

GUÍA DE ESTUDIO

CONSTRUCCIÓN DE UN ANALIZADOR DE EXPRESIONES ARITMÉTICAS CON ANTLR4 Y JAVASCRIPT

1. INTRODUCCIÓN

ANTLR4 (Another Tool for Language Recognition) es un potente generador de analizadores que permite definir gramáticas y, a partir de ellas, construir analizadores léxicos y sintácticos para procesar lenguajes formales.

Con ANTLR4, puedes definir una gramática, que describe la sintaxis de un lenguaje y, a partir de ella, generar automáticamente un Lexer (analizador léxico o scanner) y un Parser (analizador sintáctico). Luego, puedes utilizar estos analizadores para procesar cadenas de entrada válidas, en el lenguaje definido por tu gramática, construir el árbol de análisis sintáctico y aplicar transformaciones sobre él.

Si lo que quieres es construir un traductor completo, es necesario extender el parser generado agregando análisis semántico y una fase de generación de código. Esto se logra recorriendo el árbol mediante listeners o visitors, interpretando su estructura y generando una representación equivalente en otro lenguaje.

ANTLR4 puede generar analizadores en varios lenguajes de programación, incluyendo Java (su lenguaje nativo), C#, JavaScript, Python, C++, Go y Swift. En esta asignatura, trabajaremos con JavaScript, ya que es uno de los lenguajes que abordamos en el curso.

ANTLR4 es ampliamente utilizado en la industria y la academia, para la construcción de compiladores, intérpretes y herramientas de procesamiento de código, lo que lo convierte en una herramienta clave para cualquier persona interesada comprender cómo trabajan los lenguajes de programación. Por citar algunas:

- Entornos de desarrollo actuales como IntelliJ IDEA o NetBeans utilizan ANTLR4 para identificar la sintaxis de múltiples lenguajes y a partir de esto implementar características como el resaltado de texto, el autocompletado, la refactorización de código, etc.
- Presto: El motor de consultas distribuidas de Facebook también emplea ANTLR para el análisis de consultas SQL.
- SonarQube: Esta plataforma líder de análisis de código utiliza ANTLR para parsear y analizar múltiples lenguajes de programación, permitiendo la detección de problemas de calidad en el código.
- Groovy: El lenguaje de programación Groovy utiliza ANTLR para su gramática y análisis sintáctico.
- Ansible: Aunque no es su principal componente, ANTLR puede ser utilizado en herramientas de automatización para parsear configuraciones y scripts personalizados.

Esta guía te ayudará a comprender, siguiendo un caso de estudio, los aspectos básicos del trabajo que debes realizar con ANTLR y JavaScript para construir un analizador. Luego, a partir de este ejemplo, la lectura bibliográfica y la búsqueda de material complementario, podrás abordar la tarea requerida de construir tu propio traductor. En particular, es necesario que estudies y comprendas


como trabaja ANTLR4 para poder desarrollar la actividad. Para esto puedes consultar las fuentes listadas al final de esta guía.

También puedes utilizar ANTLR Lab (disponible en línea en <https://www.antlr.org/>) para probar tus gramáticas y experimentar con la herramienta.

Cuando hayas leído inicialmente esta documentación y experimentado con algunos ejemplos en ANTLR Lab, te sugerimos continuar con los puntos de la Guía.

2. REQUISITOS PREVIOS

- 2.1. Tener JRE (Java Runtime Environment) instalado en su computadora. En caso de no contar con Java preinstalado puede descargarlo e instalarlo desde <https://www.java.com/en/download/>. Se requiere Java 1.8 o superior. Este requisito responde a que ANTLR es un herramienta desarrollada en Java, y requiere del entorno de ejecución Java (JRE) para funcionar.
- 2.2. Descargar e Instalar Node.js (LTS) desde la página oficial <https://nodejs.org/es>. Se requiere Node 16 o superior. Esto es requerido porque el proyecto del analizador está construido con Node.
- 2.3. Descargar e instalar VS Code (IDE Visual Studio Code) desde la página oficial <https://code.visualstudio.com/>. VS Code será el entorno de desarrollo que utilizaremos para trabajar sobre el código de nuestro caso.

 **Importante:** durante la instalación de VS Code, habilita la opción "Add to PATH" para que puedas abrir VS Code desde una terminal con code .

- 2.4. Descargar e instalar Git desde la página oficial <https://git-scm.com/downloads>. GIT es el controlador de versiones de código que necesitarás para obtener el proyecto ejemplo.

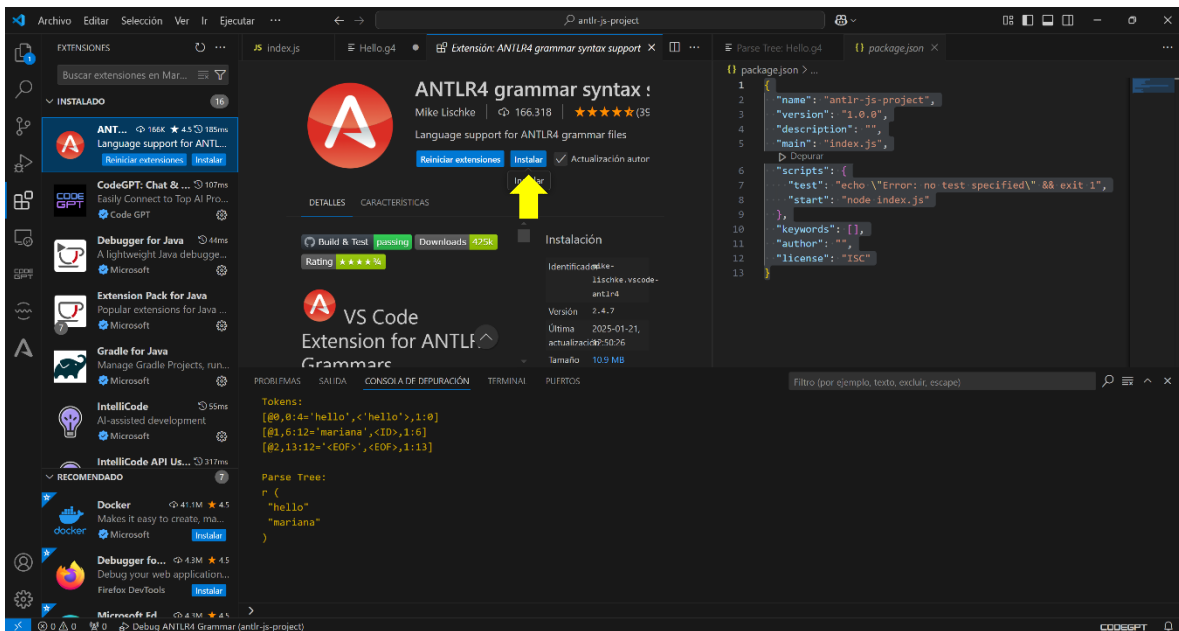
Para verificar que tienes correctamente instaladas las herramientas requeridas, puedes ejecutar los siguientes comandos en una terminal (Cmd, PowerShell, o terminal en Linux):

