ANTLR en Visual Studio Code

Maximiliano A. Eschoyez

2021

Resumen

Esta guía tiene como fin explicar la utilización de ANTLR en la IDE Visual Studio Code con Maven como gestor de proyecto. Se explican los pasos mínimos necesarios para la instalación de ANTLR, la generación de diferentes gráficos de análisis y la compilación y ejecución de código fuente.

1. Preparando el Entorno de Trabajo

Para desarrollar los contenidos de la asignatura, vamos a trabajar con ANTLR , Python y Java en la IDE Visual Studio Code. Los proyectos de software los gestionaremos con Maven y usaremos Git para versionado y GitHub como repositorio.

A continuación se explica brevemente como instalar las diferentes herramientas para Debian Linux. Las configuraciones se probaron en Debian versiones 9, 10 y 11.

Recomendación: La instalación de paquetes se debe realizar con el *superusario* o `root`, ya sea haciendo *login*, cambiando de usuario con `su -` o utilizando `sudo`. Sin embargo, por seguridad, es recomendable que la descarga manual de paquetes se realice con un usuario normal.

1.1. Python

Vamos a utilizar Python 3 para programar, que normalmente se encuentra instalado. Entonces. lo primero que debemos realizar es verificar la instalación mediante `apt` como superusario:

```
apt install python3
```

En caso de necesitar instalarlo, aceptamos y esperamos que termine la ejecución.

1.2. Java

Como herramienta base, vamos a necesitar el *Java Developer Kit (JDK)* versión 11. Pueden descargarlo desde la página de Oracle o utilizar OpenJDK, que normalmente se instala con el gestor de paquetes del sistema operativo. Necesitamos Java para poder ejecutar el *plug–in* de ANTLR. Realizamos la instalación mediante `apt` como superusario:

```
apt install openjdk-11-jdk
```

Aceptamos y esperamos que termine la ejecución.

1.3. Git

Para gestionar el versionado y evolución del proyecto, vamos a utilizar la herramienta Git. Como repositorio en la nube el servicio de GitHub. Realizamos la instalación mediante `apt` como superusario:

```
apt install git
```

Aceptamos y esperamos que termine la ejecución.

1.4. Visual Studio Code

Como herramienta de desarrollo vamos a utilizar la IDE Visual Studio Code. Para distribuciones Linux, es conveniente utilizar el gestor de paquetes apropiado (Ej. para Debian). Luego, debemos instalar los *plug–in* necesarios para trabajar:

1. Pyhon
2. Java
3. Maven
4. ANTLR

1.4.1. Instalación

En la página de Visual Studio Code se encuentran disponibles los instaladores para las plataformas soportadas. El paquete Debian para arquitectura linux64 se puede descargar con el navegador (descargar) o utilizando `wget`. Realizamos la descarga con `wget` como usuario normal y la instalación mediante `dpkg` como superusario:

```
wget https://code.visualstudio.com/sha/
          download\?build=stable\&os=linux-deb-x64
dpkg -i code_XX_amd64.deb
```

Aceptamos y esperamos que termine la ejecución. `XX` es el número de versión y *snapshot*. El comando `wget` se muestra en dos líneas por limitación del texto, pero debe escribirse en un único renglón.

La instalación del paquete se realiza con `dpkg` porque no está en los respositorios Debian. Durante la instalación, se genera la entrada necesaria para que las actualizaciones se realicen directamente con `apt`.

1.4.2. Python *plug–in*

El *plug–in* Python instala lo necesario para desarrollar y ejecutar Python.

La instalación se puede realizar de dos formas:

1. con el atajo de teclado `Ctl+Shift+X` o *clickeando* el ícono *Extensions* y buscándolo, o

2. con el atajo de teclado `Ctl+P` para ejecutar en el *VS Code Quick Open* el comando

```
ext install ms-python.python
```

1.4.3. Java *plug–in*

El *plug–in* Java Extension Pack instala todo lo necesario para trabajar con el lenguaje Java.

La instalación se puede realizar de dos formas:

1. con el atajo de teclado `Ctl+Shift+X` o *clickeando* el ícono *Extensions* y buscándolo, o

2. con el atajo de teclado `Ctl+P` para ejecutar en el *VS Code Quick Open* el comando

```
ext install vscjava.vscode-java-pack1
```

1.4.4. Maven *plug–in*

El *plug–in* Maven for Java debería instalarse automáticamente al instalar el Java Extension Pack. Si por algún motivo no se instaló, hacerlo manualmente.

