



PROYECTO FINAL DE CLASE

**DEPARTAMENTO ACADÉMICO DE INGENIERÍA EN
SISTEMAS**

REALIZADO EN AGOSTO DE 2025





UNAH CAMPUS CHOLUTECA

Proyecto Final de Clase



Fecha de Envío

03 de agosto del 2025

Asignatura

IS913 – Diseño de Compiladores

Catedrático

Ph.D. Wilson Octavio Villanueva Castillo

Integrantes

Nestor Yuvini Ordoñez Baca – 20202300158

Ury Roberto Aguirre Ramirez – 20212320167

Jorlin Fernando Rosa Arrivillaga – 20192330138



INDICE DE CONTENIDO

I. <i>INTRODUCCIÓN.....</i>	4
II. <i>CONTENIDO.....</i>	5
1. Requisitos Del Entorno De Desarrollo	5
2. Preparación Del Entorno De Trabajo.....	7
3. Descripción Del Lenguaje Easy	15
4. Estructura Lexica y Fase Léxica	16
5. Fase Sintáctica.....	25
6. Pruebas realizadas en fase sintáctica	41
7. Fase Semántica.....	52
8. Código Main.java.....	54
III. <i>CodeGenerator.java</i>	59
Pruebas	108
IV. <i>Problemas surgidos durante la realización del Proyecto</i>	125
V. <i>CONCLUSIÓN.....</i>	129
VI. <i>REFERENCIA.....</i>	130
VII. <i>ANEXOS.....</i>	131
VIII. <i>Compilación Multiplataforma.....</i>	132



I. INTRODUCCIÓN

El presente informe documenta el proceso completo de diseño y desarrollo del compilador **EasyC**, proyecto académico realizado en el marco de la asignatura *Diseño de Compiladores-IS913*. A lo largo del periodo, se ha seguido una ruta formativa progresiva que nos ha permitido comprender y aplicar las distintas etapas involucradas en la construcción de un compilador, desde el análisis léxico hasta la generación de código intermedio.

EasyC surge como una propuesta didáctica inspirada en la sintaxis del lenguaje de programación C, pero con una estructura simplificada y accesible. El objetivo principal de este proyecto ha sido facilitar el entendimiento de los conceptos teóricos mediante la creación de un lenguaje propio, permitiendo así experimentar de forma práctica con cada componente del proceso de compilación.

Durante el primer parcial, se trabajó con *Flex* para implementar el analizador léxico, encargado de identificar los tokens definidos por el lenguaje. En el segundo parcial, se utilizó *Bison* para desarrollar el analizador sintáctico, donde se definieron las reglas gramaticales que rigen la estructura del lenguaje **EasyC**. Finalmente, en el tercer parcial, se integró el uso de ANTLR 4, una herramienta moderna y robusta que permite la construcción de analizadores mediante una gramática declarativa, lo que facilitó la visualización de árboles sintácticos y el manejo de estructuras más complejas del lenguaje.

Este informe incluye la documentación detallada de cada una de las fases del proyecto, los archivos fuente utilizados, las pruebas realizadas, las decisiones de diseño tomadas, y los desafíos enfrentados durante el desarrollo del compilador.

EasyC no solo representa el producto final de un proceso académico, sino también una herramienta que sintetiza el aprendizaje adquirido en torno al diseño de lenguajes, la teoría de autómatas, y la lógica detrás de los compiladores modernos. Este documento tiene como propósito dejar constancia técnica del trabajo realizado, y a su vez, servir como guía de referencia para estudiantes que en el futuro busquen emprender proyectos similares.



II. CONTENIDO

1. Requisitos Del Entorno De Desarrollo

Esta sección presenta las herramientas y configuraciones necesarias para desarrollar y ejecutar el compilador del lenguaje **Easy**, utilizando ANTLR como base para el análisis léxico y sintáctico.

Herramientas necesarias en UBUNTU:

- **Java Development Kit (JDK)**

Permite compilar y ejecutar los archivos *.java* generados por ANTLR. Se utilizó OpenJDK 17, instalado desde los repositorios oficiales.

- **ANTLR v4**

Herramienta fundamental para el análisis léxico y sintáctico.

Versión utilizada: **4.13.2**, descargada manualmente desde el sitio oficial de ANTLR.

- **Editor de código VS Code**

Utilizado para la edición del código fuente del compilador. Compatible con Ubuntu y permite instalar la extensión oficial de ANTLR.

- **Compilador de Java**

Viene incluido en el JDK. Se utilizó para compilar los archivos generados por ANTLR, con el classpath configurado de forma manual.

- **Terminal de Ubuntu**

Se usó para realizar todas las acciones del proceso de compilación y pruebas.

Herramientas necesarias en WINDOWS:

- **Java Development Kit (JDK)**

Indispensable para compilar y ejecutar el código fuente generado por ANTLR. Se utilizó OpenJDK 17, instalado desde Adoptium o SDKMAN.

- **ANTLR v4**

Herramienta utilizada para construir el analizador léxico y sintáctico del lenguaje.

Versión empleada: **4.13.2**, descargada desde el sitio oficial de ANTLR.



- **Editor de código VS Code**
Seleccionado para editar archivos fuente del compilador. Se recomienda agregar la extensión *ANTLR4 grammar syntax support*.
- **Compilador de Java**
Incluido en el JDK, permite compilar los archivos .java generados por ANTLR.
- **Símbolo del sistema / PowerShell**
Ambiente de línea de comandos desde el cual se ejecutaron los procesos de compilación y prueba.

Herramientas necesarias en macOS:

- **Java Development Kit (JDK)**
Se requiere para compilar y ejecutar los archivos .java generados por ANTLR. Se recomienda usar la versión OpenJDK 24.0.1 disponible desde Homebrew.
- **ANTLR v4**
Herramienta principal para generar analizadores léxicos y sintácticos a partir de gramáticas.
Versión utilizada: **4.13.1**
- **Editor de código VS Code**
Recomendado para editar archivos fuente del compilador (.g4, .java, etc.). Se sugiere instalar la extensión *ANTLR4 grammar syntax support* para facilitar la navegación y validación de gramáticas.
- **Compilador de Java**
Incluido en el JDK, se utiliza para compilar los archivos generados por ANTLR. Se emplea el comando javac desde terminal.
- **Terminal de macOS**
Utilizada para ejecutar los comandos de instalación, compilación y pruebas del compilador.



2. Preparación Del Entorno De Trabajo

A continuación, se detallan los pasos necesarios para instalar ANTLR 4.13.1 y configurar el entorno de desarrollo en macOS (Sequoia):

- macOS

Paso 1: Instalar Homebrew

```
ferarrivillaga@MacBook-Pro-de-Fernando ~ % /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

¿Qué hace?

Instala **Homebrew**, el gestor de paquetes más utilizado en macOS. Te permitirá instalar fácilmente Java y ANTLR sin tener que descargarlos manualmente.

Paso 2: Instalar Java (OpenJDK)

```
ferarrivillaga@MacBook-Pro-de-Fernando ~ % brew install openjdk
```

¿Qué hace?

Instala **Java Development Kit (JDK)**. ANTLR está programado en Java, por lo que es necesario tenerlo para poder ejecutar el compilador y las herramientas asociadas.

Paso 3: Agregar Java al PATH (si aún no funciona)



```
ferarrivillaga@MacBook-Pro-de-Fernando ~ % echo 'export PATH="/opt/homebrew/opt/openjdk/bin:$PATH"' >> ~/.zprofile
file
source ~/.zprofile
```

¿Qué hace?

Agrega la ruta de instalación de Java al PATH del sistema. Esto permite que puedas ejecutar java desde cualquier carpeta en la terminal.

Paso 4: Verificar que Java esta correctamente instalado

```
ferarrivillaga@MacBook-Pro-de-Fernando ~ % java -version
```

¿Qué hace?

Muestra la versión de Java instalada. Deberías ver algo como *openjdk version "24.0.1"*.

Paso 5: Instalar ANTLR 4

```
ferarrivillaga@MacBook-Pro-de-Fernando ~ % brew install antlr
```

¿Qué hace?





Descarga e instala ANTLR 4 (versión más reciente) junto con sus herramientas (antlr4 y grun).

Paso 6: Verificar que ANTLR 4 se instaló correctamente

- Verificar el comando ANTLR4

```
ferarrivillaga@MacBook-Pro-de-Fernando ~ % antlr4
```

¿Qué hace?

Ejecuta el generador de analizadores ANTLR. Si se instaló correctamente, mostrará un mensaje con las instrucciones de uso.

- Verificar el comando *grun* (TestRig)

```
ferarrivillaga@MacBook-Pro-de-Fernando ~ % grun
```

¿Qué hace?

Ejecuta el visualizador de árboles de análisis de ANTLR (útil para probar tu gramática).



- Windows

A continuación, se detallan los pasos necesarios para instalar ANTLR 4.13.2 y configurar el entorno de desarrollo en Windows:

1. descargar el JDK de Java https://download.oracle.com/java/24/latest/jdk-24_windows-x64_bin.exe ([sha256](#))
2. descargar el binario de ANTLR. Al momento de la realización de este documento la última versión del binario es 4.13.2 <https://www.antlr.org/download.html>

Development Tools
There are plug-ins for IntelliJ, NetBeans, and Eclipse.

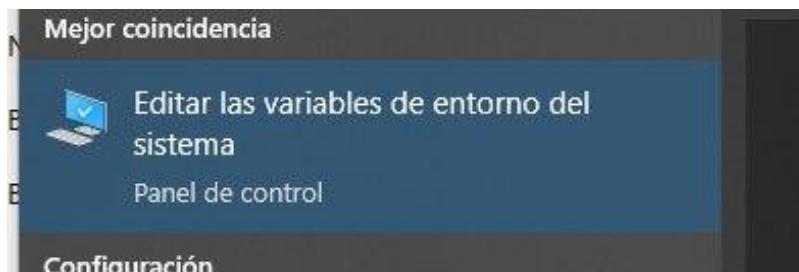
ANTLR tool and Java Target

- Complete ANTLR 4.13.2 Java binaries jar. Complete ANTLR 4.13.2 tool, Java runtime and ST 4.0.8, which lets you run the tool and the generated code.
- ANTLR 4.13.2 distribution (zip). Everything you need to build the tool and Java runtime from source.

3. instalar el JDk
4. creas una carpeta dentro del disco C:/ con el nombre antlr, ahí guardas el binario que descargaste.

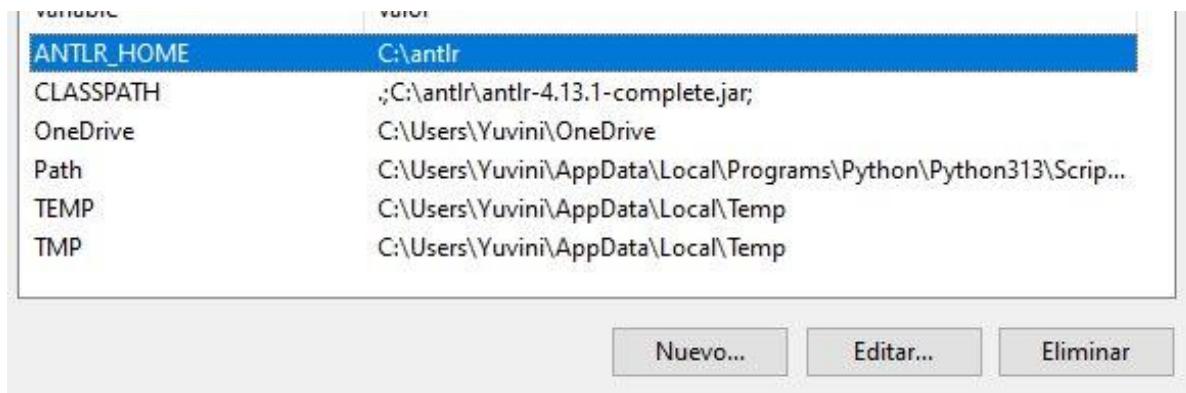
Este equipo > Disco local (C:) > antlr >				
Nombre	Fecha de modificación	Tipo	Tamaño	
.antlr	27/7/2025 13:36	Carpeta de archivos		
easy	31/7/2025 11:41	Carpeta de archivos		
pruebas	27/7/2025 13:33	Carpeta de archivos		
antlr-4.13.2-complete	26/7/2025 22:56	Executable Jar File	2,090 KB	

5. Ir a la sección de Variables de Entorno para crear las variables necesarias para el funcionamiento de ANTLR.



6. Creamos la primera variable que se llamara ANTLR_HOME a este la damos el valor de la ruta donde esta la carpeta que creamos en el disco C:/

La segunda variable se llamara CLASSPATH que es este permitirá que podamos acceder al jar de java y le asignamos el siguiente valor .;%ANTLR_HOME%\antlr-4.13.2-complete.jar;



7. Visualizamos via consola CMD si todo se instaló correctamente con el siguiente comando `java -cp "C:\antlr\antlr-4.13.2-complete.jar" org.antlr.v4.Tool` debe aparecer el mensaje que se muestra en la siguiente imagen.

```
C:\Users\Yuvini>java -cp "C:\antlr\antlr-4.13.2-complete.jar" org.antlr.v4.Tool
ANTLR Parser Generator Version 4.13.2
-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
```



8. La estructura de árbol de la carpeta debe quedar así

```
antlr-project/
|   └── antlr-4.13.2-complete.jar
|   └── Expr.g4
|   └── Main.java
```

- Linux (Ubuntu)

A continuación, se detallan los pasos necesarios para instalar ANTLR 4.13.2 y configurar el entorno de desarrollo en Linux (Ubuntu):

Paso 1 actualizamos los paquetes de nuestro sistema en esta ocasión Ubuntu

```
ubuntu@ubuntu:~$ sudo apt update && upgrade
Ign:1 cdrom://Ubuntu 24.04.2 LTS _Noble Numbat_ - Release amd64 (20250215) noble In
elease
Hit:2 cdrom://Ubuntu 24.04.2 LTS _Noble Numbat_ - Release amd64 (20250215) noble Re
ease
Hit:3 http://archive.ubuntu.com/ubuntu noble InRelease
Get:4 http://security.ubuntu.com/ubuntu noble-security InRelease [126 kB]
Get:5 http://archive.ubuntu.com/ubuntu noble-updates InRelease [126 kB]
Get:6 http://archive.ubuntu.com/ubuntu noble-backports InRelease [126 kB]
Get:8 http://security.ubuntu.com/ubuntu noble-security/main amd64 Packages [1,054 kB]
]
Get:9 http://security.ubuntu.com/ubuntu noble-security/main i386 Packages [316 kB]
Get:10 http://security.ubuntu.com/ubuntu noble-security/main Translation-en [183 kB]
Get:11 http://security.ubuntu.com/ubuntu noble-security/main amd64 Components [21.6 kB]
Get:12 http://security.ubuntu.com/ubuntu noble-security/main Icons (48x48) [13.4 kB]
Get:13 http://security.ubuntu.com/ubuntu noble-security/main Icons (64x64) [20.0 kB]
```





Paso 2 instalamos JDK de Java

```
ubuntu@ubuntu:~$ sudo apt-get install default-jdk
Reading package lists... 58%
```

Paso 3 instalación del curl .

```
sudo:
ubuntu@ubuntu:~$ sudo apt-get install curl
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following NEW packages will be installed:
  curl
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 226 kB of archives.
After this operation, 534 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu noble-updates/main amd64 curl amd64 8.5.0
-2ubuntu10.6 [226 kB]
Fetched 226 kB in 1s (213 kB/s)
```

Paso 4 Descargar el archivo .jar de ANTLR:

```
ubuntu@ubuntu:~$ curl -O https://www.antlr.org/download/antlr-4.13.2-complete.jar
% Total    % Received % Xferd  Average Speed   Time     Time      Current
          Dload  Upload   Total   Spent    Left  Speed
100 2089k  100 2089k    0     0  849k      0  0:00:02  0:00:02  --:--:--  849k
ubuntu@ubuntu:~$
```

Paso 5 Mover el archivo .jar a una ruta comúnmente usada para bibliotecas compartidas

```
ubuntu@ubuntu:~$ sudo mv antlr-4.13.2-complete.jar /usr/local/lib/
ubuntu@ubuntu:~$
```

Paso 6 Configuramos las variables de entorno:



```
ubuntubuntu:~$ sudo mv antlr-4.13.2-complete.jar /usr/local/lib/
ubuntu@ubuntu:~$ echo 'export CLASSPATH=".:/usr/local/lib/antlr-4.13.2-complete.jar:$CLASSPATH"' >> ~/.bashrc
ubuntu@ubuntu:~$ echo 'alias antlr4="java -jar /usr/local/lib/antlr-4.13.2-complete.jar"' >> ~/.bashrc
ubuntu@ubuntu:~$ echo 'alias grun="java org.antlr.v4.gui.TestRig"' >> ~/.bashrc
ubuntu@ubuntu:~$
```

Paso 7 Mediante el comando antlr4 verificamos que antlr está funcionando correctamente

```
Fredycatellon@fredycatellon-VMware-Virtual-Platform:~$ antlr4
ANTLR Parser Generator Version 4.13.2
-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
(arg: 1) antlr4 ^C
fredycatellon@fredycatellon-VMware-Virtual-Platform:~$ antlr4 |head -n 1
ANTLR Parser Generator Version 4.13.2
Fredycatellon@fredycatellon-VMware-Virtual-Platform:~$
```





3. Descripción Del Lenguaje Easy

EASY es un lenguaje de programación de alto nivel, multiparadigma (imperativo y estructurado) diseñado con fines educativos y de prototipado rápido. Su objetivo es simplificar el proceso de aprendizaje de la programación y la construcción de compiladores, combinando la familiaridad de C con una sintaxis más amigable y moderna.

Su extensión es **.ec**

- **Sintaxis clara** y simplificada, inspirada en C.
- **Tipado estático**, pero con inferencia básica de tipos.
- **Orientado a principiantes**, pero suficientemente robusto para proyectos medianos.
- **Biblioteca estándar mínima**, enfocada en entrada/salida, matemáticas y estructuras básicas.
- **Soporte para estructuras de control clásicas** (if, while, for, switch).
- **Modularidad** mediante funciones.

Actualmente Easy va evolucionando permitiendo estructuras de programación más complejas tales como Arreglos, Funciones y Clases (POO), aunque estas estructuras no son tan robustas como los lenguajes de programación tradicionales, si cumplen con la idea de Easy de ser un lenguaje de programación de uso educativo.

Una mención importante es que Easy es un lenguaje multiplataforma, lo que significa que es compatible con Windows, MAC y Linux.



4. Estructura Lexica y Fase Léxica

Token Léxico	Equivalente	Propósito
inicio	---	Marca el inicio de un bloque de código principal
fin	---	Marca el final de un bloque de código principal
si	if	Ejecuta un bloque de código si la condición es verdadera
sino	else	Bloque si ninguna condición previa es verdadera
mientras	while	Bucle que se ejecuta mientras la condición sea verdadera
para	for	Bucle controlado por contador
retornar	return	Devuelve un valor desde una función
leer	input / scanner	Entrada de datos
imprimir	print / system.out.print	Imprime datos en consola
seleccionar	switch	Selección entre múltiples opciones
caso	case	Opción específica dentro de seleccionar
defecto	default	Opción por defecto si ninguna coincide
romper	break	Finaliza una estructura como seleccionar o bucle
clase	class	Define una clase
super	super	Accede a métodos o atributos de la clase padre
metodo	method	Define un método dentro de una clase
funcion	function	Declaración de función
entero	int	Valor entero
flotante	float	Valor decimal
booleano	boolean	Valor verdadero o falso
cadena	string	Valor tipo string
VERDADERO	true	Valor booleano verdadero
FALSO	false	Valor booleano falso
::	==	Comparación de igualdad
>:	>=	Comparación mayor o igual que
<:	<=	Comparación menor o igual que
::!	!=	Comparación de desigualdad
:	=	Asignación



=	=	Asignación en casos
&	&&	Operador lógico AND
??		Operador lógico OR
>	>	Comparación mayor que
<	<	Comparación menor que
+	+	Operador de suma
-	-	Operador de resta
*	*	Operador de multiplicación
/	/	Operador de división
{	{	Delimita bloques de código
}	}	Cierre de bloques de código
((Inicio de lista de argumentos o expresiones
))	Fin de lista de argumentos o expresiones
[[Inicio de arreglo o lista
]	Fin de arreglo o lista	
;	;	Final de instrucción
,	,	Separación de elementos
.	.	Acceso a propiedades o métodos

Fase Léxica

Durante esta fase lee los caracteres de la entrada y los agrupa en “objetos token”. Junto con un símbolo de terminal que se utiliza para las decisiones de análisis sintáctico, un objeto token lleva información adicional en forma de valores de atributos. (Aho, Lam, Sethi, & Ullman, 2006)

Herramientas utilizadas en esta Fase

- ANTLR (ANother Tool for Language Recognition): Una herramienta más moderna que genera tanto analizadores léxicos como sintácticos a partir de una única gramática unificada. Puede generar código en múltiples lenguajes (Java, C#, Python, etc.), simplificando la creación del AST.

Flujo de Trabajo



1. Escribir la gramática (.g4)

Define la gramática de tu lenguaje o parser en un archivo como Easy.g4.

2. Generar los archivos de ANTLR (Lexer, Parser, etc.)

Ejecuta el *ANTLR Tool* para que genere el código fuente necesario (Lexer, Parser, etc.).

3. Compilar los archivos Java

Compila todos los archivos .java generados. Esto produce los .class.

4. Crear un visitante o listener personalizado

Puedes recorrer el árbol con un visitor (clase que extiende EasyBaseVisitor) o un listener (clase que extiende EasyBaseListener) para interpretar o compilar tu código.

5. Integración en un programa Java (Main.java)

Puedes crear tu propio archivo Main.java para cargar código fuente desde un archivo, analizarlo y recorrerlo con el visitor

Tokens Declarados en ANTLR

```
230    INICIO: 'inicio';
231    FIN: 'fin';
232    SI: 'si';
233    SINO: 'sino';
234    MIENTRAS: 'mientras';
235    PARA: 'para';
236    RETORNAR: 'retornar';
237    LEER: 'leer';
238    IMPRIMIR: 'imprimir';
239    SELECCIONAR: 'seleccionar';
240    CASO: 'caso';
241    DEFECTO: 'defecto';
242    ROMPER: 'romper';
243    CLASE: 'clase';
244    SUPER: 'super';
245    METODO: 'metodo';
246    FUNCION: 'funcion';
247
248    ENTERO: 'entero';
249    FLOTANTE: 'flotante';
250    BOOLEANO: 'booleano';
251    CADENA: 'cadena';
252
253    VERDADERO: 'verdadero';
254    FALSO: 'falso';
255
256    ID: [a-zA-Z_][a-zA-Z0-9_]*;
257
258    LITENTERO: '-'?[0-9]+;
259    LITFLOTANTE: '-'?[0-9]+(\.[0-9]+)([ff])?;
260    LITERALCADENA: """ .*? """;
```





```
262  IGUALIGUAL: '::';
263  MAYORIGUAL: '>:';
264  MENORIGUAL: '<:';
265  DIFERENTEDE: ':!';
266  IGUAL: ':';
267  IGUAL_CASO: '=';
268
269  Y: '&';
270  O: '??';
271
272  MAYORQUE: '>';
273  MENORQUE: '<';
274
275  SUMA: '+';
276  RESTA: '-';
277  MULTIPLICACION: '*';
278  DIVISION: '/';
279
280  LLAVEIZQ: '{';
281  LLAVEDER: '}';
282  PARIZQ: '(';
283  PARDER: ')';
284  CORCHETEIZQ: '[';
285  CORCHETEDER: ']';
286  PUNTOYCOMA: ';';
287  COMA: ',';
288  PUNTO: '.';
289
290  COMENTARIO: '/#' .*? '#/' -> skip;
291  ESPACIOS: [ \t\r\n]+ -> skip;
292
293  ERROR: .;
```





Explicación de código fase léxica

Palabras Claves Estructurales

```
INICIO: 'inicio';
FIN: 'fin';
SI: 'si';
SINO: 'sino';
MIENTRAS: 'mientras';
PARA: 'para';
RETORNAR: 'retornar';
```

El token INICIO representa el punto de entrada principal de un programa Easy, es decir, donde comienza la ejecución. El token FIN marca el término del programa. Para estructuras condicionales, se utilizan los tokens SI y SINO, los cuales permiten expresar decisiones dentro del flujo del programa. En cuanto a la repetición de instrucciones, Easy incluye MIENTRAS para ciclos tipo *while* y PARA para bucles controlados tipo *for*. La instrucción RETORNAR es utilizada dentro de funciones o métodos para devolver un valor.

Entrada y Salida

```
LEER: 'leer';
IMPRIMIR: 'imprimir';
```

Para el manejo de la entrada/salida, el lenguaje proporciona los tokens LEER, que permite capturar datos desde el usuario o la entrada estándar, e IMPRIMIR, que facilita la salida de resultados en pantalla.

Soporte para programación orientada a objetos y funciones

```
CLASE: 'clase';
SUPER: 'super';
METODO: 'metodo';
FUNCION: 'funcion';
```

El token CLASE permite declarar nuevas clases, mientras que SUPER permite acceder a la clase



padre (herencia). Para la definición de comportamiento dentro de clases, se usa METODO, mientras que FUNCION se utiliza para declarar funciones independientes de una clase.

Estructura de selección múltiple:

```
SELECCIONAR: 'seleccionar';
CASO: 'caso';
DEFECTO: 'defecto';
ROMPER: 'romper';
```

Easy incluye una estructura de control tipo switch a través de la palabra clave SELECCIONAR. Esta se complementa con los tokens CASO, DEFECTO y ROMPER. CASO se usa para definir valores específicos que se evalúan dentro del seleccionar, DEFECTO actúa como una cláusula por omisión y ROMPER termina la ejecución de un caso evitando la ejecución de los siguientes.

Tipos de datos primitivos

```
ENTERO: 'entero';
FLOTANTE: 'flotante';
BOOLEANO: 'booleano';
CADENA: 'cadena';

VERDADERO: 'verdadero';
FALSO: 'falso';
```

Easy reconoce los tipos básicos a través de los tokens ENTERO (para números enteros), FLOTANTE (números con punto decimal), BOOLEANO (valores lógicos) y CADENA (secuencias de caracteres). Los valores booleanos son expresados mediante los tokens VERDADERO y FALSO.





Identificadores y literales

```
ID: [a-zA-Z_][a-zA-Z0-9_]*;  
  
LITENTERO: '-'?[0-9]+;  
LITFLOTANTE: '-'?[0-9]+(\.[0-9]+)([fF])?;  
LITERALCADENA: "" .*? "";
```

Los identificadores (ID) representan nombres válidos para variables, funciones, clases, etc., y deben comenzar con una letra o guion bajo, seguidos de letras, números o guiones bajos. Los valores constantes pueden representarse mediante LITENTERO para números enteros, LITFLOTANTE para valores reales, y LITERALCADENA para textos encerrados entre comillas dobles.

Operadores relacionales y lógicos personalizados

```
IGUALIGUAL: '::';  
MAYORIGUAL: '>:';  
MENORIGUAL: '<:';  
DIFERENTEDE: ':!';  
IGUAL: ':';  
IGUAL_CASO: '=';  
  
Y: '&';  
O: '??';
```

Easy incorpora operadores propios como :: (IGUALIGUAL) para comparar igualdad, >: (MAYORIGUAL) y <: (MENORIGUAL) para comparaciones no estrictas, y :! (DIFERENTEDE) para desigualdad. El operador : (IGUAL) se utiliza para asignación. En estructuras de selección, se emplea = (IGUAL_CASO) para igualar un caso con un valor específico. También se definen operadores lógicos como & (Y) para conjunciones lógicas, y ?? (O) para disyunciones.



Operadores aritméticos y símbolos comunes

```
1  MAYORQUE: '>';
2  MENORQUE: '<';
3
4
5  SUMA: '+';
6  RESTA: '-';
7  MULTIPLICACION: '*';
8  DIVISION: '/';
9
```

Para operaciones matemáticas, se utilizan los tokens + (SUMA), - (RESTA), * (MULTIPLICACION) y / (DIVISION). Las comparaciones simples emplean > (MAYORQUE) y < (MENORQUE).

Símbolos de agrupación y puntuación

```
279
280  LLAVEIZQ: '{';
281  LLAVEDER: '}';
282  PARIZQ: '(';
283  PARDER: ')';
284  CORCHETEIZQ: '[';
285  CORCHETEDER: ']';
286  PUNTOYCOMA: ';';
287  COMA: ',';
288  PUNTO: '.';
289
```

Easy define un conjunto de símbolos que permiten agrupar instrucciones y parámetros. Estos incluyen { y } (LLAVEIZQ, LLAVEDER) para bloques, (y) (PARIZQ, PARDER) para expresiones o llamadas, y [y] (CORCHETEIZQ, CORCHETEDER) para arreglos. Además, se usan ; (PUNTOYCOMA) para finalizar sentencias, , (COMA) como separador de elementos y . (PUNTO) para acceso a atributos o métodos.



Comentarios y espacios en blanco

