Мігістерство освіти і науки України
ДВНЗ «Прикарпатський національний університет імені Василя Стефаника»
Фізико-технічний факультет
Кафедра комп’ютерної інженерії та електроніки

Самостійна робота
з курсу «Системне програмування»
на тему: «Генератор парсерів ANTLR (Java, Python)»

Виконав:
студент групи КІ-31
Бабій Богдан

.

Івано-Франківськ 2020

ЗМІСТ

ВСТУП

ANTLR (вимовляється antler (мурашник)) або Another Tool for Language Recognition - це генератор парсерів, який використовує граматики LL(*) для синтаксичного розбору. ANTLR є спадкоємцем набору **Purdue Compiler Construction Tool Set** (**PCCTS**), вперше розробленого в 1989 році, і зараз активно розвивається. ANTLR розробив професор Теренс Парр з Університету Сан-Франциско. ANTLR приймає як вхід граматику, яка визначає мову та генерує вихідний код для розпізнавача цієї мови. Версія ANTLR 3 генерує код на мовах програмування: Ada95, C, C#, Java, JavaScript, Objective-C, Perl, Python, Ruby, і Standard ML. Поточна версія підтримує такі мови як: Java, C#, C++, JavaScript, Python, Swift, і Go.

Мова визначається за допомогою безконтекстної граматики, записаної у формі Extended Backus-Naur Form (EBNF). ANTLR може генерувати лексери, аналізатори, парсери дерев та комбіновані лексери-парсери. Парсери можуть автоматично генерувати дерева розбору або абстрактні синтаксичні дерева, які можна додатково обробити за допомогою парсерів дерев. ANTLR забезпечує єдині послідовні позначення для визначення лексерів, аналізаторів та парсерів дерев. За замовчуванням ANTLR зчитує граматику і генерує розпізнавач для мови, визначеної граматикою (тобто програма, яка читає вхідний потік і генерує помилку, якщо вхідний потік не відповідає синтаксису, визначеному граматикою). Якщо немає синтаксичних помилок, то за замовчуванням програма просто виходить без жодного повідомлення. Для того, щоб зробити щось корисне з мовою, до граматичних елементів у граматиці можна додати дії. Ці дії записуються мовою програмування, на якій створюється розпізнавач. Коли генерується розпізнавач, дії вбудовуються у вихідний код розпізнавача у відповідних точках. Дії можна використовувати для складання та перевірки таблиць символів та для передачі інструкцій цільовою мовою до компілятора.

РОЗДІЛ 1. ЛЕКСИКА

Лексика ANTLR знайома більшості програмістів, оскільки вона слідує синтаксису C та його похідних з деякими розширеннями для граматичних описів.

1.1. Коментарі

Є однорядкові, багаторядкові та коментарі в стилі Javadoc:

```
/** This grammar is an example illustrating the three kinds
 * of comments.
 */
grammar T;
/* a multi-line
  comment
*/
/** This rule matches a declarator for my language */
decl : ID ; // match a variable name
```

Коментарі Javadoc приховані від аналізатора і на даний момент ігноруються. Вони призначені для використання лише на початку граматики та будь-якого правила.

1.2. Ідентифікатори

Імена токенів завжди починаються з великої літери, як визначено методом Java Character.isUpperCase. Назви правил парсера завжди починаються з малої літери (окрім не працюючих). Початковий символ може супроводжуватися великими і малими літерами, цифрами та підкресленнями. Ось кілька прикладів назв:

```
ID, LPAREN, RIGHT_CURLY // token names/rules
expr, simpleDeclarator, d2, header_file // rule names
```

Як і Java, ANTLR приймає символи Unicode в іменах ANTLR:

```
grammar 外;
a : '外' ;
```

Для підтримки парсерів і імен правил лексера на Unicode, ANTLR використовує таке правило:

```
ID : a=NameStartChar NameChar*
    {
    if ( Character.isUpperCase(getText().charAt(0)) ) setType(TOKEN_REF);
    else setType(RULE_REF);
    }
  ;
```

Правило NameChar визначає дійсні символи ідентифікатора:

```
fragment
NameChar
  : NameStartChar
  | '0'..'9'
  | '_'
  | '\u00B7'
  | '\u0300'..'\u036F'
  | '\u203F'..'\u2040'
  ;
fragment
NameStartChar
  : 'A'..'Z' | 'a'..'z'
  | '\u00C0'..'\u00D6'
  | '\u00D8'..'\u00F6'
  | '\u00F8'..'\u02FF'
  | '\u0370'..'\u037D'
  | '\u037F'..'\u1FFF'
  | '\u200C'..'\u200D'
  | '\u2070'..'\u218F'
  | '\u2C00'..'\u2FEF'
  | '\u3001'..'\uD7FF'
  | '\uF900'..'\uFDCF'
  | '\uFDF0'..'\uFFFD'
  ;
```

Правило NameStartChar - це список символів, які можуть запустити ідентифікатор (правило, токен або ім'я мітки): вони відповідають isJavaIdentifierPart та isJavaIdentifierStart в класі символів Java. Обов’язково

використовується параметр -кодування при генерації в ANTLR, якщо граматичний файл не у форматі UTF-8, щоб ANTLR читав символи належним чином.

1.3. Літерали

ANTLR не розрізняє символи, великі та малі букви, як це робить більшість мов. Усі літерні рядки довжиною в один або декілька символів поміщаються в спеціальні символи “”, наприклад ";", "if", ">=" і "\" (посилається на односимвольну рядок, що містить єдиний символ цитати). Літерали ніколи не містять регулярних виразів.

Літерали можуть містити послідовності виходу Unicode форми '\ uXXXX' (для символів Unicode до 'U + FFFF') або '\ u {XXXXXX}' (для всіх символів Unicode), де 'XXXX' - це значення Unicode в шістнадцятковій формі. ANTLR також розуміє звичайні спеціальні послідовності виходу: '\ n' (новий рядок), '\ r' (повернення каретки), '\ t' (табуляція), '\ b' (повернення назад) та '\ f' (подача форми). Можна використовувати символи коду Unicode безпосередньо в літералі або використовувати послідовності виходу Unicode:

```
grammar Foreign;
a : '外' ;
```

Розпізнавач, що генерує ANTLR, припускає словниковий запас символів, що містить усі символи Unicode. Кодування вхідного файлу, що передбачається бібліотекою виконання, залежить від цільової мови. Для мови програмування Java бібліотека часу виконання передбачає, що файли містять символи кодування UTF-8. Використовуючи методи за замовчуванням в CharStreams, можна вказати інше кодування.

1.4. Дії

Дії - це кодові блоки, написані цільовою мовою. ~~Ви~~ Можна використовувати дії в декількох місцях граматики, але синтаксис завжди однаковий: довільний текст, оточений фігурними дужками. Не потрібно уникати закритого фігурного символу, якщо він містить рядок або коментар: "}" або / *} * /. Якщо фігурні дужки врівноважені, то також не потрібно

виходити: {...}. В іншому випадку потрібно уникати зайвих фігурних дужок із зворотною похилою: \{ або \}. Текст дії повинен відповідати цільовій мові, визначеній опцією мови. Вбудований код може з'являтись у: @header та @members з назвами дій, правилами аналізатора та лексера, специфікаціями збору винятків, розділами атрибутів для правил парсеру (повернені значення, аргументи та локальні пристрої), а також деякими параметрами елемента правила (наразі предикати).

Єдине тлумачення, яке ANTLR робить всередині дій, стосується граматичних атрибутів. Дії, вбудовані в правила лексерів, виводяться без будь-якої інтерпретації чи перекладу в генеровані лексеми.