1.4.5. ANTLR *plug–in*

En la página web de ANTLR se pueden encontrar los *plug–in* para diferentes IDEs.

Si bien en Visual Studio Code existen varias herramientas para ANTLR, vamos a utilizar el *plug–in* de Mike Lischke ANTLR4 grammar syntax support (Figura 1).

Figura 1: ANTLR4 grammar syntax support – Mike Lischke

El *plug–in* completo se encuentra publicado con acceso libre en GitHub. Este documento se basa en la documentación del *plug–in ANTLR*.

La instalación se puede realizar de dos formas:

1. con el atajo de teclado `Ctl+Shift+X` o *clickeando* el ícono *Extensions* y buscándolo, o
2. con el atajo de teclado `Ctl+P` para ejecutar en el *VS Code Quick Open* el comando

```
ext install mike-lischke.vscode-antlr4
```

2. Creando Proyecto con Maven

En esta sección se mostrará el paso a paso para crear desde Visual Studio Code un proyecto Maven para ANTLR4. Al finalizar el procedimiento, tendremos el primer proyecto listo para usar.

Se puede clonar el Proyecto Base Listo para Usar desde GitHub, pero es importante leer el resto de este documento para entender mejor su configuración y uso.

2.1. Creación del Proyecto

Para crear el proyecto, se debe acceder al *Command Palette* con las teclas `Ctl+Shift+p` y comenzar a teclear la palabra "Maven". Elegir la opción *"Maven: Create Maven Project"* (Fig. 2).

Maven ofrece varios arquetipos para tomar como estructura del proyecto. Vamos a elegir el *"maven-archetype-quickstart"* (Fig. 3), verión 1.4, porque generará un proyeto simple que luego configuraremos a nuestra necesidad. La configuración del proyecto se encuentra en el archivo `pom.xml`.

Hecho esto, nos pedirá que indiquemos el directorio donde se creará el proyecto. En el directorio indicado, se creará un nuevo directorio con el nombre del proyecto.

Finalmente, nos pedirá de forma interactiva en la terminal, que ingresemos datos del proyecto. Se pueden aceptar la mayor parte de la sugerencias, pero lo importante es indicar correctamente el `groupID` y el `artifactID`. En nuestro caso, usaremos `PrimerProyecto` para ambos campos. El directorio del proyecto se corresponde con el `groupID`. Para más información sobre nombres de proyectos ver la documentación en la página de Maven.

Si el proceso termina correctamente obtendremos un *"Build success"*. El proyecto no se abre automáticamente, por lo tanto, hay abrirlo desde el menú "File" con la opción "Open Folder" o usar la combinación de teclas `Ctl+K Ctl+O`. El proyecto se verá como en la Fig. 4.

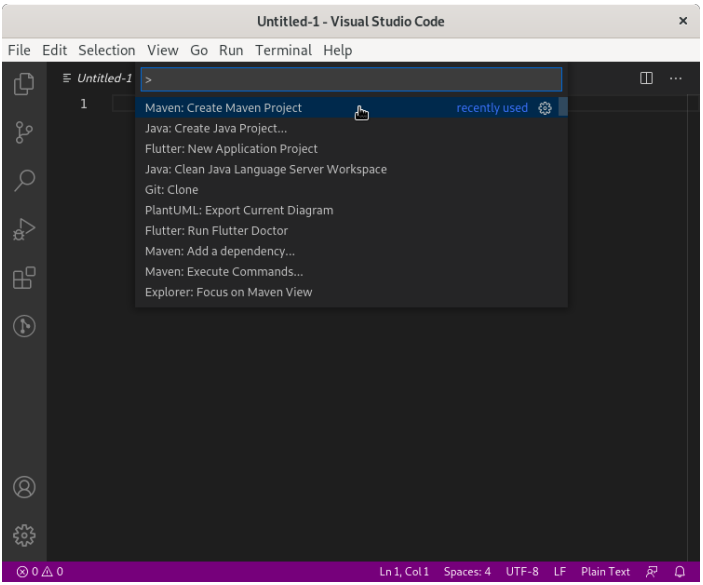

Figura 2: Nuevo proyecto Maven.