Comando	Salida esperada
<code>node -v</code>	versión de instalación de Node JS <i>Ej.: v20.10.0</i>
<code>npm -v</code>	versión de NPM instalada <i>Ej.: 10.2.3</i>
<code>code -v</code>	versión de Visual Studio Code instalada <i>Ej.: 1.96.4</i> <i>cd4ee3b1c348a13bafd8f9ad8060705f6d4b9cba</i> <i>x64</i>
<code>git -v</code>	Versión de GIT instalada <i>git version 2.43.0.windows.1</i>

3. CONFIGURACIÓN DE VISUAL STUDIO CODE PARA TRABAJAR CON ANTLR4

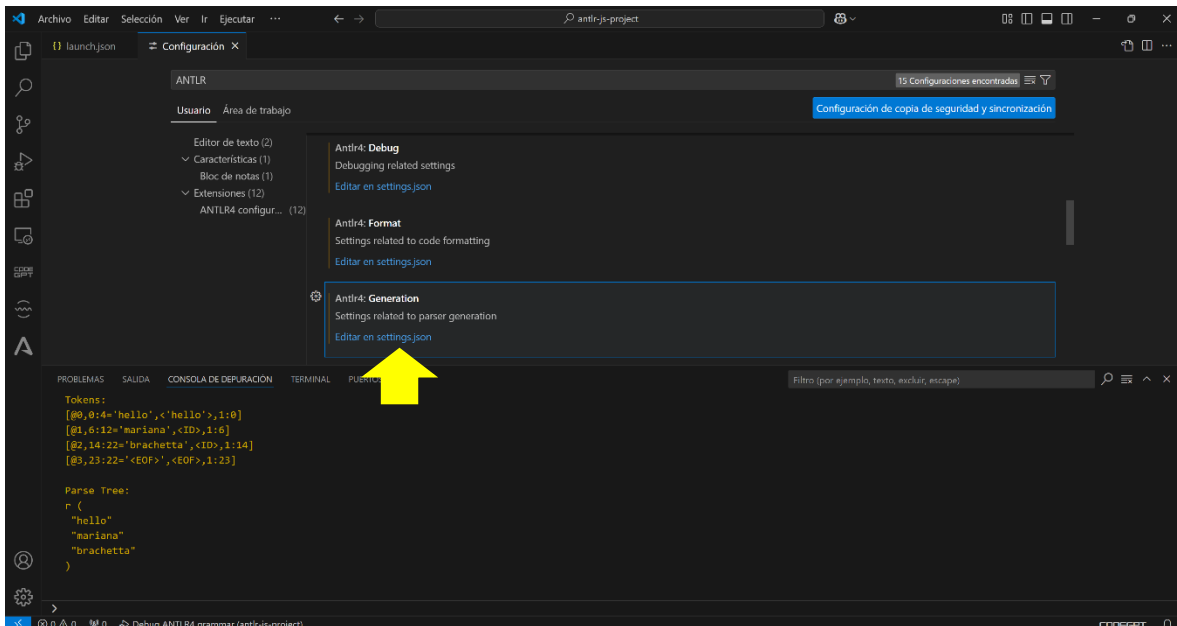
Para poder trabajar con ANTLR4 en VS Code, es recomendado que instales el plugin de ANTLR4 para VS Code y configures algunas propiedades relativas a la generación de código que hace ANTLR. Para ello:

- 3.1. Instala la extensión de ANTLR4 para Visual Studio Code: ve al menú Ver/Extensiones (CTRL + SHIFT + X), busca la extensión ANTLR4 grammar syntax support , selecciónala y dale clic a Instalar.



Importante: Luego de instalar la extensión reinicia VS Code para que tome los cambios.

3.2. Configurar las preferencias de generación de código de ANTLR: ve al menú Archivo/Preferencias/Configuración (CTL + ,) y busca las preferencias de ANTLR. Baja hasta la sección Antlr4:Generation y edita settings.json.



