UNIVERSIDAD
NACIONAL
DE COLOMBIA

Tutorial ANTLR v4

Felipe Restrepo Calle
Profesor Asociado
Última actualización: 28/05/2019
ferestrepoca@unal.edu.co

Contenido

UNIVERSIDAD
NACIONAL
DE COLOMBIA

1. Antecedentes

Ya sea para ensamblador, un lenguaje intermedio, o un lenguaje de programación de alto nivel, al escribir un procesador de lenguajes de programación (compilador, intérprete, traductor,...) "a mano" debemos realizar muchas tareas de forma repetitiva. Todo este proceso se pude automatizar enormemente utilizando herramientas de software. Debido a la necesidad de automatización aparecieron, desde los años 70, las primeras herramientas de ayuda a la construcción de procesadores de lenguajes. Estas herramientas generan código en un lenguaje de programación (C, C++, JAVA, etc.) para ahorrar al programador la parte repetitiva de la programación de compiladores, pudiendo éste dedicarse al diseño del lenguaje y de las aplicaciones de procesamiento de lenguaje.

Varias universidades construyeron herramientas de este tipo, pero fueron YACC y LEX las que más se han extendido. Al principio de los 70, Stephen C. Johnson desarrolló YACC (*Yet Another Compiler-Compiler*) laboratorios Bell, usando un dialecto portable del lenguaje C. YACC es una herramienta capaz de generar un analizador sintáctico en C a partir de una serie de reglas de sintaxis que debe cumplir. Dichas reglas se especifican en un lenguaje muy sencillo. YACC se apoya en la herramienta LEX para el análisis léxico. LEX fue desarrollada por Eric Schmidt. Esta herramienta también fue desarrollada en C, y también genera un analizador en C. LEX y YACC sirvieron como base a FLEX y BISON, que se consideran sus herederas. FLEX y BISON son dos productos de la FSF (*Free Software Foundation*).

Actualmente que el proceso de compilación está más formalizado; se admite ampliamente que es necesario crear un árbol de sintaxis abstracta si se quiere realizar un análisis semántico correctamente. Es necesario crear y recorrer de una forma estandarizada los árboles de sintaxis abstracta.

ANTLR es un software desarrollado en JAVA por varios desarrolladores, aunque la idea inicial y las decisiones principales de diseño son de Terence Parr. En su proyecto de grado, Terence presentaba una manera eficiente de implementar los analizadores sintácticos LL. Los hallazgos presentados en este proyecto fueron los que le llevaron a implementar PCCTS, que puede considerarse como la "semilla" de ANTLR. PCCTS permite generar analizadores léxicos y sintácticos. Para recorrer los árboles de sintaxis abstracta, se desarrolló un programa llamado SORCERER. ANTLR ha sufrido varias reescrituras completas desde su inicio, incluyendo el cambio del lenguaje de programación utilizado (inicialmente fue C) y varios cambios de nombre. Mientras que FLEX y BISON son herramientas dedicadas a una sola fase del análisis (análisis léxico y sintáctico, respectivamente), ANTLR es capaz de actuar a tres niveles a la vez: análisis léxico, sintáctico y semántico (cuatro, si tenemos en cuenta la generación de código).

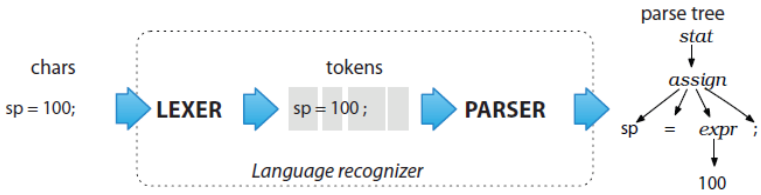
El uso de una sola herramienta para todos los niveles tiene varias ventajas. La más importante es la "estandarización": con ANTLR basta con comprender el paradigma de análisis una vez para poder implementar todas las fases de análisis. Con FLEX+BISON es necesario comprender y saber utilizar herramientas completamente diferentes. FLEX está basada en autómatas finitos deterministas y BISON en un analizador LALR(1).

2. ¿Qué es ANTLR?

