

Construcción de un analizador léxico y sintáctico para un sublenguaje de Java

Iván Illán Barraya

Javier Monescillo Buitrón

7 de mayo de 2019



Índice

1. Introducción	3
2. Descripción del problema	3
3. Solución propuesta	3
4. Diseño del sistema	4
4.1. Lenguaje fuente	4
4.2. Tabla de tokens	5
5. Análisis léxico	6
5.1. JFlex	6
6. Análisis sintáctico	8
6.1. Análisis sintáctico ascendente	8
6.2. Cup	8
6.3. Terminales y No terminales	11
6.3.1. Símbolos terminales	11
6.3.2. Símbolos no terminales	12
7. Cómo ejecutar el proyecto	16

1. Introducción

Java [1] es un lenguaje de programación de propósito general, concurrente y orientado a objetos que fue diseñado específicamente para tener las mínimas dependencias de implementación. Su intención principal es permitir que los desarrolladores de aplicaciones escriban el programa una única vez y lo ejecuten en cualquier dispositivo.

Lo que quiere decir que el código ejecutado en una plataforma no tiene que ser recompilado para correr en otra, así que se puede decir que es un lenguaje compilado e interpretado. Además Java es uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor en la web.

Esto último hace que sea el lenguaje perfecto para poder introducirlo en el problema que se quiere resolver.

2. Descripción del problema

El problema que se presenta es la construcción de un analizador léxico y sintáctico del lenguaje Java usando las herramientas Jflex y Cup [2].

El lenguaje que se propone es un sublenguaje de Java, en concreto una secuencia de métodos en Java que denotaremos como Jjava.

3. Solución propuesta

Para diseñar dichos analizadores, utilizaremos los conocimientos de la materia durante las distintas etapas del proceso.

- Diseño del sistema
- Diseño del Analizador Léxico
 - Identificar Tokens
 - Construcción del Analizador Léxico
- Diseño del Analizador Sintáctico
 - Especificación de la GLC
 - Construcción de la gramática con Cup

La primera etapa consta del diseño referente al lenguaje fuente o arquitectura general del sistema, incluido hasta el análisis léxico. Se proporcionará un diagrama total del problema. Mientras que para la segunda etapa se trata el análisis sintáctico.

4. Diseño del sistema

Para la estructura general del problema se proporciona una pequeña figura [3] donde se puede ver intuitivamente el funcionamiento del mismo.

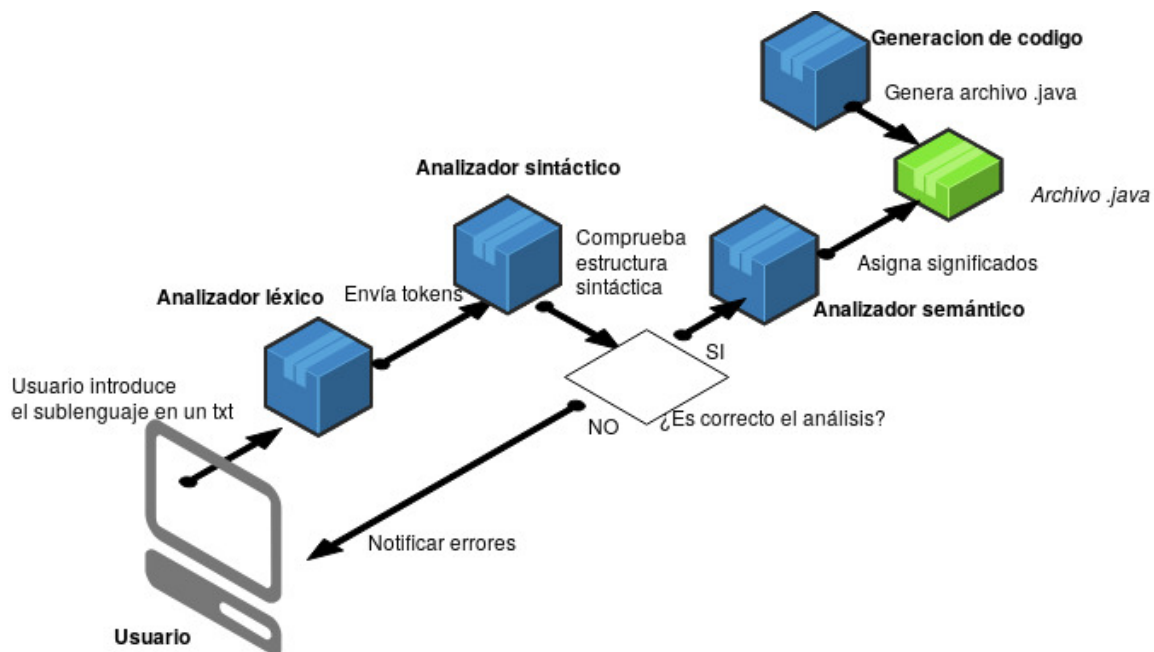


Diagrama general

Actualmente la parte de análisis semántico y generación de código no consta dentro del problema pero se incluye para futuras ampliaciones. Se limitará al mismo a la creación de la fase de análisis léxico y análisis sintáctico.