```
COMENTARIO: '/#' .*? '#/' -> skip;  
ESPACIOS: [ \t\r\n]+ -> skip;
```

Los comentarios en Easy se escriben entre `/#` y `#/`, definidos por el token COMENTARIO. El token ESPACIOS permite ignorar espacios en blanco, saltos de línea o tabulaciones, lo cual es útil para mantener legibilidad sin afectar la sintaxis del programa.

Manejo de errores léxicos

```
ERROR: .;
```

Finalmente, el token ERROR captura cualquier carácter que no coincide con ninguno de los tokens anteriores. Este token permite detectar errores de escritura o símbolos no válidos durante el análisis léxico.

5. Fase Sintáctica

La fase sintáctica de un compilador, también conocida como análisis sintáctico o parsing, tiene como objetivo fundamental verificar que la secuencia de tokens generada por el analizador léxico siga las reglas gramaticales del lenguaje fuente. Esta fase construye una estructura jerárquica, normalmente representada por un árbol de derivación o árbol de sintaxis, que refleja la estructura gramatical del programa.

Para nuestro lenguaje *Easy*, esta fase se implementa utilizando ANTLR, una herramienta poderosa para generar analizadores sintácticos basados en gramáticas definidas en formato EBNF. ANTLR convierte la gramática del lenguaje en código que puede analizar programas escritos en Easy. El proceso se basa en una gramática LL(*) que define las reglas del lenguaje, donde cada regla representa una construcción válida, como declaraciones de variables, estructuras condicionales, bucles, clases, funciones, etc. (Aho, Lam, Sethi, & Ullman, 2006).



Código creado en ANTLR

```
1 grammar Easy;
2
3 bloque_elemento
4     : sentencia
5     | funcion_definicion
6     ;
7
8 programa
9     : (declaracion | funcion_definicion | clase_definicion | declaracion_variable_sentencia)*
10    | INICIO LLAVEIZQ bloque_elemento* LLAVEDER FIN
11    ;
12
13 declaracion
14     : tipo_dato lista_declaraciones PUNTOYCOMA
15     ;
16
17 lista_declaraciones
18     : declaracion_variable (COMA declaracion_variable)*
19     ;
20
21 funcion_definicion
22     : FUNCION tipo_dato ID PARIZQ parametros? PARDER bloque
23     ;
24
25 clase_definicion
26     : CLASE ID (IGUAL ID)? LLAVEIZQ clase_cuerpo* LLAVEDER
27     ;
28
29 clase_cuerpo
30     : atributo
31     | metodo
32     ;
```





```
34  atributo
35      : tipo_dato ID PUNTOYCOMA
36      | tipo_dato ID CORCHETEIZQ LITENTERO CORCHETEDER PUNTOYCOMA
37      ;
38
39  metodo
40      : METODO tipo_dato? ID PARIZQ parametros? PARDER bloque
41      ;
42
43  parametros
44      : parametro (COMA parametro)*
45      ;
46
47  parametro
48      : tipo_dato ID
49      ;
50
51  tipo_dato
52      : ENTERO | FLOTANTE | BOOLEANO | CADENA | ID
53      ;
54
55  sentencia
56      : asignacion
57      | estructura_condicional
58      | estructura_seleccion
59      | estructura_repetitiva
60      | estructura_para
61      | sentencia_imprimir
62      | sentencia_lectura
63      | sentencia_retorno
64      | bloque
```





```
55  sentencia
62      | sentencia_lectura
63      | sentencia_retorno
64      | bloque
65      | llamada_metodo_sentencia
66      | declaracion_variable_sentencia
67      ;
68
69  declaracion_variable_sentencia
70      : tipo_dato lista_declaraciones PUNTOYCOMA
71      ;
72
73 //  FIXED: Allow method calls in variable declarations
74 declaracion_variable
75      : ID
76      | ID IGUAL expresion
77      | ID CORCHETEIZQ LITENTERO CORCHETEDER
78      | ID CORCHETEIZQ LITENTERO CORCHETEDER IGUAL LLAVEIZQ lista_valores LLAVEDER
79      | ID ('[' expresion ']')? (':' expresion)?
80      ;
81
82 declaracion_variable_simple
83      : tipo_dato lista_declaraciones
84      ;
85
86 lista_valores
87      : expresion (COMA expresion)*
88      ;
89
90 asignacion
91      : ID IGUAL expresion PUNTOYCOMA
92      | ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion PUNTOYCOMA
93      | ID TCOMA LLAVEIZQ lista_valores LLAVEDER PUNTOYCOMA
```





```
90  asignacion
91      : ID IGUAL expresion PUNTOYCOMA
92      | ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion PUNTOYCOMA
93      | ID IGUAL LLAVEIZQ lista_valores LLAVEDER PUNTOYCOMA
94      ;
95
96  asignacion_simple
97      : ID IGUAL expresion
98      | ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion
99      | ID IGUAL LLAVEIZQ lista_valores LLAVEDER
100     ;
101
102 estructura_condicional
103     : SI PARIZQ expresion PARDER bloque (SINO bloque)?
104     ;
105
106 estructura_seleccion
107     : SELECCIONAR PARIZQ expresion PARDER LLAVEIZQ caso+ caso_defecto? LLAVEDER
108     ;
109     .
110
111 caso
112     : CASO expresion IGUAL_CASO sentencia* ROMPER PUNTOYCOMA
113     ;
114
115 caso_defecto
116     : DEFECTO IGUAL_CASO sentencia*
117     ;
118
119 estructura_repetitiva
120     : MIENTRAS PARIZQ expresion PARDER bloque
121     ;
```





```
121
122 estructura_para
123     : PARA PARIZQ (asignacion_simple | declaracion_variable_simple) PUNTOYCOMA expresion PUNTOYCOMA
124     ;
125
126 bloque
127     : LLAVEIZQ sentencia* LLAVEDER
128     ;
129
130 sentencia_imprimir
131     : IMPRIMIR PARIZQ expresion PARDER PUNTOYCOMA
132     ;
133
134 sentencia_leitura
135     : LEER PARIZQ ID PARDER PUNTOYCOMA
136     ;
137
138 sentencia_retorno
139     : RETORNAR expresion? PUNTOYCOMA
140     ;
141
142 llamada_metodo
143     : (ID | SUPER) PUNTO ID PARIZQ argumentos? PARDER
144     ;
145
146 llamada_metodo_sentencia
147     : llamada_metodo PUNTOYCOMA
148     ;
149
150 argumentos
151     : expresion (COMA expresion)*
152     ;
```





E > Easy.g4 > expression_primeria

```
154     expression_primeria
155         : LITENTERO
156         | LITFLOTANTE
157         | VERDADERO
158         | FALSO
159         | LITERALCADENA
160         | ID
161         | ID CORCHETEIZQ expresion CORCHETEDER
162         | llamada_funcion
163         | llamada_metodo
164         | PARIZQ expresion PARDER
165         ;
166
167     expresion
168         : expresion_logica
169         | expresion_lista
170         ;
171
172     expresion_lista
173         : LLAVEIZQ lista_valores LLAVEDER
174         ;
175
176     expresion_logica
177         : expresion_relacional
178         | expresion_logica Y expresion_relacional
179         | expresion_logica O expresion_relacional
180         ;
181
182     expresion_relacional
183         : expresion_aritmetica
184         | expresion_aritmetica operador_relacional expresion_aritmetica
185         ;
```





```
180
187 operador_relacional
188     : MAYORQUE
189     | MENORQUE
190     | MAYORIGUAL
191     | MENORIGUAL
192     | IGUALIGUAL
193     | DIFERENTEDE
194     ;
195
196 expresion_aritmetica
197     : termino
198     | expresion_aritmetica SUMA termino
199     | expresion_aritmetica RESTA termino
200     ;
201
202 termino
203     : factor
204     | termino MULTIPLICACION factor
205     | termino DIVISION factor
206     ;
207
208 factor
209     : LITENTERO
210     | LITFLOTANTE
211     | VERDADERO
212     | FALSO
213     | LITERALCADENA
214     | ID
215     | ID CORCHETEIZQ expresion CORCHETEDER
216     | llamada_funcion
217     | llamada_metodo |
218     | PARIZQ expresion PARDER
```





```
206      ;
207
208  factor
209      : LITENTERO
210      | LITFLOTANTE
211      | VERDADERO
212      | FALSO
213      | LITERALCADENA
214      | ID
215      | ID CORCHETEIZQ expresion CORCHETEDER
216      | llamada_funcion
217      | llamada_metodo |
218      | PARIZQ expresion PARDER
219      | RESTA factor
220      ;
221
222  llamada_funcion
223      : ID PARIZQ argumentos? PARDER
224      ;
225
```

Explicación de Reglas Sintácticas

1. bloque_elemento

```
bloque_elemento
  : sentencia
  | funcion_definicion
  ;
```

Define elementos que pueden existir dentro del bloque principal del programa: una sentencia o una función.

2. Programa

```
programa
  : (declaracion | funcion_definicion | clase_definicion | declaracion_variable_sentencia)*
    INICIO LLAVEIZQ bloque_elemento* LLAVEDER FIN
  ;
```





Es la regla inicial. Define la estructura general del programa:

- ✓ Permite múltiples declaraciones, definiciones de funciones o clases antes del bloque principal.
- ✓ Luego exige que el programa comience con inicio { ... } fin, y dentro de ese bloque puede haber más sentencias o funciones.

3. declaracion y lista_declaraciones

```
declaracion
  : tipo_dato lista_declaraciones PUNTOYCOMA
  ;

lista_declaraciones
  : declaracion_variable (COMA declaracion_variable)*
  ;
```

Describe una declaración de variables:

- ✓ declaracion es un tipo de dato seguido por una o más declaraciones.
- ✓ lista_declaraciones permite múltiples variables separadas por comas.

4. funcion_definicion

```
funcion_definicion
  : FUNCION tipo_dato ID PARIZQ parametros? PARDER bloque
  ;
```

Declara una función usando la palabra clave funcion, seguida del tipo de retorno, nombre, parámetros y bloque de código.

5. clase_definicion

```
clase_definicion
  : CLASE ID (IGUAL ID)? LLAVEIZQ clase_cuerpo* LLAVEDER
  ;
```

Define una clase con la palabra clave clase. Puede extender de otra clase (= ID) y contiene atributos o métodos.



6. clase_cuerpo, atributo, método

```
clase_cuerpo
  : atributo
  | metodo
  ;

atributo
  : tipo_dato ID PUNTOYCOMA
  | tipo_dato ID CORCHETEIZQ LITENTERO CORCHETEDER PUNTOYCOMA
  ;

metodo
  : METODO tipo_dato? ID PARIZQ parametros? PARDER bloque
  ;
```

- ✓ clase_cuerpo puede ser un atributo o un método.
- ✓ atributo define una variable de clase (normal o arreglo).
- ✓ metodo define un método, con o sin tipo de retorno.

7. parametros y parámetro

```
parametros
  : parametro (COMA parametro)*
  ;

parametro
  : tipo_dato ID
  ;
```

Define los parámetros de una función o método, separados por comas.

8. tipo_dato

```
tipo_dato
  : ENTERO | FLOTANTE | BOOLEANO | CADENA | ID
  ;
```

Tipos válidos: entero, flotante, booleano, cadena o un identificador personalizado.

En nuestras primeras versiones del Lenguaje solo se podía definir los tipos de Datos Entero,

Flotante e ID, esta actualización ya se incluye el tipo de dato Cadena y Booleano.





9. Sentencia

```
sentencia
: asignacion
| estructura_condicional
| estructura_seleccion
| estructura_repetitiva
| estructura_para
| sentencia_imprimir
| sentencia_leitura
| sentencia_retorno
| bloque
| llamada_metodo
| declaracion_variable_sentencia
;
```

Agrupa todos los tipos de sentencias válidas en el lenguaje: asignaciones, estructuras de control, impresiones, declaraciones, etc.

10. declaracion_variable_sentencia

```
declaracion_variable_sentencia
: tipo_dato lista_declaraciones PUNTOYCOMA
;
```

Versión de declaracion usada dentro de una sentencia (por ejemplo, en el bloque principal).

11. declaracion_variable y declaracion_variable_simple

```
declaracion_variable
: ID
| ID IGUAL expresion
| ID CORCHETEIZQ LITENTERO CORCHETEDER
| ID CORCHETEIZQ LITENTERO CORCHETEDER IGUAL LLAVEIZQ lista_valores LLAVEDER
| ID '[' expresion ']')? (':' expresion)?
;

declaracion_variable_simple
: tipo_dato lista_declaraciones
;
```





Formas válidas de declarar una variable, incluyendo inicialización, arreglos simples o con valores.

12. lista_valores

```
lista_valores
  : expresion (COMA expresion)*
  ;
```

Lista de valores usada para inicializar arreglos o colecciones.

13. asignacion y asignacion_simple

```
asignacion
  : ID IGUAL expresion PUNTOYCOMA
  | ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion PUNTOYCOMA
  | ID IGUAL LLAVEIZQ lista_valores LLAVEDER PUNTOYCOMA
  ;

asignacion_simple
  : ID IGUAL expresion
  | ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion
  | ID IGUAL LLAVEIZQ lista_valores LLAVEDER
  ;
```

Asignación de valores a variables o arreglos. La versión simple no requiere punto y coma (útil en ciclos para).



14. Estructuras de control

```
estructura_condicional
: SI PARIZQ expresion PARDER bloque (SINO bloque)?
;

estructura_seleccion
: SELECCIONAR PARIZQ expresion PARDER LLAVEIZQ caso+ caso_defecto? LLAVEDER
;

caso
: CASO expresion IGUAL_CASO sentencia* ROMPER PUNTOYCOMA
;

caso_defecto
: DEFECTO IGUAL_CASO sentencia*
;

estructura_repetitiva
: MIENTRAS PARIZQ expresion PARDER bloque
;

estructura_para
: 'para' '(' (asignacion_simple | declaracion_variable_simple) ';' expresion ';' asignacion_simple ')' bloque
;
```

- ✓ estructura_condicional: si (...) { ... } sino { ... }.
- ✓ estructura_seleccion: tipo switch con seleccionar (...) { caso ... romper; ... }.
- ✓ estructura_repetitiva: ciclo mientras.
- ✓ estructura_para: ciclo para, con inicialización, condición y actualización.

15. bloque

```
bloque
: LLAVEIZQ sentencia* LLAVEDER
;
```

Un bloque { ... } de sentencias.

16. Sentencias especiales

```
sentencia_imprimir
: IMPRIMIR PARIZQ expresion PARDER PUNTOYCOMA
;

sentencia_leer
: LEER PARIZQ ID PARDER PUNTOYCOMA
;

sentencia_retorno
: RETORNAR expresion? PUNTOYCOMA
;
```



- ✓ sentencia_imprimir: imprime una expresión.
- ✓ sentencia_lectura: lee una variable desde entrada.
- ✓ sentencia_retorno: devuelve un valor desde una función.

17. llamada_metodo

```
llamada_metodo
  : (ID | SUPER) PUNTO ID PARIZQ argumentos? PARDER PUNTOYCOMA
  ;
```

Invoca un método desde un objeto (obj.metodo(...)), o desde super.

18. argumentos

```
argumentos
  : expresion (COMA expresion)*
  ;
```

Lista de expresiones pasadas a una función o método.



19. Expresiones

```
expresion
    : expresion_logica
    | expresion_lista
    ;

expresion_lista
    : LLAVEIZQ lista_valores LLAVEDER
    ;

expresion_logica
    : expresion_relacional
    | expresion_logica Y expresion_relacional
    | expresion_logica O expresion_relacional
    ;

expresion_relacional
    : expresion_aritmetica
    | expresion_aritmetica operador_relacional expresion_aritmetica
    ;

expresion_aritmetica
    : termino
    | expresion_aritmetica SUMA termino
    | expresion_aritmetica RESTA termino
    ;

termino
    : factor
    | termino MULTIPLICACION factor
    | termino DIVISION factor
    ;

factor
    : LITENTERO
    | LITFLOATANTE
    | VERDADERO
    | FALSO
    | LITERALCADENA
    | ID
    | ID CORCHETEIZQ expresion CORCHETEDER
    | llamada_funcion
    | PARIZQ expresion PARDER
    | RESTA factor
    ;
```

- ✓ expresion: puede ser lógica o una lista de valores.
- ✓ expresion_logica: incluye operadores &, ??.
- ✓ expresion_relacional: comparación de expresiones aritméticas.
- ✓ expresion_aritmetica: suma y resta.
- ✓ termino: multiplicación y división.
- ✓ factor: elementos básicos (literales, variables, llamadas, paréntesis).





20. llamada_funcion

```
llamada_funcion
  : ID PARIZQ argumentos? PARDER
  ;
```

Invoca una función directamente por nombre: f(...)

6. Pruebas realizadas en fase sintáctica

Ejemplo 1: Programa Básico de operaciones aritméticas (Suma, resta, multiplicación y división).

```
1  inicio {
2
3    /* ejemplo operaciones aritméticas */
4
5    entero a : 10;
6    entero b : 5;
7
8    entero suma : a + b;
9    entero resta : a - b;
10   entero producto : a * b;
11   entero division : a / b;
12
13   imprimir("Suma:");
14   imprimir(suma);
15
16   imprimir("Resta:");
17   imprimir(resta);
18
19   imprimir("Multiplicación:");
20   imprimir(producto);
21
22   imprimir("División:");
23   imprimir(division);
24 }
25 fin
```

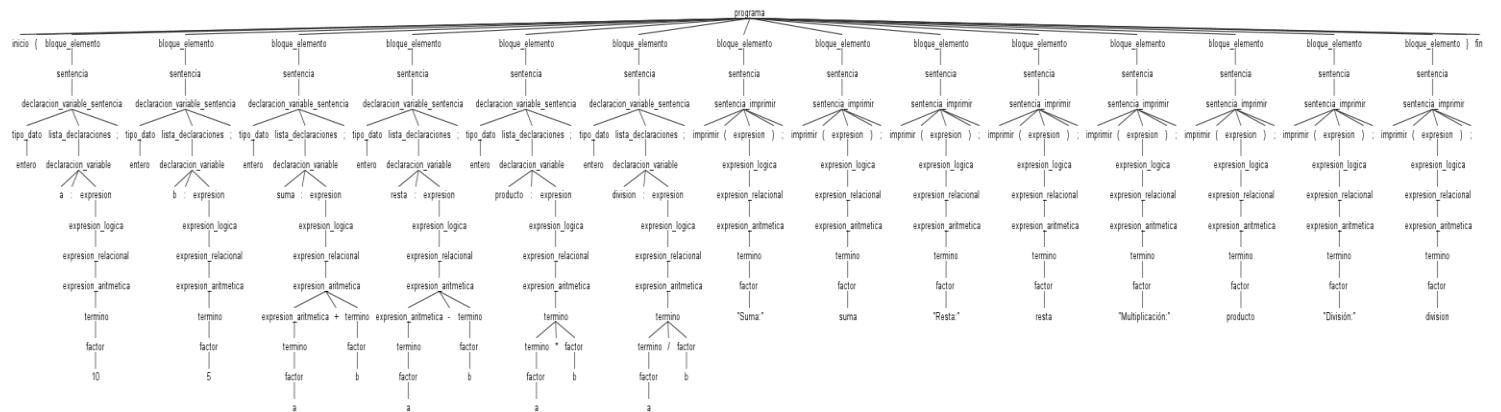
Visualización en consola

```
16   imprimir("Resta:");
17   imprimir(resta);
18
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + × ☰ ...
```

PS C:\antlr> java -cp ".;antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree .\pruebas\Aritmeticas.ec
programa inicio { (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista_declaraciones (declaracion_variabl
e a : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 10))))))) ;))) (bloque_elemento (sentencia (declaracion_vari
able_sentencia (tipo_dato entero)) (lista_declaraciones (declaracion_variable b : (expresion (expresion_logica (expresion
_relacional (expresion_aritmetica (termino (factor 5))))))) ;))) (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato ent
ero)) (lista_declaraciones (declaracion_variable suma : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (expres
ion_aritmetica (termino (factor a)) + (termino (factor b))))))) ;))) (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato ent
ero)) (lista_declaraciones (declaracion_variable resta : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (expres
ion_aritmetica (termino (factor a)) - (termino (factor b))))))) ;))) (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato ent
ero)) (lista_declaraciones (declaracion_variable producto : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (f
actor a)) * (factor b))))))) ;))) (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista
_declaraciones (declaracion_variable division : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (f
actor a)) / (factor b))))))) ;))) (bloque_elemento (sentencia (sentencia_imprimir imprimir ((expresion (expresion_logica (expresion_rel
acional (expresion_aritmetica (termino (factor "Suma:"))))))) ;))) (bloque_elemento (sentencia (sentencia_imprimir imprimir ((expresion (e



Visualización grafica del Árbol Generado



Ejemplo 2: El siguiente programa muestra una estructura básica del ciclo para (for) del lenguaje Easy.

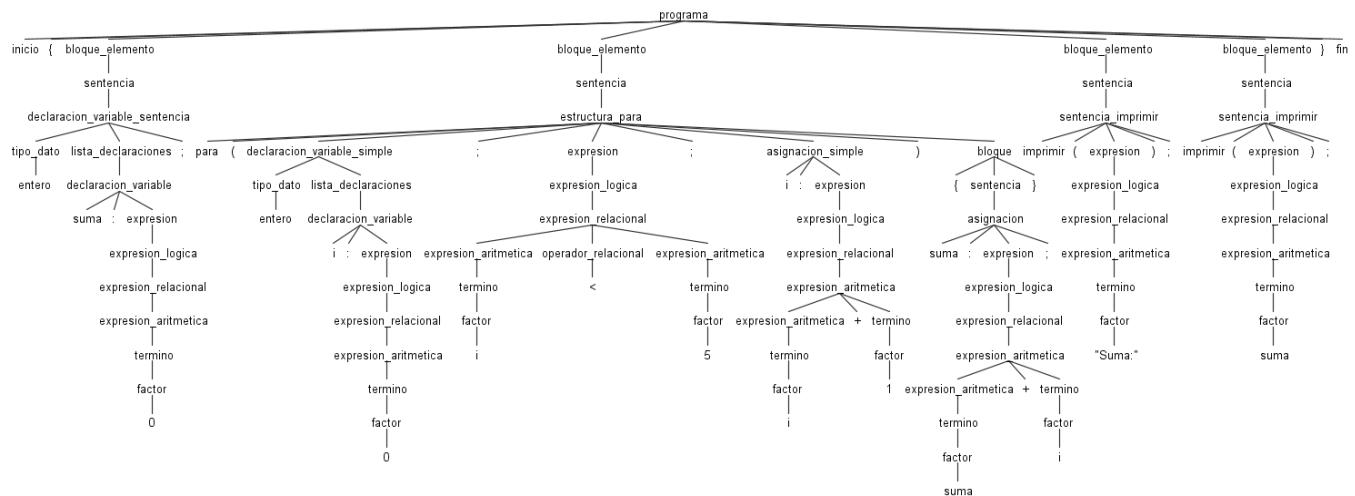
```
pruebas > cicloPara.ec
1  inicio {
2      entero suma : 0;
3
4      para (entero i : 0; i < 5; i : i + 1) {
5          suma : suma + i;
6
7          imprimir("Suma:");
8          imprimir(suma);
9      }
10     }
11     fin
12 }
```

Visualización en consola

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + - 🌐 ...
PS C:\antlr> java -cp "..;antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree .\pruebas\cicloPara.ec
(programa inicio { (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista_declaraciones (declaracion_variab
e suma : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 0))))))) ;))) (bloque_elemento (sentenc
ia (estructura_para para ( (declaracion_variable_simple (tipo_dato entero) (lista_declaraciones (declaracion_variable i : (expresion (exp
resion_logica (expresion_relacional (expresion_aritmetica (termino (factor 0))))))) ; (expresion (expresion_logica (expresion_relacional (e
xpresion_aritmetica (termino (factor 1)))) (operador_relacional <) (expresion_aritmetica (termino (factor 5)))))) ; (asignacion_simple i : (exp
resion (expresion_logica (expresion_relacional (expresion_aritmetica (expresion_aritmetica (termino (factor i)) + (termino (factor 1))))))) (bloque { (sentencia (asignacion suma : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (expresion_aritmetica (termino (factor suma)))) + (termino (factor i))))))) ;)))) (bloque_elemento (sentencia (sentencia_imprimir imprimir ( (expresion (exp
resion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Suma:")))))) ) ;))) (bloque_elemento (sentencia (sentencia_impr
imir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor suma))))))) ) ;))) } fin)
PS C:\antlr>
```



Visualización del Árbol Generado



Dificultades encontradas y como se resolvieron

Al momento de crear la regla no se determinó al inicio si se podría declarar había problemas al momento de la asignación de variables, por tanto, se recurrió a la modificación de la regla de la estructura_for, se agrego la regla de asignacion_simple

asignacion_simple

```

: ID IGUAL expresion
| ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion
| ID IGUAL LLAVEIZQ lista_valores LLAVEDER
;
  
```

estructura_for

```

: 'para' '(' (asignacion | declaracion_variable_simple) ';' expresion ';' 
asignacion ')' bloque
;
  
```



Ejemplo 3: Estructura básica de un ciclo mientras. Este programa incluye un llamado a función

```

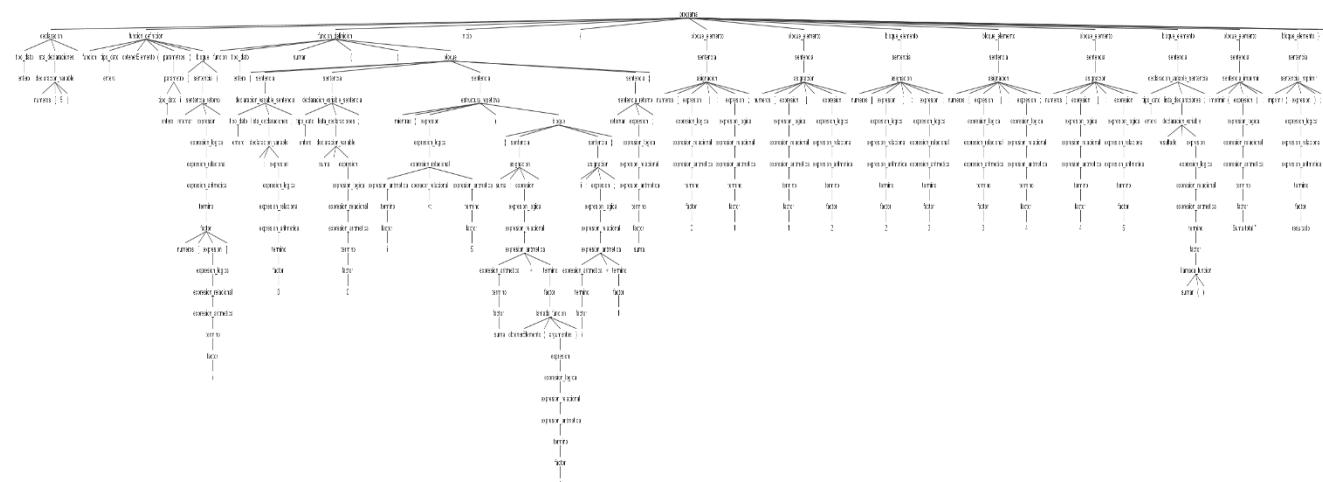
1     entero numeros[5];
2
3     <funcion entero obtenerElemento(entero i) {
4         |     retornar numeros[i];
5     }
6
7     <funcion entero sumar() {
8         entero i : 0;
9         entero suma : 0;
10        mientras (i < 5) {
11            suma : suma + obtenerElemento(i);
12            i : i + 1;
13        }
14        retornar suma;
15    }
16
17    <inicio [
18        numeros[0] : 1;
19        numeros[1] : 2;
20        numeros[2] : 3;
21        numeros[3] : 4;
22        numeros[4] : 5;
23
24        entero resultado : sumar();
25        imprimir("Suma total:");
26        imprimir(resultado);
27    ]
28 fin

```

Visualización en consola



Visualización del Árbol Generado



Dificultades encontradas y como se solucionaron

En este código al ya incluir funciones, aunque la idea principal es probar el ciclo mientras, se determinó usar funciones para probar las nuevas reglas incluidas, una de los problemas que se tuvo es que no se podía declarar una función fuera o dentro del bloque inicio {...} fin. En las pruebas de estrés que se hicieron se determinó crear una nueva regla.

```
funcion_definicion
  : FUNCION tipo_data ID PARIZQ parametros? PARDER bloque
;
```

Y se modificó la regla programa, se agregó una regla superior llamada bloque_elemento (esta se mencionará en el siguiente ejemplo), en programa se permite declarar, definir una función o una clase (se mencionara en el último ejemplo) o una declaración de variables en sentencia

```
bloque_elemento
  : sentencia
  | funcion_definicion
;

programa
  : (declaracion | funcion_definicion | clase_definicion | declaracion_variable_sentencia)*
  | INICIO LLAVEIZQ bloque_elemento* LLAVEDER FIN
;
```



Ejemplo 4: En el siguiente ejemplo se muestra lo que es una función básica a diferencia con el ejemplo anterior es que esta dentro del bloque inicio {...} fin, este fue uno de los retos en este ejemplo ya que solo teníamos en las reglas que se declaran fuera del bloque. También se logró

```
pruebas > ≡ funcion.ec
 1  inicio {
 2      funcion entero multiplicar(entero a, entero b) {
 3          retornar a * b;
 4      }
 5
 6      entero resultado : multiplicar(4, 5);
 7      imprimir("Resultado:");
 8      imprimir(resultado);
 9  }
10  fin
11
```

que la función soportara no recibir parámetros

Visualización en consola

The screenshot shows a terminal window with the following text:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powerShell + × └ ... [x]

PS C:\antlr> java -cp ".;antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree ./pruebas\funcion.ec
● PS C:\antlr> java -cp ".;antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree ./pruebas\funcion.ec
(programa inicio { (bloque_elemento (funcion_definicion funcion (tipo_dato entero) multiplicar ( (parametros (parametro (tipo_dato entero)
a) , (parametro (tipo_dato entero) b)) ) (bloque { (sentencia (sentencia_retorno retornar (expresion (expresion_logica (expresion_relaciona
l (expresion_aritmetica (termino (termino (factor a) * (factor b))))))) ;))) (bloque_elemento (sentencia (declaracion_variable sentenc
ia (tipo_dato entero) (lista_declaraciones (declaracion_variable resultado : (expresion (expresion_logica (expresion_relacional (expresion_a
ritmetica (termino (factor (llamada_funcion multiplicar ( (argumentos (expresion (expresion_logica (expresion_relacional (expresion_arimet
ica (termino (factor 4)))))) , (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 5))))))) ;))))))) ;
(bloque_elemento (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica
(termino (factor "Resultado:"))))) ;))) (bloque_elemento (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresi
on_relacional (expresion_aritmetica (termino (factor resultado))))))) ;))) } fin)
```



Prueba realizada con la función sin recibir parametros



A screenshot of a terminal window showing the output of an ANTLR parse tree for a Java-like program. The code in `funcion.ec` defines a function `multiplicar` that prints the result of multiplying two integers. The terminal shows the command `java -cp ".;antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree .\pruebas\funcion.ec` and the resulting parse tree structure.

