Universidad de Castilla-La Mancha

Escuela Superior de Informática

Entrega final del laboratorio de TAC

Autores: *Alejandro del Hoyo Abad, Sergio Pozuelo-Martín Consuegra y Alberto Barrais Bellerín*
Grupo: G02
Asignatura: Teoría de autómatas de la computación
Fecha: 3-5-2024

1. Descripción de la práctica

En primer lugar, el objetivo de esta práctica consiste en desarrollar un programa que tome como input un fichero de texto en el cual se encuentra un ejemplo que nos ha proporcionado la profesora que se tendrá que examinar y analizar de una forma concreta que se explicará más adelante. En este sentido, en este fichero se muestra un ejemplo válido de un sublenguaje de la definición de una clase en Java en donde se definen un conjunto de reglas y sintaxis específicas dentro del lenguaje principal que se utilizan para realizar una tarea específica que, en este caso, tiene una secuencia de métodos en donde se nos ha concretado todas las reglas en las que se rige este lenguaje como los tipos de datos permitidos, las instrucciones válidas, las expresiones permitidas, y la forma en que se define una clase, entre otros aspectos.

En este sentido, el propósito de esta práctica consiste en diseñar un procesador de lenguaje capaz de analizar correctamente el sublenguaje establecido por la profesora en el enunciado de la práctica. Además, aunque las etapas específicas de un procesador de lenguajes pueden diferir según el lenguaje que se esté procesando como entrada y del tipo de salida que se esté generando, todas pueden clasificarse en alguna de las siguientes categorías: análisis léxico (lexers), análisis sintáctico (parsers) y generación de código máquina.

Por lo tanto, para la primera parte, se ha desarrollado la parte que se corresponde al análisis léxico. De esta forma, en esta fase inicial el programa lee una cadena de caracteres y genera una secuencia de elementos léxicos o tokens. En este contexto, podemos entender el token como un término que se vincula con un conjunto específico de letras o números, es decir, es una unidad única que generalmente tiene un significado por sí misma. Asimismo, durante el análisis léxico de un programa, se descompone la secuencia total de caracteres en estos tokens. También, es importante resaltar que, para establecer de manera precisa qué combinaciones de letras y números pueden constituir un token, se emplean expresiones regulares. Estas expresiones son patrones formales que detallan las combinaciones válidas de las distintas cadenas de texto.

Entonces, teniendo en cuenta lo previamente mencionado, se ha usado la herramienta JFlex (Java Fast Lexical Analyzer Generator) la cual nos permite generar analizadores léxicos (scanners) en Java. De este modo, JFlex permite identificar las cadenas de texto pertenecientes a uno de los lenguajes previamente mencionados, y poder ejecutar una serie de sentencias dependiendo del token con el que se esté tratando. Consecuentemente, JFlex es capaz de identificar palabras clave, operadores, identificadores, números, y otros elementos léxicos relevantes en el código fuente.

Por último, es importante tener en cuenta que si se presenta el caso en el que una misma cadena de texto reconocida por JFlex puede coincidir con distintas expresiones regulares, esta será finalmente reconocida según el orden de precedencia en el que aparecen estas. Por ejemplo, la cadena de texto "true" puede interpretarse como un identificador o como un literal booleano. En este caso particular, es deseable que la regla para el literal booleano esté definida antes que la regla para el identificador.

2. Tabla de elementos léxicos

Esta tabla se ha actualizado con respecto a la entrega 1. En este caso tenemos la siguiente tabla

Descripción	**Expresión Regular**	**Token**	**Ejemplo**
Palabras clave o palabras reservada	public	PUBLIC	public
Palabras clave o palabras reservadas	class	CLASS	class
Palabras clave o palabras reservada	static	STATIC	static
Palabras clave o palabras reservada	void	VOID	void
Palabras clave o palabra reservada	int	INT	int
Palabras clave o palabras reservada	boolean	BOOLEAN	boolean
Palabras clave o palabras reservada	while	WHILE	while
Palabras clave o palabras reservada	for	FOR	for
Palabras clave o palabras reservada	do	DO	do
Palabras clave o palabras reservada	return	RETURN	return
Palabras clave o palabras reservada	main	MAIN	main

