PRACTICA NO. 5 SINTACTICO

Integrantes:
Patricia Fernanda Castellanos Ruiz
Mayra Esmeralda Castillo Montenegro
Jesús Eduardo Silva Vázquez

Reporte individual:

Reporte de Jesús Silva:

En esta práctica me di a la tarea de realizar el analizador sintáctico por medio de las herramientas proporcionadas por JavaCC, un generador de analizadores sintácticos muy similar a la sistemática manejada en analizadores léxicos como Flex o Yacc, pude realizar mis propios analizadores y dar una explicación paso por paso de cómo utilizar la herramienta y demostrar el funcionamiento.

Reporte de Mayra Castillo:

En esta práctica me di a la tarea de investigar las herramientas que se utilizan para simular la etapa de análisis sintáctico, además de realizar el cuadro comparativo para poder identificar las ventajas y desventajas de cada herramienta, seleccionamos tres: JavaCC, ANTLR, y Yacc.

Reporte de Patricia Castellanos:

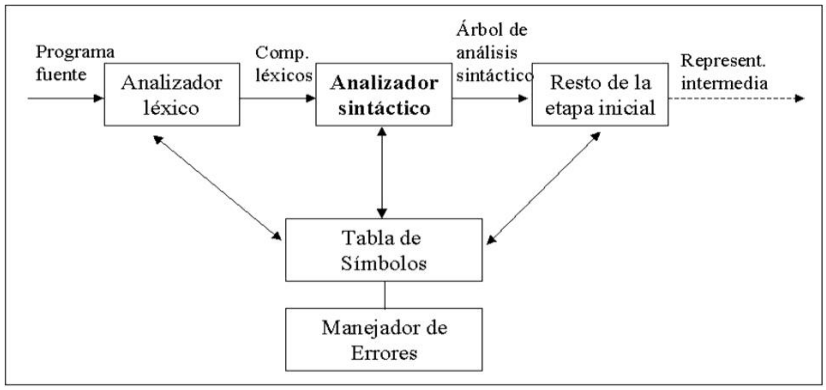
En este reporte realice la definición del analizador sintáctico ya que con esto se desarrolló la herramienta que utilizamos o sea JavaCC para la simulación del analizador sintáctico y también hice una investigación a fondo de la misma herramienta e incluí los diagrama de sintaxis que ejemplifican la estructura sintáctica de los analizadores realizados.

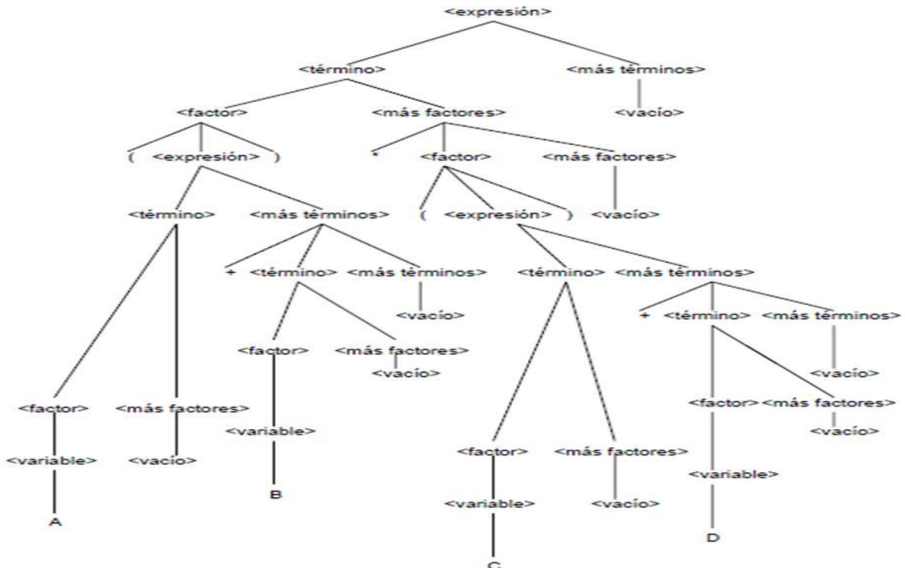
Función del análisis sintáctico.

El análisis sintáctico tiene como **función principal** generar un árbol de análisis sintáctico a partir de los componentes léxicos obtenidos en la etapa anterior.

El análisis sintáctico (en inglés parser) es un análisis a nivel de sentencias, y es mucho más complejo que el análisis léxico. Su función es tomar el programa fuente en forma de tokens, que recibe del analizador léxico y determinar la estructura de las sentencias del programa.

El análisis sintáctico agrupa a los tokens en clases sintácticas (denominadas no terminales en la definición de la gramática), tales como expresiones, procedimientos, etc., y obtiene un árbol sintáctico (u otra estructura equivalente) en la cual las hojas son los tokens y cualquier nodo, que no sea una hoja, representa un tipo de clase sintáctica.





