

Introduction à ILP

Analyse syntaxique

DLP : Développement d'un langage de programmation
Master STL, Sorbonne Université

Antoine Miné

Année 2019–2020

Cours 1
17 septembre 2019

Implantation d'un langage de programmation

Pratique et réflexion

autour de la **conception** et de l'**implantation** des langages :

- **compilation** et **interprétation** des programmes
- implantation des **traits de haut-niveau** des langages modernes
pas un cours de génération d'assembleur ou d'allocation de registres !
- motivation des traits de langages, **aspects historiques**
comparaison avec C, Java, ML, Smalltalk, etc.
- discussion des aspects **sémantiques**
- discussion des choix possibles d'implantation
- mise en pratique au sein du **langage dédié au cours : ILP**
- **lecture de code** (~ 30 Klignes de code fourni)
- **extension** du langage ILP
- programmation **robuste** et **test**

Langage cible : ILP

Langage de **haut niveau**, de type Javascript / Smalltalk.

- **syntaxe à la ML** (OCaml, etc.)
la syntaxe est la partie la moins intéressante. . .
- valeurs entières, flottantes, chaînes et booléennes
- expressions avec opérateurs unaires et binaires (+, -, ==, <=, ...)
- structures de contrôle : alternatives, boucles, blocs
- variables globales et variables locales de bloc
- **typage dynamique** (pas de type statique associé à une variable)
- primitives (`print`, `newline`)
- fonctions globales et locales, **fonctions de première classe** (*lambdas*)
- **exceptions**
- **objets**, système de **classes** avec **héritage**
- **gestion automatique de la mémoire** (*garbage collector*)

Variété de Langages et outils : Java 8, C, ANTLR 4, Eclipse, JUnit 4.

- analyseur syntaxique : grammaire, en **ANTLR 4**
- **interprète en Java 8**
- **compilateur**, écrit en Java 8 et qui génère du code **C**
- bibliothèque d'exécution **C** pour le code généré
- test automatisé, avec **JUnit 4**
- développement au sein de la plateforme Eclipse
- sources distribuées sous Git

Versions de plus en plus élaborées du langage, de ILP1 à ILP4 :

- chaque version ajoute des fonctionnalités
- chaque version réutilise le code des versions précédentes
- **ILP1** : langage de base
(pas de boucle, d'affectation, de fonction, d'exception, ni d'objet)
- **ILP2** : ajout des boucles, affectations, fonctions (globales)
- **ILP3** : ajout des exceptions, et fonctions locales et de première classe
- **ILP4** : ajout des classes et des objets

Règles :

- compatibilité ascendante : ILP4 peut exécuter un programme ILP1
- orthogonalité des extensions
- **réutilisation maximale du code** : pas (ou peu) de duplication
(en Java : grâce à l'héritage et aux *design patterns*)

Ces règles sont suivies en cours, et sont **à suivre en TME et en examen !**

Organisation du cours

Deux chargés de cours : **Antoine Miné** & **Carlos Agon**.

- **cours 1–4 : ILP1** (langage de base)
 - cours 1 : introduction, analyse syntaxique, AST
 - cours 2 : interprète, visiteur
 - cours 3–4 : compilateur, bibliothèque d'exécution
- **cours 5 : ILP2**, boucles, affectations, fonctions
- **cours 6–7 : ILP3**
 - cours 6 : exceptions
 - cours 7 : fonctions de première classe
- **cours 8–9 : ILP4**
 - cours 8 : interprétation des objets
 - cours 9 : compilation des objets
- **cours 10** : ramasse-miettes, révision

TME : une séance de 4h par semaine sur machine

- groupe 1 (jeudi) : Ghiles Ziat
- groupe 2 (jeudi) : Steven Varoumas
- groupe 3 (vendredi) : Darius Mercadier

Principe : étendre ILP (syntaxe, interprète, compilateur, test)

Évaluation : deux épreuves, sur machine

- 40 % : **partiel**, sur les cours 1 à 5 (ILP1 à ILP2)
- 60 % : **examen final**, sur les cours 1 à 10 (ILP1 à ILP4)

Même principe que les TME ! (correction à la main)

Importance du travail personnel :

- **lire** en détail chez soi les sources d'ILP
- terminer chez soi les TME
- maintenir son espace de travail Eclipse à jour (dernière version d'ILP, TME)

Historique du cours

Inspiré librement du cours original de **Christian Queinnec**.

Ressources d'archive utiles :

- 1 cours ILP à l'UPMC de Christian Queinnec, édition 2007–2008
transparents et bande son sur :
<http://www-master.ufr-info-p6.jussieu.fr/2007/Ext/queinnec/ILP/>
- 2 cours *compiler reading* de Christian Queinnec à l'INSTA :
<https://compiler-reading-1.appspot.com/course/>
- 3 annales 2015–2016 d'ILP sur le site du Master STL à l'UPMC :
<https://www-master.ufr-info-p6.jussieu.fr/2015/DLP>

