ANTLRv4

- Installing ANTLR4 in your Linux
- Página nueva

Installing ANTLR4 in your Linux

PDF - Installing ANTLR4 in your Linux (2026-04-15)

Web - Installing ANTLR4 in your Linux (2026-04-15)

Java

Make sure you have Java installed. Check it with

java -version

If you don’t have it, install it normally with your distro package manager.

ANTLR4

Option 1: Using the package manager

Modern Linux distros already include a package for ANTLR, although it is probably too old.

sudo apt-get install antlr4

Option 2: Getting the jar file

Alternatively, you can go to https://www.antlr.org and follow the instructions there, which mainly consist of:

Download antlr4 jar file to /usr/local/lib

cd /usr/local/lib

sudo wget http://www.antlr.org/download/antlr-4.13.2-complete.jar

IMPORTANT: Do NOT use earlier versions than 4.13, the API is different and it will not work with our code.

You will also need to create a command file that will ease the call to Antlr4. Edit a file named antlr4 and write into it:

```
#! /bin/bash

export CLASSPATH=".:/usr/local/lib/antlr-4.13.2-complete.jar:$CLASSPATH"

java -jar /usr/local/lib/antlr-4.13.2-complete.jar "$@"
```

save the file, and give it execution permission:

```
chmod +x antlr4
```

finally, move it to a folder in the default PATH

```
sudo mv antlr4 /usr/local/bin
```

C++ Runtime

Although Antlr4 is a Java-universe project, it can be instructed to generate C++ code. The resulting code uses some C++ classes included in a shared library (the runtime) that is not installed by default with Antlr4. To install it we can follow the steps below:

Option 1: Using the package manager

Some linux distributions have this library packaged. Installing the dev package should pull the binaries too. Make sure the version matches the 4.13 ANTLR version installed.

```
sudo apt-get install libantlr4-runtime-dev
```

Option 2: Compiling from source

If your distribution does not provide this package, you'll need to install it from source:

First, install required tools and dependencies:

```
sudo apt-get install cmake uuid-dev pkgconf
```

Then, download the source from https://www.antlr.org/download.html and compile it.

```
# download source

mkdir antlr-src && cd antlr-src

wget http://www.antlr.org/download/antlr4-cpp-runtime-4.13.2-source.zip

# unzip file

unzip antlr4-cpp-runtime-4.13.2-source.zip

# create build directory for cmake

mkdir build && cd build

# create makefiles

cmake .. -DANTLR_JAR_LOCATION=/usr/local/lib/antlr-4.13.2-complete.jar -
DCMAKE_INSTALL_PREFIX=/usr/local -DWITH_DEMO=True

# build and install

make

sudo make install
```

If you installed Antlr4 from a distribution package, your ANTLR_JAR_LOCATION will be something like /usr/share/java/antlr4-4.13.2.jar. Examine the contents of the package and set the right path in the cmake command.

Adapting Makefiles

Makefiles provided with the class exercises and lab project expect Antlr4 runtime to be in /assig/$USER/cl/runtime

In order to run them in your computer, you need to edit the Makefile and replace line 80

ANTLR_ROOT := /assig/$(USER)/cl/runtime

with

If you installed from distribution packages:

ANTLR_ROOT := /usr

If you installed the jarfile directly, set it to the folder where you created the antlr4 command file

ANTLR_ROOT := /usr/local

Once the Makefile is fixed, you should be able to compile the class examples:

cd intro/lab1/Calc

make antlr

make

./main calc1.txt

Setting up Library Paths

If you get errors when running your ./main program, it may be that the loader is not finding the runtime.

If you installed the runtime in /usr/local/lib you may need to update loader paths:

```
sudo ldconfig
```

If you installed the runtime in some other folder, you need to set the right environment variables so that the loader finds it:

```
export LD_LIBRARY_PATH=/my/runtime/folder
```

You can add this line to your .bashrc to avoid having to do it for every session.

MacOS users: in MacOS the variable name is DYLD_LIBRARY_PATH

Using grun to visualize trees

You can use grun to visualize and debug the trees your grammar is creating.

Details on how to do so are given in the Introduction to ANTLR4 page.

However, to run it in your computer you'll need to set up the paths for the java components of ANTLR4. Edit your $HOME/.bashrc file and add the commands:

```
export ANTLR4=/usr/share/java

if ( $?CLASSPATH ) then

    export CLASSPATH=.:$ANTLR4/antlr-4.13.2-complete.jar:$CLASSPATH

else

    export CLASSPATH=.:$ANTLR4/antlr-4.13.2-complete.jar
```

fi

If you installed antlr4 by getting the jarfile instead of using the package manager, you may need to change the path /usr/share/java to another location (e.g. /usr/local/share/java or wherever you installed it)

Once .bashrc has been updated, open a new terminal to get the new configuration loaded.