Herramientas para el desarrollo del análisis sintáctico

HERRAMIENTA	VENTAJAS	DESVENTAJAS
JAVACC	• Genera analizadores sintácticos descendentes basados en la gramática • Incluye la herramienta jjTREE para generar arboles sintácticos • Buena integración y los analizadores léxicos y sintáctico	• Genera analizadores menos eficientes que los generadores por YACC • Localizar errores del programa • Corregir errores en los ficheros de especificaciones
ANTLR	• Buena integración de los analizadores léxicos y sintácticos • El código generado por ANTLR es mas fácil de entender y depurar • Las especificaciones gramaticales de ANTLR permiten la notación BNF y generan arboles de análisis sintáctico	• Genera analizadores menos eficientes que los generados por YACC • Los ficheros de especificación de ANTLR son muy complejas
LEX/YACC	• Genera analizadores eficientes, incluso más que los que pudiéramos hacer de manera manual • Reconocen la mayor parte de los lenguajes	• Usa herramientas externas para que le provean los tokens necesarios • No genera arboles de análisis sintáctico • Mezcla las especificaciones sintácticas con las semánticas

JAVACC

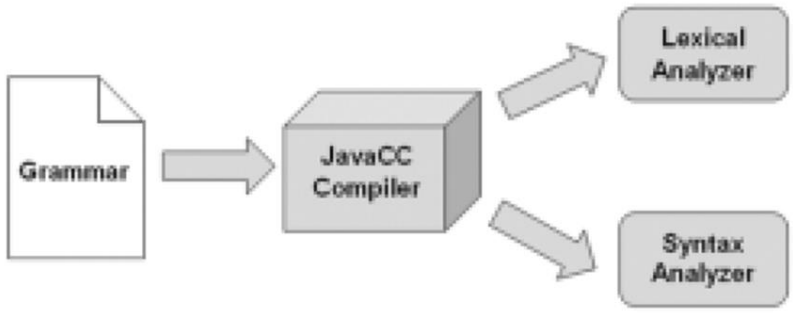
JavaCC (Java Compiler Compiler) es un generador de analizadores sintácticos de código abierto para el lenguaje de programación Java. JavaCC es similar a Yacc en que genera un parser para una gramática presentada en notación BNF, con la diferencia de que la salida es en código Java. A diferencia de Yacc, JavaCC genera analizadores descendentes (top-down), lo que lo limita a la clase de gramáticas LL(K) (en particular, la recursión desde izquierda no se puede usar). El constructor de árboles que lo acompaña, JJTree, construye árboles de abajo hacia arriba (bottom-up).

JavaCC integra en una misma herramienta al analizador lexicográfico y al sintáctico, y el código que genera es independiente de cualquier biblioteca externa, lo que le confiere una interesante propiedad de independencia respecto al entorno. A grandes rasgos, sus principales características son las siguientes:

• Genera analizadores descendentes, permitiendo el uso de gramáticas depropósito general y la la utilización de atributos tanto sintetizados como heredados durante la construcción del árbol sintáctico.

• Las especificaciones léxicas y sintácticas se ubican en un solo archivo. De esta manera la gramática puede ser leída y mantenida más fácilmente. No obstante, cuando se introducen acciones semánticas, recomendamos el uso de ciertos comentarios para mejorar la legibilidad.

• Admite el uso de estados léxicos y la capacidad de agregar acciones léxicas incluyendo un bloque de código Java tras el identificador de un token.



ANTLR

Es un potente generador de analizadores para leer, procesar, ejecutar o traducir texto binario o texto estructurado. Es ampliamente utilizado para construir lenguajes, herramientas y marcos. Desde una gramática, ANTLR genera un analizador que puede construir y caminar árboles de análisis.

Bibliotecas en tiempo de ejecución y objetivos de generación de código

La herramienta Antlr está escrita en Java, sin embargo, es capaz de generar analizadores y lexers en varios idiomas. Para ejecutar el analizador y el lexer también necesitará tener la biblioteca de ejecución de antlr junto con el analizador y el código del lexer. El idioma de destino admitido (y las bibliotecas de tiempo de ejecución) son los siguientes:

- Java
- DO
- Python (2 y 3)
- JavaScript

Hola Mundo

Una simple gramática de hola mundo se puede encontrar aquí :

```
// define a grammar called Hello
grammar Hello;
r   : 'hello' ID;
ID  : [a-z]+ ;
WS  : [ \t\r\n]+ -> skip ;
```

Para compilar este ejemplo .g4, puede ejecutar el siguiente comando desde su terminal / línea de comandos de sistemas operativos:

```
Java -jar antlr-4.5.3-complete.jar Hello.g4

//OR if you have setup an alias or use the recommended batc

antlr4 Hello.g4
```

La creación de este ejemplo debería generar el siguiente resultado en el directorio de archivos Hello.g4:

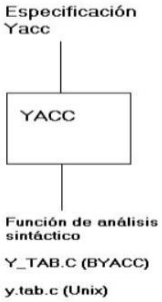
1. Hello.tokens
2. HelloBaseListener.java
3. HelloLexer.java
4. HelloLexer.tokens
5. HelloListener.java
6. HelloParser.java

Cuando utilice estos archivos en su propio proyecto, asegúrese de incluir el archivo jar ANTLR. Para compilar todos estos archivos utilizando Java, en el mismo directorio operativo o por ruta, ejecute el siguiente comando:

```
javac *.java
```

YACC

Entorno del YACC:



Para ejecutar el yacc hay que introducir desde la línea de comandos:

BYACC [opciones] fichero_especificaciones

Las opciones son:

-v	Genera la tabla de análisis sintáctico en el fichero y.out
-b prefijo	Permite cambiar el prefijo y del nombre del fichero de salida por el especificado
-t	Permite realizar un "debugging" del análisis sintácico
-d	Genera el fichero de cabecera y_tab.h, que contiene la definición de los tokens
-r	Genera dos ficheros de salida, y_code.c que contiene el código y y_tab.c que contiene la tabla de análisis

Para realizar la función del analizador de léxico, el Yacc utiliza una función entera llamada yylex(), cuyo valor se debe asociar a la variable yyval

El analizador de léxico:

- ✓ puede ser la función generada con LEX, o
- ✓ una función definida por el usuario en la zona de rutinas de usuario

La estructura del fichero de especificaciones para el YACC es:

{ Declaraciones de los símbolos }

%%

Reglas gramaticales {+acciones}

%%
{ Rutinas de usuario }
Se pueden incluir comentarios con el siguiente formato:

/* cero o más caracteres */

Analizador sintáctico hecho en JavaCC

Para realizar el analizador léxico se optó por utilizar JavaCC, un generador de analizadores sintácticos escrito en lenguaje Java, es indispensable contar con las librerías JDK y JRE de Java para su funcionamiento.

