Revision: 1.21

Master d'informatique 2009-2010
Spécialité STL
« Implantation de langages »
ILP – MI016
épisode ILP1

C.Queinnec

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.

Compétences visées :

- Mise en place de divers concepts
- Commentaires autour de la conception de langages
- Transformation de langage (descripteur vers code)

Autres buts

- Faire lire du code
- montrer comment il a été développé (écrit, testé, étendu)
- et quelques outils au passage : Eclipse, PHP, XML, DOM, RelaxNG, XPath, Java6, JUnit3,4, XMLUnit ...

Une contrainte supplémentaire :

- Aucune modification de code publié possible !

Cheminement

- (ILP1 : cours 1-4) Syntaxe, interprétation, compilation vers C
  - DOM, XML, RelaxNG, JUnit3, JUnit4, visiteur
- (ILP2 : cours 5) Bloc local, fonctions, affectation, boucle
  - Généricité, analyse statique
- (ILP3 : cours 6) Exceptions
  - C longjmp/setjmp
- (ILP4 : cours 7) Intégration (*inlining*)
  - délégation, réflexion, annotation
- (ILP6 : cours 8-9) Classe, objet, appel de méthode
- (ILP7 : cours 10) Édition de liens

Préalables

- C (pointeurs, transtypage (pour *cast*), macros `cpp`)
- Java 5 (générique, annotation, réflexion, ...)
- compilation

Structure des cours

- Savoir :
  - concepts,
  - sémantique, variantes,
  - choix de réalisation
- Savoir faire :
  - implantation
  - tours de main
  - agencement
  - tests

Cours en amphi, TME double en salle machine.
Pour préparer le TME1 (en ligne), lire documentation sur JUnit et RelaxNG.

Contrôle des connaissances

- Examen réparti de mi-semestre (sur machine) pour 40% de la note globale d'UE (aux alentours du 30 octobre)
- Examen réparti de fin de semestre (sur machine) pour 60% de la globale note d'UE (aux alentours du 16 décembre)
- Examen de rattrapage (sur machine) pour 100% de la note globale d'UE (aux alentours du 20 janvier 2010)

Aux examens, des textes raisonnés, pas du code !

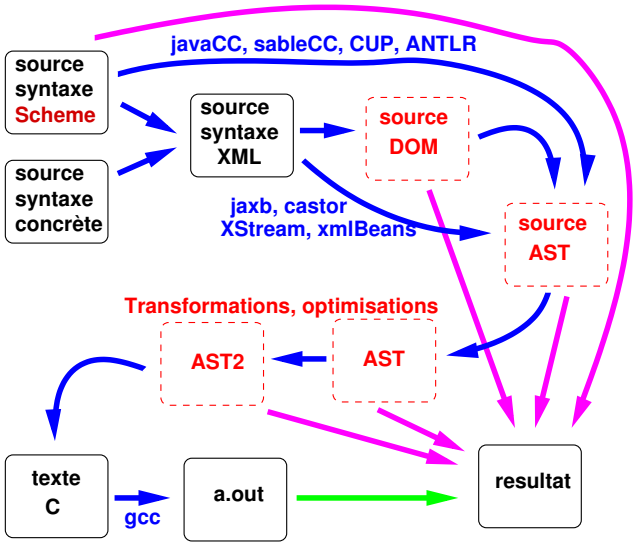
Ressources

- Le site web de master
  - micro-sujets (questions / Eclipse)
- et ses ressources additionnelles
  - transparents
  - code
  - bandes-son
  - annales
  - greffon (notamment un site de mise à jour pour Eclipse)
  - forum (modéré a priori)
- Sur les ordinateurs de l'ARI, le répertoire `/Infos/lmd/2009/master/ue/ilp-2009oct/`

Plan du cours 1

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

Grand schéma



ILP1

- Ce qu'il y a :
  - constantes (entière, flottante, chaîne de caractères, booléens)
  - alternative, séquence
  - variable, bloc local unaire
  - 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 de classe
  - pas d'exception

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
```

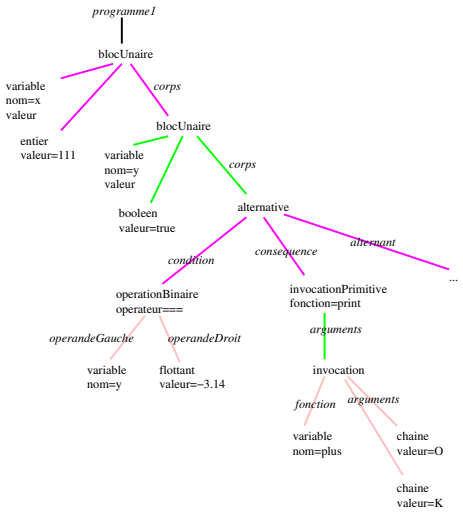
Syntaxe Scheme

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


Syntaxe C/Java

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


Syntaxe arborescente



Syntaxe XML

```
<programme1>
  <blocUnaire>
    <variable nom='x'/><valeur><entier valeur='111'/></valeu
      <corps>
        <blocUnaire>
          <variable nom='y'/><valeur><booleen valeur='true',
          <corps>
            <alternative>
              <condition>
                <operationBinaire operateur='=='>
                <operandeGauche><variable nom='y'/></operan
                <operandeDroit><flottant valeur='-3.14'/></
                </operationBinaire>
              </condition>
              <consequence>
                <invocationPrimitive fonction='print'>
                <operationBinaire operateur='+'>
```

```
            <operandeGauche><chaine>O</chaine></opera
            <operandeDroit><chaine>K</chaine></operan
          </operationBinaire></invocationPrimitive>
        </consequence>
        <alternant>
          <invocationPrimitive fonction='print'>
          <operationBinaire operateur='*'>
            <operandeGauche><variable nom='x'/></oper
            <operandeDroit><entier valeur='6.0'/></op
          </operationBinaire></invocationPrimitive>
        </alternant>
      </alternative>
    </corps>
  </blocUnaire>
  <invocationPrimitive fonction='newline'/>
</corps></blocUnaire></programme1>
```


Syntaxes

La syntaxe n'est rien !

Syntaxes

La syntaxe n'est rien !
La syntaxe est tout !

Syntaxes

La syntaxe n'est rien !
La syntaxe est tout !
On ne s'y intéressera pas ! On partira donc de la syntaxe XML.

Ressource: Grammars/Scheme/Makefile

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.

Rudiments d'XML

Un langage normalisé pour représenter des arbres (cf. mode de visualisation en Eclipse).

```
<?xml version="1.0"
      encoding='ISO-8859-1'
      standalone="yes"
?>
<ul><li> un   </li>
    <!-- attention: -->
    <li rien="du tout"/>
</ul>
```

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

Caractères bizarres en XML

Entités prédéfinies ou sections particulières (échappements) :

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

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

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`).

Validation d'XML

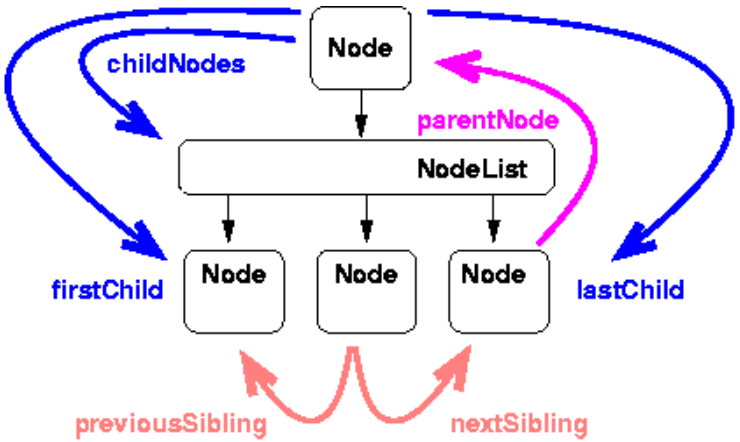
Un document XML doit être *bien formé* c'est-à-dire respectueux des conventions d'XML. Un document XML peut aussi être *valide* vis-à-vis d'une grammaire.
Les grammaires sont des DTD (pour *Document Type Definition*) ou maintenant des XML Schémas ou des schémas Relax NG.
Énorme intérêt pour la lecture de documents car pas de traitement d'erreur à prévoir !
Mais uniquement si les documents sont valides.

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*).

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*)



Interface DOM (2)

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

```
// (1) validation vis-a-vis de RNG:
// MOCHE: c'est redondant avec (2) car le programme est
// relu encore une fois avec SAX. Les phases 1 et 2
// pourraient s'effectuer ensemble.