```

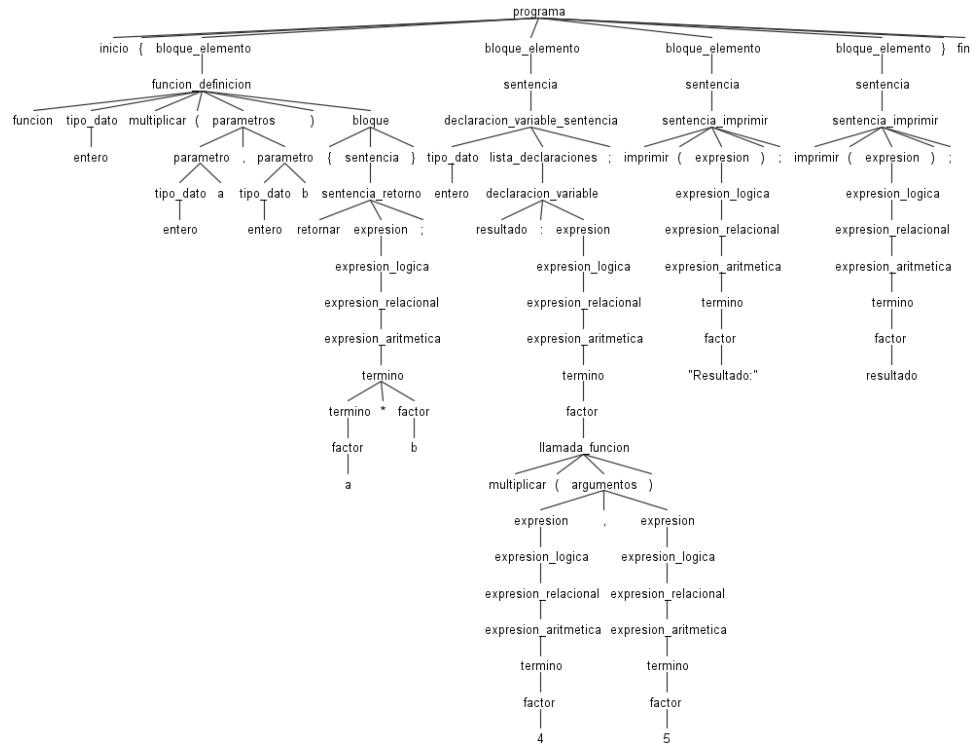
Aritmeticas.ec  funcion.ec x
pruebas > funcion.ec
1  inicio {
2    funcion entero multiplicar() {
3      retornar ;
4    }
5
6    entero resultado : multiplicar(4, 5);
7    imprimir("Resultado:");
8    imprimir(resultado);
9  }
10 fin
11

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
powershell + · · ·

● PS C:\antlr> java -cp ".;antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree .\pruebas\funcion.ec
(programa inicio { (bloque_elemento (funcion_definicion funcion (tipo_dato entero) multiplicar ( ) (bloque { (sentencia (sentencia_retorno
retornar ;)) }))) (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista_declaraciones (declaracion_variable
resultado : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor (llamada_funcion multiplicar (
argumentos (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 4))))), (expresion (expresion_logica (ex-
presión_relacional (expresion_aritmetica (termino (factor 5))))))), ))))))))) ;))) (bloque_elemento (sentencia (sentencia_imprimir imprimir (
(expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Resultado:"))))), (bloque_elemento (se-
ntencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor resultado
))))))) ;))) } fin)
○ PS C:\antlr>

```

Visualización de Árbol Generado





Dificultades encontradas y como se solucionaron

Como se menciono en el ejemplo anterior se creo una nueva regla bloque_elemento que es al que permite que la función se pueda declarar dentro o fuera de un bloque de inicio {...} fin. Y parte de las pruebas de estrés realizadas fue ver si se permitía una función vacía y como se observó en la imagen si es posible

Ejemplo 5: Este programa muestra como funcionan los arreglos en Easy. Declaramos un arreglo e incluimos una función

```
pruebas > E arreglos.ec
 1  inicio {
 2    entero numeros[5] : {1, 2, 3, 4, 5};
 3
 4    funcion entero sumarArreglo() {
 5      entero suma : 0;
 6      entero i : 0;
 7      mientras (i < 5) {
 8        suma : suma + numeros[i];
 9        i : i + 1;
10      }
11      retornar suma;
12    }
13
14    entero resultado : sumarArreglo();
15    imprimir("Suma total:");
16    imprimir(resultado);
17  }
18  fin
19
```

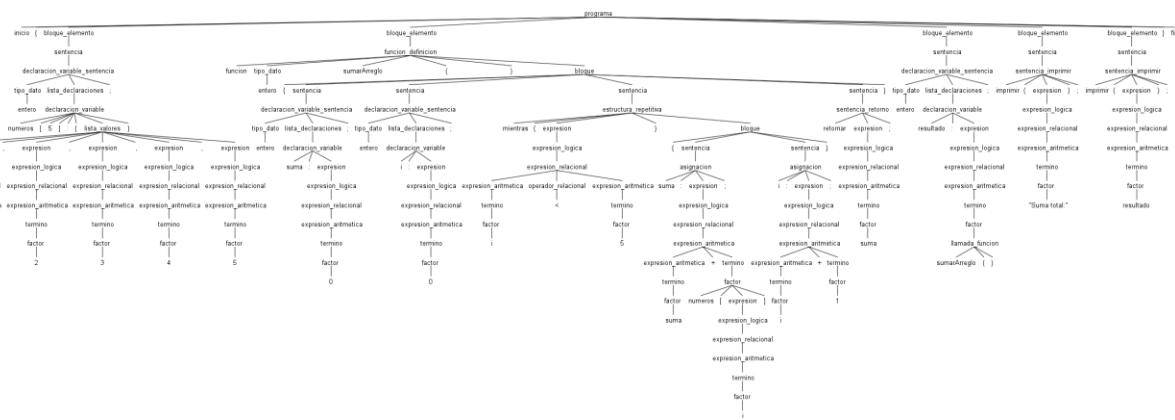
Visualización en Consola

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell + □ ■ ...
```

PS C:\antlr> java -cp ".\antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree .\pruebas\arreglos.ec

(programa inicio { (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista_declaraciones (declaracion_variabl e numeros [5] : { (lista_valores (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 1)))))) , (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 2)))))) , (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 3)))))) , (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 4)))))) , (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 5))))))))) ;))) (bloque_eleme nto (funcion_definicion_funcion (tipo_dato entero) sumarArreglo () (bloque { (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista_declaraciones (declaracion_variable suma : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (facto r 0)))))))) ;)) (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista_declaraciones (declaracion_variable i : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 0)))))))) ;)) (sentencia (estructura_repetitiva mientras ((expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor i)))))) (operador_relacional <) (expresion_aritmetica (termino (factor 5))))))) (bloque { (sentencia (asignacion suma : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (expresion_aritmetica (termino (factor summa)))) + (termino (factor numeros [(expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor i))))))))))) ;))) (sentencia (asignacion i : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (expresion_aritmetica (termino (factor i)))) + (termino (factor 1))))))) ;))) (sentencia (sentencia_retorno retornar (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor summa))))))) ;))) (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo_dato entero) (lista_declaraciones (declaracion_variable resultado : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor (llamada_funcion sumarArreglo ())))))))) ;))) (bloque_elemento (sentencia_imprimir (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Suma total:")))))) ;))) (bloque_elemento (sentencia_imprimir_imprimir (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor resultado))))))) ;))) } fin)

Visualización de Árbol Generado



Dificultades encontradas y como se solucionaron

En las dificultades que tuvimos fue que las reglas que teníamos aun no estaban lógicamente compatibles con la estructura de arreglos que deseábamos realizar, fue necesario aplicar distintas reglas y eliminar ciertas ambigüedades que teníamos.

```

declaracion_variable
: ID
| ID IGUAL expresion
| ID CORCHETEIZQ LITENTERO CORCHETEDER
| ID CORCHETEIZQ LITENTERO CORCHETEDER IGUAL LLAVEIZQ lista_valores LLAVEDER
| ID '[' expresion ']'? (':' expresion)?
;

declaracion_variable_simple
: tipo_dato lista_declaraciones
;

lista_valores
: expresion (COMA expresion)*
;

asignacion
: ID IGUAL expresion PUNTOYCOMA
| ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion PUNTOYCOMA
| ID IGUAL LLAVEIZQ lista_valores LLAVEDER PUNTOYCOMA
;

asignacion_simple
: ID IGUAL expresion
| ID CORCHETEIZQ expresion CORCHETEDER IGUAL expresion
| ID IGUAL LLAVEIZQ lista_valores LLAVEDER
;

```

Creamos una regla de lista_valores que permite inicializar el arreglo, así como en la regla asignacion es importante hacer el llamado de esta nueva regla.



Ejemplo 6: Este ejercicio se logra que Easy ya sea compatible con la Programación Orientada a Objetos (POO), no algo muy avanzado, pero si lo básico ya que es un lenguaje para entorno educativo.

```
pruebas > clases.ec
1  clase Persona {
2    entero edad;
3    cadena nombre;
4
5    metodo establecerDatos(entero e, cadena n) {
6        edad : e;
7        nombre : n;
8    }
9
10   metodo imprimirDatos() {
11       imprimir("Nombre: ");
12       imprimir(nombre);
13       imprimir("Edad: ");
14       imprimir(edad);
15   }
16 }
17
18 clase Estudiante : Persona {
19     flotante promedio;
20
21     metodo establecerPromedio(flotante p) {
22         promedio : p;
23     }
24 }
```

```
4
5     metodo imprimirDatos() {
6         super.imprimirDatos();
7         imprimir("Promedio: ");
8         imprimir(promedio);
9     }
10
11 inicio {
12     Estudiante estudiante1;
13     estudiante1.establecerDatos(20, "Juan");
14     estudiante1.establecerPromedio(88.5);
15     estudiante1.imprimirDatos();
16 }
17 fin
18
```

Como se mencionó al inicio de este informe técnico para la POO se incluyeron varios tokens claves (clase, super, metodo) y esto sumado a las nuevas reglas que permiten funcionen de manera adecuada.

```
clase_definicion
    : CLASE ID (IGUAL_ID)? LLAVEIZQ clase_cuerpo* LLAVEDER
    ;

clase_cuerpo
    : atributo
    | metodo
    ;

atributo
    : tipo_dato ID PUNTOYCOMA
    | tipo_dato ID CORCHETEIZQ LITENTERO CORCHETEDER PUNTOYCOMA
    ;

metodo
    : METODO tipo_dato? ID PARIZQ parametros? PARDER bloque
    ;
```

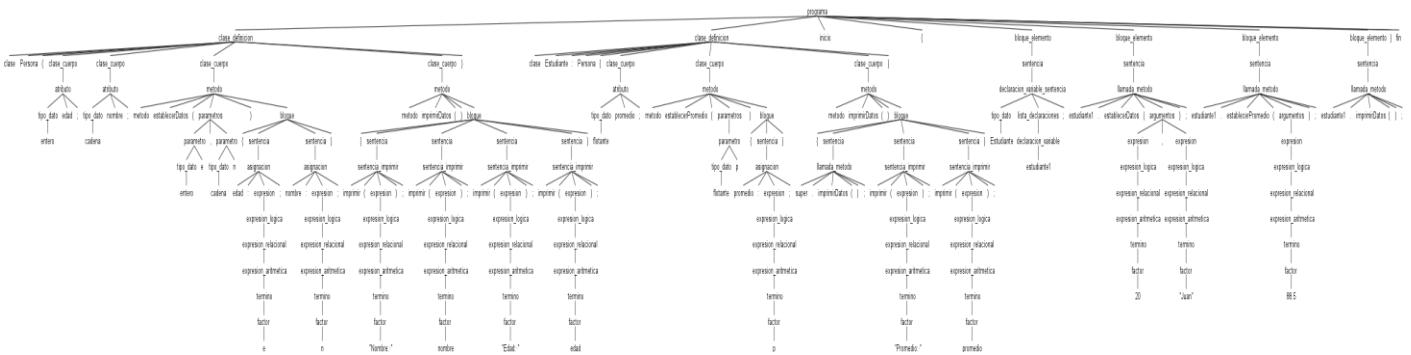
Visualización en consola

```

PS C:\antlr> java -cp ".;antlr-4.13.2-complete.jar" org.antlr.v4.gui.TestRig Easy programa -tree .\pruebas\clases.ec
● (programa (clase_definicion clase Persona { (clase_cuerpo (atributo (tipo dato entero) edad )); (clase_cuerpo (atributo (tipo dato cadena) nombre )); (clase_cuerpo (metodo metodo establecerDatos ( (parametros (parametro (tipo dato entero) e) , (parametro (tipo dato cadena) n)) ) (bloque { (sentencia (asignacion edad : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor e))))))) ;)) (sentencia (asignacion nombre : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor n))))))) ;)) }))) (clase_cuerpo (metodo metodo imprimirDatos ( ) (bloque { (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Nombre"))))) ) ;)) (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor nombre))))))) ) ;)) (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Edad"))))) ) ;)) (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor edad))))))) ) ;)) }))) } (clase_definicion clase Estudiante : Persona { (clase_cuerpo (atributo (tipo dato flotante) promedio )); (clase_cuerpo (metodo metodo establecerPromedio ( (parametros (parametro (tipo dato flotante) p) ) (bloque { (sentencia (asignacion promedio : (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor p))))))) ;)) }))) (clase_cuerpo (metodo metodo imprimirDatos ( ) (bloque { (sentencia (llamada_metodo super . imprimirDatos ( ) );) (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Promedio"))))) ) ;)) (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Edad:"))))) ) ;)) (sentencia (sentencia_imprimir imprimir ( (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor promedio))))))) ) ;)) }))) inicio { (bloque_elemento (sentencia (declaracion_variable_sentencia (tipo dato Estudiante) (lista_declaraciones (declaracion_variable estudiante1) ))));) (bloque_elemento (sentencia (llamada_metodo estudiante1 . establecerDatos ( (argumentos (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 20))))))) , (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor "Juan"))))) ) ;)) (bloque_elemento (sentencia (llamada_metodo estudiante1 . establecerPromedio ( (argumentos (expresion (expresion_logica (expresion_relacional (expresion_aritmetica (termino (factor 88.5))))))) ) ;)) (bloque_elemento (sentencia (llamada_metodo estudiante1 . imprimirDatos ( ) ;)) } fin)
PS C:\antlr> []

```

Visualización de Árbol Generado



Dificultades encontradas y como se solucionaron

Al momento de llamar a los métodos generaba conflictos porque las reglas que teníamos en ese momento eran algo ambiguas por ende se agregaron y modificaron en la parte de la definición de clases y la regla de clase cuerpo.



7. Fase Semántica

La fase semántica de un compilador es responsable de comprobar que el programa no solo sea sintácticamente correcto, sino que también tenga sentido desde el punto de vista lógico y contextual del lenguaje. Esta etapa actúa como un filtro adicional después del análisis sintáctico, asegurándose de que las operaciones, estructuras y tipos usados en el código sean coherentes con las reglas del lenguaje de programación. (Aho, Lam, Sethi, & Ullman, 2006)

En términos concretos, la fase semántica verifica aspectos como:

- **Tipos de datos compatibles:** Asegura que no se sumen cadenas con enteros, ni se asignen valores incorrectos a variables.
- **Declaraciones previas:** Comprueba que todas las variables, funciones y clases hayan sido declaradas antes de usarse.
- **Ámbitos o scopes:** Controla que los identificadores no entren en conflicto entre diferentes bloques o clases.
- **Número y tipo de parámetros:** Verifica que las llamadas a funciones y métodos coincidan con sus definiciones.
- **Reglas de herencia y sobreescritura:** En lenguajes con orientación a objetos, asegura la correcta aplicación del polimorfismo y jerarquía de clases.

En el caso de *Easy*, esta fase semántica es esencial para reforzar la robustez del lenguaje, especialmente por la inclusión de funciones, clases, arreglos, herencia, métodos y sobrecarga. Si bien ANTLR se encarga del análisis sintáctico, el análisis semántico generalmente se implementa manualmente sobre el árbol de análisis (parse tree), usando *listeners* o *visitors*, o como parte del generador de código en etapas posteriores.

Generación de Código Intermedio

La generación de código intermedio es una etapa crucial en el proceso de compilación, donde se traduce el árbol de sintaxis abstracta (AST) o el árbol de análisis en una representación intermedia (IR) más cercana al lenguaje máquina, pero aún independiente de la arquitectura específica del hardware. Esta fase actúa como un puente entre el análisis semántico y la generación de código final, permitiendo una mayor portabilidad, optimización y control sobre la traducción.

La representación intermedia tiene las siguientes características:

- **Es estructurada y simple**, facilitando el análisis y las transformaciones posteriores.



- Es más abstracta que el lenguaje máquina, pero más concreta que el código fuente.
- Puede representarse en forma de árboles, tres direcciones (three-address code), pilas, cuádruplas, etc.

Objetivos de esta fase:

1. **Portabilidad:** El uso de una IR permite generar código para diferentes plataformas sin modificar el front-end del compilador.
2. **Facilidad de optimización:** Muchas optimizaciones se realizan mejor sobre la IR que sobre el código fuente o ensamblador.
3. **Separación de responsabilidades:** Permite dividir claramente la lógica del lenguaje fuente de la lógica de la plataforma de destino.

En el lenguaje *Easy*:

Durante esta etapa, los elementos del lenguaje como expresiones aritméticas, estructuras condicionales, ciclos, llamadas a funciones, acceso a atributos y métodos de clases, y operaciones con arreglos se traducen a un código intermedio en C++ (en tu caso), que luego será compilado por un compilador C++ real.

Por ejemplo:

- Una instrucción como $a = b + c$; podría generarse como `a = b + c;` en C++, manteniendo la semántica, pero adaptada a la sintaxis del lenguaje de destino.
- Una estructura `si (x < 10) { imprimir("Hola"); }` puede transformarse en un bloque `if (x < 10) { std::cout << "Hola"; }`.

La generación de código intermedio se implementa generalmente con el patrón Visitor, donde cada nodo del árbol de análisis se recorre y se genera la porción correspondiente del código destino.

Esta etapa es la base para construir un traductor o compilador funcional y es donde se concreta la “traducción” real de *Easy* hacia C++, asegurando que el comportamiento del programa original se preserve correctamente.



8. Código Main.java

```
1 import org.antlr.v4.runtime.*;
2 import org.antlr.v4.runtime.tree.*;
3 import java.io.*;
4
5 public class Main {
6
7     Run | Debug
8     public static void main(String[] args) throws Exception {
9         if (args.length != 1) {
10             System.out.println("Uso: java Main <archivo.ec>");
11             return;
12         }
13
14         try {                                CharStreams cannot be resolved Java(570425394)
15             // Leer archivo de           Ver el problema (F8) Corrección Rápida (⌘.)
16             CharStream input = CharStreams.fromFileName(args[0]);
17
18             // Crear lexer y parser
19             EasyLexer lexer = new EasyLexer(input);
20             CommonTokenStream tokens = new CommonTokenStream(lexer);
21             EasyParser parser = new EasyParser(tokens);
22
23             // Obtener el árbol
24             ParseTree tree = parser.program();
25
26             // Verificar si hay errores de sintaxis
27             if (parser.getNumberOfSyntaxErrors() > 0) {
28                 System.out.println("Error: El archivo tiene errores de sintaxis.");
29                 return;
30             }
31
32             // Generar código
33             CodeGenerator generator = new CodeGenerator();
34             String output = generator.visit(tree);
35
36             // Escribir código C++ generado
37             String outputFile = args[0].replace(target:".ec", replacement:".cpp");
38             try (PrintWriter writer = new PrintWriter(outputFile)) {
39                 writer.println(output);
40
41                 System.out.println("Código C++ generado en " + outputFile);
42
43             } catch (Exception e) {
44                 System.out.println("Error: Verificar que el archivo " + args[0] + " existe y tiene la sintaxis correcta.");
45                 e.printStackTrace();
46             }
47         }
48     }
}
```

Flujo de trabajo en el proceso de generación de código intermedio:

- Se lee un archivo fuente escrito en el lenguaje EasyC, que contiene instrucciones declaradas utilizando la sintaxis definida por nuestro lenguaje ficticio.
- El archivo es procesado por ANTLR 4, que realiza el análisis léxico y sintáctico, reconociendo los tokens y estructuras gramaticales definidas en la gramática .g4.
- ANTLR genera automáticamente un árbol de análisis sintáctico (parse tree), que representa la estructura jerárquica del código EasyC.





- Mediante un componente denominado CodeGenerator (implementado como un Visitor en Java), se recorre el árbol generado y se transforma cada nodo en su equivalente en código C++.
- Finalmente, el resultado de esta conversión se escribe en un archivo de salida llamado **salida.cpp**, el cual contiene el código intermedio que puede ser compilado con cualquier compilador estándar de C++.

Explicación del código:

```
1 import org.antlr.v4.runtime.*;
2 import org.antlr.v4.runtime.tree.*;
3 import java.io.*;
```

Las líneas de importación mostradas en el código Java permiten acceder a las bibliotecas necesarias para el funcionamiento del compilador EasyC.

- En primer lugar, *import org.antlr.v4.runtime.**; e *import org.antlr.v4.runtime.tree.**; incorporan todas las clases del entorno de ejecución de ANTLR 4, necesarias para realizar el análisis léxico, sintáctico y para trabajar con árboles de análisis (parse trees). Estas clases permiten leer el código fuente, generar tokens, construir el árbol sintáctico y recorrerlo mediante Listeners o Visitors personalizados.
- Por otro lado, *import java.io.**; incluye las clases estándar de Java para operaciones de entrada y salida, lo cual es fundamental para poder leer archivos escritos en EasyC y escribir el código intermedio resultante (en este caso, en C++) en un archivo de salida. En conjunto, estas tres instrucciones preparan el entorno para analizar el código fuente y generar el archivo con la traducción intermedia.

```
5 public class Main {
6
7     Run | Debug
8     public static void main(String[] args) throws Exception {
9         if (args.length != 1) {
10             System.err.println("Uso: java Main <archivo.ec>");
11             return;
12     }
13 }
```

Este fragmento define el método principal (main) del programa, encargado de iniciar la ejecución del compilador. Verifica que se haya recibido exactamente un argumento (el archivo fuente en



EasyC). Si no es así, muestra un mensaje de error indicando el uso correcto del comando y termina la ejecución.

```
13     try {
14         // Leer archivo de entrada
15         CharStream input = CharStreams.fromFileName(args[0]);
16
17         // Crear lexer y parser
18         EasyLexer lexer = new EasyLexer(input);
19         CommonTokenStream tokens = new CommonTokenStream(lexer);
20         EasyParser parser = new EasyParser(tokens);
21
22         // Obtener el árbol
23         ParseTree tree = parser.programa();
24
25         // Verificar si hay errores de sintaxis
26         if (parser.getNumberOfSyntaxErrors() > 0) {
27             System.out.println("Error: El archivo tiene errores de sintaxis.");
28             return;
29     }
```

Este fragmento corresponde a la sección del programa donde se lleva a cabo el análisis sintáctico del archivo fuente escrito en EasyC. Todo el proceso está contenido dentro de un bloque try para capturar posibles excepciones relacionadas con la lectura del archivo o el análisis del contenido.

- **Lectura del archivo de entrada:** Se utiliza `CharStreams.fromFileName(args[0])` para leer el archivo cuyo nombre fue proporcionado como argumento al ejecutar el programa. El contenido se almacena en un objeto CharStream, que es el formato que ANTLR utiliza para representar flujos de caracteres de entrada.
- **Creación del analizador léxico y sintáctico:**
 1. Se instancia EasyLexer, que tokeniza el contenido del archivo (es decir, identifica palabras clave, identificadores, símbolos, etc.).
 2. Luego se crea un CommonTokenStream que organiza y almacena los tokens generados por el lexer.
 3. A partir de ese flujo de tokens, se construye el EasyParser, que es el responsable de analizar la estructura gramatical del programa.
- **Generación del árbol de análisis sintáctico (Parse Tree):** El método `parser.programa()` ejecuta la regla inicial de la gramática (llamada `programa`), y como resultado se construye un árbol sintáctico (ParseTree) que representa la jerarquía estructural del código fuente.
- **Verificación de errores de sintaxis:** Se llama a `parser.getNumberOfSyntaxErrors()` para comprobar si ocurrieron errores durante el análisis sintáctico. Si se detectan errores, se



muestra un mensaje de advertencia en la consola y se interrumpe la ejecución del compilador para evitar procesar código inválido.

```
31     // Generar código
32     CodeGenerator generator = new CodeGenerator();
33     String output = generator.visit(tree);
34
35     // Escribir código C++ generado
36     String outputFile = args[0].replace(target:".ec", replacement:".cpp");
37     try (PrintWriter writer = new PrintWriter(outputFile)) {
38         writer.println(output);
39     }
```

Este fragmento corresponde a la **etapa final del proceso de compilación**, donde se genera y guarda el **código intermedio en C++** a partir del árbol sintáctico obtenido previamente.

- **Generación del código intermedio:** Se crea una instancia de la clase CodeGenerator, que implementa el patrón Visitor personalizado. Esta clase recorre el ParseTree generado por ANTLR y construye una cadena (String output) que contiene el equivalente en código C++ del programa escrito originalmente en EasyC.
- **Construcción del nombre del archivo de salida:** Se toma el nombre del archivo de entrada (pasado como argumento al programa) y se reemplaza la extensión .ec por .cpp. Esto garantiza que el archivo resultante tenga el formato adecuado para ser compilado por un compilador de C++.
- **Escritura del archivo de salida:** Mediante un bloque try-with-resources, se crea un objeto PrintWriter vinculado al archivo de salida. Dentro del bloque, se escribe el contenido del código C++ generado (output) en ese archivo.

```
41     System.out.println("Código C++ generado en " + outputFile);
42
43     } catch (Exception e) {
44         System.out.println("Error: Verificar que el archivo " + args[0] + " existe y tiene la sintaxis correcta.");
45         e.printStackTrace();
46     }
47 }
48 }
```

Este bloque corresponde al **final del proceso de ejecución del compilador** y está relacionado con el manejo de errores y la confirmación de éxito.

