UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

GRADO EN INGENIERÍA EN INFORMÁTICA

HERRAMIENTA ANTLR

Asignatura: Teoría de Autómatas y Computación

Grupo de Trabajo: E01

Fecha: 30/04/2024

Participantes:

Elena Ballesteros Morallón

Samuel Espejo Gil

Adrián Ramos Romero

Contenido

INTRODUCCIÓN

La capacidad de analizar y procesar lenguajes es fundamental para el desarrollo de software. Así, ANTLR surge como una herramienta que ofrece una solución robusta para esta tarea. En este trabajo veremos una descripción general, su funcionamiento básico, desde la definición del léxico hasta el sintáctico, y su implementación práctica en Visual Studio Code.

DESCRIPCIÓN DE LA HERRAMIENTA

ANTLR (ANother Tool for Language Recognition) es un generador de analizadores sintácticos y léxicos utilizado para leer, procesar o traducir archivos estructurados de texto o binarios.

El proyecto de ANTLR comenzó incorporando solo el lenguaje **Java**, que hoy en día sigue siendo el lenguaje de referencia. Con el tiempo, se han incluido otros 9: C#, Python 3, JavaScript, Typescript, Go, C++, Swift, PHP y Dart. Este desarrollo ha sido posible gracias a que es de **código abierto** y tiene licencia BDS 3, que permite el uso comercial, distribución y modificación. Por ejemplo, Twitter/X la usa para las búsquedas en su plataforma, Oracle la usa en SQL Developer IDE y sus herramientas de migración y NetBeans IDE, para analizar C++.

VENTAJAS DE ANTLR

ANTLR v4 incorpora Adaptive LL* (**ALL***) como estrategia de **análisis** sintáctico **dinámico**, en lugar de hacerlo estáticamente. Esta ventaja, permite analizar la gramática mientras se procesan las secuencias de entrada reales e ir adaptándose, mientras que el análisis estático tiene que considerar todas las posibles secuencias y es más rígido y menos expresivo. Todo esto simplifica las reglas gramaticales.

Otra característica, es que automáticamente **reescribe las reglas recursivas a la izquierda** por otras que no lo sean, siempre que esta conversión sea directa y las reglas se llamen a sí mismas al principio.

Por otro lado, es capaz de desacoplar la gramática y el código de la aplicación gracias los **árboles sintácticos**, lo que permite reutilizar el mismo analizador en diferentes contextos (en otros lenguajes o recorriendo varias veces la entrada) y sin tener que crearlos manualmente.

Esta herramienta es capaz de aceptar casi cualquier gramática, sin causar conflictos, **resolviendo ambigüedades** e identificando y **corrigiendo errores** de sintaxis, lo que ayuda al programador a eliminar posibles problemas que no se hubiesen tenido en cuenta a la hora de diseñar la gramática.

COMPONENTES QUE INCLUYE

ANTRL permite definir el análisis léxico, un lexer, y el análisis sintáctico, un parser, a la misma vez, como se explicará detalladamente en la siguiente sección. A continuación, se muestra un diagrama que representa el flujo de información y cómo un reconocedor de lenguaje lo procesa:

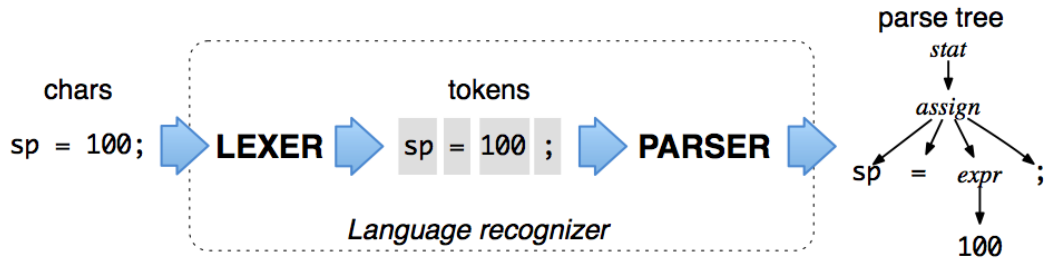


Fig. 1 Diagrama de funcionamiento

El programa de la entrada lee una secuencia de caracteres que el leer transforma en tokens y con ellos el parser genera un árbol sintáctico con las reglas.

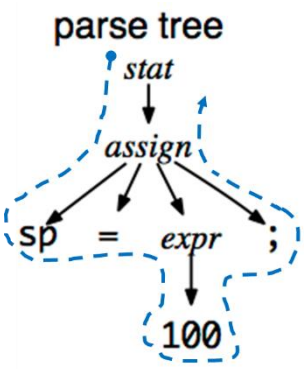


Fig. 2 Ejemplo de árbol sintáctico

También, incorpora una herramienta que permite ver gráficamente cómo se construye el árbol: las hojas son los tokens y la raíz la regla principal.

El recorrido se hace de forma recursiva y descendente. Cada nodo no terminal es una llamada a un método, que se corresponde con una regla, y en las hojas se llama al método predefinido *match()* del lexer para identificar tokens. Todas estas operaciones, ANTLR las gestiona automáticamente.

Aquí, stat, assign y expr son reglas y los tokens son: sp, =, 100, ;

La clase *ParseTreeWalker* se encarga de recorrer y realizar acciones específicas en cada nodo del árbol sintáctico. Implementa, por defecto, el patrón de oyente, aunque también se puede utilizar el patrón visitante:

- *ParseTreeListener*: genera una interfaz de escucha que responde a eventos desencadenados conforme se recorre el árbol, mediante callbacks: métodos *enter()* y *exit().* Las llamadas se hacen de forma implícita.
- ParseTreeVisitors: el programador puede controlar y personalizar la exploración. Hay que llamar explícitamente a los métodos para visitar un nodo: *visit().*

El programa ANTLR asocia a cada regla una estructura con las siguientes clases y objetos:

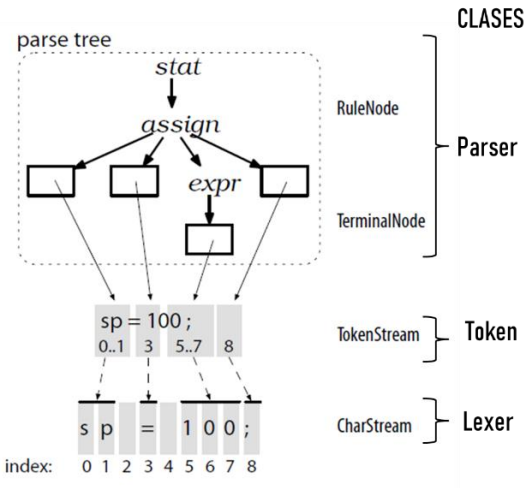


Fig. 3 Ejemplo con clases

Cada nodo no terminal, es de la clase *RuleNode* que define objetos de contexto porque registran todo lo relativo a la regla y los caracteres reconocidos. Cada nodo terminal (*TerminalNode*) contiene un puntero que apunta a un objeto de la clase *TokenStream* y este a unos caracteres de inicio y corte del token.

FICHEROS DEL PROGRAMA

La definición de la gramática se guarda en un fichero con extensión *.g4.*, y a partir de este se generan una serie de archivos. Por ejemplo, para una gramática *Name*:

- *NameParser.java*: contiene la definición del analizador sintáctico.
- *NameLexer.java*: contiene el analizador léxico de esta gramática.
- *Name.tokens* y *NameInitLexer.tokens*: archivo que indexa cada token con un número.
- *NameListener.java* y *NameBaseListener.java:* contienen las callbacks asociadas al árbol sintáctico.

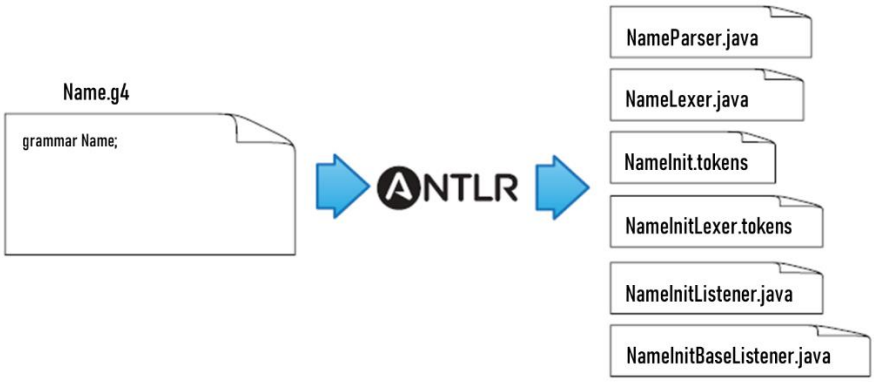


Fig. 4 Diagrama de ficheros generados

COMPARACIÓN CON JFLEX Y CUP

Las ventajas de ANTLR frente a Jflex son, el soporte de varios lenguajes y la creación de gramáticas complejas, aunque esto hace que la ejecución sea más lenta. Además, Jflex es más simple pero es una herramienta exclusivamente de análisis léxico.

En el caso de CUP, este usa LALR, que es más eficiente en términos de memoria y tiempo de ejecución, pero no es tan sencillo y expresivo como ALL* aunque no pueda manejar recursiones a izquierdas. CUP está diseñado exclusivamente para el lenguaje Java, mientras que ANTLR también se puede usar en otros lenguajes como ya se ha descrito anteriormente.

DESCRIPCIÓN DEL LENGUAJE

ESTRUCTURA GENERAL DE LA GRAMÁTICA

ANTLR puede definir el léxico y la sintaxis juntos o por separado. Si aparecen juntas, las reglas se reordenan de modo que las léxicas siempre aparezcan después de las sintácticas. Todas las reglas que se definen tienen que acabar en punto y coma. En general, un analizador de ANTLR tiene la siguiente estructura:

```
grammar Name;  // nombre de la gramática
options {...}
import ... ; // importar las reglas léxicas y sintácticas si las hemos
          definido en otro fichero
tokens {...}
@actionName {...} // código ejecutable

rules   // reglas de producción
...
```

Hay que tener en cuenta que a la hora de importar otras gramáticas:

- Las gramáticas que se hayan definido como exclusivamente léxicas, solo podrán importar reglas léxicas. Igual para las gramáticas sintácticas.
- Las gramáticas mixtas pueden importar reglas de ambos tipos.
- Las reglas del fichero principal tendrán mayor prioridad frente a las importadas.

LÉXICO

La parte léxica de un reconocedor de lenguaje se encarga de convertir los caracteres o palabras que aparecen en tokens y que después serán procesados por la parte sintáctica para crear la estructura lógica. Por lo tanto, estas reglas deben ser las primeras en ejecutarse.

DEFINICIÓN DE REGLAS LÉXICAS

Como ya se ha comentado antes, las reglas léxicas se pueden definir en el mismo fichero o en otro aparte de las reglas gramáticas. Si se separan, el fichero léxico debe comenzar con la cabecera:

```
lexer grammar GrammarName;
```

El formato de las reglas es el siguiente:

```
TOKEN_NAME : pattern;
```

- *TOKEN_NAME* es el identificador del token y debe comenzar siempre por mayúscula. Después puede estar seguido de cualquier letra, dígito y underscore.
- *pattern* es la expresión regular que define cómo se reconoce ese token.

En caso de que hubieran conflictos entre reglas y un mismo patrón pudiera reconocerse con varias expresiones regulares, ANTLR escoge la regla que aparece primero para resolver ambigüedades. Además este lenguaje permite que las reglas léxicas sean recursivas y que se puedan reconocer estructuras anidadas.

DEFINICIÓN DE ELEMENTOS PARA EXPRESIONES REGULARES:

- '.....' = reconoce el literal.
- ? = opcional.
- (..)* = 0 o más ocurrencias.
- (..)+ = 1 o más ocurrencias.
- | = unión.
- [..-..] o 'char'..'char' = rango
- . = wildcard, cualquier carácter.
- ~x = cualquier cosa menos x, que puede ser un carácter o un rango.
- \ = carácter de escape.