Identificadores	(a+…+z+A+…+Z)(a+…z+A+…+Z+0+..+9+'_')*	IDENTIFIER	Ejem_pl1o
Números enteros	(0+1+2+3+4+5+6+7+8+9)*	INTEGER_LIT	45
Operadores aritméticos	('+' + '-')	PLUS_SUB	+
Operadores aritméticos	(‘*’+’/)	MULT_DIV	*
Operadores relacionales	('<=' + '<' + '>' + '>=')	COMPARISON	>
Operadores relacionales	('==' + '!=')	EQUALITY	=
Operadores lógicos	('&&' + '||')	OR_AND	&&
Operadores lógico	‘!’	NOT	!
Booleanos	(true + false)	BOOL_LIT	true
Asignación	=	ASSIGN	=
Signo de puntuación	;	SEMI	;
Signo de puntuación	,	COMMA	,
Signo de puntuación	(	OP_PARENTH	(
Signo de puntuación	)	CL_PARENTH	)
Signo de puntuación	{	OP_BRACKET	{
Signo de puntuación	}	CL_BRACKET	}
Incremento y decremento	('++' + '--')	INCR_DECR	++

2.1 Cambios con respecto a la primera entrega

En este caso, se han tenido que añadir varios tokens para poder controlar la gramática del sublenguaje. Concretamente, hemos tenido que dividir varios tokens que teníamos previamente definidos como una sola unidad. En este sentido, los casos en cuestión son los siguientes:

- MULT_DIV y PLUS_SUB: Se ha tomado la decisión de dividir las operaciones de multiplicación y división de las de suma y resta para controlar su precedencia. Esto garantiza que las multiplicaciones y divisiones se evalúen antes que las sumas y restas, siguiendo las convenciones habituales de la aritmética.

- EQUALITY y COMPARISON: En este caso, se ha optado por dividir los tokens de operaciones relacionales en dos categorías: EQUALITY y COMPARISON. La distinción principal radica en que los operadores de EQUALITY permiten la inclusión de expresiones aritméticas intermedias, mientras que los operadores de COMPARISON se restringen a expresiones booleanas puras. Este enfoque facilita una mayor claridad y coherencia en la interpretación de las operaciones relacionales.

- NOT y OR_AND: Aquí se ha realizado una separación entre los operadores NOT y OR_AND. El operador NOT precede una expresión booleana única, mientras que los operadores OR_AND requieren expresiones booleanas a ambos lados. Esta distinción es crucial para garantizar una correcta evaluación de las expresiones lógicas y mantener la coherencia semántica en el análisis de las mismas.

3. Manual de usuario JFlex

Para ejecutar el programa con JFlex, hemos decido explicar brevemente dos formas distintas por las cuales nosotros hemos podido trabajar con esta herramienta. Entre estas opciones, se elegirá la que resulte más cómoda para el usuario, ya que ambas son igualmente válidas.

3.1. Ejecución desde la terminal

En primer lugar, para trabajar con JFlex mediante la terminal, lo primero es asegurarse de tener instalado correctamente JFlex y tener los *.jar* en el CLASSPATH correspondiente. Tras esto, solo resta seguir los siguientes comandos:

$ jflex entrega.fle

$ javac IdentificaTokens.java

Con este comando, se genera una archivo de Java que se puede ejecutar directamente. No obstante, en este caso JFlex crea el programa por la clase Scanner que implica que lee el input de los argumentos de entrada. Por tanto, para analizar léxicamente y obtener los tokens de las cadenas de texto del fichero, usaremos el siguiente comando:

$ java IdentificaTokens practica1.txt

De hecho, una forma muy cómoda de ejecutarlo mediante terminal es tener dichos comandos en un pequeño script en bash, por ejemplo *run.sh* y proporcionarle permisos de ejecución (chmod +x).

3.2. Ejecución usando la IDE de Eclipse

Por otro lado, para ejecutar JFlex en Eclipse, primero hay que asegurarse de haber integrado esta herramienta en la IDE. Para realizar esto, se deberá crear un proyecto nuevo de JFlex en un workspace y añadir los correspondientes JAR para la configuración externa.