- **Mensaje de confirmación:** Si todo el proceso de análisis y generación se ejecuta correctamente, se imprime un mensaje en consola indicando que el código C++ fue generado exitosamente, junto con el nombre del archivo de salida (outputFile).



- **Bloque catch para manejo de excepciones:** Si ocurre alguna excepción durante la lectura del archivo, el análisis sintáctico o la escritura del archivo de salida, se captura con catch. En ese caso, el programa imprime un mensaje de error indicando que se debe verificar si el archivo de entrada existe y si su contenido es sintácticamente válido. Además, se muestra la traza del error con `e.printStackTrace()` para facilitar la depuración.





III. CodeGenerator.java

```
1 import java.util.*;
2 import org.antlr.v4.runtime.tree.ParseTree;
3 import org.antlr.v4.runtime.tree.TerminalNode;
4
5 public class CodeGenerator extends EasyBaseVisitor<String> {
6     private final List<String> code = new ArrayList<>();
7     private int indentLevel = 0;
8     private Set<String> declaredVariables = new HashSet<>();
9     private Map<String, String> variableTypes = new HashMap<>();
10    private boolean inClass = false;
11    private boolean inPrivateSection = false;
12
13    private void addLine(String line) {
14        if (indentLevel == 0) {
15            code.add(line);
16        } else {
17            String indent = " ".repeat(indentLevel);
18            code.add(indent + line);
19        }
20    }
21
22    private String mapType(String easyType) {
23        switch (easyType) {
24            case "entero": return "int";
25            case "flotante": return "float";
26            case "booleano": return "bool";
27            case "cadena": return "string";
28            default: return easyType; // Para tipos de clase
29        }
30    }
31
32    @Override
33    public String visitPrograma(EasyParser.ProgramaContext ctx) {
34        addLine(line:"#include <iostream>");
35        addLine(line:"#include <string>");
36        addLine(line:"using namespace std;");
37        addLine(line:"");
```



```

38
39     List<String> classes = new ArrayList<>();
40     List<String> functions = new ArrayList<>();
41     List<String> globalVars = new ArrayList<>();
42     List<String> mainStatements = new ArrayList<>();
43
44     // PRIMERA PASADA: Procesar elementos antes del token "inicio"
45     for (int i = 0; i < ctx.getChildCount(); i++) {
46         ParseTree child = ctx.getChild(i);
47
48         if (child instanceof TerminalNode && "inicio".equals(child.getText())) {
49             break;
50         }
51
52         // Procesar clases primero
53         if (child instanceof EasyParser.Clase_definicionContext) {
54             String classCode = generateClass((EasyParser.Clase_definicionContext) child);
55             classes.add(classCode);
56         }
57         // Luego funciones
58         else if (child instanceof EasyParser.Funcion_definicionContext) {
59             String functionCode = generateFunction((EasyParser.Funcion_definicionContext) child);
60             functions.add(functionCode);
61         }
62         // Variables globales
63         else if (child instanceof EasyParser.DeclaracionContext) {
64             String globalVar = generateGlobalDeclaration((EasyParser.DeclaracionContext) child);
65             if (globalVar != null && !globalVar.isEmpty()) {
66                 globalVars.add(globalVar);
67             }
68         }
69         else if (child instanceof EasyParser.Declaracion_variable_sentenciaContext) {
70             String globalVar = generateVariableDeclaration((EasyParser.Declaracion_variable_sentenciaContext) child);
71             globalVars.add(globalVar);
72         }
73     }
74
75     // Agregar clases PRIMERO
76     for (String classCode : classes) {
77         code.add(classCode);
78     }
79
80     // Luego variables globales
81     for (String globalVar : globalVars) {
82         addLine(globalVar);
83     }
84
85     if (!globalVars.isEmpty()) {
86         addLine(line:"");
87     }
88
89     // Luego funciones
90     for (String func : functions) {
91         code.add(func);
92     }
93
94     // SEGUNDA PASADA: Procesar el contenido del bloque principal
95     if (ctx.bloque_elemento() != null) {
96         for (EasyParser.Bloque_elementoContext elemento : ctx.bloque_elemento()) {
97             extractFromBloqueElemento(elemento, new ArrayList<>(), new ArrayList<>(), mainStatements);
98         }
99     }
100
101    // Finalmente el main
102    addLine(line:"int main() {");
103    indentLevel = 1;
104
105    // Agregar statements del main
106    for (String stmt : mainStatements) {
107        addLine(stmt);
108    }

```



```

109     addLine(line:"");
110     addLine(line:"return 0;");
111     indentLevel = 0;
112     addLine(line:"}");
113
114     return String.join(delimiter:"\n", code);
115 }
116
117
118 // Nuevo método para generar clases como string
119 private String generateClass(EasyParser.Clase_definicionContext ctx) {
120     StringBuilder classBuilder = new StringBuilder();
121     String className = ctx.ID(i:0).getText();
122
123     classBuilder.append(str:"class ").append(className);
124
125     // Verificar herencia
126     if (ctx.ID().size() > 1) {
127         String parentClass = ctx.ID(i:1).getText();
128         classBuilder.append(str:" : public ").append(parentClass);
129     }
130
131     classBuilder.append(str:" {\n");
132
133     // Sección privada por defecto
134     classBuilder.append(str:"private:\n");
135
136     // Procesar atributos (van en la sección privada)
137     if (ctx.clase_cuerpo() != null) {
138         for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
139             if (cuerpo.atributo() != null) {
140                 String atributo = generateAttribute(cuerpo.atributo());
141                 classBuilder.append(str:"     ").append(atributo).append(str:"\n");
142             }
143         }
144     }
145
146     // Sección pública
147     classBuilder.append(str:"\npublic:\n");
148
149     // Procesar métodos (van en la sección pública)
150     if (ctx.clase_cuerpo() != null) {
151         for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
152             if (cuerpo.metodo() != null) {
153                 String metodo = generateMethod(cuerpo.metodo());
154                 classBuilder.append(metodo);
155             }
156         }
157     }
158
159     classBuilder.append(str:"};\n\n");
160
161     return classBuilder.toString();
162 }
163
164 // Método auxiliar para generar atributos
165 private String generateAttribute(EasyParser.AtributoContext ctx) {
166     String type = mapType(ctx.tipo_dato().getText());
167     String name = ctx.ID().getText();
168
169     if (ctx.CORCHETEIZO() != null) {
170         String size = ctx.LITENTERO().getText();
171         return type + " " + name + "[" + size + "]";
172     } else {
173         return type + " " + name + ";";
174     }
175 }
176
177 // Método auxiliar para generar métodos
178 private String generateMethod(EasyParser.MetodoContext ctx) {
179     String returnType = ctx.tipo_dato() != null ? mapType(ctx.tipo_dato().getText()) : "void";
180     String methodName = ctx.ID().getText();

```

```

181
182     String params = "";
183     if (ctx.parametros() != null) {
184         params = generateParameters(ctx.parametros());
185     }
186
187     StringBuilder method = new StringBuilder();
188     method.append(str: "    ").append(returnType).append(str: " ").append(methodName).append(str: "(").append(params).append(str: ")").append(str: "\n");
189
190     // Procesar el bloque del método
191     String body = generateBlockContent(ctx.bloque());
192     method.append(body);
193
194     method.append(str: "    }\n\n");
195     return method.toString();
196
197
198     ' Método auxiliar para generar parámetros
199     private String generateParameters(EasyParser.ParametrosContext ctx) {
200         List<String> params = new ArrayList<>();
201         for (EasyParser.ParametroContext param : ctx.parametro()) {
202             String type = mapType(param.tipo_dato().getText());
203             String name = param.ID().getText();
204             params.add(type + " " + name);
205         }
206         return String.join(delimiter:", ", params);
207
208
209     ' CORRECCIÓN: Método mejorado para generar declaraciones globales
210     private String generateGlobalDeclaration(EasyParser.DeclaracionContext ctx) {
211         // Manejar el caso específico de arrays con inicialización
212         String texto = ctx.getText();
213         System.out.println("!!! PROCESANDO DECLARACIÓN GLOBAL: " + texto);
214
215         // Caso: entero numeros[5]:{1,2,3,4,5};
216         if (texto.matches(regex:"entero\\w+\\[\\d+\\]:\\{.*\\};")) {
217             // Extraer partes usando regex
218             String processed = texto.replaceAll(regex:"entero(\\w+)\\[(\\d+)\\]:\\{([^\r\n]+)\\};", replacement:"int $1[$2] = ${3};");
219             System.out.println("!!! ARRAY CON INICIALIZACIÓN: " + processed);
220             return processed;
221         }
222         // Caso: entero numeros[5]; (sin inicialización)
223         else if (texto.matches(regex:"entero\\w+\\[\\d+\\];")) {
224             String processed = texto.replaceAll(regex:"entero(\\w+)\\[(\\d+)\\];", replacement:"int $1[$2];");
225             System.out.println("!!! ARRAY SIN INICIALIZACIÓN: " + processed);
226             return processed;
227         }
228
229         return null;
230
231
232     private void extractFromBloqueElemento(EasyParser.Bloque_elementoContext elemento,
233                                         List<String> mainFunctions,
234                                         List<String> globalVars,
235                                         List<String> mainStatements) {
236         if (elemento.sentencia() != null) {
237             extractFromSentencia(elemento.sentencia(), mainFunctions, globalVars, mainStatements);
238         } else if (elemento.funcion_definicion() != null) {
239             // Función a nivel de bloque_elemento - estas van antes del main
240             String func = generateFunction(elemento.funcion_definicion());
241             mainFunctions.add(func);
242         }
243
244
245     private void extractFromSentencia(EasyParser.SentenciaContext sentencia,
246                                         List<String> mainFunctions,
247                                         List<String> globalVars,
248                                         List<String> mainStatements) {
249         if (sentencia.declaracion_variable_sentencia() != null) {
250             String varDecl = generateVariableDeclaration(sentencia.declaracion_variable_sentencia());
251
252             if (varDecl.contains(s:["[")) || isUsedByFunctions(sentencia.declaracion_variable_sentencia())) {
253                 globalVars.add(varDecl);
254             } else {
255                 mainStatements.add(varDecl);
256             }
257         }

```

```

257     } else if (sentencia.bloque() != null) {
258         if (sentencia.bloque().sentencia() != null) {
259             for (EasyParser.SentenciaContext subSentencia : sentencia.bloque().sentencia()) {
260                 if (containsFunctionDefinition(subSentencia)) {
261                     String func = extractFunctionFromSentencia(subSentencia);
262                     if (func != null) {
263                         mainFunctions.add(func);
264                     }
265                 } else {
266                     extractFromSentencia(subSentencia, mainFunctions, globalVars, mainStatements);
267                 }
268             }
269         }
270     } else if (sentencia.estructura_condicional() != null) {
271         String ifStmt = generateIfStatement(sentencia.estructura_condicional());
272         mainStatements.add(ifStmt);
273     } else if (sentencia.estructura_seleccion() != null) {
274         String switchStmt = generateSwitchStatement(sentencia.estructura_seleccion());
275         mainStatements.add(switchStmt);
276     } else if (sentencia.estructura_repetitiva() != null) {
277         String whileStmt = generateWhileStatement(sentencia.estructura_repetitiva());
278         mainStatements.add(whileStmt);
279     } else if (sentencia.estructura_para() != null) {
280         String forStmt = generateForStatement(sentencia.estructura_para());
281         mainStatements.add(forStmt);
282     } else if (sentencia.asignacion() != null) {
283         String assignment = generateAssignment(sentencia.asignacion());
284         mainStatements.add(assignment);
285     } else if (sentencia.sentencia_imprimir() != null) {
286         String printStmt = generatePrintStatement(sentencia.sentencia_imprimir());
287         mainStatements.add(printStmt);
288     } else if (sentencia.sentencia_lecatura() != null) {
289         String readStmt = generateReadStatement(sentencia.sentencia_lecatura());
290         mainStatements.add(readStmt);
291     } else if (sentencia.llamada_metodo_sentencia() != null) {
292         String methodCall = generateMethodCall(sentencia.llamada_metodo_sentencia().llamada_metodo());
293         mainStatements.add(methodCall);
294     }
295 }

```

```

296
297 private boolean isUsedByFunctions(EasyParser.Declaracion_variable_sentenciaContext ctx) {
298     for (EasyParser.Declaracion_variableContext decl : ctx.lista_declaraciones().declaracion_variable()) {
299         String name = decl.ID().getText();
300
301         // Si es un arreglo, probablemente lo use una función
302         if (decl.CORCHETEIZQ() != null) {
303             return true;
304         }
305
306         // Variables específicas que sabemos que usan las funciones
307         if (name.equals(anObject:"numeros") || name.equals(anObject:"datos") || name.equals(anObject:"valores")) {
308             return true;
309         }
310     }
311     return false;
312 }
313
314 private boolean containsFunctionDefinition(EasyParser.SentenciaContext sentencia) {
315     return sentencia.getText().contains("funcion");
316 }
317
318 private String extractFunctionFromSentencia(EasyParser.SentenciaContext sentencia) {
319     // Para arreglos.ec, funcion.ec, etc. - funciones dentro del bloque
320     String text = sentencia.getText();
321
322     if (text.contains(s:"sumarArreglo")) {
323         return "int sumarArreglo() {\n" +
324             "    int suma = 0;\n" +
325             "    int i = 0;\n" +
326             "    while ((i < 5)) {\n" +
327             "        suma = (suma + numeros[i]);\n" +
328             "        i = (i + 1);\n" +
329             "    }\n" +
330             "    return suma;\n" +
331             "}";
332 }

```

```

332     } else if (text.contains(s:"multiplicar")) {
333         return "int multiplicar(int a, int b) {\n" +
334             "    return (a * b);\n" +
335             "}";
336     }
337
338     return null;
339 }
340
341 private String generateMethodCall(EasyParser.Llamada_metodoContext ctx) {
342     // Obtener el objeto (puede ser ID o SUPER)
343     String object;
344     if (ctx.SUPER() != null) {
345         object = "super";
346     } else {
347         // Tomar el primer ID (el objeto)
348         object = ctx.ID(i:0).getText();
349     }
350
351     // El método es el segundo ID (si existe)
352     String method = ctx.ID().size() > 1 ? ctx.ID(i:1).getText() : "";
353
354     // Procesar argumentos
355     String args = "";
356     if (ctx.argumentos() != null) {
357         List<String> argList = new ArrayList<>();
358         for (EasyParser.ExpresionContext expr : ctx.argumentos().expresion()) {
359             argList.add(visit(expr));
360         }
361         args = String.join(delimiter:", ", argList);
362     }
363
364     return object + "." + method + "(" + args + ")";
365 }
366
367 private String generateFunctionCall(EasyParser.Llamada_funcionContext ctx) {
368     String functionName = ctx.ID().getText();
369

```

```

370     String args = "";
371     if (ctx.argumentos() != null) {
372         List<String> argList = new ArrayList<>();
373         for (EasyParser.ExpresionContext expr : ctx.argumentos().expresion()) {
374             argList.add(visit(expr));
375         }
376         args = String.join(delimiter:", ", argList);
377     }
378
379     return functionName + "(" + args + ")";
380 }
381
382 private String generateFunction(EasyParser.Funcion_definicionContext ctx) {
383     String returnType = mapType(ctx.tipo_dato().getText());
384     String functionName = ctx.ID().getText();
385
386     String params = "";
387     if (ctx.parametros() != null) {
388         params = generateParameters(ctx.parametros());
389     }
390
391     StringBuilder func = new StringBuilder();
392     func.append(returnType).append(str:" ").append(functionName).append(str:"(").append(params).append(str:") {\\n");
393
394     // Procesar el bloque de la función
395     String body = generateBlockContent(ctx.bloque());
396     func.append(body);
397
398     func.append(str:"}\\n\\n");
399     return func.toString();
400 }
401
402 private String generateBlockContent(EasyParser.BloqueContext ctx) {
403     StringBuilder content = new StringBuilder();
404     if (ctx.sentencia() != null) {
405         for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
406             content.append(str:"      ").append(generateSentenciaContent(stmt)).append(str:"\\n");
407         }
408     }
409

```



```

409     return content.toString();
410 }
411
412 private String generateForStatement(EasyParser.Estructura_paraContext ctx) {
413     StringBuilder result = new StringBuilder();
414
415     String initialization = "";
416     String condition = "";
417     String increment = "";
418
419     // Misma lógica que arriba
420     if (ctx.declaracion_variable_simple() != null) {
421         initialization = generateDeclaracionVariableSimple(ctx.declaracion_variable_simple());
422         if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() > 0) {
423             increment = generateAsignacionSimple(ctx.asignacion_simple(0));
424         }
425     } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() >= 2) {
426         initialization = generateAsignacionSimple(ctx.asignacion_simple(0));
427         increment = generateAsignacionSimple(ctx.asignacion_simple(1));
428     } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() == 1) {
429         increment = generateAsignacionSimple(ctx.asignacion_simple(0));
430     }
431
432     if (ctx.expresion() != null) {
433         condition = visit(ctx.expresion());
434     }
435
436     result.append(str:"for (").append(initialization).append(str:" ")
437         .append(condition).append(str:" ")
438         .append(increment).append(str:")\n");
439
440     String body = generateBlockContent(ctx.bloque());
441     result.append(body);
442     result.append(str:")");
443
444     return result.toString();
445 }
446
447 private String generateSentenciaContent(EasyParser.SentenciaContext sentencia) {
448     if (sentencia.declaracion_variable_sentencia() != null) {
449         return generateVariableDeclaration(sentencia.declaracion_variable_sentencia());
450     } else if (sentencia.asignacion() != null) {
451         return generateAssignment(sentencia.asignacion());
452     } else if (sentencia.estructura_repetitiva() != null) {
453         return generateWhileStatement(sentencia.estructura_repetitiva());
454     } else if (sentencia.estructura_condicional() != null) {
455         return generateIfStatement(sentencia.estructura_condicional());
456     } else if (sentencia.estructura_seleccion() != null) {
457         return generateSwitchStatement(sentencia.estructura_seleccion());
458     }
459     else if (sentencia.estructura_para() != null) {
460         return generateForStatement(sentencia.estructura_para());
461     } else if (sentencia.sentencia_retorno() != null) {
462         return generateReturnStatement(sentencia.sentencia_retorno());
463     } else if (sentencia.sentencia_imprimir() != null) {
464         return generatePrintStatement(sentencia.sentencia_imprimir());
465     } else if (sentencia.llamada_metodo_sentencia() != null) {
466         return generateMethodCall(sentencia.llamada_metodo_sentencia().llamada_metodo());
467     }
468     return "";
469 }
470
471 // CORRECCIÓN: Método mejorado para generar declaraciones de variables
472 private String generateVariableDeclaration(EasyParser.Declaracion_variable_sentenciaContext ctx) {
473     String type = mapType(ctx.tipo_dato().getText());
474     StringBuilder result = new StringBuilder();
475
476     for (EasyParser.Declaracion_variableContext decl : ctx.lista_declaraciones().declaracion_variable()) {
477         String name = decl.ID().getText();
478
479         if (decl.CORCHETEIZQ() != null) {
480             String size = decl.LITENTERO().getText();
481             if (decl.IGUAL() != null && decl.lista_valores() != null) {
482                 String values = visit(decl.lista_valores());
483                 // CORRECCIÓN: Cambiar : por = en inicialización de arrays

```



```

484         result.append(type).append(str:" ").append(name).append(str:["").append(size).append(str:] = "").append(values).append(str:");
485     } else {
486         result.append(type).append(str:" ").append(name).append(str:["").append(size).append(str:"]);;
487     }
488 } else if (decl.expresion() != null && decl.IGUAL() != null) {
489     String value = visit(decl.expresion()).get(index:0);
490     // CORRECCIÓN: Cambiar : por = en asignaciones
491     result.append(type).append(str:" ").append(name).append(str:" = ").append(value).append(str:";");
492 } else {
493     result.append(type).append(str:" ").append(name).append(str:";");
494 }
495 }
496
497 return result.toString();
498 }
499
500 // CORRECCIÓN: Método mejorado para generar asignaciones
501 private String generateAssignment(EasyParser.AsignacionContext ctx) {
502     String variable = ctx.ID().getText();
503
504     if (ctx.CORCHETEIZO() != null) {
505         String index = visit(ctx.expresion(i:0));
506         String value = visit(ctx.expresion(i:1));
507         // CORRECCIÓN: Usar = en lugar de :
508         return variable + "[" + index + "] = " + value + ";";
509     } else {
510         String value = visit(ctx.expresion(i:0));
511         // CORRECCIÓN: Usar = en lugar de :
512         return variable + " = " + value + ";";
513     }
514 }
515
516 private String generateWhileStatement(EasyParser.Estructura_repetitivaContext ctx) {
517     String condition = visit(ctx.expresion());
518     StringBuilder result = new StringBuilder();
519     result.append(str:"while ()").append(condition).append(str:) {\n");
520
521     String body = generateBlockContent(ctx.bloque());
522
523     result.append(body);
524     result.append(str:");");
525
526     return result.toString();
527 }
528
529 // Método para generar la estructura switch-case
530 private String generateSwitchStatement(EasyParser.Estructura_seleccionContext ctx) {
531     String switchExpression = visit(ctx.expresion());
532     StringBuilder result = new StringBuilder();
533
534     result.append(str:"switch ()").append(switchExpression).append(str:) {\n");
535
536     // Procesar todos los casos
537     if (ctx.caso() != null) {
538         for (EasyParser.CasoContext caso : ctx.caso()) {
539             String caseCode = generateCase(caso);
540             result.append(caseCode);
541         }
542     }
543
544     // Procesar caso default si existe
545     if (ctx.caso_defecto() != null) {
546         String defaultCase = generateDefaultCase(ctx.caso_defecto());
547         result.append(defaultCase);
548     }
549
550     result.append(str:");");
551     return result.toString();
552 }
553
554 // Método para generar un caso individual
555 private String generateCase(EasyParser.CasoContext ctx) {
556     StringBuilder result = new StringBuilder();
557     String caseValue = visit(ctx.expresion());
558
559     result.append(str:"      case ").append(caseValue).append(str:":\n");

```

```

560     // Procesar las sentencias del caso
561     if (ctx.sentencia() != null) {
562         for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
563             String stmtCode = generateSentenciaContent(stmt);
564             result.append(str:"").append(stmtCode).append(str:"\n");
565         }
566     }
567
568     result.append(str:"      break;\n");
569     return result.toString();
570 }
571
572 // Método para generar el caso default
573 private String generateDefaultCase(EasyParser.Caso_defectoContext ctx) {
574     StringBuilder result = new StringBuilder();
575
576     result.append(str:"      default:\n");
577
578     // Procesar las sentencias del caso default
579     if (ctx.sentencia() != null) {
580         for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
581             String stmtCode = generateSentenciaContent(stmt);
582             result.append(str:"").append(stmtCode).append(str:"\n");
583         }
584     }
585
586     return result.toString();
587 }
588
589 private String generateIfStatement(EasyParser.Estructura_condicionalContext ctx) {
590     String condition = visit(ctx.expresion());
591     StringBuilder result = new StringBuilder();
592     result.append(str:"if (").append(condition).append(str:")\n");
593
594     String body = generateBlockContent(ctx.bloque(i:0));
595     result.append(body);
596
597     if (ctx.bloque().size() > 1) {
598         result.append(str:"} else {\n");
599         String elseBody = generateBlockContent(ctx.bloque(i:1));
600         result.append(elseBody);
601     }
602
603     result.append(str:"}");
604     return result.toString();
605 }
606
607 private String generateReturnStatement(EasyParser.Sentencia_retornoContext ctx) {
608     if (ctx.expresion() != null) {
609         String value = visit(ctx.expresion());
610         return "return " + value + ";";
611     } else {
612         return "return;";
613     }
614 }
615
616 private String generatePrintStatement(EasyParser.Sentencia_imprimirContext ctx) {
617     String value = visit(ctx.expresion());
618     return "cout << " + value + " << endl;";
619 }
620
621 private String generateReadStatement(EasyParser.Sentencia_lecturaContext ctx) {
622     String variable = ctx.ID().getText();
623     return "cin >> " + variable + ";";
624 }
625
626 // MÉTODOS VISITOR ORIGINALES (para compatibilidad)
627 @Override
628 public String visitClase_definicion(EasyParser.Clase_definicionContext ctx) {
629     String className = ctx.ID(i:0).getText();
630     inClass = true;
631
632     addLine("class " + className);
633

```



```
634     // Verificar herencia
635     if (ctx.ID().size() > 1) {
636         String parentClass = ctx.ID(i:1).getText();
637         code.set(code.size() - 1, "class " + className + " : public " + parentClass);
638     }
639
640     addLine(line:"{");
641
642     // Sección privada por defecto
643     addLine(line:"private:");
644     indentLevel++;
645     inPrivateSection = true;
646
647     // Procesar atributos (van en la sección privada)
648     if (ctx.clase_cuerpo() != null) {
649         for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
650             if (cuerpo.atributo() != null) {
651                 visit(cuerpo.atributo());
652             }
653         }
654     }
655
656     // Sección pública
657     addLine(line:"\"");
658     indentLevel--;
659     addLine(line:"public:");
660     indentLevel++;
661     inPrivateSection = false;
662
663     // Procesar métodos (van en la sección pública)
664     if (ctx.clase_cuerpo() != null) {
665         for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
666             if (cuerpo.metodo() != null) {
667                 visit(cuerpo.metodo());
668             }
669         }
670     }
671 }
```





```
672     indentLevel--;
673     addLine(line:"};");
674     addLine(line:"");
675 
676     inClass = false;
677     return null;
678 }
679
680
681     @Override
682 public String visitEstructura_seleccion(EasyParser.Estructura_seleccionContext ctx) {
683     String switchExpression = visit(ctx.expresion());
684     addLine("switch (" + switchExpression + "){");
685 
686     // Procesar todos los casos
687     if (ctx.caso() != null) {
688         for (EasyParser.CasoContext caso : ctx.caso()) {
689             visit(caso);
690         }
691     }
692 
693     // Procesar caso default si existe
694     if (ctx.caso_defecto() != null) {
695         visit(ctx.caso_defecto());
696     }
697 
698     addLine(line:"}");
699     return null;
700 }
701
702     // Visitor para caso individual
703     // @Override
704     public String visitCase(EasyParser.CasoContext ctx) {
705         String caseValue = visit(ctx.expresion());
706         addLine("case " + caseValue + ":");

707         indentLevel++;
708 }

710     // Procesar las sentencias del caso
711     if (ctx.sentencia() != null) {
712         for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
713             visit(stmt);
714         }
715     }
716 
717     addLine(line:"break;");
718     indentLevel--;
719 
720     return null;
721 }

722     // Visitor para caso default
723     @Override
724     public String visitCaso_defecto(EasyParser.Caso_defectoContext ctx) {
725         addLine(line:"default:");

726         indentLevel++;

727         // Procesar las sentencias del caso default
728         if (ctx.sentencia() != null) {
729             for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
730                 visit(stmt);
731             }
732         }
733 
734         indentLevel--;
735 
736         return null;
737 }

738     @Override
739     public String visitAtributo(EasyParser.AtributoContext ctx) {
740         String type = mapType(ctx.tipo_dato().getText());
741         String name = ctx.ID().getText();
742 
743         if (ctx.CORCHETEIZQ() != null) {
```



```

748     String size = ctx.LITENTERO().getText();
749     addLine(type + " " + name + "[" + size + "]");
750 } else {
751     addLine(type + " " + name + ";");
752 }
753 return null;
754 }

755 @Override
756 public String visitMetodo(EasyParser.MetodoContext ctx) {
757     String returnType = ctx.tipo_dato() != null ? mapType(ctx.tipo_dato().getText()) : "void";
758     String methodName = ctx.ID().getText();

759     String params = "";
760     if (ctx.parametros() != null) {
761         params = visit(ctx.parametros());
762     }

763     addLine(returnType + " " + methodName + "(" + params + ") {" );
764     indentLevel++;

765     visit(ctx.bloque());
766
767     indentLevel--;
768     addLine(line:"}");
769     addLine(line:"");
770     return null;
771 }

772 @Override
773 public String visitFuncion_definicion(EasyParser.Fucion_definicionContext ctx) {
774     String returnType = mapType(ctx.tipo_dato().getText());
775     String functionName = ctx.ID().getText();

776     String params = "";
777     if (ctx.parametros() != null) {
778         params = visit(ctx.parametros());
779     }

780     addLine(returnType + " " + functionName + "(" + params + ") {" );
781     indentLevel++;

782     visit(ctx.bloque());
783
784     indentLevel--;
785     addLine(line:"}");
786     addLine(line:"");
787     return null;
788 }

789 @Override
790 public String visitParametros(EasyParser.ParametrosContext ctx) {
791     List<String> params = new ArrayList<>();
792     for (EasyParser.ParametroContext param : ctx.parametro()) {
793         params.add(visit(param));
794     }
795     return String.join(delimiter:", ", params);
796 }

797 @Override
798 public String visitParametro(EasyParser.ParametroContext ctx) {
799     String type = mapType(ctx.tipo_dato().getText());
800     String name = ctx.ID().getText();
801     return type + " " + name;
802 }

803 // CORRECCIÓN: Método mejorado para manejar declaraciones globales
804 @Override
805 public String visitDeclaracion(EasyParser.DeclaracionContext ctx) {
806     // Recorrer todos los hijos y buscar declaraciones de variables
807     for (int i = 0; i < ctx.getChildCount(); i++) {
808         ParseTree child = ctx.getChild(i);

809         // Si encuentra una Declaracion_variable_sentenciaContext, procesarla
810         if (child instanceof EasyParser.Declaracion_variable_sentenciaContext) {
811             visitDeclaracion_variable_sentencia((EasyParser.Declaracion_variable_sentenciaContext) child);
812         }
813     }
814 }