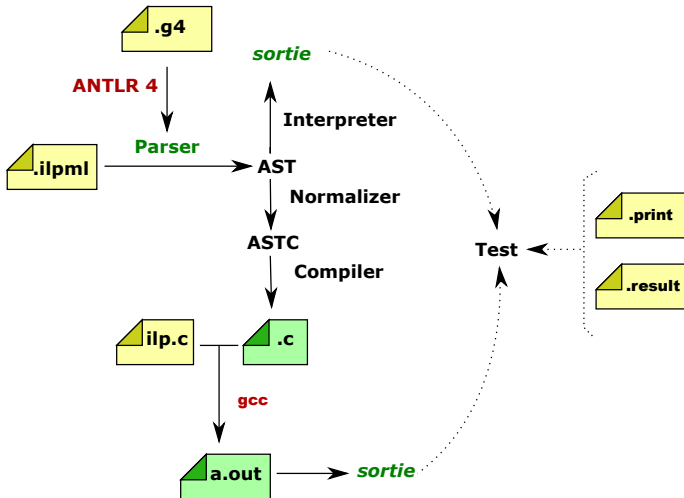
Attention cependant, **modifications importantes** cette année :

- **changement de la syntaxe d'entrée** : pseudo-ML, au lieu de XML
- **analyse syntaxique** basée sur une grammaire ANTLR, au lieu de DOM

Cours 1 : du **programme source** à l'**arbre syntaxique abstrait**

- **schéma général** d'ILP
- langage **ILP1** :
 - syntaxe
 - éléments (informels) de sémantique
- générateur d'analyseur syntaxique : **ANTLR 4**
- **AST** (arbre syntaxique abstrait)
- tests **JUnit 4**

Schéma général d'ILP



Le projet Eclipse ILP contient de nombreux répertoires.

Aujourd'hui, nous regardons surtout :

- `Java/src/`
 - `com.paracamplus.ilp1.ast`
 - `com.paracamplus.ilp1.interfaces`
 - `com.paracamplus.ilp1.parser`
 - `com.paracamplus.ilp1.parser.ilpml`
 - `com.paracamplus.ilp1.test`
 - `com.paracamplus.ilp1.interpreter.test`
 - `com.paracamplus.ilp1.compiler.test`
- `ANTLRGrammars/` (fichiers de grammaire ANTLR pour une syntaxe ML)
- `target/generated-sources/` (fichiers générés par ANTLR)
- `SamplesILP1/` (exemples de source ILP1)

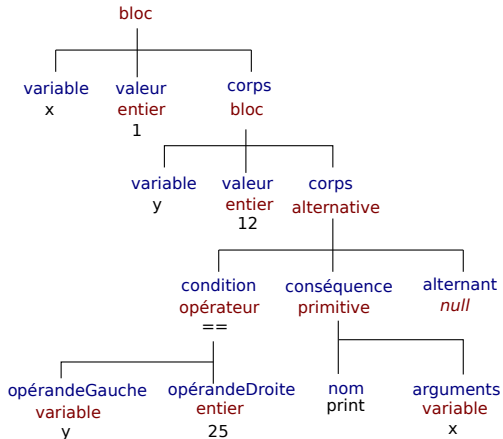
Analyse syntaxique

But

L'analyse syntaxique transforme une séquence de caractères, i.e., le programme source, en une représentation arborescente décrivant la **structure** du programme et précisant la **nature** de chaque mot ou symbole (l'AST).

source

```
let x = 1 in
let y = 12 in
if y == 25 then print(x)
```



Outil : les expressions régulières

Des séquences de lettres aux “mots” (*tokens*).

Une expression régulière représente un ensemble de mots avec :

- des caractères : **a**, **b**, ...
- l'opérateur de concaténation (invisible)
- l'opérateur de choix : **|**
- les opérateurs de répétition : **+** (une fois au moins), ***** (zéro, une fois, ou plus)
- l'opérateur d'option : **?**
- des parenthèses : **(,)**

⇒ utile pour décrire les entiers, flottants, chaînes, noms de variable, etc. de notre langage.

cf. **grep**, **Perl**

Outil : les grammaires

Des séquences de mots aux “phrases”.

Grammaire hors contexte : ensemble de règles de la forme

```

<non-terminal1> ::= expression | ... | expression
<non-terminal2> ::= expression | ... | expression
...

```

où une expression contient :

- des non terminaux **<non-terminal>** (i.e., des variables)
- des terminaux “a”, “let”, ... (i.e., des caractères, des mots)
- des parenthèses

⇒ c’est la **notation BNF** (Backus Naur Form)

- par extension, les opérateurs **+**, *****, **?**, **|** des expressions régulières
(utiles seulement pour la lisibilité : ils peuvent être exprimés dans BNF)

Règles récursives : une phrase est composée de sous-phrases.

Plus expressif que les expressions régulières

⇒ bien adapté aux langages structurés.

Syntaxe concrète du langage ILP1

Syntaxe **ILPML** (programmes ILP avec une syntaxe ML).

	<programme> ::= (<expr> ";"?)* EOF
	<expr> ::= <variable>
(constantes)	"true" "false" <entier> <flottant> "\"" <chaîne> "\"
(opérations unaires)	("-" "!") <expr>
(opérations binaires)	<expr> ("+" "-" ... ">=" "==") <expr>
(variables locales)	"let" <variable> "=" <expr> ("and" <variable> "=" <expr>)* "in" <expr>
(alternative)	"if" <expr> "then" <expr> ("else" <expr>)?
(séquence)	"(" <expr> (";"? <expr>)* ";"? ")"
(appel)	<variable> "(" <expr> ? ("," <expr>)* ")"
	<entier> ::= ("0" ... "9")+
	<variable> ::= ("a" ... "Z" "_") ("0" ... "9" "a" ... "Z" "_")*
	⋮

Note : les points-virgules sont en fait optionnels dans notre langage, mais utiles pour la lisibilité !