Tras haber creado el workspace, será necesario crear un archivo .jflex que contendrá las reglas y expresiones regulares de la práctica que estamos realizando. Posteriormente, ejecutaremos dicho archivo mediante la herramienta externa ya configurada de JFlex que generará un archivo *.java*, en este caso, *IdentificaTokens.java* en la carpeta src del proyecto. En este sentido, *IdentificaTokens.java* será la clase principal para ejecutar el programa.

Finalmente, para correr el Main, se deberá añadir en las configuraciones de ejecución (Run Configurations) como argumento el archivo *practica1.txt* en el que tenemos el input que queremos analizar para la correspondiente prueba. Una vez hecho esto, solo restará pulsar 'Run' para que podamos observar las cadenas de texto reconocidas por JFlex del archivo en la consola.

4. Ejemplo de fichero de entrada y salidas con JFLEX

- El fichero de entrada que hemos usado como input es el que nos ha proporcionado la profesora:

```
public class Ejem_TAC {
	public static boolean comparar(int valor, int tope) {
		int aux;
		aux = calculaValor(valor);
		return ((tope < 5*2) || (aux < 1)) && true;
	}

	public static int calculaValor(int valor1) {
		int aux = 1;
		for (int i = 0; i < valor1; i++) {
			for (int j = i ; j < i; j--) {
				aux = aux + 2;
			}
		aux = aux + 1;
		}
		return aux;
	}
	public static void main() {
		boolean esCierto = true;
		int x = 0;
		while (esCierto){
			x++;
			esCierto = comparar(x,5);
		}
	}
}
```

- El output que nos genera habiendo usado JFlex es el siguiente:

```
PUBLIC ==> public
CLASS ==> class
IDENTIFIER ==> Ejem_TAC
OP_BRACKET ==> {
```

```
PUBLIC ==> public
STATIC ==> static
BOOLEAN ==> boolean
IDENTIFIER ==> comparar
OP_PARENTH ==> (
INT ==> int
IDENTIFIER ==> valor
COMMA ==> ,
INT ==> int
IDENTIFIER ==> tope
CL_PARENTH ==> )
OP_BRACKET ==> {

INT ==> int
IDENTIFIER ==> aux
SEMI ==> ;

IDENTIFIER ==> aux
ASSIGN ==> =
IDENTIFIER ==> calculaValor
OP_PARENTH ==> (
IDENTIFIER ==> valor
CL_PARENTH ==> )
SEMI ==> ;

RETURN ==> return
OP_PARENTH ==> (
OP_PARENTH ==> (
IDENTIFIER ==> tope
REL_OP ==> <
INTEGER_LIT ==> 5
ARITH_OP ==> *
INTEGER_LIT ==> 2
CL_PARENTH ==> )
LOGIC_OP ==> ||
OP_PARENTH ==> (
IDENTIFIER ==> aux
REL_OP ==> <
INTEGER_LIT ==> 1
CL_PARENTH ==> )
```

CL_PARENTH ==>)
LOGIC_OP ==> &&
BOOL_LIT ==> true
SEMI ==> ;

CL_BRACKET ==> }

PUBLIC ==> public
STATIC ==> static
INT ==> int
IDENTIFIER ==> calculaValor
OP_PARENTH ==> (
INT ==> int
IDENTIFIER ==> valor1
CL_PARENTH ==>)
OP_BRACKET ==> {

INT ==> int
IDENTIFIER ==> aux
ASSIGN ==> =
INTEGER_LIT ==> 1
SEMI ==> ;

FOR ==> for
OP_PARENTH ==> (
INT ==> int
IDENTIFIER ==> i
ASSIGN ==> =
INTEGER_LIT ==> 0
SEMI ==> ;
IDENTIFIER ==> i
REL_OP ==> <
IDENTIFIER ==> valor1
SEMI ==> ;
IDENTIFIER ==> i
INCR_DECR ==> ++
CL_PARENTH ==>)
OP_BRACKET ==> {