```





```
862     private void visitDeclaracionVariable(EasyParser.Declaracion_variableContext ctx, String type) {
863         String name = ctx.ID().getText();
864         declaredVariables.add(name);
865         variableTypes.put(name, type);
866
867         if (ctx.CORCHETEIZQ() != null) {
868             String size = ctx.LITENTERO().getText();
869             if (ctx.IGUAL() != null && ctx.lista_valores() != null) {
870                 String values = visit(ctx.lista_valores());
871                 // CORRECCIÓN: Usar = en lugar de :
872                 addLine(type + " " + name + "[" + size + "] = " + values + ";");
873             } else {
874                 addLine(type + " " + name + "[" + size + "];");
875             }
876         } else if (ctx.expresion() != null && ctx.IGUAL() != null) {
877             String value = visit(ctx.expresion().get(index:0));
878             // CORRECCIÓN: Usar = en lugar de :
879             addLine(type + " " + name + " = " + value + ";");
880         } else {
881             addLine(type + " " + name + ";");
882         }
883     }
884
885     @Override
886     public String visitLista_valores(EasyParser.Lista_valoresContext ctx) {
887         List<String> values = new ArrayList<>();
888         for (EasyParser.ExpresionContext expr : ctx.expresion()) {
889             values.add(visit(expr));
890         }
891         return "{" + String.join(delimiter:", ", values) + "}";
892     }
893
894     @Override
895     public String visitBloque(EasyParser.BloqueContext ctx) {
896         if (ctx.sentencia() != null) {
897             for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
898                 visit(stmt);
899             }
900         }
901     }
902 }
```



```

900     }
901     return null;
902 }
903
904 @Override
905 public String visitBloque_elemento(EasyParser.Bloque_elementoContext ctx) {
906     if (ctx.sentencia() != null) {
907         visit(ctx.sentencia());
908     } else if (ctx.funcion_definicion() != null) {
909         visit(ctx.funcion_definicion());
910     }
911     return null;
912 }
913
914 @Override
915 public String visitAsignacion(EasyParser.AsignacionContext ctx) {
916     String variable = ctx.ID().getText();
917
918     if (ctx.CORCHETEIZQ() != null) {
919         String index = visit(ctx.expresion(i:0));
920         String value = visit(ctx.expresion(i:1));
921         // CORRECCIÓN: Usar = en lugar de :
922         addLine(variable + "[" + index + "] = " + value + ";");
923     } else if (ctx.lista_valores() != null) {
924         String values = visit(ctx.lista_valores());
925         // CORRECCIÓN: Usar = en lugar de :
926         addLine(variable + " = " + values + ";");
927     } else {
928         String value = visit(ctx.expresion(i:0));
929         // CORRECCIÓN: Usar = en lugar de :
930         addLine(variable + " = " + value + ";");
931     }
932     return null;
933 }
934
935 @Override
936 public String visitEstructura_condicional(EasyParser.Estructura_condicionalContext ctx) {
937     String condition = visit(ctx.expresion());
938
939     addLine("if (" + condition + ") {");
940
941     indentLevel++;
942     visit(ctx.bloque(i:0));
943     indentLevel--;
944
945     if (ctx.bloque().size() > 1) {
946         addLine("} else {");
947         indentLevel++;
948         visit(ctx.bloque(i:1));
949         indentLevel--;
950     }
951
952     addLine("}");
953     return null;
954 }
955
956 @Override
957 public String visitEstructura_para(EasyParser.Estructura_paraContext ctx) {
958     // Debug: Imprimir información del contexto
959     System.out.println("x:== DEBUG ESTRUCTURA_PARA ===");
960     System.out.println("Texto completo: " + ctx.getText());
961     System.out.println("Número de asignacion_simple: " + (ctx.asignacion_simple() != null ? ctx.asignacion_simple().size() : 0));
962     System.out.println("¿Tiene declaracion_variable_simple? " + (ctx.declaracion_variable_simple() != null));
963     System.out.println("Expresión: " + (ctx.expresion() != null ? ctx.expresion().getText() : "null"));
964
965     String initialization = "";
966     String condition = "";
967     String increment = "";
968
969     // Inicialización: puede ser declaración de variable o primera asignación
970     if (ctx.declaracion_variable_simple() != null) {
971         // Caso: para (entero i : 0; ...)
972         initialization = generateDeclaracionVariableSimple(ctx.declaracion_variable_simple());
973         System.out.println("Inicialización (declaración): " + initialization);
974
975         // El incremento será la primera (y posiblemente única) asignacion_simple

```



```

976     if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() > 0) {
977         increment = generateAsignacionSimple(ctx.asignacion_simple(1:0));
978         System.out.println("Incremento: " + increment);
979     }
980 } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() >= 2) {
981     // Caso: para (i : 0; i < 5; i : i + 1) - dos asignaciones
982     initialization = generateAsignacionSimple(ctx.asignacion_simple(i:0));
983     increment = generateAsignacionSimple(ctx.asignacion_simple(i:1));
984     System.out.println("Inicialización (asignación): " + initialization);
985     System.out.println("Incremento: " + increment);
986 } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() == 1) {
987     // Solo una asignación - probablemente el incremento
988     increment = generateAsignacionSimple(ctx.asignacion_simple(i:0));
989     System.out.println("Solo incremento: " + increment);
990 }
991
992 // Condición
993 if (ctx.expresion() != null) {
994     condition = visit(ctx.expresion());
995     System.out.println("Condición: " + condition);
996 }
997
998 // Generar el for en C++
999 String forStatement = "for (" + initialization + "; " + condition + "; " + increment + ")";
1000 System.out.println("FOR generado: " + forStatement);
1001
1002 addLine(forStatement);
1003 indentLevel++;
1004
1005 // Procesar el bloque del for
1006 if (ctx.bloque() != null) {
1007     visit(ctx.bloque());
1008 }
1009
1010 indentLevel--;
1011 addLine(line:"}");
1012
1013 return null;
1014 }

1015
1016
1017 // Método auxiliar para generar asignación simple
1018 private String generateAsignacionSimple(EasyParser.Asignacion_simpleContext ctx) {
1019     String variable = ctx.ID().getText();
1020
1021     if (ctx.CORCHETEIZQ() != null) {
1022         String index = visit(ctx.expresion(i:0));
1023         String value = visit(ctx.expresion(i:1));
1024         // CORRECCIÓN: Usar = en lugar de :
1025         return variable + "[" + index + "] = " + value;
1026     } else if (ctx.lista_valores() != null) {
1027         String values = visit(ctx.lista_valores());
1028         // CORRECCIÓN: Usar = en lugar de :
1029         return variable + " = " + values;
1030     } else {
1031         String value = visit(ctx.expresion(i:0));
1032         // CORRECCIÓN: Usar = en lugar de :
1033         return variable + " = " + value;
1034     }
1035 }

1036
1037 // Método auxiliar para generar declaración de variable simple
1038 private String generateDeclaracionVariableSimple(EasyParser.Declaracion_variable_simpleContext ctx) {
1039     String type = mapType(ctx.tipo_dato().getText());
1040     StringBuilder result = new StringBuilder();
1041
1042     for (EasyParser.Declaracion_variableContext decl : ctx.lista_declaraciones().declaracion_variable()) {
1043         String name = decl.ID().getText();
1044         declaredVariables.add(name);
1045         variableTypes.put(name, type);
1046
1047         if (decl.IGUAL() != null && decl.expresion() != null) {
1048             String value = visit(decl.expresion().get(index:0));
1049             // CORRECCIÓN: Usar = en lugar de :
1050             result.append(type).append(str:" ").append(name).append(str:" = ").append(value);
1051         } else {
1052             result.append(type).append(str:" ").append(name);
1053         }
1054     }

```



```

1054     }
1055 
1056     return result.toString();
1057 }
1058 
1059 @Override
1060 public String visitEstructura_repetitiva(EasyParser.Estructura_repetitivaContext ctx) {
1061     String condition = visit(ctx.expresion());
1062     addLine("while (" + condition + ") {");
1063 
1064     indentLevel++;
1065     visit(ctx.bloque());
1066     indentLevel--;
1067 
1068     addLine(line:"}");
1069     return null;
1070 }
1071 
1072 @Override
1073 public String visitSentencia_imprimir(EasyParser.Sentencia_imprimirContext ctx) {
1074     String value = visit(ctx.expresion());
1075     addLine("cout << " + value + " << endl;");
1076     return null;
1077 }
1078 
1079 @Override
1080 public String visitSentencia_lecatura(EasyParser.Sentencia_lecaturaContext ctx) {
1081     String variable = ctx.ID().getText();
1082     addLine("cin >> " + variable + ";");
1083     return null;
1084 }
1085 
1086 @Override
1087 public String visitSentencia_retorno(EasyParser.Sentencia_retornoContext ctx) {
1088     if (ctx.expresion() != null) {
1089         String value = visit(ctx.expresion());
1090         addLine("return " + value + ";");
1091     } else {
1092         addLine(line:"return;");
1093     }
1094     return null;
1095 }
1096 
1097 @Override
1098 public String visitLlamada_funcion(EasyParser.Llamada_funcionContext ctx) {
1099     String functionName = ctx.ID().getText();
1100     String args = "";
1101     if (ctx.argumentos() != null) {
1102         args = visit(ctx.argumentos());
1103     }
1104     return functionName + "(" + args + ")";
1105 }
1106 
1107 @Override
1108 public String visitArgumentos(EasyParser.ArgumentosContext ctx) {
1109     List<String> args = new ArrayList<>();
1110     for (EasyParser.ExpresionContext expr : ctx.expresion()) {
1111         args.add(visit(expr));
1112     }
1113     return String.join(delimiter:", ", args);
1114 }
1115 
1116 @Override
1117 public String visitExpresion(EasyParser.ExpresionContext ctx) {
1118     if (ctx.expresion_logica() != null) {
1119         return visit(ctx.expresion_logica());
1120     } else if (ctx.expresion_lista() != null) {
1121         return visit(ctx.expresion_lista());
1122     }
1123     return "";
1124 }
1125 
1126 @Override
1127 public String visitExpresion_lista(EasyParser.Expresion_listaContext ctx) {
1128     return visit(ctx.lista_valores());
1129 }

```



```

1130
1131     @Override
1132     public String visitExpresion_logica(EasyParser.Expresion_logicaContext ctx) {
1133         if (ctx.getChildCount() == 1) {
1134             return visit(ctx.expresion_relacional());
1135         } else {
1136             String left = visit(ctx.expresion_logica());
1137             String operator = ctx.children.get(1).getText();
1138             String right = visit(ctx.expresion_relacional());
1139
1140             String cppOperator = operator.equals(anObject:"&") ? "&&" : "||";
1141             return "(" + left + " " + cppOperator + " " + right + ")";
1142         }
1143     }
1144
1145     @Override
1146     public String visitExpresion_relacional(EasyParser.Expresion_relacionalContext ctx) {
1147         if (ctx.getChildCount() == 1) {
1148             return visit(ctx.expresion_aritmetica(i:0));
1149         } else {
1150             String left = visit(ctx.expresion_aritmetica(i:0));
1151             String operator = visit(ctx.operador_relacional());
1152             String right = visit(ctx.expresion_aritmetica(i:1));
1153             return "(" + left + " " + operator + " " + right + ")";
1154         }
1155     }
1156
1157     @Override
1158     public String visitOperador_relacional(EasyParser.Operador_relacionalContext ctx) {
1159         String op = ctx.getText();
1160         switch (op) {
1161             case "==": return "==";
1162             case ">=": return ">=";
1163             case "<=": return "<=";
1164             case "!=": return "!=";
1165             default: return op;
1166         }
1167     }

```



```

1168
1169     @Override
1170     public String visitExpresion_aritmetica(EasyParser.Expresion_aritmeticaContext ctx) {
1171         if (ctx.getChildCount() == 1) {
1172             return visit(ctx.termino());
1173         } else {
1174             String left = visit(ctx.expresion_aritmetica());
1175             String operator = ctx.children.get(1).getText();
1176             String right = visit(ctx.termino());
1177             return "(" + left + " " + operator + " " + right + ")";
1178         }
1179     }
1180
1181     @Override
1182     public String visitTermino(EasyParser.TerminoContext ctx) {
1183         if (ctx.getChildCount() == 1) {
1184             return visit(ctx.factor());
1185         } else {
1186             String left = visit(ctx.termino());
1187             String operator = ctx.children.get(1).getText();
1188             String right = visit(ctx.factor());
1189             return "(" + left + " " + operator + " " + right + ")";
1190         }
1191     }
1192
1193     @Override
1194     public String visitFactor(EasyParser.FactorContext ctx) {
1195         if (ctx.llamada_funcion() != null) {
1196             return generateFunctionCall(ctx.llamada_funcion()).replace(target:";", replacement:"");
1197         }
1198         if (ctx.llamada_metodo() != null) {
1199             return generateMethodCall(ctx.llamada_metodo()).replace(target:";", replacement:"");
1200         }
1201         if (ctx.LITENTERO() != null) {
1202             return ctx.LITENTERO().getText();
1203         } else if (ctx.LITFLOATANTE() != null) {
1204             return ctx.LITFLOATANTE().getText();
1205         } else if (ctx.VERDADERO() != null) {
1206             return "true";
1207         } else if (ctx.FALSO() != null) {
1208             return "false";
1209         } else if (ctx.LITERALCADENA() != null) {
1210             return ctx.LITERALCADENA().getText();
1211         } else if (ctx.ID() != null) {
1212             if (ctx.CORCHETEIZQ() != null) {
1213                 String index = visit(ctx.expresion());
1214                 return ctx.ID().getText() + "[" + index + "]";
1215             } else {
1216                 return ctx.ID().getText();
1217             }
1218         } else if (ctx.llamada_funcion() != null) {
1219             return visit(ctx.llamada_funcion());
1220         } else if (ctx.PARIZQ() != null) {
1221             return "(" + visit(ctx.expresion()) + ")";
1222         } else if (ctx.RESTA() != null) {
1223             return "-" + visit(ctx.factor());
1224         }
1225     }
1226 }
1227 }
```

Explicación del código:

```

1 import java.util.*;
2 import org.antlr.v4.runtime.tree.ParseTree;
3 import org.antlr.v4.runtime.TerminalNode;
```

Esta parte del código realiza las importaciones necesarias para implementar el generador de código





intermedio.

1. *import java.util.*;*

Importa todas las clases del paquete `java.util`, como listas, mapas, y otras estructuras de datos que podrían ser utilizadas en la clase `CodeGenerator`.

2. *import org.antlr.v4.runtime.tree.ParseTree;*

Importa la clase `ParseTree`, que representa el árbol de análisis sintáctico generado por ANTLR a partir del código fuente EasyC.

3. *import org.antlr.v4.runtime.tree.TerminalNode;*

Importa `TerminalNode`, que representa los nodos hoja del árbol (es decir, los tokens individuales reconocidos por el lexer).

```
5  public class CodeGenerator extends EasyBaseVisitor<String> {
6      private final List<String> code = new ArrayList<>();
7      private int indentLevel = 0;
8      private Set<String> declaredVariables = new HashSet<>();
9      private Map<String, String> variableTypes = new HashMap<>();
10     private boolean inClass = false;
11     private boolean inPrivateSection = false;
12
13     private void addLine(String line) {
14         if (indentLevel == 0) {
15             code.add(line);
16         } else {
17             String indent = "    ".repeat(indentLevel);
18             code.add(indent + line);
19         }
20     }
21
22     private String mapType(String easyType) {
23         switch (easyType) {
24             case "entero": return "int";
25             case "flotante": return "float";
26             case "booleano": return "bool";
27             case "cadena": return "string";
28             default: return easyType; // Para tipos de clase
29         }
30     }
31 }
```

La clase `CodeGenerator` extiende de `EasyBaseVisitor<String>`, lo que permite recorrer el árbol de análisis sintáctico generado por ANTLR y construir el código intermedio como una cadena de texto.

Se declaran atributos privados que gestionan la generación del código:



- code: una lista donde se almacenan las líneas del código generado.
- indentLevel: controla el nivel de indentación.
- declaredVariables y variableTypes: almacenan las variables declaradas y sus tipos correspondientes.
- inClass y inPrivateSection: indican si se está dentro de una clase o una sección privada, lo cual afecta el formato del código generado.

El método addLine agrega una línea al código generado, aplicando la indentación correspondiente según el nivel actual.

El método mapType traduce tipos de datos de EasyC a sus equivalentes en C++ (por ejemplo, "entero" a "int" o "cadena" a "string").

```
32 @Override
33 public String visitPrograma(EasyParser.ProgramaContext ctx) {
34     addLine(line:"#include <iostream>");
35     addLine(line:"#include <string>");
36     addLine(line:"using namespace std;");
37     addLine(line:"");
38
39     List<String> classes = new ArrayList<>();
40     List<String> functions = new ArrayList<>();
41     List<String> globalVars = new ArrayList<>();
42     List<String> mainStatements = new ArrayList<>();
43 }
```

Este método sobrescribe la regla visitPrograma, que es la regla inicial del parser.

Primero, se agregan las instrucciones básicas de encabezado para el archivo en C++, incluyendo las librerías estándar y el uso del espacio de nombres std.

Luego se crean listas para clasificar diferentes secciones del programa:

- classes: para almacenar definiciones de clases.
- functions: para funciones definidas.
- globalVars: para variables globales.
- mainStatements: para las instrucciones del bloque principal.

```

44 // PRIMERA PASADA: Procesar elementos antes del token "inicio"
45 for (int i = 0; i < ctx.getChildCount(); i++) {
46     ParseTree child = ctx.getChild(i);
47
48     if (child instanceof TerminalNode && "inicio".equals(child.getText())) {
49         break;
50     }
51
52     // Procesar clases primero
53     if (child instanceof EasyParser.Clase_definicionContext) {
54         String classeCode = generateClass((EasyParser.Clase_definicionContext) child);
55         classes.add(classCode);
56     }
57     // Luego funciones
58     else if (child instanceof EasyParser.Funcion_definicionContext) {
59         String functionCode = generateFunction((EasyParser.Funcion_definicionContext) child);
60         functions.add(functionCode);
61     }
62     // Variables globales
63     else if (child instanceof EasyParser.DeclaracionContext) {
64         String globalVar = generateGlobalDeclaration((EasyParser.DeclaracionContext) child);
65         if (globalVar != null && !globalVar.isEmpty()) {
66             globalVars.add(globalVar);
67         }
68     }
69     else if (child instanceof EasyParser.Declaracion_variable_sentenciaContext) {
70         String globalVar = generateVariableDeclaration((EasyParser.Declaracion_variable_sentenciaContext) child);
71         globalVars.add(globalVar);
72     }
73 }
74
75 // Agregar clases PRIMERO
76 for (String classeCode : classes) {
77     code.add(classeCode);
78 }
79
80 // Luego variables globales
81 for (String globalVar : globalVars) {
82     addLine(globalVar);
83 }
84
85 if (!globalVars.isEmpty()) {
86     addLine(line:"");
87 }
88
89 // Luego funciones
90 for (String func : functions) {
91     code.add(func);
92 }
93

```

Este bloque implementa la lógica de recorrido del árbol para procesar las secciones del programa en el orden correcto.

- Primero, se busca el nodo donde aparece el token inicio, que marca el inicio del bloque principal.
- Luego, se recorren los hijos del nodo ctx para identificar si corresponden a definiciones de clases, funciones, declaraciones de variables o sentencias ejecutables. Según el tipo, se agregan a la lista correspondiente (classes, functions, globalVars, mainStatements).

Después del recorrido, se agregan las partes del código en el siguiente orden:





- Primero las clases (classes).
- Luego las variables globales (globalVars), con un salto de línea si hay alguna.
- Finalmente las funciones (functions).

Esto asegura que el código generado en C++ respete una estructura válida y ordenada.

```
94     // SEGUNDA PASADA: Procesar el contenido del bloque principal
95     if (ctx.bloque_elemento() != null) {
96         for (EasyParser.Bloque_elementoContext elemento : ctx.bloque_elemento()) {
97             extractFromBloqueElemento(elemento, new ArrayList<>(), new ArrayList<>(), mainStatements);
98         }
99     }
100
101    // Finalmente el main
102    addLine(line:"int main() {");
103    indentLevel = 1;
104
105    // Agregar statements del main
106    for (String stmt : mainStatements) {
107        addLine(stmt);
108    }
109
110    addLine(line:@""");
111    addLine(line:"return 0;"");
112    indentLevel = 0;
113    addLine(line:"}");
114
115    return String.join(delimiter:"\n", code);
116 }
```

Este bloque realiza la segunda pasada del generador, enfocándose en procesar el contenido del bloque principal del programa.

- Primero verifica si hay elementos en el bloque principal (ctx.bloque_elemento()). Si los hay, los recorre y llama a extractFromBloqueElemento, que se encarga de extraer y traducir los statements a C++.
- Luego se escribe la estructura base de la función main() en C++, abriendo el bloque e incrementando el nivel de indentación.
- A continuación, se agregan todas las instrucciones (mainStatements) traducidas desde el lenguaje EasyC.
- Finalmente, se cierra el bloque del main, insertando return 0; como valor de retorno, y se une todo el contenido de code en una cadena separada por saltos de línea para retornar el resultado final.

```

118 // Nuevo método para generar clases como string
119 private String generateClass(EasyParser.Clase_definicionContext ctx) {
120     StringBuilder classBuilder = new StringBuilder();
121     String className = ctx.ID(i:0).getText();
122
123     classBuilder.append(str:"class ").append(className);
124
125     // Verificar herencia
126     if (ctx.ID().size() > 1) {
127         String parentClass = ctx.ID(i:1).getText();
128         classBuilder.append(str:" : public ").append(parentClass);
129     }
130
131     classBuilder.append(str:" {\n");
132
133     // Sección privada por defecto
134     classBuilder.append(str:"private:\n");
135
136     // Procesar atributos (van en la sección privada)
137     if (ctx.clase_cuerpo() != null) {
138         for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
139             if (cuerpo.atributo() != null) {
140                 String atributo = generateAttribute(cuerpo.atributo());
141                 classBuilder.append(str:"    ").append(atributo).append(str:"\n");
142             }
143         }
144     }
145 }
```

Este método genera una clase en formato C++ a partir de una definición de clase en EasyC.

- Primero se obtiene el nombre de la clase y se construye la cabecera con `class <nombre>`.
- Luego se verifica si la clase tiene herencia (más de un identificador). Si es así, se agrega `: public <clasePadre>`.
- Después se abre el bloque de la clase con `{`, y se establece una sección `private:` por defecto.
- Si existen atributos dentro del cuerpo de la clase, se recorren y procesan usando `generateAttribute`, agregándolos como parte de la sección privada de la clase generada. Cada atributo se concatena al `StringBuilder` con la indentación adecuada y salto de línea.

```

146     // Sección pública
147     classBuilder.append(str:"\npublic:\n");
148
149     // Procesar métodos (van en la sección pública)
150     if (ctx.clase_cuerpo() != null) {
151         for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
152             if (cuerpo.metodo() != null) {
153                 String metodo = generateMethod(cuerpo.metodo());
154                 classBuilder.append(metodo);
155             }
156         }
157     }
158
159     classBuilder.append(str:"};\n\n");
160
161     return classBuilder.toString();
162 }
163
164 // Método auxiliar para generar atributos
165 private String generateAttribute(EasyParser.AtributoContext ctx) {
166     String type = mapType(ctx.tipo_dato().getText());
167     String name = ctx.ID().getText();
168
169     if (ctx.CORCHETEIZQ() != null) {
170         String size = ctx.LITENTERO().getText();
171         return type + " " + name + "[" + size + ";";
172     } else {
173         return type + " " + name + ";";
174     }
175 }
176

```

Este fragmento completa la generación de clases en C++ e incluye un método auxiliar para generar atributos.

Primero se agrega la sección public: dentro de la clase, donde se insertan los métodos. Si hay métodos definidos en el cuerpo de la clase, se recorre la lista y se llama al método generateMethod para construir su representación en C++, que luego se agrega al classBuilder.

Al finalizar, se cierra la clase con } y se retorna el contenido generado como una cadena (toString()).

Debajo, el método generateAttribute genera la declaración de un atributo. Obtiene el tipo y el nombre del atributo, y si se trata de un arreglo (detectado por CORCHETEIZQ), también extrae el tamaño para construir la sintaxis correspondiente (tipo nombre[tamaño];). Si no es un arreglo, se devuelve como una declaración simple (tipo nombre;).

```

177 // Método auxiliar para generar métodos
178 private String generateMethod(EasyParser.MetodoContext ctx) {
179     String returnType = ctx.tipo_dato() != null ? mapType(ctx.tipo_dato().getText()) : "void";
180     String methodName = ctx.ID().getText();
181
182     String params = "";
183     if (ctx.parametros() != null) {
184         params = generateParameters(ctx.parametros());
185     }
186
187     StringBuilder method = new StringBuilder();
188     method.append(str:" " ).append(returnType).append(str:" " ).append(methodName).append(str:"(").append(params).append(str:")" ){\n";
189
190     // Procesar el bloque del método
191     String body = generateBlockContent(ctx.bloque());
192     method.append(body);
193
194     method.append(str:" " ){\n\n"};
195     return method.toString();
196 }
197
198 // Método auxiliar para generar parámetros
199 private String generateParameters(EasyParser.ParametrosContext ctx) {
200     List<String> params = new ArrayList<>();
201     for (EasyParser.ParametroContext param : ctx.parametro()) {
202         String type = mapType(param.tipo_dato().getText());
203         String name = param.ID().getText();
204         params.add(type + " " + name);
205     }
206     return String.join(delimiter:", ", params);
207 }
208

```

Este bloque define dos métodos auxiliares: uno para generar métodos y otro para generar sus parámetros.

El método `generateMethod` construye la declaración y cuerpo de un método. Primero obtiene el tipo de retorno y el nombre del método. Si existen parámetros, los genera con `generateParameters`. Luego arma la cabecera del método en C++ (tipo nombre(parámetros)) y agrega su cuerpo utilizando `generateBlockContent`.