Link de descarga de JavaCC: https://javacc.github.io/javacc/
Link de descarga de JDK y JRE: https://docs.oracle.com/javase/10/install/installation-jdk-and-jre-microsoft-windows-platforms.htm#JSJIG-GUID-A7E27B90-A28D-4237-9383-A58B416071CA

En el desarrolló del analizador sintáctico se optó por utilizar dos códigos, uno simple y otro complejo, que serán vistos a continuación:

Pasos para generar un analizador sintáctico:

1. Para utilizar JavaCC se requiere crear un código escrito en java con extensión ".jj" el cual se convertirá en nuestro analizador sintáctico

```
PARSER_BEGIN(SyntaxChecker)

public class SyntaxChecker {
    public static void main(String[] args) {
        try {
            new SyntaxChecker(new java.io.StringReader(args[0])).S();
            System.out.println("Syntax is okay");
        } catch (Throwable e) {
            // Catching Throwable is ugly but JavaCC throws Error objects!
            System.out.println("Syntax check failed: " + e.getMessage());
        }
    }
}

PARSER_END(SyntaxChecker)

SKIP:  { " " | "\t" | "\n" | "\r"                    }
TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }

void S(): {} { E() <EOF>           }
void E(): {} { T() ("+" T())*      }
void T(): {} { F() ("*" F())*      }
void F(): {} { <NUM> | "(" E() ")" }
```

El primer código es un analizador de expresiones aritméticas que determinará si la expresión ingresada es correcta o incorrecta.

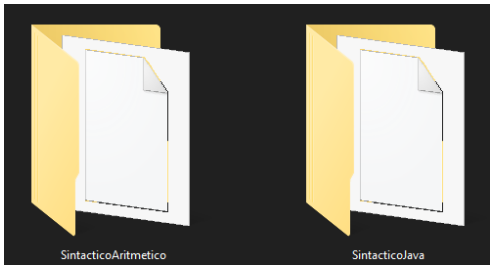
```
PARSER_BEGIN(comp)
import java.io.*;
class comp
{
        public static void main( String[] args )throws ParseException, Exception
        {
                try
                {
                comp analizador = new comp( System.in ) ;
                        analizador.Programa();
                        System.out.println("\tAnalizador ha terminado.");
                }
                catch(ParseException e)
                {
                        System.out.println(e.getMessage());
                        System.out.println("\tAnalizador ha terminado.");
                }
        }
}
PARSER_END(comp)

TOKEN :
{
          <ASIGNACION : "="> //1
        | <PLUS : "+" >  //2
        | <MINUS:"-"> //3
        | <MULTIPLY:"*">  //4
        | <DIVIDE:"/"> //5
        | <INCR: "++" > //6
        | <DECR: "--" > //7

}
TOKEN:
{
          <PUBLIC: "public"> //8
        | <PRIVATE: "private"> //9
        | <STATIC: "static"> //10
        | <VOID: "void"> //11
        | <MAIN: "public static void Main()"> //12
        | <PROGRAMA: "Programa" >  //13
```

El segundo código es un analizador sintáctico que analizará la estructura de un código escrito en Java (es más extenso el código pero será incluido en la documentación de la práctica).

2. Una vez definido el/los códigos, estos deben ser almacenados en una carpeta de forma individual:



3. Ahora abrimos el cmd y nos dirigimos a la carpeta en donde está almacenado un código (este proceso es el mismo para ambos ejemplos):

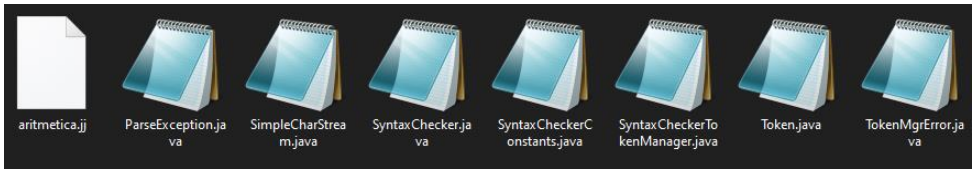
```
C:\Users\JesusEduardo>cd C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico

C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>
```

