# PL3
# PROCESADORES DEL LENGUAJE

Alejandro Fernandez Ambrós - 03248484X
Cristina Martínez Toledo - 09109126E
Álvaro Paniagua Cortijo - 06593675N

# PARTE 1

## 1.1 Introducción

Siguiendo la misma gramática tanto en el Lexer y el Parser de la práctica 2 a la hora de leer archivos csv, se ha mantenido casi en su totalidad para facilitar la lectura de archivos con extensión .csv

Las primeras diferencias vienen en la clase Parser con la creación de las etiquetas:

#JsonArch, la cual define todas y cada una de las filas del archivo csv para luego su conversión a JSON.
Esta regla a su vez contiene la regla lftln para obtener la primera línea del .csv, que se corresponde con la cabecera y la regla ln para el resto de las líneas del archivo .csv

#JsonLine, esta etiqueta define la estructura de una única línea de un archivo csv.

En campo se diferencia mediante las etiquetas txtLit y strLit para diferenciar entre una cadenas con o sin comillas, ya que a la hora de pasarlas al JSON, las cadenas con comillas exteriores se eliminan para evitar repeticiones.

Estas etiquetas son utilizadas para que el JsonVisitor definido pueda recorrer con mayor facilidad el árbol y generar el json más fácilmente.

## 1.2 EJ1Lexer

El lexer creado debe aceptar cualquier cadena de caracteres, incluyendo letras y números y aceptando espacios entre ellos. Las distintas cadenas se separarán mediante comas, punto y comas o barras, y las distintas líneas del CSV se dividirán mediante saltos de línea, que se representan como ‘\n’

Primero, el token TEXTO representa cualquier secuencia de caracteres que forme parte de un campo no citado y que no incluya ninguno de los elementos prohibidos: separadores de campos (,, ; o |), saltos de línea (\n o \r\n), ni comillas dobles ("). La regla obliga a que exista al menos un carácter válido, de modo que los campos vacíos no se reconocen como TEXTO, sino que se gestionan en la parte sintáctica. Esta definición permite capturar correctamente cadenas siempre que no interfieran con la estructura del CSV.

Seguidamente, la gramática define el token STRING para capturar campos entre comillas dobles, común en archivos CSV avanzados. La regla establece que el campo debe comenzar y terminar con comillas ("). Dentro del contenido se permite cualquier secuencia de caracteres excepto comillas simples, solo dobles (""). De este modo, expresiones como "hola", "hola

""mundo""" o "123;456" se interpretan como un único campo, independientemente de los caracteres internos.

El token SEPARADOR capta cualquiera de los caracteres utilizados para dividir los campos dentro de una misma fila: la coma ,, el punto y coma ; o la barra vertical |.

Para reconocer el final de una fila se define el token SALTO_DE_LINEA, que contempla el salto de línea típico de sistemas UNIX (\n) y el formato clásico de Windows (\r\n). Esta regla asegura que las filas se delimiten coherentemente incluso cuando provienen de plataformas distintas.

Finalmente, el token ESPACIO reconoce bloques de espacios y tabulaciones, que son ignorados mediante la instrucción -> skip.

## 1.3 EJ1Parser

En el parser, primero cogemos las reglas producidas por el lexer, con: options { tokenVocab=EJ1Lexer; }. Tras esto, definimos las distintas producciones. Las reglas son:

**archivo : fila+ EOF;**
Primero, la gramática parte del supuesto de que un archivo CSV está compuesto por una o más filas, organizadas secuencialmente y terminadas siempre por un salto de línea. Por ello, la regla inicial “archivo”  permite una lista no vacía de filas, finalizando con un “end of file”

**filas : lftl=fila (SALTO_DE_LINEA ln=fila)* (SALTO_DE_LINEA)?  #JsonArch;**
La regla filas, obliga a que mínimo tiene que existir una fila inicial que sería el header del json, y estas estarían seguidas de más líneas separadas por saltos de líneas. Además, la regla **SALTO_DE_LINEA?** permite que la última fila tenga o no un salto de línea. Por ejemplo:
L1 /n L2 /n L3  donde L representa una línea de la regla **fila**

**fila: campo? (SEPARADOR campo?)*  #JsonLine;**
Cada fila se define mediante la regla fila, que consiste en un primer campo posiblemente vacio, seguido opcionalmente de múltiples repeticiones de separadores y campos que podrían estar vacíos, y finalizando siempre con un salto de línea. Por ejemplo: Procesadores,del;lenguaje|pl2

**campo : txtLit=TEXTO | strLit=STRING | ;**
La regla campo permite tres posibilidades: un campo de texto sin comillas (TEXTO), un campo entre comillas (STRING), o un campo vacío, lo que permite reconocer correctamente situaciones frecuentes en archivos CSV tales como:
a,,b (campo vacío entre separadores)
; "hola"; ; "adiós"
Esto permite representar filas en las que algunos campos están deliberadamente en blanco, común en datos incompletos o en archivos generados por sistemas externos.

## 1.4 JsonVisitor

Mediante esta clase, se realiza la conversión mediante visitors del árbol generado por el Parser a un archivo json que se le pasará a la función main para guardarlo en un archivo de texto json.txt para su análisis.

Esta clase contiene los siguientes métodos para poder realizar el paso de árbol a json:

### getJSON():
Esta función que es la llamada por el método main, es la encargada de devolver el árbol generado en formato String tras visitarlo en su totalidad y guardarlo en la variable jsonBuilder.

### visitArchivo(EJ1Parser.ArchivoContext tree):
Esta función va recorriendo el árbol proporcionado por la función main, este cuando termina de leer todas las filas elimina la última coma y salto de línea que permiten separar los distintos componentes del json, esto se realiza debido a que cuando las filas se terminan de generar en visitJsonArch, añaden una coma y salto de linea para preparar el comienzo de la siguiente línea a visitar.

