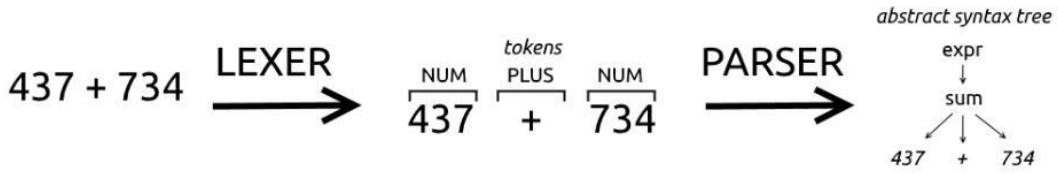
# INTRODUCCIÓN

## ¿Qué es ANTLR?

Es una herramienta que ayuda a crear analizadores. Un analizador toma una pieza de texto y lo transforma en una estructura organizada.

Veamos como analizar una expresión




| | |
|---|---|
| | **Reglas del analizador** |
| | |
| | operation: NUMBER ‘+’ NUMBER |
| | |
| | **Reglas léxicas** |
| | |
| | NUMBER: [0-9]+; |
| | |
| | WHITESPACE: ‘ ‘ -> skip; |
| | |

Esto no es una gramática completa, pero ya podemos ver que las reglas de léxicas están en mayúsculas, mientras que las reglas del analizador son todas minúsculas. Técnicamente, la regla sobre el caso se aplica solo al primer carácter de sus nombres, pero por lo general son en mayúsculas o en minúsculas para mayor claridad.

Las reglas normalmente se escriben en este orden: primero las reglas del analizador y luego las léxicas, aunque lógicamente se aplican en el orden opuesto. También es importante recordar que las reglas léxicas se analizan en el orden en que aparecen, y pueden ser ambiguas.

El ejemplo típico es el identificador: en muchos lenguajes de programación puede ser cualquier cadena de letras, pero ciertas combinaciones, como "**class**" o "**function**" están prohibidas porque indican una clase o una función. Así que el orden de las reglas resuelve la ambigüedad utilizando la primera coincidencia y por eso los tokens que identifican palabras clave, como clase o función, se definen primero, mientras que las del identificador se pone al final.

La sintaxis básica de una regla es fácil: hay un nombre, dos puntos, la definición de la regla y un punto y coma que termina.

La definición de NÚMERO contiene un rango típico de dígitos y un símbolo "+" para indicar que una o más instancias están permitidos. Estas son todas las indicaciones muy típicas con las que asumo que estás familiarizado.

La parte más interesante es al final, la regla lexer que define el token WHITESPACE. Es interesante porque muestra cómo indicar a ANTLR que ignore algo. Considerar como ignorar los espacios en blanco simplifica las reglas del analizador: si no pudiéramos decir que ignore (WHITESPACE), lo tendríamos que hacer entre cada regla una subregla en el analizador, para que el usuario ponga espacios donde quiera:

| | |
|---|---|
| **1** | operation: WHITESPACE* NUMBER WHITESPACE* '+' WHITESPACE* NUMBER; |

Y lo mismo aplicaría para los comentarios: pueden aparecer en todas partes y no queremos. Los manejamos específicamente en cada pieza de nuestra gramática, así que simplemente los ignoramos (al menos en el análisis).

Ahora que hemos visto la sintaxis básica de una regla, podemos echar un vistazo a los dos enfoques diferentes para definir una gramática:

1) Top-down (De arriba hacia abajo).
2) Button-up (de abajo hacia arriba).

## Nuestro primer ejemplo

La mejor manera de aprender acerca de ANTLR es caminar a través de un simple pero ejemplo útil. En este taller, construiremos una expresión aritmética.

Un evaluador de expresiones que soporta unos pocos operadores y asignaciones de variables. De hecho, implementaremos el evaluador de dos maneras diferentes.

1. Primero vamos a construir una gramática de análisis para reconocer el lenguaje de expresión y luego agregar acciones para evaluar e imprimir el resultado.
2. Segundo, modificaremos la gramática del analizador para construir una estructura de datos de árbol de forma intermedia.

En lugar de calcular inmediatamente el resultado, construiremos una gramática de árbol para recorrer esos árboles, agregando acciones para evaluar e imprimir el resultado. Cuando haya terminado con este ejemplo, tendrá una vista general de cómo construir traductores con ANTLR.