4.1. Lenguaje fuente

En esta tabla se pretende dar una descripción inicial de los elementos que tendrá el lenguaje.

Elementos del lenguaje	Ejemplo
Tipos de datos	int boolean
Declaración de tipos	int a; int a,b,c;
Instrucción de asignación	type_var a = b + c; type_var a = (Expresión); c = b;
Decremento	var- -; var-=1;
Incremento	var++; var+=1;
Bucle for	for(int i = 0; i<hola.length; i++){ /* Secuencia instrucciones */ }
Llamadas a métodos	metodoVoid();
Retorno de valores	return x; return a+b;
Secuencia de instrucciones	a = b; b * 2; //etc
Operaciones aritméticas	a + b; a - b; a * b; a / b;
Operaciones relacionales	a <b; a <= b; a >= b; a >b; a == b; a!=b;
Operaciones lógicas	a && b a b y !a
Cabecera de los métodos	public static
Tipo devuelto de un método	int, boolean, void

Notese, que para cerrar una sentencia es necesario de indicar al final de dicha sentencia el carácter ';' como se hace típicamente en Java.

4.2. Tabla de tokens

En la siguiente tabla de tokens, se muestran los lexemas de ejemplo, los tokens y las expresiones regulares asociadas a cada token.

Token	Lexema	Patrón
return	return	r·e·t·u·r·n
for	for	f·o·r
int	int	i·n·t
boolean	boolean	b·o·o·l·e·a·n
void	void	v·o·i·d
public static	public static	p·u·b·l·i·c·s·t·a·t·i·c
true	true	t·r·u·e
false	false	f·a·l·s·e
Asignación	=	=
Suma	+	+
División	/	/
Resta	-	-
Multiplicación	*	*
Distinto	!=	!·=
Negación	!	!
Or		·
Incremento	++ +=	+·+ +·=
Decremento	-- -=	-·- -·=
Relacionales	<	< <·= > >·= =·=
Paréntesis abierto	((
Paréntesis cerrado))
Llave abierta	{	{
Llave cerrada	}	}
Punto y coma	;	;
Coma	,	,
Punto	.	.
Asterisco barra	*/	*·/
Barra asterisco	/*	/·*
ID	hola	[A-Za-z][A-Zaz0-9_]*
NUM	4	0 [1-9][0-9]*

5. Análisis léxico

El analizador léxico tiene varias funciones:

- Reconocer los símbolos que componen el texto fuente.
- Eliminar comentarios del texto fuente.
- Eliminar espacios en blanco, saltos de línea, tabulaciones, etc.
- Informar de los errores léxicos detectados

Como salida del analizador léxico, se obtiene una representación de la cadena de entrada en forma de cadena de **tokens**, que será posteriormente utilizada en la fase de análisis sintáctico. Para construir el analizador léxico se ha utilizado JFlex.

JFlex [4] es una herramienta software en la que se declaran los tokens que componen nuestro lenguaje fuente, así como las expresiones regulares asociadas a los mismos para poder generar el analizador léxico.

5.1. JFlex

En esta sección se muestra el código en JFlex donde se construye el analizador léxico del lenguaje Jjava.

