Traductores con ANTLR (listeners)

Felipe Restrepo Calle

ferestrepoca@unal.edu.co

Profesor Asociado

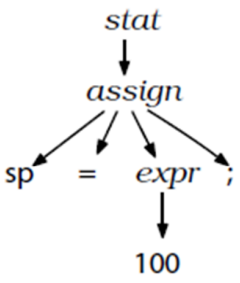
Universidad Nacional de Colombia

Actualizado: **18/04/2023**

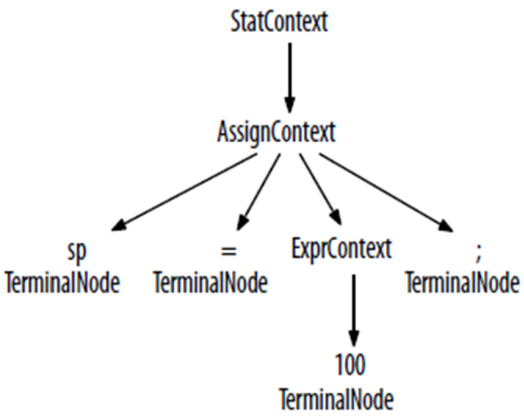
En esta guía se presenta cómo trabajar con ANTLR para construir aplicaciones de análisis automático de código fuente usando el patrón de diseño Listeners. De esta manera, se facilita el desarrollo de traductores de una sola pasada del árbol sintáctico.

Árboles sintácticos en ANTLR v4

A continuación se presenta un ejemplo de un árbol sintáctico generado por ANTLR v4 (con los nombres de las clases):



Parse tree



Parse tree node class names

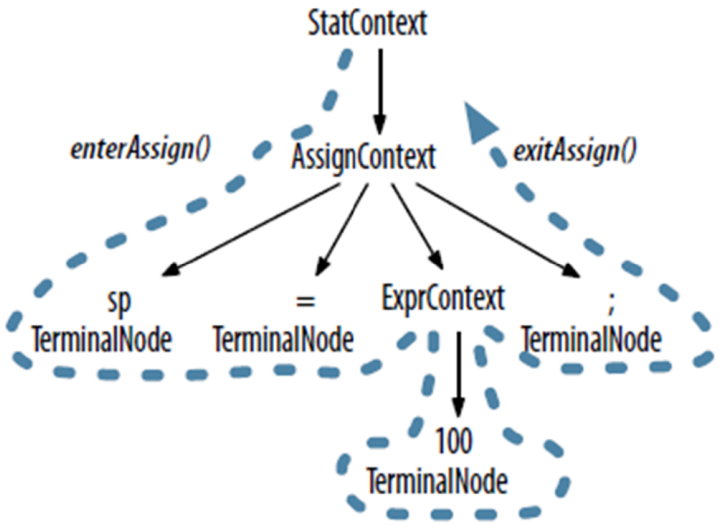
Observe que las hojas del árbol corresponden a nodos terminales y los demás nodos corresponden a alguna regla de producción de la gramática. Estos últimos objetos se conocen en ANTLR v4 como contextos (*context*) porque almacenan todo lo que sabemos del reconocimiento de una frase por una regla particular de la gramática. Cada contexto conoce su token inicial y final para la frase y proporciona acceso a todos los elementos de esa frase. Por ejemplo, *AssignContext* (*Assign*: **ID TK_EQUAL** *expr*) proporciona los métodos: ID() y expr() para acceder al nodo identificador y el subárbol de la expresión.

Con esta estructura de datos podríamos implementar a mano nuestros algoritmos para recorrer el árbol sintáctico en profundidad y programar las acciones requeridas durante el análisis, las cuales serán ejecutadas a medida que se vayan visitando los nodos del árbol. Sin embargo, para evitar tener que escribir estos métodos cada vez que implementemos un procesador de lenguaje, ANTLR proporciona sus propios mecanismos para hacer esto por nosotros, mediante dos patrones de diseño: *Listeners* y *Visitors*. En esta guía se detalla el primero.