Aprenderás sobre analizador de gramáticas, tokens, acciones, ASTs y gramáticas de árbol por ejemplo, lo que facilitará la comprensión de los ejercicios siguientes.

Para que sea sencillo, restringiremos el lenguaje de expresión para que sea compatible con las siguientes construcciones:

- ✓ Operadores +, - y * con el orden habitual del operador de evaluación, permitiendo expresiones como esta:
  3+4*5-1
- ✓ Expresiones entre paréntesis para alterar el orden de evaluación del operador, permitiendo expresiones como esta:
  (3+4)*5

- ✓ Asignaciones y referencias variables, permitiendo expresiones tales como:
  a=3
  b=4
  2+a*b

**Esto es lo que queremos que haga el traductor:**
Cuando vea 3 + 4, debería emitir 7. Cuando ve **edad** = 21, debe asignar **edad** a un valor de 21. Si el traductor vuelve a ver **edad**, debe fingir que escribimos 21 en lugar de **edad**.
¿Cómo empezamos a resolver este problema? Bueno hay dos tareas generales:

1. Construye una gramática que describa la estructura sintáctica general de expresiones y asignaciones. El resultado de esa entrada, es un reconocedor que responde sí o no en cuanto a si la entrada fue válida, ya sea una *expresión* o *asignación*.

2. Insertar código entre los elementos gramaticales en posiciones apropiadas para evaluar piezas de la expresión. Por ejemplo, una entrada dada 3, el traductor debe ejecutar una acción que convierte el carácter tres a su valor entero. Para la entrada 3 + 4, el traductor debe ejecutar una acción que agrega los resultados de dos ejecuciones de acción anteriores, es decir, las acciones que convirtieron los caracteres 3 y 4 a sus equivalentes enteros.

Después de completar esas dos grandes tareas, tendremos un traductor que *traduce expresiones* al valor aritmético usual. En las siguientes secciones, vamos a cubrir a través de esas tareas los detalles. Seguiremos este proceso:

1. Construye la expresión gramática.
2. Examine los archivos generados por ANTLR.
3. Construye un banco de pruebas y prueba el reconocedor.
4. Añadir acciones a la gramática para evaluar expresiones y emitir resultados
5. Aumentar el banco de pruebas y probar el traductor.

Ahora que hemos definido el idioma, construyamos una gramática ANTLR que reconoce oraciones en ese lenguaje y calcula resultados.

## Reconociendo la Sintaxis del Lenguaje

Necesitamos construir una gramática que describa completamente la sintaxis de nuestro lenguaje de expresión, incluyendo la forma de identificadores y enteros.

Desde la gramática, ANTLR generará un programa que reconoce expresiones válidas, emitiendo automáticamente errores para expresiones inválidas.

Comience por pensar en la estructura general de la entrada, y luego divide esa estructura en subestructuras, y así sucesivamente, hasta que alcance una estructura que no puedas descomponer más. En este caso, la entrada general consiste en una serie de expresiones y asignaciones, que vamos a desglosar en más detalles a medida que avanzamos.

Ok, comencemos tu primera gramática ANTLR. En ANTLR la gramática más común, es una gramática combinada que especifica tanto el analizador como reglas lexer. Estas reglas especifican

la estructura gramatical de una expresión así como su estructura léxica (las denominadas tokens). Por ejemplo, un asignación es un identificador, seguido de un signo igual, seguido de una expresión, y termina con una nueva línea; un identificador es una secuencia de letras. Defina una gramática combinándola usando la palabra clave **`grammar`**:

```
grammar Expr;
«rules»
```

Coloque esta gramática en el archivo ***Expr.g4*** porque el nombre del archivo debe coincidir con el Nombre de la gramática.

Un programa en este lenguaje parece una serie de declaraciones seguidas por el carácter de nueva línea (nueva línea en sí misma es una declaración vacía e ignorada). Más formalmente, estas reglas en inglés se parecen a las siguientes cuando se escribe en notación ANTLR donde: *comienza una definición de regla y* **|** *una barra vertical para reglas alternativas*:

```
prog: stat+ ;

stat: expr NEWLINE
| ID '=' expr NEWLINE
| NEWLINE
;
```

Una regla gramatical es una lista con nombre de una o más alternativas, como ***prog*** y ***stat***. Lea ***prog*** como sigue: la declaración ***prog*** es una lista de reglas de ***stat***. Para leer la regla ***stat*** se hace de la siguiente manera: una regla ***stat*** es una de las tres alternativas:

- ✓ Una regla ***expr*** seguida de una nueva línea (token ***NEWLINE***)
- ✓ El ***ID*** de secuencia (un identificador), '**=**', ***expr***, ***NEWLINE***
- ✓ Un token de ***NEWLINE***

Ahora tenemos que definir cómo se ve una expresión, la regla ***expr***. Los patrones prescriben una serie de reglas, una para cada operador un nivel precedencia y uno para el nivel más bajo que describe expresiones atómicas tales como enteros. Comience con una regla general llamada ***expr*** que representa una expresión completa. La regla ***expr*** hará coincidir los operadores con la precedencia más débil, + y -, y se referirán a una regla que coincida con sub-expresiones para operadores con la siguiente prioridad más alta. En este caso, el operador de multiplicación * es el siguiente. Podemos llamar a la regla ***multExpr***. Reglas ***expr***, ***multexpr***, y el ***atom*** se ve así:

```
expr: multExpr (('+' |'-' ) multExpr)*
;
multExpr
: atom ('*' atom)*
;
atom: INT
| ID
| '(' expr ')'
;
```

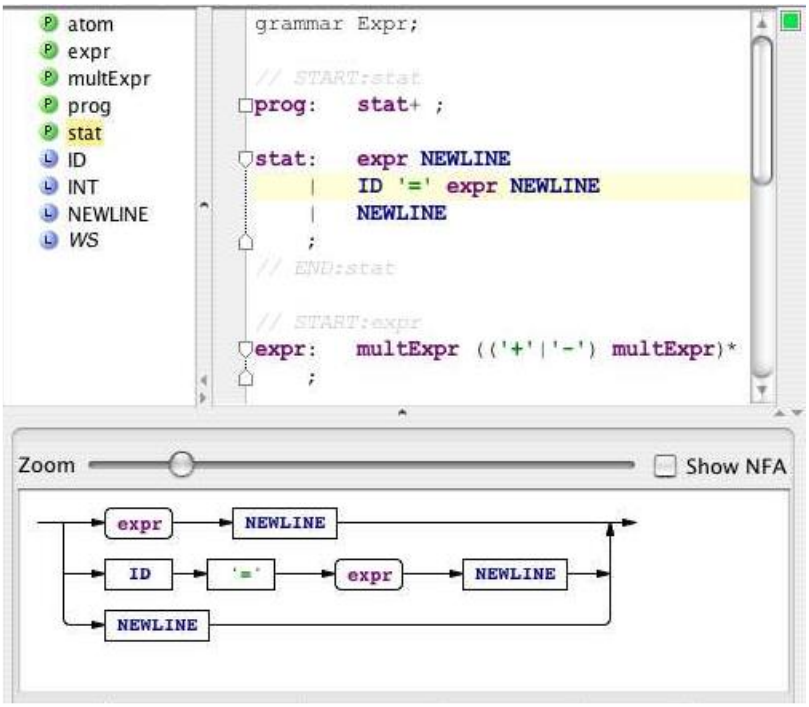
Volviendo al nivel léxico, definamos los símbolos de vocabulario (tokens): Identificadores, enteros y el carácter de nueva línea. Cualquier otro espacio en blanco es ignorado Todas las reglas léxicas

comienzan con una letra mayúscula en ANTLR y normalmente se refieren a caracteres y literales de cadenas, no tokens. Como analizador léxico (parser), aquí están todas las reglas léxicas que necesitaremos:

```
ID : ('a'..'z' |'A'..'Z' )+ ;
INT : '0'..'9' + ;
NEWLINE:'\r' ? '\n' ;
WS : (' ' |'\t' |'\n' |'\r' )+ {skip();} ;
```

La regla ***WS*** (espacio en blanco) es la única con una acción (***skip ()***;) que le indica a ANTLR ignorar y buscar otro token.