Ejemplos:

```
ID : [a-zA-Z]+;  //1 o más letras en mayúscula o minúscula
```

Si queremos que una regla no tenga en cuenta las mayúsculas ni las minúsculas, se puede especificar en la expresión regular o se puede añadir la siguiente opción:

```
ID options {caseInsensitive=true;} : [a-z]+;
```

Si queremos crear un token auxiliar para usarlo en una expresión regular pero que no sea visible para el analizador sintáctico, podemos definirlo con la palabra "*fragment*".

```
SEQUENCE: PATTERN ',' PATTERN;
SEQUENCE_DOT: PATTERN '.' PATTERN;

fragment PATTERN: abc|ABC;
```

Así, en este ejemplo PATTERN equivale a la secuencia abc en mayúscula o minúscula.

COMANDOS LÉXICOS

Para asignar una acción específica cuando se identifica un token, se pueden usar comandos. Estos se definen con el operador (->) al final de la regla. Por ejemplo: ***"skip"*** ignora ese token y no lo pasar al analizador sintáctico.

```
COMMENT : '/*' .*? '*/' -> skip ; // Match "/*" …… "*/"
WHITESPACE: ‘ ‘ -> skip ;
```

SINTÁCTICO

La parte sintáctica de un reconocedor de lenguaje se encarga de procesar los tokens reconocidos y analizar cómo estos se relacionan entre sí.

DEFINICIÓN DE REGLAS

Si se crea un archivo específico para las reglas sintácticas debe comenzar con la cabecera:

```
parser gramar GrammarName;
```

El formato más general de las reglas es el siguiente:

```
rulename : structure;
```

Las reglas se escriben en minúscula y pueden tener atributos, valores de retorno o variables locales, igual que una función en un lenguaje de programación.

También se puede usar el símbolo **'|'** para generar alternativas dentro de la misma regla. Un caso especial es si la alternativa está vacía, es decir, no especificamos nada, esto equivale a una producción vacía y se convierte en opcional.

```
argument: ID | ;
```

En este ejemplo, la regla message puede estar formada por cualquiera de las reglas o tokens especificados.

```
message: (link | WORD | WHITESPACE)+ ;
```

SUBREGLAS

Una regla puede estar formada por otras reglas y elementos que definen su comportamiento:

- (x | y | z): Reconoce al menos una de las alternativas.
- (x | y | z)? : Indica que esta regla es opcional y reconoce una o ninguna alternativa.
- (x | y | z)* : Reconoce la regla cero o más veces.
- (x | y | z)+: Reconoce la regla una o más veces.

Extensión de la definición de las reglas

```
rulename [args] returns [return values] locals [localvars]: structure;
```

Esta forma es más general y completa, ya que permite agregar más información a las reglas y realizar varias acciones. Es útil cuando se necesita realizar operaciones a la vez que reconocemos una regla en concreto. Aunque esto se detallará en la siguiente sección.

- args: argumentos que la regla puede recibir.
- returns: permite que la regla devuelva un valor.
- locals: define variables locales que se usan en la acción de la regla.

Ejemplo para el lenguaje Java:

```
// Devuelve el argumento más el valor del token INT leido
add[int x] returns [int result] : '+=' INT {$result = $x + $INT.int;} ;
```

ACCIONES

Como ya hemos visto en el ejemplo anterior, las acciones se escriben entre llaves y son bloques de código escritos en el lenguaje objetivo, por lo que deben seguir sus normas. ANTLR se encarga automáticamente de copiar el contenido dentro del código que autogenera.

Las acciones se pueden usar en cualquier parte de la gramática como:

- Acciones específicas (@header y @members). La acción header introduce el código antes de la definición de la clase de reconocimiento, por ejemplo para importar librerías o definir variables globales. Y member inserta el código dentro, como métodos o definición de variables.
- Asociadas a una regla léxica o sintáctica.

Un ejemplo de las primeras acciones es:

```
@header {
    import java.util.List;
}

@members {  //define variables
    private List<String> variableList = new ArrayList<>();
}
```

Para usar algunas de estas acciones, se incluyen atributos asociados a los tokens (*label*) a los que se accede con *$label.attribute:*

- text: devuelve el texto que se ha reconocido.
- line: número de línea en la que aparece el token.
- pos: la posición del primer carácter del token dentro de la línea.
- index: posición que ocupa dentro del stream de caracteres.
- int: devuelve el valor numérico del texto.

Ejemplos:

```
// devuelve el ID leido
ID : [a-zA-Z]+ { System.out.println("ID found : "+ $ID.text); };

// devuelve la linea en la que se encuentra el main
main: m='main' args bloque {System.out.println("line="+$m.line);} ;
```

Atributos asociados a las reglas:

- text: texto reconocido por una regla o texto reconocido hasta la llamada a *text*.
- start: primer token que podría ser reconocido por la regla.
- stop: último token que ha sido reconocido.
- ctx: regla asociada al contexto de la regla invocada.

```
exit: 'return' expr ';' {System.out.println($ctx.start +$expr.text);} ;
```

GUIA DE INSTALACION

A continuación se explicará como ejecutar ANTLR4 en Windows y en el anexo se incluye también como instalarlo en Linux.

Para poder integrar la herramienta con VS Code, lo primero es instalar la última versión de ANTLR, esta es la versión 4. Así, accedemos a la página oficial de ANTLR (*antlr.org*) y a la pestaña de descargas.

Encontramos distintos ficheros para los cada uno de los lenguajes que soporta y, en este caso, descargaremos los binarios de Java completos.