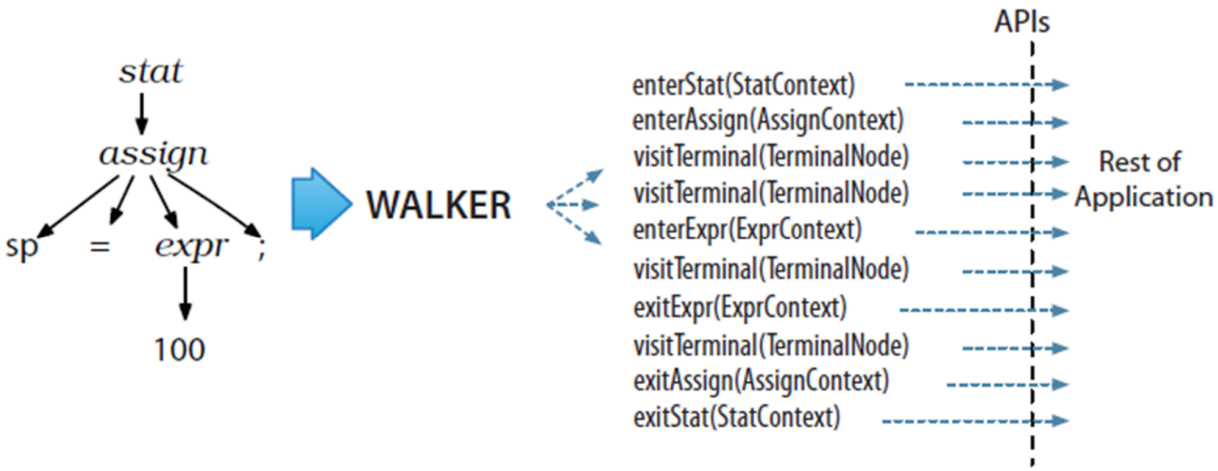
Parse-tree Listeners

Por defecto, ANTLR genera una interfaz para un parse-tree listener que responde a los eventos desencadenados por el objeto que recorre el árbol sintáctico (*walker*). Los Listeners son como los objetos manejadores de documentos SAX (Simple API for XML) para los analizadores de XML. Estos reciben una notificación de eventos como *startDocument()* y *endDocument()* y ejecutan las acciones correspondientes. **Los métodos de un Listener son simplemente *callbacks* que responden a eventos que ocurren al recorrer el árbol sintáctico.** Pueden ser comparados con los métodos que implementamos para responder a un evento de *click* en un botón de una aplicación con interfaz gráfica (GUI).

Para recorrer el árbol y por ende desencadenar las llamadas a los métodos del *Listener*, ANTLR proporciona la clase *ParseTreeWalker*. Para implementar una aplicación, necesitamos desarrollar una implementación de *ParseTreeListener* que contenga el código específico de la aplicación. ANTLR genera automáticamente una subclase de *ParseTreeListener* para cada gramática específica con los métodos *enter* y *exit* para cada regla. Cuando el *walker* llega al nodo para la regla *Assign*, se dispara el evento *enterAssign()*, el cual recibe como parámetro toda la información almacenada en ese contexto (*AssignContext*). Asimismo, después de que el *walker* ha visitado todos los hijos del nodo *assign*, se dispara el evento *exitAssign()*.



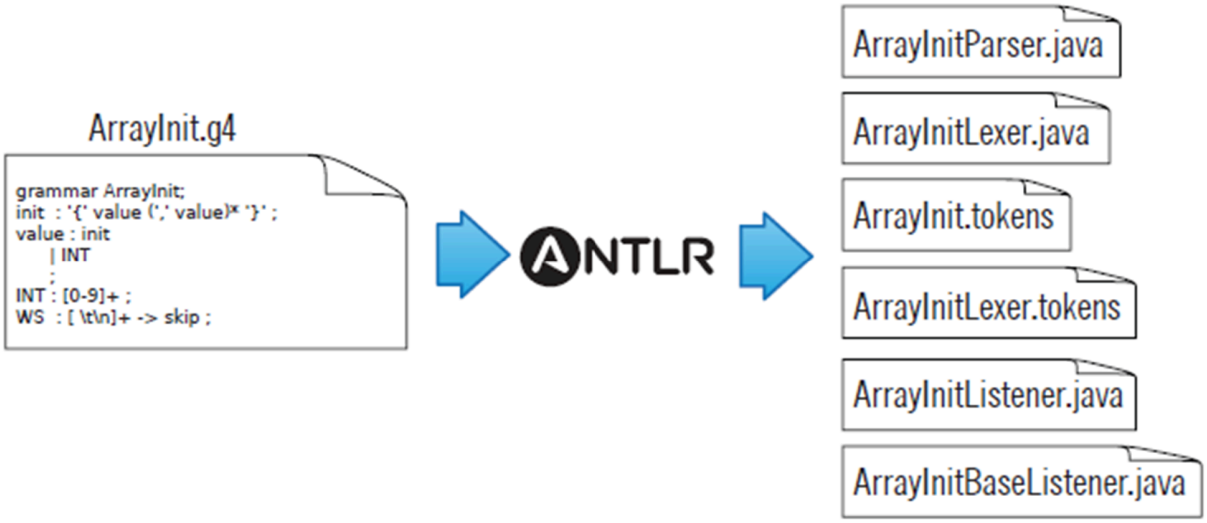
A continuación se presenta toda la secuencia de métodos desencadenados por el walker para la sentencia de ejemplo (**sp=100;**):