ANTLR (*ANother Tool for Language Recognition*) es un generador de analizadores. Mucha gente llama a estas herramientas compiladores de compiladores, dado que el ayudar a implementar compiladores es su uso más popular. ANTLR es una herramienta que integra la generación de analizadores léxicos, sintácticos, árboles de sintaxis abstracta y evaluadores de atributos. ANTLR está escrito en Java y genera: Java, C#, Javascript, Python2, Python3, Go, C++, Swift. Todos los detalles se encuentran en su página oficial: http://www.antlr.org/

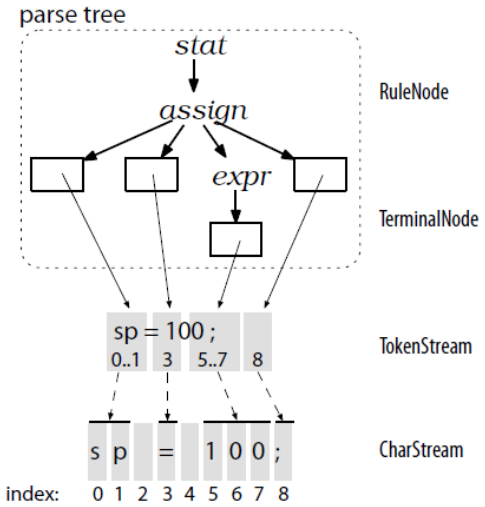
Su funcionamiento básico es el siguiente:

- Se diseña un analizador léxico y un analizador sintáctico usando una gramática descrita en un archivo de especificación (escrito con la sintaxis propia de ANTLR).
- ANTLR genera el código fuente del analizador léxico y sintáctico correspondientes.



Características:

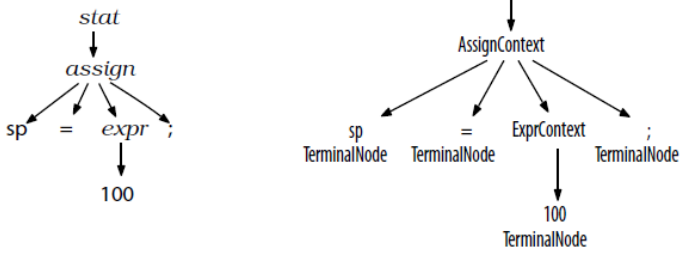
- ANTLR "ANother Tool for Language Recognition"
- Construcción automática de procesadores de lenguaje
- Genera el código fuente de analizadores en distintos lenguajes que pueden no sólo realizar el análisis, sino también construir árboles de derivación de forma automática. También pueden generar "*tree walkers*" que son estructuras que se usan para visitar los nodos de los árboles sintácticos y ejecutar código específico de la aplicación cuando se visita alguno de los nodos del árbol.
- Permite incorporar acciones semánticas independientes de las gramáticas.
- Multiplataforma
- Posee un plug-ins para distintos IDEs (Intellij, Eclipse, Netbeans): http://www.antlr.org/tools.html



Importante: ANTLR v4 genera analizadores LL(*) Adaptativos – ALL(*). Gracias a esto, ANTLR v4 se ha llamado a ANTLR v4 como la versión "Honey Badger" en honor a ese curioso animal. "*It takes whatever grammar you give it; it doesn't give a damn!*"
Ver video: *The Crazy Nastyass Honey Badger* - http://www.youtube.com/watch?v=4r7wHMg5Yjg

3. 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

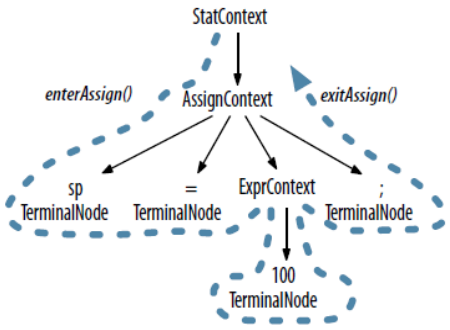
Observe que las hojas del árbol corresponden a nodos terminales y los demás nodos corresponden a alguna regla de producción de la gramática. Estos últimos objetos se conocen en ANTLR v4 como contextos (*context*) porque almacenan todo lo que sabemos del reconocimiento de una frase por una regla particular 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*.