ValidationDriver vd = new ValidationDriver();
InputSource isg = ValidationDriver.fileInputSource(
      rngfile.getAbsolutePath());
vd.loadSchema(isg);

InputSource isp = ValidationDriver.fileInputSource(
      xmlfile.getAbsolutePath());
if ( ! vd.validate(isp) ) {
    throw new ASTException("programme XML non valide");
};
```



```
// (2) convertir le fichier XML en DOM:
DocumentBuilderFactory dbf =
   DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.parse(xmlfile);
```

Ressource: Java/src/fr/upmc/ilp/ilp1/fromxml/Main.java

Ressource: suites de tests JUnit

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);
```


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 = programme1

programme1 = element programme1 {
   instruction +
}

instruction =
   alternative
 | sequence
 | blocUnaire
 | expression
```

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

sequence = element sequence {
   instruction +
}

blocUnaire = element blocUnaire {
   variable,
   element valeur { expression },
   element corps  { instruction + }
}

expression =
   constante
 | variable
 | operation
 | invocationPrimitive
```



```
variable = element variable {
   attribute nom { xsd:string - ( xsd:string {
       pattern = "(ilp|ILP)" } ) },
   empty
}

invocationPrimitive = element invocationPrimitive {
   attribute fonction { xsd:string },
   expression *
}

operation =
   operationUnaire
 | operationBinaire
```



```
operationUnaire = element operationUnaire {
   attribute operateur { "-" | "!" },
   element operande { expression }
}

operationBinaire = element operationBinaire {
   element operandeGauche { expression },
   attribute operateur {
      "+" | "-" | "*" | "/" | "%" |
# arithmetiques
      "|" | "&" | "^" |
# booleens
      "<" | "<=" | "==" | ">=" | ">" | "<>" | "!="
# comparaisons
   },
   element operandeDroit { expression }
}
```

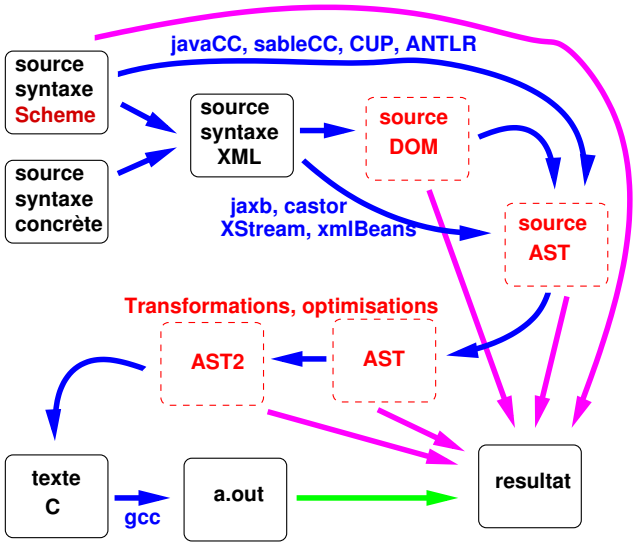


```
constante =
   element entier    {
      attribute valeur { xsd:integer },
      empty }
 | element flottant  {
    attribute valeur { xsd:float },
    empty }
 | element chaine    { text }
 | element booleen   {
    attribute valeur { "true" | "false" },
    empty }
```

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. Comme on souhaite que vous puissiez écrire vos propres syntaxes et les faire interpréter ou compiler par le système, on procure des interfaces pour toutes les concepts syntaxiques.
NOTA : on aurait pu passer directement de la syntaxe concrète à l'AST.

Grand schéma



IAST

Le paquetage `fr/upmc/ilp/ilp1/interfaces` fournit une interface pour chaque concept syntaxique :

```
IAST                        // Un marqueur
  IASTalternative
  IASTconstant
    IASTboolean
    IASTinteger
    IASTfloat
    IASTstring
  IASTinvocation
  IASToperation
    IASTunaryOperation
    IASTbinaryOperation
  IASTsequence
  IASTunaryBlock
  IASTvariable
```

Remarque : on ne peut typer une exception avec une interface (il faut attendre Java 7 mais on peut utiliser des classes génériques).

Ressource: Java/src/fr/upmc/ilp/ilp1/interfaces/

Alternative

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

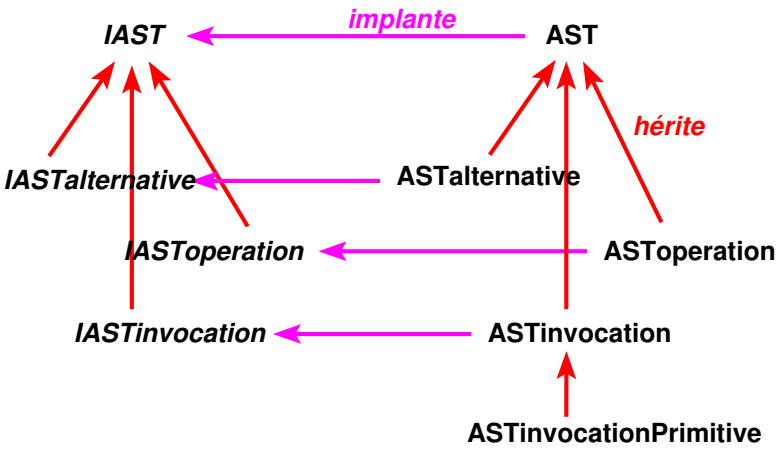
```
alternative = element alternative {
   element condition   { expression },
   element consequence { instruction + },
   element alternant   { instruction + } ?
}
```

IASTalternative

```
package fr.upmc.ilp.ilp1.interfaces;

public interface IASTalternative
extends IAST {

  IAST getCondition ();
  IAST getConsequent ();
  @OrNull IAST getAlternant ();

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

Remarque : on ne peut typer une exception avec une interface.
Les interfaces seront utiles par la suite.

Hiérarchies



AST

Les classes `fr.upmc.ilp.ilp1.fromxml.AST*` implantent les interfaces `fr.upmc.ilp.ilp1.interfaces.IAST*` respectives.

```
AST                 implante IAST
  ASTalternative    implante IASTalternative
  ASTblocUnaire     implante IASTunaryBlock
  ASTbooleen        implante IASTboolean
  ASTchaine         implante IASTstring
  ASTentier         implante IASTinteger
  ASTflottant       implante IASTfloat
  ASTinvocation     implante IASTinvocation
    ASTinvocationPrimitive
  ASToperation      implante IASToperation
    ASToperationUnaire    implante IASTunaryOperation
    ASToperationBinaire   implante IASTbinaryOperation
  ASTsequence       implante IASTsequence
  ASTvariable       implante IASTvariable
```


Autres classes

```
ASTfromXML
ASTParser
ASTException
ASTParserTest       TestCase JUnit
Main
MainTest
MainTestSuite
```

MOCHE ! les noms ne sont pas tout à fait semblables !

ASTalternative

```
package fr.upmc.ilp.ilp1.fromxml;
import fr.upmc.ilp.ilp1.interfaces.*;

public class ASTalternative extends AST
implements IASTalternative {

  public ASTalternative (AST condition,
                         AST consequence,
                         AST alternant ) {
    this.condition   = condition;
    this.consequence = consequence;
    this.alternant   = alternant;
  }
  public ASTalternative (AST condition, AST conseque
    this(condition, consequence, null);
  }
```



```
  private final AST condition;
  private final AST consequence;
  private final AST alternant;

  public IAST getCondition () {
    return this.condition;
  }
  public IAST getConsequent () {
    return this.consequence;
  }
  // Attention aux NullPointerException
  public @OrNull IAST getAlternant () {
    return this.alternant;
  }
```



```
  public boolean isTernary () {
    return alternant != null;
  }
  public String toXML () {
    StringBuffer sb = new StringBuffer();  // Vitesse
    sb.append("<alternative><condition>");
    sb.append(condition.toXML());
    sb.append("</condition><consequence>");
    sb.append(consequence.toXML());
    sb.append("</consequence>");
    if ( isTernary() ) {
      sb.append("<alternant>");
      sb.append(alternant.toXML());
      sb.append("</alternant>");
    };
    sb.append("</alternative>");
    return sb.toString();
  }
}
```

Proverbe : ne jamais laisser fuir les nuls !

Exceptions

```
package fr.upmc.ilp.ilp1.fromxml;

public class ASTException {

  public ASTException (Throwable cause) {
    super(cause);
  }
  public ASTException (String message) {
    super(message);
  }
}
```

Conversion DOM vers AST

La conversion est effectuée par la grande fonction nommée `ASTParser.parse(Node)` que voici :

```
public AST parse (Node n) throws ASTException {
  switch ( n.getNodeType() ) {

  case Node.ELEMENT_NODE: {
    Element e = (Element) n;
    NodeList nl = e.getChildNodes();
    String name = e.getTagName();

    if ( "programme1".equals(name) ) {
      return new ASTsequence(parseList(nl));
```



```
} else if ( "alternative".equals(name) ) {
  AST cond  = findThenParseChild(nl, "condition");
  AST conseq = findThenParseChild(nl, "consequence
  try {
    AST alt = findThenParseChild(nl, "alternant");
    return new ASTalternative(cond, conseq, alt);
  } catch (ASTException exc) {
    return new ASTalternative(cond, conseq);
  }

} else if ( "sequence".equals(name) ) {
  return new ASTsequence(this.parseList(nl));
} else if ( "entier".equals(name) ) {
  return new ASTentier(e.getAttribute("valeur"));
...
```


Parcours de l'AST

`toXML()` est une méthode des `AST` mais pas des `IAST`, il y a une méthode équivalente sur DOM.
Un exemple de mise en œuvre est :