2.2. Configuración para Java 11

Para indicar la versión correcta de Java a usar, debemos editar el archivo `pom.xml`. En la sección `properties` pondremos el número de versión Java como puede verse en el Código 1.

Al grabar el archivo, se detectará el cambio y preguntará si se desea actualizar la configuración del proyecto (Fig. 5). Al aceptar, se podrá ejecutar el proyecto abriendo el archivo `App.java` para verificar la correcta configuración (Fig. 6). Si no encontramos ningún inconveniente, entonces el proyecto se encuentra listo para usar Java 11.

2.3. Configuración para ANTLR4

Los gestores de proyectos tienen, entre otros objetivos, resolver las dependencias necesarias para la compilación y ejecución del software con las diferentes bibliotecas de software. Para usar ANTLR4,

Código 1: Sección `properties` del archivo `pom.xml`.

```
<!-- Utilizar version 11 o mayor -->
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.
        sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
```

Código 2: Sección `dependency` del archivo `pom.xml`.

```
<dependencies>
    ...

    <dependency>
    <!-- Utilizar version ANTLR que genera el codigo
        fuente -->
        <groupId>org.antlr</groupId>
        <artifactId>antlr4</artifactId>
        <!-- ultima version Agosto 2021 -->
        <version>4.8</version>
    </dependency>
<dependencies>
```

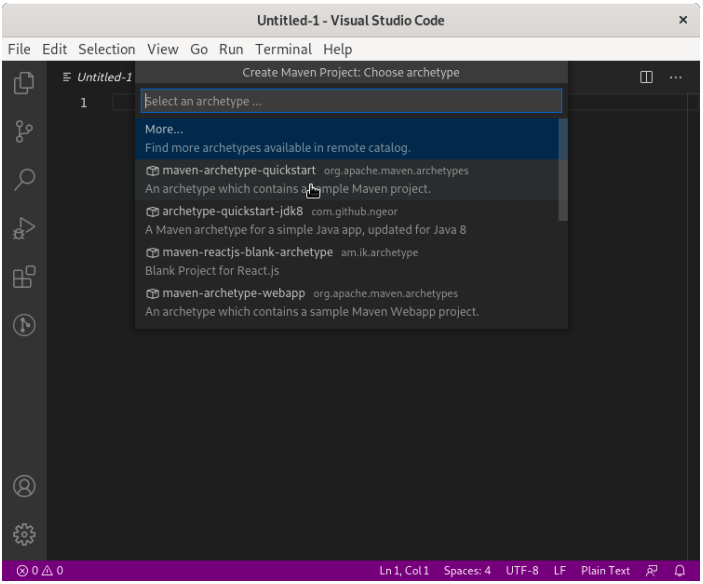

Figura 3: Arquetipo Maven para Java.

debemos configurar su dependencia en el archivo `pom.xml`. El Código 2 se copió de MVNRepository.

Finalizados los pasos indicados, tenemos listo el proyecto Maven para compilar en Java un software que utiliza ANTLR4. En la próxima sección abordaremos como configurar el *plug–in* de ANTLR.

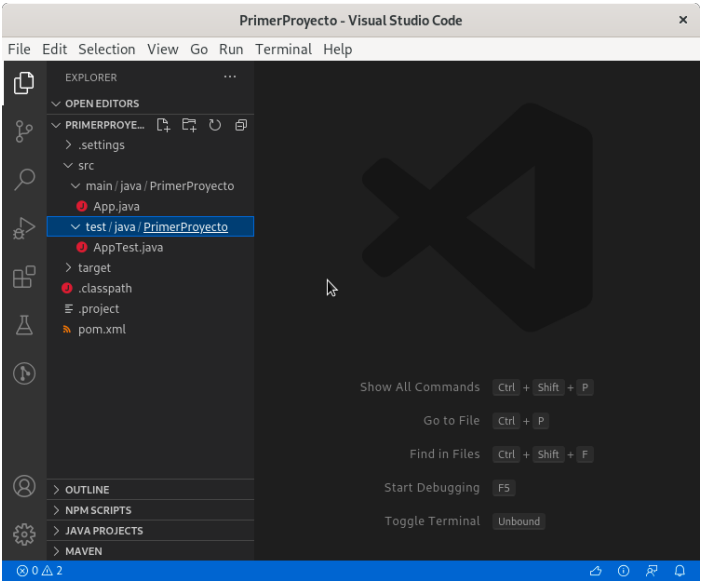

Figura 4: Proyecto Maven para Java recién creado.