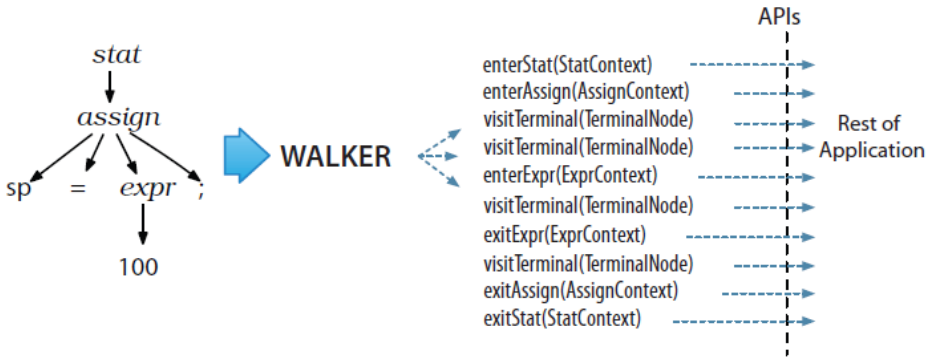
Parse-tree Listeners

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 correspondiente. 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*().



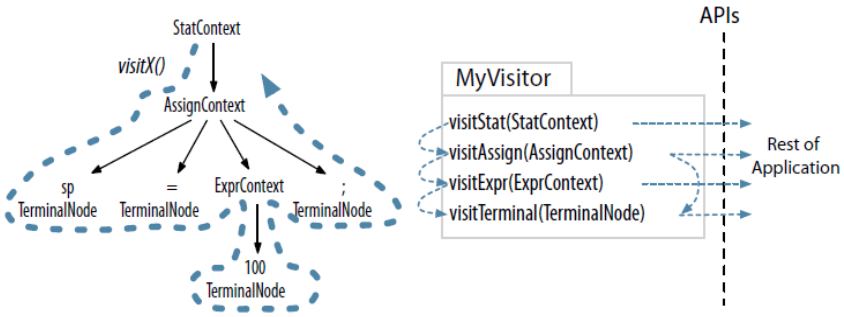
A continuación se presenta toda la secuencia de métodos desencadenados por el *walker* para la sentencia de ejemplo (sp=100;):



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.

Parse-tree Visitors

Sin embargo, en algunas situaciones queremos controlar el recorrido del árbol, llamando a los métodos para visitar a los hijos de algún nodo de forma explícita. En este caso debemos usar la opción de ANTLR **–visitor** en la línea de comandos. Esta opción hace que ANTLR genere una interface de *Visitor* a partir de una gramática, la cual contiene un método *visit()* por cada una de las reglas. A continuación se puede ver el patrón de diseño *visitor* operando en nuestro árbol sintáctico:



Para dar inicio al recorrido del árbol, nuestra aplicación creará una implementación del *visitor* y llamará al método *visit()* de la raíz del árbol (regla inicial de la gramática).

```
ParseTree tree = ... ; // tree is result of parsing
MyVisitor v = new MyVisitor();
v.visit(tree);
```

Al hacer esto, ANTLR al ver la regla inicial de la gramática, llamará al método *visitStat()*. A partir de aquí, la implementación de *visitStat()*, llamará al método *visit()* de los nodos hijos para continuar el recorrido del árbol. También se pueden llamar los métodos explícitamente de la forma *visitAssign()*.

4. Aplicaciones en el mundo actual

- Las búsquedas en Twitter usan ANTLR para "*parsear*" las consultas (más de 2 millones de consultas al día).
- Oracle utiliza ANTLR dentro del IDE SQL Developer y sus herramientas de migración.
- El IDE NetBeans analiza C++ con ANTLR.
- Los lenguajes de Apache Hive, Apache Pig, y en general, los sistemas de bodegas de datos de Hadoop, todos utilizan ANTLR.
- Lex Machina1 utiliza ANTLR para la extracción de información de textos legales.
- El lenguaje HQL en el *Hibernate object-relational mapping framework* está construido con ANTLR.
- Entorno de endurecimiento automático de código fuente – SHE.
- Muchos más.