1.5. Ключові слова

Список зарезервованих слів у граматиці ANTLR:

```
import, fragment, lexer, parser, grammar, returns,
locals, throws, catch, finally, mode, options, tokens
```

Також, хоча це не ключове слово, не використовується слово “rule” як назва правила. Крім того, не використовується жодне ключове слово цільової мови як ім’я токена, позначки чи правила.

РОЗДІЛ 2. ГРАМАТИКА

Граматика - це по суті граматична декларація, що супроводжується переліком правил, але має загальну форму:

```
/** Optional javadoc style comment */
grammar Name; ①
options {...}
import ... ;


tokens {...}
channels {...} // lexer only
@actionName {...}

rule1 // parser and lexer rules, possibly intermingled
```

```
...
ruleN
```

Ім'я файлу, що містить граматику X, повинно називатися X.g4. Можна вказати опції, імпорт, характеристики токенів та дії в будь-якому порядку. Тут може бути один із параметрів або імпорт, або характеристики токену. Усі ці елементи необов'язкові, крім заголовка та принаймні одного правила. Правила приймають основну форму:

```
ruleName : alternative1 | ... | alternativeN ;
```

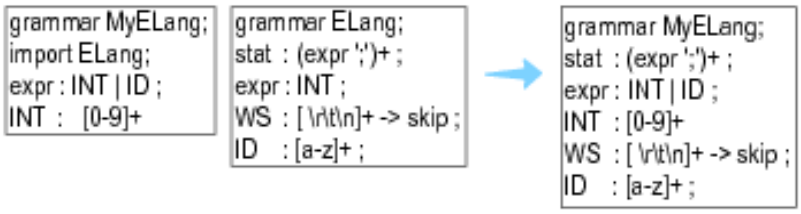
Назви правил парсера повинні починатися з малої літери, а правил лексера - з великої літери.

2.1. Імпорт граматики

Імпорт граматики дозволяє розбивати граматику на логічні та багаторазові частини, як видно в імпорті граматики. ANTLR сприймає імпортні граматики, дуже подібно до об'єктно-орієнтованих мов програмування. Граматика успадковує всі правила, специфікації та названі дії від імпортованої граматики.

Правила в "основній граматиці" мають вищий пріоритет ніж правила імпортованих граматик для здійснення успадкування. Імпорт є інтелектуальним оператором include (який не включає правила, які вже визначені). Результатом всього імпорту є одна комбінована граматика; генератор коду ANTLR бачить повну граматику і не знає, чи були імпортовані граматики.

Для обробки основної граматики інструмент ANTLR завантажує всі імпортовані граматики в підпорядковані граматичні об'єкти. Потім він об'єднує правила, типи токенів та іменовані дії з імпортованих граматик у основні граматики. На діаграмі нижче граматика праворуч ілюструє ефект граматики MyELang, що імпортує граматику ELang.

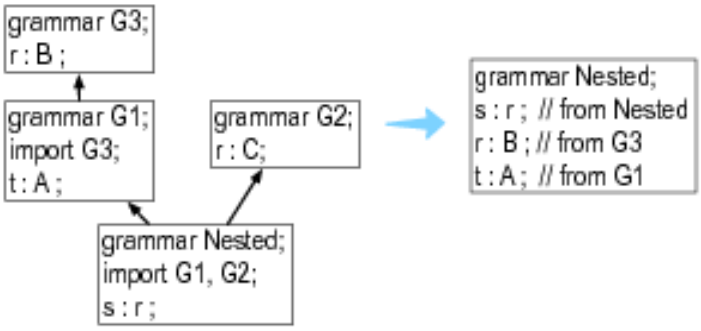


MyELang успадковує правила stat, WS та ID, але замінює правило expr та додає INT. Ось збірка зразків і тестовий запуск, який показує, що MyELang може розпізнавати цілі вирази, тоді як оригінальний ELang не може. Третє, помилкове введення значення викликає повідомлення про помилку, яке також демонструє, що аналізатор шукав expr MyELang, а не ELang.

```
$ antlr4 MyELang.g4
$ javac MyELang*.java
$ grun MyELang stat
=>      34;
=>      a;
=>      ;
=>      EOF
<=      line 3:0 extraneous input ';' expecting {INT, ID}
```

Якщо в основній граматиці є будь-які режими або будь-яка з імпортованих граматик, то процес імпорту буде імпортувати ці режими та об'єднає їх правила там, де вони не перекриті. У випадку, якщо будь-який режим стає порожнім, оскільки всі його правила були замінені правилами поза режимом, цей режим буде відкинутий.

Імпортовані граматики можуть також імпортувати інші граматики. Якщо дві або більше імпортованих граматик визначають правило r, ANTLR вибирає першу версію r, яку він знайде. На наступній діаграмі ANTLR перевіряє граматики у такому порядку: Nested, G1, G3, G2.



Nested включає правило r від G3, оскільки воно бачить цю версію перед r у G2.

Не кожен вид граматики може імпортувати всі інші види граматики:

• Граматики Lexer можуть імпортувати лексери, включаючи лексери, що містять режими.

• Парсери можуть імпортувати парсери.

• Комбіновані граматики можуть імпортувати парсери або лексери без режимів.

ANTLR додає імпортовані правила до кінця списку правил в основній граматиці лексера. Це означає, що правила лексера в основній граматиці мають перевагу над імпортованими правилами. Наприклад, якщо основна граматика визначає правило IF: 'if'; і імпортована граматика визначає ідентифікатор правила: [a-z]+; (яка також визнає if) імпортований ідентифікатор не приховає визначення основної граматики IF-лексера IF.

2.2. Розділ токенів

Розділ токенів визначає типи токенів, необхідних граматиці, для якої немає пов’язаного лексичного правила. Основний синтаксис:

```
tokens { Token1, ..., TokenN }
```

Здебільшого розділ токенів використовується для визначення типів токенів, необхідних для дій у граматиці

```
// explicitly define keyword token types to avoid implicit definition warnings
tokens { BEGIN, END, IF, THEN, WHILE }

@lexer::members { // keywords map used in lexer to assign token types
Map<String,Integer> keywords = new HashMap<String,Integer>() {{
        put("begin", KeywordsParser.BEGIN);
        put("end", KeywordsParser.END);
        ...
}};
}
```

Розділ токенів дійсно просто визначає набір токенів, які потрібно додати до загального набору.

```
$ cat Tok.g4
grammar Tok;
tokens { A, B, C }
a : X ;
$ antlr4 Tok.g4
warning(125): Tok.g4:3:4: implicit definition of token X in parser
$ cat Tok.tokens
A=1
B=2
C=3
X=4
```

2.3. Дії на рівні граматики

Наразі існує лише дві іменовані дії (для Java), що використовуються поза граматичними правилами: заголовок та члени. Перший вводить код в згенерований файл класу розпізнавача перед визначенням класу розпізнавача, а останній вводить код у визначення класу розпізнавача як поля та методи. Для комбінованих граматик ANTLR вводить дії і в аналізатор, і в лексер. Щоб обмежити дію згенерованим парсером або лексером, використовуйте @parser :: name або @lexer :: name.