Exemples de programmes ILP1

```
print("Un, ");  
print("deux et ");  
print("trois.")
```

```
let x = 1  
and y = 2 in  
x + y
```

```
print (2.5 - 1)
```

```
if false then print("invisible");  
47
```

Voir également les fichiers [.ilpml](#) du répertoire [SamplesILP1](#).

Éléments de sémantique : les expressions

Certains **choix sémantiques** sont déjà visibles au niveau de la syntaxe !

La plus part des langages distinguent :

- les instructions, qui ont un effet et ne renvoient pas de valeur
- les expressions, qui renvoient une valeur

e.g., C, Java, JavaScript, Python, etc.

D'autres, comme Lisp, ML ou ILP ne font pas cette distinction

⇒ **tout est expression** et **renvoie une valeur** :

- `(expr1; expr2; ...; exprN)`
renvoie la valeur de `exprN`
- `if expr1 then expr2 else expr3`
renvoie la valeur de `expr2` ou `expr3` (selon la valeur de `expr1`)
- `let x = expr1 in expr2`
renvoie la valeur de `expr2`

Éléments de sémantique : les expressions

En ILP, tout est expression et renvoie une valeur.

Avantages :

- uniformité, simplicité du langage.
- expressivité; on peut écrire : `let x = if y then 1 else 2`

Inconvénient :

Certaines expressions n'ont pas de valeur naturelle :

- `if expr1 then expr2`, quand `expr1` est fausse
- `print(x)`
- ce sera aussi le cas pour les boucles `while` dans ILP2

⇒ ILP choisit de renvoyer le booléen `false`

Éléments de sémantique : portée des variables

Portée :

Les variables sont déclarées locales avec `let`.

nous verrons des variables globales dans ILP2

- les variables n'existent que dans leur **portée**
e.g., dans `let x = expr1 in expr2`,
`x` n'existe que dans `expr2`.
- une variable peut en masquer une autre
e.g. : `let x = 5 in (let x = 12 in x) + x`

Tous les langages modernes suivent ces convention !

(avec des variations sur la notion de bloc lexical, e.g. en Python...)

Éléments de sémantique : typage dynamique

Typage : de nombreux systèmes de types existent !

- typage statique avec déclaration (C, Java)
- typage statique avec inférence (ML)
- **typage dynamique** (JavaScript, Python, ILP)

En ILP, une variable n'est pas déclarée avec un type.

Elle prend le type de la valeur qui lui est affectée.

e.g., `let x = if ... then 12 else "Coucou" in print(x)`

Nous verrons en ILP2, avec l'affectation, que le type de la valeur affectée dans une variable peut varier au cours de l'exécution.

⇒ les variables n'ont pas de type, seules les valeurs ont un type.

Éléments de sémantique : sûreté du typage

Erreur de type :

utilisation d'un opérateur avec une valeur d'un type non adapté.

(e.g., additionner des booléens)

Sûreté du typage :

- typage faible : comportement indéfini (e.g., C)
- **typage fort** : détection systématique et gestion “propre” des erreurs
(e.g., Java, ML, ILP)

En ILP, toute valeur a un type bien défini.

Toutes les erreurs de type sont détectées à l'exécution.

e.g., `2 + "Coucou"` termine le programme avec un message d'erreur.

Éléments de sémantique : les opérateurs

- `+` : additionne deux nombres ou deux chaînes
- `-`, `*`, `/` : opération sur deux nombres
- `%` : modulo sur deux entiers
- `==`, `>=`, `<=`, `<`, `>`, `!=` : comparaison de deux nombres
- `&`, `|`, `^` : opération logique sur deux booléens
- `-` : opposé d'un nombre
- `!` : négation d'un booléen

“nombre” signifie ici un entier ou un flottant :

- si tous les arguments sont entiers, le résultat est entier
- si un argument au moins est flottant, le résultat est flottant

De plus, une valeur non-boléenne est considérée comme `true` pour un opérateur booléen.

cf. `com.paracampus.ilp1.interpreter.operator`

Éléments de sémantique : les primitives

Primitive : variable ou fonction prédéfinie.

- `pi` : la constante 3.1415926535
- `print(expr)` : affiche la valeur de l'expression `expr` (quel que soit son type)
- `newline()` : affiche un saut de ligne

Considéré comme une variable / fonction par l'analyseur syntaxique, reconnu spécialement par l'interprète et le compilateur.

Avantages :

- évite d'encombrer la grammaire
- très facile d'ajouter de nouvelles primitives

cf. `com.paracampus.ilp1.interpreter.primitive`

ANTLR 4

ANTLR : **générateur** d'analyseurs syntaxiques.

(cf., `lex` et `yacc` / `bison`, etc.)

