# Introducción para trabajar con ANTLR v3 y con ant

José Miguel Rivero Almeida

## Índice



## 1. Descarga e instalación de ANTLR-v3

1. Bajad de www.antlr.org uno de los siguientes ficheros .jar, según queráis trabajar con o sin la interfície gráfica antlrworks. También podéis bajar los dos:

   - Para trabajar SIN la interfície gráfica antlrworks: bajad el fichero antlr-3.4-complete.jar
     *Complete ANTLR 3.4 Java binaries jar (complete ANTLR 3.4 tool, Java runtime, ST 3.2.1, ANTLR v2, and ST 4.0.4; for use when you use output=template)*
   - Para trabajar CON la interfície gráfica antlrworks: bajad el fichero antlrworks-1.4.3.jar
     *Version 1.4.3 - for Windows, Linux and Mac OS X*

2. Cread el directorio antlr-v3/lib en vuestro ${HOME}:

   ```
   $ mkdir -p ${HOME}/antlr-v3/lib
   ```

3. Llevad allí los ficheros antlr-3.4-complete.jar y/o antlrworks-1.4.3.jar

4. Definid la variable de *environment* ANTLR_HOME para que señale el camino al directorio ${HOME}/antlr-v3, y también añadid a la variable CLASSPATH el camino hasta los ficheros antlr-3.4-complete.jar y antlrworks-1.4.3.jar.
   Dependiendo del *shell* con el que trabajéis (*tcsh* o *bash*), tendréis que escribir unas líneas al final del fichero de configuración correspondiente (∼/.tcshrc o ∼/.bashrc).
   Para conocer el *shell*:

   ```
   $ echo ${SHELL}
   ```

   a) Si es con *tcsh* (*csh*), añadid el siguiente código al final de ${HOME}/.tcshrc

```
setenv ANTLR_HOME ${HOME}/antlr-v3
if (${?CLASSPATH}) then
    setenv CLASSPATH ${ANTLR_HOME}/lib/antlrworks-1.4.3.jar:${CLASSPATH}
    setenv CLASSPATH ${ANTLR_HOME}/lib/antlr-3.4-complete.jar:${CLASSPATH}
    setenv CLASSPATH .:${CLASSPATH}
else
    setenv CLASSPATH ${ANTLR_HOME}/lib/antlrworks-1.4.3.jar
    setenv CLASSPATH ${ANTLR_HOME}/lib/antlr-3.4-complete.jar:${CLASSPATH}
    setenv CLASSPATH .:${CLASSPATH}
endif
```

b) Si es con *bash*, añadid el siguiente código al final de `${HOME}/.bashrc`

```
export ANTLR_HOME=${HOME}/antlr-v3
if [ -z "${CLASSPATH}" ]; then
    export CLASSPATH=${ANTLR_HOME}/lib/antlrworks-1.4.3.jar:${CLASSPATH}
    export CLASSPATH=${ANTLR_HOME}/lib/antlr-3.4-complete.jar:${CLASSPATH}
    export CLASSPATH=.:${CLASSPATH}
else
    export CLASSPATH=${ANTLR_HOME}/lib/antlrworks-1.4.3.jar
    export CLASSPATH=${ANTLR_HOME}/lib/antlr-3.4-complete.jar:${CLASSPATH}
    export CLASSPATH=.:${CLASSPATH}
fi
```

Comprobad que el valor de la variable `CLASSPATH` es correcto cuando entréis en un nuevo terminal, o bien, sin cambiar de terminal, ejecutad el fichero de configuración:

```
$ source ~/.tcshrc            (o $ source ~/.bashrc)
$ echo ${CLASSPATH}
```

## 1.1. Para utilizar la interfície gráfica `antlrworks` [opcional]

Si habéis bajado `antlrworks-1.4.3.jar`, para invocarlo debéis escribir:

```
$ java -jar ${ANTLR_HOME}/lib/antlrworks-1.4.3.jar
```

Pero también podéis crear un pequeño *script* para hacerlo más rápido:

1. Cread el directorio `${HOME}/antlr-v3/bin`

   ```
   $ mkdir ${HOME}/antlr-v3/bin
   ```

2. Copiad el siguiente texto en un fichero de nombre `antlrworks` y llevadlo a dicho directorio:

   ```
   #!/bin/sh
   java -jar ${ANTLR_HOME}/lib/antlrworks-1.4.3.jar
   ```

3. Permitid que se pueda ejecutar:

   ```
   $ chmod u+x ${ANTLR_HOME}/bin/antlrworks
   ```

4. Añadid el directorio `${ANTLR_HOME}/bin` a la variable `PATH`, de forma similar a como se hizo con `CLASSPATH` y dependiendo de con que *shell* trabajáis:

   a) Si es con *tcsh* (*csh*), añadid el siguiente código al final de `${HOME}/.tcshrc`

   ```
   setenv PATH ${ANTLR_HOME}/bin:${PATH}
   ```

   b) Si es con *bash*, añadid el siguiente código al final de `${HOME}/.bashrc`

   ```
   export PATH=${ANTLR_HOME}/bin:${PATH}
   ```

5. Comprobad que funciona:

   ```
   $ source ~/.tcshrc            (o $ source ~/.bashrc)
   $ antlrworks
   ```

## 2. Uso de ANTLR desde línea de comandos

Una vez escrito un cierto *fichero*.g con la gramática y los tokens del lenguaje, podéis invocar a ANTLR para que genere el analizador léxico (*lexer*) y el sintáctico (*parser*)

```
$ java -cp /camino/hasta/antlr-3.4-complete.jar org.antlr.Tool fichero.g
```

Como en la variable CLASSPATH ya se ha incorporado ese camino, podéis simplemente escribir:

```
$ java org.antlr.Tool fichero.g          (OJO! Tool con mayúscula)
```

Ésto generará, entre otros, los ficheros *fichero*Lexer.java y *fichero*Parser.java. El programa principal (*main*) que llama al *parser* (para que lea y analize la entrada) se escribirá en una clase aparte, por ejemplo en el fichero *fichero*Main.java Debéis compilar este último fichero junto a los generados previamente por ANTLR:

```
$ javac fichero*.java
```

Se obtienen varios ficheros .class, entre ellos *fichero*Main.class, que contiene el método *main*. Para ejecutar este método, que leerá y analizará la entrada escribiréis, según sea el caso:

a) Si la entrada se lee de *System.in* (entrada standard) y la introducís a través del teclado:

```
$ java fichero Main
```

y picad la entrada. Para finalizar escribid <CTRL-D> tras un <RETURN>

b) Si se lee de *System.in*, pero utilizáis un fichero donde previamente habéis escrito la entrada y la redireccionáis:

```
$ java fichero Main < fichero_con_la_entrada
```

c) Si se lee de un fichero cuyo nombre ponéis como argumento en la propia orden java

```
$ java fichero Main fichero_con_la_entrada
```

## 3. Uso de ANTLR con ant

*Ant* es una herramienta para desarrollar aplicaciones *Java*. De forma similar a *make*, permite definir cómo es el proceso de generación de ficheros Java, cómo se debe realizar su compilación, su ejecución o el posterior test de la aplicación en base a diferentes juegos de prueba (utilizando *junit*).
*Ant* necesita que especifiquemos los diferentes objetivos (*targets*) que tenemos, cuáles son las dependencias entre ellos, y cómo lograr cada uno de esos objetivos. Todo ésto lo realizamos en un fichero en formato *xml* llamado build.xml.

### 3.1. Preparación y configuración de ant

1. Cread el directorio ${HOME}/.ant/lib:

   ```
   $ mkdir -p ${HOME}/.ant/lib
   ```

2. Llevad allí la lista de ficheros que aparece más abajo **a menos que** ya esten en el directorio *home_ant*/lib, habitualmente /usr/share/ant/lib. Podeis averiguar cual es ese directorio ejecutando:

   ```
   $ ant -diagnostics | egrep ant.home
   ```

   Tambiés es posible que alguno de estos ficheros exista en el directorio de jars de Java (usualmente /usr/share/java). En este caso tampoco será necesario colocarlo en nuestro nuevo directorio siempre que se pueda definir un enlace simbólico dentro de /usr/share/ant/lib hasta /usr/share/java con el nombre del fichero. Para ello se necesitan privilegios de administrador.