Listing 1: Analizador Léxico en JFlex

```
import java.util.*;
import java.io.*;
import java_cup.runtime.Symbol;

%%

%class AnalizadorLexico
%unicode
%cup
%cupdebug

%line
%column

%{

private Symbol symbol(int type) {
return new Symbol(type, yyline, yycolumn);
}

private Symbol symbol(int type, Object value) {
return new Symbol(type, yyline, yycolumn, value);
}

%}

TERMINAR_LINEA = \r|\n|\r\n
```

```

CARACTERIN = [^\r\n]
ESPACIOBLANCO = {TERMINAR_LINEA} | [ \t\f]
COMENTARIO = {COMENTARIOT} | {FINLINEACOMENT}
COMENTARIOT = "/*" [^*] ~"*/" | "*/" "*" + "/";
FINLINEACOMENT = "//" {CARACTERIN}* {TERMINAR_LINEA}?
ID = [a-zA-Z][a-zA-Z0-9_"'-"]*
RELACIONALES = "<" | "<=" | ">" | ">=" | "=="
ASIGNACION = "="
NUM = 0 | [1-9][0-9]*

%%
"--" {return symbol(sym.DECremento, new String(yytext()));}
"*" {return symbol(sym.POR, new String(yytext()));}
"/" {return symbol(sym.DIV, new String(yytext()));}
"-" {return symbol(sym.MENOS, new String(yytext()));}
"+" {return symbol(sym.MAS, new String(yytext()));}
"return" {return symbol(sym.RETURN, new String(yytext()));}
"+=" {return symbol(sym.MASIGUAL, new String(yytext()));}
"-=" {return symbol(sym.MENOSIGUAL, new String(yytext()));}
"for" {return symbol(sym.FOR, new String(yytext()));}
"int" {return symbol(sym.INT, new String(yytext()));}
"boolean" {return symbol(sym.BOOLEAN, new String(yytext()));}
"void" {return symbol(sym.VOID, new String(yytext()));}
"public static" {return symbol(sym.CABECERA_METODOS, new String(yytext()));}
"true" {return symbol(sym.TRUE, new String(yytext()));}
>false" {return symbol(sym.FALSE, new String(yytext()));}
{RELACIONALES} {return symbol(sym.RELACIONALES, new String(yytext()));}
";" {return symbol(sym.PUNTOCOMA, new String(yytext()));}
{ASIGNACION} {return symbol(sym.ASIGNACION, new String(yytext()));}
"++" {return symbol(sym.INCREMENTO, new String(yytext()));}
"&&" {return symbol(sym.AND, new String(yytext()));}
"!" {return symbol(sym.NOT, new String(yytext()));}
"!=" {return symbol(sym.DISTINTO, new String(yytext()));}
"{" {return symbol(sym.LL_A, new String(yytext()));}
"}" {return symbol(sym.LL_C, new String(yytext()));}
"(" {return symbol(sym.PAR_A, new String(yytext()));}
")" {return symbol(sym.PAR_C, new String(yytext()));}
"," {return symbol(sym.COMA, new String(yytext()));}
"." {return symbol(sym.PUNTO, new String(yytext()));}
"||" {return symbol(sym.OR, new String(yytext()));}
{ID} {return symbol(sym.ID, new String(yytext()));}
{COMENTARIO} {/*Ignoramos comentarios*/}
{ESPACIOBLANCO} {/*Ignoramos espacios en blanco*/}
{NUM} {return symbol(sym.NUM, new String(yytext()));}
[^] {System.out.println("Error lexico" + yytext());}

```

6. Análisis sintáctico

Las principales funciones del analizador sintáctico son las siguientes:

- Analizar la secuencia de **tokens** y verificar si son correctos sintácticamente.
- Obtener una representación interna del texto.
- Informar de los errores sintácticos detectados.

En resumen, dada una secuencia de **tokens** obtenida como resultado de la fase de análisis léxico, se comprueba que dicha secuencia está escrita correctamente y se obtiene una representación interna de la misma, que servirá como entrada para el Análisis semántico.

Existen dos estrategias en el *Análisis sintáctico*

- Análisis sintáctico ascendente
- Análisis sintáctico descendente

6.1. Análisis sintáctico ascendente

CUP o (*Construction of Useful Parsers*) [5] es un generador de analizadores LALR para Java. Fue desarrollado por C. Scott Ananian, Frank Flannery, Dan Wang, Andrew W. Appel y Michael Petter. Implementa la generación de analizadores LALR(1) estándar.

La estrategia del análisis sintáctico ascendente funciona construyendo el árbol sintáctico desde las hojas hasta la raíz. Se busca en la cadena de tokens una subcadena que pueda ser reducida a uno de los símbolos no terminales que forman la gramática.

El analizador LALR(1) nace de la simplificación de estados del analizador LR(1). No se entra en detalle de la construcción del AFD reconocedor de prefijos viables ni del analizador sintáctico LR(1) ya que la herramienta CUP realiza el proceso de simplificación de forma autónoma, así como la inclusión de marcadores.

6.2. Cup

En esta sección se muestran tanto las producciones asociadas a la gramática y utilizadas para realizar el análisis sintáctico.