### visitJsonLine(EJ1Parser.JsonLineContext ctx)
Esta función es la encargada de ir pasando cada línea a su correspondiente formato json.

Mediante la regla del parser filas, se sabe que la primera línea corresponde al header del archivo csv. Por lo tanto tras obtener esta primera línea mediante la comprobación de los campos Texto y String, esta no se almacena como un objeto json, esta se guarda en la variable header para poder crear las siguientes filas con el nombre del campo correspondiente. Tras esto, el número de líneas leídas se incrementa en una unidad, para indicar que ya no es el header.

Posteriormente, como las siguientes líneas leídas ya no son la de header, se procede a guardar su contenido en formato json. Sin embargo, como se puede dar que una línea no contenga los campos suficientes, se calcula el mínimo de campos a generar entre el número de headers y el número de campos. Posteriormente, ese número será el total de campos de este objeto. Tras crear esta línea se sigue incrementado las líneas totales.

## 1.5 EJ1_Main

La clase EJ1_Main contiene el método main, encargado de ejecutar el proceso completo de análisis de un archivo CSV utilizando ANTLR. Este programa toma un archivo de entrada en formato CSV, lo analiza para construir su árbol sintáctico y finalmente guarda una representación textual de dicho árbol en un archivo de salida.

Al inicio, se importan las librerías necesarias para trabajar con ANTLR, incluyendo la creación del lexer, el parser y el recorrido del árbol sintáctico. También se importan las clases de java.io y java.nio.charset para gestionar archivos y garantizar que se usen caracteres en formato UTF-8.

En el método main, lo primero que hace el programa es verificar los argumentos de entrada. Se espera que el usuario proporcione exactamente dos rutas: el archivo CSV de entrada y la ruta del archivo de texto de salida. Si no se cumplen estas condiciones, el programa muestra un mensaje de error indicando el uso correcto y se detiene. Esto asegura que el programa se ejecute solo con los parámetros adecuados.

Una vez verificados los argumentos, el programa lee el archivo CSV mediante CharStreams.fromFileName, que carga el contenido del archivo en un CharStream. Este se pasa luego al lexer (EJ1Lexer), una clase generada automáticamente por ANTLR a partir de la gramática del CSV. El lexer se encarga de dividir el texto en unidades de informacion llamadas tokens, como palabras, comas o saltos de línea, que representan los elementos básicos del lenguaje CSV.

Posteriormente, los tokens producidos se pasan al parser (EJ1Parser) a través del canal de tokens creado con CommonTokenStream. El parser utiliza las reglas de la gramática para construir el AST que representa la estructura jerárquica del archivo CSV.

A continuación tras haber construido el árbol, este se le pasa al JsonVisitor mediante la llamada a la función visitArchivo, donde el propio visitor será el encargado de generar el archivo json.

Finalmente, el programa guarda el resultado en un archivo de salida usando un objeto PrintWriter. Se escribe el texto generado en formato UTF-8 para asegurar la compatibilidad con caracteres especiales. Gracias al bloque try-with-resources, el archivo se cierra automáticamente una vez terminada la escritura. Al final del proceso, el programa muestra un mensaje por consola indicando la ubicación del archivo donde se ha guardado el json generado.

# PARTE 2

## Introducción

Los ejemplos en la carpeta Parte2 son pequeñas clases Jasmin que muestran patrones básicos de programación en bytecode: métodos vacíos, impresiones por consola, operaciones aritméticas, control de flujo, bucles y llamadas a métodos (con y sin parámetros y con retorno). Cada archivo contiene la definición de la clase, el método main y, en algunos casos, métodos auxiliares. A continuación se describe cada ejercicio detallando la intención del código, la secuencia de instrucciones relevantes y el estado de la pila y de las variables locales en los momentos clave.

## Ejercicios:

### Ex01 Vacio

Este ejercicio presenta la plantilla mínima de una clase y su método main en Jasmin, sin ejecutar ninguna operación visible. La clase se declara pública y el main reserva recursos con .limit stack 1 y .limit locals 1 por claridad didáctica, aunque no son estrictamente necesarios aquí. Al inicio del main la pila está vacía y la variable local 0 contiene la referencia al array de argumentos por convención, aunque nunca se usa. La única instrucción activa es return, que termina el método y devuelve el control a la JVM.

### Ex02 PrintString

En este ejemplo se muestra cómo imprimir una cadena en consola usando System.out.println. Primero se reserva espacio con .limit stack 2 (uno para la referencia a PrintStream y otro para la String) y .limit locals 1. La secuencia principal carga System.out sobre la pila mediante getstatic, después empuja la cadena literal con ldc "Hello there". En ese momento la pila contiene, de abajo a arriba, la referencia a PrintStream y la String. La instrucción invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V consume ambas entradas y provoca la impresión en consola; a continuación return finaliza el main.

### Ex03 Multiply

Aquí se demuestra el manejo de enteros y operaciones aritméticas en la pila. Con .limit stack 4 y .limit locals 1 se deja margen para el PrintStream y varios enteros temporales. La rutina coloca System.out en la pila y carga tres constantes enteras con ldc 6, ldc 7, ldc 7. Tras esas cargas, la pila contiene PrintStream, 6, 7, 7. La primera imul multiplica los dos operandos superiores (7 * 7 → 49), dejando PrintStream, 6, 49. La segunda imul multiplica 6 * 49 → 294 y deja PrintStream, 294. Finalmente, invokevirtual println(I)V consume el PrintStream y el entero, imprimiendo 294.