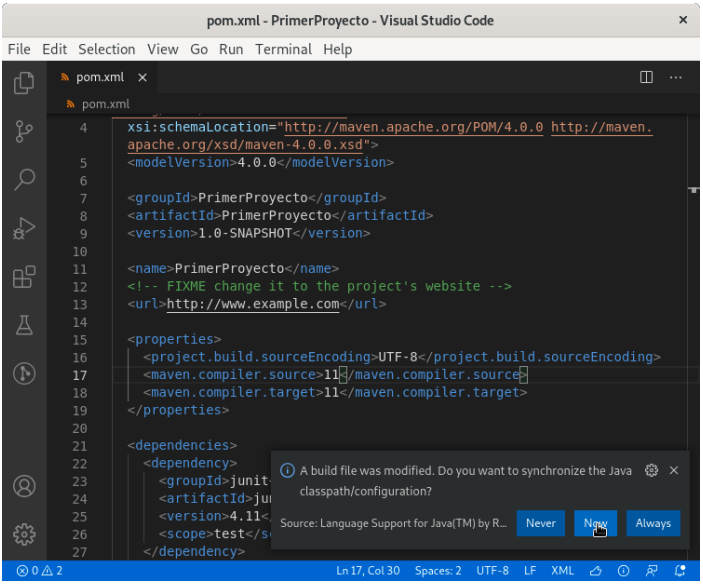

Figura 5: Actualizar configuración proyecto Maven.

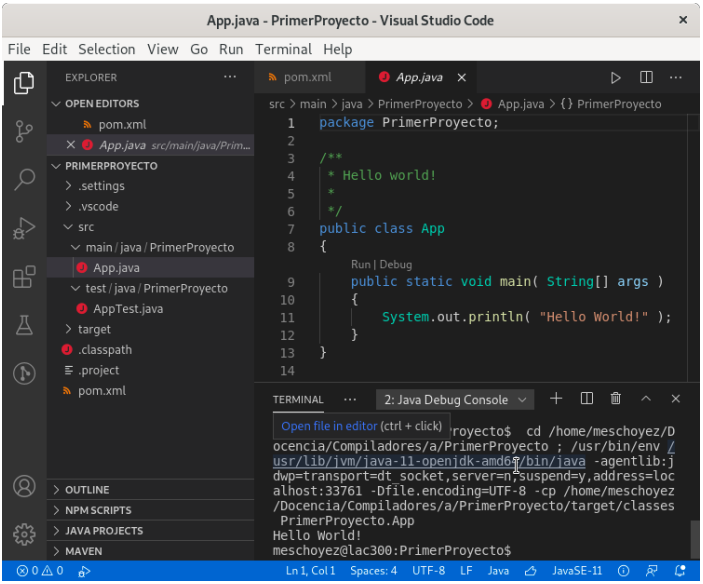

Figura 6: Prueba configuración del proyecto Maven para Java.

Código 3: Sección ANTLR de `launch.json`.

```
{
    "name": "Debug ANTLR4 grammar",
    "type": "antlr-debug",
    "request": "launch",
    // Archivo con la gramatica
    "grammar": "${file}",
    // Archivo a parsear
    "input": "input/entrada.txt",
    // Simbolo inicial
    "startRule": "s",
    // Muestra Arbol Sintactico
    "printParseTree": true,
    "visualParseTree": true
},
```

3. Proyecto con ANTLR

Con el proyecto Java creado y listo para trabajar, vamos a realizar las configuraciones necesarias y luego crear un archivo simple para ANTLR.

3.1. Configuración de ANTLR

Para que la generación de los archivos y *debugging* responda a los requerimientos, es encesario realizar las configuraciones necesarias.