Listing 2: Analizador Sintáctico en CUP

```
import java.io.*;
import java_cup.runtime.*;
import java.util.ArrayList;
import java.util.Hashtable;

parser code {

public void report_error(String message, Object info) {

StringBuffer m = new StringBuffer("Error");
```



```

if (info instanceof java_cup.runtime.Symbol) {
java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
if (s.left >= 0) {
m.append(" en la linea "+(s.left+1));
if (s.right >= 0)
m.append(", columna "+(s.right+1));
}
}

m.append(" : "+message);

System.err.println(m);
}
public void report_fatal_error(String message, Object info) {
System.out.println("Análisis incorrecto");
report_error(message, info);
System.exit(1);
}

:}

non terminal dec_metodos, metodo_simple, dec_argumentos, cuerpo_metodo, dec_for,
    tipo_boolean, metodo_void, tipos_metodos, cuerpo_metodo_void, bloque_inst_void,
    dec_incr, dec_incr_exp;
non terminal tipo_variables, tipo_void, dec_tipos, dec_llamada, dec_exp, dec_exp_n1,
    dec_exp_n2, bloque_inst, inst, generar_tipos, dec_exp_n4, dec_exp_n5,
    exp_metodo;
non terminal dec_return, cuerpo_for, for_st_1, for_st_2, for_st_3, cabecera_for,
    dec_llamada_args, inst_void, dec_asigs, dec_exp_n3, parte_tipos, dec_for_void,
    cuerpo_for_void;

terminal String LL_A, LL_C, FOR ,ID, RETURN, PUNTOCOMA, NUM, PUNTO, OR, DISTINTO,
    DECREMENTO;
terminal String RELACIONALES, ASIGNACION, INCREMENTO, PAR_A, PAR_C, COMA, POR, DIV,
    MENOS, MAS;
terminal String CABECERA_METODOS, AND, NOT, BOOLEAN, TRUE, FALSE, INT, VOID,
    MASIGUAL, MENOSIGUAL, NULL;

/* Metodos */
dec_metodos ::= tipos_metodos | tipos_metodos dec_metodos;

tipos_metodos ::= metodo_simple | metodo_void;

metodo_simple ::= CABECERA_METODOS tipo_variables ID PAR_A dec_argumentos PAR_C
    LL_A cuerpo_metodo LL_C;

metodo_void ::= CABECERA_METODOS tipo_void ID PAR_A dec_argumentos PAR_C LL_A
    cuerpo_metodo_void LL_C;

dec_argumentos ::= tipo_variables ID | tipo_variables ID COMA dec_argumentos | ;

cuerpo_metodo ::= bloque_inst | ;

```

```

cuerpo_metodo_void ::= bloque_inst_void | ;

/* Instrucciones */
bloque_inst ::= inst | inst bloque_inst;

bloque_inst_void ::= inst_void | inst_void bloque_inst_void;

inst ::= dec_for | dec_llamada | dec_tipos| dec_asigs | dec_return | dec_incr_exp;

inst_void ::= dec_for_void | dec_llamada | dec_tipos | dec_asigs;

/* For */
dec_for ::= FOR PAR_A cabecera_for PAR_C LL_A cuerpo_for LL_C;

dec_for_void ::= FOR PAR_A cabecera_for PAR_C LL_A cuerpo_for_void LL_C;

cabecera_for ::= for_st_1 PUNTOCOMA for_st_2 PUNTOCOMA for_st_3 ;

for_st_1 ::= tipo_variables ID ASIGNACION dec_exp | ID ASIGNACION dec_exp;

for_st_2 ::= ID RELACIONALES dec_exp | ID DISTINTO dec_exp;

for_st_3 ::= dec_incr | ID ASIGNACION dec_exp;

cuerpo_for ::= bloque_inst | ;

cuerpo_for_void ::= bloque_inst_void | ;

/*Generar tipos */
dec_tipos ::= parte_tipos PUNTOCOMA;

parte_tipos ::= tipo_variables generar_tipos;

generar_tipos ::= ID | ID COMA generar_tipos;

/*Asignaciones */
dec_asigs ::= parte_tipos ASIGNACION dec_exp PUNTOCOMA
| ID ASIGNACION dec_exp PUNTOCOMA;

/*Llamadas a metodos */
dec_llamada ::= exp_metodo PUNTOCOMA;

dec_incr_exp ::= dec_incr PUNTOCOMA;

dec_incr ::= ID INCREMENTO | ID DECREMENTO | ID MASIGUAL dec_exp | ID MENOSIGUAL
dec_exp;

exp_metodo ::= ID PAR_A dec_llamada_args PAR_C;

dec_llamada_args ::= dec_exp | dec_exp COMA dec_llamada_args | ;

dec_return ::= RETURN dec_exp PUNTOCOMA;