## Ex04 PrintBoolean

Este ejercicio evalúa una comparación y convierte el resultado en un booleano imprimible. Se usan .limit stack 3 y .limit locals 1. El código carga 5 y 3 con ldc y aplica if_icmple LFALSE para comprobar si 5 ≤ 3; como no se cumple, salta la etiqueta de falso y ejecuta iconst_1 para empujar el valor entero 1 que representa true. Si la comparación fallara, se empujaría iconst_0 en la etiqueta LFALSE. Después se carga System.out con getstatic y se usa swap para dejar el orden correcto antes de llamar a println(Z), ya que println espera PrintStream debajo y el argumento encima. Con invokevirtual java/io/PrintStream/println(Z)V se imprime true o false según el flujo.

## Ex05 ConcatStringNumber

En este ejemplo se emplea StringBuilder para concatenar una cadena y un número. Se reserva .limit stack 4 y .limit locals 2. La secuencia crea un nuevo StringBuilder con new, duplica la referencia con dup para poder llamar al constructor con invokespecial <init>()V, y luego encadena llamadas append —primero con la cadena "La suma es: ", después con el entero 42—, aprovechando que append devuelve la misma instancia de StringBuilder. A continuación toString produce la cadena final. Para imprimir, se coloca System.out y se reorganiza la pila con swap de modo que la llamada println(Ljava/lang/String;)V reciba el PrintStream y el String en el orden correcto.

## Ex06 NestedIfs

Este archivo muestra condicionales anidados: primero se evalúa if (5 > 3) y, si es verdadero, se imprime "A" y dentro de ese bloque se comprueba if (2 < 4) para potencialmente imprimir "B". Con .limit stack 4 y .limit locals 2 se dejan recursos para las operaciones temporales. Las comprobaciones se implementan con ldc para cargar valores y if_icmple / if_icmpge para realizar saltos condicionales hacia etiquetas que delimitan los extremos de cada if. Tras cada comprobación que resulte verdadera se ejecuta un getstatic, ldc y invokevirtual println para mostrar la cadena correspondiente. Las etiquetas (LEND_IF1, LEND_INNER) organizan el flujo sin estructuras de bloque explícitas, pues en bytecode todo control se expresa mediante saltos y etiquetas.

## Ex07 ForLoop

Este ejemplo implementa un bucle equivalente a for (int i = 1; i <= 5; i++) e imprime el valor de i en cada iteración. Se definen .limit stack 3 y .limit locals 2 (local 0 para args, local 1 para i). La inicialización se realiza con iconst_1; istore_1. El cuerpo del bucle comienza en una etiqueta LOOP_START, donde se carga i con iload_1, se compara con 5 mediante if_icmpgt LOOP_END y, si la condición permite continuar, se invoca getstatic seguido de iload_1 y invokevirtual println(I)V. El incremento i++ se hace con la instrucción iinc 1 1, que modifica la variable local sin tocar la pila, lo que resulta más eficiente que cargar, sumar y almacenar manualmente.

## Ex08 WhileLoop

Aquí se muestra un bucle while que cuenta hacia abajo desde 3 hasta 1. Con .limit stack 3 y .limit locals 2, el programa inicializa n con ldc 3; istore_1 y entra en WHILE_START, donde evalúa iload_1; ifle WHILE_END para terminar cuando n sea menor o igual a cero. Si la comprobación pasa, imprime n con getstatic, iload_1 y println(I)V, luego decrementa n con iinc 1 -1 y salta de nuevo al inicio del bucle. Al llegar a n = 0, el salto condicional conduce a WHILE_END y se ejecuta return.

## Ex09 CallVoidFunction

Este archivo muestra cómo invocar un método estático que no devuelve ningún valor. El main simplemente hace invokestatic Ex09_CallVoidFunction/greet()V y return. El método greet tiene su propio .limit stack 2 y .limit locals 0, y dentro obtiene System.out, carga una cadena literal con ldc y llama a println para mostrar "Hola desde la función". Cuando greet finaliza con return, el control vuelve al main.

## Ex10 CallReturnInt

En este caso se demuestra la llamada a un método estático que devuelve un entero. main usa invokestatic Ex10_CallReturnInt/getValue()I y espera que el valor retornado quede en la pila del llamador. Inmediatamente después, getstatic carga System.out y swap reordena la pila para que println(I)V reciba primero el PrintStream y luego el entero. El método getValue estático simplemente hace ldc 123 y ireturn, poniendo 123 en la pila del llamador.

## Ex11 CallWithParam

Aquí se muestra el paso de un parámetro a un método estático y el uso del valor retornado. El main empuja ldc 7 y luego invoca invokestatic Ex11_CallWithParam/inc(I)I. El método inc(I)I declara .limit locals 1 y trata su parámetro como local 0: carga el parámetro con iload_0, suma iconst_1, y devuelve el resultado con ireturn. De regreso en main, el entero retornado se imprime tras colocar System.out y usar swap para ordenar la pila.

## Ex12 CallWithMultipleParams

El último ejemplo extiende el anterior al paso de múltiples parámetros. main empuja ldc 10 y ldc 20 y llama invokestatic Ex12_CallWithMultipleParams/sum(II)I. El método sum(II)I reserva .limit

locals 2 y conoce sus parámetros en local 0 y local 1; carga ambos con iload_0 y iload_1, los suma con iadd y devuelve el resultado con ireturn. Al volver al main se imprime la suma colocándola junto a System.out y aplicando swap antes de println(I)V.

# PARTE 3

## 3.1 Introducción

Se piden realizar 4 ampliaciones en la gramática que amplíen la funcionalidad del lenguaje para acercarlo más a un lenguaje de programación habitual. Se han realizado las siguientes ampliaciones:

- Operadores aritméticos con multiplicación, división, potencias, módulo  y operadores lógicos (and, or y not).
Los operadores de multiplicación, división, potencias, módulo completan el conjunto mínimo de operadores básicos presente en prácticamente todos los lenguajes generalistas (C, Java, Python), permitiendo expresar cálculos no triviales de forma natural. Los operadores lógicos, permiten construir condiciones compuestas del mismo modo que en lenguajes como Java o C#, aumentando la expresividad de las estructuras de control.