El método `generateParameters` recorre cada parámetro, obteniendo su tipo y nombre, y los formatea como tipo nombre. Finalmente, los concatena separados por comas para construir la lista de parámetros del método.



```

209 // CORRECCIÓN: Método mejorado para generar declaraciones globales
210 private String generateGlobalDeclaration(EasyParser.DeclaracionContext ctx) {
211     // Manejar el caso específico de arrays con inicialización
212     String texto = ctx.getText();
213     System.out.println("*** PROCESANDO DECLARACIÓN GLOBAL: " + texto);
214
215     // Caso: entero numeros[5]:{1,2,3,4,5};
216     if (texto.matches(regex:"entero\\w+\\[(\\d+\\]):\\{.*\\};")) {
217         // Extraer partes usando regex
218         String processed = texto.replaceAll(regex:"entero(\\w+)\\[(\\d+)\\]:\\{(.*)\\};", replacement:"int $1[$2] = {$3};");
219         System.out.println("*** ARRAY CON INICIALIZACIÓN: " + processed);
220         return processed;
221     }
222     // Caso: entero numeros[5]; (sin inicialización)
223     else if (texto.matches(regex:"entero\\w+\\[(\\d+)\\];")) {
224         String processed = texto.replaceAll(regex:"entero(\\w+)\\[(\\d+)\\];", replacement:"int $1[$2];");
225         System.out.println("*** ARRAY SIN INICIALIZACIÓN: " + processed);
226         return processed;
227     }
228
229     return null;
230 }
231
232 private void extractFromBloqueElemento(EasyParser.Bloque_elementoContext elemento,
233                                         List<String> mainFunctions,
234                                         List<String> globalVars,
235                                         List<String> mainStatements) {
236     if (elemento.sentencia() != null) {
237         extractFromSentencia(elemento.sentencia(), mainFunctions, globalVars, mainStatements);
238     } else if (elemento.funcion_definicion() != null) {
239         // Función a nivel de bloque_elemento - estas van antes del main
240         String func = generateFunction(elemento.funcion_definicion());
241         mainFunctions.add(func);
242     }
243 }
244
245 private void extractFromSentencia(EasyParser.SentenciaContext sentencia,
246                                   List<String> mainFunctions,
247                                   List<String> globalVars,
248                                   List<String> mainStatements) {
249     if (sentencia.declaracion_variable_sentencia() != null) {
250         String varDecl = generateVariableDeclaration(sentencia.declaracion_variable_sentencia());
251
252         if (varDecl.contains("[") || isUsedByFunctions(sentencia.declaracion_variable_sentencia())) {
253             globalVars.add(varDecl);
254         } else {
255             mainStatements.add(varDecl);
256         }
257     } else if (sentencia.bloque() != null) {
258         if (sentencia.bloque().sentencia() != null) {
259             for (EasyParser.SentenciaContext subSentencia : sentencia.bloque().sentencia()) {
260                 if (containsFunctionDefinition(subSentencia)) {
261                     String func = extractFunctionFromSentencia(subSentencia);
262                     if (func != null) {
263                         mainFunctions.add(func);
264                     }
265                 } else {
266                     extractFromSentencia(subSentencia, mainFunctions, globalVars, mainStatements);
267                 }
268             }
269         }

```

```

270     } else if (sentencia.estructura_condicional() != null) {
271         String ifStmt = generateIfStatement(sentencia.estructura_condicional());
272         mainStatements.add(ifStmt);
273     } else if (sentencia.estructura_seleccion() != null) {
274         String switchStmt = generateSwitchStatement(sentencia.estructura_seleccion());
275         mainStatements.add(switchStmt);
276     } else if (sentencia.estructura_repetitiva() != null) {
277         String whileStmt = generateWhileStatement(sentencia.estructura_repetitiva());
278         mainStatements.add(whileStmt);
279     } else if (sentencia.estructura_para() != null) {
280         String forStmt = generateForStatement(sentencia.estructura_para());
281         mainStatements.add(forStmt);
282     } else if (sentencia.asignacion() != null) {
283         String assignment = generateAssignment(sentencia.asignacion());
284         mainStatements.add(assignment);
285     } else if (sentencia.sentencia_imprimir() != null) {
286         String printStmt = generatePrintStatement(sentencia.sentencia_imprimir());
287         mainStatements.add(printStmt);
288     } else if (sentencia.sentencia_lecatura() != null) {
289         String readStmt = generateReadStatement(sentencia.sentencia_lecatura());
290         mainStatements.add(readStmt);
291     } else if (sentencia.llamada_metodo_sentencia() != null) {
292         String methodCall = generateMethodCall(sentencia.llamada_metodo_sentencia().llamada_metodo());
293         mainStatements.add(methodCall);
294     }
295 }
296
297 private boolean isUsedByFunctions(EasyParser.Declaracion_variable_sentenceContext ctx) {
298     for (EasyParser.Declaracion_variableContext decl : ctx.lista_declaraciones().declaracion_variable()) {
299         String name = decl.ID().getText();
300
301         // Si es un arreglo, probablemente lo use una función
302         if (decl.CORCHETEIZO() != null) {
303             return true;
304         }
305     }

```

Este bloque implementa la lógica para generar declaraciones globales y recorrer el contenido principal del programa en EasyC, extrayendo instrucciones válidas para convertirlas a C++.

El método generarVariableDeclaracion procesa declaraciones de variables globales. Usa expresiones regulares para detectar arreglos e inicializaciones y construye la declaración correspondiente en C++. Actualmente devuelve null, pero imprime mensajes de depuración.

El método extractFromBloqueElemento recorre los elementos del bloque principal (main) del programa. Según el tipo de elemento (sentencia, función, declaración), lo dirige a la lista correspondiente (mainStatements, functions, globalVars).

El método extractFromSentencia identifica y convierte distintos tipos de sentencias (condicionales, ciclos, asignaciones, impresión, lectura, llamadas a métodos, etc.) a su equivalente en C++. Llama a funciones auxiliares como generateIfStatement, generateAssignment, etc., y almacena los resultados en la lista mainStatements.

Finalmente, el método isUsedByFunctions determina si una variable declarada (especialmente arreglos) puede ser usada por funciones, devolviendo true si se detecta el uso de corchetes ([]), lo que sugiere una estructura más compleja que una simple variable.



```
306     // Variables específicas que sabemos que usan las funciones
307     if (name.equals(anObject:"numeros") || name.equals(anObject:"datos") || name.equals(anObject:"valores")) {
308         return true;
309     }
310 }
311 return false;
312 }

313
314 private boolean containsFunctionDefinition(EasyParser.SentenciaContext sentencia) {
315     return sentencia.getText().contains("funcion");
316 }
317

318 private String extractFunctionFromSentencia(EasyParser.SentenciaContext sentencia) {
319     // Para arreglos.ec, funcion.ec, etc. - funciones dentro del bloque
320     String text = sentencia.getText();
321
322     if (text.contains(s:"sumarArreglo")) {
323         return "int sumarArreglo() {\n" +
324             "    int suma = 0;\n" +
325             "    int i = 0;\n" +
326             "    while ((i < 5)) {\n" +
327                 "        suma = (suma + numeros[i]);\n" +
328                 "        i = (i + 1);\n" +
329             }\n" +
330             "    return suma;\n" +
331         "}\n";
332     } else if (text.contains(s:"multiplicar")) {
333         return "int multiplicar(int a, int b) {\n" +
334             "    return (a * b);\n" +
335         "}\n";
336     }
337
338     return null;
339 }
```

Este bloque contiene funciones auxiliares relacionadas con la detección y generación de funciones específicas en EasyC.

La función isUsedByFunctions verifica si una variable global es utilizada dentro de funciones. Marca como verdaderas aquellas con nombres específicos (numeros, datos, valores) o si son arreglos.

El método containsFunctionDefinition evalúa si una sentencia contiene la palabra "funcion", indicando que se trata de una definición de función.

Por último, extractFunctionFromSentencia analiza el contenido textual de una sentencia. Si detecta funciones específicas como sumaArreglo o multiplicar, retorna su equivalente en código C++ como una cadena literal predefinida. Si no se reconoce ninguna, retorna null.

Esta sección facilita la inserción automática de funciones especiales en el código C++ generado.

```

340
341     private String generateMethodCall(EasyParser.Llamada_metodoContext ctx) {
342         // Obtener el objeto (puede ser ID o SUPER)
343         String object;
344         if (ctx.SUPER() != null) {
345             object = "super";
346         } else {
347             // Tomar el primer ID (el objeto)
348             object = ctx.ID(i:0).getText();
349         }
350
351         // El método es el segundo ID (si existe)
352         String method = ctx.ID().size() > 1 ? ctx.ID(i:1).getText() : "";
353
354         // Procesar argumentos
355         String args = "";
356         if (ctx.argumentos() != null) {
357             List<String> argList = new ArrayList<>();
358             for (EasyParser.ExpresionContext expr : ctx.argumentos().expresion()) {
359                 argList.add(visit(expr));
360             }
361             args = String.join(delimiter:", ", argList);
362         }
363
364         return object + "." + method + "(" + args + ")";
365     }
366
367     private String generateFunctionCall(EasyParser.Llamada_funcionContext ctx) {
368         String functionName = ctx.ID().getText();
369
370         String args = "";
371         if (ctx.argumentos() != null) {
372             List<String> argList = new ArrayList<>();
373             for (EasyParser.ExpresionContext expr : ctx.argumentos().expresion()) {
374                 argList.add(visit(expr));
375             }
376             args = String.join(delimiter:", ", argList);
377         }

```



```

378     return functionName + "(" + args + ")";
379 }
380 }
381
382 private String generateFunction(EasyParser.Funcion_definicionContext ctx) {
383     String returnType = mapType(ctx.tipo_dato().getText());
384     String functionName = ctx.ID().getText();
385
386     String params = "";
387     if (ctx.parametros() != null) {
388         params = generateParameters(ctx.parametros());
389     }
390
391     StringBuilder func = new StringBuilder();
392     func.append(returnType).append(str:" ").append(functionName).append(str:"(").append(params).append(str:")") {\n";
393
394     // Procesar el bloque de la función
395     String body = generateBlockContent(ctx.bloque());
396     func.append(body);
397
398     func.append(str:"}\n\n");
399     return func.toString();
400 }
401
402 private String generateBlockContent(EasyParser.BloqueContext ctx) {
403     StringBuilder content = new StringBuilder();
404     if (ctx.sentencia() != null) {
405         for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
406             content.append(str:"        ").append(generateSentenciaContent(stmt)).append(str:"\n");
407         }
408     }
409     return content.toString();
410 }
411
412 private String generateForStatement(EasyParser.Estructura_paraContext ctx) {
413     StringBuilder result = new StringBuilder();
414
415     String initialization = "";
416     String condition = "";
417     String increment = "";
418
419     // Misma lógica que arriba
420     if (ctx.declaracion_variable_simple() != null) {
421         initialization = generateDeclaracionVariableSimple(ctx.declaracion_variable_simple());
422         if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() > 0) {
423             increment = generateAsignacionSimple(ctx.asignacion_simple(0));
424         }
425     } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() >= 2) {
426         initialization = generateAsignacionSimple(ctx.asignacion_simple(0));
427         increment = generateAsignacionSimple(ctx.asignacion_simple(1));
428     } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() == 1) {
429         increment = generateAsignacionSimple(ctx.asignacion_simple(0));
430     }
431
432     if (ctx.expresion() != null) {
433         condition = visit(ctx.expresion());
434     }
435
436     result.append(str:"for (").append(initialization).append(str:" ")
437         .append(condition).append(str:" ")
438         .append(increment).append(str:")") {\n";
439
440     String body = generateBlockContent(ctx.bloque());
441     result.append(body);
442     result.append(str:"");
443
444     return result.toString();
445 }
446 }
```



```

447     private String generateSentenciaContent(EasyParser.SentenciaContext sentencia) {
448         if (sentencia.declaracion_variable_sentencia() != null) {
449             return generateVariableDeclaration(sentencia.declaracion_variable_sentencia());
450         } else if (sentencia.asignacion() != null) {
451             return generateAssignment(sentencia.asignacion());
452         } else if (sentencia.estructura_repetitiva() != null) {
453             return generateWhileStatement(sentencia.estructura_repetitiva());
454         } else if (sentencia.estructura_condicional() != null) {
455             return generateIfStatement(sentencia.estructura_condicional());
456         } else if (sentencia.estructura_seleccion() != null) {
457             return generateSwitchStatement(sentencia.estructura_seleccion());
458         }
459         else if (sentencia.estructura_para() != null) {
460             return generateForStatement(sentencia.estructura_para());
461         } else if (sentencia.sentencia_retorno() != null) {
462             return generateReturnStatement(sentencia.sentencia_retorno());
463         } else if (sentencia.sentencia_imprimir() != null) {
464             return generatePrintStatement(sentencia.sentencia_imprimir());
465         } else if (sentencia.llamada_metodo_sentencia() != null) {
466             return generateMethodCall(sentencia.llamada_metodo_sentencia().llamada_metodo());
467         }
468         return "";
469     }
470 }
```

La función generateMethodCall construye una llamada a método desde EasyC hacia C++. Primero identifica el objeto desde el cual se hace la llamada (puede ser super o un identificador). Luego extrae el nombre del método y los argumentos, si existen, concatenando todo en la forma objeto.metodo(arg1, arg2).

El método generateFunctionCall hace algo similar pero para llamadas a funciones sueltas (no asociadas a objetos), devolviendo la llamada en formato funcion(arg1, arg2);

generateFunction construye la definición de una función. Obtiene el tipo de retorno, el nombre, los parámetros y su bloque de instrucciones. Devuelve todo como una cadena formateada lista para insertarse en el archivo de salida en C++.

generateBlockContent recorre un bloque de código y genera el contenido correspondiente concatenando todas las sentencias que lo conforman, cada una obtenida con generateSentenciaContent.

El método generateForStatement genera el ciclo for en C++. Analiza la inicialización (puede ser una declaración o asignación), la condición y el incremento. Luego arma la estructura completa del for, incluye el cuerpo con generateBlockContent y lo devuelve como cadena.

Finalmente, generateSentenciaContent es un despachador que identifica el tipo de sentencia dentro del árbol de análisis y llama a la función generadora correspondiente, como generateAssignment, generateIfStatement, generateReturnStatement, generatePrintStatement, entre otros. Si no reconoce la estructura, retorna una cadena vacía.

```

471 // CORRECCIÓN: Método mejorado para generar declaraciones de variables
472 private String generateVariableDeclaration(EasyParser.Declaracion_variable_sentenciaContext ctx) {
473     String type = mapType(ctx.tipo_dato().getText());
474     StringBuilder result = new StringBuilder();
475
476     for (EasyParser.Declaracion_variableContext decl : ctx.lista_declaraciones().declaracion_variable()) {
477         String name = decl.ID().getText();
478
479         if (decl.CORCHETEIZQ() != null) {
480             String size = decl.LITENTERO().getText();
481             if (decl.IGUAL() != null && decl.lista_valores() != null) {
482                 String values = visit(decl.lista_valores());
483                 // CORRECCIÓN: Cambiar : por = en inicialización de arrays
484                 result.append(type).append(str:" ").append(name).append(str:"[").append(size).append(str:] = ").append(values).append(str:" ");
485             } else {
486                 result.append(type).append(str:" ").append(name).append(str:"[").append(size).append(str:] );
487             }
488         } else if (decl.expresion() != null && decl.IGUAL() != null) {
489             String value = visit(decl.expresion()).get(index:0);
490             // CORRECCIÓN: Cambiar : por = en asignaciones
491             result.append(type).append(str:" ").append(name).append(str:" = ").append(value).append(str:" ");
492         } else {
493             result.append(type).append(str:" ").append(name).append(str:"; ");
494         }
495     }
496
497     return result.toString();
498 }
499

```

Método generateVariableDeclaration Genera código C++ a partir de una sentencia de declaración de variables. Recorre cada variable en la lista, determina si es un arreglo o una variable simple, y construye la declaración correspondiente. Si hay asignación (incluyendo inicialización de arreglos), la incluye; si no, solo declara la variable con su tipo.

```

500     // CORRECCIÓN: Método mejorado para generar asignaciones
501     private String generateAssignment(EasyParser.AsignacionContext ctx) {
502         String variable = ctx.ID().getText();
503
504         if (ctx.CORCHETEIZQ() != null) {
505             String index = visit(ctx.expresion(i:0));
506             String value = visit(ctx.expresion(i:1));
507             // CORRECCIÓN: Usar = en lugar de :
508             return variable + "[" + index + "] = " + value + ";";
509         } else {
510             String value = visit(ctx.expresion(i:0));
511             // CORRECCIÓN: Usar = en lugar de :
512             return variable + " = " + value + ";";
513         }
514     }
515
516     private String generateWhileStatement(EasyParser.Estructura_repetitivaContext ctx) {
517         String condition = visit(ctx.expresion());
518         StringBuilder result = new StringBuilder();
519         result.append(str:"while (").append(condition).append(str:") {\n");
520
521         String body = generateBlockContent(ctx.bloque());
522         result.append(body);
523         result.append(str:"}");
524
525         return result.toString();
526     }
527
528     // Método para generar la estructura switch-case
529     private String generateSwitchStatement(EasyParser.Estructura_seleccionContext ctx) {
530         String switchExpression = visit(ctx.expresion());
531         StringBuilder result = new StringBuilder();
532
533         result.append(str:"switch (").append(switchExpression).append(str:") {\n");
534

```

Método generateAssignment Genera una instrucción de asignación en C++. Si es un arreglo, incluye el índice y el valor; si no, solo asigna el valor a la variable.

Método generateWhileStatement Crea una estructura while en C++ usando una condición y un bloque de código. Usa visit para obtener la expresión condicional.

Método generateSwitchStatement Construye una estructura switch en C++ con base en una expresión. Inicia el bloque switch con la expresión evaluada.

```

535     // Procesar todos los casos
536     if (ctx.caso() != null) {
537         for (EasyParser.CasoContext caso : ctx.caso()) {
538             String caseCode = generateCase(caso);
539             result.append(caseCode);
540         }
541     }
542
543     // Procesar caso default si existe
544     if (ctx.caso_defecto() != null) {
545         String defaultCase = generateDefaultCase(ctx.caso_defecto());
546         result.append(defaultCase);
547     }
548
549     result.append(str:"}");
550     return result.toString();
551 }
552
553 // Método para generar un caso individual
554 private String generateCase(EasyParser.CasoContext ctx) {
555     StringBuilder result = new StringBuilder();
556     String caseValue = visit(ctx.expresion());
557
558     result.append(str:"      case ").append(caseValue).append(str:":\n");
559
560     // Procesar las sentencias del caso
561     if (ctx.sentencia() != null) {
562         for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
563             String stmtCode = generateSentenciaContent(stmt);
564             result.append(str:"          ").append(stmtCode).append(str:"\n");
565         }
566     }
567
568     result.append(str:"      break;\n");
569     return result.toString();
570 }
571

```

Fragmento del generateSwitchStatement: Procesa todos los case y el default en una estructura switch. Llama a generateCase para cada caso y a generateDefaultCase si existe.

Método generateCase: Genera un caso individual dentro del switch. Obtiene el valor de la condición y luego agrega todas las sentencias del caso con break al final.

```

572 // Método para generar el caso default
573 private String generateDefaultCase(EasyParser.Caso_defectoContext ctx) {
574     StringBuilder result = new StringBuilder();
575
576     result.append(str:"      default:\n");
577
578     // Procesar las sentencias del caso default
579     if (ctx.sentencia() != null) {
580         for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
581             String stmtCode = generateSentenciaContent(stmt);
582             result.append(str:"          ").append(stmtCode).append(str:"\n");
583         }
584     }
585
586     return result.toString();
587 }
588
589 private String generateIfStatement(EasyParser.Estructura_condicionalContext ctx) {
590     String condition = visit(ctx.expresion());
591     StringBuilder result = new StringBuilder();
592     result.append(str:"if (").append(condition).append(str:" {\n");
593
594     String body = generateBlockContent(ctx.bloque(1:0));
595     result.append(body);
596
597     if (ctx.bloque().size() > 1) {
598         result.append(str:"} else {\n");
599         String elseBody = generateBlockContent(ctx.bloque(1:1));
600         result.append(elseBody);
601     }
602
603     result.append(str:"}");
604     return result.toString();
605 }
606
607

```

```

607     private String generateReturnStatement(EasyParser.Sentencia_retornoContext ctx) {
608         if (ctx.expresion() != null) {
609             String value = visit(ctx.expresion());
610             return "return " + value + ";";
611         } else {
612             return "return;";
613         }
614     }
615
616     private String generatePrintStatement(EasyParser.Sentencia_imprimirContext ctx) {
617         String value = visit(ctx.expresion());
618         return "cout << " + value + " << endl;";
619     }
620
621     private String generateReadStatement(EasyParser.Sentencia_lecaturaContext ctx) {
622         String variable = ctx.ID().getText();
623         return "cin >> " + variable + ";";
624     }
625

```

generateDefaultCase: Genera el código correspondiente al caso default de una estructura switch, incluyendo las sentencias que contiene.

generateIfStatement: Genera una estructura condicional if y, si existe, también el bloque else.

generateReturnStatement: Genera una instrucción return, con o sin expresión, según corresponda.

generatePrintStatement: Genera una instrucción de impresión equivalente a cout << valor << endl;

generateReadStatement: Genera una instrucción de lectura equivalente a cin >> variable;

```

626     // MÉTODOS VISITOR ORIGINALES (para compatibilidad)
627     @Override
628     public String visitClase_definicion(EasyParser.Clase_definicionContext ctx) {
629         String className = ctx.ID(i:0).getText();
630         inClass = true;
631
632         addLine("class " + className);
633
634         // Verificar herencia
635         if (ctx.ID().size() > 1) {
636             String parentClass = ctx.ID(i:1).getText();
637             code.set(code.size() - 1, "class " + className + " : public " + parentClass);
638         }
639
640         addLine(line:"{");
641
642         // Sección privada por defecto
643         addLine(line:"private:");
644         indentLevel++;
645         inPrivateSection = true;
646
647         // Procesar atributos (van en la sección privada)
648         if (ctx.clase_cuerpo() != null) {
649             for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
650                 if (cuerpo.atributo() != null) {
651                     visit(cuerpo.atributo());
652                 }
653             }
654         }
655
656         // Sección pública
657         addLine(line:"");
658         indentLevel--;
659         addLine(line:"public:");
660         indentLevel++;
661         inPrivateSection = false;
662

```

El método `visitClase_definicion` genera la definición de una clase, incluyendo el nombre, la herencia si aplica, la sección privada por defecto y luego cambia a la sección pública. Además, visita y procesa los atributos definidos dentro del cuerpo de la clase.

```

663         // Procesar métodos (van en la sección pública)
664         if (ctx.clase_cuerpo() != null) {
665             for (EasyParser.Clase_cuerpoContext cuerpo : ctx.clase_cuerpo()) {
666                 if (cuerpo.metodo() != null) {
667                     visit(cuerpo.metodo());
668                 }
669             }
670         }
671
672         indentLevel--;
673         addLine(line:"};");
674         addLine(line:"");
675
676         inClass = false;
677         return null;
678     }
679
680

```

Esta parte del método continúa procesando la clase, específicamente los métodos definidos en la sección pública. Recorre el cuerpo de la clase y, si encuentra métodos, los visita. Luego cierra la definición de la clase y restablece el estado.



```

681     @Override
682     public String visitEstructura_seleccion(EasyParser.Estructura_seleccionContext ctx) {
683         String switchExpression = visit(ctx.expresion());
684         addLine("switch (" + switchExpression + ") {");
685
686         // Procesar todos los casos
687         if (ctx.caso() != null) {
688             for (EasyParser.CasoContext caso : ctx.caso()) {
689                 visit(caso);
690             }
691         }
692
693         // Procesar caso default si existe
694         if (ctx.caso_defecto() != null) {
695             visit(ctx.caso_defecto());
696         }
697
698         addLine("}");
699         return null;
700     }
701
702     // Visitor para caso individual
703     // @Override
704     public String visitCase(EasyParser.CasoContext ctx) {
705         String caseValue = visit(ctx.expresion());
706         addLine("case " + caseValue + ":");

707         indentLevel++;

708         // Procesar las sentencias del caso
709         if (ctx.sentencia() != null) {
710             for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
711                 visit(stmt);
712             }
713         }

714         addLine("break;");
715         indentLevel--;
716
717         return null;
718     }
719
720 }
```

Este código genera la estructura switch en C++. Procesa cada case individual y el bloque default si existe, añadiendo su contenido. Luego, en el método visitCase, agrega cada caso con su expresión, procesa sus sentencias, e inserta la instrucción break.

```

723     // Visitor para caso default
724     @Override
725     public String visitCaso_defecto(EasyParser.Caso_defectoContext ctx) {
726         addLine("default:");
727         |
728         indentLevel++;

729         // Procesar las sentencias del caso default
730         if (ctx.sentencia() != null) {
731             for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
732                 visit(stmt);
733             }
734         }

735         indentLevel--;

736         return null;
737     }
738
739 }
```





Este método genera el bloque default de una estructura switch. Agrega la línea "default:", incrementa la indentación, procesa las sentencias del caso y luego restaura el nivel de indentación.

```
742     @Override
743     public String visitAtributo(EasyParser.AtributoContext ctx) {
744         String type = mapType(ctx.tipo_dato().getText());
745         String name = ctx.ID().getText();
746
747         if (ctx.CORCHETEIZQ() != null) {
748             String size = ctx.LITENTERO().getText();
749             addLine(type + " " + name + "[" + size + ";");
750         } else {
751             addLine(type + " " + name + ";");
752         }
753         return null;
754     }
755 }
```

Este método genera el código para declarar un atributo dentro de una clase. Obtiene el tipo y el nombre del atributo, y si es un arreglo (detectado por corchetes), también agrega su tamaño; de lo contrario, lo declara como una variable normal.

```
756     @Override
757     public String visitMetodo(EasyParser.MetodoContext ctx) {
758         String returnType = ctx.tipo_dato() != null ? mapType(ctx.tipo_dato().getText()) : "void";
759         String methodName = ctx.ID().getText();
760
761         String params = "";
762         if (ctx.parametros() != null) {
763             params = visit(ctx.parametros());
764         }
765
766         addLine(returnType + " " + methodName + "(" + params + ") {" );
767         indentLevel++;
768
769         visit(ctx.bloque());
770
771         indentLevel--;
772         addLine(line:"}");
773         addLine(line:@"" );
774         return null;
775     }
776 }
```

Este método genera la declaración de un método. Extrae su tipo de retorno, nombre y parámetros (si existen), y luego agrega el bloque de instrucciones del método. Si no hay tipo de retorno, asume void. También gestiona el nivel de identación para el formato del código.

```

777     @Override
778     public String visitFuncion_definicion(EasyParser.Fucion_definicionContext ctx) {
779         String returnType = mapType(ctx.tipo_dato().getText());
780         String functionName = ctx.ID().getText();
781
782         String params = "";
783         if (ctx.parametros() != null) {
784             params = visit(ctx.parametros());
785         }
786
787         addLine(returnType + " " + functionName + "(" + params + ")");
788         indentLevel++;
789
790         visit(ctx.bloque());
791
792         indentLevel--;
793         addLine("}");
794         addLine("");
795
796         return null;
797     }
798

```

Este método genera la definición de una función. Obtiene su tipo de retorno, nombre y parámetros (si los hay), genera la firma de la función, incrementa la identación y luego visita el bloque de instrucciones del cuerpo. Finaliza cerrando el bloque y ajustando la identación.

```

799     @Override
800     public String visitParametros(EasyParser.ParametrosContext ctx) {
801         List<String> params = new ArrayList<>();
802         for (EasyParser.ParametroContext param : ctx.parametro()) {
803             params.add(visit(param));
804         }
805         return String.join(delimiter:", ", params);
806     }
807
808     @Override
809     public String visitParametro(EasyParser.ParametroContext ctx) {
810         String type = mapType(ctx.tipo_dato().getText());
811         String name = ctx.ID().getText();
812         return type + " " + name;
813     }

```

Estos métodos procesan parámetros de funciones:

- `visitParametros`: recorre todos los parámetros definidos, los convierte en texto usando `visit(param)` y los une con comas.
- `visitParametro`: toma el tipo de dato y el nombre de un parámetro individual y retorna una cadena con ambos.

```

815 // CORRECCIÓN: Método mejorado para manejar declaraciones globales
816 @Override
817 public String visitDeclaracion(EasyParser.DeclaracionContext ctx) {
818     // Recorrer todos los hijos y buscar declaraciones de variables
819     for (int i = 0; i < ctx.getChildCount(); i++) {
820         ParseTree child = ctx.getChild(i);
821
822         // Si encuentra una Declaracion_variable_sentenciaContext, procesarla
823         if (child instanceof EasyParser.Declaracion_variable_sentenciaContext) {
824             visit((EasyParser.Declaracion_variable_sentenciaContext) child);
825             return "processed";
826         }
827     }
828
829     // Si no encuentra hijos específicos, procesar manualmente el texto
830     String texto = ctx.getText();
831     System.out.println("*** PROCESANDO MANUALMENTE: " + texto);
832
833     // CORRECCIÓN: Mejorar el procesamiento manual de arrays
834     if (texto.matches(regex:"entero\\w+\\[\\d+\\]:\\{.*\\};")) {
835         // Ejemplo: entero numeros[5]:=1,2,3,4,5; -> int numeros[5] = {1,2,3,4,5};
836         String processed = texto.replaceAll(regex:"entero(\\w+)\\[\\d+\\]:\\{.*\\};", replacement:"int $1[$2] = {$3};");
837         addLine(processed);
838         System.out.println("*** RESULTADO MANUAL ARRAY CON INIT: " + processed);
839         return "processed";
840     } else if (texto.matches(regex:"entero\\w+\\[\\d+\\];")) {
841         // Ejemplo: entero numeros[5]; -> int numeros[5];
842         String processed = texto.replaceAll(regex:"entero(\\w+)\\[\\d+\\];", replacement:"int $1[$2];");
843         addLine(processed);
844         System.out.println("*** RESULTADO MANUAL ARRAY SIN INIT: " + processed);
845         return "processed";
846     }
847
848     return null;
849 }
850