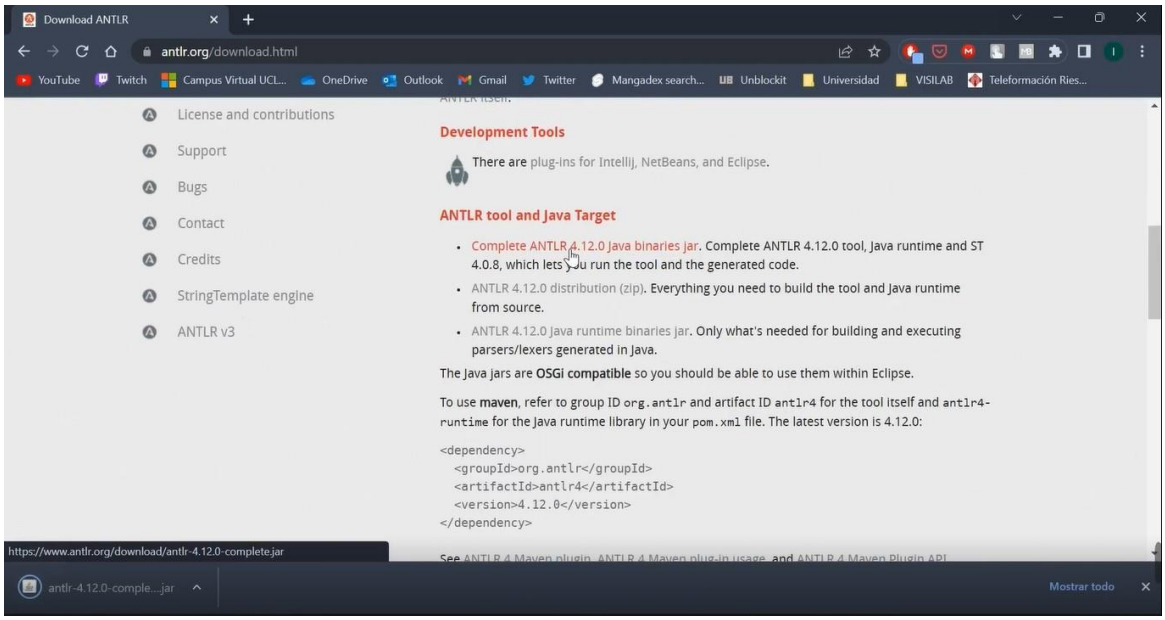


Fig. 5 Descarga de ANTLR

Una vez descargado, lo guardaremos en una carpeta que recordemos. Por ejemplo, podemos crear una carpeta *javalib* con las diferentes herramientas que tengamos de Java y donde guardaremos el archivo jar.

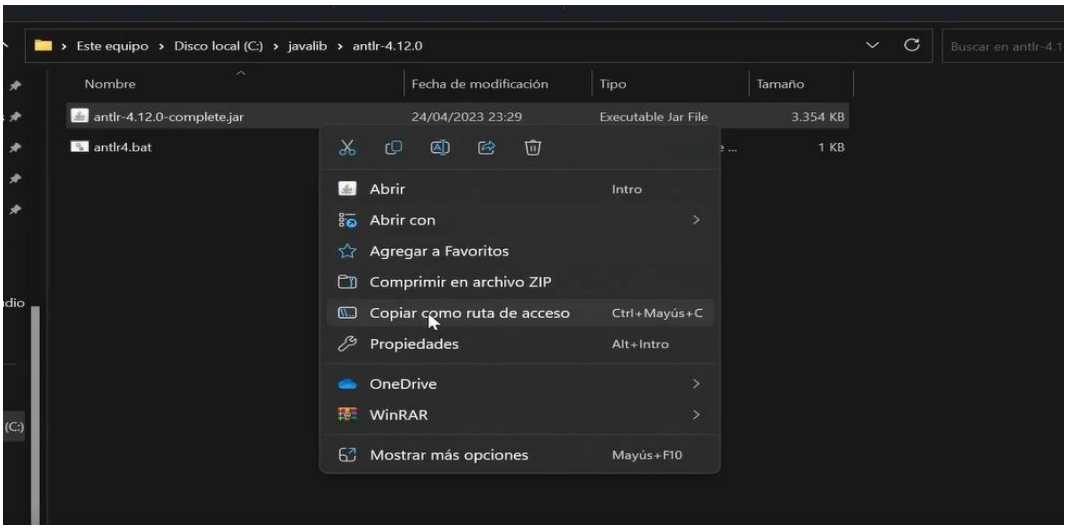


Fig. 6 Guardar el fichero

Después, hay que añadir el jar a nuestras variables de entorno *classpath* y para ello copiamos la ruta de acceso del jar y abrimos el menú de variables de entorno del sistema.

Si no tenemos creada la variable de entorno classpath podemos añadirla pulsando en el botón *"Nueva…"* y asignándole el nombre classpath.

Ahora, al editarla, añadimos una nueva entrada con la ruta de acceso del fichero jar de antlr.

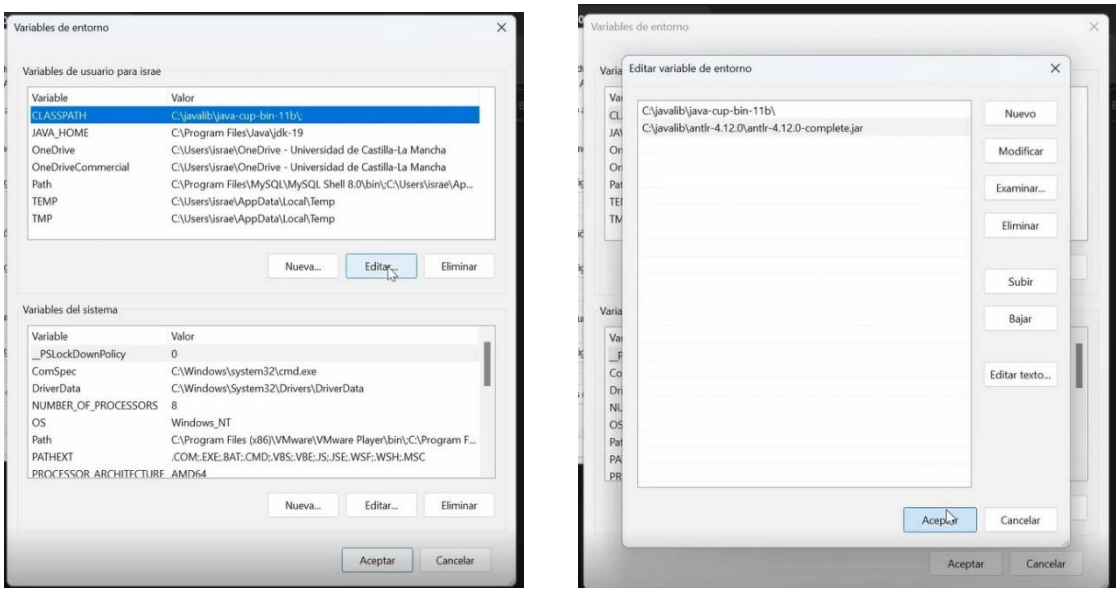


Fig. 7 Variable classpath

Opcionalmente, podemos crear un fichero, con extensión bat (*antlr4.bat*), para que en lugar de escribir el comando siempre que vayamos a ejecutar ANTLR, podamos simplemente poner el nombre del archivo. El contenido se muestra en la imagen de la derecha.