5. Instalación

0. Si aún no está instalada, instalar la máquina virtual de JAVA (JVM versión 1.8 o superior)
    a. En la consola, verificar que funcionan los comandos: java y javac. Si no, incluir en la variable de entorno PATH, el directorio donde están dichos programas.
1. Descargar la versión más reciente de ANTLR (Complete ANTLR 4.7.2 Java binaries jar) desde https://www.antlr.org/download/antlr-4.7.2-complete.jar
2. Guardar el archivo en una ubicación conveniente, por ejemplo: "C:\apps\ANTLR" (sin espacios en blanco)
3. Añadir la ruta al CLASSPATH:

    Windows:

    a. Permanentemente:

    Propiedades del sistema -> Variables de entorno -> Variables del sistema -> CLASSPATH -> Editar: adicionar ".;C:\apps\ANTLR\antlr-4.7.2-complete.jar" (es necesario reiniciar la consola)

    IMPORTANTE: No olvidar el "." al principio de la cadena anterior. Sirve para configurar el directorio actual dentro del CLASSPATH.

    b. Temporalmente, escribiendo en consola:

    ```
    SET CLASSPATH=.;C:\apps\ANTLR\antlr-4.7.2-complete.jar;%CLASSPATH%
    ```

    Unix:

    a. Ejecutar el siguiente comando desde consola o adicionarlo al script de inicio del shell (.bash_profile):

    ```
    $ export CLASSPATH=".:/usr/local/lib/antlr-4.7.2-complete.jar:$CLASSPATH"
    ```

4. En consola, verificar que funciona el siguiente comando:

   java org.antlr.v4.Tool

   Deberá aparecer lo siguiente:

```
ANTLR Parser Generator  Version 4.7.2
 -o ___                 specify output directory where all output is generated
 -lib ___               specify location of grammars, tokens files
 -atn                   generate rule augmented transition network diagrams
 -encoding ___          specify grammar file encoding; e.g., euc-jp
 -message-format ___    specify output style for messages in antlr, gnu, vs2005
 -long-messages         show exception details when available for errors and warnings
 -listener              generate parse tree listener (default)
 -no-listener           don't generate parse tree listener
 -visitor               generate parse tree visitor
 -no-visitor            don't generate parse tree visitor (default)
 -package ___           specify a package/namespace for the generated code
 -depend                generate file dependencies
 -D<option>=value       set/override a grammar-level option
 -Werror                treat warnings as errors
 -XdbgST                launch StringTemplate visualizer on generated code
 -XdbgSTWait            wait for STViz to close before continuing
 -Xforce-atn            use the ATN simulator for all predictions
 -Xlog                  dump lots of logging info to antlr-timestamp.log
```

   Si no funciona correctamente, verifique la instalación antes de continuar. Vuelva al paso 0.

5. Para no tener que escribir este comando todo el tiempo, crearemos comandos de batch que llamen lo que necesitamos:

   Windows:

   - Crear el archivo "**antlr4.bat**" sin "txt" ni ninguna otra extensión.
   - Escribir en el archivo: `java org.antlr.v4.Tool %*`
   - Crear el archivo "**grun.bat**" sin "txt" ni ninguna otra extensión.
   - Escribir en el archivo: `java org.antlr.v4.gui.TestRig %*`
   - Añadir la ruta de estos archivos a la variable de entorno PATH, similar a lo que hicimos en el paso 3 con la variable CLASSPATH.

   Linux:

   - `$ alias antlr4='java -jar /usr/local/lib/antlr-4.7.2-complete.jar'`
   - `$ alias grun='java org.antlr.v4.gui.TestRig'`

6. En la consola, ir al directorio donde vamos a trabajar. Por ejemplo: "C:\workspace"
   a. Ejecutar y verificar que funciona nuevamente:

      java -jar C:\apps\ANTLR\antlr-4.7.2-complete.jar

      antlr4

      Debe aparecer lo mismo que en la prueba anterior.

UNIVERSIDAD
NACIONAL
DE COLOMBIA

6. Estructura de un archivo fuente de ANTLR (especificación de la gramática)

Los archivos con los que trabaja ANTLR tienen la extensión *.g4 (*.g para versiones anteriores), y en adelante los llamaremos archivos de especificación de gramáticas o, directamente, archivos de gramáticas. Contienen la definición de uno o varios analizadores. Cada uno de estos analizadores se traducirá a código de alto nivel (Java, Python o C#, dependiendo de ciertas opciones) en forma de clases. Es decir, por cada analizador descrito en el archivo de gramáticas se generará una clase.