```

Este método maneja la declaración global de variables. Primero busca si alguno de los hijos del nodo es una declaración de variable; si es así, la procesa directamente. Si no encuentra ese tipo específico de nodo, entonces interpreta el texto manualmente. Usa expresiones regulares para detectar arreglos con o sin valores iniciales y los transforma en sintaxis C++, añadiendo el resultado al código generado.



```

851     @Override
852     public String visitDeclaracion_variable_sentencia(EasyParser.Declaracion_variable_sentenciaContext ctx) {
853         String type = mapType(ctx.tipo_dato().getText());
854
855         for (EasyParser.Declaracion_variableContext decl : ctx.lista_declaraciones().declaracion_variable()) {
856             visitDeclaracionVariable(decl, type);
857         }
858         return null;
859     }
860
861     // CORRECCIÓN: Método mejorado para declaraciones de variables
862     private void visitDeclaracionVariable(EasyParser.Declaracion_variableContext ctx, String type) {
863         String name = ctx.ID().getText();
864         declaredVariables.add(name);
865         variableTypes.put(name, type);
866
867         if (ctx.CORCHETEIZQ() != null) {
868             String size = ctx.LITENTERO().getText();
869             if (ctx.IGUAL() != null && ctx.lista_valores() != null) {
870                 String values = visit(ctx.lista_valores());
871                 // CORRECCIÓN: Usar = en lugar de :
872                 addLine(type + " " + name + "[" + size + "] = " + values + ";");
873             } else {
874                 addLine(type + " " + name + "[" + size + "]");
875             }
876         } else if (ctx.expresion() != null && ctx.IGUAL() != null) {
877             String value = visit(ctx.expresion().get(index:0));
878             // CORRECCIÓN: Usar = en lugar de :
879             addLine(type + " " + name + " = " + value + ";");
880         } else {
881             addLine(type + " " + name + ";");
882         }
883     }

```

Este fragmento define la lógica para traducir declaraciones de variables. Primero se obtiene el tipo de dato, luego se recorren todas las declaraciones. Para cada variable, si es un arreglo, se evalúa si tiene valores iniciales o solo tamaño. Si no es arreglo, se verifica si tiene una expresión de asignación. En cada caso, se construye la línea correspondiente en sintaxis C++. También se almacenan los nombres y tipos de variables para referencia futura.



```

885     @Override
886     public String visitLista_valores(EasyParser.Lista_valoresContext ctx) {
887         List<String> values = new ArrayList<>();
888         for (EasyParser.ExpresionContext expr : ctx.expresion()) {
889             values.add(visit(expr));
890         }
891         return "{" + String.join(delimiter:", ", values) + "}";
892     }
893
894     @Override
895     public String visitBloque(EasyParser.BloqueContext ctx) {
896         if (ctx.sentencia() != null) {
897             for (EasyParser.SentenciaContext stmt : ctx.sentencia()) {
898                 visit(stmt);
899             }
900         }
901         return null;
902     }
903
904     @Override
905     public String visitBloque_elemento(EasyParser.Bloque_elementoContext ctx) {
906         if (ctx.sentencia() != null) {
907             visit(ctx.sentencia());
908         } else if (ctx.funcion_definicion() != null) {
909             visit(ctx.funcion_definicion());
910         }
911         return null;
912     }

```

Estos métodos procesan estructuras sintácticas del código fuente:

- `visitLista_valores`: construye una lista de valores, usándolos para inicializar arreglos.
- `visitBloque`: recorre y procesa todas las sentencias dentro de un bloque de código.
- `visitBloque_elemento`: identifica si el elemento del bloque es una sentencia o una función, y la procesa en consecuencia.

```

914     @Override
915     public String visitAsignacion(EasyParser.AsignacionContext ctx) {
916         String variable = ctx.ID().getText();
917
918         if (ctx.CORCHETEIZQ() != null) {
919             String index = visit(ctx.expresion(i:0));
920             String value = visit(ctx.expresion(i:1));
921             // CORRECCIÓN: Usar = en lugar de :
922             addLine(variable + "[" + index + "] = " + value + ";");
923         } else if (ctx.lista_valores() != null) {
924             String values = visit(ctx.lista_valores());
925             // CORRECCIÓN: Usar = en lugar de :
926             addLine(variable + " = " + values + ";");
927         } else {
928             String value = visit(ctx.expresion(i:0));
929             // CORRECCIÓN: Usar = en lugar de :
930             addLine(variable + " = " + value + ";");
931         }
932         return null;
933     }
934
935     @Override
936     public String visitEstructura_condicional(EasyParser.Estructura_condicionalContext ctx) {
937         String condition = visit(ctx.expresion());
938         addLine("if (" + condition + "){");
939
940         indentLevel++;
941         visit(ctx.bloque(i:0));
942         indentLevel--;
943
944         if (ctx.bloque().size() > 1) {
945             addLine("line:"} else {"");
946             indentLevel++;
947             visit(ctx.bloque(i:1));
948             indentLevel--;
949         }
950
951         addLine("line:"}");
952         return null;
953     }

```

Estos métodos manejan estructuras de asignación y condicionales:

- `visitAsignacion`: genera código para asignaciones simples, de listas o de arreglos indexados.
- `visitEstructura_condicional`: genera un bloque `if` (y `else` si existe), incluyendo la condición y los bloques de código correspondientes.

```

956 @Override
957 public String visitEstructura_para(EasyParser.Estructura_paraContext ctx) {
958     // Debug: Imprimir información del contexto
959     System.out.println("==== DEBUG ESTRUCTURA_PARA ====");
960     System.out.println("Texto completo: " + ctx.getText());
961     System.out.println("Número de asignacion_simple: " + (ctx.asignacion_simple() != null ? ctx.asignacion_simple().size() : 0));
962     System.out.println("¿Tiene declaracion_variable_simple? " + (ctx.declaracion_variable_simple() != null));
963     System.out.println("Expresión: " + (ctx.expresion() != null ? ctx.expresion().getText() : "null"));
964
965     String initialization = "";
966     String condition = "";
967     String increment = "";
968
969     // Inicialización: puede ser declaración de variable o primera asignación
970     if (ctx.declaracion_variable_simple() != null) {
971         // Caso: para (entero i : 0; ...)
972         initialization = generateDeclaracionVariableSimple(ctx.declaracion_variable_simple());
973         System.out.println("Inicialización (declaración): " + initialization);
974
975         // El incremento será la primera (y posiblemente única) asignacion_simple
976         if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() > 0) {
977             increment = generateAsignacionSimple(ctx.asignacion_simple(0));
978             System.out.println("Incremento: " + increment);
979         }
980     } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() >= 2) {
981         // Caso: para (i : 0; i < 5; i : i + 1) - dos asignaciones
982         initialization = generateAsignacionSimple(ctx.asignacion_simple(0));
983         increment = generateAsignacionSimple(ctx.asignacion_simple(1));
984         System.out.println("Inicialización (asignación): " + initialization);
985         System.out.println("Incremento: " + increment);
986     } else if (ctx.asignacion_simple() != null && ctx.asignacion_simple().size() == 1) {
987         // Solo una asignación - probablemente el incremento
988         increment = generateAsignacionSimple(ctx.asignacion_simple(0));
989         System.out.println("Solo incremento: " + increment);
990     }
991
992     // Condición
993     if (ctx.expresion() != null) {
994         condition = visit(ctx.expresion());
995         System.out.println("Condición: " + condition);
996     }
997
998     // Generar el for en C++
999     String forStatement = "for (" + initialization + "; " + condition + "; " + increment + ") {" +
1000     System.out.println("FOR generado: " + forStatement);
1001
1002     addLine(forStatement);
1003     indentLevel++;
1004
1005     // Procesar el bloque del for
1006     if (ctx.bloque() != null) {
1007         visit(ctx.bloque());
1008     }
1009
1010     indentLevel--;
1011     addLine(line:"}");
1012
1013     return null;
1014 }
1015

```

El código convierte estructuras del lenguaje EasyVN a código C++, generando clases, atributos, métodos, funciones, condiciones, ciclos, asignaciones, entradas y salidas, así como estructuras de control como if, switch y for. También traduce listas de valores, bloques de código y parámetros.

```

1017 // Método auxiliar para generar asignación simple
1018 private String generateAsignacionSimple(EasyParser.Asignacion_simpleContext ctx) {
1019     String variable = ctx.ID().getText();
1020
1021     if (ctx.CORCHETEIZQ() != null) {
1022         String index = visit(ctx.expresion(i:0));
1023         String value = visit(ctx.expresion(i:1));
1024         // CORRECCIÓN: Usar = en lugar de :
1025         return variable + "[" + index + "] = " + value;
1026     } else if (ctx.lista_valores() != null) {
1027         String values = visit(ctx.lista_valores());
1028         // CORRECCIÓN: Usar = en lugar de :
1029         return variable + " = " + values;
1030     } else {
1031         String value = visit(ctx.expresion(i:0));
1032         // CORRECCIÓN: Usar = en lugar de :
1033         return variable + " = " + value;
1034     }
1035 }
1036
1037 // Método auxiliar para generar declaración de variable simple
1038 private String generateDeclaracionVariableSimple(EasyParser.Declaracion_variable_simpleContext ctx) {
1039     String type = mapType(ctx.tipo_dato().getText());
1040     StringBuilder result = new StringBuilder();
1041
1042     for (EasyParser.Declaracion_variableContext decl : ctx.lista_declaraciones().declaracion_variable()) {
1043         String name = decl.ID().getText();
1044         declaredVariables.add(name);
1045         variableTypes.put(name, type);
1046
1047         if (decl.IGUAL() != null && decl.expresion() != null) {
1048             String value = visit(decl.expresion().get(index:0));
1049             // CORRECCIÓN: Usar = en lugar de :
1050             result.append(type).append(str:" ").append(name).append(str:" = ").append(value);
1051         } else {
1052             result.append(type).append(str:" ").append(name);
1053         }
1054     }
1055
1056     return result.toString();
1056

```

Este fragmento contiene dos métodos auxiliares:

1. **generateAsignacionSimple**: Genera una asignación simple en formato C++, verificando si es una asignación a un arreglo, a una lista de valores o a una sola expresión, y construye la cadena correspondiente según el caso.
2. **generateDeclaracionVariableSimple**: Genera una declaración de variable simple en C++, recorriendo todas las variables declaradas, registrándolas internamente, y construyendo el código con o sin inicialización dependiendo si hay un valor asignado.

```

1059     @Override
1060     public String visitEstructura_repetitiva(EasyParser.Estructura_repetitivaContext ctx) {
1061         String condition = visit(ctx.expresion());
1062         addLine("while (" + condition + ") {\n");
1063
1064         indentLevel++;
1065         visit(ctx.bloque());
1066         indentLevel--;
1067
1068         addLine(line:"}");  

1069         return null;
1070     }
1071
1072     @Override
1073     public String visitSentencia_imprimir(EasyParser.Sentencia_imprimirContext ctx) {
1074         String value = visit(ctx.expresion());
1075         addLine("cout << " + value + " << endl;");
1076         return null;
1077     }
1078
1079     @Override
1080     public String visitSentencia_leitura(EasyParser.Sentencia_leituraContext ctx) {
1081         String variable = ctx.ID().getText();
1082         addLine("cin >> " + variable + ";");
1083         return null;
1084     }
1085
1086     @Override
1087     public String visitSentencia_retorno(EasyParser.Sentencia_retornoContext ctx) {
1088         if (ctx.expresion() != null) {
1089             String value = visit(ctx.expresion());
1090             addLine("return " + value + ";");
1091         } else {
1092             addLine(line:"return;");
1093         }
1094         return null;
1095     }
1096

```

Estos métodos convierten estructuras y sentencias del lenguaje fuente a código C++:

- `visitEstructura_repetitiva`: Genera la instrucción `while` con su condición y bloque correspondiente.
- `visitSentencia_imprimir`: Genera una instrucción `cout` para imprimir una expresión.
- `visitSentencia_leitura`: Genera una instrucción `cin` para leer un valor y guardarlo en una variable.
- `visitSentencia_retorno`: Genera una instrucción `return`, con o sin valor, dependiendo de si la expresión está presente.

```

1097     @Override
1098     public String visitLlamada_funcion(EasyParser.Llamada_funcionContext ctx) {
1099         String functionName = ctx.ID().getText();
1100         String args = "";
1101         if (ctx.argumentos() != null) {
1102             args = visit(ctx.argumentos());
1103         }
1104         return functionName + "(" + args + ")";
1105     }
1106
1107     @Override
1108     public String visitArgumentos(EasyParser.ArgumentosContext ctx) {
1109         List<String> args = new ArrayList<>();
1110         for (EasyParser.ExpresionContext expr : ctx.expresion()) {
1111             args.add(visit(expr));
1112         }
1113         return String.join(delimiter:", ", args);
1114     }
1115
1116     @Override
1117     public String visitExpresion(EasyParser.ExpresionContext ctx) {
1118         if (ctx.expresion_logica() != null) {
1119             return visit(ctx.expresion_logica());
1120         } else if (ctx.expresion_lista() != null) {
1121             return visit(ctx.expresion_lista());
1122         }
1123         return "";
1124     }
1125
1126     @Override
1127     public String visitExpresion_lista(EasyParser.Expresion_listaContext ctx) {
1128         return visit(ctx.lista_valores());
1129     }
1130

```

Estos métodos generan código para llamadas a funciones y manejo de expresiones:

- `visitLlamada_funcion`: Genera una llamada a función con sus argumentos.
- `visitArgumentos`: Genera una lista de argumentos separados por coma.
- `visitExpresion`: Devuelve el resultado de una expresión lógica o una lista.
- `visitExpresion_lista`: Devuelve el resultado de una lista de valores.

```

1131     @Override
1132     public String visitExpresion_logica(EasyParser.Expresion_logicaContext ctx) {
1133         if (ctx.getChildCount() == 1) {
1134             return visit(ctx.expresion_relacional());
1135         } else {
1136             String left = visit(ctx.expresion_logica());
1137             String operator = ctx.children.get(1).getText();
1138             String right = visit(ctx.expresion_relacional());
1139
1140             String cppOperator = operator.equals(anObject:"&") ? "&&" : "||";
1141             return "(" + left + " " + cppOperator + " " + right + ")";
1142         }
1143     }
1144
1145     @Override
1146     public String visitExpresion_relacional(EasyParser.Expresion_relacionalContext ctx) {
1147         if (ctx.getChildCount() == 1) {
1148             return visit(ctx.expresion_aritmetica(i:0));
1149         } else {
1150             String left = visit(ctx.expresion_aritmetica(i:0));
1151             String operator = visit(ctx.operador_relacional());
1152             String right = visit(ctx.expresion_aritmetica(i:1));
1153             return "(" + left + " " + operator + " " + right + ")";
1154         }
1155     }
1156
1157     @Override
1158     public String visitOperador_relacional(EasyParser.Operador_relacionalContext ctx) {
1159         String op = ctx.getText();
1160         switch (op) {
1161             case "::": return "==";
1162             case ">": return ">=";
1163             case "<": return "<=";
1164             case ";!": return "!=";
1165             default: return op;
1166         }
1167     }
1168 }
```

Estos métodos generan código para expresiones lógicas y relacionales:

- `visitExpresion_logica`: Traduce operaciones lógicas (como `&&`, `||`) entre expresiones.
- `visitExpresion_relacional`: Traduce comparaciones (como `>`, `<=`) entre expresiones aritméticas.
- `visitOperador_relacional`: Convierte operadores del lenguaje fuente a sus equivalentes en C++ (`=: a ==, >: a >=`, etc.).

```

1169     @Override
1170     public String visitExpresion_aritmetica(EasyParser.Expresion_aritmeticaContext ctx) {
1171         if (ctx.getChildCount() == 1) {
1172             return visit(ctx.termino());
1173         } else {
1174             String left = visit(ctx.expresion_aritmetica());
1175             String operator = ctx.children.get(1).getText();
1176             String right = visit(ctx.termino());
1177             return "(" + left + " " + operator + " " + right + ")";
1178         }
1179     }
1180
1181     @Override
1182     public String visitTermino(EasyParser.TerminoContext ctx) {
1183         if (ctx.getChildCount() == 1) {
1184             return visit(ctx.factor());
1185         } else {
1186             String left = visit(ctx.termino());
1187             String operator = ctx.children.get(1).getText();
1188             String right = visit(ctx.factor());
1189             return "(" + left + " " + operator + " " + right + ")";
1190         }
1191     }
1192 }
```



Estos métodos traducen expresiones matemáticas:

- `visitExpresion_aritmetica`: Genera código para sumas y restas, combinando expresiones aritméticas y términos.
- `visitTermino`: Genera código para multiplicaciones o divisiones, combinando términos y factores.

```
1193     @Override
1194     public String visitFactor(EasyParser.FactorContext ctx) {
1195         if (ctx.llamada_funcion() != null) {
1196             return generateFunctionCall(ctx.llamada_funcion()).replace(target:";", replacement:"");
1197         }
1198         if (ctx.llamada_metodo() != null) {
1199             return generateMethodCall(ctx.llamada_metodo()).replace(target:";", replacement:"");
1200         }
1201         if (ctx.LITENTERO() != null) {
1202             return ctx.LITENTERO().getText();
1203         } else if (ctx.LITFLOATANTE() != null) {
1204             return ctx.LITFLOATANTE().getText();
1205         } else if (ctx.VERDADERO() != null) {
1206             return "true";
1207         } else if (ctx.FALSO() != null) {
1208             return "false";
1209         } else if (ctx.LITERALCADENA() != null) {
1210             return ctx.LITERALCADENA().getText();
1211         } else if (ctx.ID() != null) {
1212             if (ctx.CORCHETEIZQ() != null) {
1213                 String index = visit(ctx.expresion());
1214                 return ctx.ID().getText() + "[" + index + "]";
1215             } else {
1216                 return ctx.ID().getText();
1217             }
1218         } else if (ctx.llamada_funcion() != null) {
1219             return visit(ctx.llamada_funcion());
1220         } else if (ctx.PARIZQ() != null) {
1221             return "(" + visit(ctx.expresion()) + ")";
1222         } else if (ctx.RESTA() != null) {
1223             return "-" + visit(ctx.factor());
1224         }
1225     }
1226 }
1227 }
```

Este método interpreta factores dentro de expresiones. Devuelve funciones, métodos, literales (números, booleanos, cadenas), variables, arreglos, expresiones entre paréntesis y expresiones negadas. Devuelve la representación en texto de cada uno.





Pruebas

Comandos utilizados en Windows para compilar correctamente el archivo .g4, los archivos .java y los creados en el lenguaje Easy así como también el archivo de c++

```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL
powershell - E + ▾ [I]

● PS C:\Users\Yuvini\Downloads\Easy6.0\E> java -jar antlr-4.13.1-complete.jar -visitor Easy.g4
● PS C:\Users\Yuvini\Downloads\Easy6.0\E> javac *.java
● PS C:\Users\Yuvini\Downloads\Easy6.0\E> java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\cicloPara.ec
Código C++ generado en ./pruebas/cicloPara.cpp
● PS C:\Users\Yuvini\Downloads\Easy6.0\E> g++ ./pruebas/cicloPara.cpp -o para
● PS C:\Users\Yuvini\Downloads\Easy6.0\E> .\para
La suma es:
10
○ PS C:\Users\Yuvini\Downloads\Easy6.0\E> [ ]
```

1. Generar el analizador con ANTLR

java -jar antlr-4.13.1-complete.jar -visitor Easy.g4

¿Qué hace?

Este comando ejecuta ANTLR para **generar automáticamente el lexer, el parser y el visitor** a partir del archivo de gramática Easy.g4.

Componentes generados:

- EasyLexer.java (análisis léxico).
- EasyParser.java (análisis sintáctico).
- EasyBaseVisitor.java y EasyVisitor.java (para recorrer el árbol sintáctico).

¿Qué significa el flag -visitor?

Le dice a ANTLR que también genere las clases necesarias para aplicar el patrón de diseño **Visitor**, que es útil para recorrer el árbol de análisis y realizar acciones como generar código.

2. Compilar todos los archivos .java

javac *.java

¿Qué hace?

Este comando usa el compilador de Java para **compilar todos los archivos .java** del directorio actual, incluyendo los generados por ANTLR (EasyLexer.java, EasyParser.java, etc.) y tus propios archivos como Main.java y CodeGenerator.java.





Resultado:

Se generan los archivos .class correspondientes, que son ejecutables por la máquina virtual de Java (JVM).

3. Ejecutar el compilador Easy

```
java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\cicloPara.ec
```

¿Qué hace?

Este comando ejecuta tu programa Java (Main) y le pasa como argumento un archivo fuente escrito en tu lenguaje Easy: cicloPara.ec.

Explicación de las partes:

- `-cp "antlr-4.13.1-complete.jar;."`
Especifica el *classpath*, es decir, los recursos que Java debe cargar. Aquí estás diciendo: “Carga antlr-4.13.1-complete.jar y también el directorio actual (.) donde están los .class compilados”.
- `Main`
Es la clase principal que se ejecuta.
- `.\pruebas\cicloPara.ec`
Es el archivo escrito en Easy que quieras traducir.

Resultado:

Si todo está correcto, se genera un archivo cicloPara.cpp que contiene el código equivalente en C++.

4. Compilar el código C++ generado

```
g++ .\pruebas\cicloPara.cpp -o para
```

¿Qué hace?

Usa el compilador g++ (de C++) para **compilar el archivo generado .cpp** y crear un ejecutable llamado para.

Significado de las opciones:

- `.\pruebas\cicloPara.cpp` → Código fuente en C++ que quieres compilar.
- `-o para` → El nombre del archivo ejecutable de salida.



5. Ejecutar el programa compilado

.\para

¿Qué hace?

Ejecuta el archivo binario para que fue generado en el paso anterior. Este archivo contiene la lógica que originalmente fue escrita en Easy y traducida a C++.

Programa 1

Código Entrada

```
E > pruebas > Aritmeticas.ec
 1  inicio []
 2      entero num1;
 3      entero num2;
 4      entero resultado;
 5      flotante division;
 6
 7      imprimir("Ingrese el primer número: ");
 8      leer(num1);
 9
10     imprimir("Ingrese el segundo número: ");
11     leer(num2);
12
13     /* Operaciones adicionales */
14     imprimir("Resultado");
15     imprimir("=====");
16     imprimir("El resultado de la suma es:");
17
18     imprimir(num1 + num2);
19     imprimir("-----");
20     imprimir("El resultado de la resta es:");
21     imprimir(num1 - num2);
22     imprimir("-----");
23     imprimir("El resultado de la multiplicacion es:");
24     imprimir(num1 * num2);
25     imprimir("-----");
26     imprimir("El resultado de la division es:");
27     imprimir(num1 / num2);
28 }
29 fin
```





Este programa es una calculadora básica. El programa está estructurado dentro del bloque principal delimitado por las palabras clave inicio y fin. Su propósito es interactuar con el usuario para recibir dos números enteros, realizar operaciones aritméticas básicas directamente (sin el uso de funciones auxiliares) y mostrar los resultados en pantalla de manera ordenada.

Bloque principal de ejecución

Dentro del bloque inicio { ... }, el programa realiza las siguientes acciones:

Declaración de variables

Se definen las siguientes variables:

- num1 y num2: de tipo entero, para almacenar los números ingresados por el usuario.
- resultado: una variable entera no utilizada directamente en este fragmento.
- division: de tipo flotante, también declarada pero no usada explícitamente.

Entrada del usuario

El programa solicita al usuario ingresar dos valores numéricos:

1. Muestra un mensaje con imprimir pidiendo el primer número.
2. Usa leer(num1) para capturar la entrada del usuario y almacenarla en num1.
3. Repite el mismo proceso para num2.

Salida estructurada de resultados

Después de recibir las entradas, el programa imprime una sección titulada "**Resultado**", seguida de separadores visuales como "===== y -----" para organizar la información que se muestra al usuario.

Operaciones aritméticas realizadas directamente

Sin llamar funciones externas, el programa realiza directamente las siguientes operaciones sobre num1 y num2, imprimiendo los resultados en pantalla:

- num1 + num2: suma.
- num1 - num2: resta.



- num1 * num2: multiplicación.
- num1 / num2: división.

Cada resultado se acompaña de un mensaje que lo identifica, seguido de una línea divisoria para mantener el formato limpio y entendible.

Nota sobre la división: al dividir dos enteros directamente, el lenguaje Easy (si se traduce a C++ sin casting) truncará los decimales y devolverá solo la parte entera del cociente. Para obtener un resultado con decimales, sería necesario convertir explícitamente los operandos a flotante.

Salida

```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL powershell - e + □

● PS C:\Users\Yuvini\Downloads\Easy6.0> cd e
● PS C:\Users\Yuvini\Downloads\Easy6.0\> java -jar antlr-4.13.1-complete.jar -visitor Easy.g4
● PS C:\Users\Yuvini\Downloads\Easy6.0\> javac *.java
● PS C:\Users\Yuvini\Downloads\Easy6.0\> java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\Aritmeticas.ec
Código C++ generado en .\pruebas\Aritmeticas.cpp
● PS C:\Users\Yuvini\Downloads\Easy6.0\> g++ .\pruebas\Aritmeticas.cpp -o Aritmeticas
● PS C:\Users\Yuvini\Downloads\Easy6.0\> .\Aritmeticas
Ingrese el primer numero:
8
Ingrese el segundo numero:
4
Resultado
=====
El resultado de la suma es:
12
-----
El resultado de la resta es:
4
-----
El resultado de la multiplicacion es:
32
-----
El resultado de la division es:
2
```





Programa 2

Código Entrada

```
1
2     funcion entero multiplicar(entero a, entero b) {
3         |     retornar a * b;
4     }
5     inicio {
6
7
8         entero resultado : multiplicar(4, 5);
9         imprimir("Resultado:");
10        imprimir(resultado);
11    }
12    fin
13 }
```

El código muestra un pequeño programa escrito en el lenguaje **Easy**, el cual define una función llamada multiplicar que toma dos parámetros enteros y devuelve su producto. Luego, dentro del bloque principal de ejecución (inicio { ... }), se llama a esta función con dos valores constantes (4 y 5), se almacena el resultado en una variable y se imprime dicho resultado en la pantalla.

Este programa ejemplifica el uso básico de funciones definidas por el usuario, la asignación de valores mediante expresiones, y la salida de datos al usuario usando imprimir.

Detalle del funcionamiento

- **Definición de función personalizada:**

Se define una función llamada multiplicar que recibe dos parámetros de tipo entero (a y b) y retorna su multiplicación utilizando la palabra clave retornar. Esta función encapsula una operación aritmética simple que puede ser reutilizada en distintas partes del programa.

- **Bloque principal del programa:**

Dentro del bloque inicio { ... }, se declara una variable llamada resultado de tipo entero, la cual se inicializa con el valor que retorna la función multiplicar al ser invocada con los argumentos



literales 4 y 5. Es decir, se realiza la operación $4 * 5$ y se almacena el valor 20.

- **Salida de resultados:**

Posteriormente, el programa utiliza dos instrucciones imprimir. La primera imprime el texto "Resultado:" como título o indicación, y la segunda muestra el valor contenido en la variable resultado, es decir, 20.

- **Estructura y cierre del programa:**

El programa está delimitado por las palabras clave inicio y fin, que en Easy marcan el comienzo y fin del bloque de ejecución principal, similar a main en C++ o Java.

Salida

The screenshot shows the Easy IDE interface. At the top, there's a file tree with 'E > pruebas > funcion.ec'. Below it is the code editor with the following pseudocode:

```
1
2     funcion entero multiplicar(entero a, entero b) {
3         |     retornar a * b;
4     }
5     inicio {
6
7
8         entero resultado : multiplicar(4, 5);
9         imprimir("Resultado:");
10        imprimir(resultado);
11    }
12    fin
13
```

Below the code editor is a terminal window with the following output:

- PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL
- PS C:\Users\Yuvini\Downloads\Easy6.0\e> g++ .\pruebas\funcion.cpp
- PS C:\Users\Yuvini\Downloads\Easy6.0\e> .\funcion
- Resultado:
- 20
- PS C:\Users\Yuvini\Downloads\Easy6.0\e>





Programa 3

Código de Entrada

```
E > pruebas > mientras.ec
 1  entero numeros[5];
 2
 3  funcion entero obtenerElemento(entero i) {
 4    |   retornar numeros[i];
 5  }
 6
 7  funcion entero sumar() {
 8    entero i : 0;
 9    entero suma : 0;
10    mientras (i <= 5) {
11      |   suma : suma + obtenerElemento(i);
12      |   i : i + 1;
13    }
14    |   retornar suma;
15  }
16
17  inicio []
18    numeros[0] : 1;
19    numeros[1] : 2;
20    numeros[2] : 3;
21    numeros[3] : 4;
22    numeros[4] : 5;
23
24    entero resultado : sumar();
25    imprimir("Suma total:");
26    imprimir(resultado);
27  []
28  fin
29
```

Este programa escrito en el lenguaje **Easy** ilustra el uso de arreglos, funciones, ciclos y llamadas a funciones dentro de otras. El objetivo principal es almacenar cinco números en un arreglo, sumar todos sus elementos utilizando una función personalizada llamada `sumar`, y mostrar el resultado final en pantalla. También se define una función auxiliar llamada `obtenerElemento` para acceder a los elementos del arreglo.