```

```

/* Expresiones */
dec_exp ::= dec_exp OR dec_exp_n1 | dec_exp_n1;

dec_exp_n1 ::= dec_exp_n1 AND dec_exp_n2 | dec_exp_n2;

dec_exp_n2 ::= dec_exp_n2 RELACIONALES dec_exp_n3 | dec_exp_n2 DISTINTO dec_exp_n3
| dec_exp_n3;

dec_exp_n3 ::= dec_exp_n3 MAS dec_exp_n4 | dec_exp_n3 MENOS dec_exp_n4 | dec_exp_n4;

dec_exp_n4 ::= dec_exp_n4 POR dec_exp_n5 | dec_exp_n4 DIV dec_exp_n5 | dec_exp_n5;

dec_exp_n5 ::= MENOS dec_exp_n5 | NOT dec_exp_n5 | PAR_A dec_exp PAR_C | NUM | ID |
ID PUNTO ID | tipo_boolean | exp_metodo;

/*Tipos*/
tipo_variables ::= INT | BOOLEAN ;

tipo_void ::= VOID;

tipo_boolean ::= TRUE | FALSE;

```

6.3. Terminales y No terminales

6.3.1. Símbolos terminales

Los símbolos terminales son expresados en mayúsculas durante la escritura de la gramática, se presenta una lista de todos los símbolos terminales de la gramática con su respectivo token asociado.

Terminales en Cup	Token asociado
ID	Identificador
LL_A	{
LL_C	}
PAR_A	(
PAR_C)
FOR	for
RETURN	return
PUNTOCOMA	;
NUM	4
PUNTO	.
OR	
DISTINTO	!=
DECREMENTO	--
RELACIONALES	< <= > >= ==
ASIGNACION	=
INCREMENTO	++
COMA	,
POR	for
DIV	/
MENOS	-
MAS	+
CABECERA_METODOS	public static
AND	&&
NOT	!
BOOLEAN	boolean
TRUE	true
FALSE	false
INT	int
VOID	void
MASIGUAL	+=
MENOSIGUAL	-=

6.3.2. Símbolos no terminales

La gramática comienza con la producción *dec_metodos* por la cual podemos generar un único método o una lista de métodos.

```
dec_metodos ::= tipos_metodos | tipos_metodos dec_metodos;
```

Un método puede retornar un tipo (int o boolean) o ser de tipo void:

```
tipos_metodos ::= metodo_simple | metodo_void;
```

```
metodo_simple ::= CABECERA_METODOS tipo_variables ID PAR_A dec_argumentos PAR_C
LL_A cuerpo_metodo LL_C;
```

```
metodo_void ::= CABECERA_METODOS tipo_void ID PAR_A dec_argumentos PAR_C LL_A
cuerpo_metodo_void LL_C;
```

La producción *tipo_variables* es una producción que tiene los símbolos terminales de retorno,

al igual que *tipo_void* contiene el tipo de no retorno.

Los argumentos se declaran como un único argumento o una lista de argumentos (un argumento seguido de una coma) en la producción *dec_argumentos*.

```
dec_argumentos ::= tipo_variables ID | tipo_variables ID COMA dec_argumentos | ;
```

Ambos métodos tienen un cuerpo que puede estar vacío o bien puede tener un bloque de instrucciones que contendrá las instrucciones de cada método. Podrá ser una única instrucción o bien una lista de instrucciones.

```
cuerpo_metodo ::= bloque_inst | ;
```

```
cuerpo_metodo_void ::= bloque_inst_void | ;
```

```
/* Instrucciones */
```

```
bloque_inst ::= inst | inst bloque_inst;
```

```
bloque_inst_void ::= inst_void | inst_void bloque_inst_void;
```

La diferencia entre *inst_void* e *inst* no es más que la instrucción de retorno *dec_return*.

```
inst ::= dec_for | dec_llamada | dec_tipos | dec_asigs | dec_return | dec_incr_exp;
```

```
inst_void ::= dec_for_void | dec_llamada | dec_tipos | dec_asigs | dec_incr_exp;
```

Estas son todas las instrucciones explicadas y desglosadas en la gramática.