La configuración debe quedar del siguiente modo.

```
"antlr4.generation": {
  "alternativeJar": "antlr-4.13.2-complete.jar",
  "mode": "external",
  "listeners": true,
  "visitors": true,
  "language": "JavaScript",
  "outputDir": "./generated"
}
```

 **Importante:** Guarda el archivo de configuración antes de continuar (CTRL+S)

4. CLONAR EL PROYECTO EJEMPLO ssl-antlr-calculator

Vamos ahora a obtener desde un repositorio público GitHub el proyecto de código que seguiremos como caso de estudio. Para ello:

- 4.1. Abre una ventana de comandos (Cmd, PowerShell, o terminal en Linux).
- 4.2. Clona el proyecto desde el repositorio GitHub utilizando el siguiente comando:

```
git clone https://github.com/mbrachetta/ssl-antlr-calculator.git
```

Si el proyecto se clonó correctamente deberías ver el mensaje “done” en tu salida

```
Cloning into 'ssl-antlr-calculator'...
remote: Enumerating objects: 113, done.
remote: Counting objects: 100% (113/113), done.
remote: Compressing objects: 100% (93/93), done.
remote: Total 113 (delta 12), reused 113 (delta 12), pack-reused 0 (from 0)
Receiving objects: 40% (46/113), 1.98 MiB | 3.94 MiB/s
Receiving objects: 100% (113/113), 2.20 MiB | 4.12 MiB/s, done.
Resolving deltas: 100% (12/12), done.
```

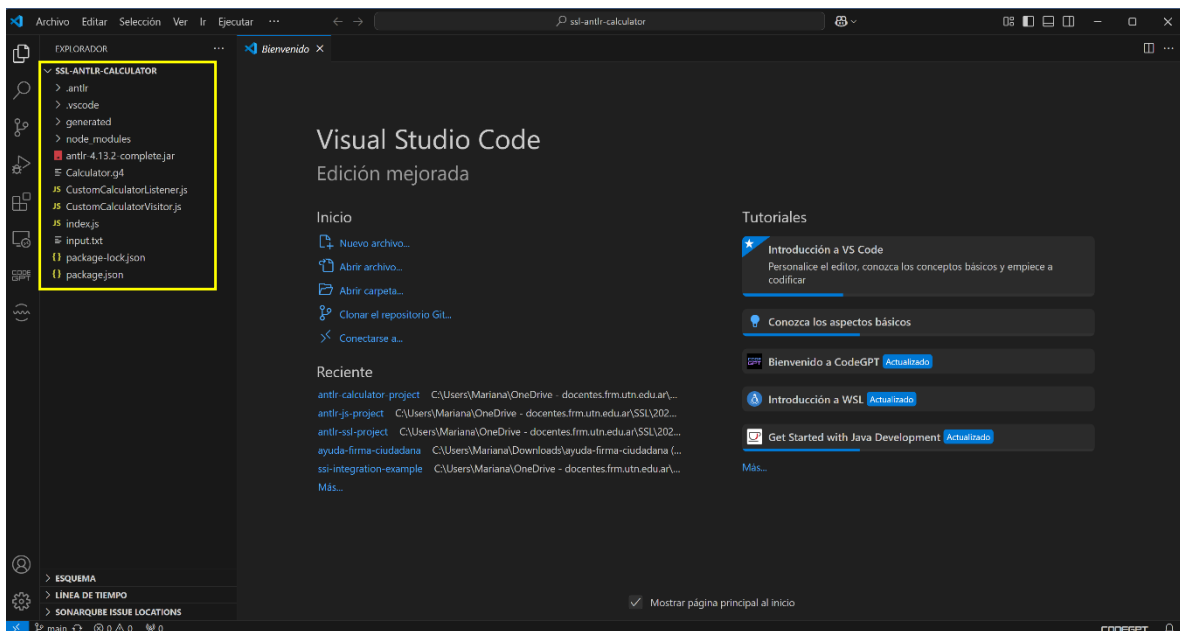
4.3. Cambia al directorio raíz del proyecto con el comando:

```
cd ssl-antlr-calculator
```

4.4. Abre VS Code para trabajar con el código del proyecto. Para esto ejecuta en la ventana de comandos abierta

```
code .
```

Si todo esta correcto, deberías ver tu proyecto editable en VS Code.



5. ANALISIS DE LA GRAMÁTICA ANTLR4 DEFINIDA PARA EL CASO DE ESTUDIO

Al seleccionar el archivo `Calculator.g4` deberías ver la gramática propuesta para el caso de estudio.

```

1 grammar Calculator;
2
3 //Gramatica
4 prog: stat*;
5
6 stat: expr NEWLINE? #printExpr
7      | ID EQ expr NEWLINE? #assign
8      | NEWLINE #blank
9      ;
10
11 expr: expr op=(MUL|DIV) expr #MulDiv
12      | expr op=(ADD|SUB) expr #AddSub
13      | INT #int
14      | ID #id
15      | LPAREN expr RPAREN #parens
16      ;
17
18 //Lexemas
19 MUL : '*';
20 DIV : '/';
21 ADD : '+';
22 SUB : '-';
23 EQ : '=';
24 ID : [a-zA-Z]+;
25 INT : [0-9]+;
26 LPAREN : '(';
27 RPAREN : ')';
28 NEWLINE: '\n'? '\n';
29 WS: [ \t]+ -> skip;
  
```

Todos los archivos que contengan la descripción de una gramática en ANTLR4 deben tener extensión `.g4` y su nombre debe coincidir con el nombre dado a la gramática en la sentencia inicial `grammar`.

La gramática `Calculator.g4` define las reglas léxicas y sintácticas de un lenguaje de expresiones aritméticas simples como por ejemplo `3 + 5 * (2 - 8)`. El lenguaje también soporta la definición y uso de identificadores y expresiones de asignación.

El archivo `Calculator.g4` se divide en dos secciones principales:

- Léxico (Definición de Tokens): define los elementos básicos del lenguaje, como números, operadores y paréntesis.
- Sintaxis (reglas sintácticas): define cómo se combinan los tokens para formar expresiones válidas.

Recuerda: Los tokens son las unidades más pequeñas del lenguaje que la gramática reconoce. Aquí hay algunos tokens principales en `Calculator.g4`:

- **NUM**: representa números enteros. Por ejemplo, 123.
- **ADD**: representa el operador de suma +.
- **SUB**: Representa el operador de resta -.
- **MUL**: representa el operador de multiplicación *.
- **DIV**: representa el operador de división /.