- part d'une grammaire au format `.g4`
proche de la syntaxe BNF
- génère un “reconnaisseur” pour le langage de la grammaire, en Java
paramétré par une classe visiteur ou *Listener* fournie par l'utilisateur

Sources fournies et sources générées

- grammaire fournie : [ANTLRGrammars/ILPMLgrammar1.g4](#)
- ANTLR 4 génère les sources Java
 - [ILPMLgrammar1Lexer.java](#) : analyse lexicale (lettres en mots)
 - [ILPMLgrammar1Parser.java](#) : analyse syntaxique (mots en phrases)
 - [ILPMLgrammar1Listener.java](#) : interface de *Listener* à implanter
 - [ILPMLgrammar1BaseListener.java](#) : implantation vide du *Listener* à sous-classer
 - [ILPMLgrammar1Visitor.java](#) : interface de visiteur à implanter
 - [ILPMLgrammar1BaseVisitor.java](#) : implantation vide de visiteur à sous-classer

les sources sont dans `target/generated-sources` et dans le package `antlr4`

- nous fournissons également le source Java :
 - [ILPMLListener.java](#) dans `com.paracampus.ilp1.parser.ilpml`
implantation du *Listener*, i.e., de [ILPMLgrammar1Listener](#)

Grammaire ANTLR pour ILP1

ANTLRGrammars/ILPMLgrammar1.g4 (extrait)

```

grammar ILPMLgrammar1;
@header { package antlr4; }

// Règles syntaxiques
prog returns [com.paracampus.ilp1.interfaces.IASTprogram node]
    : (exprs+=expr ';'?) * EOF
    ;
expr returns [com.paracampus.ilp1.interfaces.IASTexpression node]
    : '(' exprs+=expr ';'?'?' exprs+=expr)* ';'?'?' # Sequence
    | fun=expr '(' args+=expr? '('',' args+=expr)* ')' # Invocation
    | op=('-' | '!') arg=expr # Unary
    | arg1=expr op=('*' | '/' | '%') arg2=expr # Binary
    | arg1=expr op=('+' | '-') arg2=expr # Binary
    | arg1=expr op=('<' | '<=' | '>' | '>=') arg2=expr # Binary
    | ...
    | 'let' vars+=IDENT '=' vals+=expr ('and' vars+=IDENT '=' vals+=expr)*
    | 'in' body=expr # Binding
    | 'if' condition=expr 'then' consequence=expr
    | ('else' alternant=expr)? # Alternative
    ;

// Règles lexicales
IDENT : [a-zA-Z_] [a-zA-Z0-9_]*;
INT : [0-9]+;
...
SPACE : [ \t\r\n]+ -> skip;

```

Reconnaisseur ANTLR et *Listener*

schéma de règle ANTLR

```
non_terminal returns [type attr]
: var1=expr1 ... varN=exprN # NomRègle1
| var1=expr1 ... varM=exprM # NomRègle2 ;
```

Nous utilisons l'interface *Listener* d'ANTLR.

Principe :

- ANTLR parcourt le texte du source caractère par caractère
- dès qu'une règle *NomRègle* est applicable, ANTLR appelle le *Listener*

```
enterNomRègle(NomRègleContext ctx)
exitNomRègle(NomRègleContext ctx)
```
- si une règle contient des non-terminaux
leur *enter* / *exit* sont appelés *entre* *enterNomRègle* et *exitNomRègle*
- le contexte *ctx* a un champ *attr* de type *type*
- le contexte *ctx* a des champs *var1*, ..., *varN* (dépendant de la règle)
donnant accès au contexte des sous-expressions
⇒ *ctx.attr* est calculé à partir de *ctx.var1.attr*, ..., *ctx.varN.attr*

Reconnaisseur ANTLR : exemple

exemple de grammaire

```

expr returns [Integer val]
  : e1=expr '+' e2=expr # Add
  | '-' e=expr           # Neg
  | i=INT                # Const ;

```

exemple de source

```
12 + -4
```

ANTLR appellera dans l'ordre :

```

enterAdd
enterConst
exitConst
enterNeg
enterConst
exitConst
exitNeg
exitAdd

```

ANTLR fournit les types :

```

AddContext ctx :
    Integer ctx.val, ctx.e1.val, ctx.e2.val

NegContext ctx :
    Integer ctx.val, ctx.e.val

ConstContext ctx :
    Integer ctx.val

Token ctx.i (type des terminaux ANTLR)

```

Exemple : implantation d'une calculatrice simple

```

exitAdd(AddContext ctx) { ctx.val = ctx.e1.val + ctx.e2.val; }
exitNeg(NegContext ctx) { ctx.val = -ctx.e.val; }
exitConst(ConstContext ctx) { ctx.val = Integer.parseInt(ctx.i.getText()); }

```

Priorité des règles en ANTLR

La règle BNF : $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle$
est ambiguë : $1+2*3$ peut être vu comme $1+(2*3)$ ou $(1+2)*3$.

ANTLR résout ce problème à l'aide d'un **système de priorités implicites**.

exemple : opérations binaires

```
expr returns [com.paracampus.ilp1.interfaces.IASTexpression node]
...
| arg1=expr op=('*' | '/' | '%') arg2=expr # Binary
| arg1=expr op=('+' | '-') arg2=expr # Binary
| arg1=expr op=('<' | '<=' | '>' | '>=') arg2=expr # Binary
| arg1=expr op=('==' | '!=') arg2=expr # Binary
| arg1=expr op='&' arg2=expr # Binary
| arg1=expr op=('|' | '^')
```

- les lignes en premier ont une priorité plus élevée
 $1+2*3$ est vu comme $1+(2*3)$
- tous les opérateurs d'une ligne ont la même priorité
avec associative à gauche par défaut
 $1+2-3$ est vu comme $(1+2)-3$

autre exemple : "if true then 1 + 2" est vu comme "if true then (1+2)"

Arbre syntaxique abstrait

Arbre syntaxique abstrait

Principe : représentation du programme

- **abstraite**, oubliant les détails non importants (espaces, parenthèses, sucre syntaxique)
- **non-ambiguë** (priorité des opérateurs résolue)
- **structuré** (arbre)
- facile à manipuler

Dans ILP, nous utilisons des bonnes pratiques de programmation objet :

- programmation vis à vis d'une **interface**
- **héritage** pour factoriser le code
- utilisation de *design patterns* : fabrique, visiteur.

