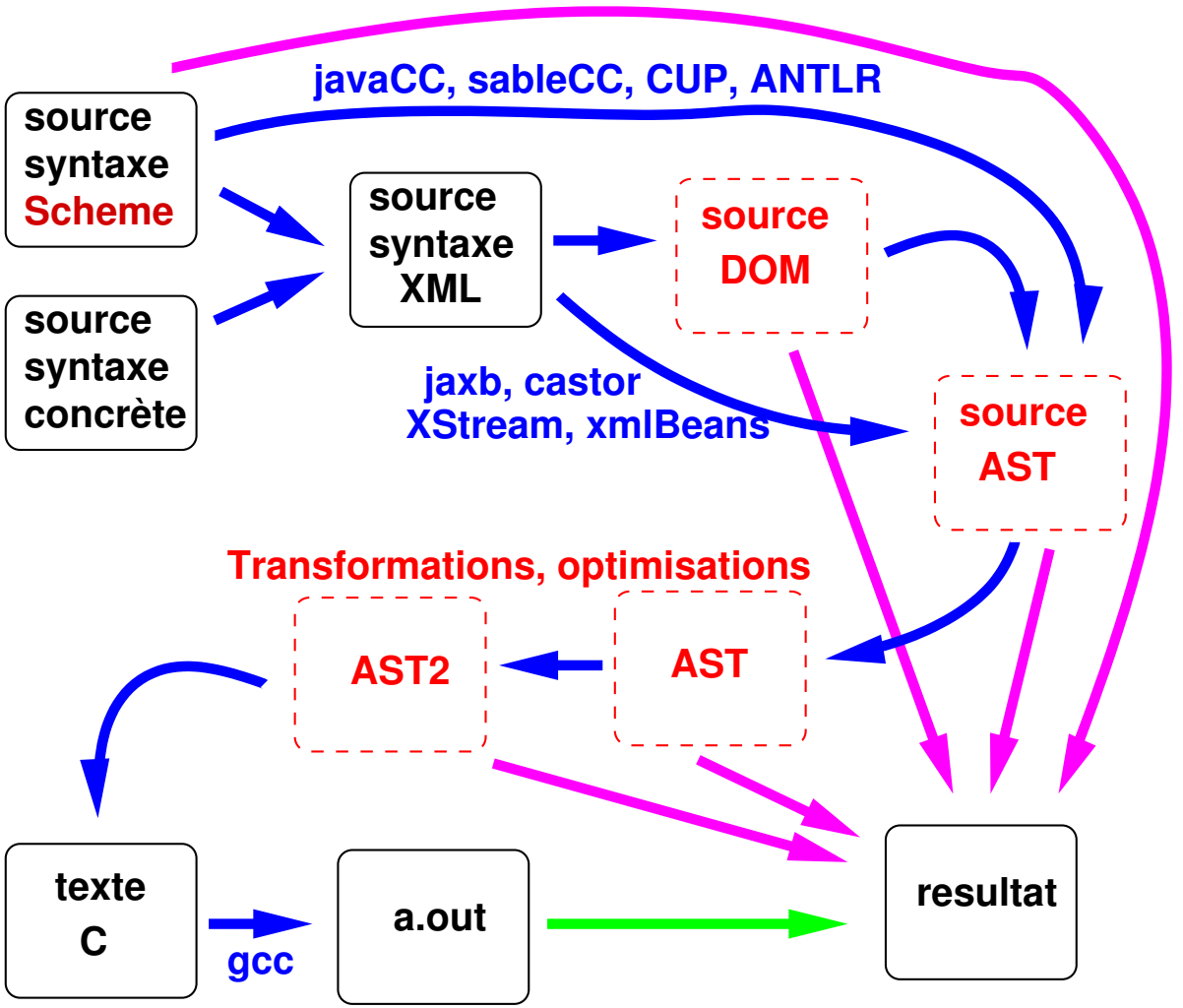
AST

DOM est une façon simple de réifier un document XML (quelques lignes de programme), mais il est peu adapté à la manipulation d'arbres de syntaxe AST (pour *Abstract Syntax Tree*) car il est non typé, trop coûteux, mal extensible.