- Booleanos.
La incorporación de un tipo booleano sigue el modelo presente en muchos lenguajes como Java, C# o Python, donde los booleanos son tipos primitivos esenciales para expresiones condicionales.

- Bucle for
La introducción de un bucle for replica el patrón clásico de iteración contada adoptado por muchos lenguajes como C y Java.

- Bucle while
El bucle while proporciona un mecanismo de iteración controlada por condición, equivalente al existente en lenguajes como C, Java, Python.
La inclusión de break y continue en los bucles while y for, también presentes en estos lenguajes, permite controlar de manera precisa el flujo dentro del bucle, mejorando la expresividad y alineando el diseño con las prácticas imperativas habituales.

## 3.2 Lenguaje E++

### 3.2.1 Lexer

El lenguaje E++ incluye un conjunto reducido de palabras reservadas que definen su semántica básica. Entre ellas se encuentran:

- asignar, utilizada para inicializar variables.
- mostrar, para imprimir valores por pantalla.
- si, no y terminar, que forman parte de las estructuras condicionales tipo if-else.

Las ampliaciones 3 y 4 introducidas añaden nuevas palabras clave para los bucles:

- Mientras y hacer para bucles tipo while.
- Para, hasta, desde y paso para bucles tipo for.
- Romper y continuar para el control de flujo dentro de dichos bucles.

Estas palabras clave se han definido como tokens literales dentro del lexer y se ubican antes de la regla de identificadores, garantizando que ANTLR las reconozca como tokens propios del lenguaje y no como simples nombres de variable.
Dado que ANTLR distingue entre mayúsculas y minúsculas por defecto, las palabras reservadas deben escribirse en minúsculas para ser reconocidas correctamente.

Además, se empleará el símbolo '???' para indicar el inicio de una condición y la secuencia '->' para abrir los bloques si y no. Cada bloque se cierra con la palabra terminar, lo que facilita el análisis estructural y permite la anidación de condicionales de forma natural.

Para señalar el fin de una instrucción, el lenguaje utiliza el delimitador ;P, token denominado FIN_LINEA en la gramática.

El lexer contempla los operadores de asignación y comparación habituales en la mayoría de lenguajes de programación: =, ==, !=, <, >, <=, >=.
Se han definido como tokens individuales, situando las versiones de dos caracteres (==, !=, <=, >=) antes de las de un solo carácter para evitar ambigüedades durante el reconocimiento.

Del mismo modo, se incluyen los operadores aritméticos básicos (+, -) y, como parte de las ampliaciones propuestas, se añaden los operadores de multiplicación, división, potencias, módulo (*, /, ^ y %).
Estos se definen en el lexer de manera independiente, de modo que las reglas de precedencia se establezcan posteriormente en la gramática sintáctica.

En cuanto a los operadores lógicos, como ampliación se incorporan los tokens and, or y not, permitiendo la combinación de condiciones dentro de expresiones booleanas. Estas ampliaciones aproximan el lenguaje E++ a la funcionalidad de lenguajes de programación reales.

El lexer reconoce tres tipos principales de literales:

- Números enteros (INT): secuencias de uno o más dígitos.
- Números de coma flotante (FLOAT): números formados por una secuencia de dígitos seguida opcionalmente de un punto y otra secuencia de dígitos.
- Cadenas de texto (STRING): acepta cualquier carácter Unicode. Cadenas delimitadas por comillas dobles y con soporte para secuencias de escape (por ejemplo, \" o \\).

Debido a que los FLOAT aceptan números sin punto igual que un INT, la definición de INT se coloca antes de FLOAT para que se detecte como entero.
Como se suele hacer realmente en compiladores, los INTs y FLOATs al nivel del lexer solamente se identifican como números positivos, separados del signo, si lo hubiese, que se identificaría como un token de operador aritmético. El parser será el que “junte” ambos tokens para crear o operaciones aritméticas o valores numéricos negativos.

Además, el lenguaje introduce los valores booleanos verdadero y falso, que se reconocen como tokens específicos (VERDADERO y FALSO), lo que permite trabajar con expresiones lógicas de forma más legible y natural.

Los identificadores (ID) representan los nombres de variables y otros símbolos definidos por el usuario. Se componen de una letra o guión bajo al inicio, seguidos de letras, dígitos o guiones bajos. En esta implementación se ha incluido la letra ñ dentro del rango permitido. El orden de definición también es relevante, al situarse después de las palabras clave, se evita que lexemas como asignar o mostrar sean tratados como identificadores.

Los comentarios comienzan con el símbolo # y se extienden hasta el final de la línea.
La regla **COMENTARIO : '#' ~[\r\n]* -> skip;** significa “cero o más caracteres que no sean salto de línea”, lo que hace que el lexer ignore por completo los comentarios desde que se detecta almohadilla hasta el salto de línea, impidiendo que lleguen al parser.

Esta decisión cumple con la especificación del lenguaje, que indica que los comentarios deben ocupar una línea completa y no pueden coexistir con instrucciones en la misma línea.

Por su parte, la regla **ESPACIO : [ \t\r\n]+ -> skip;** descarta espacios, tabulaciones y saltos de línea, simplificando el flujo de tokens que se entrega al parser.

Dado que el final de instrucción está delimitado por ;P, no es necesario conservar los saltos de línea como parte de la sintaxis.