```
...
Document d = db.parse(this.xmlfile);

// (3) conversion vers un AST donc un IAST:
ASTParser ap = new ASTParser();
AST ast = (AST) ap.parse(d);

// (3bis) Impression en XML:
System.out.println(ast.toXML());
```


Architecture

Deux paquetages et quelques archives jar pour l'instant :

```
fr.upmc.ilp.tool1       // quelques utilitaires
fr.upmc.ilp.annotation
fr.upmc.ilp.ilp1.interfaces
fr.upmc.ilp.ilp1.fromxml
trang, jing, junit3, junit4
```

6000 lignes de Java

Ressource: Java/jars/

Ressource: Java/src/

Ressource: Java/bin/

Ressource: Java/doc/

Récapitulation

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

Bibliographie

- Cours de C `http://www.infop6.jussieu.fr/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.

Plan du cours 2

- Tests JUnit3
- Interprétation
- Représentation des concepts
- bibliothèque d'exécution

Tests

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

```
package fr.upmc.ilp.ilp1.fromxml;
import junit.framework.TestCase;

public class MainTest extends TestCase {
  public void processFile (String grammarName,
                           String fileName)
    throws ASTException {
    Main m = new Main(new String[] { // reutilisation Mai
      grammarName, fileName });
    assertTrue(m != null);
    m.run();
    assertEquals(1, 1);
  }
  public void testP1 () throws ASTException {
    processFile("Grammars/grammar1.rng",
                "Grammars/Samples/p1-1.xml");
  }
```

Séquencement JUnit3

Pour une classe de tests :

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

Suites de tests

Regrouper et ordonner des tests unitaires :

```
package fr.upmc.ilp.ilp1.fromxml;
import junit.framework.Test;
import junit.framework.TestSuite;

/** Regroupement de classes de tests pour fromxml. */

public class MainTestSuite extends TestSuite {

  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new TestSuite(ASTParserTest.class));
    suite.addTest(new TestSuite(MainTest.class));
    return suite;
  }
}
```

Mise en œuvre en ligne de commande ou Eclipse.

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) :

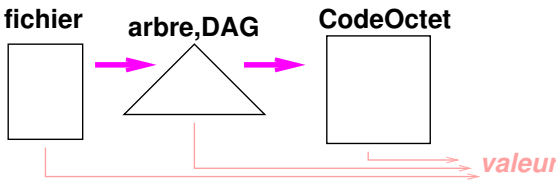
```
@RunWith   @SuiteClasses
           @Parameters
```


Automatiser

Sous Eclipse, les classes de tests JUnit3 et Junit4 sont dans les bibliothèques pré-existantes.

Interprétation

Analyser la représentation du programme pour en calculer la valeur et l'effet.
Un large spectre de techniques :



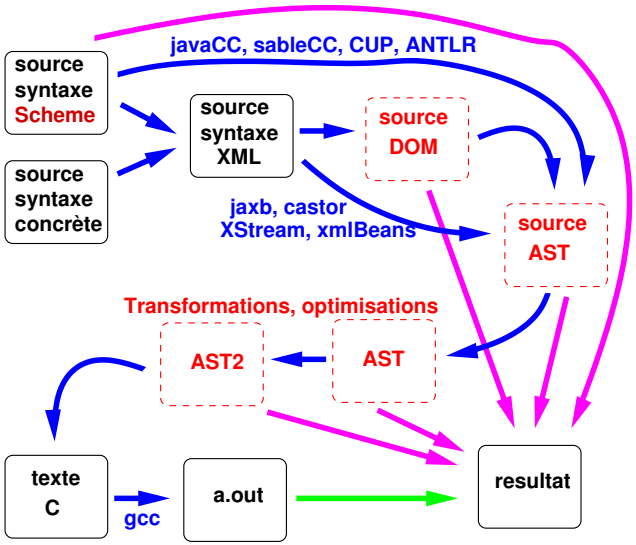
- interprétation pure sur chaîne de caractères : lent
- interprétation d'arbre (ou DAG) : rapide, traçable
- interprétation de code-octet : rapide, compact, portable

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.

Grand schéma



Hypothèses

L'interprète est écrit en Java 5.

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 = 12345678901234567890123456789012345678901234567890;
  f = 1.2345678901234567890123456789012345e-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 des variables locales.
- 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**)

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.

La 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.

Opérateurs

ILP1 a deux espaces de noms :

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

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

Interprète en Java

- On souhaite ajouter à tous les objets représentant un morceau de code une méthode `eval` quelque chose comme :

```
Object eval (LexicalEnvironment lexenv,
             Common common )
    throws Exception;
```

- On sépare environnement lexical et global.
- Les opérateurs sont dans l'environnement global.
- Des exceptions peuvent surgir !
- On souhaite se réserver le droit de changer d'implantation d'environnements (pourquoi ?) :

La classe abstraite racine (pour factoriser le code commun) :

```
package fr.upmc.ilp.ilp1.eval;
import fr.upmc.ilp.ilp1.interfaces.*;
import fr.upmc.ilp.ilp1.runtime.*;

public abstract class EAST implements IAST {

  public abstract Object eval (
         ILexicalEnvironment lexenv,
         ICommon common )
    throws EvaluationException;

  ...
}
```

ILexicalEnvironment

```
package fr.upmc.ilp.ilp1.runtime;
import fr.upmc.ilp.ilp1.interfaces.*;

public interface ILexicalEnvironment {

  /** Renvoie la valeur d'une variable si presente dans
   * l'environnement.
   *
   * @throws EvaluationException si variable absente.
   */
  Object lookup (IASTvariable variable)
    throws EvaluationException;

  /** etend avec un nouveau couple variable-valeur. */
  ILexicalEnvironment extend (IASTvariable variable,
                              Object value);
}
```

Une implantation naïve est une liste chaînée.

Ressource: Java/src/fr/upmc/ilp/ilp1/runtime/LexicalEnvironment.java

ICommon

```
package fr.upmc.ilp.ilp1.runtime;

public interface ICommon {
  /** Appliquer un operateur unaire sur un operande. *
  Object applyOperator(String opName, Object operand)
        throws EvaluationException;
  /** Appliquer un operateur binaire sur deux operande
  Object applyOperator(String opName,
                       Object leftOperand,
                       Object rightOperand)
        throws EvaluationException;
}
```

Un opérateur n'est pas un « citoyen de première classe », il ne peut qu'être appliqué.

Ressource: Java/src/fr/upmc/ilp/ilp1/runtime/Common.java

Hiérarchies et résumé

```
EAST // pour evaluable AST
   eval(ILexicalEnvironment, ICommon)

ILexicalEnvironment
   lookup(IASTvariable)
   extend(IASTvariable, Object)

ICommon
   applyOperator(opName, operand)
   applyOperator(opName, leftOpnd, rightOpnd)
```


Opérateurs

Le code de bien des opérateurs se ressemblent à quelques variations syntaxiques près : il faut factoriser !
Pour ce faire, j'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

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;
```

Alternative

```
public Object eval (ILexicalEnvironment lexenv,
                    ICommon common)
  throws EvaluationException {           // transmission
  Object c = condition.eval(lexenv, common);
  if ( c instanceof Boolean ) {       // typage
    Boolean b = (Boolean) c;
    if ( b.booleanValue() ) {
      return consequence.eval(lexenv, common);
    } else {
      if ( isTernary() ) {
        return alternant.eval(lexenv, common);
      } else {
        return EAST.voidConstantValue();
      }
    }
  } else {
    return consequence.eval(lexenv, common);
  }
}
```


Séquence

```
public Object eval (ILexicalEnvironment lexenv,
                    ICommon common)
  throws EvaluationException {
  Object last = EAST.voidConstantValue();
  for (int i = 0 ; i < instruction.length ; i++) {
    last = instruction[i].eval(lexenv, common);
  }
  return last;
}
```


Constante

Toutes les constantes ont une valeur décrite par une chaîne.

```
public abstract class EASTConstant extends EAST
{

  protected EASTConstant (Object value) {
    this.valueAsObject = value;
  }
  protected final Object valueAsObject;

  /** Les constantes valent leur propre valeur */

  public Object eval (ILexicalEnvironment lexenv,
                      ICommon common) {
    return valueAsObject;
  }
}
```


Flottant

Chaque sous-classe de constante décrit comment convertir la chaîne décrivant leur valeur en un objet Java :

```
public class EASTflottant
  extends EASTConstant
  implements IASTfloat {

  public EASTflottant (String valeur) {
    super(new Double(valeur));
  }
}
```