Ось приклад, коли граматика вказує пакет для згенерованого коду:

```
grammar Count;

@header {
package foo;
}

@members {
int count = 0;
}

list
```

```
@after {System.out.println(count+" ints");}
: INT {count++;} (',' INT {count++;} )*
;

INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

Сама граматика повинна знаходитись у каталозі foo, щоб ANTLR генерував код у тому самому каталозі foo (принаймні, коли не використовується параметр -o ANTLR):

```
$ cd foo
$ antlr4 Count.g4 # generates code in the current directory (foo)
$ ls
Count.g4            CountLexer.java    CountParser.java
Count.tokens        CountLexer.tokens
CountBaseListener.java CountListener.java
$ javac *.java
$ cd ..
$ grun foo.Count list
=>          9, 10, 11
=>          EOF
<=          3 ints
```

Компілятор Java очікує, що класи пакета foo будуть в каталозі foo.

РОЗДІЛ 3. ПРАВИЛА ПАРСЕРА

Парсери складаються з набору правил розбору або в аналізаторі, або в комбінованій граматиці. Програма Java запускає аналізатор, викликаючи функцію правила, згенерованою ANTLR, пов'язану з потрібним правилом запуску. Найбільш основне правило - це лише правило, яке супроводжується єдиною альтернативою, закінченою крапкою з комою:

```
/** Javadoc comment can precede rule */
            retstat : 'return' expr ';' ;
```

Правила можуть також мати альтернативи, відокремлені |

```
operator:
        stat: retstat
        | 'break' ';'
        | 'continue' ';'
        ;
```

Альтернативи містять або список елементів правила, або вони є порожні. Наприклад, ось правило з порожньою альтернативою, яке робить все правило необов’язковим:

```
superClass
        : 'extends' ID
        | // empty means other alternative(s) are optional
        ;
```

3.1. Альтернативні позначки ~~мітки~~

Усі альтернативи в межах правила повинні мати позначку або жодна з альтернатив може не мати позначки. Ось два правила з альтернативами які мають позначку.

```
grammar T;
stat: 'return' e ';' # Return
        | 'break' ';' # Break
        ;
e   : e '*' e # Mult
    | e '+' e # Add
    | INT # Int
    ;
```

Альтернативні позначки не повинні бути в кінці рядка, а після символу # не повинно бути пробілу. ANTLR формує визначення правила контекстного класу для кожної позначки. Наприклад, ось listener, який генерує ANTLR:

```
public interface AListener extends ParseTreeListener {
        void enterReturn(AParser.ReturnContext ctx);
```

```
        void exitReturn(AParser.ReturnContext ctx);
        void enterBreak(AParser.BreakContext ctx);
        void exitBreak(AParser.BreakContext ctx);
        void enterMult(AParser.MultContext ctx);
        void exitMult(AParser.MultContext ctx);
        void enterAdd(AParser.AddContext ctx);
        void exitAdd(AParser.AddContext ctx);
        void enterInt(AParser.IntContext ctx);
        void exitInt(AParser.IntContext ctx);
}
```

Існують методи входу та виходу, пов'язані з кожною альтернативою, яка має позначку. Параметри цих методів специфічні для альтернатив.

Можна повторно використовувати одну і ту ж позначку на кількох альтернативах, щоб вказати, що walker дерева розбору повинен викликати ту саму подію для цих альтернатив. Наприклад, ось варіант щодо правила e з граматики A:

```
        e : e '*' e # BinaryOp
        | e '+' e # BinaryOp
        | INT # Int
        ;
```

ANTLR генерує такі listener методи для e:

```
        void enterBinaryOp(AParser.BinaryOpContext ctx);
        void exitBinaryOp(AParser.BinaryOpContext ctx);
        void enterInt(AParser.IntContext ctx);
        void exitInt(AParser.IntContext ctx);
```

ANTLR видає помилки, якщо альтернативна назва суперечить імені правила. Ось ще один перепис правила e, де дві альтернативні позначки суперечать іменам правил:

```
        e : e '*' e # e
        | e '+' e # Stat
        | INT # Int
        ;
```

Об'єкти контексту, згенеровані з імен правил та позначок ~~міток~~, отримують великі літери, і тому позначка Stat конфліктує з stat правилом:

```
$ antlr4 A.g4
error(124): A.g4:5:23: rule alt label e conflicts with rule e
error(124): A.g4:6:23: rule alt label Stat conflicts with rule stat
warning(125): A.g4:2:13: implicit definition of token INT in parser
```

3.2. Об'єкти контексту правила

ANTLR генерує методи доступу до об'єктів контексту правила (розбору вузлів дерева), пов'язаних з кожним посиланням на правило. Для правил з єдиним посиланням на правило ANTLR генерує метод без аргументів. Розглянемо наступне правило.

```
inc : e '++' ;
```

ANTLR генерує цей контекстний клас:

```
public static class IncContext extends ParserRuleContext {
    public EContext e() { ... } // return context object associated with e
    ...
}
```

ANTLR також надає підтримку доступу до контекстних об'єктів, коли правило має більше ніж одне посилання:

```
field : e '.' e ;
```

ANTLR генерує метод з індексом для доступу до i-го елемента, а також метод отримання контексту для всіх посилань на це правило:

```
public static class FieldContext extends ParserRuleContext {
    public EContext e(int i) { ... } // get ith e context
    public List<EContext> e() { ... } // return ALL e contexts
    ...
}
```

Якби було інше правило, s, з посиланням на field, вбудована дія могла б отримати доступ до списку відповідностей електронних правил, виконаних за полем:

```
s : field
        {
        List<EContext> x = $field.ctx.e();
        ...
        }
;
```

3.3. Позначки елементів правила

Можна позначити елементи правила за допомогою оператора = для додавання полів до об'єктів правила:

```
stat: 'return' value=e ';' # Return
        | 'break' ';' # Break
        ;
```

Тут значення — позначка для зворотного значення правила e, яке визначено в іншому місці. Позначки стають полями у відповідному класі вузла розбору дерева. У цьому випадку значення позначки стає полем у ReturnContext через альтернативну позначку Return:

```
public static class ReturnContext extends StatContext {
        public EContext value;
        ...
}
```

Часто зручно відстежувати кількість лексем, що можна зробити з оператором + = "позначка списку". Наприклад, наступне правило створює список об'єктів токену, зібраних для простої конструкції масиву:

```
array : '{' el+=INT (',' el+=INT)* '}' ;
```

ANTLR генерує поле «Список» у відповідному контекстному класі правила:

```
        public static class ArrayContext extends ParserRuleContext {
        public List<Token> el = new ArrayList<Token>();
        ...
        }
```

Ці мітки списку також працюють для посилань на правила:

```
        elist : exprs+=e (',' exprs+=e)* ;
```

ANTLR формує поле, що містить список об'єктів контексту:

```
public static class ElistContext extends ParserRuleContext {
public List<EContext> exprs = new ArrayList<EContext>();
...
}
```

3.4. Елементи правила

Елементи правила вказують, що повинен робити аналізатор в даний момент, як і вираз на мові програмування. Елементами можуть бути правило, токен, стрічковий літерал на зразок виразу, ідентифікатор та ‘return’. Ось повний перелік елементів правила:

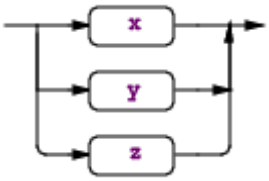
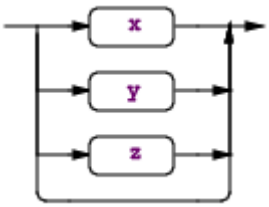
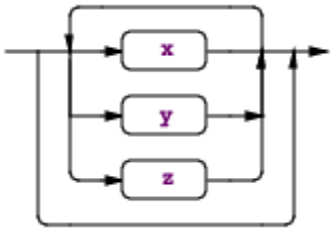
Синтаксис	Опис
T	Токен T у поточному положенні вводу. Токени завжди починаються з великої літери.
’literal’	Літеральний рядок у поточному положенні введення. Літеральний рядок - це просто токен із фіксованим рядком.
r	Правило r у поточному положенні введення, що означає виклик правила так само, як виклик функції. Назви правил парсера завжди починаються з малої літери.
r [«args»]	Правило r у поточній позиції введення, передаючи список аргументів так само, як виклик функції. Аргументи всередині квадратних дужок знаходяться в синтаксисі цільової мови і зазвичай є списком виразів, розділених комами.
{«action»}	Виконує дію відразу після попереднього альтернативного елемента та безпосередньо перед наступним альтернативним елементом. Дія відповідає синтаксису цільової мови. ANTLR копіює код дії в генерований клас дослівно, за винятком заміни посилань на атрибути та токени, таких як $ x і $ x.y.
{«p»}?	Оцінка семантичного предиката «p». Не продовжує аналізувати присудок, якщо «p» під час виконання отримує значення false. Предикати, що виникають під час прогнозування, коли ANTLR розрізняє альтернативи, включають або вимикають альтернативні варіанти, що оточують один або декілька

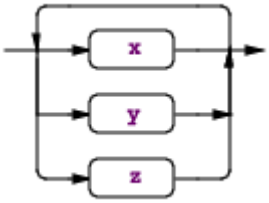
	предикатів.
.	Будь-який токен, за винятком кінця файлу токена. Оператор "крапка" називається підстановкою.

Коли потрібно порівняти все, крім певного токена або набору токенів, використовується оператором ~ "не". Цей оператор рідко використовується в аналізаторі, але він доступний. ~ INT відповідає будь-якому токену, крім токена INT. ~ ', Відповідає будь-якому токену, крім коми. ~ (INT | ID) відповідає будь-якому токену, крім INT або ID.

3.5. Підправила

Правило може містити альтернативні блоки, що називаються підправилами. Підправило - це правило, яке не має імені та записується в дужках. Підправила можуть мати одну або декілька альтернатив всередині дужок. Підправила не можуть визначати локальні атрибути і повертаються так, як можуть. Існує чотири види підправил (x, y і z - фрагменти граматики):

Позначення	Опис
	(x | y | z). Співставляє будь-яку альтернативу в підправилі рівно один раз. Приклад: returnType : (type | 'void') ;
	(x | y | z)? Ніщо або будь-яка альтернатива в межах підправила. Приклад: classDeclaration : 'class' ID (typeParameters)? ('extends' type)? ('implements' typeList)? classBody ;
	(x | y | z) * Співставляє альтернативу в межах підправила нуль або більше разів. Приклад: annotationName: ID ('.' ID) *;

	(x | y | z) + Співставляє альтернативу в межах підправила один чи більше разів. Приклад: annotations : (annotation)+ ;

Як скорочення, то можна опустити круглі дужки для підпунктів, що складаються з єдиної альтернативи, з єдиним посиланням на елемент правила. Наприклад, анотація + те саме, що (анотація) +, а ID + - те саме, що (ID) +. Позначки також працюють зі скороченням. ids + = INT + складає список об’єктів токена INT.

3.6. Catching Exceptions

Коли в правилі виникає синтаксична помилка, ANTLR обробляє виняток, повідомляє про помилку, намагається відновити (можливо, використовуючи більше токенів), а потім повертається з правила. Кожне правило загортається у визначення try/catch/finally:

```
void r() throws RecognitionException {
	try {
		rule-body
	}
	catch (RecognitionException re) {
		_errHandler.reportError(this, re);
		_errHandler.recover(this, re);
	}
	finally {
		exitRule();
	}
}
```

Щоб змінити обробку винятків для одного правила, вказується виняток після визначення правила:

```
r : ...
```

```
 ;
 catch[RecognitionException e] { throw e; }
```

Цей приклад показує, як уникнути повідомлень про помилки та відновлення за замовчуванням. Повідомлення від будь-якого винятку, запобігає ANTLR від генерації пункту для обробки RecognitionException.

Можна також вказати інші винятки:

```
r : ...
 ;
 catch[FailedPredicateException fpe] { ... }
 catch[RecognitionException e] { ... }
```

Фрагменти коду всередині фігурних дужок і дії "аргумент" виключення повинні бути записані на цільовій мові; Java, в цьому випадку. Коли потрібно виконати дію, навіть якщо виняток має місце, то вона ставиться в останній пункт:

```
r : ...
 ;
 // catch blocks go first
 finally { System.out.println("exit rule r"); }
```

Останній пункт виконується безпосередньо перед тим, як правило запускає exitRule перед поверненням. Якщо потрібно виконати дію після того, як правило закінчиться відповідністю альтернатив, але перед тим, як виконати очищення, використовується післядія.

Ось повний список винятків:

Виняток	Опис

RecognitionException	Суперклас всіх винятків, відкинутих розпізнавачем, згенерованим ANTLR. Це підклас RuntimeException, щоб не перевіряти винятки. Цей виняток записує, де розпізнавач (лексер або аналізатор) був у вході, де він знаходився в ATN граматиці (внутрішня структура даних графіків, що представляє граматику), стек виклику правил і яка проблема виникала.
NoViableAltException	Вказує, що аналізатор не міг вирішити, який з двох або більше шляхів пройти, переглянувши решту вхідних даних. Цей виняток відстежує початковий токен вхідних даних, а також знає, де був аналізатор у різних шляхах, коли сталася помилка.
LexerNoViableAltException	Еквівалент NoViableAltException, але лише для лексерів.
InputMismatchException	Поточний вхідний токен не відповідає тому, що очікував аналізатор.
FailedPredicateException	Семантичний предикат, який оцінюється як хибний під час прогнозування, робить оточуючу альтернативу недійсною. Прогнозування відбувається, коли правило передбачає, яку альтернативу приймати. Якщо всі дійсні шляхи зникнуть, парсер видасть NoViableAltException. Цей предикат вмводиться парсером, коли семантичний предикат оцінюється хибним поза межами передбачення, під час звичайного процесу розбору відповідних лексем та правил виклику.

3.7. Визначення атрибутів правила

Є цілий ряд дій, пов'язаних з елементами синтаксису, з правилами, які треба пам'ятати. Правила можуть мати аргументи, повертати значення та локальні змінні, як і функції мови програмування. ANTLR збирає всі визначені ~~вами~~ змінні та зберігає їх у об'єкті правила. Ці змінні зазвичай називають атрибутами. Ось загальний синтаксис, який показує всі можливі місця визначення атрибутів:

```
rulename[args] returns [retvals] locals [localvars] : ... ;
```

Атрибути, визначені як [...], можна використовувати, як і будь-яку іншу змінну. Ось зразок правила, який копіює параметри для повернення значень:

```
// Return the argument plus the integer value of the INT token
add[int x] returns [int result] : '+=' INT {$result = $x + $INT.int;} ;
```

Аргументи, локалі та повернення [...], як правило, є цільовою мовою, але з деякими обмеженнями. Рядок [...] - це список декларацій, розділених комами, або з позначенням типу префікса або постфіксу, або з позначенням no-type. Елементи можуть мати ініціалізатор, такий як [int x = 32, float y].

• Java, C #, C ++ використовують int x notation, але C++ має використовувати трохи змінене позначення для посилань на масив, int [ ] x, щоб вписатись у синтаксис типу id.

• Python та JavaScript не визначають статичні типи, тому дії є лише списками ідентифікаторів, такими як [i, j].

Як і у випадку граматики, можна вказати дії, названі на рівні правил. Для правил дійсні імена - init та after. Парсери виконують дії init безпосередньо перед тим, як спробувати відповідати пов’язаному правилу та виконувати після дій безпосередньо після відповідності правилу. ANTLR після дій не виконується як частина остаточно кодового блоку згенерованої функції правила. Використовується дія ANTLR finally, щоб розмістити код у створеній функції правила, finally, блоку коду.

Дії відбуваються після будь-яких дій аргументу, повернення значення чи локальних атрибутів.