Une fois le DOM obtenu, on le transforme en un AST.

NOTA : on aurait pu passer directement de la syntaxe concrète à l'AST.

Grand schéma



IAST

Le paquetage `com.paracamplus.ilp1.interfaces` fournit une interface pour chaque concept syntaxique :

```
IAST
  IASTalternative
  IASTconstant
    IASTboolean
    IASTinteger
    IASTfloat
    IASTstring
  IASTinvocation
  IASToperation
    IASTunaryOperation
    IASTbinaryOperation
  IASTsequence
  IASTBlock
  IASTvariable
  IASTprogram
```

Alternative

D'un point de vue syntaxique, une alternative est une entité ayant trois composants dont un optionnel :

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

IASTalternative

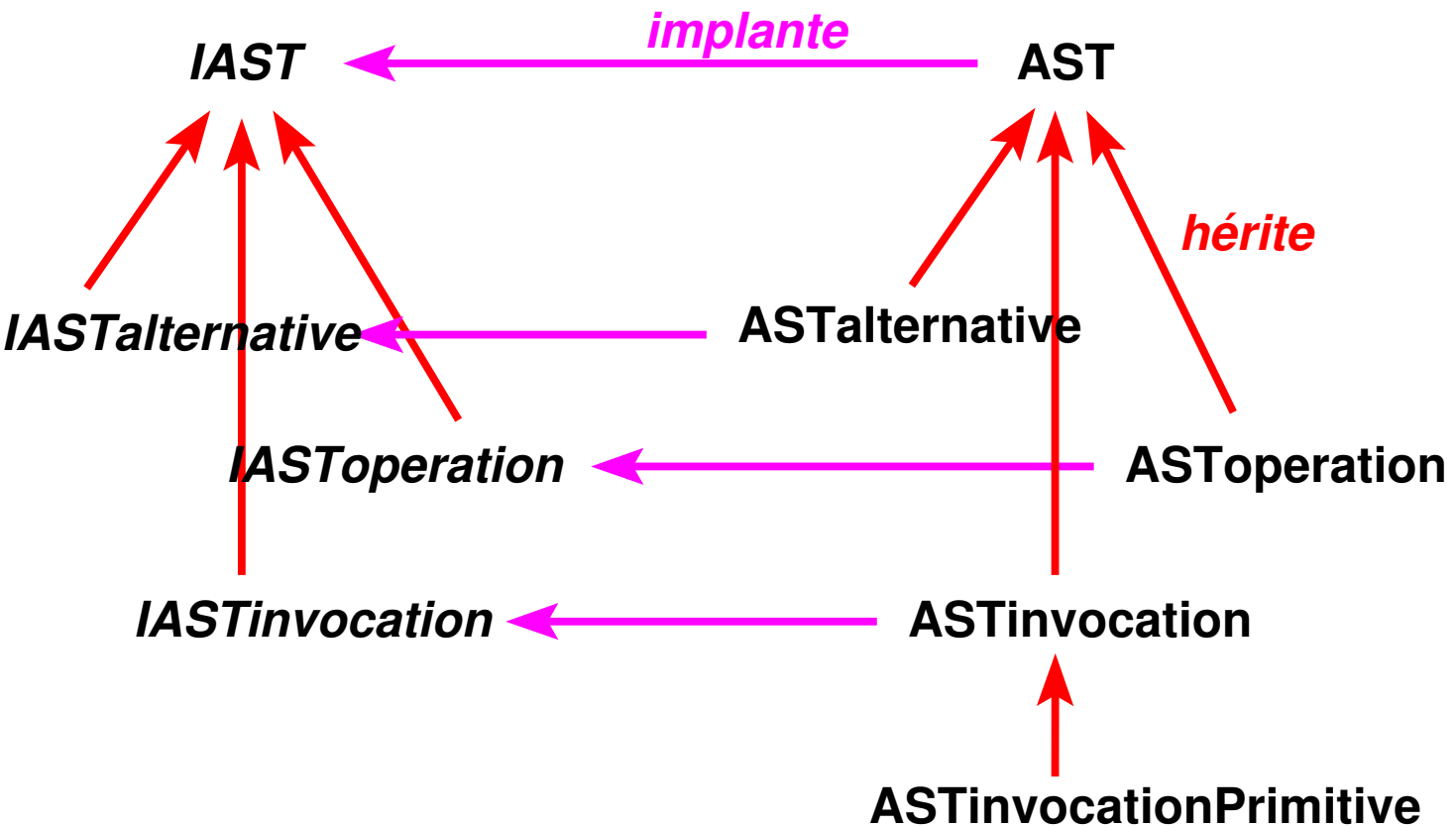
```
package com.paracamplus.ilp1.interfaces;

public interface IASTalternative
extends IAST {

        IASTexpression getCondition();
        IASTexpression getConsequence();
        @OrNull IASTexpression getAlternant();

  /** Indique si l'alternative est ternaire. */
        boolean isTernary ();
}
```