### 3.2.2 Parser

La regla inicial de la gramática es “programa”, que representa el código fuente completo.
Define que un programa puede estar compuesto por cero o más elementos, cada uno de los cuales puede ser una sentencia simple o un condicional, y que el programa finaliza con el token especial EOF (End Of File).

Esta regla superior permite que el parser acepte programas vacíos, programas con una sola instrucción o con múltiples estructuras anidadas. La inclusión de EOF asegura que el análisis finalice correctamente y que no existan tokens residuales tras la última instrucción.

#### Sentencias simples

Las sentencias simples constituyen las unidades básicas del lenguaje. La regla sentencia agrupa distintos tipos de instrucciones que comparten un formato común: ejecutan una acción concreta y terminan con el delimitador ;P. Las instrucciones son declaración, asignación, impresión, bucle y control_bucle.
Cada tipo de sentencia se define de manera independiente para reflejar la semántica particular de cada operación:

- Asignación (declaración): inicializa una variable utilizando la palabra clave asignar. Por ejemplo, asignar a = 20 ;P. Esta regla exige la presencia explícita de asignar, un identificador, el operador ‘=’ y una expresión válida, seguida del delimitador ;P.
- Reasignación: reasigna un valor a una variable ya existente. A diferencia de la declaración, aquí no se utiliza la palabra clave asignar, lo que permite distinguir entre inicialización y actualización de variables.
- Impresión: enviaría un valor a la salida estándar. Esta instrucción se utiliza para mostrar números, cadenas o resultados de operaciones, cumpliendo la función de salida del lenguaje.
- Bucle y control de bucles añaden estructuras repetitivas y control de flujo dentro de las mismas como ampliación.

Las declaraciones, asignaciones e impresiones permiten tanto expresiones aritméticas (a=-3, a=5+2 …) como booleanas (a=verdadero , a= A or B …).

#### Condicionales si, no y terminar

La sintaxis del lenguaje para las estructuras condicionales está basada en los símbolos ??? para marcar la condición, si -> y no -> para definir los bloques de ejecución, y terminar para cerrar el condicional.

La regla correspondiente es la siguiente:
**cond=condicion MARCA_COND SI FLECHA thenBlock=bloque (SI_NO FLECHA elseBlock=bloque)? TERMINAR**

Esta definición reconoce una condición booleana, seguida del marcador ???, el bloque si ->, un bloque opcional no -> y la palabra terminar como cierre.

Gracias a la definición de bloque, los condicionales pueden anidarse de forma recursiva, lo que permite estructuras complejas de decisión.

### Bucles for, while y control de flujo

La ampliación del lenguaje incorpora bucles, que se gestionan mediante una regla general 'bucle' que puede derivar a un while o a un for.

El bucle for se modela con una regla específica bucle_para, cuyo patrón es: **PARA var=ID DESDE inicio=expresion HASTA fin=expresion (PASO paso=expresion)? HACER FLECHA cuerpo=bloque TERMINAR**
Esto permite escribir construcciones del tipo para i desde 0 hasta 10 hacer -> ... terminar y, opcionalmente, añadir un paso como paso -1, paso 2 o incluso paso distancia*3, usando una expresión completa.

El bucle while tendrá la siguiente estructura:
**bucle_mientras : MIENTRAS cond=condicion HACER FLECHA cuerpo=bloque TERMINAR**
Este bucle evaluaría la condición indicada; mientras sea verdadera, ejecuta el bloque de instrucciones asociado. El uso de las palabras clave mientras, hacer, terminar y la flecha -> mantiene la coherencia con las estructuras condicionales.

Adicionalmente, se incluyen las instrucciones romper y continuar para el control del flujo dentro de los bucles:
**control_bucle : ROMPER FIN_LINEA | CONTINUAR FIN_LINEA;**
Estas instrucciones permiten finalizar la iteración actual o salir completamente del bucle, replicando la funcionalidad de los clásicos break y continue en otros lenguajes.

### Bloques

Un bloque representa un conjunto de instrucciones que pueden ejecutarse como unidad.
La regla: **bloque : (sentencia | condicional)* ;**
permite que un bloque contenga tanto sentencias simples como estructuras condicionales anidadas, o incluso estar vacío como en algunos lenguajes (C, C++...).

## Expresiones booleanas y lógicas

En la mayoría de lenguajes de programación (C, Java, Python…) el orden de precedencia de los operadores lógicos es not > and > or. Por ejemplo, si analizamos de izquierda a derecha sin prioridad A or B and C sería (A or B) and C, aunque siguiendo las prioridades sería A or (B and C), que es lo "correcto". Si en este ejemplo A fuese verdadero y B y C falso incluso darían resultados distintos dependiendo de cómo se interpretase.

Por ello, en E++ también se sigue la misma regla de prioridades. Esto se logra con la siguiente estructura de reglas:

```
condicion : bool_o ;
bool_o :left=bool_y (ops+=OR rights+=bool_y)* #BoolOr
bool_y :left=bool_no (ops+=AND rights+=bool_no)* #BoolAnd
bool_no : NOT negado=bool_no                    #BoolNot
  | PAREN_ABRE inner=condicion PAREN_CIERRA  #BoolParen
  | comp=comparacion                        #BoolComp
  | bool=booleano                          #BoolAtom
  ;
```

- bool_o representa disyunciones (OR) de varias conjunciones.
- bool_y representa conjunciones (AND) de unidades booleanas.
- bool_no maneja la negación (NOT) y las unidades básicas: paréntesis, comparaciones y valores booleanos.

## Comparaciones y operadores relacionales

Las comparaciones binarias se definen a través de la regla:

```
comparacion : left=operando_cmp op=operador_relacional right=operando_cmp
```

donde operador_relacional puede ser cualquiera de los operadores definidos (==, !=, <, <=, >, >=). Esto permite condiciones del tipo a > 2, x == y o nota <= 10.