```
/** Derived from rule "row : field (',' field)* '\r'? '\n' ;" */
```

```
row[String[] columns]
  returns [Map<String,String> values]
  locals [int col=0]
        @init {
        $values = new HashMap<String,String>();
        }
        @after {
        if ($values!=null && $values.size()>0) {
        System.out.println("values = "+$values);
        }
        }
        : ...
        ;
```

Рядок правила містить стовпці аргументів, повертає значення та визначає локальну змінну col. "Дії" у квадратних дужках копіюються безпосередньо в створений код:

```
public class CSVParser extends Parser {
        ...
        public static class RowContext extends ParserRuleContext {
                public String [] columns;
                public Map<String,String> values;
                public int col=0;
                ...
        }
        ...
}
```

Створені функції правил також задають аргументи правила як аргументи функції, але вони швидко копіюються в локальний об'єкт RowContext:

```
public class CSVParser extends Parser {
        ...
        public final RowContext row(String [] columns) throws RecognitionException {
                RowContext _localctx = new RowContext(_ctx, 4, columns);
                enterRule(_localctx, RULE_row);
```

```
                    ...
            }
            ... }
```

ANTLR відстежує nested [...] в межах дії, щоб рядки [] стовпців правильно розбирались. ANTLR також відстежує кутові дужки, щоб коми в параметрі загального типу не означали початку іншого атрибута. Значення Map <String, String> - це одне визначення атрибутів. У кожній дії може бути кілька атрибутів, навіть для повернених значень. Використовується кома для відокремлення атрибутів у межах однієї дії:

```
a[Map<String,String> x, int y] : ... ;
```

ANTLR інтерпретує цю дію для визначення двох аргументів, x і y:

```
public final AContext a(Map<String,String> x, int y)
            throws RecognitionException
{
            AContext _localctx = new AContext(_ctx, 0, x, y);
            enterRule(_localctx, RULE_a);
            ...
}
```

3.8. Правила запуску та EOF

Правило запуску - це правило, задіяне спочатку парсером; це функція правила, яку викликає компілятор. Наприклад, компілятор, що аналізує код Java, може викликати parser.compilationUnit () для об’єкта JavaParser, який називається парсер. Будь-яке правило в граматиці може діяти як правило запуску.

Правила запуску не обов'язково використовують усі дані. Вони використовують лише стільки, скільки потрібно, щоб відповідати альтернативі правилу. Наприклад, розглянемо наступне правило, яке відповідає одному, двом або трьом токенам, залежно від вводу.

```
s : ID
  | ID '+'
  | ID '+' INT
```

```
;
```

При a + 3 правило s відповідає третій альтернативі. Після позначення a + b воно відповідає другій альтернативі і ігнорує кінцевий токен b. Після b, воно відповідає першій альтернативі, ігноруючи токен b. Аналізатор не використовує повного вводу в двох останніх випадках, оскільки правило s прямо не говорить про те, що кінець файлу повинен відбуватися після відповідності альтернативі правилу.

Цей функціонал за замовчуванням дуже корисний для створення таких речей, як IDE. Уявіть, що IDE аналізує метод десь посередині великого файлу Java. Виклик правила methodDeclaration повинен намагатися відповідати лише методу і ігнорувати все, що настане далі.

З іншого боку, правила, що описують цілі вхідні файли, повинні посилатися на спеціальні EOF заздалегідь визначених токенів. Якщо вони цього не роблять, можна-поцікавитись, чому правило запуску не повідомляє про помилки при будь-якому вводі, незалежно від того, що їм надається. Ось правило, яке є частиною граматики для читання конфігураційних файлів:

```
config : element*; // can "match" even with invalid input.
```

Неправильне введення призведе до негайного повернення конфігурації без відповідності будь-якого вводу та без повідомлення про помилку. Ось відповідна специфікація:

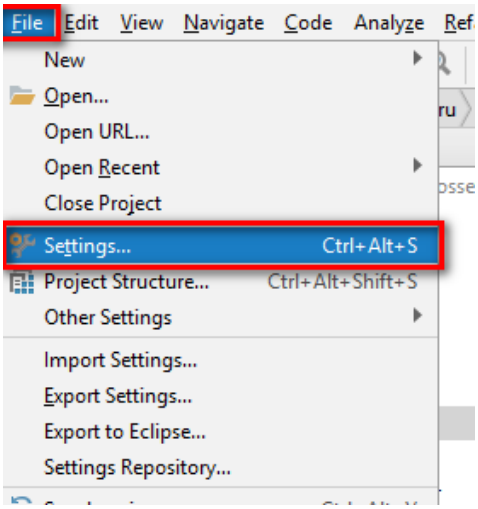
```
file : element* EOF; // don't stop early. must match all input
```

РОЗДІЛ 4. ANTLR З MAVEN ПРОЕКТОМ ДЛЯ JAVA

В цьому розділі розглядується приклад роботи з ANTLR з Maven проектом для Java через Intellij Idea. Це можна зробити у вигляді покрокової інструкції:

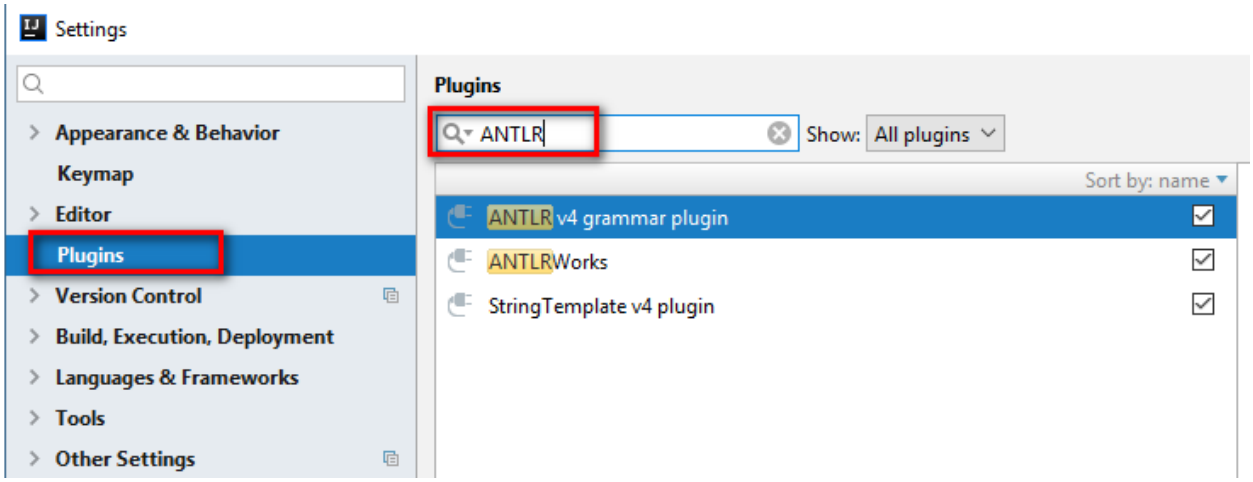
1) Встановити Oracle Java JDK і Intellij Idea, і запустити Intellij Idea.

2) Вибрати мишкою File-Setting або можна скористатись гарячими клавішами Ctrl+Alt+S. Після чого вибирається пункт Plugins.



Ввести в поле пошуку ANTLR і поставити плагін ANTLR v4 grammar plugin.

Можливо, знадобиться додатковий пошук по всіх репозиторіях.



3) Для Maven проекту додати в pom.xml або створити новий проект.

в dependencies