Lo mejor de este mecanismo de Listeners es que todo funciona automáticamente. No tenemos que implementar el walker, y nuestros métodos de los Listeners no tienen que visitar explícitamente a sus hijos. ANTLR proporciona esto.

Archivos generados por ANTLR

Si estuviéramos especificando la gramática del lenguaje en un archivo llamado **PRUEBA.g4, y** quisiéramos trabajar con Listeners, ANTLR se encargará de generar automáticamente los siguientes archivos (en el caso que se esté usando JAVA):



- **PRUEBALexer.java:** código fuente del analizador léxico
- **PRUEBAParser.java:** código fuente del analizador sintáctico

- **PRUEBA.tokens:** tokens para el analizador sintáctico
- **PRUEBALexer.tokens:** tokens para el analizador léxico
- **PRUEBAListener.java:** Interface que describe los “eventos” que podemos llamar al recorrer el árbol sintáctico que genera ANTLR automáticamente
- **PRUEBABaseListener.java:** Clase con un conjunto de implementaciones vacías. Basta con **sobreescribir** los métodos que nos interesan.

Ejemplo

Reconocedor de números enteros encerrados entre llaves - ArrayInit

Vamos a construir una gramática para desarrollar un reconocedor de conjuntos de números enteros separados por comas y encerrados entre llaves (posiblemente anidados), como por ejemplo: {1, 2, 3} y {1, {2, 3}, 4}. Esto podría ser útil para reconocer expresiones de inicialización de arreglos en Java, por ejemplo. Cuando hayamos desarrollado el reconocedor, implementaremos un traductor de este tipo de cadenas basado en Listeners.

1. Especificación de la gramática

Archivo: ArrayInit.g4

```
/** Grammars always start with a grammar header. This grammar is called
* ArrayInit and must match the filename: ArrayInit.g4
*/
grammar ArrayInit;
// A rule called init that matches comma-separated values between {...}.
init : '{' value (',' value)* '}' ; // must match at least one value
// A value can be either a nested array/struct or a simple integer (INT)
value : init
        | INT
        ;
// parser rules start with lowercase letters, lexer rules with uppercase
INT : [0-9]+ ; // Define token INT as one or more digits
WS : [ \t\r\n]+ -> skip ; // Define whitespace rule, toss it out
```

2. Integrando el reconocedor generado con un programa en Java (main)

Archivo: Test.java

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.File;

public class Test {
  public static void main(String[] args) throws Exception {
      try{
                        // crear un analizador léxico
                        ArrayInitLexer lexer;
                        if (args.length>0)
                          lexer = new
ArrayInitLexer(CharStreams.fromFileName(args[0]));
                        else
                          lexer = new
ArrayInitLexer(CharStreams.fromStream(System.in));
                        // Identificar al analizador léxico como fuente de
tokens para el sintactico
                        CommonTokenStream tokens = new
CommonTokenStream(lexer);
                        // Crear el analizador sintáctico que se alimenta a
partir del buffer de tokens
                        ArrayInitParser parser = new
ArrayInitParser(tokens);
                        ParseTree tree = parser.init(); // comienza el
análisis en la regla inicial
                  System.out.println(tree.toStringTree(parser)); // imprime
el árbol en forma textual
      } catch (Exception e){
                        System.err.println("Error (Test): " + e);
      }
  }
}
```

3. Especificación de un traductor para este tipo de cadenas (usando Listeners)

Vamos a traducir los elementos de inicialización de un arreglo en Java como **{99, 3, 451}** (los cuales ya reconocemos) a una cadena de constantes Unicode como: **“\u0063\u0003\u01c3”**, donde cada uno de estos corresponde a la notación hexadecimal del valor original (por ejemplo: 99d = 63h).

Ejemplo de traducción:

De aquí podemos identificar las reglas de traducción necesarias:

- Traducir { a “.
- Traducir } a “.
- Traducir los números enteros a una cadena de 4 dígitos con su representación hexadecimal, precedida de \u.

4. Implementación de los métodos del traductor (en el Listener)

Para lograr el objetivo, lo único que tendremos que hacer es implementar algunos métodos en una subclase de *ArrayInitBaseListener*. La estrategia básica consiste en que cada método del Listener imprima una pequeña parte de la traducción correspondiente a la cadena de entrada cuando sea llamado por el objeto (*walker*) que recorre el árbol sintáctico, es decir, implementaremos el Esquema de Traducción Dirigido por la Sintaxis (ETDS).

A continuación se presenta la implementación del *Listener* de acuerdo a nuestras reglas de traducción:

Archivo: ShortToUnicodeString.java