Variable

```
  public Object eval (ILexicalEnvironment lexenv,
                      ICommon common)
    throws EvaluationException {
    return lexenv.lookup(this);
  }
```

et l'environnement (une liste chaînée de couples (nom, valeur)) :

```
public class LexicalEnvironment
  implements ILexicalEnvironment {

  public LexicalEnvironment (IASTvariable variable,
                             Object value,
                             ILexicalEnvironment next )
  {
    this.variableName = variable.getName();
    this.value = value;
    this.next = next;
  }
```



```
  private final String variableName;
  private volatile Object value;
  private final ILexicalEnvironment next;

  public Object lookup (IASTvariable variable)
    throws EvaluationException {
    if (variableName.equals(variable.getName())) {
      return value;
    } else {
      return next.lookup(variable);
    }
  }

  /** On peut etendre tout environnement. */
  public ILexicalEnvironment extend (
          IASTvariable variable, Object value) {
    return new LexicalEnvironment(
          variable, value, this);
  }
}
```


Hiérarchies

```
EAST    // pour evaluable AST
   eval(ILexicalEnvironment, ICommon)
          fr.upmc.ilp.ilp1.eval.*

ILexicalEnvironment
   lookup(IASTvariable)
   extend(IASTvariable, Object)
          fr.upmc.ilp.ilp1.runtime.LexicalEnvironmen
          fr.upmc.ilp.ilp1.runtime.EmptyLexicalEnvir

ICommon
   applyOperator(opName, operand)
   applyOperator(opName, leftOperand, rightOperand)
          fr.upmc.ilp.ilp1.runtime.Common
```

Ressource: Java/src/fr/upmc/ilp/ilp1/runtime/*.java

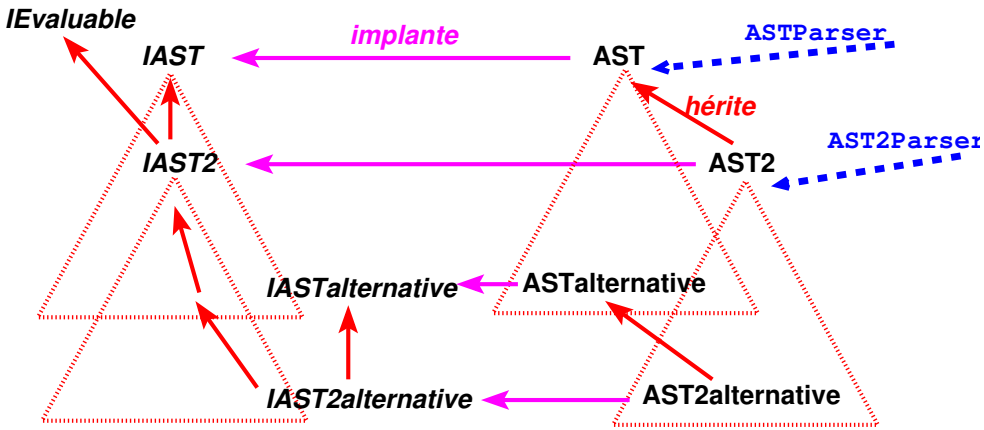
Ressource: Java/src/fr/upmc/ilp/ilp1/eval/*.java

Problème !

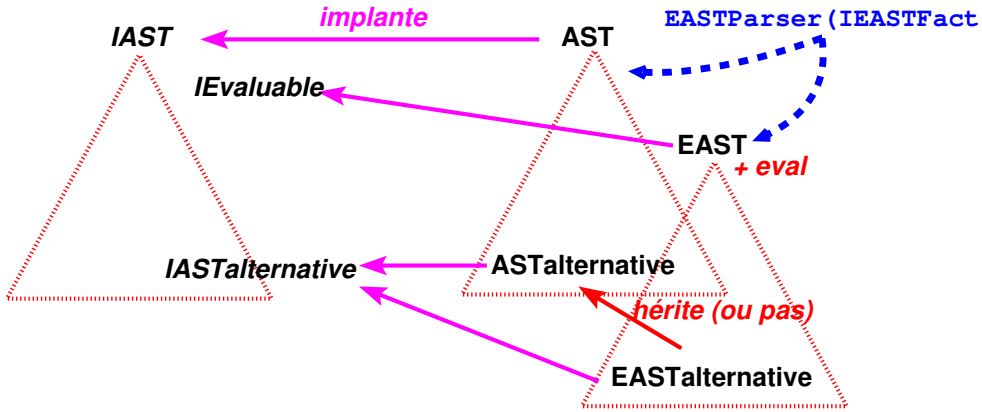
Comment installer la méthode `eval` ?

1. il est interdit de modifier une interface comme `IAST`
2. on ne peut modifier le code du cours précédent `ASTParser`

Solution 1 : duplication



Solution 2 : analyseur partagé



Fabrique : interface

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

```
package fr.upmc.ilp.ilp1.eval;
import fr.upmc.ilp.ilp1.interfaces.*;

public interface IEASTFactory {
  /** Creer une sequence d'AST. */
  IASTsequence newSequence (List<IAST> asts);
  /** Creer une alternative binaire. */
  IASTalternative newAlternative (
     IAST condition, IAST consequent);
  /** Creer une alternative ternaire. */
  IASTalternative newAlternative (
     IAST condition, IAST consequent,
                     IAST alternant);
  ...
```


Fabrique : implantation

```
package fr.upmc.ilp.ilp1.eval;
import fr.upmc.ilp.ilp1.interfaces.*;

/** Une fabrique pour fabriquer des EAST. */

public class EASTFactory implements IEASTFactory {

  /** Creer une sequence d'AST. */
  public IASTsequence newSequence (List<IAST> asts) {
    return new EASTsequence(asts);
  }

  /** Creer une alternative binaire. */
  public IASTalternative newAlternative (
       IAST condition, IAST consequent) {
    return new EASTalternative(condition, consequent);
  }
```

Emploi de la fabrique

```
public class EASTParser {

  public EASTParser (final IEASTFactory factory) {
    this.factory = factory;
  }
  private final IEASTFactory factory;

  public IAST parse (final Node n)
    ...
      case Node.ELEMENT_NODE: {
        final Element e = (Element) n;
        final NodeList nl = e.getChildNodes();
        final String name = e.getTagName();
        ...
        } else if ( "sequence".equals(name) ) {
          return factory.newSequence(this.parseList(nl));

        } else if ( "alternative".equals(name) ) {
          final IAST cond   = findThenParseChild(nl, "condition");
          final IAST conseq = findThenParseChild(nl, "consequence");
          try {
            final IAST alt = findThenParseChild(nl, "alternant");
            return factory.newAlternative(cond, conseq, alt);
          } catch (ILPException exc) {
            return factory.newAlternative(cond, conseq);
          }
```


Architecture de tests

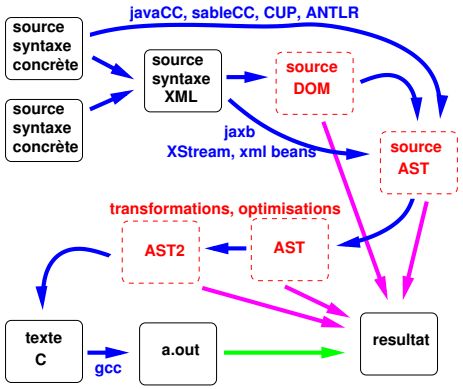
Tests unitaires et suite de tests :

Ressource: Java/src/fr/upmc/ilp/ilp1/eval/EASTTest.java

Ressource: Java/src/fr/upmc/ilp/ilp1/eval/EASTPrimitiveTest.java

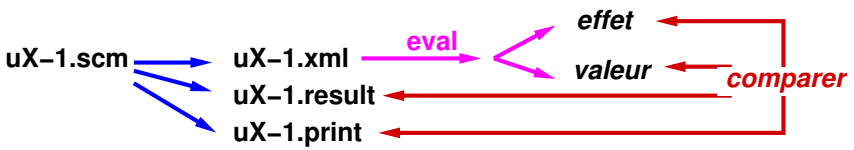
Ressource: Java/src/fr/upmc/ilp/ilp1/eval/EASTTestSuite.java

Pour des tests plus conséquents ...



Batterie de tests

Ressource: Grammars/Scheme/u*-1.scm



Ressource: Java/src/fr/upmc/ilp/ilp1/eval/EASTFileTest.java

Récapitulation des paquetages

```
fr.upmc.ilp.annotation
fr.upmc.ilp.tool1
fr.upmc.ilp.ilp1
fr.upmc.ilp.ilp1.interfaces
fr.upmc.ilp.ilp1.fromxml
fr.upmc.ilp.ilp1.runtime
fr.upmc.ilp.ilp1.eval
fr.upmc.ilp.ilp1.cgen      // prochain cours...
```

3000 lignes environ (commentaires compris) dont 600 lignes de tests JUnit.

Récapitulation

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

- JUnit
- ajout de fonctionnalité
- fabrique

Plan du cours 3

- Compilation vers C
- Représentation des concepts en C
- Bibliothèque d'exécution

Compilation vers C

Analyser la représentation du programme pour le transformer en un programme calculant sa valeur et son effet.
Un interprète fait, un compilateur fait faire.