FOR ==> for

```
OP_PARENTH ==> (
INT ==> int
IDENTIFIER ==> j
ASSIGN ==> =
IDENTIFIER ==> i
SEMI ==> ;
IDENTIFIER ==> j
REL_OP ==> <
IDENTIFIER ==> i
SEMI ==> ;
IDENTIFIER ==> j
INCR_DECR ==> --
CL_PARENTH ==> )
OP_BRACKET ==> {

IDENTIFIER ==> aux
ASSIGN ==> =
IDENTIFIER ==> aux
ARITH_OP ==> +
INTEGER_LIT ==> 2
SEMI ==> ;

CL_BRACKET ==> }

IDENTIFIER ==> aux
ASSIGN ==> =
IDENTIFIER ==> aux
ARITH_OP ==> +
INTEGER_LIT ==> 1
SEMI ==> ;

CL_BRACKET ==> }

RETURN ==> return
IDENTIFIER ==> aux
SEMI ==> ;

CL_BRACKET ==> }

PUBLIC ==> public
```

```
STATIC ==> static
VOID ==> void
MAIN ==> main
OP_PARENTH ==> (
CL_PARENTH ==> )
OP_BRACKET ==> {

BOOLEAN ==> boolean
IDENTIFIER ==> esCierto
ASSIGN ==> =
BOOL_LIT ==> true
SEMI ==> ;

INT ==> int
IDENTIFIER ==> x
ASSIGN ==> =
INTEGER_LIT ==> 0
SEMI ==> ;

WHILE ==> while
OP_PARENTH ==> (
IDENTIFIER ==> esCierto
CL_PARENTH ==> )
OP_BRACKET ==> {

IDENTIFIER ==> x
INCR_DECR ==> ++
SEMI ==> ;

IDENTIFIER ==> esCierto
ASSIGN ==> =
IDENTIFIER ==> comparar
OP_PARENTH ==> (
IDENTIFIER ==> x
COMMA ==> ,
INTEGER_LIT ==> 5
CL_PARENTH ==> )
SEMI ==> ;

CL_BRACKET ==> }
```

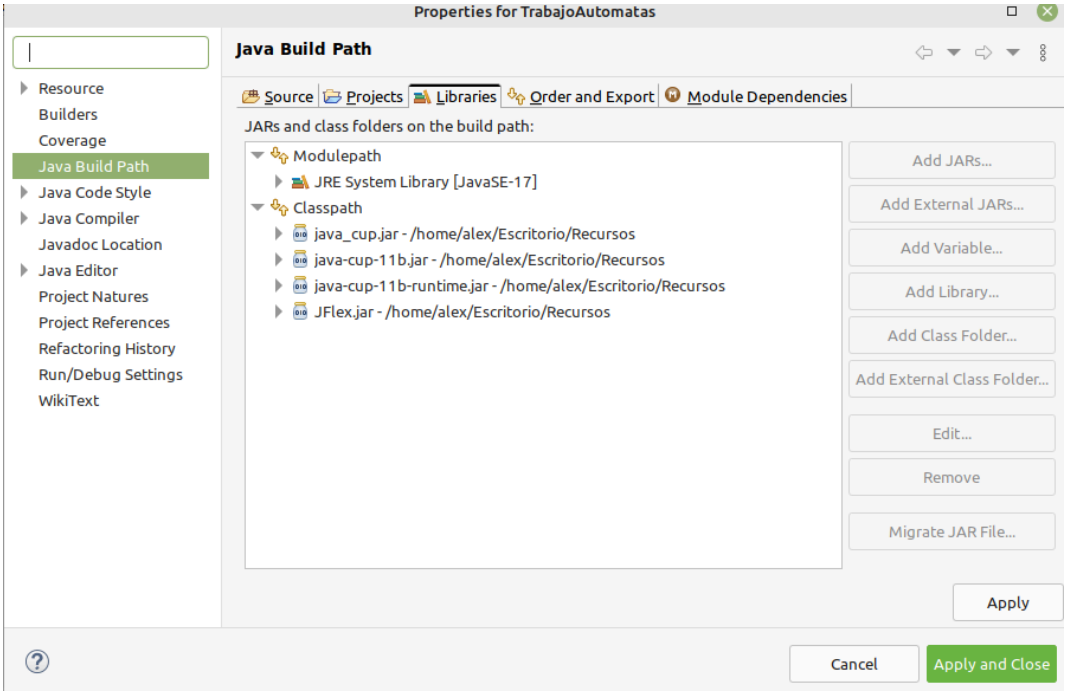
CL_BRACKET ==> }

CL_BRACKET ==> }

5. Manual de usuario para CUP.

En este caso será necesario llevar a cabo los siguientes pasos para compilar el programa de forma correcta. En este caso, hemos decidido hacer una parte por terminal y la otra parte, la ejecución del parser, mediante la IDE de Eclipse en Linux. En este sentido, destacamos lo siguiente:

En primer lugar, se creará un proyecto Java en Eclipse en donde añadiremos los siguientes import como *jar* externos para que funcione correctamente.