Página nueva

Te ayudo a generar el código en C++ para tu gramática de ABB y a entender la diferencia entre Listener y Visitor.

1. Generación del Código C++ con ANTLR

Primero, descarga los archivos abbLexer.g4 y abbParser.g4 en el mismo directorio. Luego, ejecuta este comando en la terminal para generar el código C++ :

```bash
antlr4 -Dlanguage=Cpp -visitor -listener abbLexer.g4 abbParser.g4
```

Esto creará una serie de archivos .h y .cpp. Los más importantes son:

· abbLexer.h/cpp: El Analizador Léxico.

· abbParser.h/cpp: El Analizador Sintáctico.

· abbBaseListener.h/cpp y abbBaseVisitor.h/cpp: Clases base para tu propio código.

2. Crear un main.cpp para probar un programa

Para probarlo, necesitas un archivo main.cpp que lea un archivo de entrada, lo procese y muestre el árbol sintáctico. Aquí tienes una plantilla que puedes usar :

```cpp
#include <iostream>
#include <fstream>

#include "antlr4-runtime.h"
#include "abbLexer.h"
#include "abbParser.h"

using namespace antlr4;

int main(int argc, const char* argv[]) {
    if (argc < 2) {
        std::cerr << "Uso: " << argv[0] << " <archivo_de_entrada>" << std::endl;
        return 1;
    }

    // 1. Crear el flujo de entrada desde el archivo
    std::ifstream stream;
    stream.open(argv[1]);
    if (!stream) {
        std::cerr << "Error: No se pudo abrir el archivo " << argv[1] << std::endl;
        return 1;
    }
```

```
    // 2. Crear el lexer y el parser
    ANTLRInputStream input(stream);
    abbLexer lexer(&input);
    CommonTokenStream tokens(&lexer);
    abbParser parser(&tokens);

    // 3. Obtener el token de errores e iniciar el parsing
    parser.removeErrorListeners();
    parser.addErrorListener(new ConsoleErrorListener());

    tree::ParseTree* tree = parser.module_(); // 'module_' es la regla principal en abbParser.g4

    // 4. Imprimir el árbol sintáctico en formato LISP
    std::cout << "Árbol Sintáctico: " << tree->toStringTree(&parser) << std::endl;

    return 0;
}
```

3. Para qué sirven Listener y Visitor

Ambos son patrones de diseño para recorrer el árbol sintáctico y ejecutar tu propia lógica (como calcular valores, comprobar errores o traducir el código), pero su filosofía es distinta .

Cuándo usar cada uno:

· Visitor (Visitante): Te da el control total. Debes llamar explícitamente a visit() para recorrer los nodos hijos. Es ideal si necesitas retornar valores de los nodos (ej. al evaluar una expresión matemática) y quieres controlar el orden de la visita .

· Listener (Oyente): Es automático. El ParseTreeWalker de ANTLR recorre el árbol y va llamando a los métodos enterRegla y exitRegla . Es muy útil cuando necesitas ejecutar acciones antes y después de procesar un bloque, como manejar alcances de variables (entrar/salir de una función) o transformar el formato de salida.

Resumen práctico:

· Si estás creando un intérprete que calcula valores, te conviene Visitor.

· Si estás haciendo un traductor o analizador de alcances, el Listener puede ser más sencillo.

Importante: Para compilar, necesitarás la librería de runtime de ANTLR para C++. Si usas CMake, te recomiendo el script de este repositorio de ejemplo, que descarga y compila el runtime por ti automáticamente .

Espero que esto te aclare los pasos. Si tienes una duda más concreta sobre qué patrón usar para tu proyecto, puedes decírmelo

Por supuesto. Aquí tienes una versión más divulgativa y estructurada, lista para ser usada como la primera parte de un artículo de blog sobre ANTLR y C++.

De la Gramática al Código: Tu Primer Compilador con ANTLR y C++

Si alguna vez te has enfrentado a la tarea de leer un archivo de configuración extraño, un lenguaje de juguete o un formato de datos propietario, sabrás lo tedioso que es escribir un analizador (parser) a mano contando paréntesis y comas. Ahí es donde brilla ANTLR (ANother Tool for Language Recognition).