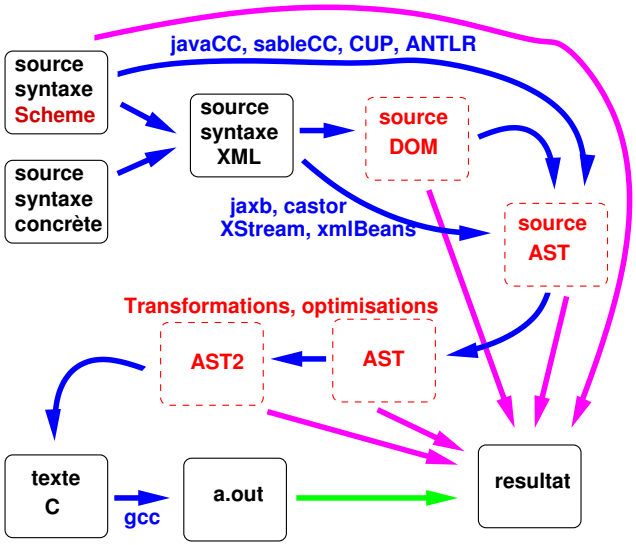
- Programme : données $\longrightarrow$ résultat
- Interprète : programme $\times$ données $\longrightarrow$ résultat
- Compilateur : programme $\longrightarrow$ (données $\longrightarrow$ résultat)

Concepts présents dans ILP1

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

mais, en C, pas de typage dynamique, pas de gestion de la mémoire.
Par contre, C connaît la notion d'environnement.

Grand schéma



Hypothèses

Le compilateur est écrit en Java.

1. Il prend un IAST,
2. le compile en C.

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

- que ce soit lui qui le traite (propriété **statique**)
- ou le code engendré qui le traite (propriété **dynamique**).

Statique/dynamique

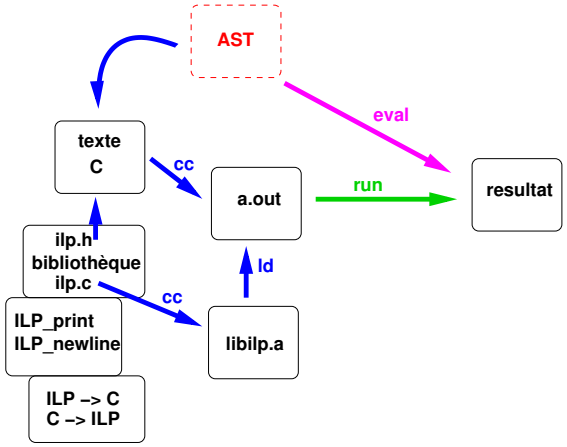
Est **dynamique** ce qui ne peut être résolu qu'à l'exécution.
Est **statique** ce qui peut être déterminé avant l'exécution.

Statique et dynamique

```
{ int x = round(2.78);
  print(y);              // y variable inconnue!
  float z;
  print(z);              // z non initialisee!
  if ( foo(x) ) {
    z = x/(3 - x);       //
    print(z);            // z est definie
  };
  print(z)               // qu'est z ?
}
```

Composantes

On souhaite que le compilateur ne dépende pas de la représentation exacte des données.

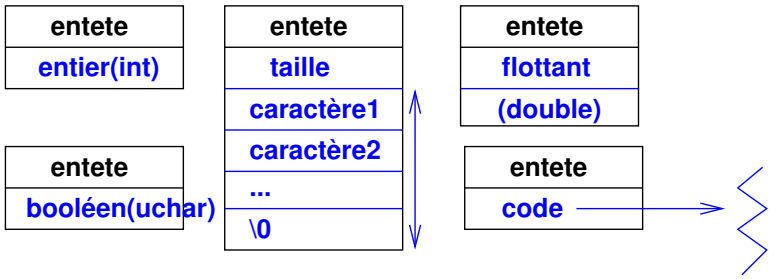


Représentation des valeurs

On s'appuie sur C :

Ressource: C/ilp.c

Afin de pouvoir identifier leur type à l'exécution (propriété dynamique), toute valeur est une structure allouée dotée d'un entête (indiquant son type) et d'un corps et manipulée par un pointeur.




```
typedef struct ILP_Object {
     enum ILP_Kind       _kind;
     union {
          unsigned char asBoolean;
          int           asInteger;
          double        asFloat;
          struct asString {
               int      _size;
               char     asCharacter[1];
          } asString;
          struct asPrimitive {
               void*    _code;
          } asPrimitive;
     }                  _content;
} *ILP_Object;
```



```
enum ILP_Kind {
     ILP_BOOLEAN_KIND       = 0xab010ba,
     ILP_INTEGER_KIND       = 0xab020ba,
     ILP_FLOAT_KIND         = 0xab030ba,
     ILP_STRING_KIND        = 0xab040ba,
     ILP_PRIMITIVE_KIND     = 0xab050ba
};
enum ILP_BOOLEAN_VALUE {
     ILP_BOOLEAN_FALSE_VALUE = 0,
     ILP_BOOLEAN_TRUE_VALUE  = 1
};
```

```
struct ILP_Object ILP_object_true = {
     ILP_BOOLEAN_KIND,
     { ILP_BOOLEAN_TRUE_VALUE }
};

#define ILP_TRUE  (&ILP_object_true)
```


Structures

Pour chaque type de données d'ILP :

- constructeurs (allocateurs)
- reconnaisseur (grâce au type présent à l'exécution)
- accesseurs
- opérateurs divers

et, à chaque fois, les macros (l'interface) et les fonctions (l'implantation).

Autour des booléens

Fonctions ou macros d'appoint :

```
#define ILP_Boolean2ILP(b) \
  ILP_make_boolean(b)
#define ILP_isBoolean(o) \
  ((o)->_kind == ILP_BOOLEAN_KIND)
#define ILP_CheckIfBoolean(o) \
  if ( ! ILP_isBoolean(o) ) { \
       ILP_domain_error("Not a boolean", o); \
  };
#define ILP_isEquivalentToTrue(o) \
  ((o) != ILP_FALSE)
```



```
ILP_Object
ILP_make_boolean (int b)
{
     if ( b ) {
          return ILP_TRUE;
     } else {
          return ILP_FALSE;
     }
}
```

Autour des entiers

Fonctions ou macros d'appoint :

```
#define ILP_Integer2ILP(i) \
  ILP_make_integer(i)
#define ILP_isInteger(o) \
  ((o)->_kind == ILP_INTEGER_KIND)
#define ILP_CheckIfInteger(o) \
  if ( ! ILP_isInteger(o) ) { \
       ILP_domain_error("Not an integer", o); \
  };
#define ILP_AllocateInteger() \
  ILP_malloc(sizeof(struct ILP_Object), ILP_INTEGER_
```



```
#define ILP_Minus(o1,o2) \
  ILP_make_subtraction(o1, o2)
#define ILP_LessThan(o1,o2) \
  ILP_compare_less_than(o1,o2)
#define ILP_LessThanOrEqual(o1,o2) \
  ILP_compare_less_than_or_equal(o1,o2)
```



```
ILP_Object
ILP_make_integer (int d)
{
     ILP_Object result = ILP_AllocateInteger();
     result->_content.asInteger = d;
     return result;
}

ILP_Object
ILP_malloc (int size, enum ILP_Kind kind)
{
     ILP_Object result = malloc(size);
     if ( result == NULL ) {
          return ILP_error("Memory exhaustion");
     };
     result->_kind = kind;
     return result;
}
```



```
ILP_Object
ILP_make_addition (ILP_Object o1, ILP_Object o2)
{
  if ( ILP_isInteger(o1) ) {
    if ( ILP_isInteger(o2) ) {
      ILP_Object result = ILP_AllocateInteger();
      result->_content.asInteger =
          o1->_content.asInteger
        + o2->_content.asInteger;
      return result;
    } else if ( ILP_isFloat(o2) ) {
      ILP_Object result = ILP_AllocateFloat();
      result->_content.asFloat =
          o1->_content.asInteger
        + o2->_content.asFloat;
      return result;
    } else {
      return ILP_domain_error("Not a number", o2);
    }
    ...
```

Attention : l'addition consomme de la mémoire (comme en Java) !

```
#define DefineComparator(name,op)
\
ILP_Object
\
ILP_compare_##name (ILP_Object o1, ILP_Object o2)
\
{
\
  if ( ILP_isInteger(o1) ) {
\
    if ( ILP_isInteger(o2) ) {
\
      return ILP_make_boolean(
\
          o1->_content.asInteger op o2->_content.asInteger); \
    } else if ( ILP_isFloat(o2) ) {
\
      return ILP_make_boolean(
\
          o1->_content.asInteger op o2->_content.asFloat);
\
    } else {
\
      return ILP_domain_error("Not a number", o2);
\
    }
   ...
```


Primitives

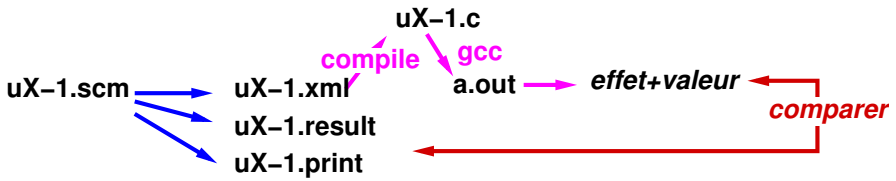
```
ILP_Object
ILP_print (ILP_Object o)
{
  switch (o->_kind) {
    case ILP_INTEGER_KIND: {
       fprintf(stdout, "%d", o->_content.asInteger);
       break;
    }
    case ILP_FLOAT_KIND: {
       fprintf(stdout, "%12.5g", o->_content.asFloat);
       break;
    }
    case ILP_BOOLEAN_KIND: {
       fprintf(stdout, "%s", (ILP_isTrue(o) ? "true" : "fals
       break;
    }
    ...
  return ILP_FALSE;
}
```