   Los ficheros necesarios son:

- `ant-antlr3.jar` Necesario para usar ANTLR-v3 desde ant y generar el scanner y el parser
- `ant-junit.jar` Necesario para pasar los tests con los juegos de prueba
- `junit-4.4.jar` Necesario para pasar los tests con los juegos de prueba
- `hamcrest-core.jar` Necesario para generar informes con los resultados de los tests

3. Hay que modificar el fichero de *startup* del *shell*, para incorporar a la variable `CLASSPATH` estos nuevos ficheros

   a) Si el *shell* es *tcsh*, hay que añadir al final de `${HOME}/.tcshrc`

   ```
   setenv ANT_LOCAL ${HOME}/.ant
   setenv CLASSPATH ${CLASSPATH}:${ANT_LOCAL}/lib/ant-antlr3.jar
   setenv CLASSPATH ${CLASSPATH}:${ANT_LOCAL}/lib/ant-junit.jar
   setenv CLASSPATH ${CLASSPATH}:${ANT_LOCAL}/lib/junit-4.4.jar
   setenv CLASSPATH ${CLASSPATH}:${ANT_LOCAL}/lib/hamcrest-core.jar
   setenv CLASSPATH classes:${CLASSPATH}
   ```

   b) Si el *shell* es *bash*, hay que añadir al final de `${HOME}/.bashrc`

   ```
   export ANT_LOCAL=${HOME}/.ant
   export CLASSPATH=${CLASSPATH}:${ANT_LOCAL}/lib/ant-antlr3.jar
   export CLASSPATH=${CLASSPATH}:${ANT_LOCAL}/lib/ant-junit.jar
   export CLASSPATH=${CLASSPATH}:${ANT_LOCAL}/lib/junit-4.4.jar
   export CLASSPATH=${CLASSPATH}:${ANT_LOCAL}/lib/hamcrest-core.jar
   export CLASSPATH=classes:${CLASSPATH}
   ```

4. Ahora ya podéis trabajar con `ant`. La estructura del fichero `build.xml` es fácil de entender. Los principales objetivos (*targets*) se muestran con

   `$ ant -p` (o `$ ant -projecthelp`)

   Normalmente el objetivo por defecto se llama *compile*, y consiste en invocar en primer lugar a `ANTLR` para que trate el *`fichero`*`.g` y genere los ficheros `.java` con el *lexer* y el *parser*, y, a continuación, se compilan todos los ficheros `.java`.

   Se han definido ficheros `build.xml` para cada apartado del *tutorial* y obviamente también, aunque algo más complejo, para la *práctica* (compilador de `CL`). Comentaremos ambos casos por separado

## 3.2. Uso de `ant` en el *tutorial*

1) Para invocar a `ANTLR` generando, a partir del fichero `Example`*`X`*`.g`, el lexer (`Example`*`X`*`Lexer.java`) y el parser (`Example`*`X`*`Parser.java`), y posteriormente compilar estos ficheros, junto con el que contiene la clase con el *main* (`Example`*`X`*`Main.java`):

   `$ ant` (o `$ ant compile`)

   O si solo queréis generar el *lexer* y el *parser*, pero no compilar:

   ```
   $ ant parser
   ```

2) Para hacer todo lo anterior y ejecutar a continuación el método *main* que es quien llama al *parser* para que lea, analize y trate la entrada.
   Previamente tenemos que haber escrito esa entrada en un fichero cuyo nombre se nos pide:

   ```
   $ ant run
   ```

   También podemos dar ese nombre en la misma orden `ant`, definiendo el valor de la propiedad `infile`, por ejemplo si la expresión se encuentra en el fichero `expr3`:

   ```
   $ ant run -Dinfile=expr3
   ```

   La ejecución genera varios ficheros:

- `output`, de texto, con los mensajes de error, si hay, o si no, con el *AST* si éste se ha generado y visualizado, y también con el resultado de la evaluación/interpretación de la entrada, si ésta se ha producido (en todos los casos excepto en `Example0`)
- `ast.dot`, con el *AST* en formato *dot* (`graphviz`)
- `ast.ps`, con la traducción del fichero anterior a *postscript*

Además de la ejecución, si el *AST* ha sido generado, se lanza un visualizador del mismo con el comando `gv`.