4. Después tenemos que ingresar el comando “javacc” con el nombre de nuestro archivo.

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>javacc aritmetica.jj
Java Compiler Compiler Version 5.0 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file aritmetica.jj . . .
File "TokenMgrError.java" does not exist.  Will create one.
File "ParseException.java" does not exist.  Will create one.
File "Token.java" does not exist.  Will create one.
File "SimpleCharStream.java" does not exist.  Will create one.
Parser generated successfully.
```

Si el proceso fue realizado correctamente, dentro de la carpeta que contenía el código con extensión “.jj” aparecerán los siguientes archivos:

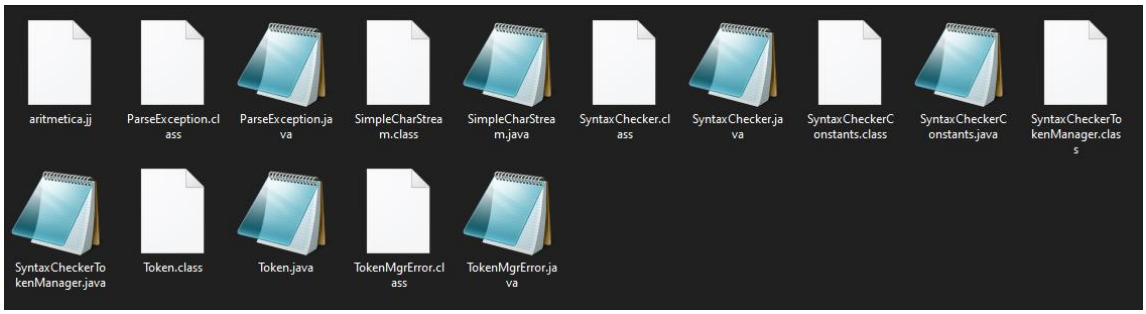


Estos archivos son extensiones del archivo “main” o base que en nuestro caso sería “SyntaxCheker.java” ya que en el código “aritmética.jj” se declaró como base principal dicho archivo.

5. Ahora se tienen que generar clases de todos los archivos de tipo Java para que funcionen dentro del cmd, para ello se utiliza el comando “javac *.java”.

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>javac *.java
```

El comando nos generará archivos con extensión “.class” los cuales estarán listos para su uso.



6. Finalmente, para probar el código, se tiene que ejecutar el comando “java SyntaxCheker” y seguido la expresión aritmética que queremos comprobar:

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>java SyntaxChecker 1+1
Syntax is okay
```

Si la expresión es correcta, el programa lo indicará enviando un mensaje diciendo “Syntax is okay” pero al comprobar una expresión errónea o incompleta nos indica el siguiente error:

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>java SyntaxChecker 1+
Syntax check failed: Encountered "<EOF>" at line 1, column 2.
Was expecting one of:
    "(" ...
    <NUM> ...
```

Lo cual se refiere a que después del “1+” debe existir otro componente y como sugerencia el analizador sintáctico indica que puede ser un “(” o un número.

Ejemplos de expresiones correctas:

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>java SyntaxChecker (2+1) / 4
Syntax is okay

C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>java SyntaxChecker (5)+(7*3)
Syntax is okay

C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>java SyntaxChecker 1+2*3 = 7
Syntax is okay
```

Ejemplos de expresiones incorrectas:

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>java SyntaxChecker ((1
Syntax check failed: Encountered "<EOF>" at line 1, column 3.
Was expecting one of:
    ")" ...
    "+" ...
    "*" ...


C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoAritmetico>java SyntaxChecker ((1*3)
Syntax check failed: Encountered "<EOF>" at line 1, column 6.
Was expecting one of:
    ")" ...
    "+" ...
    "*" ...
```

Estructura Léxica del analizador.

El archivo con extensión “.jj” (en nuestro caso el de ambos analizadores sintácticos) contiene una estructura de tokens válidos, ya que sin ella no se podría analizar sintácticamente si nuestro programa está correctamente compilado o no.

El analizador sintáctico aritmético contiene una estructura de tokens muy simple:

```
TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
```

Mientras que el analizador sintáctico de un código escrito en Java contiene una estructura más completa y específica:

```
TOKEN :
{
          <ASIGNACION : "="> //1
        | <PLUS : "+" >  //2
        | <MINUS:"-"> //3
        | <MULTIPLY:"*">  //4
        | <DIVIDE:"/"> //5
        | <INCR: "++" > //6
        | <DECR: "--" > //7

}
TOKEN:
{
          <PUBLIC: "public"> //8
        | <PRIVATE: "private"> //9
        | <STATIC: "static"> //10
        | <VOID: "void"> //11
        | <MAIN: "public static void Main()"> //12
        | <PROGRAMA: "Programa" >  //13
        | <IF: "ien" > //14
        | <ELSE:"iotro" > //15
        | <ELSEIF:"ien otro" > //16
        | <FOR:"irepetir" > //17
        | <SWITCH:"ialternativa" > //18
        | <CASE:"iopcion" > //19
        | <BREAK:"itermina" > //20
        | <DEFAUL:"ipredef" > //21
        | <DO:"ia" > //22
        | <WHILE:"iespera" > //23
        | <WRITE:"iescribir" > //24
        | <READ:"ileer" > //25
}
```

```
TOKEN:
{
      <LPAREN: "(" > //26
        | <RPAREN: ")" > //27
        | <LBRACE: "{" > //28
        | <RBRACE: "}" > //29
        | <LBRACKET: "[" > //30
        | <RBRACKET: "]" > //31
        | <SEMICOLON: ";" > //32
        | <COMMA: "," > //33
        | <DOT: "." > //34
        | <DOUBLECOMMA: "\""> //35
}
TOKEN:
{
          <EQ: "==" > //36
        | <LE: "<=" > //37
        | <MN: "<" > //38
        | <GE: ">=" > //39
        | <GR: ">" > //40
        | <NE: "!=" > //41
        | <OR: "||" > //42
        | <AND: "&&"> //43
}
```

```
TOKEN:
{
          <INT:"inum"> //44
        | <DEC: "idec"> //45
        | <CHR: "ichr"> //46
        | <STR: "istr"> //47
        | <NUMBER : (["0"-"9"])+>  //48
        | <IDENTIFIER : ["a"-"z","A"-"Z"](["a"-"z","A"-"Z","0"-"9","_"])*>  //49
        | <DECIMAL : (["0"-"9"])+["."](["0"-"9"])+ > //50
        | <CADENA : <DOUBLECOMMA>["a"-"z","A"-"Z","0"-"9"," "](["a"-"z","A"-"Z","0"-"9"," ",":",";","."])*<DOUBLECOMMA>> //51
        | <CARAC : "'"["a"-"z","A"-"Z","0"-"9"]"'" >//52
        | <DOUBLEPOINT : ":"> //53
}
```

Estructura Sintática del analizador.