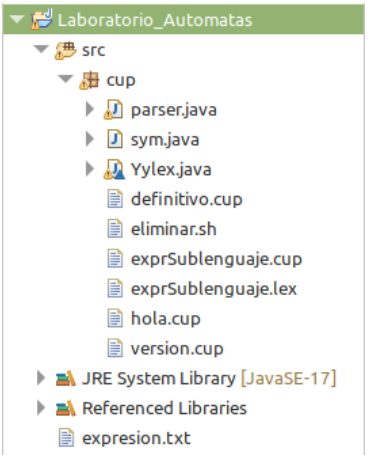


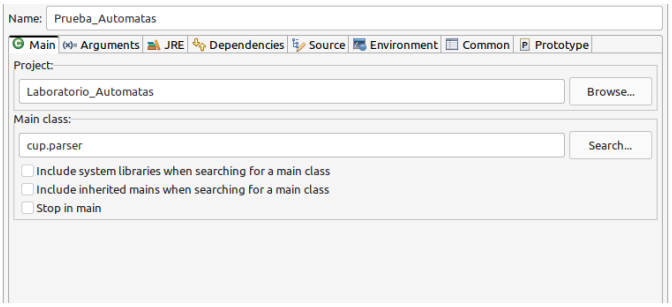
Tras esto, nos situamos en la carpeta del proyecto en el workspace en donde tendremos el archivo *.lex* donde se define la sintaxis y *.cup* en donde se define la gramática. En este caso, se ejecutarán los siguientes comandos:

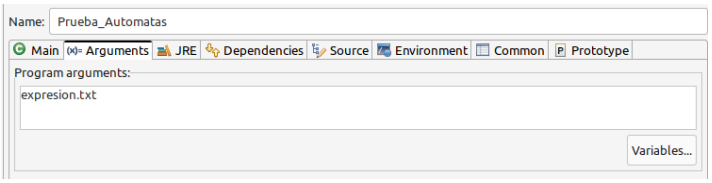
$ jflex exprSublenguaje.flex

$ cup exprSublenguaje.cup

Posteriormente, se configura en run configurations indicando en project el proyecto actual donde tenemos todos los archivos y teniendo como main class la clase *cup.parser,* puesto que en este caso lo tenemos en el paquete cup. Posteriormente, se añade en arguments el fichero. *txt* que deseamos que sea analizado para que nos compruebe que el parser reconoce correctamente el lenguaje. Por último, si se realiza un cambio en algún fichero, será necesario actualizar Eclipse para que esos cambios se hagan efectivos.







En este sentido, al ejecutar el run, CUP analizará si es correcto el lenguaje indicando que en la consola que el análisis es correcto o indicará un Syntax error imprimiendo por pantalla que el análisis es incorrecto aunque no se indicará en qué punto concreto se produce dicho error.

6. Gramática para analizar el sublenguaje en CUP

Una vez habiendo realizado el análisis léxico de JFlex, pasamos a la parte correspondiente al análisis sintáctico que se realizará mediante CUP (Construction of Useful Parsers). En este sentido, el análisis sintáctico consiste en identificar aquellas estructuras presentes en un código, aquellas que se ajustan a unos conjuntos de reglas sintácticas, las cuales constituyen una gramática. De esta forma, el analizador sintáctico se encarga de construir el árbol sintáctico a partir de las estructuras que se han encontrado. Por lo tanto, gracias a la herramienta CUP seremos capaces de generar automáticamente un analizador sintáctico o parser a partir de dicha gramática.

Por otro lado, creemos que es destacable mencionar que quien realmente luego se encarga de comprobar que se cumplen las restricciones semánticas del lenguaje es el análisis semántico. En este sentido, algunos de los ejemplos que el análisis

semántico tendrá en cuenta son: la necesidad de definir las variables antes de utilizarlas, seguir las reglas sobre los tipos de datos o asegurarse de que los parámetros de las funciones coincidan entre sus definiciones y las veces que se llama entre otros. Por lo tanto, el análisis sintáctico que se definirá para la gramática del sublenguaje Java obviara lo anteriormente comentado, aunque algunos sí se podrán tener en cuenta. De todas formas, estos casos se explicarán posteriormente cuando se explique lo que reconoce la gramática.