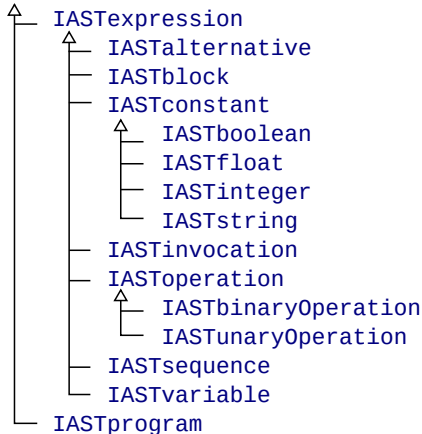
L'extension de ILP1 à ILP4, et les TME, nous donneront l'occasion d'apprécier les bénéfices de ces bonnes pratiques !

Hiérarchie d'interfaces IAST

Les interfaces de l'AST ont pour préfixe IAST et dérivent de l'interface [IAST](#).

Package [com.paracampus.ilp1.interfaces](#) :

IAST



Interfaces d'AST : `IAST`, `IASTexpression`

`IAST.java`

```
package com.paracampus.ilp1.interfaces;  
public interface IAST {  
    /* vide ! */  
}
```

Interface `marqueur` (sans méthode)
servant de racine à la hiérarchie d'IAST.

`IASTexpression.java`

```
package com.paracampus.ilp1.interfaces;  
public interface IASTexpression extends IAST, IASTvisitable {  
    /* vide ! */  
}
```

`IASTexpression` distingue les expressions des programmes complets.

ILP2 introduira des déclarations de fonctions dans les programmes ; ce ne sont pas des expressions.

`IASTvisitable` sert à implanter le *design pattern* visiteur, cf. prochain cours.

Interfaces d'AST : `IASTalternative`

— `IASTalternative.java` —

```
public interface IASTalternative extends IASTexpression {  
    IASTexpression getCondition();  
    IASTexpression getConsequence();  
    @Nullable IASTexpression getAlternant();  
    boolean isTernary();  
}
```

Modélise : *if condition then consequence else alternant.*

`@Nullable` est une **annotation** Java

indiquant que `getAlternant` peut retourner `NULL`.

en l'absence d'annotation `@Nullable`, nous pouvons supposer que la méthode ne renvoie jamais `null` ;

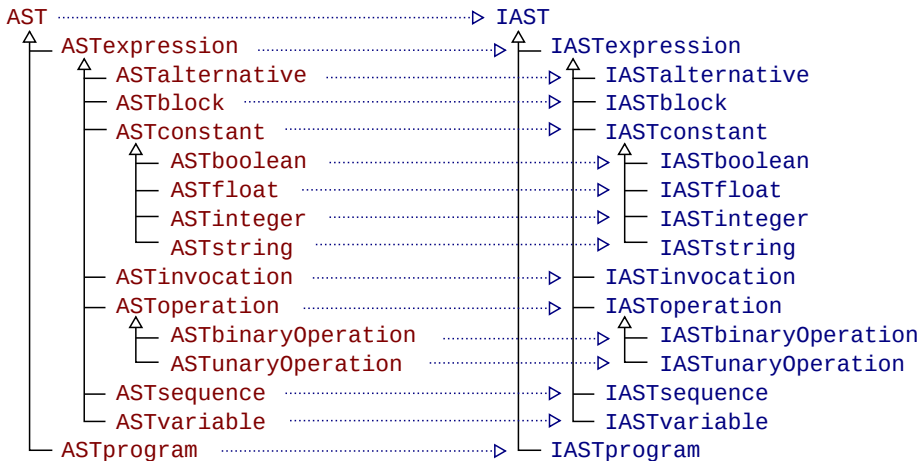
cette annotation, spécifique à ILP, n'est là qu'à titre informatif ;

elle n'est pas exploitée ni vérifiée par Java. . .

cf. [com.paracampus.ilp1.annotation.Nullable](https://com.github.paracampus.ilp1.annotation.Nullable)

Hiérarchie de classes d'AST

Package `com.paracamplus.ilp1.ast` :



Chaque classe AST implante une interface IAST correspondante, et a les mêmes relation d'héritage que l'IAST.

Classe d'AST : `ASTalternative`

`ASTalternative.java`

```
public class ASTalternative extends ASTexpression implements IASTalternative {

    private final IASTexpression condition;
    private final IASTexpression consequence;
    private @Nullable final IASTexpression alternant;

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

    @Override public IASTexpression getCondition()    { return condition; }
    @Override public IASTexpression getConsequence() { return consequence; }
    @Override @Nullable public IASTexpression getAlternant() { return alternant; }
    @Override public boolean isTernary() { return this.alternant != null; }
}
```

Simple classe conteneur pour les attributs d'un nœud alternative.

Note : les méthodes retournent une interface `IASTexpression` !

⇒ programmer vis à vis d'une interface, pas d'une implantation

Associations : IASTbinding

IASTblock.java

```
public interface IASTblock extends IASTexpression {  
  
    interface IASTbinding extends IAST {  
        IASTvariable getVariable();  
        IASTexpression getInitialisation();  
    }  
  
    IASTbinding[] getBindings();  
    IASTexpression getBody();  
}
```

Un bloc `IASTblock` contient un nombre arbitraire de variables locales i.e., d'associations variable / expression.