- **LPAREN** : representa el paréntesis de apertura (.
- **RPAREN** : representa el paréntesis de cierre).
- **NEWLINE** : reconoce un salto de línea, como `\r\n` (para Windows) o `\n` (para Unix/Linux).
- **WS** : **reconoce** la ocurrencia de uno o más espacios en blanco o tabulaciones, en cuyo caso las ignora (skip).

Las reglas definen cómo se combinan los tokens para formar expresiones válidas. En nuestra gramática ejemplo, tenemos las siguientes reglas principales:

- **prog** : Es el axioma de la gramática y define que un programa **prog** consiste en una o más declaraciones **stat** (sentencias).
- **stat** : define las posibles declaraciones o sentencias en el programa.
 - **printExpr** : Una expresión seguida de una nueva línea.
 - **assign** : Una asignación de una expresión a un identificador (**ID**) seguida de una nueva línea.
 - **blank** : Una línea en blanco.

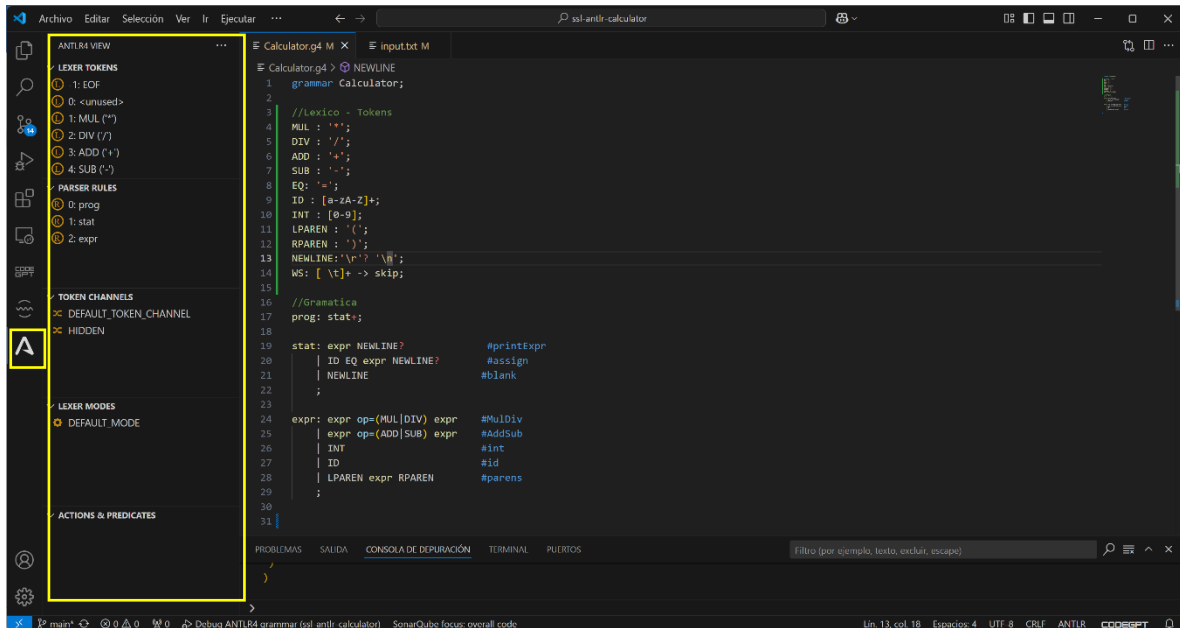
`#printExpr`, `#assign`, `#blank` son etiquetas que nos servirán luego para identificar la ocurrencia de cada una de estas producciones alternativas al procesar una entrada.

- **expr** : Es la regla que define la estructura sintáctica de una expresión. Puede ser una suma, resta, multiplicación, división o una expresión entre paréntesis.
- **op** : Define los operadores que se pueden usar en las expresiones (**MUL**, **DIV**, **ADD**, **SUB**). ANTLR4 define la precedencia de operadores en función de cómo se ordenan las reglas en la gramática (de arriba hacia abajo). Por eso es importante que la definición de las operaciones de multiplicación y división estén descriptas en la gramática antes que las operaciones de suma y resta.

Cuando la gramática procesa como entrada una expresión, por ejemplo: $3 + 5 * (2 - 8)$, se siguen las reglas definidas para analizar léxica y sintácticamente la expresión y construir el árbol de análisis sintáctico, partiendo de la raíz **prog** (axioma).

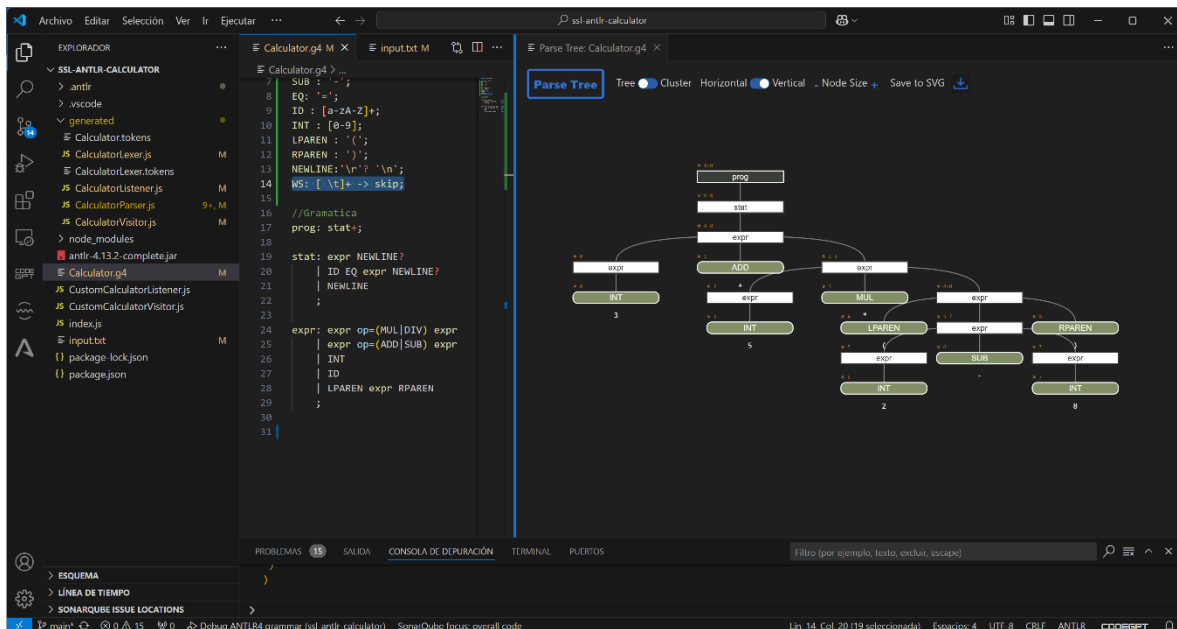
Veamos cómo trabaja la gramática sobre una expresión en particular:

- 5.1. Edita el archivo `input.txt` y tipea una expresión algebraica. Por ejemplo: $3 + 5 * (2 - 8)$. Guarda el archivo (CTRL + S).
- 5.2. Edita el archivo `Calculator.g4` y haz click en el botón del plugin de ANTLR en la barra lateral izquierda.



Si el plugin de ANTLR4 quedó correctamente instalado deberías ver en la vista de ANTLR, los tokens y las reglas gramaticales identificadas como lo ilustra la figura anterior.

5.3. Vamos ahora a utilizar el plugin de ANTLR4 para construir el árbol de sintaxis para la cadena de entrada ingresada en `input.txt`. Para ello, presiona F5 (lo que equivale a ejecutar y depurar el proyecto a partir de la especificación dada en el archivo `launch.JSON`). Deberías ver el árbol de sintaxis concreta para la expresión ingresada en `input.txt`



Puedes cambiar la entrada en `input.txt` (recuerda guardad CTRL + S) y regenerar el árbol (F5) hasta que comprendas cómo está trabajando la gramática `Calculator.g4`

Si no lo hiciste antes, este es un buen momento para que estudies como escribir gramáticas con ANTLR4. Para ello puedes consultar la bibliografía, el material de referencia y utilizar ANTLR Lab.

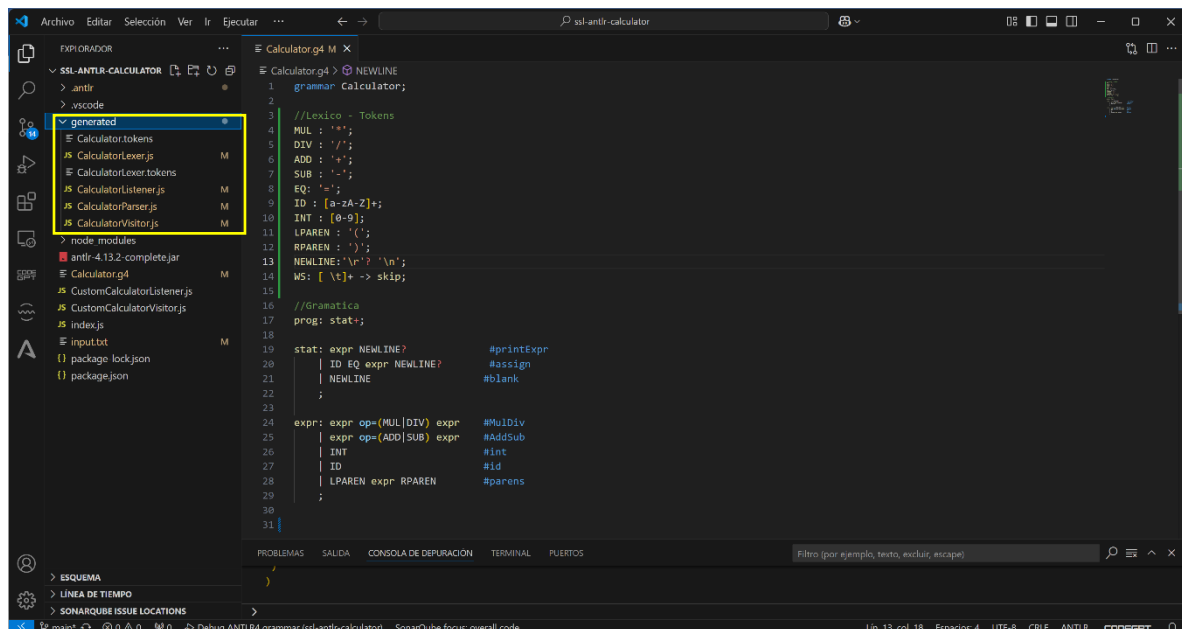
6. GENERACIÓN AUTOMÁTICA DEL LEXER Y PARSER CON ANTLR

Para que al analizar una cadena de entrada el analizador pueda construir el árbol de sintaxis (tal como vimos en el punto anterior para la expresión $3 + 5 * (2 - 8)$), ANTLR generó previamente y de forma automática el analizador léxico (lexer o scanner) y el analizador sintáctico (parser) para la gramática dada.

Recuerda que ANTLR (Another Tool for Language Recognition) es una herramienta poderosa que nos permite definir gramáticas para lenguajes de programación y automáticamente generar el código necesario para analizarlos. En nuestro proyecto, hemos configurado ANTLR para generar el lexer y el parser en JavaScript. Por lo tanto, si no lo has hecho antes, a partir de ahora será necesario que comprendas conceptos básicos de programación en JavaScript para poder avanzar con el estudio y la comprensión del caso.