Detalle del funcionamiento

- **Declaración del arreglo:**

Antes de cualquier función, se declara un arreglo global de enteros llamado `numeros` con



capacidad para 5 elementos. Esto permite su acceso desde cualquier parte del programa, incluyendo funciones.

- **Función obtenerElemento:**

Esta función recibe un índice *i* como parámetro y retorna el valor almacenado en la posición *i* del arreglo numeros. Se utiliza para encapsular el acceso a los elementos del arreglo, promoviendo un diseño más estructurado.

- **Función sumar:**

La función sumar tiene como propósito recorrer el arreglo y acumular la suma de sus elementos:

- Declara dos variables locales: *i* (inicializado en 0) y *suma* (inicializado en 0).
- Usa un ciclo mientras que se ejecuta mientras *i* sea menor que 5 (*i* <: 5).
- Dentro del ciclo, se llama a *obtenerElemento(i)* para acceder al valor actual del arreglo y se suma a la variable *suma*.
- El índice *i* se incrementa en cada iteración.
- Al final, se retorna el valor acumulado en *suma*.

- **Bloque principal inicio { ... }:**

Dentro del bloque principal:

- Se inicializan los cinco elementos del arreglo numeros con los valores del 1 al 5.
- Se declara una variable resultado que almacena el valor devuelto por la función *sumar()*.
- Finalmente, se imprime en pantalla el mensaje "Suma total:" seguido del resultado de la suma (que en este caso es 15).



- **Finalización del programa:**

El programa concluye con la palabra clave `fin`, que marca el cierre del bloque de ejecución principal.

Salida

```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL
powershell - e + v

20
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\mientras.ec
*** PROCESANDO DECLARACIÓN GLOBAL: enteronumeros[5];
*** ARRAY SIN INICIALIZACIÓN: int numeros[5];
Código C++ generado en .\pruebas\mientras.cpp
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> g++ .\pruebas\mientras.cpp -o mientras
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> .\mientras
Suma total:
15
diamond PS C:\Users\Yuvini\Downloads\Easy6.0\e> []
```

Programa 3

Código de Entrada

```
E > pruebas > condicion.ec
 1  inicio {
 2      entero numero;
 3
 4      imprimir("Ingrese un numero: ");
 5      leer(numero);
 6
 7      si (numero > 0) {
 8          imprimir("El numero es positivo");
 9      } sino {
10          si (numero < 0) {
11              imprimir("El numero es negativo");
12          } sino {
13              imprimir("El numero es cero");
14          }
15      }
16  }
17 fin
```





Este programa escrito en el lenguaje **Easy** permite al usuario ingresar un número entero desde la consola y luego determina si ese número es positivo, negativo o igual a cero. Utiliza estructuras condicionales (si y sino) para evaluar el valor ingresado y dar una respuesta adecuada.

Detalle del funcionamiento

- **Entrada del usuario:**

- Se declara una variable numero de tipo entero.
- Se solicita al usuario que ingrese un número con el mensaje: "Ingrese un numero: ".
- El valor ingresado se almacena en la variable numero mediante la instrucción leer(numero).

- **Estructura condicional anidada:**

- El programa evalúa si el número ingresado es **mayor que cero** (numero > 0). Si lo es, imprime: "El numero es positivo".
- Si no es mayor que cero, entra en el bloque sino donde evalúa si el número es **menor que cero** (numero < 0). Si esta condición se cumple, imprime: "El numero es negativo".
- Si tampoco es menor que cero (es decir, es exactamente cero), el bloque sino final imprime: "El numero es cero".



Programa 4

Código de Entrada

```
E > pruebas > ̄ cicloPara.ec
 1  inicio {
 2      entero suma : 0;
 3
 4      para [entero i : 0; i < 10; i : i + 1] {
 5          suma : suma + i;
 6      }
 7
 8      imprimir("La suma es:");
 9      imprimir(suma);
10  }
11  fin
```

Este programa en el lenguaje Easy calcula la suma de los números enteros del 0 al 9 utilizando una estructura de control repetitiva para (similar al bucle for en otros lenguajes). Al final, muestra el resultado de la suma en la consola.

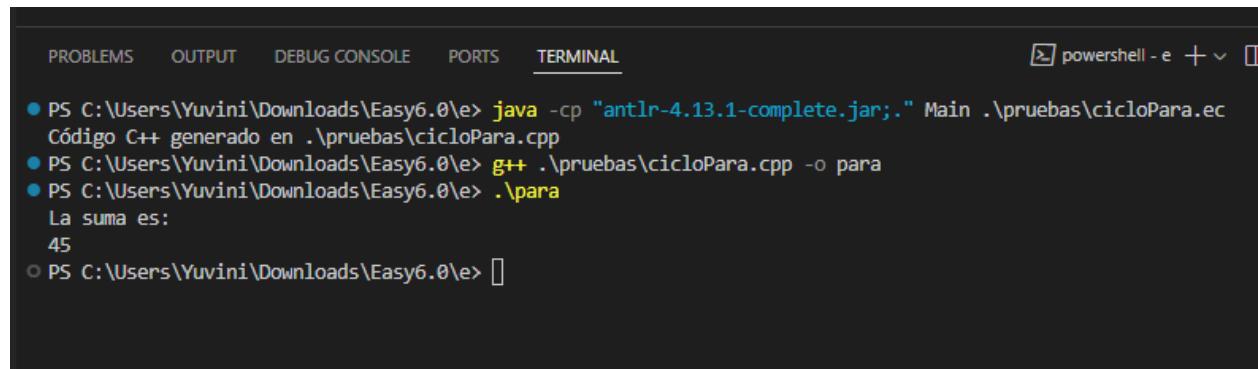
Detalle del funcionamiento

- Inicialización:
 - Se declara una variable suma de tipo entero e inicializa en 0. Esta variable servirá como acumulador para guardar el resultado de la suma.
- Estructura de control para:
 - Se utiliza un ciclo para con tres partes:
 - Inicialización: entero i : 0 → se declara una variable i e inicia en 0.
 - Condición: i < 10 → el ciclo se repite mientras i sea menor que 10.
 - Incremento: i : i + 1 → en cada iteración, i se incrementa en 1.
- Cuerpo del ciclo:
 - En cada iteración, el valor actual de i se suma a suma. Esto hace que suma acumule los valores de i desde 0 hasta 9.
- Salida de resultados:
 - Despues del ciclo, se imprime el mensaje "La suma es:".



- Luego, se imprime el valor final de suma, que será 45 (la suma de los enteros del 0 al 9).

Salida



```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL
powershell -e + v □

● PS C:\Users\Yuvini\Downloads\Easy6.0\e> java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\cicloPara.ec
Código C++ generado en .\pruebas\cicloPara.cpp
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> g++ .\pruebas\cicloPara.cpp -o para
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> .\para
La suma es:
45
○ PS C:\Users\Yuvini\Downloads\Easy6.0\e> □
```

Programa 5

Código Entrada

```
1  entero numeros[5] : {1, 2, 3, 4, 5};
2
3  función entero sumarArreglo() {
4      entero suma : 0;
5      entero i : 0;
6      mientras (i < 5) {
7          suma : suma + numeros[i];
8          i : i + 1;
9      }
10     retornar suma;
11 }
12
13 inicio {
14     entero resultado : sumarArreglo();
15     imprimir("Suma total:");
16     imprimir(resultado);
17 }
18 fin
```





Este programa en el lenguaje **Easy** está diseñado para calcular la suma de los elementos de un arreglo de enteros. Usa una función llamada `sumarArreglo()` que recorre el arreglo con un ciclo mientras y acumula los valores. Luego, en el bloque principal, imprime la suma total.

Descripción general

- Se declara y se inicializa un arreglo llamado `numeros` con cinco valores: 1, 2, 3, 4, 5.
- La función `sumarArreglo()` recorre el arreglo usando un índice y suma cada elemento.
- El resultado de la suma se guarda en una variable `resultado` dentro del bloque principal (inicio).
- Finalmente, se muestra en pantalla el texto “**Suma total:**” seguido del resultado.

Declaración del arreglo

```
entero numeros[5] : {1, 2, 3, 4, 5};
```

Se crea un arreglo de 5 elementos de tipo entero, y se inicializa directamente con los valores mencionados.

Función `sumarArreglo()`

- Declara una variable `suma` para acumular el total y un índice `i` que empieza en 0.
- El ciclo `mientras (i < 5)` recorre el arreglo desde la posición 0 hasta la 4.
- En cada vuelta del ciclo, se suma `numeros[i]` a `suma`, y luego `i` se incrementa en 1.
- Cuando finaliza el ciclo, se retorna el valor total acumulado.

Bloque principal inicio

- Se declara la variable `resultado` y se le asigna el valor que devuelve `sumarArreglo()`.
- Luego, se imprimen en pantalla dos mensajes: el primero con el texto "Suma total:", y el segundo con el valor almacenado en `resultado`.



Código Salida

```
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\arreglos.ec
*** PROCESANDO DECLARACIÓN GLOBAL: enteronumeros[5]:{1,2,3,4,5};
*** ARRAY CON INICIALIZACIÓN: int numeros[5] = {1,2,3,4,5};
Código C++ generado en .\pruebas\arreglos.cpp
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> g++ .\pruebas\arreglos.cpp -o arreglo
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> .\arreglo
Suma total:
● 15
○ PS C:\Users\Yuvini\Downloads\Easy6.0\e> []
```

Programa 6

Código Entrada

```
E > pruebas > ≡ clases.ec
 1  clase Persona {
 2      cadena nombre;
 3      entero edad;
 4
 5      metodo obtenerNombre(cadena n) {
 6          |   nombre : n;
 7      }
 8
 9      metodo obtenerEdad(entero e) {
10          |   edad : e;
11      }
12
13      metodo mostrarDatos() {
14          |   imprimir(nombre);
15          |   imprimir(edad);
16      }
17  }
18
19  inicio [
20      Persona p1;
21
22      p1.obtenerNombre("Juan");
23      p1.obtenerEdad(25);
24      p1.mostrarDatos();
25  ]
26  fin
```





Este programa en el lenguaje **Easy** demuestra el uso de **clases, métodos y objetos** para trabajar con estructuras orientadas a objetos. En este caso, se define una clase Persona que encapsula dos atributos (nombre y edad), y tres métodos que permiten asignar y mostrar dichos datos. Luego, en el bloque principal (inicio), se crea una instancia de la clase y se interactúa con ella utilizando los métodos definidos.

Descripción general

- Se define una clase llamada Persona con dos atributos: nombre (cadena) y edad (entero).
- Se implementan tres métodos:
 - obtenerNombre(n): asigna un valor al atributo nombre.
 - obtenerEdad(e): asigna un valor al atributo edad.
 - mostrarDatos(): imprime en pantalla el nombre y la edad.
- En el bloque inicio, se crea un objeto p1 de tipo Persona.
- Se invocan los métodos para asignar nombre y edad al objeto.
- Finalmente, se muestra la información del objeto en pantalla.

Funcionamiento detallado

Clase Persona

La clase actúa como una plantilla para crear personas. Sus elementos son:

- **Atributos:**
 - nombre: almacena el nombre de la persona.
 - edad: almacena la edad de la persona.
- **Métodos:**
 - obtenerNombre(cadena n): recibe una cadena como parámetro y la asigna al atributo nombre.
 - obtenerEdad(entero e): recibe un número entero y lo asigna al atributo edad.
 - mostrarDatos(): imprime los valores de nombre y edad.

Bloque inicio



- Se crea una variable p1 de tipo Persona.
- Se llama al método p1.obtenerNombre("Juan"), asignando el nombre "Juan" al atributo correspondiente.
- Luego, con p1.obtenerEdad(25), se asigna la edad 25.
- Finalmente, p1.mostrarDatos() muestra ambos valores en pantalla.

Salida

```
PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL powershell - e + □

15
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\clases.ec
Código C++ generado en .\pruebas\clases.cpp
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> g++ .\pruebas\clases.cpp -o clase
● PS C:\Users\Yuvini\Downloads\Easy6.0\e> ./clase
Juan
25
○ PS C:\Users\Yuvini\Downloads\Easy6.0\e> []
```





IV. Problemas surgidos durante la realización del Proyecto

1. Funcionamiento de clases y objetos

Problema:

Implementar el soporte para clases y objetos fue uno de los aspectos más complejos. Se presentaron dificultades al momento de manejar el contexto de los atributos y métodos dentro del árbol de análisis.

Detalles:

- Fue necesario diseñar una estructura de almacenamiento adecuada para los atributos y métodos de cada instancia.
- El manejo de llamadas a métodos a través del operador de punto (objeto.metodo()) requería una correcta verificación del tipo y del alcance.
- La generación de código para métodos tuvo que incluir referencias al objeto actual y a sus atributos, lo cual complicó la fase de generación intermedia.

The screenshot shows an IDE interface with several tabs open. The main tab displays Java code for a `CodeGenerator` class. The code includes methods for generating code based on visitor patterns and handling different data types. Below the code editor is a terminal window showing the output of a build process. The terminal output indicates multiple errors, primarily related to undeclared variables like `estudiante1` and `null`. The IDE's sidebar shows various project files and a timeline.

```
private boolean inClass = false;
private boolean inPrivateSection = false;

private void addLine(String line) {
    if (indentLevel == 0) {
        code.add(line);
    } else {
        String indent = ".repeat(indentLevel)";
        code.add(indent + line);
    }
}

private String mapType(String easyType) {
    switch (easyType) {
        case "entero": return "int";
        case "cadena": return "string";
        case "booleano": return "bool";
        case "flotante": return "float";
        default: return easyType; // Para tipos personalizados
    }
}

@Override
public String visitPrograma(EasyParser.ProgramaContext ctx) {
    StringBuilder code = new StringBuilder();
    // Incluir librería estándar
    code.append("#include <iostream>\n");
    code.append("#include <string>\n");
    code.append("using namespace std;\n\n");

    // Procesar definiciones de clases antes del main
    for (ParseTree child : ctx.children) {
        String result = visit(child);
        if (result != null) {
            code.append(result);
        }
    }

    estudiante1.mostrarDatos();
    Estudiante
pruebas/classes.cpp: In function 'int main()':
pruebas/classes.cpp:29:1: error: 'null' was not declared in this scope
29 | null
| ^
pruebas/classes.cpp:31:1: error: 'estudiante1' was not declared in this scope; did you mean 'Estudiante'?
31 | estudiante1.mostrarDatos();
| ^~~~~~
| Estudiante
```



2. Llamado de funciones

Problema:

Los errores comunes incluían referencias a funciones no definidas o problemas en la validación del número y tipo de parámetros.

Detalles:

- La tabla de símbolos debió ser extendida para incluir funciones y su respectiva firma.
- En la visita del árbol, la validación semántica requería verificar la cantidad y tipos de argumentos pasados a una función.
- Algunos errores no eran detectados hasta tiempo de ejecución, lo cual complicó la depuración inicial.

The screenshot shows the Eclipse IDE interface. On the left is the 'EXPLORER' view showing various Java files. In the center, there are two tabs: 'output.cpp' and 'arreglos.ec'. The 'output.cpp' tab contains C++ code that includes a function to sum an array of integers. The 'arreglos.ec' tab contains the generated C++ code, which includes a global variable 'entero resultado', a function 'sumarArreglo()', and a main function that prints the sum. The terminal at the bottom shows the command 'java Main pruebas/arreglos.ec output.cpp' being run, followed by the output of the program which prints 'Suma total: 15'.

3. Ciclo para

Problema:

El ciclo para necesitó un tratamiento especial en cuanto a la estructura del parser y la generación del código, ya que combinaba declaración, condición y actualización en una sola línea.

Detalles:

- La sintaxis del para fue difícil de integrar con las reglas de la gramática ANTLR, especialmente en lo relacionado con el alcance de variables declaradas dentro del ciclo.
- La generación de código intermedio tuvo que representar correctamente el control del



flujo, asegurando la evaluación adecuada del inicio, condición y actualización.

- Algunos errores surgieron cuando se intentaba modificar la variable del ciclo dentro de su cuerpo.

4. Manejo de arreglos

Problema:

El acceso a elementos de arreglos y la inicialización de arreglos con múltiples valores causaron errores en la etapa semántica y en la generación de código.

Detalles:

- Fue difícil validar correctamente el rango de índices en tiempo de compilación.
- La asignación de listas de valores (`numeros[5] : {1, 2, 3, 4, 5};`) requería un procesamiento adicional en el visitor para ser traducida a código C++.
- Se detectaron errores cuando se intentaba acceder a índices no definidos o cuando los arreglos no estaban correctamente inicializados.

The screenshot shows the Eclipse IDE interface. On the left, the Project Explorer displays several Java and C++ files. The main workspace contains four code editors: `Easy.java`, `Main.java`, `output.cpp`, and `arreglos.ec`. The `arreglos.ec` editor contains the following EASY script:

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6      int numeros[5] = {1, 2, 3, 4, 5};
7      // Función anidada no soportada en C++
8      int resultado = sumarArreglo();
9      cout << "Suma total:" << endl;
10     cout << resultado << endl;
11     return 0;
12 }
```

The `output.cpp` editor contains the following C++ code:

```
1  inicio {
2      entero numeros[5] : {1, 2, 3, 4, 5};
3
4      funcion entero sumarArreglo() {
5          entero suma : 0;
6          entero i : 0;
7          mientras (i < 5) {
8              suma : suma + numeros[i];
9              i : i + 1;
10         }
11         retornar suma;
12     }
13
14     entero resultado : sumarArreglo();
15     imprimir("Suma total:");
16     imprimir(resultado);
17 }
18 fin
```

The terminal window at the bottom shows the command-line output of the compilation process:

```
ury@ury-Virtual-Platform:~/Documentos/AnalizadorEasy$ java Main pruebas/arreglos.ec output.cpp
Código C++ generado en: output.cpp
ury@ury-Virtual-Platform:~/Documentos/AnalizadorEasy$ g++ -o programa output.cpp && ./programa
output.cpp:8:21: error: 'sumarArreglo' was not declared in this scope
          int resultado = sumarArreglo();
```

5. Expresiones complejas

Problema:

Evaluar expresiones que combinaban múltiples operaciones aritméticas y condicionales causó errores de precedencia o ambigüedad en el análisis sintáctico.



Detalles:

- Fue necesario ajustar cuidadosamente las prioridades y asociaciones en la gramática para que se evaluaran correctamente.
- Algunos errores no se mostraban como errores de sintaxis, pero causaban resultados incorrectos en la ejecución.

The screenshot shows the Visual Studio Code interface with the following details:

- Project Explorer:** Shows files under the 'ANALIZADOREASY' folder, including 'arreglos.ec' which is currently selected.
- Code Editor:** Displays the content of 'arreglos.ec' (C++ code) and 'pruebas > arreglos.ec' (Java code).
- Terminal:** Shows the command-line output of a build process:

```
ury@ury-VMware-Virtual-Platform:~/Documentos/AnalizadorEasy$ java Main pruebas/arreglos.ec output.cpp
Código C++ generado en: output.cpp
ury@ury-VMware-Virtual-Platform:~/Documentos/AnalizadorEasy$ g++ -o programa output.cpp && ./programa
output.cpp: In function 'int main()':
output.cpp:8:21: error: 'sumarArreglo' was not declared in this scope
    8 |     int resultado = sumarArreglo();
```



V. CONCLUSIÓN

El desarrollo del compilador **EasyC** ha representado una experiencia integral de aprendizaje que ha permitido poner en práctica los conocimientos teóricos adquiridos a lo largo de la clase. Desde el diseño del análisis léxico con Flex, pasando por la construcción del análisis sintáctico con Bison, hasta la implementación final con ANTLR 4, el proyecto ha recorrido cada una de las etapas fundamentales en la construcción de un compilador real.

A través de EasyC, se buscó simplificar la sintaxis y el funcionamiento del lenguaje C, manteniendo su estructura lógica, pero adaptándola a un entorno más didáctico y accesible. Esta aproximación facilitó la comprensión de conceptos clave como la definición de tokens, reglas gramaticales, manejo de errores, y generación de árboles sintácticos, así como la integración de distintas herramientas en múltiples entornos operativos (Ubuntu, Windows y macOS).

Durante el proceso, se enfrentaron diversos retos técnicos, desde conflictos gramaticales hasta diferencias de configuración entre sistemas operativos. Sin embargo, estos desafíos favorecieron el desarrollo de habilidades de resolución de problemas, trabajo en equipo y documentación técnica. Además, se logró estandarizar el entorno de trabajo entre los distintos sistemas, garantizando la portabilidad y funcionalidad del compilador en todos ellos.

EasyC no solo es un producto funcional, sino también el reflejo del proceso de maduración académica del equipo, consolidando una base sólida para futuros proyectos en lenguajes de programación, procesamiento de lenguajes formales o diseño de herramientas personalizadas.

Finalmente, este proyecto no solo demuestra el dominio de las herramientas y técnicas estudiadas, sino que también fortalece la capacidad de pensamiento computacional, abstracción de problemas, y visión estructurada del desarrollo de software a nivel de sistemas.

VI. REFERENCIA

- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Boston: Pearson.
- ANTLR. (s.f.). Obtenido de ANTLR: <https://www.antlr.org/about.html>
- Béjar Hernández, R. (s.f.). *Introducción a Flex y Bison*. Obtenido de https://webdiis.unizar.es/asignaturas/LGA/material_2004_2005/Intro_Flex_Bison.pdf
- Cerúlea, K. (26 de Septiembre de 2024). *Garu 99*. Obtenido de Garu 99: <https://www.guru99.com/es/compiler-design-phases-of-compiler.html>
- GeeksforGeeks. (4 de septiembre de 2024). Obtenido de <https://www.geeksforgeeks.org/compiler-design/code-optimization-in-compiler-design/>
- IBM. (05 de 12 de 2024). Obtenido de <https://www.ibm.com/docs/es/aix/7.2.0?topic=information-example-program-lex-yacc-programs>
- Parr, T. (2013). *The Definitive ANTLR 4 Reference*. Dallas: Pragmatic Bookshelf.
- RajpL, H. (30 de Agosto de 2020). *Medium*. Obtenido de ANTLR and code generation: ANTLR and code generation
- Tomassetti, G. (s.f.). *Instrumentos*. Obtenido de Listeners and Visitors: <https://tomassetti.me/listeners-and-visitors/>
- Yunlin, S., & Song, Y. Y. (2011). *Principios de los compiladores*.



VII. ANEXOS



Ilustración 1: Imagen Compilador EasyC



Ilustración 2: Imagen Lenguaje Easy

VIII. Compilación Multiplataforma

✓ Windows



A screenshot of a Windows desktop environment showing a code editor window for a C++ project named "EASY6.0". The code editor displays a file named "clases.ec" containing the following C++ code:

```
1 clase Persona {
2     cadena nombre;
3     entero edad;
4
5     metodo obtenerNombre(cadena n) {
6         nombre : n;
7     }
8
9     metodo obtenerEdad(entero e) {
10        edad : e;
11    }
12
13     metodo mostrarDatos() {
14         imprimir(nombre);
15         imprimir(edad);
16     }
17
18
19 inicio {
20     Persona p1;
```

The terminal window below shows the command-line steps to compile the code:

- PS C:\Users\Yuvini\Downloads\Easy6.0> java -cp "antlr-4.13.1-complete.jar;." Main .\pruebas\clases.ec
- Código C++ generado en .\pruebas\clases.cpp
- PS C:\Users\Yuvini\Downloads\Easy6.0> g++ .\pruebas\clases.cpp -o clase
- PS C:\Users\Yuvini\Downloads\Easy6.0> ./clase
- Juan
- 25

The desktop taskbar at the bottom includes icons for File Explorer, Task View, Taskbar settings, BLACKBOX Chat, Add Logs, CyberCode, Buscar, and several pinned application icons.





✓ Ubuntu

```
g++ prueba.cpp -o prueba
[...]
prueba: prueba.cpp:(.text+0x10): undefined reference to `main'
collect2: error: ld returned 1 exit status
[...]
```

```
Parse Tree Inspector
programa
  mdo
    {
      <-- bloque_elemento
      <-- bloque_elemento
      <-- bloque_elemento
      <-- bloque_elemento
    }
  fin
```

To return to your computer, move the mouse pointer outside or press Ctrl+Alt.



✓ macOS

The screenshot shows a Java code editor interface on a Mac OS X system. The left sidebar displays a project structure with files like Arithmeticas.cpp, Arreglo.exe, and CodeGenerator.class. The main window shows the CodeGenerator.java file with code for generating arithmetic expressions. The terminal below shows the execution of the program, which prompts for input and displays results for addition, subtraction, multiplication, and division.

```
J CodeGenerator.java 9+ x
J CodeGenerator.java > J CodeGenerator > visitFactor(FactorContext)
  public class CodeGenerator extends EasyBaseVisitor<String> {
    public String visitTermino(EasyParser.TerminoContext ctx) {
      ...
    }
    @Override
    public String visitFactor(EasyParser.FactorContext ctx) {
      ...
      if (ctx.llamada_funcion() != null) {
        return generateFunctionCall(ctx.llamada_funcion()).replace(target:"", replacement:"");
      }
      if (ctx.llamada_metodo() != null) {
        return generateMethodCall(ctx.llamada_metodo()).replace(target:"", replacement:"");
      }
      if (ctx.LITERALERO() != null) {
        return ctx.LITERALERO().getText();
      } else if (ctx.LITFLOTANTE() != null) {
        return ctx.LITFLOTANTE().getText();
      } else if (ctx.VERDADERO() != null) {
        return "true";
      } else if (ctx.FALSO() != null) {
        return "false";
      } else if (ctx.LITERALCADENA() != null) {
        return ctx.LITERALCADENA().getText();
      } else if (ctx.ID() != null) {
        ...
      }
    }
  }
PROBLEMAS 48 SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS
ferarrivillaga@MacBook-Pro-de-Fernando E % g++ pruebas/Aritmeticas.cpp -o programa
ferarrivillaga@MacBook-Pro-de-Fernando E % ./programa
Ingrese el primer número:
5
Ingrese el segundo número:
5
Resultado
El resultado de la suma es:
10
El resultado de la resta es:
0
El resultado de la multiplicación es:
25
El resultado de la división es:
1
El resultado de la división es:
1
ferarrivillaga@MacBook-Pro-de-Fernando E %
Lín. 1223, col. 46 Espacios: 4 UTF-8 LF ⓘ Java ⌂ Sesión cerrada
```

This screenshot shows the same Java code editor environment on a Mac OS X system, but with a different set of source files. The project structure includes files like EasyParser\$Expresion_relatacionalContext.class, Main.java, and Main.class. The terminal output shows the execution of the Main.java program, which performs arithmetic operations and prints results to the console.

```
J CodeGenerator.java 9+ x
J CodeGenerator.java > J CodeGenerator > visitFactor(FactorContext)
  public class CodeGenerator extends EasyBaseVisitor<String> {
    public String visitTermino(EasyParser.TerminoContext ctx) {
      ...
    }
    @Override
    public String visitFactor(EasyParser.FactorContext ctx) {
      ...
      if (ctx.llamada_funcion() != null) {
        return generateFunctionCall(ctx.llamada_funcion()).replace(target:"", replacement:"");
      }
      if (ctx.llamada_metodo() != null) {
        return generateMethodCall(ctx.llamada_metodo()).replace(target:"", replacement:"");
      }
      if (ctx.LITERALERO() != null) {
        return ctx.LITERALERO().getText();
      } else if (ctx.LITFLOTANTE() != null) {
        return ctx.LITFLOTANTE().getText();
      } else if (ctx.VERDADERO() != null) {
        return "true";
      } else if (ctx.FALSO() != null) {
        return "false";
      } else if (ctx.LITERALCADENA() != null) {
        return ctx.LITERALCADENA().getText();
      } else if (ctx.ID() != null) {
        ...
      }
    }
  }
PROBLEMAS 48 SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS
El resultado de la resta es:
0
El resultado de la multiplicación es:
25
El resultado de la división es:
1
ferarrivillaga@MacBook-Pro-de-Fernando E % g++ pruebas/funcion.cpp -o funcion
ferarrivillaga@MacBook-Pro-de-Fernando E % ./funcion
Resultado:
20
ferarrivillaga@MacBook-Pro-de-Fernando E %
Lín. 1223, col. 46 Espacios: 4 UTF-8 LF ⓘ Java ⌂ Sesión cerrada
```