

TP 6 : Analyse syntaxique avec ANTLR

ANTLR est un logiciel écrit en Java qui génère des analyseurs lexicaux et syntaxiques (et aussi des parcours d'arbres) à partir de fichiers de spécification. Plusieurs langages de programmation sont supportés pour les fichiers générés, dont bien sûr Java.

1 Écriture de grammaires avec ANTLR

Les analyseurs syntaxiques générés par ANTLR sont de type récursif descendant, avec une résolution par *lookahead* non borné dite $LL(*)$, et la possibilité de faire du *backtrack* en cas de mauvais choix lors d'une résolution. Du fait de cette stratégie d'analyse, la classe de grammaires acceptées n'est pas celle des grammaires algébriques (la récursivité gauche est à prohiber, et en cas d'ambiguïté, seule une analyse est retournée), mais reste suffisamment expressive pour nos besoins assez limités. De plus, ANTLR permet d'écrire des productions de grammaires contenant des opérateurs rationnels dans les parties droites.

1.1 Grammaire d'expressions

Le fichier `src/antlr/Expr.g` contient une grammaire d'expressions arithmétiques équivalente à la grammaire algébrique suivante :

$$\begin{aligned}\langle prog \rangle &\rightarrow \langle prog \rangle \langle stat \rangle \\ &\quad | \langle stat \rangle \\ \langle stat \rangle &\rightarrow \langle expr \rangle nl \\ &\quad | nl \\ \langle expr \rangle &\rightarrow \langle expr \rangle * \langle atom \rangle \\ &\quad | \langle atom \rangle \\ \langle atom \rangle &\rightarrow int \\ &\quad | (\langle expr \rangle)\end{aligned}$$

où *nl* dénote un caractère de fin de ligne et *int* un entier en base 10. Cette grammaire algébrique est traduite pour ANTLR par les règles

```
prog
: stat+
;

stat
: expr NL
| NL
;

expr
: atom ('*' atom)*
```

```

;

atom
: INT
| '(' expr ')'
;

```

Le fichier contient en sus de ces règles des règles pour l'analyseur lexical

```

INT: ('1'..'9')('0'..'9')*;
NL:  '\r'? '\n';
WS:  (' ' | '\t' | '\r' | '\n')+      { skip(); };

```

qui permettent d'identifier les entiers et les retours à la ligne, et d'éliminer les espaces. Enfin, le fichier contient un préambule

```
grammar Expr;
```

qui définit le nom de la grammaire, et diverses options, ici

```

@header {
package fr.ens_cachan.dptinfo.tp06;
}
@lexer::header {
package fr.ens_cachan.dptinfo.tp06;
}

```

qui ajouteront le texte en question au début des fichiers générés pour l'analyseur syntaxique et l'analyseur lexical respectivement, tandis que le contenu de

```

@members {
    ...
}

```

est ajouté dans la classe de l'analyseur syntaxique (ici pour redéfinir le traitement des messages d'erreur afin de les rendre plus explicites).

La commande **ant gen** va compiler tous les fichiers ANTLR présents dans le répertoire `src/antrl` (voir dans `antlr.xml` comment cette tâche **ant** est réalisée). Le résultat est l'apparition de deux fichiers `ExprParser.java` et `ExprLexer.java` dans le répertoire `src/main/fr/ens_cachan/dptinfo/tp06/`. Il ne reste plus qu'à utiliser ces fichiers; le fichier `Main.java` contient :

```

package fr.ens_cachan.dptinfo.tp06;
import java.io.*;
import org antlr.runtime.*;

public class Main {
    public static void main(String args[]) throws Exception {
        ExprLexer lex = new ExprLexer(new ANTLRFileStream(args[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);
        ExprParser parser = new ExprParser(tokens);
        parser.prog(); // launch parsing
    }
}

```

1.2 Exercice

Modifier la grammaire `Expr.g` pour permettre des affectations de valeurs à des variables et des additions et soustractions binaires. La commande `ant` vous permet de tester votre analyseur contre le fichier (incorrect) `samples/program.expr`.

2 Actions

Notre analyseur syntaxique ne fait pour l'instant pas grand-chose. ANTLR permet d'associer des calculs aux règles de grammaires (ou des réécritures vers des arbres, ce que nous verrons dans la section suivante).

2.1 Attributs synthétisés

Par exemple, on pourrait associer une valeur de retour entière à chaque expression en décorant nos règles par

```
expr returns [int value]
: e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
;

atom returns [int value]
: INT {$value = Integer.parseInt($INT.text);}
| '(' expr ')' {$value = $expr.value;}
;
```

Il ne reste alors plus qu'à afficher le résultat de chaque instruction par

```
stat
: expr NL {System.out.println($expr.value);}
| NL
;
```

2.2 Attributs hérités

On peut aussi décorer des règles avec des attributs hérités, par exemple avec

```
expr [Map<String,Integer> ids] returns [int value]
: e=atom[ids] {$value = $e.value;} ('*' e=atom[ids] {$value *= $e.value;})*
;

atom [Map<String,Integer> ids] returns [int value]
: INT {$value = Integer.parseInt($INT.text);}
| '(' expr[ids] ')' {$value = $expr.value;}
;
```

pour propager une table des identifiants de haut en bas dans l'arbre de dérivation en cours d'analyse (en l'occurrence, il serait plus simple d'ajouter le membre `Map<String,Integer> ids` à la classe `ExprParser` via l'option `@members`).