En esta primera entrada, vamos a tomar una gramática real de un robot industrial (ABB RAPID) y la vamos a convertir en un programa funcional en C++ que lea y valide archivos .sys. Veremos cómo generar el esqueleto del compilador y, lo más importante, entenderemos para qué demonios sirven el Listener y el Visitor (las dos grandes superestrellas o grandes dolores de cabeza de ANTLR).

1. Paso 1: Las Materias Primas (Lexer y Parser)

Partimos de dos archivos que definen el lenguaje:

· abbLexer.g4: Define cómo se parten las palabras y símbolos (módulos, :=, PROC, números).

· abbParser.g4: Define la gramática, es decir, el orden lógico de esas palabras (un módulo contiene datos, procedimientos, etc.).

2. Paso 2: Invocando al Mago (Generación de Código C++)

El primer paso es casi mágico. Con los archivos .g4 en una carpeta, abrimos la terminal y lanzamos el hechizo de ANTLR para C++:

```bash
antlr4 -Dlanguage=Cpp -visitor -listener abbLexer.g4 abbParser.g4
```

¿Qué hace este comando?

· -Dlanguage=Cpp: Le dice a ANTLR que no queremos Java ni Python, queremos clases C++ de alto rendimiento.

· -visitor -listener: Le pedimos que nos prepare el terreno para ambas estrategias de recorrido del árbol (luego veremos la diferencia).

Tras ejecutar esto, la carpeta se llenará de archivos .h y .cpp. No te asustes, tú solo vas a tocar main.cpp.

3. Paso 3: El Punto de Entrada (main.cpp)

Necesitamos un programa que coja un archivo de texto, se lo pase a nuestro recién nacido analizador y nos diga si es válido o no. Aquí tienes el main.cpp mínimo viable para C++:

```cpp
#include <iostream>
#include <fstream>
#include "antlr4-runtime.h"
#include "abbLexer.h"
#include "abbParser.h"

using namespace antlr4;

int main(int argc, const char* argv[]) {
    // 1. Abrir el archivo de ejemplo
    std::ifstream stream(argv[1]);
```

```
    ANTLRInputStream input(stream);

    // 2. Fase Léxica: Romper el texto en Tokens
    abbLexer lexer(&input);
    CommonTokenStream tokens(&lexer);

    // 3. Fase Sintáctica: Construir el Árbol
    abbParser parser(&tokens);
    tree::ParseTree* tree = parser.module_(); // 'module_' es la regla raíz

    // 4. Imprimir el resultado
    std::cout << "Árbol generado: " << tree->toStringTree(&parser) << std::endl;

    return 0;
}
```

Si compilas este código (enlazando con la librería de runtime de ANTLR) y le pasas un archivo .sys válido, verás en consola un montón de paréntesis representando la estructura jerárquica del código. ¡Ya tienes un reconocedor sintáctico!

4. El Dilema: ¿Listener o Visitor?

Aquí viene la parte filosófica. Ya sabemos leer el archivo, pero ahora queremos hacer algo con esa información (traducirlo a otro lenguaje, calcular posiciones del robot, pintar un diagrama...). ANTLR nos da dos herramientas para recorrer ese árbol que acabas de imprimir. Elegir bien te ahorrará muchas horas de depuración.

Característica	Listener (El Piloto Automático)	Visitor (El Conductor Manual)
Recorrido	Automático. Un "Andador" (ParseTreeWalker) recorre cada nodo en profundidad.	Manual. Tú escribes visit() explícitamente para navegar a los hijos que quieras.
Control de Flujo	No puedes pararlo. Siempre visita todos los nodos.	Tú decides si visitas el hijo izquierdo, el derecho o ninguno.
Retorno de Datos	No devuelves valores directamente (normalmente usas variables externas o una pila).	Sí devuelve valores. Ideal para expresiones matemáticas (visitSuma devuelve un int).
Caso de Uso Ideal	Traducción de formato (ej. pasar de ABB a Python) o Análisis de alcance (entrar/salir de una función).	Evaluación/Interpretación directa o Análisis Semántico que requiere calcular tipos complejos.

En resumen para tu blog:

· Si tu objetivo es imprimir el código en otro lenguaje o contar cuántas veces aparece la palabra PROC, usa Listener.

· Si tu objetivo es ejecutar el programa del robot en un simulador virtual (hacer que el robot se mueva según las coordenadas del archivo), usa Visitor.

5. Próximos Pasos

En la siguiente parte de esta serie, implementaremos un Visitor personalizado para extraer todas las declaraciones de variables TOOLDATA del archivo del robot y generar un archivo JSON con la configuración. ¡Nos vemos en el código!