- **Lexer:** se encarga de dividir el texto de entrada en tokens. Los tokens son las unidades básicas del lenguaje, como números, operadores y paréntesis.
- **Parser:** toma los tokens generados por el lexer y los organiza según las reglas de la gramática para formar una estructura que represente la expresión completa.

Si en la vista del Explorador del proyecto en VS Code, despliegas la carpeta `generated`, verás los archivos que ANTLR4 ha generado automáticamente para la gramática `Calculator.g4`.



Los archivos `CalculatorLexer.js` y `CalculatorParser.js` han sido generados automáticamente por ANTLR y contienen el código JavaScript que implementa el analizador léxico y el analizador sintáctico respectivamente para la gramática `Calculator`.

✍ **Importante:** no es una buena práctica que toques código autogenerado, porque cualquier cambio que hagas se perderá cuando los archivos se regeneren automáticamente ante cualquier cambio en la definición de la gramática. Es decir, si cambias algo en `Calculator.g4`, los archivos `CalculatorLexer.js` y `CalculatorParser.js` se regeneran.

7. AGREGANDO SEMÁNTICA PARA EVALUAR EXPRESIONES ARITMÉTICAS

En este punto deberías tener claro cómo trabaja ANTLR4 para generar automáticamente un analizador (léxico y sintáctico) en JavaScript, que reconozca el lenguaje de expresiones aritméticas simples definido en nuestro caso de estudio `Calculator.g4`.

Este analizador es capaz de tomar una expresión en la entrada y evaluar si es sintácticamente correcta, en cuyo caso podrá construir el árbol. En caso de detectar un error, lo informará.

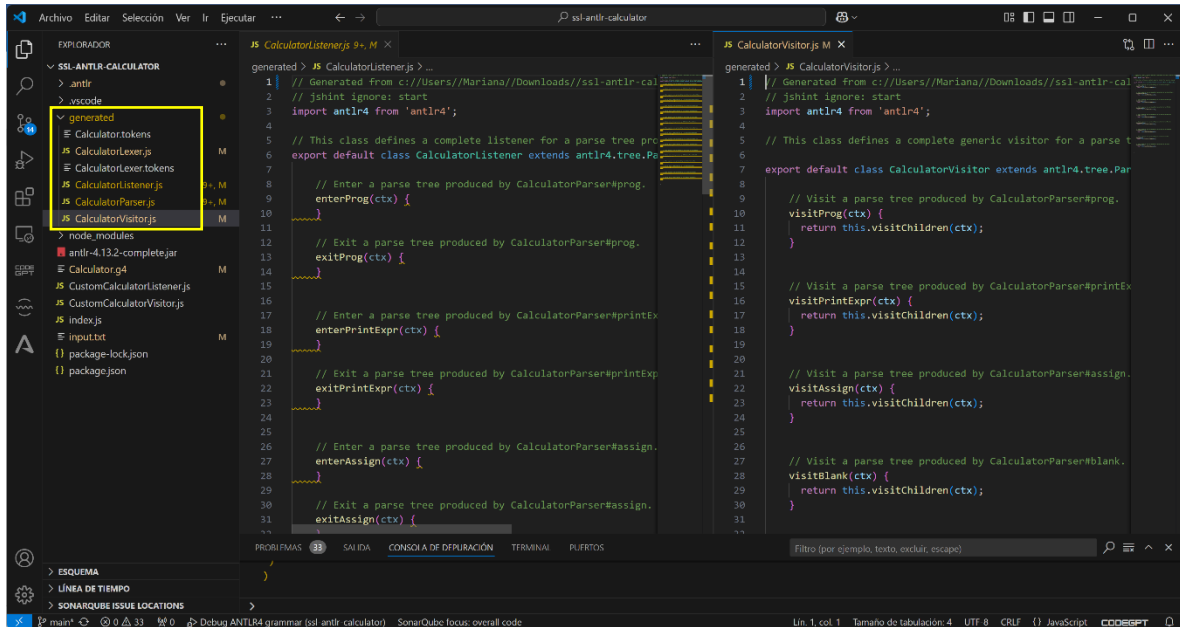
Ahora, nos planteamos como ir más allá en la construcción de nuestro traductor, para incorporar una etapa de análisis semántico y ejecución, que nos permita interpretar expresiones algebraicas válidas y evaluarlas obteniendo un resultado.

ANTLR no proporciona elementos para definir la semántica de nuestro lenguaje, ni genera automáticamente un analizador semántico, pero si proporciona las bases para que podamos construirlo programando directamente en JavaScript (o el lenguaje de trabajo elegido). Estas bases, son los `Listeners` y `Visitors` cuyo concepto y funcionamiento deberías profundizar con lectura bibliográfica y material complementario.

Un `listener`, es un bloque de código, generado automáticamente por ANTLR, que se ejecutará cuando se produzcan eventos específicos durante el recorrido del árbol de análisis sintáctico (`parse tree`). Estos eventos, se producen cada vez que se entra o se sale de un nodo del árbol (Por ejemplo, para nuestro caso de estudio, algunos de estos eventos podrían ser: `enterProg`, `exitProg`, `enterStat`, `exitStat`, `enterExpr` o `exitExpr`. Luego, tu podrías programar en JavaScript la forma en que nuestro traductor respondería a la ocurrencia de cada uno de estos eventos.

Un `Visitor`, es un bloque de código, que ANTLR puede generar, pero en este caso para recorrer el árbol de análisis sintáctico en una manera explícita, controlada por el programador. Para cada nodo del árbol, se define una operación a realizar. A diferencia de los `Listeners`, los `Visitors` requieren que se llame explícitamente a los métodos de visita para cada nodo del árbol. Esto proporciona un mayor control sobre el recorrido del árbol y permite realizar operaciones específicas en cada nodo. En nuestro caso de estudio, por ejemplo: ANTLR generará métodos vacíos `visitProg`, `visitStat`, `visitExpr`, `visitAssign`, `visitID`, etc. Luego, tu podrías implementar en JavaScript las acciones específicas que quieres que se ejecuten cuando mediante el método `visitExpr` se visite una expresión.