En el Código 3 se muestra la parte del archivo `settings.json` con la configuración del *plug–in* de ANTLR para que se generen automáticamente los archivos correspondientes a los analizadores léxico y sintáctico y los *listeners* y *visitor*. Estos archivos se vuelven a generar cada vez que se guarda el archivo ANTLR, por lo cual, no deben modificarse. Las entradas a configurar y su significado son:

- `name`, `type` y `request` agregan ANTLR a las opciones de depuración,
- `grammar` indica el archivo ANTLR a usar, siendo `${file}` el archivo que tenemos abierto (hacer foco y luego usar `F5`),
- `input` es el archivo a interpretar (ajustar según necesidad),

Código 4: Sección ANTLR de `settings.json`.

```
{
    "java.configuration.updateBuildConfiguration": "
        interactive",
    // Generacion archivos ANTLR en el paquete Java
    "antlr4.generation": {
        "mode": "external",
        "language": "Python3",
        "listeners": true,
        "visitors": true
    },
    "antlr4.debug": {
        "visualParseTreeHorizontal": false,
        "visualParseTreeClustered": true
    }
}
```

- `startRule` es la regla sintáctico o *símbolo* inicial (ajustar según necesidad),
- `printParseTree` y `visualParseTree` se usan para indicar la generación o no del árbol sintáctico en modo texto y gráfico, respectivamente.

Para poder analizar en detalle las reglas, el comportamiento y la ejecución del analizador léxico y el analizador sintáctico se debe configurar el archivo `settings.json` como se muestra en el Código 4. La sección `antlr4.generation` configura la ejecución de ANTLR para la genera de archivos que, por simplicidad, se recomienda usarla así. La sección `antlr4.debug` setea la vista inicial durante la depuración y puede cambiarse durante la visualización.

3.2. Archivos de ANTLR

Los archivos de ANTLR llevan extensión `.g` o `.g4`, pero utilizaremos la segunda opción. El atajo de teclado para crear un archivo vacío es `Ctl+N`, que para hacer efectivo el coloreo hay que guardarlo (`Ctl+S`) con la extensión apropiada.

ANTLR permite la generación del *lexer* y del *parser*. Por lo tanto, los archivos `.g4` pueden ser para el primero, el segundo o ambos combinados. Nosotros utilizaremos archivos combinados dentro del paquete que contiene el método `main()` para facilitar la ejecución y visualización de resultados. En particular, en el proyecto ejemplo guardaremos el archivo `.g4` en la carpeta `src/main/java`.

3.3. Ejemplo Archivo ANTLR

A modo de ejemplo, el Código 5 es un archivo `.g4` del cual se generará el *lexer* y el *parser* que realizan la búsqueda de identificadores tipo Java (nombres de variable, clase o método). Tanto las reglas léxicas como las gramaticales comienzan con un identificador o etiqueta y terminan en punto y coma (`;`). Los dos puntos (`:`) indican el comienzo de la regla y la barra vertical o *pipe* (`|`) separan las distintas reglas alternativas. Las expresiones regulares para la detección de *tokens* se etiquetan con nombres en mayúsculas, como ser `ID` en la línea 10. Las reglas gramaticales se etiquetan con nombres en minúsculas, como ser `s`.

Las palabras reservadas del Código 5 significan lo siguiente:

`grammar` Al comienzo del archivo (línea 1) se indica qué queremos generar, siendo `grammar` la palabra reservada tanto para un *parser* como para un archivo combinado. La otra alternativa sería `lexer`. La palabra `id` es el nombre del *parser*.

`@header` En la línea 3 se utiliza el bloque indicado como `@header` para colocar código fuente en Java que necesitamos que aparezca en todos los código fuente generados por ANTLR. En el ejemplo se consigna solamente el `package` al que pertenecen.

`fragment` En las líneas 7 y 8 la palabra `fragment` indica que la expresión regular se utilizará para construir expresiones regulares más complejas, por lo tanto, no se utiliza durante el análisis léxico.

Cada vez que se grabe el archivo en disco, el *plug–in* de ANTLR regenerará todos los archivos.

Código 5: Ejemplo de archivo `.g4`.

```
grammar id;

@header {
package compiladores;
}

fragment LETRA : [A-Za-z] ;
fragment DIGITO : [0-9] ;

ID : (LETRA | '_')(LETRA | DIGITO | '_')* ;
NUMERO : DIGITO+ ;
OTRO : . ;


s : ID { System.out.println("ID -> " + $ID.getText() + "
    <--"); } s
  | NUMERO { System.out.println("NUMERO -> " + $NUMERO.
      getText() + "<--"); } s
  | OTRO { System.out.println("Otro -> " + $OTRO.getText()
      + "<--"); } s
  | EOF
  ;
```