Todo archivo de gramática tiene la siguiente estructura:

```
grammar NombreGramatica;

options {
/* opciones generales a todo el archivo */
}

@header {...}

@members {...}

Reglas
```

Tras **grammar** indicamos el nombre de la gramática, con ello fijamos el nombre que tendrán las correspondientes clases del analizador léxico NombreGramaticaLexer.java y sintáctico NombreGramaticaParser.java (si estamos usando Java como lenguaje de salida). Más adelante se detalla mejor cada uno de los archivos de salida que genera ANTLR.

Options: es la zona de opciones generales. Es opcional. Permite controlar algunos parámetros de ANTLR mediante "opciones". Las opciones se representan como asignaciones: nombreOpcion=valor;. Se utilizan mucho en ANTLR. Las opción más importante de esta zona es la que permite elegir el lenguaje nativo en el que se generarán los analizadores (Java, Python, C#). Su valor por defecto es "JAVA". Tras las opciones generales del archivo vienen las definiciones de analizadores. Es posible que en un mismo archivo se especifiquen varios analizadores. Sin embargo, también es posible definir cada analizador en un archivo aparte, sobre todo cuando se trata de analizadores extensos.

@header es una zona opcional. Delimitada por "**header** {" y "}". Aquí se incluyen los elementos en código nativo (Java, Python o C#) que deben preceder a la definición de las diferentes clases de los analizadores. Esta sección se utiliza para incluir otros archivos (import, etc.), definir el paquete al que pertenecerá la clase del analizador (package), etc.

Por otra parte, **@members** permite insertar atributos y métodos definidos por el usuario en esa clase.

La especificación de las **reglas** se hace con notación gramatical, de la siguiente forma:

ANTECEDENTE: CONSECUENTE;

Importante: ANTLR se ha diseñado para incluir las especificaciones léxica y sintáctica tanto juntas como separadas. El caso de tener ambas especificaciones juntas, se cumple que si el antecedente está escrito en mayúsculas, ANTLR lo interpretará como una regla léxica. Sin embargo, si el antecedente está escrito en minúsculas, ANTLR lo interpretará como una regla sintáctica.

Para ANTLR las minúsculas se asocian a los símbolos no terminales y las mayúsculas a los terminales. Cuando se trabaja con las especificaciones separadas, hay que definir la especificación léxica y la gramática sintáctica. Para la especificación léxica es necesario introducir al principio del archivo que contenga dicha especificación: **lexer gramar NombreArchivo**. Esta sentencia indica a ANTLR que se trata de una especificación léxica. En la especificación sintáctica hay que incluir solamente: **grammar Nombre**, para indicar que se trata de una especificación sintáctica. Además, es necesario importar la especificación léxica correspondiente.

7. Reglas (analizadores léxicos y sintácticos)

- Símbolos léxicos: comienzan con mayúscula
- Símbolos auxiliares (no terminales): comienzan con minúscula
- Comentarios:
  - // Una línea
  - /* varias
    líneas */
- Expresiones regulares:
  - Literales entre comillas simples p.e. 'a'
  - Rangos: 'a'..'z'
  - Negación: ~x
  - Alternativas (o): |
  - 0 o más ocurrencias: *
  - 1 o más ocurrencias: +
  - 0 o 1 ocurrencia: ?

- Reglas:
  - <símbolo léxico o auxiliar> : <definición1>
    | <definición2>
    | <definición_i>
    ;

- Reglas sintácticas en EBNF (Extended Backus-Naur Form): Se permite +, *, ? en las partes derechas de las reglas.

8. Salida ANTLR v4

Ejecutar el siguiente comando en consola, donde PRUEBA.g4 será cualquier gramática con la que vayamos a trabajar:

antlr4 PRUEBA.g4

Se crean como resultado 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.

9. Probar el analizador generado

Para realizar la prueba de un analizador generado con ANTLR v4 usaremos el siguiente ejemplo simple:

Archivo: Hello\Hello.g4

```
grammar Hello;          // Define a grammar called 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, \r (Windows)
```

A partir de la especificación de la gramática Hello.g4 generamos el analizador:

```
>antlr4 Hello.g4
```

Lo cual genera los siguientes archivos:

HelloLexer.java **HelloParser.java**
Hello.tokens **HelloLexer.tokens**
HelloListener.java **HelloBaseListener.java**

Aunque los archivos que genera ANTLR v4 contienen todo lo necesario para ejecutar nuestro analizador, aún no contamos con un programa main que inicie el proceso de procesamiento del lenguaje dada una cadena/archivo de entrada.