⇒ l'`interface locale IASTblock.ASTbinding` dénote une telle association.

Associations : ASTbinding

ASTblock.java

```
public class ASTblock extends ASTexpression implements IASTblock {

    public static class ASTbinding extends AST implements IASTbinding {
        public ASTbinding (IASTvariable variable, IASTexpression initialisation) {
            this.variable = variable;
            this.initialisation = initialisation;
        }
        private final IASTvariable variable;
        private final IASTexpression initialisation;

        @Override public IASTvariable getVariable() { return variable; }
        @Override public IASTexpression getInitialisation() { return initialisation; }
    }

    public ASTblock(IASTbinding[] binding, IASTexpression body) {
        this.binding = binding;
        this.body = body;
    }
    private final IASTbinding[] binding;
    private final IASTexpression body;

    @Override public IASTbinding[] getBindings() { return binding; }
    @Override public IASTexpression getBody() { return body; }
}
```

`ASTblock.ASTbinding` est une classe locale conteneur pour une association.

Programmes : (I)ASTprogram

IASTprogram.java

```
public interface IASTprogram extends IAST {  
    IASTexpression getBody();  
}
```

ASTprogram.java

```
public class ASTprogram extends AST implements IASTprogram {  
  
    public ASTprogram(IASTexpression expression)  
    { this.expression = expression; }  
  
    protected IASTexpression expression;  
  
    @Override public IASTexpression getBody()  
    { return this.expression; }  
}
```

Pour l'instant : un simple conteneur pour une expression.

Fabrique d'AST 1/2

IASTfactory.java

```
public interface IASTfactory {

    IASTexpression newIntegerConstant(String value);
    IASTexpression newFloatConstant(String value);
    IASTexpression newStringConstant(String value);
    IASTexpression newBooleanConstant(String value);
    IASTexpression newSequence(IASTexpression[] asts);
    IASTexpression newInvocation(IASTexpression function, IASTexpression[] arguments);
    IASTexpression newBlock(IASTbinding[] binding, IASTexpression body);
    IASTexpression newUnaryOperation(IASToperator operator, IASTexpression operand);
    IASTexpression newBinaryOperation(
        IASToperator operator, IASTexpression leftOperand, IASTexpression rightOperand);
    IASTexpression newAlternative(
        IASTexpression condition, IASTexpression consequence, IASTexpression alternant);

    IASTprogram newProgram(IASTexpression expression);
    IASToperator newOperator(String name);
    IASTvariable newVariable(String name);
    IASTbinding newBinding(IASTvariable v, IASTexpression exp);
}
```

Design pattern *factory method* :

- une interface **IASTfactory** indique comment créer tous les types d'AST

Fabrique d'AST 2/2

ASTfactory.java

```
public class ASTfactory implements IASTfactory {  
    @Override public IASTprogram newProgram(IASTexpression expression)  
    { return new ASTprogram(expression); }  
  
    @Override public IASToperator newOperator(String name)  
    { return new ASToperator(name); }  
  
    @Override public IASTsequence newSequence(IASTexpression[] asts) {  
    { return new ASTsequence(asts); }  
    ...  
}
```

Design pattern *factory method* :

- une classe concrète **ASTfactory** implante l'interface par **new**
 - un client d'AST est paramétré par une instance d'**IASTfactory**
 - il n'appelle pas **new** directement
 - il ne référence aucune classe concrète, seulement des interfaces
- ⇒ encapsulation de la création d'AST, indépendance et extensibilité
(cf. transparent suivant)

Construction de l'AST via le *Listener* ANTLR

ILPMLListener.java

```
package com.paracampus.ilp1.parser.ilpml;

public class ILPMLListener implements ILPMLGrammar1Listener {

    protected IASTfactory factory;

    public ILPMLListener(IASTfactory factory) {
        super();
        this.factory = factory;
    }

    @Override public void exitVariable(VariableContext ctx) { ... }
    @Override public void enterVariable(VariableContext ctx) { }
    ...
}
```

- implantation de l'interface `ILPMLGrammar1Listener` générée par ANTLR
- classe paramétrée par une fabrique `IASTfactory`
- une méthode `enterXXX` et `exitXXX` par règle de grammaire `XXX`
- les méthodes `enterXXX` sont vides
- les méthodes `exitXXX` font tout le travail (transparents suivants)

Construction de l'AST, exemple 1/3

Exemple : opération binaire dans une expression.

ILPMLgrammar1.g4

```
expr returns [com.paracampus.ilp1.interfaces.IASTexpression node]
    ...
    | arg1=expr op=('*' | '/' | '%') arg2=expr # Binary
```

ILPMLListener.java

```
@Override
public void exitBinary(BinaryContext ctx) {
    ctx.node = factory.newBinaryOperation(
        factory.newOperator(ctx.op.getText()),
        ctx.arg1.node,
        ctx.arg2.node);
}
```

- récupère le texte de l'opération : `ctx.op.getText()`
- récupère l'AST des sous-expressions : `ctx.arg1.node` et `ctx.arg2.node`
- construit le nouveau nœud AST : `factory.newBinaryOperation(...)`
- stocke le résultat dans le nœud de grammaire courant `ctx.node`
 \implies il sera disponible lors de l'application des règles englobantes

Construction de l'AST, exemple 2/3

Exemple : alternative.