dec_for

```
/* For */
```

```
dec_for ::= FOR PAR_A cabecera_for PAR_C LL_A cuerpo_for LL_C;
```

```
dec_for_void ::= FOR PAR_A cabecera_for PAR_C LL_A cuerpo_for_void LL_C;
```

```
cabecera_for ::= for_st_1 PUNTOCOMA for_st_2 PUNTOCOMA for_st_3 ;
```

```
for_st_1 ::= tipo_variables ID ASIGNACION dec_exp | ID ASIGNACION dec_exp;
```

```
for_st_2 ::= ID RELACIONALES dec_exp | ID DISTINTO dec_exp;
```

```
for_st_3 ::= dec_incr | ID ASIGNACION dec_exp;
```

```
cuerpo_for ::= bloque_inst | ;
```

```
cuerpo_for_void ::= bloque_inst_void | ;
```

Dentro de la instrucción *for* se pueden observar las producciones auxiliares *cabecera_for* que subdivide en otras tres producciones la parte de la creación del *for* (*int a = 0; i < 2*3; i++*) en las producciones *for_st_1*, *for_st_2*, y *for_st_3*. Cada una de ellas subdivide en las diferentes expresiones que tiene el *for*.

En el primer bloque hay una asignación a una variable que es una expresión, las expresiones se realizan en la producción *dec_exp*, en el segundo bloque solo podrá haber expresiones relacionales y en el tercer bloque podrá haber expresiones e incrementos.

Respecto al cuerpo de cada metodo no pueden existir un for con instrucciones de return, por ello se crean dos declaraciones de for, una void y otra con retorno.

dec_tipos

```
/*Generar tipos */
dec_tipos ::= parte_tipos PUNTOCOMA;

parte_tipos ::= tipo_variables generar_tipos;

generar_tipos ::= ID | ID COMA generar_tipos;
```

La declaración de tipos puede ser un único tipo como `int a`; o de varios `int a,b,c`;. Esto se hace con las producciones *parte_tipos* que hacen uso de otras producciones como *tipo_variables* y *generar_tipos*.

dec_asigs

```
/*Asignaciones */
dec_asigs ::= parte_tipos ASIGNACION dec_exp PUNTOCOMA
| ID ASIGNACION dec_exp PUNTOCOMA;
```

Las asignaciones pueden llevar asociada una declaración de tipos previa o un identificador. Se asignan expresiones con la producción *dec_exp*.

dec_llamada

```
/*Llamadas a metodos */
dec_llamada ::= exp_metodo PUNTOCOMA;

exp_metodo ::= ID PAR_A dec_llamada_args PAR_C;

dec_llamada_args ::= dec_exp | dec_exp COMA dec_llamada_args | ;
```

Las llamadas a los métodos se hacen con la producción *exp_metodo*. Los argumentos que se introducen a las llamadas a metodos son una única expresión o una lista de expresiones.

dec_incr_exp

```
dec_incr_exp ::= dec_incr PUNTOCOMA;

dec_incr ::= ID INCREMENTO | ID DECREMENTO | ID MASIGUAL dec_exp | ID MENOSIGUAL
dec_exp;
```

Los incrementos que forman parte del tercer bloque en una instrucción `for` o en cualquier tipo de operación, se declaran mediante la expresión *dec_incr*.

dec_return

```
dec_return ::= RETURN dec_exp PUNTOCOMA;
```

La instrucción `return` es simple, solo retorna una expresión seguida de punto y coma.

dec_exp

/ Expresiones */*

dec_exp ::= dec_exp OR dec_exp_n1 | dec_exp_n1;

dec_exp_n1 ::= dec_exp_n1 AND dec_exp_n2 | dec_exp_n2;

dec_exp_n2 ::= dec_exp_n2 RELACIONALES dec_exp_n3 | dec_exp_n2 DISTINTO dec_exp_n3
| dec_exp_n3;

dec_exp_n3 ::= dec_exp_n3 MAS dec_exp_n4 | dec_exp_n3 MENOS dec_exp_n4 | dec_exp_n4;

dec_exp_n4 ::= dec_exp_n4 POR dec_exp_n5 | dec_exp_n4 DIV dec_exp_n5 | dec_exp_n5;

dec_exp_n5 ::= MENOS dec_exp_n5 | NOT dec_exp_n5 | PAR_A dec_exp PAR_C | NUM | ID |
ID PUNTO ID | tipo_boolean | exp_metodo;

La producción de declaración de expresiones consta de varios niveles, en concreto de n1 a n5, en ella se declaran todos los tipos de expresiones aritméticas y lógicas. En el primer nivel se encuentran las operaciones de menor prioridad, y a medida que se avanza se encuentran las más prioritarias. En el último nivel están las operaciones unarias, la llamada a métodos, los tipos de variable booleana, los identificadores, los números y la negación lógica.