La regla 'booleano' complementa este sistema incluyendo los valores literales verdadero y falso.

### Expresiones aritméticas

En las operaciones aritméticas, primero se definen las expresiones generales, luego los términos de mayor prioridad (multiplicación y división), y finalmente los factores, que representan los elementos atómicos de una operación:

```
expresion :left=termino (ops+=(SUMA | RESTA) rights+=termino)*   #ExprSumRes
termino : left=potencia (ops+=(MULT | DIV | MOD) rights+=potencia)* #ExprMulDivMod;
potencia : base=factor (POT exp=potencia)?                       #ExprPow
factor : RESTA neg=factor                                        #ExprNeg
  | PAREN_ABRE inner=expresion PAREN_CIERRA                      #ExprParen
  | atom=atomo                                                   #ExprAtom
  ;
```

Esta jerarquía asegura que las operaciones de multiplicación y división se evalúen antes que las de suma y resta, y que los paréntesis puedan alterar el orden natural de evaluación.
La posibilidad de aplicar el signo negativo unitario (-a) está contemplada en la primera alternativa de factor.

El nivel más bajo, atomo, incluye los posibles operandos de una expresión:

```
atomo : intLit=INT
  | floatLit=FLOAT
  | strLit=STRING
  | id=ID
  ;
```

De esta manera, las expresiones pueden involucrar números, cadenas o variables, permitiendo combinaciones como b = (b ^ 2) * 2 ;P.

## 3.3 Detección de errores

### Léxicos y sintácticos

ANTLR detecta automáticamente la mayoría de errores estructurales, pero se ha configurado un ErrorListener personalizado para producir mensajes claros y homogéneos.

Error 1: carácter fuera del alfabeto (léxico)

- Ejemplo:
  asignar @ = 5 ;P

- Mensaje generado:
  Tipo: Léxico. línea 5:8 -> token recognition error at: '@'

---

Error 2: uso de no -> sin un si previo (sintaxis)

- Ejemplo:
  no ->
    mostrar "huérfano" ;P
  terminar

- Mensaje generado:
  Tipo: Sintaxis. línea 7:0 -> mismatched input 'no' expecting {<EOF>, 'asignar', 'mostrar', 'mientras', 'para', 'romper', 'continuar', 'not', '-', '(', 'verdadero', 'falso', INT, FLOAT, STRING, ID}

---

Error 3: falta de ;P al final de una sentencia (sintaxis)

- Ejemplo:
  asignar a = 10

- Mensaje generado:
  Tipo: Sintaxis. línea 12:0 -> missing ';P' at '<EOF>'

---

Error 4: si sin terminar (sintaxis con recuperación)

- Ejemplo:
  verdadero ???
  si ->
    mostrar "falta el terminar" ;P

- Mensaje generado:
  Tipo: Sintaxis. línea 12:0 -> mismatched input '<EOF>' expecting {'no', 'terminar'}

- (Aquí ANTLR intenta recuperarse y el visitor llega a ejecutar el mostrar, pero el compilador ya ha informado del error de sintaxis.)

## Semánticos

Además de los errores léxicos y sintácticos, el compilador de E++ implementa una capa de comprobaciones semánticas apoyada en la tabla de símbolos. Estas comprobaciones se inspiran en el comportamiento de compiladores de lenguajes como C y Java, pero adaptadas a la simplicidad del lenguaje de la práctica.

La tabla de símbolos almacena para cada identificador su nombre, tipo estático (INT, FLOAT, STRING, BOOL) y la dirección asociada en las variables locales de Jasmin. A partir de esa información se detectan distintos tipos de errores:

**Error 5:**
Cuando se hace referencia a un identificador que no existe en la tabla de símbolos (por ejemplo al asignar o al usarlo en una expresión), se registra un error del tipo “Variable 'x' no declarada”. Este comportamiento es análogo al de C/Java, donde cualquier uso de una variable sin declaración previa provoca un error de compilación.

- Ejemplo:
  a ???
  si ->
    mostrar "nunca" ;P
  terminar

- Mensaje generado:
  ERROR SEMÁNTICO: Variable 'a' no declarada (línea X, columna Y)

---

**Error 6:**
Si se intenta declarar dos veces la misma variable en el ámbito global, la inserción en la tabla de símbolos lo detecta y emite el mensaje *“Variable 'x' ya declarada”*. Esta restricción evita ambigüedades de significado y sigue el modelo de lenguajes imperativos clásicos que no permiten redefinir identificadores en el mismo ámbito.

- Ejemplo:
  asignar a = 1 ;P
  asignar a = 2 ;P

- Mensaje generado (forma):
  ERROR SEMÁNTICO: Variable 'a' ya declarada (línea X, columna Y)

---

**Error 7:**
En cada asignación se compara el tipo estático de la expresión con el tipo de la variable de destino. El compilador solo permite una promoción controlada de INT a FLOAT, ya que el Jasmin subyacente también soporta esta conversión. Cualquier otro caso (por ejemplo asignar un STRING a un INT, o un BOOL a un FLOAT) se rechaza con un mensaje “No se puede asignar valor de tipo A a variable de tipo B”. Se sigue así un enfoque estáticamente tipado similar al de Java, donde las conversiones peligrosas no se realizan de forma implícita.

- Ejemplo:
  asignar a = 3 ;P  # a es INT
  a = "hola" ;P  # intento asignar STRING a INT

- Mensaje generado (forma):

ERROR SEMÁNTICO: No se puede asignar valor de tipo STRING a variable de tipo INT (línea X, columna Y)

---

**Error 8:**
No se permite una expresión aritmética con operandos no numéricos

- Ejemplo:
  asignar b = verdadero ;P
  asignar x = b + 1 ;P

- Mensaje generado (forma):
  ERROR SEMÁNTICO: Se esperaba valor numérico y se encontró tipo BOOL (línea X, columna Y)

- (Este mismo tipo de mensaje aparece si intentas usar STRING en +,-,*,/.)