Mise en œuvre du compilateur

Ressource: Java/src/fr/upmc/ilp/ilp1/cgen/CgeneratorTest.java

```
public void setUp () {
  ICgenEnvironment common = new CgenEnvironment();
  compiler = new Cgenerator(common);
  factory = new EASTFactory();
  lexenv = CgenEmptyLexicalEnvironment.create();
  lexenv = common.extendWithPrintPrimitives(lexenv);
}
private Cgenerator compiler;
private EASTFactory factory;
private ICgenLexicalEnvironment lexenv;
```


Mise en œuvre et test du compilateur



```
...
EAST east = (EAST) parser.parse(d);
// Compiler vers C
String ccode = compiler.compile(
                 east, lexenv, "return");
FileTool.stuffFile(basefilename + ".c", ccode);
String program = "bash "
  + "C/compileThenRun.sh "                    //
  + basefilename + ".c";
ProgramCaller pc = new ProgramCaller(program);
pc.setVerbose();
pc.run();
String expectedResult =
  readExpectedResult(basefilename).trim();
String expectedPrinting =
  readExpectedPrinting(basefilename).trim();
assertEquals(expectedPrinting + expectedResult,
             pc.getStdout().trim());
```


Récapitulation

- le compilateur fait faire !
- bibliothèque d'exécution
- représentation des données en C
  - constructeur, reconnaisseur, accesseur
- conversion ILP <-> C
- statique/dynamique

Plan du cours 4

- Génération de code
- Récapitulation
- Techniques Java

Principes de compilation

- Les variables ILP sont compilées en variables C
- Les expressions ILP sont compilées en expressions C
- Les instructions ILP sont compilées en instructions C

Compilation

Le compilateur doit avoir connaissance des environnements en jeu.
Il est initialement créé avec un environnement global :

Ressource: Java/fr/upmc/ilp/ilp1/cgen/Cgenerator.java

```
public Cgenerator (final ICgenEnvironment common)
  this.common = common;
}
private final ICgenEnvironment common;
```

et compile avec l'environnement lexical courant :

```
public synchronized String compile (
            final IAST iast,
            final ICgenLexicalEnvironment lexenv,
            final String destination)
  throws CgenerationException;
```


Environnement global

- Compiler les appels aux opérateurs,
- Compiler les appels aux primitives,
- Vérifier l'existence, l'arité,
- Coordonner les ressources communes.


```
package fr.upmc.ilp.ilp1.cgen;
import fr.upmc.ilp.ilp1.interfaces.*;
public interface ICgenEnvironment {
  /** Comment convertir un operateur unaire en C. */
  String compileOperator1 (String opName)
    throws CgenerationException ;

  /** Comment convertir un operateur binaire en C. *
  String compileOperator2 (String opName)
    throws CgenerationException ;

  /** un generateur de variables temporaires. */
  IASTvariable generateVariable ();

  /** L'enrichisseur d'environnement lexical avec le
  ICgenLexicalEnvironment
    extendWithPrintPrimitives (ICgenLexicalEnvironme
}
```


Environnement lexical

- Compiler une variable locale
- Détecter les variables manquantes

```
package fr.upmc.ilp.ilp1.cgen;
import fr.upmc.ilp.ilp1.interfaces.*;
public interface ICgenLexicalEnvironment  {
  /** Renvoie le code C d'acces a cette variable. */
  String compile (IASTvariable variable)
    throws CgenerationException;
  /** etend l'environnement avec une nouvelle variable
   * et vers quel nom en C la compiler. */
  ICgenLexicalEnvironment extend (IASTvariable variabl
                                  String compiledName
  /** etend l'environnement avec une nouvelle variable
   * qui sera compilee par son propre nom. */
  ICgenLexicalEnvironment extend (IASTvariable variabl
}
```

Génération de code

Le compilateur va essayer de produire du C ressemblant :

```
void analyzeExpression (IAST iast,
                        ICgenLexicalEnvironment le
                        ICgenEnvironment common)
  throws CgenerationException;
void analyzeInstruction (IAST iast,
                         ICgenLexicalEnvironment
                         ICgenEnvironment common,
                         String destination)
  throws CgenerationException;
```

Tout repose sur une méthode `analyze` qui utilisera une méthode privée surchargée `generate` (afin de ne pas modifier les programmes précédents !)


```
private void analyze (      // Discriminant
       IAST iast,
       ICgenLexicalEnvironment lexenv,
       ICgenEnvironment common,
       String destination)
 throws CgenerationException {
 if ( iast instanceof IASTconstant ) {
   if ( iast instanceof IASTboolean ) {
     generate((IASTboolean) iast,
              lexenv, common, destination);
   } else if ( iast instanceof IASTfloat ) {
     generate((IASTfloat) iast,
              lexenv, common, destination);
   ...
   } else {
     String msg = "Unknown type of constant: " + i
     throw new CgenerationException(msg);
   }
```



```
 } else if ( iast instanceof IASTalternative ) {
   generate((IASTalternative) iast,
            lexenv, common, destination);
 } else if ( iast instanceof IASTinvocation ) {
   generate((IASTinvocation) iast,
            lexenv, common, destination);
 } else if ( iast instanceof IASToperation ) {
   if ( iast instanceof IASTunaryOperation ) {
         ...
}
private void generate (     // methode
       IASTunaryOperation iast,
       ICgenLexicalEnvironment lexenv,
       ICgenEnvironment common,
       String destination)
  throws CgenerationException {
     ...
}
```


Destination

Toute expression doit rendre un résultat.
Toute fonction doit rendre la main avec `return`.
La **destination** indique que faire de la valeur d'une expression ou d'une instruction.
Notations pour ILP1 :

$\overrightarrow{\textit{expression}}$	laisser la valeur en place
$\longrightarrow$`return` *programme*	sortir de la fonction avec la valeur
$\longrightarrow$`(void)` *instruction*	finir l'instruction en perdant la valeur

Grandes règles

- les variables ILP sont compilées en variables C
- les expressions ILP sont compilées en expressions C
- les instructions ILP sont compilées en instructions C

Compilation de l'alternative

$\overrightarrow{alternative}$

```
if ( ILP_isEquivalentToTrue( condition ) ) {
   consequence ;
} else {
   alternant ;
}
```


Compilation de la séquence

$\overrightarrow{séquence}$

```
{
   (void)
   instruction1 ;
   (void)
   instruction2 ;
   ...
   dernièreInstruction ;
}
```


Compilation du bloc unaire I

Comme au judo, utiliser la force du langage cible !

$\overrightarrow{bloc}$

```
{
   ILP_Object variable = initialisation ;

   (void)
   instruction1 ;
   (void)
   instruction2 ;
   ...
   dernièreInstruction ;
}
```

Mais ...

Compilation du bloc unaire II

$\overrightarrow{d}$
bloc

```
{
  ILP_Object temporaire = initialisation ;
  ILP_Object variable = temporaire;

  →(void)
  instruction1  ;
  →(void)
  instruction2  ;
  ...
        →d
  dernièreInstruction  ;
}
```

En C, une variable existe dès qu'elle est nommée.

Compilation d'une constante

$\overrightarrow{d}$
constante

```
d  ILP_Integer2ILP(constanteEntière)
                    /* ou CgenerationException */
d  ILP_Float2ILP(constanteFlottante)
d  ILP_TRUE
d  ILP_FALSE
d  ILP_String2ILP("constanteChaînePlusProtection")
```


Compilation d'une invocation

On utilise la force du langage C. La bibliothèque d'exécution comprend également les implantations des fonctions prédéfinies `print` et `newline` (respectivement `ILP_print` et `ILP_newline`).

$\overrightarrow{d}$
invocation

```
d fonctionCorrespondante(
          →
        argument1 ,
          →
        argument2 ,
        ... )
```


Compilation d'une opération

À chaque opérateur d'ILP1 correspond une fonction dans la bibliothèque d'exécution.

$\overrightarrow{d}$
opération

```
d fonctionCorrespondante(
          →
        opérandeGauche ,
          →
        opérandeDroit  )
```

Ainsi, `+` correspond à `ILP_Plus`, `-` correspond à `ILP_Minus`, etc.

Compilation d'une variable

$$\xrightarrow{d} variable$$

```
d variable     /* ou CgenerationException */
```

Attention aussi une conversion (*mangling*) est parfois nécessaire !