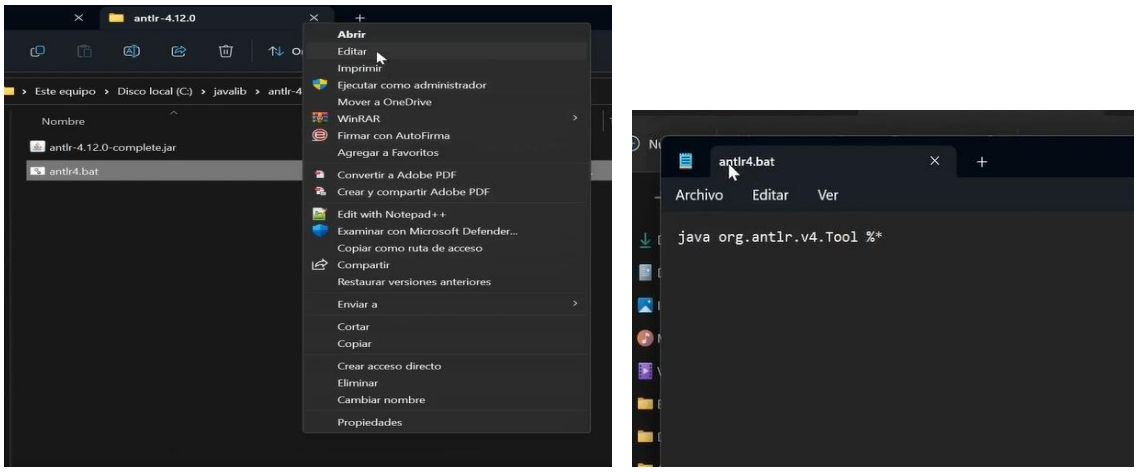


Fig. 8 Creación del fichero bat

Para poder ejecutar la herramienta desde cualquier directorio, deberemos de añadir la ruta de acceso a la variable de entorno path (las acciones son similares a las que hemos hecho en el paso anterior).

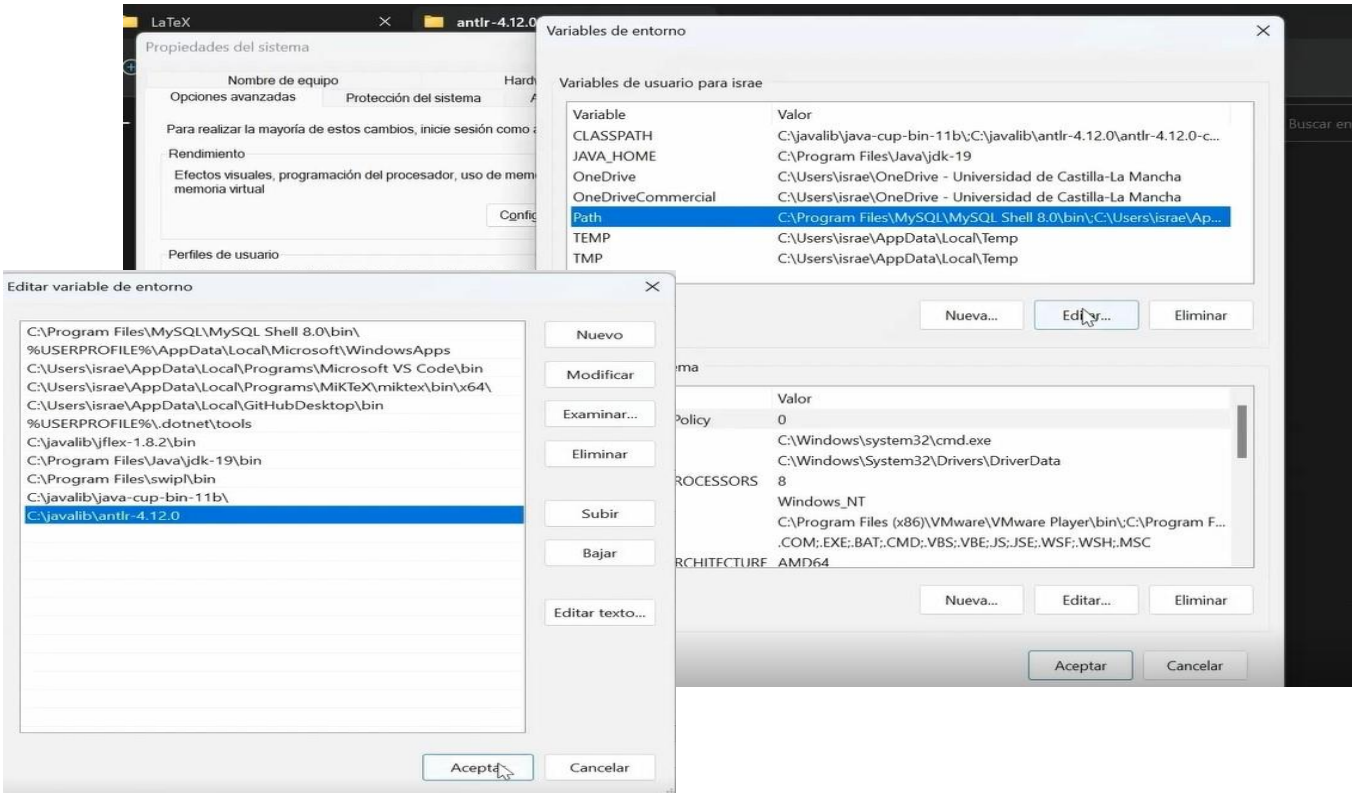


Fig. 9 Variable path

Una vez finalizados todos estos pasos, hay que comprobar que la herramienta ha sido instalada. Para ello, abrimos una terminal y desde cualquier directorio ejecutamos el nombre del fichero bat (*antlr4*). Si el fichero se ha instalado correctamente, debería salir una serie de opciones para la herramienta:

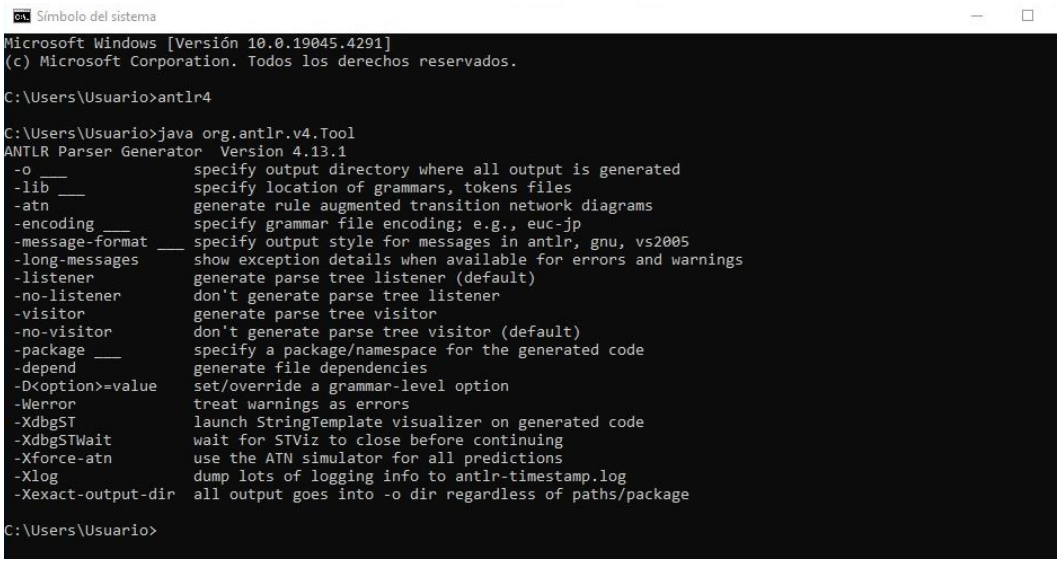


Fig. 10 Comprobación de la instalación

INSTALACION EN VISUAL STUDIO

- **Instalación ANTLR4**

A continuación, mostraremos los pasos necesarios para instalar ANTLR4 como complemento en Visual Studio Code y cómo ejecutar un archivo para crear los analizadores.

- **Paso 1**: Acceder al Administrador de Extensiones.
  - Abrir el menú "Extensions" (Extensiones) en la barra de menú superior.
  - Seleccionar "Manage Extensions" (Administrar extensiones).

- **Paso 2**: Buscar ANTLR4 en Marketplace.
  - Utilizar el cuadro de búsqueda para buscar "ANTLR4".
  - Seleccionar el plugin "ANTLR Language Support" de los resultados.

- **Paso 3**: Instalar el Plugin ANTLR4:
  - Descargar el plugin haciendo clic en "Download" (Descargar).
  - Luego, instalarlo haciendo clic en "Install" (Instalar).
  - Seguir las instrucciones en pantalla y, si es necesario, reiniciar Visual Studio.

- **Ejecución de un archivo desde VSCode**

- **Paso 4**: Creación de una Gramática ANTLR4:
    - Después de instalar el plugin, crea un nuevo archivo en tu proyecto con la extensión ".g4" para definir tu gramática ANTLR4.
    - Visual Studio detectará automáticamente la extensión ".g4" y generará los analizadores correspondientes.

- **Paso 5:** Escribir la gramática
    - Abrir el archivo ".g4" recién creado y definir la gramática. En este caso, se ha elegido como ejemplo una gramática que

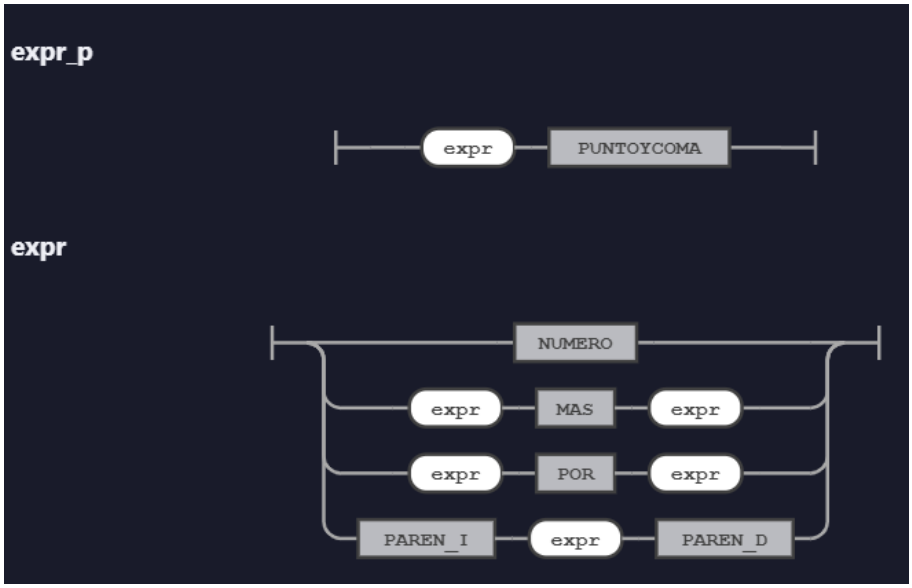
```
grammar ExpresionesAritmeticas;
// Parser rules

lista
     : expr_p lista
     | expr_p
     | expr_p
     ;

expr_p
      : expr PUNTOYCOMA
      ;

expr
    : NUMERO
    | expr MAS expr
    | expr POR expr
    | PAREN_I expr PAREN_D
    ;


// Lexer rules
PUNTOYCOMA : ';';
MAS        : '+';
POR        : '*';
PAREN_I    : '(';
PAREN_D    : ')';
NUMERO     :[0-9]+ ;
ESPACIO    :[\t\r\n\f]+ -> skip ;
```

- **Paso 6:** Generar el Analizador Sintáctico:
    - Guardar el archivo después de escribir la gramática.
    - Haz clic derecho en el archivo ".g4".