ILPMLgrammar1.g4

```
| 'if' condition=expr 'then' consequence=expr
  ('else' alternant=expr)? # Alternative
```

ILPMLListener.java

```
@Override
public void exitAlternative(AlternativeContext ctx) {
    ctx.node = factory.newAlternative(
        ctx.condition.node,
        ctx.consequence.node,
        (ctx.alternant == null ? null : ctx.alternant.node));
}
```

- penser au cas où la branche else optionnelle est absente
`ctx.alternant == null`

Construction de l'AST, exemple 3/3

Exemple : séquence d'instructions.

ILPMLgrammar1.g4

```
: '(' exprs+=expr (';'? exprs+=expr)* ';'? ')' # Sequence
```

ILPMLListener.java

```
protected IASTExpression[] toExpressions(List<ExprContext> ctxs) {
    if (ctxs == null) return new IASTExpression[0];
    IASTExpression[] r = new IASTExpression[ctxs.size()];
    int pos = 0;
    for (ExprContext e : ctxs) r[pos++] = e.node;
    return r;
}

@Override
public void exitSequence(SequenceContext ctx) {
    ctx.node = factory.newSequence(toExpressions(ctx.exprs));
}
```

- ANTLR se charge d'accumuler toutes les expressions dans une liste de contextes `ctxs`, grâce à `exprs+=expr`
- l'utilitaire `toExpressions` convertit la liste en tableau et extrait le nœud de chaque contexte

Pilote d'analyse syntaxique ILPML

ILPMLParser.java

```
protected IASTfactory factory;
public ILPMLParser(IASTfactory factory) { this.factory = factory; }

protected Input input;
public void setInput(Input input) { this.input = input; }

public IASTprogram getProgram() throws ParseException {
    try {
        ANTLRInputStream in = new ANTLRInputStream(input.getText());
        // flux de caractères -> analyseur lexical
        ILPMLgrammar1Lexer lexer = new ILPMLgrammar1Lexer(in);
        // analyseur lexical -> flux de tokens
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // flux de tokens -> analyseur syntaxique
        ILPMLgrammar1Parser parser = new ILPMLgrammar1Parser(tokens);
        // démarrage de l'analyse syntaxique
        ILPMLgrammar1Parser.ProgContext tree = parser.prog();
        // parcours de l'arbre syntaxique et appel du Listener
        ParseTreeWalker walker = new ParseTreeWalker();
        ILPMLListener extractor = new ILPMLListener(factory);
        walker.walk(extractor, tree);
        return tree.node;
    } catch (Exception e) { throw new ParseException(e); }
}
```

Appelle les services ANTLR dans le bon ordre, en passant notre *Listener*.
setInput spécifie la source de l'entrée texte (chaîne, fichier, etc.).

Pilote d'analyse syntaxique générique

Parser.java

```
public class Parser {  
  
    protected ILPMLParser ilpmlparser;  
    public void setILPMLParser(ILPMLParser parser) { this.ilpmlparser = parser; }  
  
    public IASTprogram parse(File file) throws ParseException {  
        Input input = new InputFromFile(file);  
        ...  
        if (file.getName().endsWith(".ilpml")) {  
            if (ilpmlparser == null)  
                throw new ParseException("ILPML parser not set");  
            ilpmlparser.setInput(input);  
            IASTprogram program = ilpmlparser.getProgram();  
            return program;  
        }  
        throw new ParseException("file extension not recognized");  
    }  
}
```

ILP supporte en réalité plusieurs analyseurs syntaxiques : ILPML et XML !

Parser choisit le bon analyseur en fonction de l'extension (.ilpml ou .xml).

Parser est paramétré par l'analyseur ILPML (celui-ci changera pour ILP2, etc.)

De l'arbre syntaxique abstrait à la syntaxe concrète

But : transformation inverse, de l'AST en programme source ILPML

- utile pour le débogage des transformations d'AST (cf. TME 6)
- traduction d'une syntaxe à l'autre (e.g., XML ↔ ILPML)

Principe :

- `MLPrinter`, dans `com.paracamplus.ilp1.tools`
classe utilitaire pour afficher du texte bien indenté
 - `insertBreak()` : saut de ligne et indentation
 - `openParen(String)` / `closeParen(String)` : parenthésage avec mise à jour du niveau d'indentation
- `ILPMLProgramPrinter`, dans `com.paracamplus.ilp1.parser.ilpml`
visiteur qui parcourt l'AST récursivement pour l'afficher
nous devons appeler :
 - `setInput(IASProgram)`
 - `visit(IASProgram, MLprinter)`
 - `getString()`

Principes pour l'extension de la syntaxe

L'extension du langage commence par l'extension de la syntaxe d'entrée :

- si des nouveaux nœuds AST sont nécessaires :
 - ajouter les interfaces **IAST** et les classes **AST**
 - enrichir la **fabrique IASTfactory** / **ASTfactory** (par héritage)
- écrire un nouveau fichier de **grammaire ANTLR .g4**
- écrire un nouveau **Listener**
obéissant à l'interface produite par ANTLR pour la nouvelle grammaire
copier-coller depuis **ILPMLgrammar1.g4** et **ILPMLListener.java** si nécessaire ;
l'héritage de ces fichiers est difficile à cause de limitations d'ANTLR !
- hériter d'**ILPMLParser** en spécifiant notre nouveau **Listener**
et notre nouvelle fabrique
- écrire des exemples de tests utilisant la nouvelle syntaxe
et vérifier que le comportement des anciens programmes n'a pas changé !