Por último, se han seguido las características del sublenguaje que ha sido definido por la profesora.

6.1 Explicación detallada de la construcción del análisis sintáctico en CUP.

Teniendo ya lo previamente comentado en cuenta, hemos hecho un análisis exhaustivo de cómo organizar la sintaxis del sublenguaje de Java que ha sido propuesto. En este sentido, hemos organizado estructuralmente los distintos bloques sintácticos, nombrando simbólicamente a cada bloque con un no terminal. En este caso, hemos partido del bloque principal a subloques. En este sentido, se destaca los siguientes símbolos no terminales con sus correspondientes explicaiones:

- **declaración_clase.** En primer lugar, todo el bloque de nuestra gramática tiene que empezar con la definición de la clase. En este caso, este símbolo se encarga de comprobar de la definición del bloque. Asimismo, la gramática sería correcta con solo esta primera parte de la estructura o esta estructura, teniendo entre medias una lista de métodos.

- **secuencia_metodos**. Este símbolo no terminal se trata de una regla recursiva que básicamente llama de forma recursiva al símbolo no terminal *estructura_metodo*, que llevará la estructura del método.

- **estructura_metodo**. Este símbolo no terminal compone la estructura que todo método debe de seguir. En este sentido, todos empezarán por PUBLIC STATIC, el tipo de función, la cabecera, y una secuencia de instrucciones, junto con los correspondientes paréntesis y corchetes. Asimismo, también se tiene en cuenta el método Main el cual no tiene una lista de parámetros en la cabecera y puede o no componerse de una secuencia de instrucciones. No obstante, debido a esto, semánticamente no diferencia entre un método void, integer y boolean por lo que la gramática puede aceptar, por ejemplo, un método int que no devuelve ningún return.

- **tipo_función**. Se trata de un no terminal auxiliar que acepta los tipos de datos que puede ser una función. En este las producciones son los tokens de boolean, int y void.

- **lista_parametros_cabecera.** Este no terminal trata una regla recursiva para la definición de los argumentos que se produce durante la definición de un método. En este sentido, se trata de una secuencia de declaraciones de variables los cuales están separados por comas.

- **declaración_variable**. Este no terminal simplemente concatena los tipos de variables que existen (boolean e int) con un identificador.

- **secuencia_instrucciones.** Este no terminal trata una regla recursiva para llevar a cabo la secuenica de las expresiones que son válidas en este sublenguaje.

- **lista_expresiones_cuerpo**. Este no terminal acepta la lista de expresiones válidas en este subleguaje. En este sentido, se acepta como válido la declaración de variables booleanas o de integers, el postincremento o postdecremento de una variable, las asignaciones y los returns, todas ellas acabando en punto y coma. Asimismo, también acepta una serie de expresiones con bucles que serán explicados posteriormente.

- **expresion_return**. No terminal auxiliar que acepta todos los tipos de return posibles. Estos son el return solo, el return seguido de una expresión booleana y el return seguido de una expresión aritmética. Todos estos no terminales se detallarán posteriormente.

- **expresion_bucles**. No terminal auxiliar que agrupa los tipos de bucles que existen en el sublenguaje. En este caso son el bucle for, el bucle do while y el bucle while.

- **expresión_asignacion**. Este no terminal simplemente agrupa a dos tipos de expresiones para las asignaciones. En este caso pueden ser aquellas que tiene la declaración o las que no la tienen.

- **expresión_asignacion_sin_declaracion**. Como indica su nombre, realiza la asignación mediante el token del igual entre una variable y una expresión booleana o aritmética. La explicación de estos terminales se hará posteriormente.

- **expresión_asignación_declaración**. Igual que el anterior, pero haciendo la declaración del tipo en donde sí se controla semánticamente que el tipo de expresión detrás de un boolean sea booleano o el de un integer que sea aritmético. También, acepta como expresión booleana un identificador o llamada a método. Esto se explicará posteriormente, puesto que ha sido el mayor problema a la hora de definir los distintos tipos de expresión.