Para probar los analizadores generados por ANTLR v4 primero que todo debemos compilar los fuentes generados (java):

```
>javac *.java
```

Esto nos genera los archivos *.class correspondientes y a partir de aquí tenemos 2 opciones: utilizar la herramienta de prueba que proporciona ANTLR llamada TestRig; o desarrollar un programa main para integrar todo.

Opción 1: TestRig

Es una herramienta flexible de prueba que ofrece ANTLR. De acuerdo a las opciones utilizadas puede mostrar información útil acerca del proceso de análisis. Se puede ejecutar así:

```
>grun Hello r –opción
```

Donde:

- **Hello**: es el nombre de la gramática a probar
- **r:** es el símbolo inicial (la primer regla) de la gramática
- **opción:** entre otras, puede ser:

  - **tokens:** si queremos que imprima los tokens que reconoce durante el análisis de una cadena de entrada. Ejemplo:

    ```
    >grun Hello r –tokens
    ⇒ hello unal                # input for the recognizer that you type
    ⇒EOF                        # type ctrl-D on Unix or Ctrl+Z on Windows
    ```

    A lo cual retorna:

    ```
    [@0,0:4='hello',<1>,1:0]
    [@1,6:9='unal',<2>,1:6]
    [@2,12:11='<EOF>',<-1>,2:0]
    ```

    Cada línea de la salida representa la información conocida acerca de un token. Por ejemplo, **[@1,6:9='unal',<2>,1:6]** indica que el token es el segundo (indexado desde 0 - @1), va desde el carácter ubicado en la posición 6 de la línea hasta la 9 (inclusive y empezando en 0), corresponde al lexema 'unal', tiene el token tipo 2 (ID), y está en la línea 1 (contando desde 1) y en la posición 6 (empezando desde 0 y contando las tabulaciones como un solo carácter).

  - **tree:** imprime el árbol de derivación al estilo LISP, en forma textual. Así:

    ```
    >grun Hello r –tree
    ⇒ hello unal
    ⇒EOF
    ```

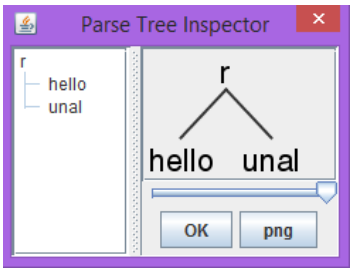
    A lo cual retorna:

    (r hello unal)

  - **gui:** permite ver el árbol de derivación gráficamente. La forma más fácil de verlo. Ejemplo:

    ```
    >grun Hello r –gui
    ⇒ hello unal
    ⇒EOF
    ```

A lo cual retorna:



Opción 2: main

La segunda opción, y la que utilizaremos después de haber hecho las pruebas básicas con TestRig, es realizar la implementación del método **main** del analizador. A continuación se muestra una plantilla para dicho método. Reemplazar NOMBRE_GRAMATICA y reglaInicialGramatica por los identificadores correspondientes:

```
// import de librerias de runtime de ANTLR
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 que se alimenta a partir de la entrada (archivo  o consola)
                NOMBRE_GRAMATICALexer lexer;
                if (args.length>0)
                  lexer = new NOMBRE_GRAMATICALexer(CharStreams.fromFileName(args[0]));
                else
                  lexer = new NOMBRE_GRAMATICALexer(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
                NOMBRE_GRAMATICAParser parser = new NOMBRE_GRAMATICAParser(tokens);
                ParseTree tree = parser.reglaInicialGramatica(); // comienza el análisis en la regla inicial
                System.out.println(tree.toStringTree(parser)); // imprime el árbol de derivación en forma textual
    } catch (Exception e){
                System.err.println("Error (Test): " + e);
    }
  }
}
```

Compilando una vez más todos los archivos (javac *.java), obtenemos el ejecutable de la clase Test (nuestro analizador). Se tiene como resultado el analizador completo que admite un archivo de entrada desde la línea de comandos, lo procesa y emite los mensajes de error correspondientes (en caso que sea necesario).