```
<dependency>
        <groupId>org.antlr</groupId>
        <artifactId>antlr4-runtime</artifactId>
        <version>4.7</version>
 </dependency>
```

і в plugins

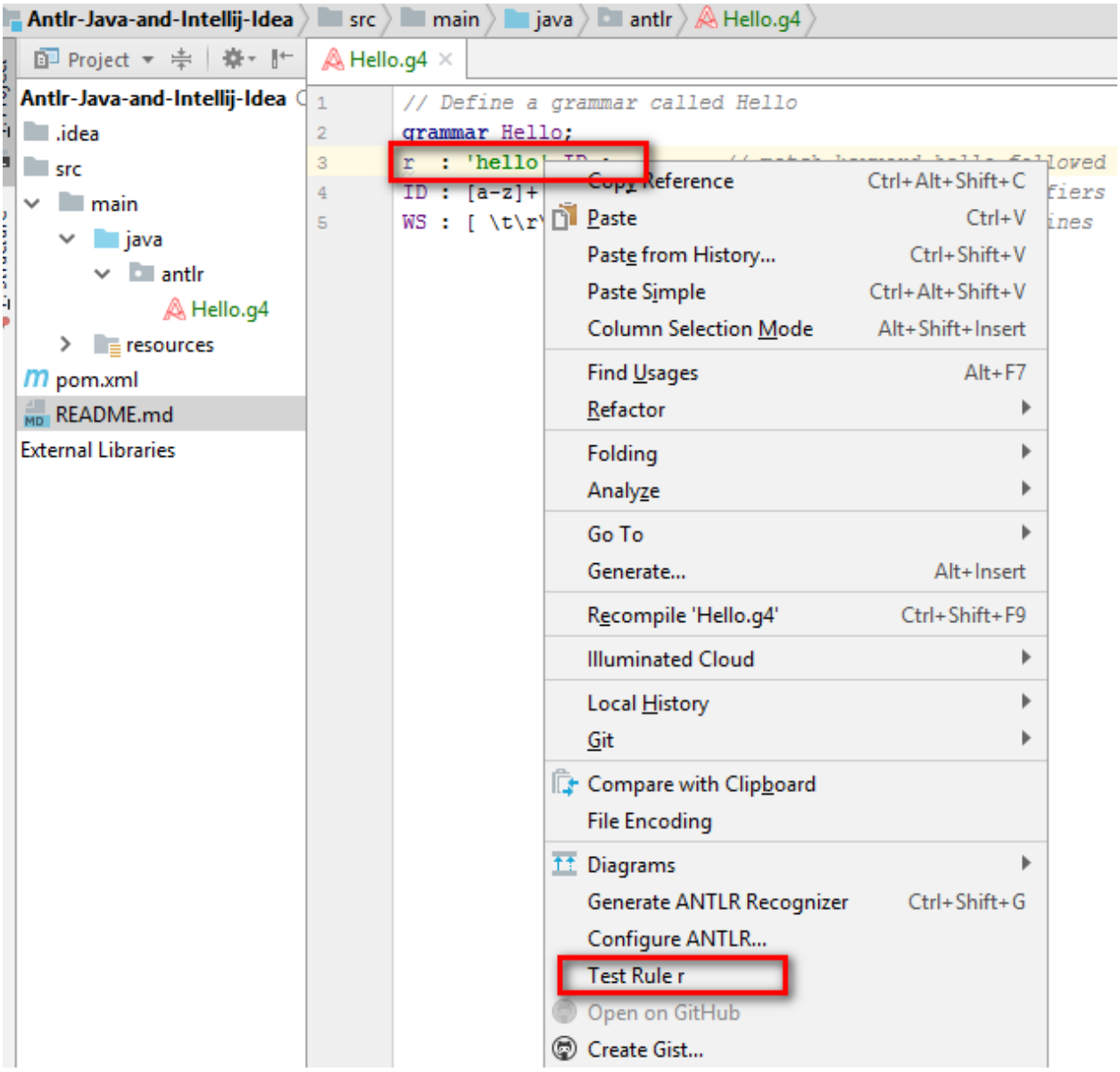
```
<plugin>
  <groupId>org.antlr</groupId>
          <artifactId>antlr4-maven-plugin</artifactId>
          <version>4.7</version>
          <executions>
             <execution>
                <goals>
                   <goal>antlr4</goal>
                </goals>
```

```
            </execution>
        </executions>
    </plugin>
```

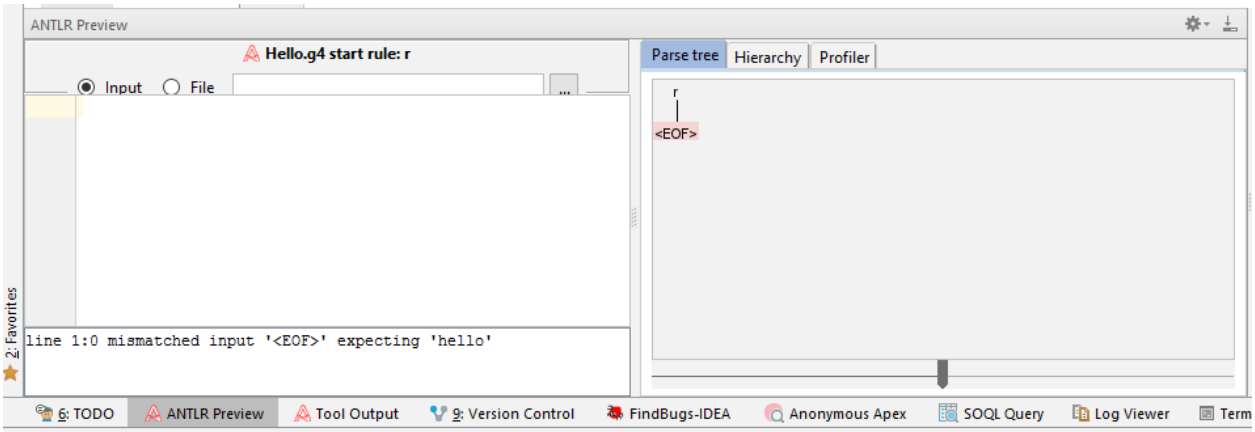
4) Створити і додати вручну файл граматики з розширенням .g4. Назва файлу повинна збігатися з словом після grammar в першому рядку. Складається вона приблизно так: береться те, що потрібно аналізувати, і розбивається на окремі токени. Для токенів описуються лексеми, наприклад всі англійські букви [a-zA-Z];, всі числа [0-9] і т.д..

```
// Define a grammar called Hello
grammar Hello;
r  : 'hello' ID ;         // match keyword hello followed by an identifier
ID : [a-z]+ ;             // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

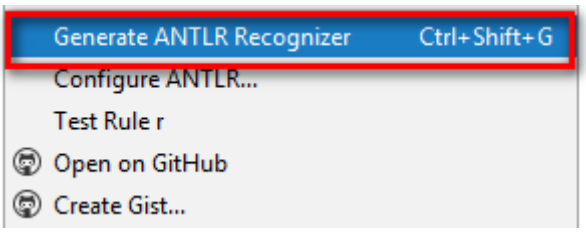
5) Далі правою кнопкою миші натискається на другий рядок файлу, який починається з r і вибирається пункт меню Test Rule r.