Interfaces décalquables de la grammaire !

Pourquoi ?

Pour dissocier les interfaces des implantations afin de permettre d'autres analyseurs syntaxiques

Hiérarchies



AST

Les classes dans le package com.paracamplus.ilp1.ast implantent les interfaces respectives dans com.paracamplus.ilp1.interfaces.

```
AST                  implante IAST
  ASTalternative     implante IASTalternative
  ASTBlock     implante IASTBlock
  ASTboolean         implante IASTboolean
  ASTstring          implante IASTstring
  ASTinteger         implante IASTinteger
  ASTfloat           implante IASTfloat
  ASTinvocation      implante IASTinvocation
  ASToperation       implante IASToperation
    ASTunaryOperation      implante IASTunaryOperation
    ASTbinaryOperation     implante IASTbinaryOperation
  ASTsequence        implante IASTsequence
  ASTvariable        implante IASTvariable
```

ASTalternative

```
package com.paracamplus.ilp1.ast;

import com.paracamplus.ilp1.annotation.OrNull;
import com.paracamplus.ilp1.interfaces.IASTalternative;
import com.paracamplus.ilp1.interfaces.IASTexpression;

public class ASTalternative extends ASTexpression
implements IASTalternative {

        public ASTalternative(IASTexpression condition,
                              IASTexpression consequence,
                              IASTexpression alternant ) {
                this.condition = condition;
                this.consequence = consequence;
                this.alternant = alternant;
        }
```

```
private final IASTexpression condition;
private final IASTexpression consequence;
private @OrNull final IASTexpression alterna

public IASTexpression getCondition() {
        return condition;
}

public IASTexpression getConsequence() {
        return consequence;
}

public IASTexpression getAlternant() {
        return alternant;
}

public boolean isTernary () {
    return this.alternant != null;
}
```

Conversion DOM vers AST

```
public IASTprogram getProgram()
    throws ParseException {
    try {
        ... Validation
         }
        ... XML to DOM
         final Document document = db.parse(is);
        IASTprogram program = parse(document);
         return program;
    } catch (ParseException e) {
         throw e;
    } catch (Exception e) {
         throw new ParseException(e);
    }
}
```