En el caso del segundo código, se trata de un analizador sintáctico de variables y/o expresiones en un código escrito en Java, puede analizar códigos desde un archivo “.txt” y determinar si existe error alguno, este es el archivo .txt puesto a prueba:

```
Programa test
{
	public static void Main()
	{
		istr s1;
		ichr c1 = '1', c2;
		inum n1 = 0, n2 = 2, n3 = 2, x;
		idec d1;


		irepetir(n2 = 0; n2 < 10; n2++ )
		{
			iescribir(n2);
		}

		ien( n1 >= 0 )
		{
			n1 = n3 / n2;
		}
		ien otro(n1 < 10)
		{
			n1 = n2 - n3;
		}
		iotro
		{
			iescribir("Ninguna opcion");
		}

		ia
		{
			x = x + 1;
			iescribir(x);
		}
		iespera(x < 5)

		iespera(n1 <= 20)
		{
			n1++;
			iescribir(n1);
		}

		ialternativa(n3)
		{
			iopcion 3:
				n1 = 2 + 2;
			itermina;
			ipredef:
				n1 = 10;
			itermina;
		}
	}
}
```

Cuando ejecutamos el analizador sintáctico para qué lea el archivo, nos indicará que finalizó el análisis sin ningún otro mensaje, esto quiere decir que no encontró error sintáctico alguno:

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoJava>java comp < code.txt
        Analizador ha terminado.
```

Y si queremos comprobar que reconoce errores, solo basta con editar de manera incorrecta el código y al volverlo a analizar, detectará errores:

```
C:\Users\JesusEduardo\Pictures\Analizadores\SintacticoJava>java comp < code.txt
Was expecting:
    "{" ...
Error: No se puede convertir 1 a Caracter
Linea: 6
Error: El identificador n1 No ha sido declarado
Linea: 7
Encountered " "," ", "" at line 7, column 23.
Was expecting:
    ";" ...