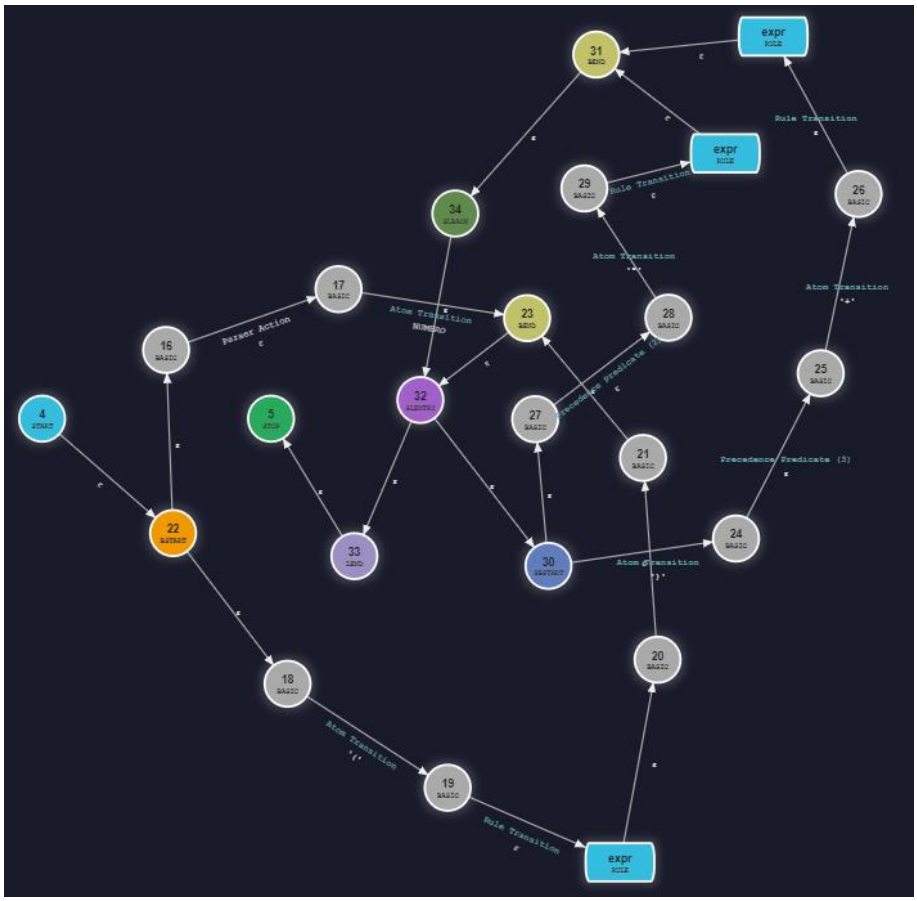
- Seleccionar "Run ANTLR" (Ejecutar ANTLR) en el menú.
- Esto generará los archivos Java correspondientes al analizador sintáctico.

La extensión permite varias acciones como generar una entrada válida para la regla, por ejemplo, para la regla expr sería: (0 + (⨯)) * (8246 + ⨯)

También muestra el árbol sintáctico de las reglas, por ejemplo:



Este es el grafo que se genera con la representación interna de la regla *expr.*



ANEXO

Aunque esta breve guía permite llevar a cabo la instalación completa, es importante mencionar que se recomienda seguir los pasos iniciales proporcionados por la compañía desarrolladora de ANTLR para su instalación en Windows y varias distribuciones de Linux. Nos enfocaremos en la instalación de ANTLR en la distribución de Linux más popular, Ubuntu.

INSTALACION EN LINUX

Para instalar ANTLR en cualquier sistema operativo, es necesario tener la capacidad de compilar otro lenguaje, como C++ o Java, que permita compilar el código generado por el analizador sintáctico creado en ANTLR.

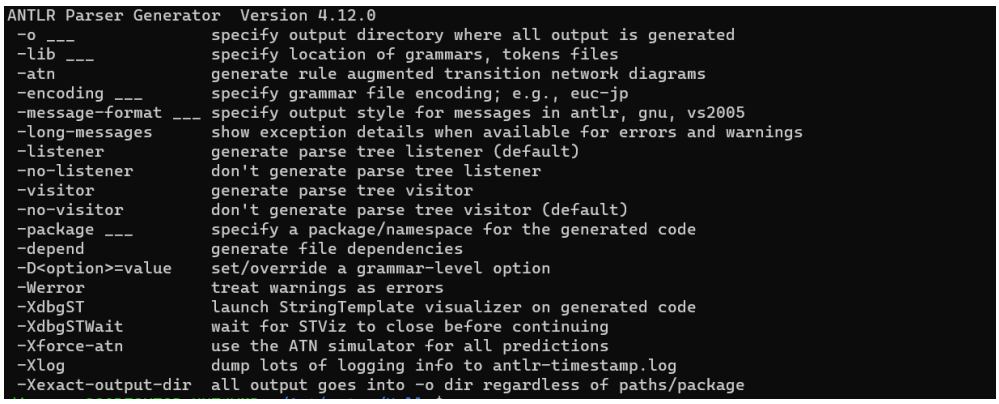
Para llevar a cabo la instalación de ANTLR en Ubuntu, simplemente ejecutaremos el siguiente comando.

```
$ pip install antlr4-tools
```

Una vez que se haya ejecutado sin errores, el sistema continuará con la instalación estándar de ANTLR. Después de completar la instalación, es aconsejable realizar un "apt-update" para prevenir posibles problemas después de la descarga.