- **llamada_metodo**. No terminal que se compone de la concatenación de un identificador con unos paréntesis, pudiendo tener entre medias una serie de parámetros.

- **lista_parametros_llamada_metodo.** No terminal que, al igual que otras reglas recursivas que ya han sido explicadas, permite tener unos argumentos de llamadas a métodos separados por comas.

- **argumentos_llamada_metodo.** No terminal que acepta el lenguaje de todos los posibles argumentos que puede tener una llamada a método. En este sentido, puede ser una expresión aritmética, una expresión booleana o una asignación sin declaraciones.

- **expresion_aritmética**. No terminal que acepta el lenguaje de lo que se considera una expresión aritmética. En este caso, como no terminales se tiene un número, una variable con incremento y un identificador o llamada al método. En este sentido, una expresión aritmética es también la combinación de dos expresiones aritméticas, teniendo entre medias de estas los signos de suma y resta y de multiplicación y división, además de que también una expresión aritmética puede estar entre paréntesis, permitiendo de esta forma el anidamiento de estas.

- **identificador_o_llamada_metodo**. No terminal que acepta el lenguaje de una llamada a método o un identificador.

- **expresion_booleana.** Este no terminal abarca todo lo que se considera una expresión booleana. Incluye el literal booleano como punto inicial. Además, una expresión booleana puede ser una combinación de dos expresiones aritméticas con un token de igualdad entre ellas, o dos expresiones booleanas con un token de igualdad. También se acepta una estructura donde dos expresiones aritméticas están separadas por un token de comparación. Del mismo modo, dos expresiones booleanas o dos *identificadores o llamadas a*

métodos, unidos por operadores lógicos, se consideran expresiones booleanas. Estas expresiones pueden ser anidadas y estar entre paréntesis. Es importante destacar que toda expresión booleana puede ser precedida por el operador lógico not, utilizando el símbolo no terminal *expresión_not*, que se detallará en el siguiente punto.

- **expresion_not**. Este no terminal acepta el lenguaje de aquellas expresiones booleana que van precedidos del operador lógico not. En este caso, los no terminales son identificadores o llamadas a métodos, puesto que funcionan como expresiones booleanas. Asimismo, para que una expresión booleana con la precedencia del not puede generarse correctamente, deberá de ir obligatoriamente entre paréntesis, puesto que si no se podrían aceptar casos como este: *! 1 > 2*, cuando debería de ser: *!(1 > 2)*. No obstante, un literal booleano no necesariamente tiene que ir entre paréntesis, por lo que un terminal de expresion_not es un literal booleano. Por último, tiene una producción que admite que comprueba el caso que se daba con el de las expresiones booleanas, es decir, cuando son identificadores o llamadas a métodos directamente.

- **bucle_for**. Este no terminal se encarga de definir la estructura que tiene un bucle for. En este sentido, todo bucle for comienza con dicha palabra reservada, seguido de una cabecera for, y obligatoriamente abre seguido de corchetes, los cuales pueden o no tener una secuencia de instrucciones en su interior. Esto es así para que se permita el anidamiento de las con otros bucles y que se permitan la presencia de las correspondientes expresiones.

- **bucle_do_while**. Este no terminal se encarga de la estructura del bloque do while. Del mismo modo que el bucle for, permite una serie de secuencias de instrucciones entre corchetes, las cuales puede estar o no y la parte del while puede tener expresiones booleneas. Además, se toma el caso de un identificador o llamada a método que se reconoce como booleano.

- **bucle_while.** La explicación de este no terminal es exactamente igual del que se ha comentado previamente. En este sentido, la única diferencia deriva evidentemente en la forma que tiene la estructura del while.

- **cabecera_bucle_for**. Este no terminal consiste en la sentencia que encapsula un bucle for consiste en la concatenación de varias partes: expresiones de asignación con declaración, seguidas de un punto y coma; luego una expresión booleana, seguida de otro punto y coma; y finalmente expresiones de

asignación sin declaración. Es importante tener en cuenta que un identificador o llamada a método se considera booleano, al igual que un identificador con un postincremento como una expresión de asignación sin declaración. Es posible omitir algunas de estas expresiones, ya que lo único constante son los dos puntos y coma que separan las diferentes partes.