Toujours **travailler dans un nouveau package Java**.

Ne jamais modifier les fichiers existants ;
l'ancienne grammaire doit rester accessible !

cf. détails dans le TME 1.

Syntaxe XML (bonus)

Les anciennes versions d'ILP utilisent une syntaxe **XML**.
L'analyseur syntaxique XML est toujours inclus. . .

Dans le répertoire **XMLGrammars** :

- **grammar1.rnc** : grammaire XML au format **RelaxNG**
automatiquement transformé en fichier **grammar1.rng**

Dans le package **com.paracampus.ilp1.parser.xml** :

- **AbstractExtensibleParser** :
analyseur syntaxique générique XML basé sur la représentation **DOM**
utilisant la **réflexivité** Java pour donner un aspect déclaratif
- **XMLParser extends AbstractExtensibleParser**
instance d'analyse syntaxique adaptée au langage ILP1
- **XMLProgramPrinter** :
conversion inverse, d'AST en XML
très utile au débogage : le format XML est très proche de la représentation interne de l'AST

Syntaxe XML : exemple (bonus)

exemple .ilpml

```
if true then print("invisible");
48
```

exemple .xml correspondant

```
<program>
  <sequence>
    <alternative>
      <condition>
        <boolean value='true' />
      </condition>
      <consequence>
        <invocation>
          <function>
            <variable name='print' />
          </function>
          <arguments>
            <string>invisible</string>
          </arguments>
        </invocation>
      </consequence>
    </alternative>
    <integer value='48' />
  </sequence>
</program>
```

À regarder chez soi : l'analyseur syntaxique XML et les exemples .xml dans SamplesILP1/

Tests

Principe

But du test :

- vérifier qu'un programme vérifie sa spécification
e.g. : pas d'exception au *top-level*, retourne bien le résultat attendu
- vérifier l'absence de régression (évolution du programme)

L'exécution d'un programme ILP a deux résultats :

- la valeur de retour
(un programme est une expression, et toute expression ILP a une valeur)
- le texte affiché sur la sortie standard par `print` et `newline`

Principe du test ILP :

On se donne un ensemble de triplets de fichiers :

- `test.ilpml` : programme source ILP
- `test.result` : valeur de retour attendue
- `test.print` : sortie texte attendue

cf. le répertoire `SamplesILP1`

puis, on lance le compilateur ou l'interprète sur chaque `test.ilpml`
pour vérifier que les sorties sont identiques à `test.result` et `test.print`.

JUnit 4

JUnit 4 : *framework* de test unitaire utilisant les annotations Java.

Une classe de test doit :

- contenir un constructeur sans argument
- contenir une ou plusieurs **méthodes de test** :
 - annotées avec `@Test`
 - public et sans argument (donc bien différente d'une méthode `main`)
- utiliser des assertions de la bibliothèque JUnit 4
`assertTrue`, `assertFalse`, `assertEqual`, etc.
- éventuellement des méthode `@Before` et `@After`

Il est également possible de lancer un test sur une **famille de données** :

- une méthode `static @Parameters` fournit une liste de paramètres
- un constructeur avec argument est alors utilisé

JUnit 4 : principe d'exécution

Le lancement du test peut se faire avec "*Run As*" puis "*JUnit Test*" sous Eclipse.

Pour chaque méthode `@Test`
et éventuellement chaque valeur de paramètre,
JUnit va :

- instancier la classe
- appeler les méthode `@Before`
- appeler la méthode de test
l'exécution du test s'arrête au premier échec d'assertion
- appeler les méthodes `@After`

Eclipse affiche le nombre de tests sans aucune erreur.

Test d'ILP : exemple de l'interprète

InterpreterTest.java (extrait)

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

    public InterpreterTest(final File file) { this.file = file; }
    protected File file;

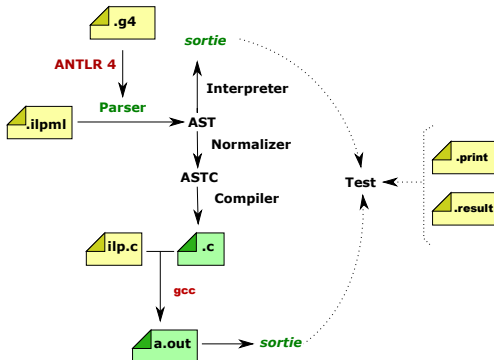
    @Test public void processFile() throws ... {
        InterpreterRunner run = new InterpreterRunner();
        configureRunner(run);
        run.testFile(file);
        run.checkPrintingAndResult(file);
    }

    public void configureRunner(InterpreterRunner run) throws EvaluationException {
        IASTfactory factory = new ASTfactory();
        run.setILPMLParser(new ILPMLParser(factory));
        ...
    }
}
```

La classe utilitaire **InterpreterRunner** automatise :

- l'extraction de la liste de fichiers sources (`getFileList`)
- l'appel à l'analyseur syntaxique et l'interprète (`testFile`)
- la comparaison entre le résultat attendu et obtenu (`checkPrintingAndResult`)

Résumé



Nous avons vu :

- syntaxe concrète d'ILP1
- encodage en grammaire ANTLR 4
- actions de grammaire par *Listener*
- représentation de programmes avec l'arbre syntaxique abstrait
- choix sémantiques d'ILP1
- test automatisé JUnit 4

Prochain cours :
l'interprète ILP1 en Java.