La forma más fácil de trabajar con las gramáticas de ANTLR es usar ANTLRWorks, que proporciona un entorno de desarrollo sofisticado. La Figura en la página siguiente, muestra qué gramática tiene ***Expr*** dentro de ANTLRWorks. Observe que la vista del diagrama de sintáctico de una regla hace que sea fácil de entender exactamente lo que coincide con la regla.



```
grammar Expr;

// START:stat
prog:   stat+ ;

stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;
// END:stat

// START:expr
expr:   multExpr (('+'|'-') multExpr)*
    ;

multExpr
    :   atom ('*' atom)*
    ;

atom:   INT
    |   ID
    |   '(' expr ')'
    ;
// END:expr

// START:tokens
ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t')+ {skip();} ;
// END:tokens
```

Ha logrado un progreso significativo en este punto porque tiene un parser y lexer, todo sin escribir ningún código Java! Como puedes ver, escribiendo. Una gramática es mucho más fácil que escribir tu propio código. Se puede pensar en la notación de ANTLR como un lenguaje específico de dominio específicamente diseñado para que los reconocedores y traductores sean fáciles de construir.

Para pasar de un reconocedor a un traductor o intérprete, necesitamos añadir acciones a la gramática, pero ¿qué acciones y dónde?

Para ello, deberemos realizar las siguientes acciones:

**Ejemplo**

1. Defina una variable en memoria de tipo HashMap para almacenar los valores.
2. Sobre la expresión, imprima el resultado que tiene ***exp***.
3. Tras la asignación, evalúe la expresión del lado derecho y asigne el resultado a la Variable del lado izquierdo. Almacena los resultados en memoria.
4. En **INT**, devuelve su valor entero como resultado.
5. Sobre el ***ID***, devuelva el valor almacenado en la variable de memoria. Si la variable no ha sido definida, emite un mensaje de error.
6. Sobre la *expresión(**expr**)* entre paréntesis, devolver el resultado al mismo valor de la expresión.
7. Al multiplicar dos ***atom***, devuelve la multiplicación de los resultados de dos atom.
8. Al sumar dos expresiones ***multExpr***, devuelva como resultado la suma las dos subexpresiones.

9. Tras restar dos expresiones ***multExpr***, devuelva como resultado la resta de las dos subexpresiones.

Ahora solo tenemos que implementar estas acciones en Java y colocarlas en la gramática según la ubicación que implique. Comience por definir la memoria utilizada para asignar variables a sus valores.

(Paso 1):

```
@header {
import java.util.HashMap;
}
@members {
/** Map variable name to Integer object holding value */
HashMap memory = new HashMap();
}
```

No necesitamos una acción para la regla ***prog*** porque solo le dice al analizador que debe buscar uno o más instancias de ***stat***. Las lineas 2 y 3 de la lista detallada deben ir en ***stat*** para imprimir y almacenar los resultados de las expresiones:

```
prog: stat+ ;
stat: // evalua expr e imprime el resultado
// $expr.value  devuelve el valor del atributo desde la llamada a expr
expr NEWLINE {System.out.println($expr.value);}
// $ID.text se usa la propiedad text del ID
| ID '=' expr NEWLINE
{memory.put($ID.text, new Integer($expr.value));}
// no hacer nada si no hay sentencias
| NEWLINE
;
```

Para las reglas involucradas en la evaluación de ***expr***, es realmente conveniente que devuelva el valor de la subexpresión que coincida. Por lo tanto, cada regla coincidirá y evaluará una parte de la expresión, devolviendo el resultado como un valor de retorno del método. Mira el ***atom***, la subexpresión más simple primero:

```
atom returns [int value]
: // el valor de INT es un int calculado a partir del carácter de entrada
INT {$value = Integer.parseInt($INT.text);}
| ID // referencia a la variable
{
// mira lo que hay en la variable
Integer v = (Integer)memory.get($ID.text);
// Si no se encuentra devuelve error
if ( v!=null ) $value = v.intValue();
else System.err.println("variable indefinida "+$ID.text);
}
// El valor de expr es solo el valor de expr
| '(' expr ')' {$value = $expr.value;}
;
```

Según la cuarta acción de la lista detallada anterior, el resultado de ***atom*** INT es solo el valor entero del texto del token INT. El token INT con texto 91 resulta en el valor 91. La acción 5 nos dice que busquemos en la propiedad text del ID de la variable memory tipo hasmap para ver si tiene un valor.