2.3 Exercice

Compléter votre grammaire pour calculer et afficher les résultats des expressions de `samples/program.expr`.

3 Construction d'arbres

Un autre mode d'utilisation d'ANTLR est celui de construction d'arbres à l'issue de l'analyse syntaxique. Ce mode est activé par l'option

```
options {  
    output=AST;  
}
```

dans le préambule du fichier (AST pour *Abstract Syntax Tree*, qui est l'utilisation la plus courante de ce mode).

3.1 Règles de réécriture

Ce mode s'utilise en indiquant quels éléments vont servir de nœuds parents dans l'arbre de sortie :

```
prog  
    : (stat {System.out.println($stat.tree.toStringTree());})+  
    ;  
  
stat  
    : expr NL -> expr  
    | NL      ->  
    ;  
  
expr  
    : atom ('*' atom)* -> ^('*' atom*)  
    ;  
  
atom  
    : INT  
    | '(' expr ')' -> expr  
    ;
```

qui résulte en des analyses comme `(* 3 (* 4 2))` pour l'entrée `(3) * (4 * 2)`.

De manière presque équivalente, on peut désigner un symbole comme parent de la règle en le faisant suivre par « `^` », et désigner un symbole comme inutile en le faisant suivre par « `!` ». Par exemple,

```
stat  
    : expr NL!  
    | NL!  
    ;
```

```

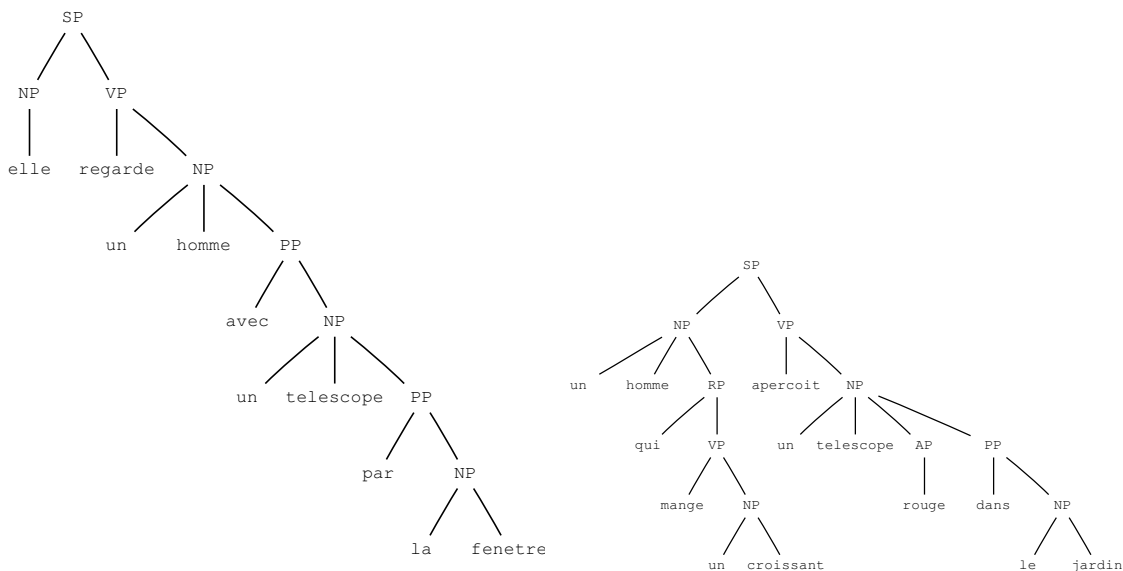
expr
: atom ('*' ^ atom)*
;

atom
: INT
| '(' ! expr ')' !
;

```

3.2 Un peu de langue naturelle

1. Le fichier `src/antlr/NLP.g` contient un embryon de grammaire pour le français. Compléter cette grammaire de manière à permettre les analyses suivantes des phrases du fichier `samples/text.txt` :



2. On souhaite raffiner l'analyse lexicale en plusieurs passes. La première, dite de *stemming*, se contente d'identifier les mots dans ANTLR par les règles

```

WORD: ('a'..'z'|'A'..'Z'|'-'')+;
WS:   (~('a'..'z'|'A'..'Z'|'.'|'-''))+ { skip(); };

```

La seconde phase identifie les catégories lexicales des mots, par exemple grâce à une table de hashage

```

categories = new HashMap<String,Integer>();
// ADJ

```

```
categories.put("rouge", ADJ);
// DET
categories.put("le", DET);
categories.put("la", DET);
categories.put("l", DET);
categories.put("un", DET);
categories.put("une", DET);
// NOUN
categories.put("homme", NOUN);
categories.put("femme", NOUN);
categories.put("telescope", NOUN);
categories.put("jardin", NOUN);
categories.put("fenetre", NOUN);
categories.put("croissant", NOUN);
// PREP
categories.put("avec", PREP);
categories.put("par", PREP);
categories.put("dans", PREP);
// PRO
categories.put("il", PRO);
categories.put("elle", PRO);
// REL
categories.put("qui", REL);
// VERB
categories.put("regarde", VERB);
categories.put("aperçoit", VERB);
categories.put("mange", VERB);
```

Modifier votre fichier `NLP.g` pour permettre cette identification en deux phases.

4 Syntaxe concrète des scénarios

Pour les plus courageux, le fichier `samples/demo.scn` contient une version textuelle du scénario de jeu `demo.xml` du projet. Écrire une grammaire ANTLR correspondante.