3) Para borrar los ficheros generados por la compilación y la ejecución:

```
$ ant clean
```

**Llegados a este punto ya podéis comenzar con el tutorial guiado de ANTLR-v3 escrito en el fichero `intro-antlr-v3.pdf`. El resto de este documento lo podéis volver a consultar cuando vayáis a comenzar con la práctica.**

## 3.3. Uso de `ant` en la *práctica*

¿Cómo podemos usar `ant` para generar automáticamente el compilador de `CL`? Siempre desde el directorio donde se encuentra el fichero `build.xml`:

1) Para generar a partir de `CL.g`[1] el *parser* (fichero `CLParser.java`) y el *lexer* (fichero `CLLexer.java`) y después compilar todos los ficheros Java:

```
$ ant            (o   $ ant compile)
```

Los ficheros `.class` no quedan en el directorio actual, sino en el subdirectorio `classes`. El fichero con el *main* es `CL.class` (fruto de compilar `CL.java`)

2) Para ejecutar a continuación nuestro compilador de `CL`, es decir, para compilar un programa en lenguaje `CL` tenemos dos opciones: a) lo podemos hacer directamente, llamando a la clase que contiene el *main*, o b) indirectamente a través de `ant`:

   a) Directamente, invocando a `java` desde la línea de comandos. Por ejemplo, para compilar el fichero `CL test-data/jps/jp00_cl.txt`

   ```
   $ java CL test-data/jps/jp00_cl.txt
   ```

   Ved lo que ocurre con este otro juego de pruebas (`jp20_cl.txt`):

   ```
   $ java CL test-data/jps/jp20_cl.txt
   ```

   En el primero la compilación se detuvo en el *typecheck* porque se encontraron errores semánticos en ese juego de pruebas. En el segundo la compilación llegó hasta el final y vemos el *t-code* generado.

   Podemos especificar diferentes *step*'s (puntos donde la compilación se detiene y, eventualmente, muestra información). En total hay 11 *step*'s (si no especificamos nada llegará hasta el número 8, a menos que se produzcan errores en etapas previas).
   El significado de estos *step*'s es el siguiente:

   - *step* 1: Se hace el análisis léxico-sintáctico y muestra los errores léxicos (excepciones del *lexer*) que se produjeron
   - *step* 2: Se hace el análisis léxico-sintáctico y muestra los errores sintácticos (excepciones del *parser*) que se produjeron
   - *step* 3: Se hace el análisis léxico-sintáctico y muestra el *AST* creado por el *parser*

[1] En realidad se tratan también los ficheros `TypeCheck.g` para obtener el analizador semántico (fichero `TypeCheck.java`), y el fichero `CodeGen.g` para obtener la etapa de generación de código (fichero `CodeGen.java`).

- *step* 4: Se hace el análisis léxico-sintáctico y el semántico, y muestra las excepciones producidas durante el recorrido del $AST$ realizado por el *typecheck*
- *step* 5: Se hace el análisis léxico-sintáctico y el semántico, y muestra el $AST$ tras ser decorado por el *typecheck*
- *step* 6: Se hace el análisis léxico-sintáctico y el semántico, y muestra los errores semánticos detectados en el *typecheck*
- *step* 7: Se hace el análisis léxico-sintáctico, el análisis semántico y la generación de código, y muestra las excepciones producidas durante el recorrido del $AST$ realizado por *codegen*
- *step* 8: Se hace el análisis léxico-sintáctico, el análisis semántico y la generación de código, y muestra el *t-code* generado por esta última etapa
- *step* 9: Se hacen todas las etapas de análisis del programa *CL*, la generación de *t-code*, y un analizador léxico-sintáctico posterior comprueba la corrección de ese *t-code*
- *step* 10: Se hacen todas las etapas de análisis del programa *CL* i la generación de *t-code*, i el análisis léxico, sintáctico y también semántico del *t-code*
- *step* 11: Se hacen todas las etapas de análisis del programa *CL* y la generación de *t-code*; se analiza la correción del *t-code*, y finalmente un intérprete lo ejecuta mostrando los resultados obtenidos.