Si es así, simplemente devuelva el valor entero almacenado en el hasMap, o imprima un error. La tercera alternativa de la regla ***atom*** recursivamente invoca la regla ***expr***. Ahí no hay nada que calcular, por lo que el resultado de esta evaluación **atom**, es solo el resultado de llamar a ***expr*** (); esto satisface la acción 6. Como puedes ver, ***$value*** es la variable de resultado como se define en la cláusula de ***returns***; ***$expr.value*** es el Resultado calculado por una llamada a ***expr***. Pasando a las subexpresiones, Aquí está la regla ***multExpr***:

```
multExpr returns [int value]
: e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
;
```

Las acciones en la regla ***expr***, la regla de expresión más externa, reflejan la acciones en ***multExpr***, excepto que estamos sumando y restando en lugar de multiplicando

```
expr returns [int value]
: e=multExpr {$value = $e.value;}
( '+' e=multExpr {$value += $e.value;}
| '-' e=multExpr {$value -= $e.value;}
)*
;
```

Para ejecutar acciones solo para una construcción particular, todo lo que tenemos que hacer es colocar acciones alternativas gramaticales que coincide con esa construcción.

**Ejemplos de entrada para la gramatica**

Entrada 1
3+4*5
EOF
23
Entrada 2
(3+4)*5
EOF
35
a=3
b=4
2+a*b
Entrada 3
(3
EOF
3

## Con esto concluye tu primer ejemplo completo de ANTLR

**Aquí todo el código de la gramatica de ejemplo y completa**

```
grammar Expr;

// START:members
@header {
import java.util.HashMap;
}

@members {
HashMap memory = new HashMap();
}
// END:members

// START:stat
prog:   stat+ ;

stat:   expr NEWLINE {System.out.println($expr.value);}


    |   ID '=' expr NEWLINE
        {memory.put($ID.text, new Integer($expr.value));}


    |   NEWLINE
    ;
// END:stat

// START:expr
expr returns [int value]
    :   e=multExpr {$value = $e.value;}
        (   '+' e=multExpr {$value += $e.value;}
        |   '-' e=multExpr {$value -= $e.value;}
        )*
    ;
// END:expr

multExpr returns [int value]
    :   e=atom {$value = $e.value;} ('*' e=atom {$value *= $e.value;})*
    ;
// END:multExpr

// START:atom
atom returns [int value]
    :
        INT {$value = Integer.parseInt($INT.text);}

    |   ID // variable reference
        {

        Integer v = (Integer)memory.get($ID.text);
        if ( v!=null ) $value = v.intValue();
        else System.err.println("undefined variable "+$ID.text);
        }
```

```
    |   '(' expr ')' {$value = $expr.value;}
    ;
// END:atom

// START:tokens
ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
NEWLINE:'\r'? '\n' ;
WS  :   (' '|'\t'|'\n'|'\r')+ {skip();} ;
```

**Ejemplo 2 Gramática de ANTLR que permite declarar variables, asignar valores a variables e imprimir la salida de una variable. Tiene más tokens pero algunos no se usan.**

```
grammar Simple;

@parser::header {
        import java.util.Map;
        import java.util.HashMap;
}

@parser::members {
        Map<String, Object> symbolTable = new HashMap<String,Object>();
}
program: PROGRAM ID BRACKET_OPEN
        sentence*
        BRACKET_CLOSE;

sentence: var_decl | var_assign | println;

var_decl: VAR ID SEMICOLON
        {symbolTable.put($ID.text,0);};
var_assign: ID ASSIGN expression SEMICOLON
        {symbolTable.put($ID.text, $expression.value);};
println: PRINTLN expression SEMICOLON
        {System.out.println($expression.value);}
        |
        PRINTLN COPEN ID COPEN COMA expression ((COMA|SEMICOLON) expression?)*
        {System.out.println( $ID.text + " "+ $expression.value);};
expression returns [Object value]:
        NUMBER {$value = Integer.parseInt($NUMBER.text);}
        |
        ID {$value = symbolTable.get($ID.text);};
PROGRAM: 'program';
VAR: 'var';
PRINTLN: 'println';

PLUS: '+';
MINUS: '-';
MULT: '*';
DIV: '/';

AND: '&&';
OR: '||';
NOT: '!';

GT: '>';
LT: '<';
GEQ: '>=';
LEQ: '<=';
EQ: '==';
NEQ: '!=';

COPEN: '"';

ASSIGN: '=';
BRACKET_OPEN: '{';
BRACKET_CLOSE: '}';

PAR_OPEN: '(';
PAR_CLOSE: ')';
COMA: ',';
SEMICOLON: ';';

ID: [a-zA-Z_][a-zA-Z0-9_]*;

NUMBER: [0-9]+;

WS: [ \t\r\n]+ -> skip;
```