La classe Parser

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

```
package com.paracamplus.ilp1.ast;

public class Parser extends AbstractExtensibleParser {

        public Parser(IParserFactory factory) {
                super(factory);
        addMethod("alternative", Parser.class);
        addMethod("sequence", Parser.class);
        addMethod("integerConstant", Parser.class, "integer");
        addMethod("floatConstant", Parser.class, "float");
        addMethod("stringConstant", Parser.class, "string");
        addMethod("booleanConstant", Parser.class, "boolean");
        addMethod("unaryOperation", Parser.class);
        addMethod("binaryOperation", Parser.class);
        addMethod("block", Parser.class);
        addMethod("binding", Parser.class);
        addMethod("variable", Parser.class);
        addMethod("invocation", Parser.class);
        }
```

```
public abstract class AbstractExtensibleParser extends AbstractParser

        public AbstractExtensibleParser(IParserFactory factory) {
                 super(factory);
        this.parsers = new HashMap<>();
        }
    private final HashMap<String, Method> parsers;

    public void addParser (String name, Method method) {
        this.parsers.put(name, method);
    }

    public void addMethod (String name, Class<?> clazz) {
        addParser(name, findMethod(name, clazz));
    }

    public void addMethod (String name, Class<?> clazz,
                                        String tagName) {
        addParser(tagName, findMethod(name, clazz));
    }
```

```
public Method findMethod (String name, Class<?> clazz) {
        try {
          for ( Method m : clazz.getMethods() ) {
              if ( ! name.equals(m.getName()) ) {
                  continue;
              }
              if ( Modifier.isStatic(m.getModifiers()) ) {
                  continue;
              }
              Class<?>[] parameterTypes = m.getParameterTypes();
              if ( parameterTypes.length != 1 ) {
                  continue;
              }
              if ( ! Element.class.isAssignableFrom(parameterTypes[0]
                  continue;
              }
              return m;
          }
          if ( Object.class == clazz ) {
              final String msg = "Cannot find suitable parsing method
              throw new RuntimeException(msg);
          } else {
              return findMethod(name, clazz.getSuperclass());
          }
        } catch (SecurityException e1) {
          final String msg = "Cannot access parsing method!";
          throw new RuntimeException(msg);}}
```

```
public IAST parse (final Node n) throws ParseException {
      switch ( n.getNodeType() ) {
      case Node.ELEMENT_NODE: {
        final Element e = (Element) n;
        final String name = e.getTagName();
       if ( parsers.containsKey(name) ) {
            final Method method = parsers.get(name);
            try {
              Object result = method.invoke(this, new Object[]{e});
              if ( result != null && result instanceof IAST ) {
                  return (IAST) result;
              } else {
                  final String msg = "Not an IAST " + result;
                  throw new ParseException(msg);
              }
            } catch (IllegalArgumentException exc) { ...
            } catch (IllegalAccessException exc) { ...
            } catch (InvocationTargetException exc) { ...
            }

        } else {
          final String msg = "Unknown element name: " + name;
          throw new ParseException(msg);
        }
      }
      default: {
        final String msg = "Unknown node type: "
```