6.2 Observaciones generales de la gramática y conclusiones

En primer lugar, la mayor dificultad y problemas que tiene la gramática se presenta en la cuestión de ambigüedad que presenta los términos de identificadores y variables. En este sentido, mediante la sintaxis no es posible discernir si una variable o una llamada a método es una expresión booleana o una expresión aritmética y, en este caso, si ambos no terminales de las expresiones tienen el no terminal de *identificador_o_llamada_metodo*, CUP no podrá generar la gramática, puesto que no sabe a qué tipo de expresión se refiere. Debido a esto, durante el desarrollo de la práctica, hemos tenido muchos conflictos de tipo Reduce/Reduce y Shift/Reduce que han sido muy frustrantes. Por lo tanto, para solventar esta situación, en diversas situaciones donde se ha querido tratar este no terminal como una expresión booleana, se ha tenido que añadir producciones extras, como se han explicado en el anterior punto, para completar estas situaciones de forma que se pueda crear el árbol sintáctico. No obstante, esto provoca que los lenguajes que generan los no terminales sean algo más compleja de seguir en ciertos puntos. De todas formas, si se hubiese hecho más simple, se aceptaría muchas gramáticas no válidas.

Asimismo, para evitar gramáticas ambiguas con las expresiones booleanas y aritméticas cuando se esté generando el árbol sintáctico, ha sido necesario gestionar el orden de precedencia de los distintos operadores. Esto se consigue mediante el operador precedence left, en donde el último token en el orden en el que está presentado, es el que más precedencia tiene. En este caso, el que más tiene es el OR_AND, seguido del EQUALITY, MUL_DIV Y PLUS_SUB.

En conclusión, durante esta práctica hemos podido comprender a grandes rasgos las etapas del funcionamiento de un compilador. En este caso, hemos aprendido como funciona el análisis léxico para los tokens y el análisis sintáctico para la gramática, teniendo en cuenta un poco la semántica.

6.3 Ejemplos de gramáticas con CUP

A continuación, se muestran varias capturas de pantalla en donde se utiliza la herramienta CUP para poder discernir si una gramática es válida o no.

```
1public class Ejem_TAC {
2    public static boolean comparar(int valor, int tope) {
3            int aux;
4            aux = calculaValor(valor);
5            return !(a && hola(a));
6    }
7
8    public static int calculaValor(int valor1) {
9            int aux = 1;
10           for (int i = 0; i < valor1; i++) {
11                   for (int j = i ; j < i; j--) {
12                       aux = aux + 2;
13                   }
14               aux = aux + 1;
15           }
16       return aux;
17   }
18   public static void main() {
19           boolean esCierto = true;
20           int x = 0;
21           while (a && a){
22                   x++;
23                   esCierto = comparar(x,true);
24           }
25   }
26}
```

Console × *Prueba.java parser.java

erminated> Prueba_Automatas [Java Application] /app/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_1

¡ El Análisis es Correcto !!

```
1public class Ejem_TAC {
2    public static boolean comparar(int valor, int tope) {
3            int aux;
4            aux = calculaValor(valor,1);
5            return (a != 2) && !(a < 2);
6    }
7}
```

Console × *Prueba.java parser.java

<terminated> Prueba_Automatas [Java Application] /app/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v20221102-09

¡¡ El Análisis es Correcto !!

```
1public class Ejem_TAC {
2    public static boolean comparar(int valor, int tope) {
3            int aux;
4            aux = calculaValor(valor);
5            return !(a + hola(a));
6    }
7}
```

```
Console ×  *Prueba.java  parser.java

erminated> Prueba_Automatas [Java Application] /app/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_

yntax error
ouldn't repair and continue parse
¡ El análisis es INCORRECTO!!
```

Operador aritmético no puede estar dentro de una expresión booleana

```
1public class Ejem_TAC {
2    public static boolean comparar(int valor, int tope) {
3            int aux;
4            aux = calculaValor(valor,1);
5            return ((a < true) && (a < 2));
6    }
7}
```

```
Console ×  *Prueba.java  parser.java

<terminated> Prueba_Automatas [Java Application] /app/eclipse/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.linux.x86_64_17.0.5.v20221102-093

Syntax error
¡¡ El análisis es INCORRECTO!!
Couldn't repair and continue parse
```

No se puede comparar un booleano literal.