```
/** Convert short array inits like {1,2,3} to "\u0001\u0002\u0003" */
public class ShortToUnicodeString extends ArrayInitBaseListener {
@Override           /** Translate { to " */
public void enterInit(ArrayInitParser.InitContext ctx) {
System.out.print('"');
}

@Override           /** Translate } to " */
public void exitInit(ArrayInitParser.InitContext ctx) {
System.out.print('"');
}

@Override    /** Translate integers to 4-digit hex strings prefixed with \\u */
public void enterValue(ArrayInitParser.ValueContext ctx) {
// Assumes no nested array initializers
int value = Integer.valueOf(ctx.INT().getText());
System.out.printf("\\u%04x", value);
}
}
```

No es necesario sobreescribir todos los métodos *enter/exit*, sólo los que vamos a usar. La única expresión poco familiar en este ejemplo es *ctx.INT()*, la cual le solicita al objeto del contexto el token del número entero **INT** (capturado por la regla **value**).

5. Crear la aplicación (traductor)

Sólo nos falta crear la aplicación que integre todo, basándonos en la clase Test mostrada previamente.

Archivo: Translate.java

```
// import ANTLR's runtime libraries
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.File;

public class Translate {
public static void main(String[] args) throws Exception {
// create a CharStream that reads from standard input / file
// create a lexer that feeds off of input CharStream
ArrayInitLexer lexer;
```

```
if (args.length>0)
    lexer = new ArrayInitLexer(CharStreams.fromFileName(args[0]));
else
    lexer = new ArrayInitLexer(CharStreams.fromStream(System.in));
// create a buffer of tokens pulled from the lexer
CommonTokenStream tokens = new CommonTokenStream(lexer);
// create a parser that feeds off the tokens buffer
ArrayInitParser parser = new ArrayInitParser(tokens);
ParseTree tree = parser.init(); // begin parsing at init rule

➤ // Create a generic parse tree walker that can trigger callbacks
➤ ParseTreeWalker walker = new ParseTreeWalker();
➤ // Walk the tree created during the parse, trigger callbacks
➤ walker.walk(new ShortToUnicodeString(), tree);
➤ System.out.println(); // print a \n after translation
}
}
```

En este caso creamos un objeto walker de la clase *ParseTreeWalker*. Llamamos su método *walk()*, el cual recorre el árbol sintáctico retornado por el parser. A medida que se va recorriendo el árbol, se van desencadenando los llamados a los métodos de nuestro Listener (*ShortToUnicodeString*).

Finalmente, compilamos y probamos el traductor

Ejercicio

Crear un traductor (prueba de concepto) usando *listeners* con ANTLR para una de las gramáticas de los lenguajes de programación disponibles en:
https://github.com/antlr/grammars-v4

Documentación (recursos recomendados)

- Parr Terence. The Definitive ANTLR 4 Reference. The pragmatic bookshelf. 2012. Ver capítulo 4.
- Tutorial ANTLR: http://www.xfront.com/ANTLR/
- Tomassetti. The ANTLR mega tutorial. March 8, 2017. Disponible en: https://tomassetti.me/antlr-mega-tutorial/

- Ver la clase de Terence Parr (creador de ANTLR) en la Universidad de San Francisco, donde presenta ANTLR v4 - explica Listeners/Visitors y los analizadores ALL(*). En este enlace: https://vimeo.com/59285751
- Ver la conferencia del profesor Terence Parr, creador de ANTLR, acerca de algoritmos para la construcción de analizadores sintácticos: **The Quest for the One True Parser**.
- Listado de gramáticas disponibles para ANTLR: https://github.com/antlr/grammars-v4

Referencias

[1] Parr Terence. The Definitive ANTLR 4 Reference. The pragmatic bookshelf. 2012. Ver capítulo 4.

[2] Tutorial ANTLR: http://www.xfront.com/ANTLR/

[3] Tomassetti. The ANTLR mega tutorial. March 8, 2017. Disponible en: https://tomassetti.me/antlr-mega-tutorial/