Méthode pour l'alternative

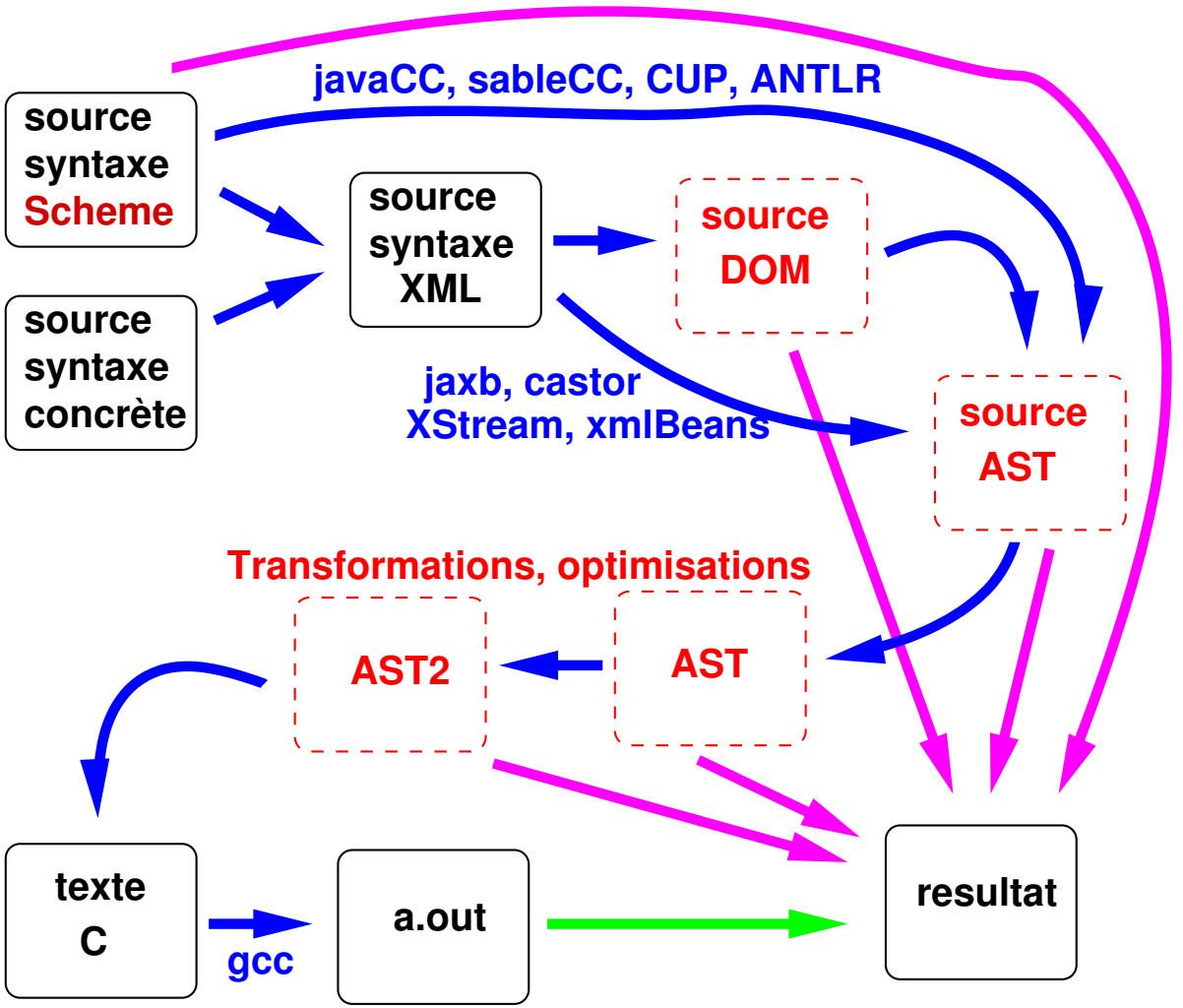
```
public IASTexpression alternative (Element e) throws ParseException {
   IAST iastc = findThenParseChildContent(e, "condition");
   IASTexpression condition = narrowToIASTexpression(iastc);
   IASTexpression[] iaste =
      findThenParseChildAsExpressions(e, "consequence");
   IASTexpression consequence = 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);
      }
    }
```

Récapitulation

- syntaxe d'ILP1 (grammaire RelaxNG, XML, IAST)
- représentation d'un programme ILP1 (AST)

- RelaxNG, DOM, XML

Grand schéma



Bibliographie

- Cours de C `http://www-ari.ufr-info-p6.jussieu.fr/RESSOURCES/doc/cederoms/Videoc2000/`
- Cours de Java `http://www.pps.jussieu.fr/~emmanuel/Public/enseignement/JAVA/SJP.pdf`
- developper en java avec Eclipse `http://www.jmdoudoux.fr/java/dejae/` (500 pages)
- Cours sur XML `http://apiacoa.free.fr/teaching/xml/`
- RelaxNG `http://www.oasis-open.org/committees/relax-ng/tutorial.html` ou le livre « Relax NG » d'Éric Van der Vlist, O'Reilly 2003.

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)