---

**Error 9:**
No se permite una condición no booleana en si / mientras

- Ejemplo:
  asignar x = 5 ;P
  x ???  # x es INT, no BOOL
  si ->
  mostrar "nunca" ;P
  terminar

- Mensaje generado (forma):
  ERROR SEMÁNTICO: Se esperaba booleano y se encontró tipo INT (línea X, columna Y)

- Esto aplica tanto a si como a mientras.

---

**Error 10:**
Error por comparaciones numéricas con operandos no numéricos

- Ejemplo:
  asignar s = "hola" ;P
  asignar x = 1 ;P
  (s > x) ???  # intento usar STRING en '>'
  si ->

```
  mostrar "..." ;P
terminar
```

- Mensaje generado (forma):
  ERROR SEMÁNTICO: Se esperaba valor numérico y se encontró tipo STRING (línea X, columna Y)

---

**Error 11:**
En las operaciones aritméticas que implican división, el compilador analiza si el divisor es una constante literal igual a cero. Esta decisión se inspira en el tratamiento de la división por cero en otros lenguajes, como java o c/c++.

- Ejemplo:
```
asignar x = 5 ;P
asignar z = x / 0 ;P
```

- Mensaje generado (forma):
  ERROR SEMÁNTICO: División por cero (línea X, columna Y)

---

**Error 12:**
Error de control de bucles (romper / continuar). El compilador mantiene dos pilas de etiquetas para gestionar break y continue (pilaBreak y pilaContinue). Cuando aparece un romper o continuar fuera de cualquier bucle, las pilas están vacías y se registra un error de control del tipo “'romper' o 'continuar' fuera de un bucle”, incluyendo la línea y columna. Este diseño replica la restricción típica de lenguajes como C/Java, donde break y continue deben encontrarse dentro del cuerpo de un bucle.

- Ejemplo:
```
continuar ;P
```

- Mensaje generado:
  ERROR DE CONTROL DE BUCLE: 'romper' o 'continuar' fuera de un bucle.

---

**Error 13:**
Como en Python, el paso del bucle no puede ser cero.

- Ejemplo:
```
asignar fin = 10 ;P
para i desde 0 hasta fin paso 0 hacer ->
  mostrar i ;P
terminar
```

- Mensaje generado (forma):
  ERROR SEMÁNTICO: El paso de un bucle 'para' no puede ser 0 (línea X, columna Y)

---

**Error 14:**
La variable de control debe ser siempre de tipo INT. Si no existe, se declara automáticamente como entera; si existe y su tipo es distinto, se produce un error.

- Ejemplo:
  ```
  asignar i = verdadero ;P
  para i desde 0 hasta 10 hacer ->   # i ya existe pero no es INT
    mostrar i ;P
  terminar
  ```

- Mensaje generado (forma):
  ERROR SEMÁNTICO: La variable del bucle 'para' debe ser de tipo INT (línea X, columna Y)

---

**Error 15:**
Las expresiones de inicio y fin del rango también deben ser enteras, evitando así la combinación de bucles con límites reales, como en Python.

- Ejemplo:
  ```
  asignar i = verdadero ;P
  para i desde 0 hasta 10 hacer ->   # i ya existe pero no es INT
    mostrar i ;P
  terminar
  ```

- Mensaje generado (forma):
  - Si es el primer valor(desde):
    ERROR SEMÁNTICO: El inicio del bucle 'para' debe ser entero (tipo actual: FLOAT) (línea X, columna Y)

  - Si es el segundo valor (hasta):
    ERROR SEMÁNTICO: El límite del bucle 'para' debe ser entero (tipo actual: FLOAT) (línea X, columna Y)

---

**Error 16:**
Error debida a la comparación de tipos incompatibles

- Ejemplo:
  ```
  asignar a = verdadero ;P      # a : BOOL
  asignar b = 3 ;P              # b : INT
  a == b ???
  si ->
    mostrar "iguales" ;P
  no ->
    mostrar "distintos" ;P
  terminar
  ```

- Mensaje generado (forma):
  ERROR SEMÁNTICO: Comparación de igualdad entre tipos incompatibles: BOOL y INT (línea X, columna Y)

---

**Error 17:**
Error por una comparación relacional (>, < , ...) con booleanos

- Ejemplo:
  ```
  asignar a = verdadero ;P
  asignar b = falso ;P
  a < b ???
  si ->
    mostrar "menor" ;P
  no ->
    mostrar "no menor" ;P
  terminar
  ```

- Mensaje generado (forma):
  ERROR SEMÁNTICO: Operador relacional '<' solo se puede usar con operandos numéricos (no con BOOL y BOOL) (línea X, columna Y)

---

**Error 18:**
Error por una comparación relacional con strings (semántico)

- Ejemplo:
  ```
  asignar a = "hola" ;P
  asignar b = "adios" ;P
  a < b ???
  si ->
    mostrar "menor" ;P
  no ->
    mostrar "no menor" ;P
  terminar
  ```

- Mensaje generado (forma):
  ERROR SEMÁNTICO: Operador relacional '<' solo se puede usar con operandos numéricos (no con STRING y STRING) (línea X, columna Y)

## 3.4 Clases Importantes

### 3.4.1 CompileVisitor

Es la clase central del generador de código para el lenguaje definido por EJ3Parse. Su propósito es recorrer el árbol AST del programa y traducir cada nodo a su equivalente en código jasmine, gestionando, al mismo tiempo, la verificación de tipos estáticos y la detección de errores semánticos. El código generado se devuelve en formato String.