Exemples

```
;;; $Id: u10-1.scm 405 2006-09-13 17:21:53Z queinnec $
(comment "operateur binaire -" 9)
(- 43 34)

;;; end of u10-1.scm

{
  return ILP_Minus( ILP_Integer2ILP(43) ,
                    ILP_Integer2ILP(34) ) ;
}
```



```
;;; $Id: u29-1.scm 405 2006-09-13 17:21:53Z queinnec $
(comment "bloc unaire (portee des initialisations)" 36
(let ((x 3))
  (let ((x (+ x x)))
    (* x x) ) )

;;; end of u29-1.scm
```



```
{
  { /* let x = 3 in */
    ILP_Object TEMP6 = ILP_Integer2ILP(3) ;
    ILP_Object x = TEMP6 ;
    {
      { /* let x = x + x in */
        ILP_Object TEMP7 = ILP_Plus( x , x ) ;
        ILP_Object x = TEMP7 ;
        {  /* x * x */
          return ILP_Times( x , x ) ;
        }
      };
    }    /* conflit si mot cle */
  };     /* conflit possible TEMPi */
}
```

Habillage final

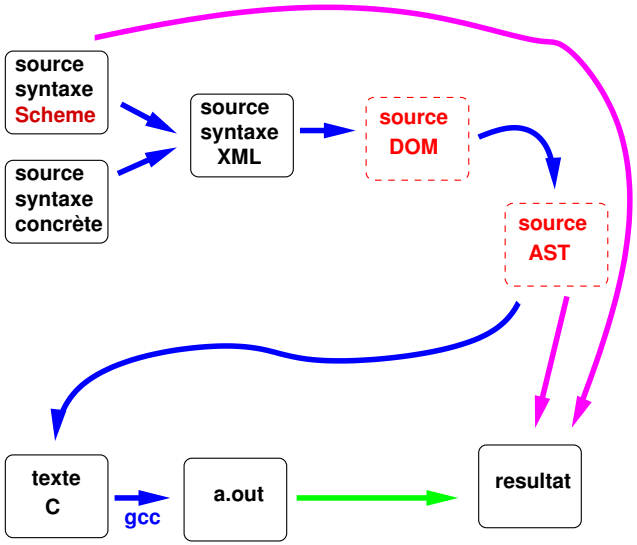
Le script `compileThenRun.sh` insère le C engendré dans un vrai programme syntaxiquement complet :

```
#include <stdio.h>
#include <stdlib.h>
#include "ilp.h"

ILP_Object program ()
#   include FICHIER_C

int main (int argc, char *argv[]) {
     ILP_print(program());
     ILP_newline();
     return EXIT_SUCCESS;
}
```

D'autres habillages sont possibles !

Ne pas oublier en compilant de lier avec la bibliothèque `libilp.a`.

Grandes masses



Les grandes masses, paquetages et leur fonction :

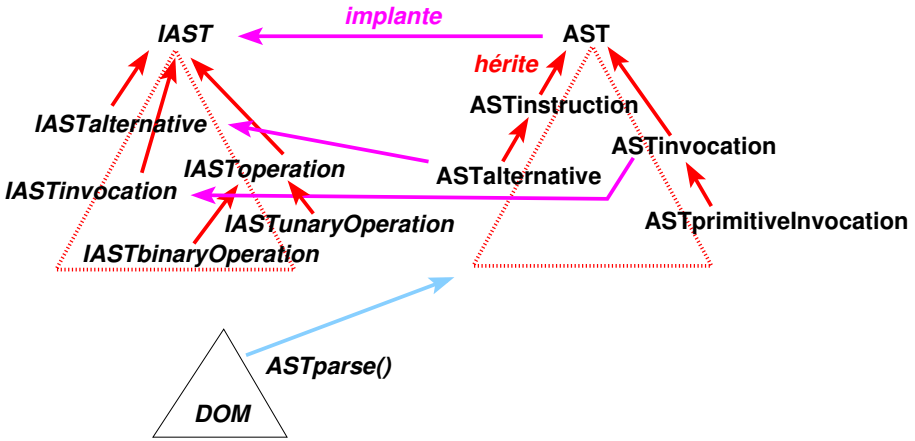
```
fr.upmc.ilp.ilp1
  .interfaces interfaces d'AST
  .fromxml    texte -> AST
  .runtime    bibliotheque d'execution interpretation
  .eval       interprete
  .cgen       compilateur

C/libilp.a    bibliotheque d'execution compilation
```


Interfaces

```
    // En fr.upmc.ilp.ilp1.interfaces
IAST              // Hierarchie minimale
  IASTalternative
  IASTconstant
    IASTboolean
    IASTfloat
    IASTinteger
    IASTstring
  IASTinvocation
  IASToperation
    IASTunaryOperation
    IASTbinaryOperation
  IASTsequence
  IASTvariable
...
```

Analyse syntaxique

```
     // En fr.upmc.ilp.ilp1.fromxml
AST                        // Hierarchie plate
  ASTalternative
  ASTblocUnaire
  ASTbooleen
  ASTchaine
  ASTentier
  ASTinvocation
    ASTinvocationPrimitive
...
ASTParser
ASTException
```

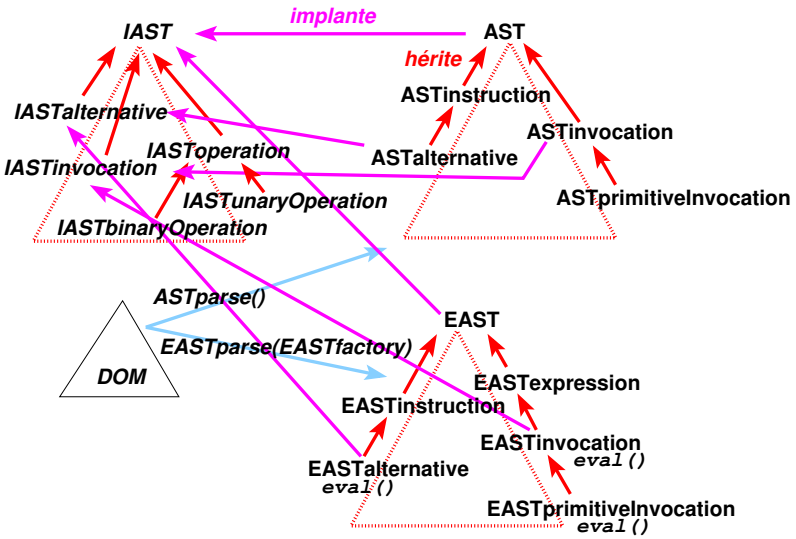
Bibliothèque d'interprétation

```
     // En fr.upmc.ilp.ilp1.runtime
ILexicalEnvironment
  LexicalEnvironment
  EmptyLexicalEnvironment
ICommon
  Common
  CommonPlus
PrintStuff            // Extenseurs d'ICommon
ConstantsStuff
Invokable             // Pour les primitives
  AbstractInvokableImpl
```


Interprétation

```
     // En fr.upmc.ilp.ilp1.eval
EAST                    // avec methode eval()
  EASTalternative
  EASTblocUnaire
  EASTConstant
    EASTbooleen
    EASTchaine
    EASTentier
    EASTflottant
  EASToperation
    EASToperationUnaire
    EASToperationBinaire
  EASTvariable
  ...
```

```
IEASTFactory        // fabrique
  EASTFactory
EASTParser          // parametre par fabrique
EASTException
```

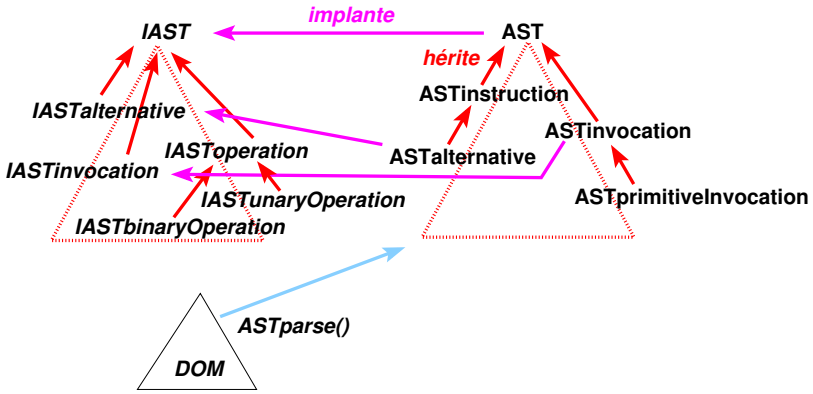
Compilation

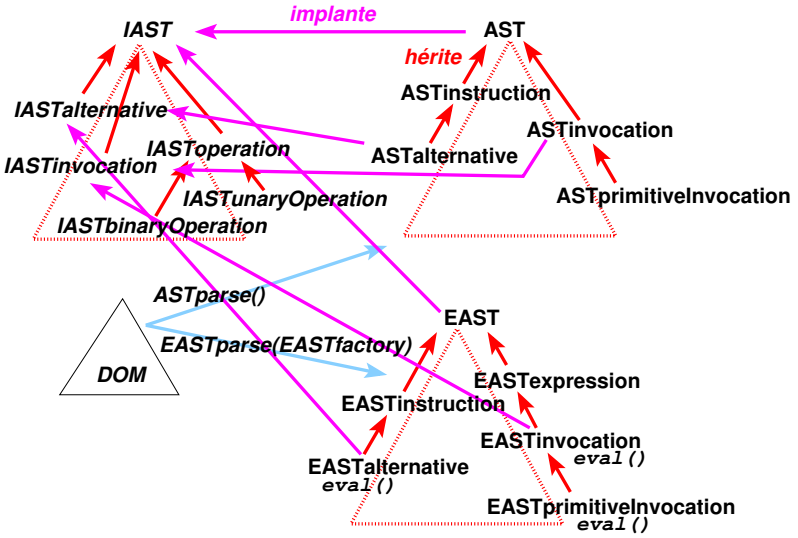
```
    // En fr.upmc.ilp.ilp1.cgen
ICgenEnvironment
  CgenEnvironment
ICgenLexicalEnvironment
  CgenLexicalEnvironment
    CgenEmptyLexicalEnvironment
CgenerationException
Cgenerator    // discriminant + methodes/AST
```