Внизу відкриються вікна перевірки граматики. В даному випадку, плагін показує помилку, швидше за все пов'язану з тим, що це тестовий приклад, проте парсер генерується. В даний момент на це звертається увага, але в реальних проектах, варто слідкувати за такими помилками.

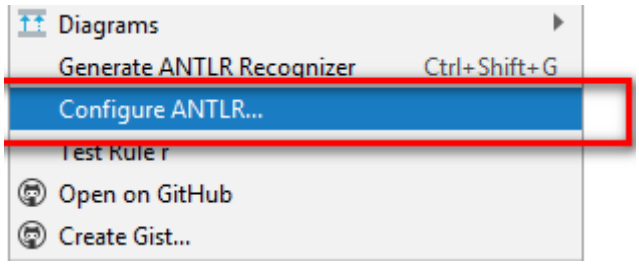


6) Натиснувши по файлу граматики правою кнопкою миші, вибирається пункт меню Configute ANTLR Recoqnizer і генерується парсер.

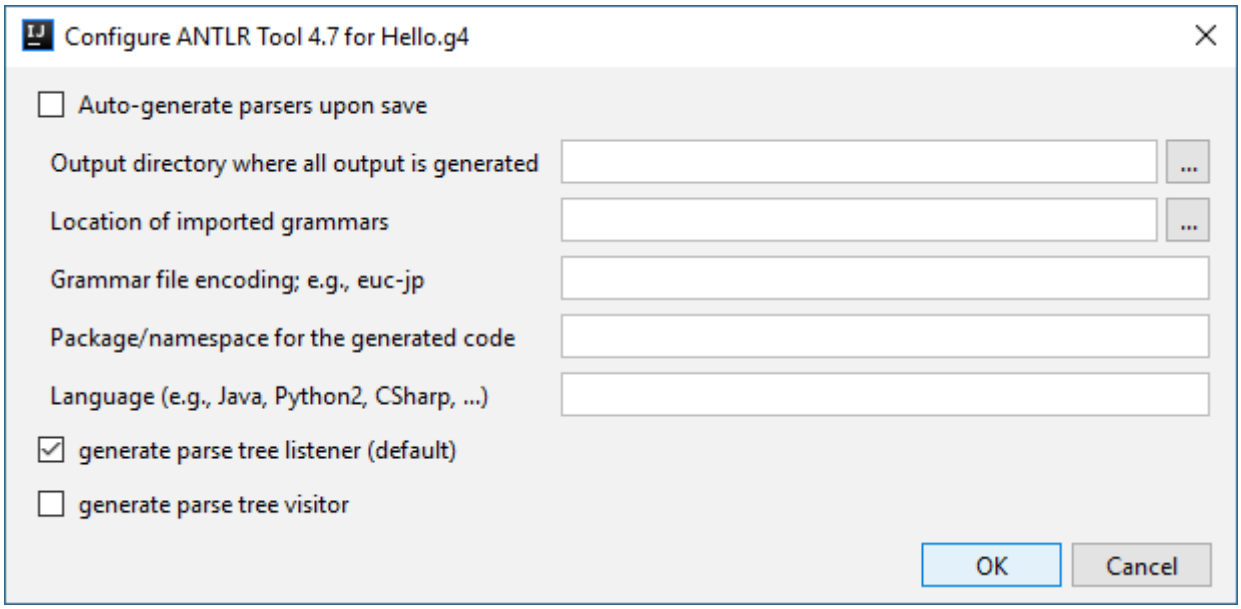


Після цього з'явиться в правому нижньому кутку повідомлення про завершення.

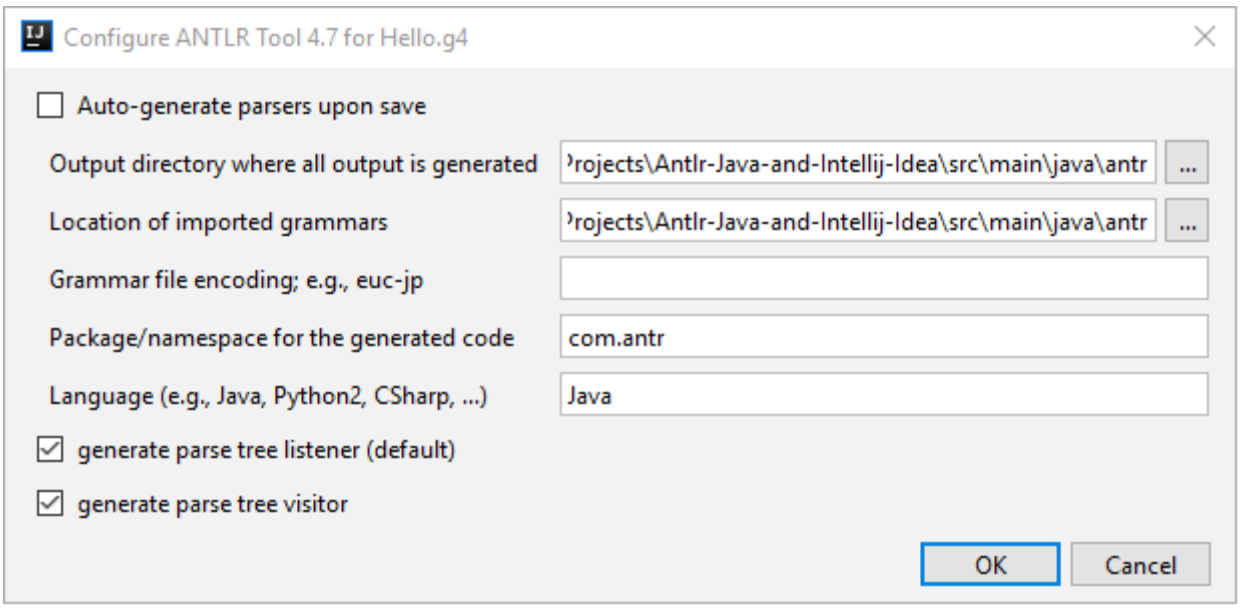
7) Далі знову натискається на файл правою кнопкою миші і вибирається пункт меню Configute ANTLR,



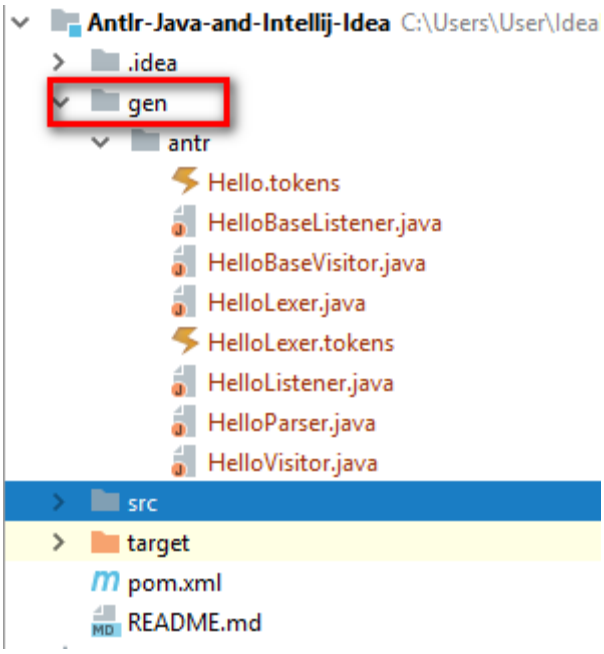
і з'являється вікно для конфігурації генерації файлів