Tipos

*/*Tipos*/*

tipo_variables ::= INT | BOOLEAN ;

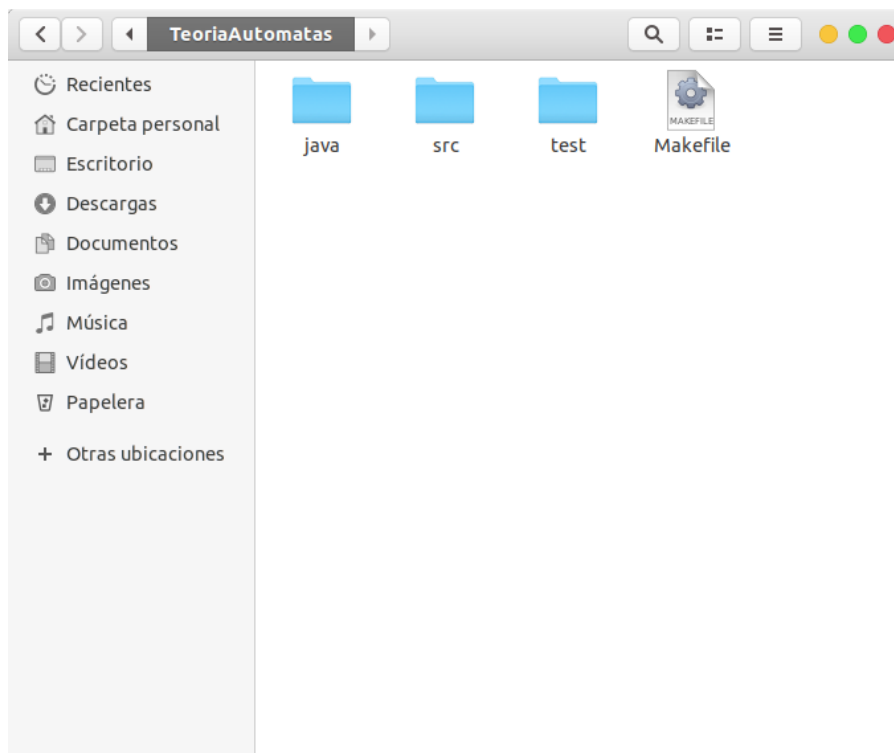
tipo_void ::= VOID;

tipo_boolean ::= TRUE | FALSE;

Los tipos que son utilizados a lo largo de toda la gramática.

7. Cómo ejecutar el proyecto

Situese dentro del directorio **TeoriaAutomatas/**



Directorio **TeoriaAutomatas/**

Ejecute:

```
make
```

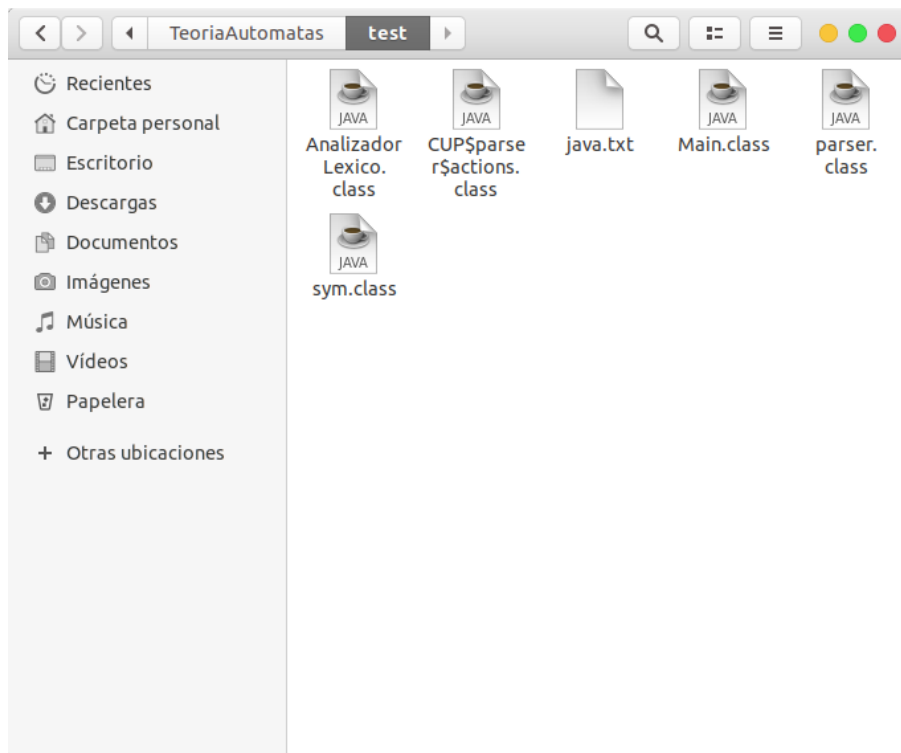
Si no lo tiene instalado lance el siguiente comando:

```
sudo apt-get update  
sudo apt-get install make
```