7.1. Si en el Explorador del proyecto, abres la carpeta `generated`, verás los archivos `CalculatorListener.js` y `CalculatorVisitor.js` generados automáticamente por ANTLR.



El archivo `CalculatorListener.js` define una interfaz Listener generada por ANTLR. Esta interfaz define métodos que se llaman automáticamente cuando se entra o se sale de las reglas de la gramática durante el análisis sintáctico.

El archivo `CalculatorVisitor.js` define una interfaz Visitor generada por ANTLR. Esta interfaz define métodos que se llaman explícitamente para recorrer y operar sobre los nodos del árbol de análisis sintáctico.

Como antes explicáramos, estos archivos son esenciales para personalizar el comportamiento del análisis sintáctico y realizar operaciones específicas en las expresiones matemáticas definidas en nuestra gramática ejemplo.

Puedes observar que tanto los métodos de `CalculatorListener` como de `CalculatorVisitor` están vacíos. Por ello decimos que definen una interfaz. Si bien JavaScript no soporta interfaces como tal, lo que estamos expresando con esta idea es que constituyen un protocolo, una plantilla de comportamiento que podemos utilizar para agregar semántica y capacidades de ejecución a nuestro traductor.

Importante: No es una buena práctica implementar nuestro código directamente sobre los métodos de las clases `CalculatorListener.js` o `CalculatorVisitor.js`. Dado que estos archivos se regeneran automáticamente cada vez que cambiamos algo en la gramática, perderíamos lo que hayamos programado cada vez. Por ello, vamos a programar en clases personalizadas que hereden de estas clases.

Nota: Los conceptos de interfaces, clases y objetos, así como el concepto de herencia y polimorfismo escapan al alcance de esta asignatura, por lo que simplemente basta con que observes que en este caso las funciones y/o procedimientos (métodos) se implementan dentro de una estructura de código superior llamada clase.

7.2. En el marco de nuestro caso de estudio, utilizaremos un Visitor para recorrer nuestro árbol, ejecutando el código que corresponde en cada nodo, a fin de evaluar una expresión aritmética. Este Visitor, está implementado en el archivo `CustomCalculatorVisitor.js` que, si bien se vincula con su padre `CalculatorVisitor`, no se regenerará/modificará ante cada cambio en la gramática. Vamos ahora a explicar con detalle la lógica implementada en este archivo para nuestro caso de estudio.

La clase `CustomCalculatorVisitor` extiende la funcionalidad del Visitor generado por ANTLR para evaluar expresiones matemáticas. Implementa métodos específicos para cada regla de la gramática, permitiendo calcular el resultado de las expresiones.

No explicaremos en detalle el funcionamiento de cada método en esta clase, pero damos a continuación una breve explicación que te permita entender la lógica general de la semántica implementada en nuestro caso de estudio.

Cada vez que se detecte un entero `#int` al recorrer el árbol (`parseTree`), se devuelve el valor.

Si lo que se detecta es una expresión del tipo `#MulDiv`, se sabe que la misma tendrá dos sub-expresiones (un término a la derecha y otro a la izquierda), entonces se pide al visitor que las visite recursivamente. Cuando tanto por la izquierda como por la derecha hemos llegado a obtener un valor, se obtendrá el valor final multiplicando o dividiendo ambos términos, según sea el operador (`MUL` | `DIV`) detectado.

Si lo que se detecta es una expresión del tipo `#AddSub`, se sabe que la misma tendrá dos sub-expresiones (un término a la derecha y otro a la izquierda), entonces se pide al visitor que las visite recursivamente. Cuando tanto por la izquierda como por la derecha hemos llegado a obtener un valor, se obtendrá el valor final sumando o restando ambos términos, según sea el operador (`ADD` | `SUB`) detectado.

Si lo que se detecta es una expresión del tipo `#parens`, se sabe que hay que evaluar la sub-expresión encerrada entre paréntesis. Entonces se ordena al visitor que visite dicha sub-expresión.

Para manejar la asignación de un valor a un identificador, utilizamos una memoria temporal que se implementa con la variable `memory` del tipo `Map` definida en el método constructor. Un `Map`, es una lista dinámica que almacena pares del tipo `<clave, valor>`. En ella almacenamos cada par `<id, valor>`. De esta forma, cuando se detecta un `#assign`, se obtiene el nombre del ID asignado y su valor y se guardan en esta memoria. Luego, cuando se detecta un `#id` en una expresión algebraica, el método que visita este ID lo busca en la memoria y obtiene su valor.

En definitiva, el código de `CustomCalculatorVisitor` implementa métodos para evaluar expresiones matemáticas, manejar asignaciones y recuperar valores de variables.

Utiliza un `Map` para almacenar resultados intermedios y variables, permitiendo un procesamiento eficiente y correcto de las expresiones.

No tiene sentido profundizar más en su explicación, porque este código no es más que la resolución puntual de la semántica de nuestro caso de estudio en particular. Será tu tarea, implementar el `CustomVisitor` de la gramática que tu diseñes.

Nota: En el proyecto encontrarán una mínima implementación del patrón Listener, en el archivo `CustomCalculatorListener.js`; no obstante esto se agrega solo con fines ilustrativos. No forma parte de la lógica necesaria para implementar el traductor en este caso.

7.3. Hasta aquí, tenemos todos los elementos listos: el lexer que resuelve el análisis léxico, el parser que resuelve el análisis sintáctico y el visitor que agrega semántica a nuestro analizador permitiendo resolver las expresiones algebraicas. Vamos ahora a ensamblar estas partes en nuestro programa principal para obtener un analizador/traductor ejecutable. Para ello, revisemos el contenido del archivo `index.js` analizando línea por línea de la función principal `main()`.