Para especificar el *step* 3 y que el compilador nos muestre el $AST$:

```
$ java CL -3 test-data/jps/jp00_cl.txt
```

b) También se puede ejecutar el compilador de `CL` indirectamente a través de `ant` (objetivo `run`) introduciendo después la información requerida: el fichero `CL` (*infile*) a compilar y el *step* hasta el que llegar (por defecto *step*=8).

```
$ ant run
```

También se pueden especificar el valor que les damos en la propia llamada a `ant`, por ejemplo:

```
$ ant run -Dinfile=mijp5 -Dstep=3
```

`Ant` pedirá aquellas propiedades no definidas en la línea de comandos. El fichero `output` guarda la salida de la ejecución con los errores/excepciones producidos o los resultados asociados al *step* seleccionado.

También podemos hacer una ejecución y visualizar el $AST$ generado en formato *postscript* (con `gv`). Se nos pide el nombre del fichero a compilar (como antes, lo podemos dar en la misma línea). Aquí *step* vale forzosamente 3 (construir y mostrar $AST$).

```
$ ant ast
```

3) Por último se puede utilizar *ant* con *junit*, y con todo un conjunto de ficheros Java (dentro del subdirectorio `test-junit`) que contienen tests para comprobar qué juegos de prueba se pasan y hasta qué punto cada uno de ellos.
Se comprueban los juegos de prueba desde el `test-data/jps/jp00_cl.txt` al `jp16_cl.txt` (que contienen errores semánticos), y desde el `jp20_cl.txt` al `jp34_cl.txt` (sin errores semánticos y que por tanto generan *t-code*)

```
$ ant run-tests
```

Se generará un *report* con los resultados en un fichero *html* situado debajo del directorio `reports`. La `URL` del fichero con el *report* sería de la forma:

`file:///`*`camino_hasta_el_directorio_del_build_xml`*`/reports/html/index.html`

*Ant* además lanza un proceso que visualiza ese fichero (usa el navegador `firefox`).

# 4. Uso del *debugger* en `antlrworks`

Podemos utilizar el *debugger* de `antlrworks` para comprobar paso a paso cómo evoluciona el análisis sintáctico (*parsing*), y así poder corregir errores en la gramática.

- Podemos llamar al *parser* directamente desde `antlrworks` y debuguear.

  `antlrworks` → `Run` → `Debug`

  En este caso, `antlrworks` se encarga de llamar a `ANTLR` para que trate la gramática, después compilar el *lexer* y el *parser* generados, y llamar al parser para que analize la entrada que le daremos.

- Pero también podemos llamarlo indirectamente, a través de la clase que contiene el *main*, que es desde donde en realidad se llama al *parser*.

  `antlrworks` → `Run` → `Debug Remote...`

  En este último caso, el *debugger* `antlrworks` controla la evolución del análisis sintáctico comunicándose con el proceso que ejecuta el *main* a través del *port* 49100.

Para que el *debugger* remoto funcione correctamente debemos seguir los siguientes 4 pasos:

1. Invocar a `antlr` para que genere el *lexer* y el *parser* con la opcion `-debug`

   ```
   $ java org.antlr.Tool -debug Example1.g
   ```

2. Compilar el *lexer* y el *parser* generados, y también la clase con el *main*

   ```
   $ javac *.java
   ```

3. Invocar a la clase del *main* que llama al *parser* pasándole la expressión (entrada) a analizar (que en este caso está en el fichero `expr`) **y dejar el proceso en background (&)**

   ```
   $ java Example1Main <expr &
   ```

4. Llamar a `antlrworks` y hacer *Remote Debug*

   ```
   $ antlrworks
         ↪ Run
             ↪ Debug Remote
                 ↪ Address: localhost
                 ↪ Port:    49100          (OJO! no el 49153)
                 ↪ Connect
   ```

   Ahora ya podemos ir paso a paso en el *debugger* comprobando cómo se realiza el análisis sintáctico, y viendo cómo se construye el árbol correspondiente.

   El puerto por defecto es el 49153. Para cambiarlo:

   ```
   $ antlrworks
         ↪ File
             ↪ Preferences
                 ↪ Debugger
                     ↪ Default local port:  49100
                 ↪ Apply
   ```