Por ejemplo, para el caso de Hello.g4, el archivo Test.java contiene el main como se muestra a continuación:

```
// import de librerias de runtime de ANTLR
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{
                        //File f = new File(args[0]);
                        // crear un analizador léxico que se alimenta apartir de la entrada (archivo  o consola)
                        HelloLexer lexer;
                        if (args.length>0)
                                lexer = new HelloLexer(CharStreams.fromFileName(args[0]));
                        else
                                lexer = new HelloLexer(CharStreams.fromStream(System.in));
                        // Identificar al analizador léxico como fuente de tokens para el sintactico
                        CommonTokenStream tokens = new CommonTokenStream(lexer);
                        // Crear objeto del analizador sintáctico que se alimenta apartir del buffer de tokens
                        HelloParser parser = new HelloParser(tokens);
                        ParseTree tree = parser.r(); // begin parsing at init rule
                        System.out.println(tree.toStringTree(parser)); // print LISP-style tree
                } catch (Exception e){
                        System.err.println("Error (Test): " + e);
                }
        }
}
```

A continuación se genera el código del analizador con ANTLR:

>antlr4 Hello.g4

Se compila todo (incluyendo Test.java):

>javac *.java Suponiendo que Test.java está en el mismo directorio con los archivos generador por ANTLR

Se prueba:

>java Test input.in Donde input.in es el archivo de prueba. En este caso contiene la cadena “hello unal”

La salida generada es:

(r hello unal)

10. Ejemplos

Ejemplo 1: Reconocedor de expresiones aritméticas - Expr (sin Listeners ni Visitors)

1. Especificación de la gramática:
   Archivo: Expr.g

```
grammar Expr;

// REGLAS SINTACTICAS
expr        : term ( (MAS | MENOS) term)*
              { System.out.println("Análisis terminado.");
              };

term        : factor ( (MULT | DIV) factor)*;

factor      : ENTERO;


// TOKENS
MAS         : '+';
MENOS       : '-';
MULT        : '*';
DIV         : '/';

// REGLAS LEXICAS
ENTERO      : ('0'..'9')+;

ESPACIO     : ( ' '
      | '\t'
      | '\r'
      | '\n'
      )+ -> channel(HIDDEN)
              ;
```

2. Generar el código fuente del analizador léxico y sintáctico:

   >antlr4 Expr.g

3. Programa main:

   Archivo: MiExpr.java

```
// import de librerias de runtime de ANTLR
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.File;

public class MiExpr {
 public static void main(String[] args) throws Exception {
  try{
        // crear un analizador léxico que se alimenta a partir de la entrada (archivo  o consola)
        ExprLexer lexer;
        if (args.length>0)
         lexer = new ExprLexer(CharStreams.fromFileName(args[0]));
        else
         lexer = new ExprLexer(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
            ExprParser parser = new ExprParser(tokens);
            ParseTree tree = parser.expr(); // 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);
    }
   }
}
```

4. Compilar todo

   >javac ExprLexer.java ExprParser.java MiExpr.java

5. Editar archivo de entrada para probar
   Archivo: entrada.txt
   23+3 -9 *3

6. Probar funcionamiento:

   >java MiExpr entrada.txt
   Análisis terminado.
   (expr (term (factor 23)) + (term (factor 3)) - (term (factor 9) * (factor 3)))

UNIVERSIDAD
NACIONAL
DE COLOMBIA

Ejemplo 2: Reconocedor de números enteros encerrados entre llaves - ArrayInt

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*.

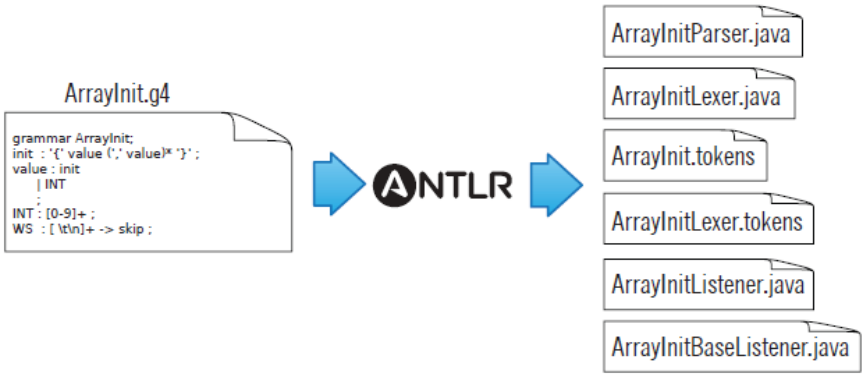
- Especificación de la gramática:

  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          ;
```

- Generar el código fuente usando ANTLR:

  >antlr4 ArrayInit.g4



- Compilar todo

  >javac *.java

- Probar el reconocedor generado usando TestRig:
  - ✓ **Tokens**

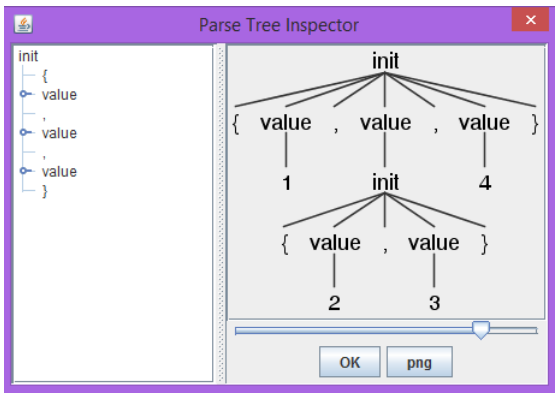
```
>grun ArrayInit init -tokens
{99, 3, 491}
^Z
[@0,0:0='{',<1>,1:0]
[@1,1:2='99',<4>,1:1]
[@2,3:3=',',<2>,1:3]
[@3,5:5='3',<4>,1:5]
[@4,6:6=',',<2>,1:6]
[@5,8:10='491',<4>,1:8]
[@6,11:11='}',<3>,1:11]
[@7,14:13='<EOF>',<-1>,2:0]
```

  - ✓ **Tree**

```
>grun ArrayInit init -tree
{99, 3, 491}
^Z
(init { (value 99) , (value 3) , (value 491) })
```

  - ✓ **Gui** (nótese que no importan los espacios en blanco en la cadena a reconocer)

```
>grun ArrayInit init -gui
{1, {2,      3     },4}
^Z
```



- Integrando el reconocedor generado con un programa en Java (main):

Archivo: Test.java

```
// import de librerias de runtime de ANTLR
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 que se alimenta a partir de la entrada (archivo  o consola)
                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);
    }
  }
}
```

- Compilamos nuevamente para generar el archivo .class de Test:

>javac ArrayInit*.java Test.java

- Probamos el funcionamiento desde consola:

  - ✓ Cadena sin errores sintácticos:

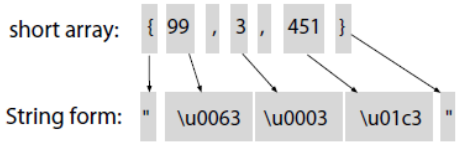
```
>java Test
{1, {2, 3}, 4}
^Z
(init { (value 1) , (value (init { (value 2) , (value 3) })) , (value 4) })
```

  - ✓ Cadena con errores sintácticos:

```
>java Test
{1, 2,
^Z
line 2:0 missing '}' at '<EOF>'
(init { (value 1) , (value 2) <missing '}'>)
```

- **Desarrollar 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).

✓ **Análisis previo:**
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.

✓ **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.

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 hexadecimal 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*). A continuación se presenta la implementación del *Listener* de acuerdo a nuestras reglas de traducción.

✓ **Crear la aplicación (traductor)**

Sólo nos falta crear la aplicación como tal, 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.*;
public class Translate {
        public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input
        // create a lexer that feeds off of input CharStream
        ArrayInitLexer 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**

```
>javac ArrayInit*.java Translate.java

>java Translate
{99, 3, 451}
^Z
"\u0063\u0003\u01c3"
```

UNIVERSIDAD
NACIONAL
DE COLOMBIA

11. Recursos adicionales (recomendados)

- Tutorial ANTLR: **http://www.xfront.com/ANTLR/**
- Revisar (y utilizar) las herramientas disponibles en: http://www.antlr.org/tools.html
  - ✓ **Plugins para: NetBeans, Eclipse, Intellij.**
- 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* y *Visitors* y los analizadores ALL(*). En este enlace: **https://vimeo.com/59285751**
- Listado de gramáticas disponibles para ANTLR**: https://github.com/antlr/grammars-v4**

12. Referencias

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