

Analyse syntaxique

David Delahaye

David.Delahaye@lirmm.fr

Faculté des Sciences

Master M1 2019-2020



Introduction

Analyse syntaxique

- Passer de la syntaxe concrète à la syntaxe abstraite ;
- Vérifier qu'un programme est bien formé ;
- Syntaxe abstraite = arbre ;
- AST = « Abstract Syntax Tree ».

De nombreux outils

- Lex et Yacc (Flex et Bison) ;
- Pour OCaml : ocamllex, ocamlyacc, Menhir, Camlp4 ;
- ANTLR (que nous utiliserons).

ANTLR

Qu'est-ce que ANTLR ?

- Générateur de parseurs (T. Parr) ;
- Très « Java-friendly » ;
- Plusieurs langages cibles : Java, C#, JavaScript, Python ;
- Un seul fichier de spécification de la grammaire ;
- Mélange de règles lexicales et grammaticales.

Obtenir ANTLR

- Ici : <http://www.antlr.org/> ;
- Installation triviale (un simple « .jar »).

Un premier petit exemple

Grammaire « Hello » (fichier « Hello.g4 »)

```
grammar Hello;  
  
r  : 'hello' ID ;  
ID : [a-z]+ ;  
WS : [ \t\r\n]+ -> skip ;
```

Compilation

```
$ antlr4 Hello.g4  
$ javac Hello*.java
```

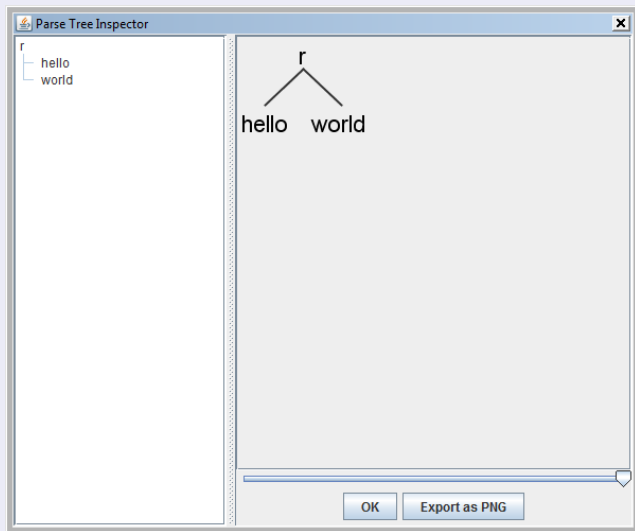
Un premier petit exemple

Exécution

```
$ grun Hello r -tree  
hello world  
^D  
(r hello world)  
$ grun Hello r -gui  
hello world  
^D
```

Un premier petit exemple

Affichage graphique de l'AST



Un autre exemple plus complexe

Expressions arithmétiques

- Expressions arithmétiques entières ;
- Constantes entières ;
- Opérateurs binaires : $+$, $-$, \times , $/$;
- Opérateur unaire : $-$;
- Attention aux précédences :
 - ▶ $\{-\}$ (un.) $>$ $\{\times, /\}$ $>$ $\{+, -\}$.
- En ANTLR, les précédences doivent être gérées à la main.

Un autre exemple plus complexe

La grammaire « ExprArith »

```
grammar ExprArith;

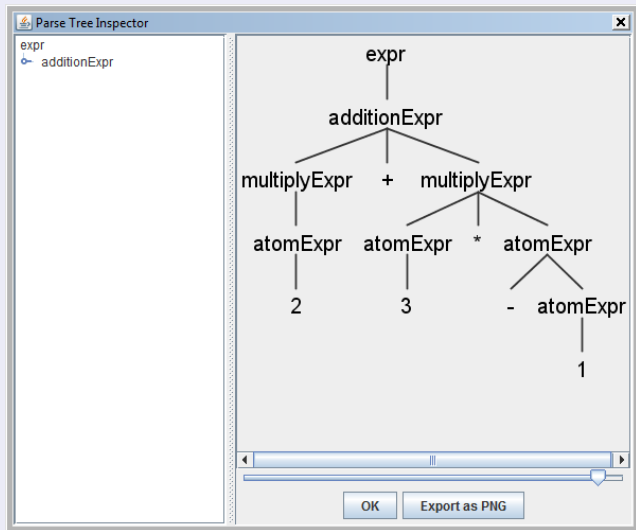
expr : additionExpr ;
additionExpr : multiplyExpr
    ('+' multiplyExpr | '-' multiplyExpr)* ;
multiplyExpr : atomExpr ('*' atomExpr | '/' atomExpr)* ;
atomExpr : Number | '(' additionExpr ')' | '-' atomExpr ;
Number : ('0'..'9')+ ;
WS : [ \t\r\n]+ -> skip ;
```

Exécution

```
$ grun ExprArith expr -gui
2+3*-1
^D
```


Un autre exemple plus complexe

AST obtenu



Le même exemple mais avec des actions

La grammaire « ExprArithEval » (calculatrice)

```
grammar ExprArithEval;

expr returns [int value] :
    e = additionExpr {$value = $e.value;} ;

additionExpr returns [int value] :
    e1 = multiplyExpr {$value = $e1.value;}
    ('+' e2 = multiplyExpr {$value += $e2.value;}
    | '-' e2 = multiplyExpr {$value -= $e2.value;}) * ;

multiplyExpr returns [int value] :
    e1 = atomExpr {$value = $e1.value;}
    ('*' e2 = atomExpr {$value *= $e2.value;}
    | '/' e2 = atomExpr {$value /= $e2.value;}) * ;
```