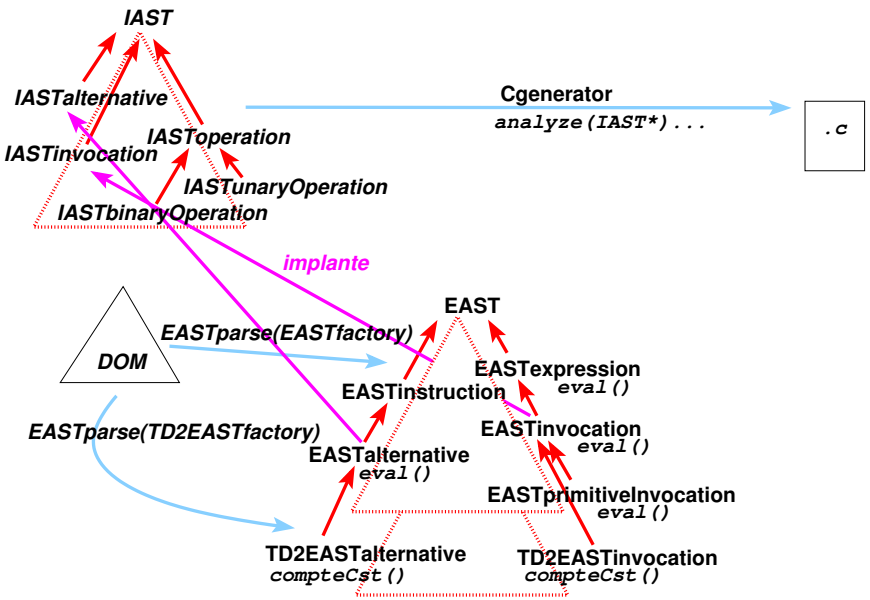

Mise en oeuvre et tests

```
   En fr.upmc.ilp.ilp1
  Process       // et notifieur
  ProcessTest
  WholeTestSuite

fromxml.ASTParserTest

eval.Test
eval.EASTPrimitiveTest
eval.FileTest

cgen.CgeneratorTest
```

Techniques Java



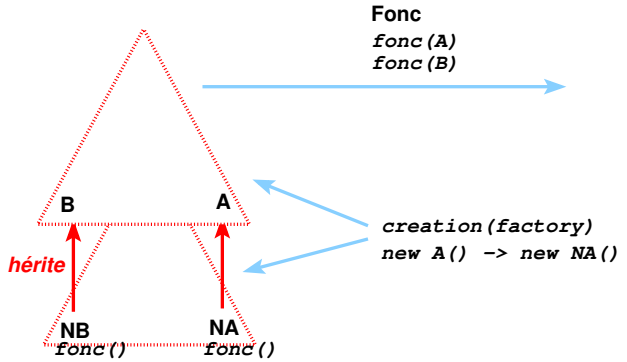


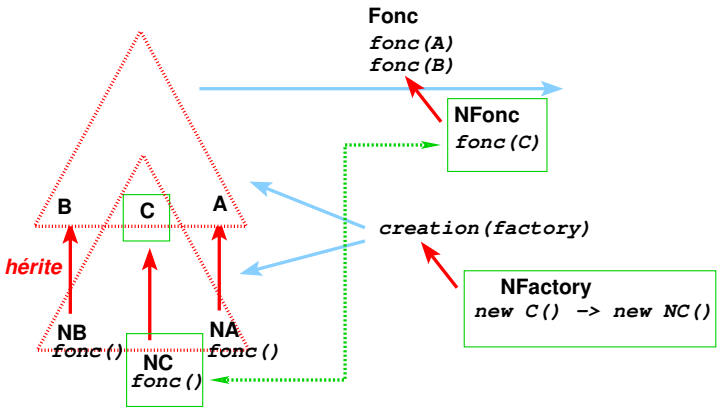


Extensions

Deux sortes d'évolution :

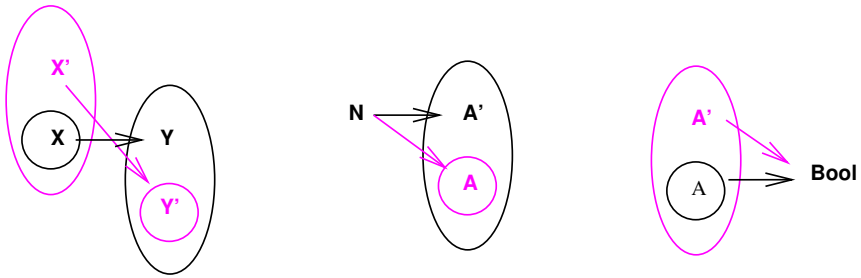
- introduction de nouveaux noeuds d'AST
- introduction de nouvelles fonctionnalités



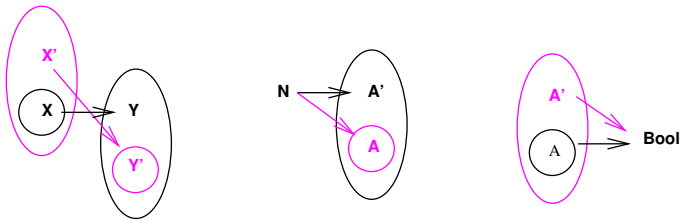


Contravariance/covariance

A est un sous-type de B si un $a \in A$ peut remplacer un $b \in B$ dans tous ses emplois possibles.
Une fonction $X' \to Y'$ est un sous-type de $X \to Y$ ssi $X \subset X'$ et $Y' \subset Y$.

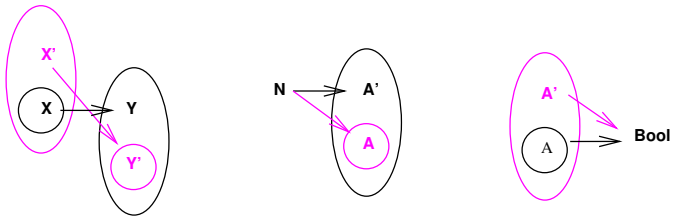


NB : J'utilise l'inclusion ensembliste comme notation pour le sous-typage. N est l'ensemble des entiers.



Cas des tableaux : si $A \subset A'$ alors $N \to A$ sous-type de $N \to A'$ donc `A[]` sous-type de `A'[]`.
Attention, en Java, le type d'un tableau est statique et ne dépend pas du type de ses éléments :

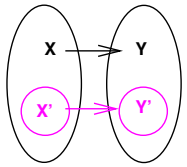
```
Point[] ps = new Point[]{ new PointColore() };
// PointColore[] pcs = (PointColore[]) ps;   // FAUX
PointColore[] pcs = new PointColore[ps.length];
for ( int i=0 ; i<pcs.length ; i++ ) {
   pcs[i] = (PointColore) ps[i];
}
```

Cas des ensembles : Si $A \subset A'$ alors $A' \to$ Bool sous-type de $A \to$ Bool mais `Set<A'>` n'est pas en Java un sous-type de `Set<A>` ni vice-versa.
Par contre `Set<A>` est un sous-type de `Collection<A>`.

```
Set<Point> sp = new HashSet<Point>();
sp.add(new PointColore());
Set<PointColore> spc = new HashSet<PointColore>();
// spc.add(new Point());          // INCORRECT!
// spc.addAll(sp);                // INCORRECT!
// sp = (Set<Point>) spc;         // FAUX!
// spc = (Set<PointColore>) sp;   // FAUX!
sp.addAll(spc);
```

Exemple de covariance




```
public interface
 fr.upmc.ilp2.interfaces.IEnvironment<V> {
    IEnvironment<V> getNext ();
    ...
}
public interface
 fr.upmc.ilp2.interfaces.ICgenLexicalEnvironment
 extends IEnvironment<IAST2variable> {
  // Soyons covariant:
  ICgenLexicalEnvironment getNext ();
  ...
}
```


Récapitulation

- statique/dynamique
- choix de représentation (à l'exécution) des valeurs
- bibliothèque d'exécution
- environnements de compilation
- environnements d'exécution de C
- destination

- ajout de classe ou fonctionnalité