        Analizador ha terminado.
```

En este caso detectó 4 errores, cada error seguido de su ubicación debido a que:

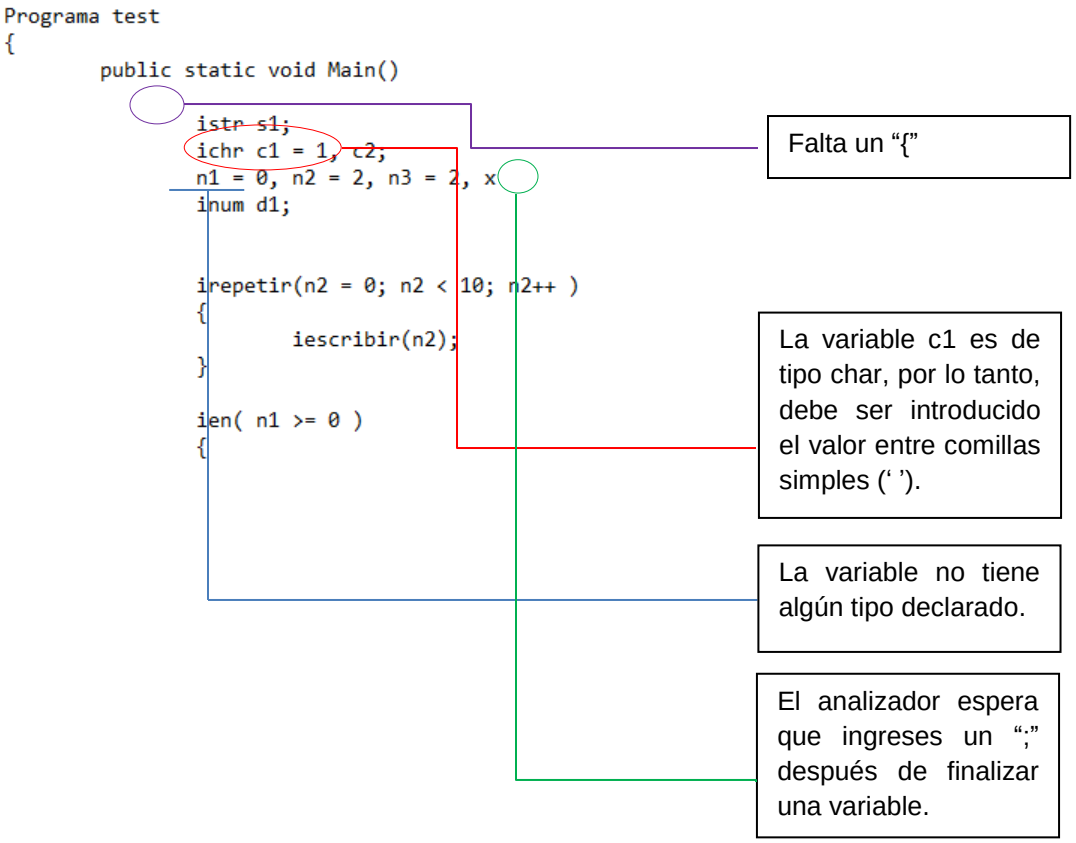


Diagrama de sintaxis:

Diagrama del analizador sintáctico aritmético:

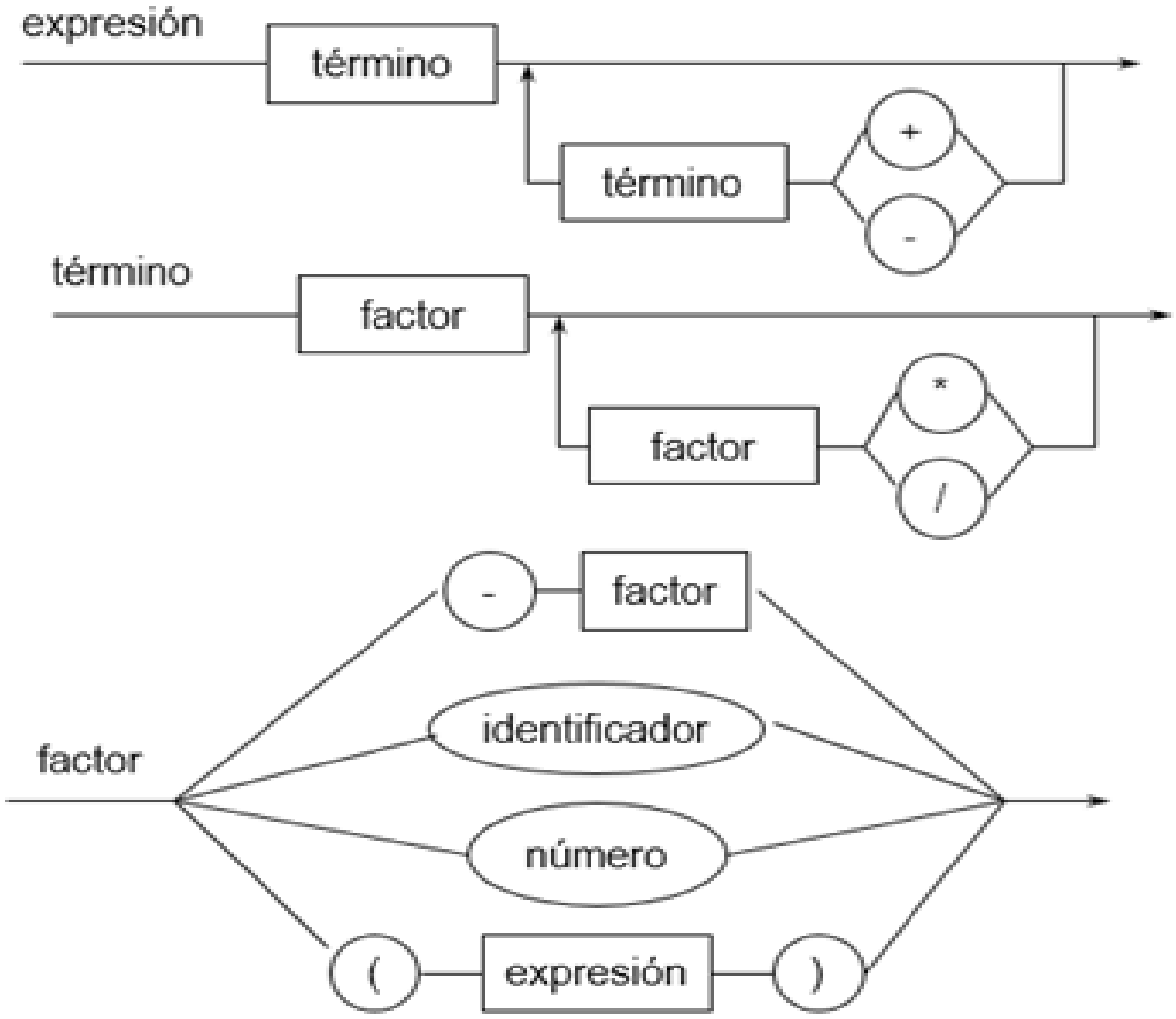
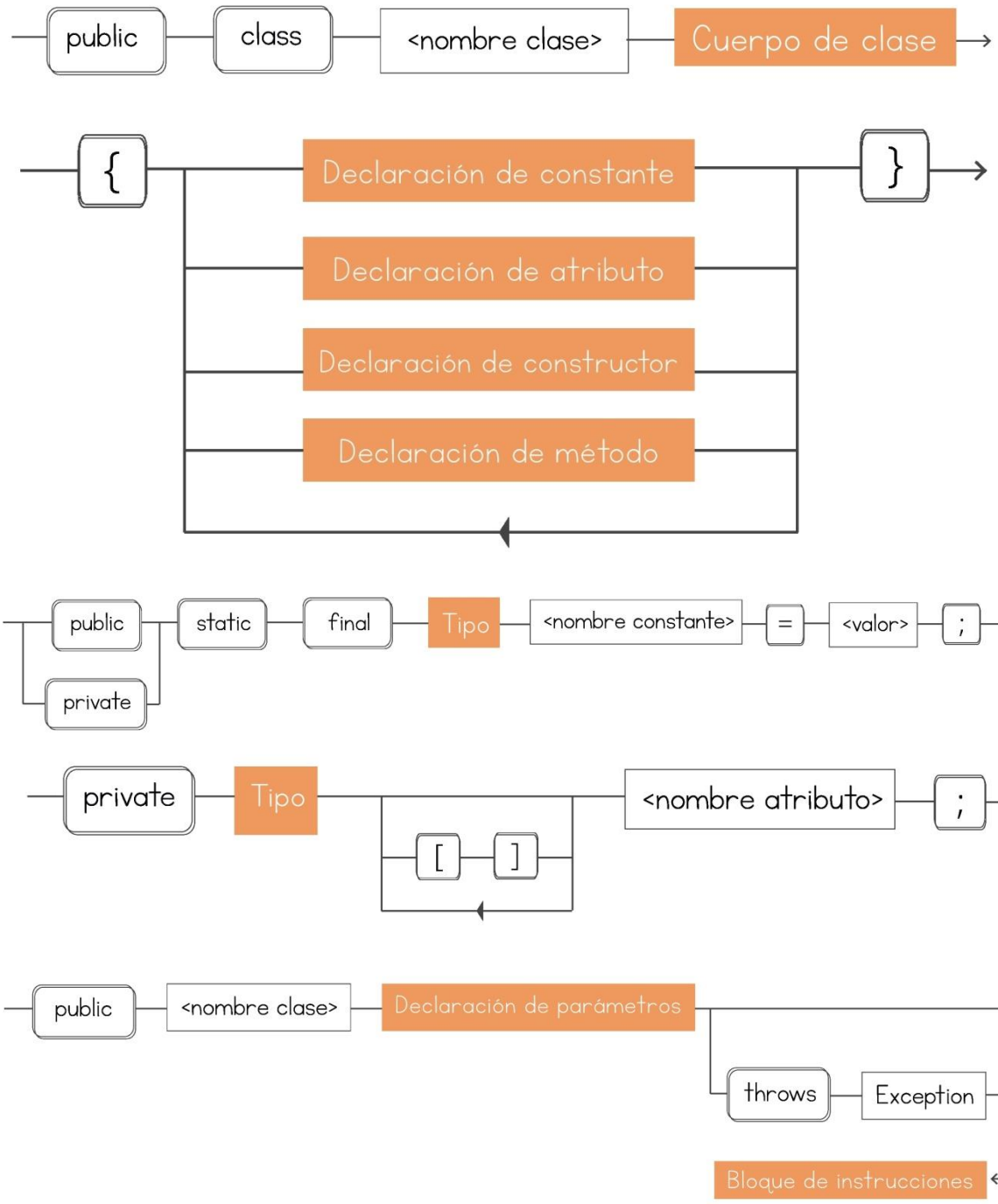
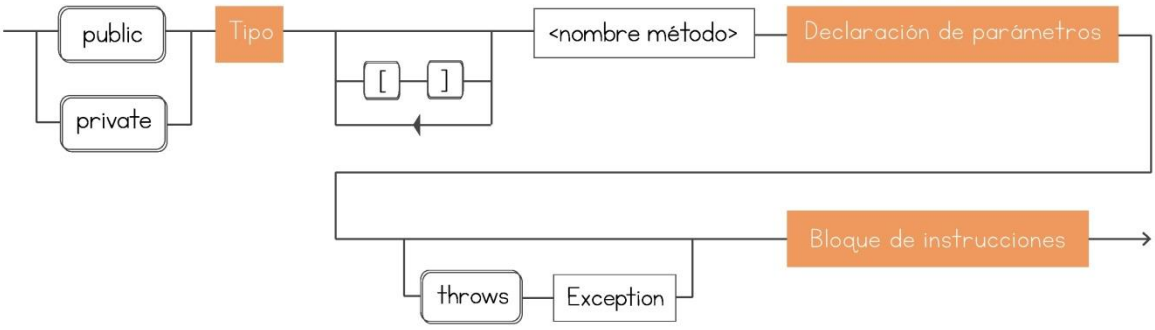


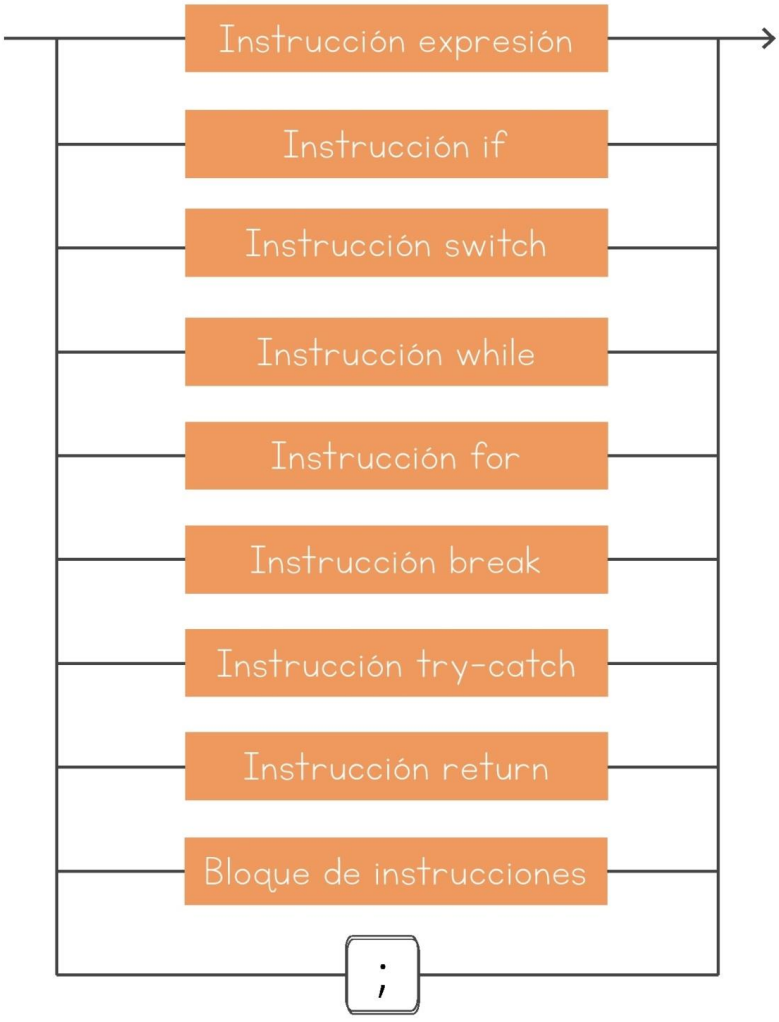