Le même exemple mais avec des actions

La grammaire « ExprArithEval » (calculatrice)

```
atomExpr returns [int value] :  
    c = Number {$value = Integer.parseInt($c.text);}  
    | '(' e1 = additionExpr ')' {$value = $e1.value;}  
    | '-' e2 = atomExpr {$value = -$e2.value;} ;  
  
Number : ('0'..'9')+ ;  
  
WS : [ \t\r\n]+ -> skip ;
```

Le même exemple mais avec des actions

Programme de test

```
import org.antlr.v4.runtime.*;

public class ExprArithEvalTest {

    public static void main(String [] argv) {
        ANTLRInputStream stream =
            new ANTLRInputStream(argv[0]);
        ExprArithEvalLexer lexer =
            new ExprArithEvalLexer(stream);
        CommonTokenStream tokens =
            new CommonTokenStream(lexer);
        ExprArithEvalParser parser =
            new ExprArithEvalParser(tokens);
        System.out.println(parser.expr().value);
    }
}
```

Le même exemple mais avec des actions

Compilation et exécution

```
$ javac ExprArithEvalTest.java
$ java ExprArithEvalTest "2+3"
5
$ java ExprArithEvalTest "2+3*-1"
-1
```

Le même exemple mais avec un AST

AST des expressions arithmétiques

- Programmation orientée objets :
 - ▶ L'AST est une hiérarchie de classes (héritage);
 - ▶ Le code est attachée aux données (fonction d'évaluation).

```
abstract class ExprArith {  
  
    abstract int eval ();  
  
}
```

Le même exemple mais avec un AST

AST des expressions arithmétiques

```
class Cte extends ExprArith {  
  
    int val;  
  
    Cte (int val) {  
        this.val = val;  
    }  
  
    int eval () {  
        return val;  
    }  
}
```

Le même exemple mais avec un AST

AST des expressions arithmétiques

```
class Inv extends ExprArith {  
  
    ExprArith e;  
  
    Inv (ExprArith e) {  
        this.e = e;  
    }  
  
    int eval () {  
        return -e.eval();  
    }  
}
```


Le même exemple mais avec un AST

AST des expressions arithmétiques

- Factorisation possible sur les opérations binaires ;
- On pourrait aussi factoriser la fonction binaire de calcul.

```
abstract class BinOp extends ExprArith {  
  
    ExprArith e1, e2;  
  
}
```

Le même exemple mais avec un AST

AST des expressions arithmétiques

```
class Add extends BinOp {  
  
    Add (ExprArith e1, ExprArith e2) {  
        this.e1 = e1;  
        this.e2 = e2;  
    }  
  
    int eval () {  
        return e1.eval() + e2.eval();  
    }  
  
} [...]
```

Le même exemple mais avec un AST

La grammaire « ExprArithAST »

```
grammar ExprArithAST;
```

```
expr returns [ExprArith value] :
```

```
    e = additionExpr {$value = $e.value;} ;
```

```
additionExpr returns [ExprArith value] :
```

```
    e1 = multiplyExpr {$value = $e1.value;}
```

```
    ('+' e2 = multiplyExpr
```

```
        {$value = new Add($value, $e2.value);}
```

```
    | '-' e2 = multiplyExpr
```

```
        {$value = new Sub($value, $e2.value);}) * ;
```

```
multiplyExpr returns [ExprArith value] :
```

```
    e1 = atomExpr {$value = $e1.value;}
```

```
    ('*' e2 = atomExpr {$value = new Mul($value, $e2.value);}
```

```
    | '/' e2 = atomExpr
```

```
        {$value = new Div($value, $e2.value);}) * ;
```

Le même exemple mais avec un AST

La grammaire « ExprArithAST »

```
atomExpr returns [ExprArith value] :  
    c = Number {$value = new Cte(Integer.parseInt($c.text));}  
    | '(' e1 = additionExpr ')' {$value = $e1.value;}  
    | '-' e2 = atomExpr {$value = new Inv($e2.value);} ;  
  
Number : ('0'..'9')+ ;  
  
WS : [ \t\r\n]+ -> skip ;
```

Le même exemple mais avec un AST

Programme de test

```
import org.antlr.v4.runtime.*;

public class ExprArithASTTest {

    public static void main(String [] argv) {
        ANTLRInputStream stream =
            new ANTLRInputStream(argv[0]);
        ExprArithASTLexer lexer =
            new ExprArithASTLexer(stream);
        CommonTokenStream tokens =
            new CommonTokenStream(lexer);
        ExprArithASTParser parser =
            new ExprArithASTParser(tokens);
        ExprArith e = parser.expr().value;
        System.out.println(e.eval());
    }
}
```

Le même exemple mais avec un AST

Compilation et exécution

```
$ javac ExprArithASTTest.java
$ java ExprArithASTTest "2+3"
5
$ java ExprArithASTTest "2+3*-1"
-1
```

Reconnaître des listes

La grammaire « ExprArithList »

```
grammar ExprArithList;

listExpr returns [ArrayList<ExprArith> value]
@init{$value = new ArrayList<ExprArith>();} :
    (e = expr {$value.add($e.value);})+ ;

expr returns [ExprArith value] :
    e = additionExpr {$value = $e.value;} ;

...

```

Reconnaître des listes

Programme de test

```
public class ExprArithListTest {

    public static void main(String [] argv) {
        ANTLRInputStream stream =
            new ANTLRInputStream(argv[0]);
        ExprArithListLexer lexer =
            new ExprArithListLexer(stream);
        CommonTokenStream tokens =
            new CommonTokenStream(lexer);
        ExprArithListParser parser =
            new ExprArithListParser(tokens);
        ArrayList<ExprArith> l = parser.listExpr().value;
        for (ExprArith e : l)
            System.out.println(e.eval());
    }
}
```


Reconnaître des listes

Compilation et exécution

```
$ javac ExprArithListTest.java
$ java ExprArithListTest "1+1 2*3"
2
6
```