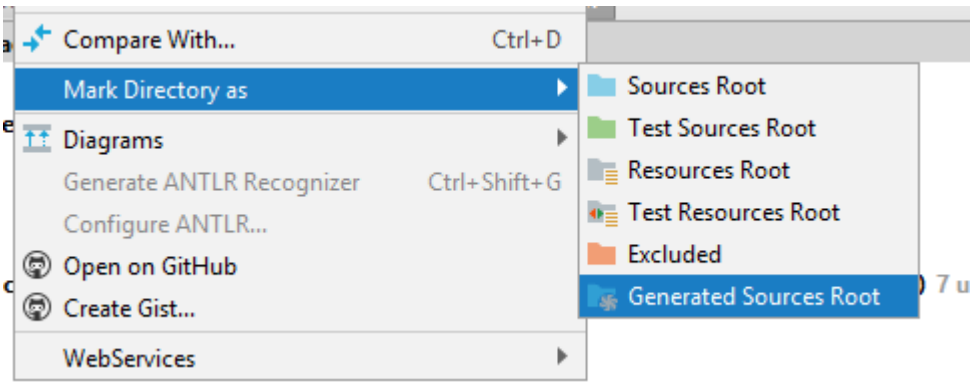
У цьому вікні вводяться дані про папку призначення і мову програмування, в даному випадку Java, чи потрібні visitor або listener, а також іншу необхідну інформацію, і натискається кнопка OK.



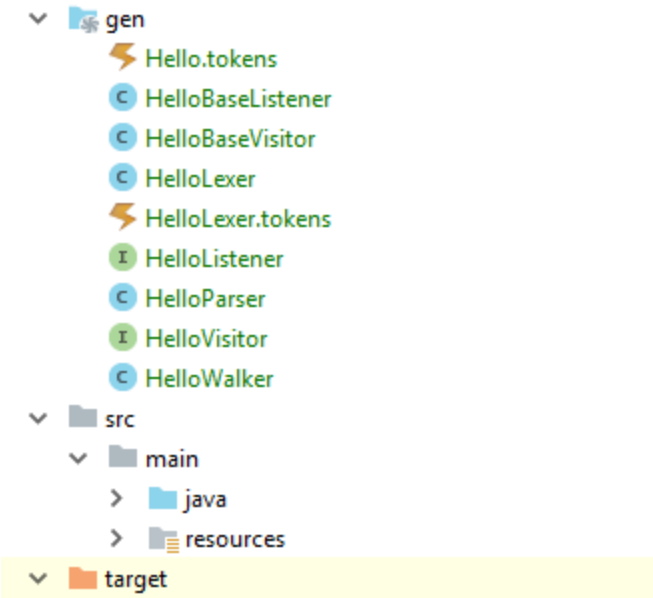
ANTLR після цього генерує файли для розпізнавання. Проте, хоча вихідний каталог вказано, часто створюється нова папка gen в корені проекту, причому java не розпізнає ці файли.



Для того, щоб java побачила ці файли, папку потрібно позначити правою кнопкою миші «Mark Directory As» на «Generated Sources Root» на папку gen.



І повинно вийти так:



8) ANTLR згенерував такі класи:

Клас HelloParser.java - це опис класу парсеру, тобто синтаксичного аналізатора, що відповідає граматиці Hello:

public class HelloParser extends Parser { ... }

Клас HelloLexer.java - це опис класу Лексера, або лексичного аналізатора, що відповідає граматиці HelloInit:

public class HelloLexer extends Lexer { ... }

Hello.tokens, HelloLexer.tokens - це допоміжні класи, які містять інформацію щодо токенів HelloListener.java, HelloBaseListener.java, HelloBaseVisitor, HelloVisitor - це класи, що містять описи методів, які дозволяють виконувати певні дії при обході синтаксичного дерева.

9) Після цього додамо клас HelloWalker (хоча це клас не обов'язковий, цей код можна змінити і додати в Main для виведення інформації)

public class HelloWalker extends HelloBaseListener {

```
    public void enterR(HelloParser.RContext ctx ) {
        System.out.println( "Entering R : " + ctx.ID().getText() );
    }
    public void exitR(HelloParser.RContext ctx ) {
        System.out.println( "Exiting R" );
    }
}
```

10) В цьому пункті створюється клас Main - точку входу в програму.

```
public class Main {
    public static void main( String[] args) throws Exception
    {
       HelloLexer lexer = new HelloLexer(CharStreams.fromString("hello world"));
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        HelloParser parser = new HelloParser(tokens);
        ParseTree tree = parser.r();
        ParseTreeWalker walker = new ParseTreeWalker();
        walker.walk(new HelloWalker(), tree);
    }
}
```

11) Запускається метод main, і отримується на виході в консолі успішно відпрацьований парсер.

```
Entering R : world
Exiting R
```

РОЗДІЛ 5. ВИКОРИСТАННЯ ANTLR З PYTHON

ANTLR орієнтується тільки на 2 версії Python: Python2 та Python3. Це тому, що сумісність між цими двома версіями мови є лише обмеженою.

Як створити лексер або аналізатор Python? Це майже те саме, що створити їх у Java, за винятком того, що потрібно вказати цільову мову, наприклад:

```
$ antlr4 -Dlanguage=Python2 MyGrammar.g4
```

або

```
$ antlr4 -Dlanguage=Python3 MyGrammar.g4
```

5.1. Запуск створеного лексера та або аналізатора

Припустимо, граматика названа, як "MyGrammar". Припустимо, аналізатор містить правило під назвою "startRule". Інструмент створить такі файли:

- MyGrammarLexer.py
- MyGrammarParser.py
- MyGrammarListener.py (якщо ви не активували опцію -no-listener)
- MyGrammarVisitor.py (якщо ви активували опцію -visitor)

Можна побачити, що немає базового слухача або відвідувача, що генерується, це тому, що Python не підтримує інтерфейси, генерований слухач і відвідувач є повноцінними класами.

Повністю функціонуючий сценарій виглядає наступним чином:

```
import sys
from antlr4 import *
from MyGrammarLexer import MyGrammarLexer
from MyGrammarParser import MyGrammarParser

def main(argv):
    input_stream = argv[1])
    lexer = MyGrammarLexer(input_stream)
    stream = CommonTokenStream(lexer)
    parser = MyGrammarParser(stream)
    tree = parser.startRule()

if __name__ == '__main__':
    main(sys.argv)
```

5. 2. Створення та запуск власного слухача

Припустимо, граматика MyGrammar містить 2 правила: "ключ" та "значення". Інструмент ANTLR створить наступного слухача:

```
class MyGrammarListener(ParseTreeListener):
    def enterKey(self, ctx):
```

```
        pass
    def exitKey(self, ctx):
        pass
    def enterValue(self, ctx):
        pass
    def exitValue(self, ctx):
        pass
```

Для забезпечення користувацької поведінки ~~ви~~ можна створити наступний клас:

```
class KeyPrinter(MyGrammarListener):
    def exitKey(self, ctx):
        print("Oh, a key!")
```

Щоб запустити цього слухача, потрібно просто додати наступні рядки до вищевказаного коду:

```
        ...
        tree = parser.startRule() - only repeated here for reference
    printer = KeyPrinter()
    walker = ParseTreeWalker()
    walker.walk(printer, tree)
```

Реалізація ANTLR Python максимально наближена до Java, тому не повинно бути важко адаптувати приклади для Python.

СПИСОК ЛІТЕРАТУРИ

1. ANTLR [Електронний ресурс]. – Режим доступу: https://en.wikipedia.org/wiki/ANTLR
2. ANTLR 4 Documentation [Електронний ресурс]. – Режим доступу: https://github.com/antlr/antlr4/blob/4.8/doc/index.md
3. Terence Parr. The Definitive ANTLR 4 Reference. University of San Francisco, 2007