Posteriormente, para verificar la instalación, se debe ejecutar el comando "antlr4". Si todo funciona como se espera y la instalación se ha realizado correctamente, nos permitirá acceder a las funciones de ANTLR sin inconvenientes.

El resultado debería ser algo como esto:



Una vez hecho esto, solo necesitaremos instalar la biblioteca correspondiente al lenguaje que deseemos utilizar para compilar el analizador sintáctico. En nuestro caso, hemos seleccionado Java. No obstante, recordamos que se puede consultar la guía inicial mencionada al inicio de la sección 3 para instalar la biblioteca de otros lenguajes.

Para instalar el archivo JAR correspondiente en Java, primero debemos ubicarnos en la carpeta raíz de las bibliotecas y proceder con la descarga desde allí.

```
$ cd /usr/local/lib
$ curl -O https://www.antlr.org/download/antlr-4.12.0-complete.jar
```

También es posible descargar el archivo JAR manualmente utilizando cualquier navegador y luego pegarlo en la carpeta /usr/local/lib.

Después de descargar el JAR, necesitaremos agregarlo a nuestro CLASSPATH con el siguiente comando.

```
$ export CLASSPATH=".:/usr/local/lib/antlr-4.12.0-complete.jar:$CLASSPATH"
```

Para verificar que el archivo JAR se ha instalado correctamente y que se reconoce dentro del CLASSPATH, podemos ejecutar el siguiente comando:

```
$ java org.antlr.v4.Tool
```

Si el jar ha sido instalado exitosamente nos debería salir algo asi:

```
ANTLR Parser Generator  Version 4.12.0
 -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
 -Xexact-output-dir  all output goes into -o dir regardless of paths/package
```

Una vez completados estos pasos, estaremos listos para ejecutar los ejemplos mencionados en la sección 2. Supongamos que deseamos crear el analizador sintáctico para la gramática Hello.gr4 como se especifica en la sección 2.2. En este caso, necesitaremos ubicarnos en la ubicación del archivo y ejecutar el siguiente comando:

```
$ antlr4-parse hello.g4 r -gui
hello carretera
^D → simboliza el EOF de linux
```

El programa por tanto nos devuelve lo siguiente:

Dado que la opción "-gui" solicita la visualización del árbol de la gramática, podríamos explorar las diferentes opciones que el comando ofrece, como "-tokens", que muestra todos los tokens utilizados, o "-trace", que proporciona las trazas seguidas hasta la aceptación o no de la cadena introducida como parte del lenguaje.

Sin embargo, si nuestro objetivo es generar el archivo Java responsable de reconocer cadenas en ese lenguaje, entonces deberíamos ejecutar el siguiente comando:

```
$ antlr4 Hello.g4
```

El comando nos genera los siguientes archivos java:

```
$ ls Hello*.java

HelloBaseListener.java
HelloLexer.java
HelloListener.java
HelloParser.java
```

Para obtener el archivo ejecutable en Java, necesitamos compilar utilizando el siguiente comando:

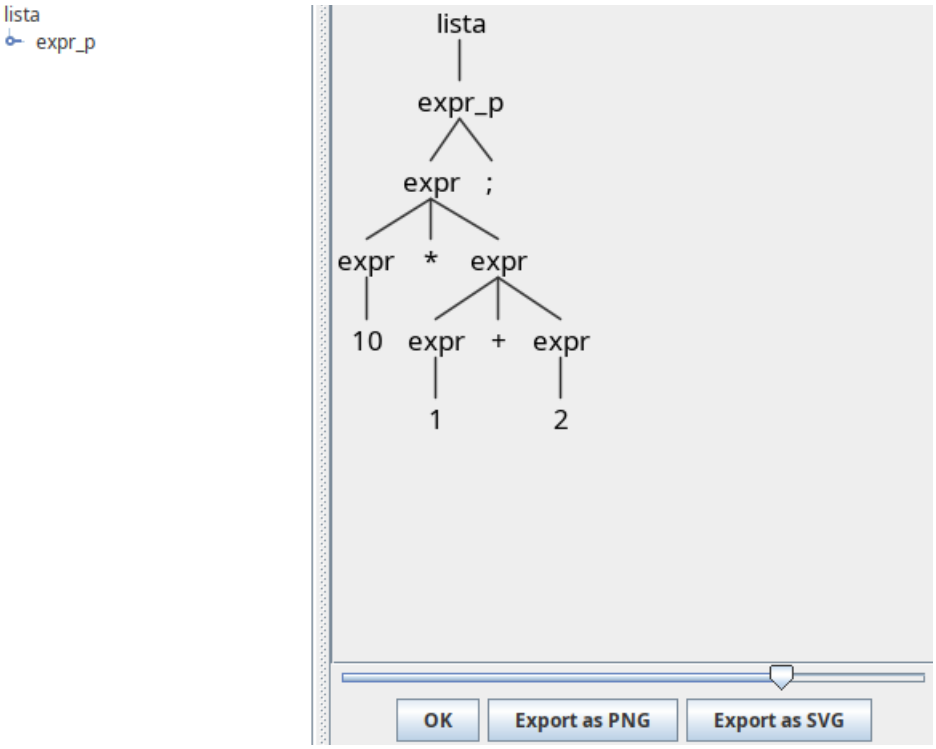
```
$javac Hello*.java
```

Con este paso completado, ahora disponemos de los archivos ejecutables en Java necesarios. A partir de este punto, podemos utilizar este archivo de manera similar a como lo hacíamos con el comando "antlr4" de la siguiente manera:

```
$ grun Hello r -tree
```

java org.antlr.v4.gui.TestRig ExpresionesAritmeticas lista -gui

Al ejecutarlo como hicimos anteriormente con "antlr-parser", debería devolvernos la misma representación del árbol que obtuvimos previamente.



Exactamente, al ejecutar el comando anterior para verificar la instalación del JAR, deberíamos ver las opciones equivalentes a ANTLR listadas, lo que indicaría que la instalación ha sido exitosa. Una vez completado este paso y confirmado que todo se

ejecuta correctamente, podemos considerar que la instalación en Ubuntu Linux ha concluido con éxito.