Se pueden separar distintos campos dentro de este código:

1. UtilidadesnewLabel()  Genera etiquetas únicas (L0, L1, etc.) para los saltos de control de flujo (if, while, goto).

2. visitPrograma() y  visitBloque() Recorren secuencialmente los hijos del nodo (sentencias) y concatenan el código generado por cada uno.

3. visitDeclaracion(): Genera el código para evaluar la expresión y lo almacena en una nueva variable tras registrarla en la tabla de símbolos.

4. visitAsignacion(): Busca la variable en la tabla. Genera código para la expresión RH. Implementa la promoción de INT a FLOAT si la variable es FLOAT. Genera un Error Semántico si los tipos son incompatibles y no se puede realizar la promoción.

5. instruccionStore(): Genera la instrucción de almacenamiento adecuada de jasmin basada en el tipo de la variable.

6. instruccionLoad(): Genera la instrucción de carga adecuada de jasmin.

7. visitImpresion(): Genera el código para llamar al método System.out.println apropiado (println(I), println(F), println(Ljava/lang/String;)V) según el tipo de la expresión a imprimir.

8. visitCondicional(): Genera el código para la estructura if-else.

9. visitBucle_mientras()  Genera el código para el bucle while. Define etiquetas de inicio y fin, usa ifeq para salir del bucle y gestiona las pilas pilaBreak/pilaContinue.

10. visitBucle_para(): Genera el código complejo del bucle for (tipo para). En este caso si se indica el paso a seguir. Por lo tanto sigue una gramática como un bucle mientras, para evitar bucles de ejecución infinita como ocurre en python. De la siguiente manera:
    a. For inicio, fin, paso:
       Si paso> 0 e inicio >= fin → no se ejecuta ninguna vez.
       Si paso>0 e inicio < fin → se ejecuta, la variable incrementa y se comprueba

       Si paso< 0 y inicio<= fin→no se ejecuta ninguna vez.
       Si paso< 0 y inicio >  fin→ se ejecuta, la variable decrementa y se comprueba

11. visitControl_bucle() Genera saltos incondicionales (goto) a las etiquetas break o continue del bucle más interno, usando las pilas pilaBreak/pilaContinue. Reporta un error si se usan fuera de un bucle.

12. Booleanos visitBoolOr(), visitBoolAnd(): Generan código para los operadores booleanos OR y AND ya que los booleanos se representan como enteros (1/0).

13. visitBoolNot(): Genera código para el NOT lógico, invirtiendo el valor booleano usando saltos (ifeq).

14. visitComparacion(): Implementa la lógica de tipado para comparaciones para generar las instrucciones correctas  y promociona INT a FLOAT si es necesario.

15. visitExprSumRes(): Genera código para sumas y restas. Implementa la promoción de tipos: si algún operando es FLOAT, el código promueve el otro a FLOAT.

16. visitExprMulDivMod() Genera código para multiplicación, división y módulo. Implementa la promoción de tipos y la verificación de división por cero.

17. visitExprPow(): Genera código para la potencia. Promociona la base y el exponente a Double, a excepción si ambos operandos son INT, convierte el resultado a INT.

18. visitExprNeg(): Genera código para la negación.

19. visitAtomo(): Genera el código para constantes (ldc) o para cargar variables.

20. tipoExpresion(), tipoTermino()... Son funciones de Inferencia de Tipos. Recorre recursivamente el AST para determinar el tipo estático resultante de una expresión, aplicando reglas de promoción como INT + FLOAT = FLOAT..

### 3.4.2 EJ3_Main

La clase EJ3_Main es la clase encargada desde leer el archivo fuente para generar y guardar el código intermedio en formato Jasmin. Su función esencial es transformar un archivo de entrada de formato .txt en un archivo ensamblador .j compatible con Jasmin, para su posterior paso a .class mediante el comando:

```
java -jar $env:Jasmine -d "Code_Ex_Generado" Ex04_PrintBoolean.j
```

donde $env:Jasmine es el nombre de la variable en el path llamada Jasmine para poder acceder al .jar

Se preparan las clases de ANTLR necesarias EJ3Lexer y EJ3Parser, junto con los manejadores de errores personalizados EJ3MiErrorListener.

El contenido del archivo se carga como CharStream, pasa por el Lexer para el análisis léxico y, posteriormente, los tokens resultantes se entregan al Parser para el análisis sintáctico. En esta fase se construye el AST, invocando la regla inicial parser.programa().

Cuando el AST está listo y no se han detectado errores léxicos o sintácticos, comienza la generación de código. Para ello se utiliza la clase CompileVisitor, que recorre el árbol y produce el cuerpo del código Jasmin.

Al mismo tiempo, el Visitor lleva a cabo el análisis semántico, registrando errores de tipos o variables en su tabla de símbolos. Si se encuentra algún problema semántico, el proceso se detiene antes de generar el archivo final.

Si todo ha ido bien en cada etapa, el código Jasmin resultante se envuelve dentro de la estructura de clase y del método main mediante la función createJasminFile, que utiliza el nombre del archivo de entrada para nombrar la clase.

Finalmente, el compilador guarda el archivo generado en la carpeta PL3/Parte3/Exj_jasmine/ con extensión .j, e informa al usuario de la ubicación del archivo listo para ser ensamblado con Jasmin.

Posteriormente con el código de la terminal proporcionado si se tiene la variable Jasmine bien configurada en el PATH, se crearía el archivo .class correspondiente al .j . Con lo que mediante la llamada: *java -cp Exj_jasmine\Code_Ex_Generado 'nombre_archivo.class'* se podría ejecutar esta clase y ver el resultado del código realizado en el archivo de texto.