<pre>let input; try { input = fs.readFileSync('input.txt', 'utf8'); } catch (err) { input = await leerCadena(); }</pre>	<p>En este bloque se declara la variable <code>input</code> (del tipo <code>string</code>) y se intenta cargar en ella una cadena de entrada leída desde el archivo <code>input.txt</code>. Si el archivo no existe o se produce un error al leerlo, se pide al usuario que ingrese una cadena desde la consola.</p>
<pre>let inputStream=CharStreams.fromString(input); let lexer=new CalculatorLexer(inputStream); let tokenStream=new CommonTokenStream(lexer); let parser=new CalculatorParser(tokenStream); let tree=parser.prog();</pre>	<p>En este bloque se obtiene en <code>inputStream</code> el flujo de caracteres de entrada desde la cadena <code>input</code>. Se pasa ese flujo al <code>lexer</code> para obtener el código tokenizado. Luego, el flujo de tokens (código tokenizado o <code>tokenStream</code>) es pasado al <code>parser</code> para obtener el árbol de derivación que se almacena en la variable <code>tree</code>.</p>
<pre>if (parser.syntaxErrorsCount > 0) { console.error("\nSe encontraron errores de sintaxis en la entrada."); } else { console.log("\nEntrada válida."); const cadena_tree = tree.toStringTree(parser.ruleNames); console.log(`Árbol de derivación: \${cadena_tree}`); const visitor= new CustomCalculatorVisitor(); visitor.visit(tree); }</pre>	<p>En este bloque se consulta si el <code>parser</code> encontró errores sintácticos, en cuyo caso muestra el mensaje de error.</p> <p>En caso que la cadena sea válida y se haya podido construir el árbol correctamente, el mismo se imprime desde la consola (en modo texto).</p> <p>Luego, se crea una instancia de un <code>visitor</code> y se activa su método <code>visit</code> sobre el árbol (<code>tree</code>) para que recorra recursivamente los nodos y vaya resolviendo la expresión aritmética evaluada.</p>

Nota: En JavaScript, el parser que construye ANTLR, muestra el árbol en formato de texto estructurado, no hay una forma gráfica de hacerlo. Se puede escribir una función que convierta ese texto a formato JSON y luego, con un graficador en línea que convierta JSON a gráficos jerárquicos, interpretar y construir el árbol.

7.4. Con esto ya hemos completado la explicación de nuestro proyecto y estamos en condiciones de ejecutarlo para ver nuestro analizador funcionando. Para ello, ve a la consola de comando, en el directorio raíz del proyecto y ejecuta el comando

```
npm start
```

Deberías ver la salida para la expresión de entrada que hayas configurado en `input.txt`

```
C:\Users\Mariana\Downloads\ssl-antlr-calculator>npm start
> antlr-calculator-project@1.0.0 start
> node index.js

Entrada válida.
Árbol de derivación: (prog (stat (expr (expr 3) + (expr (expr 5) * (expr ( (expr (expr 2) - (expr 8)) ))))))
Resultado: -27
C:\Users\Mariana\Downloads\ssl-antlr-calculator>
```

En esta salida verás, si la cadena es válida o no. En caso que sea válida el árbol de derivación en formato texto y el resultado de evaluar la expresión.

Puedes también probar quitando el archivo `input.txt` e ingresando cadenas por teclado. O incorporando múltiples líneas en el archivo `input.txt` para que evalúe en secuencia todas las expresiones.

8. CONCLUSIÓN Y PRÓXIMOS PASOS

Esta guía ilustra, sobre un caso de estudio concreto, el uso de ANTLR4 y JavaScript para construir una analizador y evaluador de expresiones aritméticas. A la vez, sienta la estructura básica de proyecto que hacia adelante se puede utilizar para definir nuevas gramáticas y analizadores.

Te sugerimos ahora como ejercicio, definir en el proyecto o en una copia de él, una nueva gramática (.g4), que sea simple y que describa alguna estructura sintáctica de un lenguaje de programación, real o ficticio. Luego, implementar un Listener o Visitor, según estimes conveniente para agregar semántica a tu analizador y traducir una entrada en el lenguaje de base a una salida en otro lenguaje destino (también real o ficticio). Por ejemplo, podrías traducir la siguiente gramática de declaraciones y asignaciones en pseudocódigo a código C/C++.

```
<programa> ::= {<declaracion> | <asignacion>}
<declaracion> ::= ('Definir' | 'Declarar') <identificador> {',' <identificador>} como <tipo> ';'
<asignacion> ::= <identificador> '=' <expresion> ';';
<expresion> ::= <termino> { ('+' | '-') <termino> }
<termino> ::= <factor> { ('*' | '/') <factor> }
<factor> ::= '(' <expresion> ')' | <identificador> | <numero>
<tipo> ::= 'Entero' | 'Real' | 'Cadena' | 'Booleano'
<identificador> ::= <letra> { <letra> | <digito> | '_' }
<numero> ::= [ '-' ] <digito> { <digito> }
<letra> ::= 'a' | 'b' | 'c' | ... | 'z' | 'A' | 'B' | ... | 'Z'
<digito> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

9. REFERENCIAS BIBLIOGRÁFICAS Y MATERIAL DE CONSULTA

- Book: Terrence Par, *The Definitive ANTLR 4 Reference*, 2013, The Pragmatic Programmers. [Disponible online en el campus virtual de la cátedra].
- Sitio oficial de ANTLR: <https://www.antlr.org>
- Video que ilustra la definición de una gramática simple. [Disponible online en <https://youtu.be/wrVxkMN27p0?si=plb1tlbbNJ5hu5Cj>]