Este *Makefile* nos ahorrará la compilación de los ficheros, lo que hace es compilar el archivo *flex* y *cup* que se encuentran en *src/* en el directorio *java/*. Luego compila los archivos *.java* en el directorio *test/*.

Para ejecutar la prueba entre en el directorio *test/*

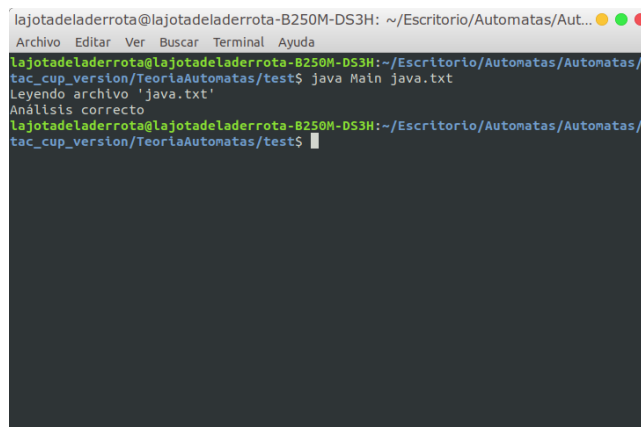
```
cd test/
```

Directorio test/

Para ejecutar un fichero .txt lance el siguiente comando:

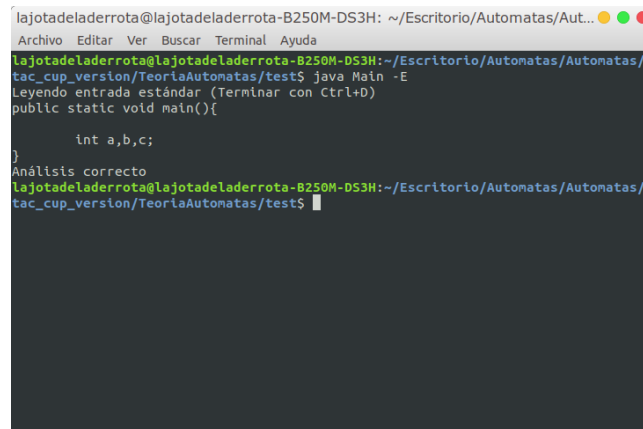
```
java Main java.txt
```



Ejecutando fichero

En el caso de que usted desee escribir por la entrada estándar el lenguaje, entonces tiene que lanzar este comando.

```
java Main -E
```



```
lajotadeladerrota@lajotadeladerrota-B250M-DS3H: ~/Escritorio/Automatas/Aut...
Archivo Editar Ver Buscar Terminal Ayuda
lajotadeladerrota@lajotadeladerrota-B250M-DS3H:~/Escritorio/Automatas/Automatas/
tac_cup_version/TeoriaAutomatas/test$ java Main -E
Leyendo entrada estándar (Terminar con Ctrl+D)
public static void main(){
    int a,b,c;
}
Análisis correcto
lajotadeladerrota@lajotadeladerrota-B250M-DS3H:~/Escritorio/Automatas/Automatas/
tac_cup_version/TeoriaAutomatas/test$
```

Ejecutando desde la entrada estándar

Referencias

- [1] James Gosling, Bill Joy, Guy Steele, y Gilad Bracha, The Java language specification, tercera edición. Addison-Wesley, 2005. ISBN 0-321-24678-0.
- [2] Java a Tope: Traductores Y Compiladores Con Lex/yacc, Jflex/cup Y Javacc, Rojas, Sergio Gálvez and Mata, Miguel Ángel Mora, 2005
- [3] Draw.io, Alder Gaudenz, Benson David, En <https://about.draw.io/> 2019
- [4] Jflex, lexical analyzer generator for Java, Klein Gerwin, Rowe Steve, Décamps Régis. En <https://www.jflex.de/>, 2018
- [5] CUP, C. Scott Ananian, Frank Flannery, Dan Wang, Andrew W. Appel y Michael Petter. En <http://www2.cs.tum.edu/projects/cup/index.php>