Diagrama del analizador sintáctico de Java

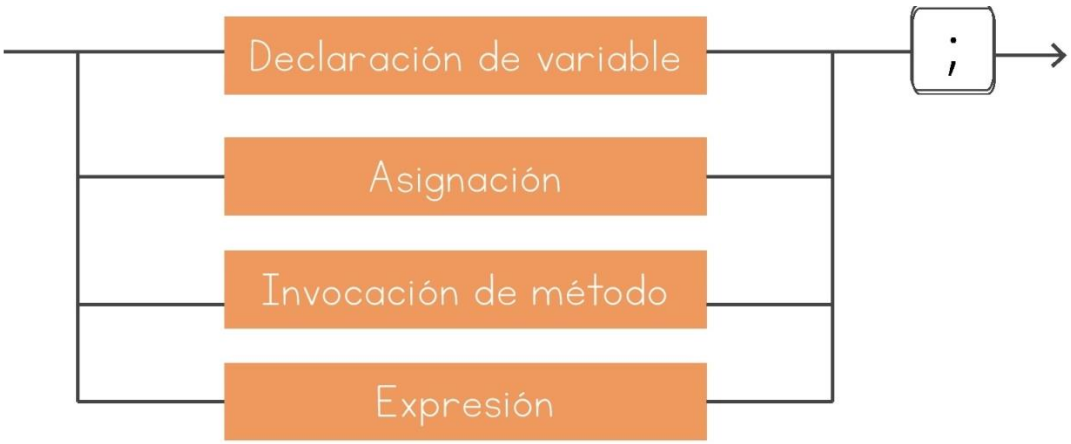
El siguiente diagrama de sintaxis es el que manejan los códigos escritos en Java en general:

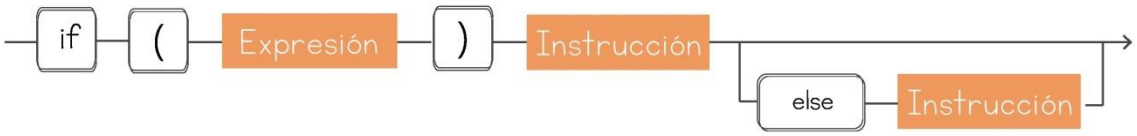


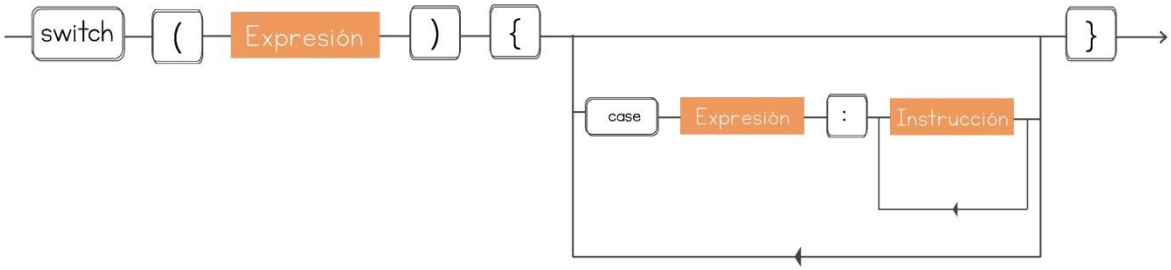

public
private
Tipo
[
]
<nombre método>
Declaración de parámetros
Bloque de instrucciones
throws
Exception


(
)
Declaración de variable
,


Instrucción expresión
Instrucción if
Instrucción switch
Instrucción while
Instrucción for
Instrucción break
Instrucción try-catch
Instrucción return
Bloque de instrucciones
;

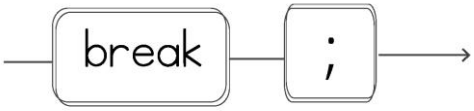

Declaración de variable
Asignación
Invocación de método
Expresión
;

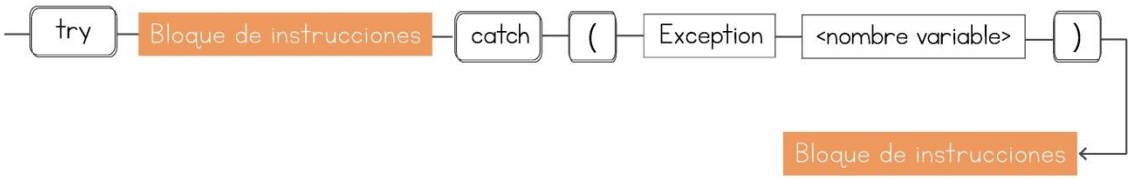

if
(
Expresión
)
Instrucción
else
Instrucción

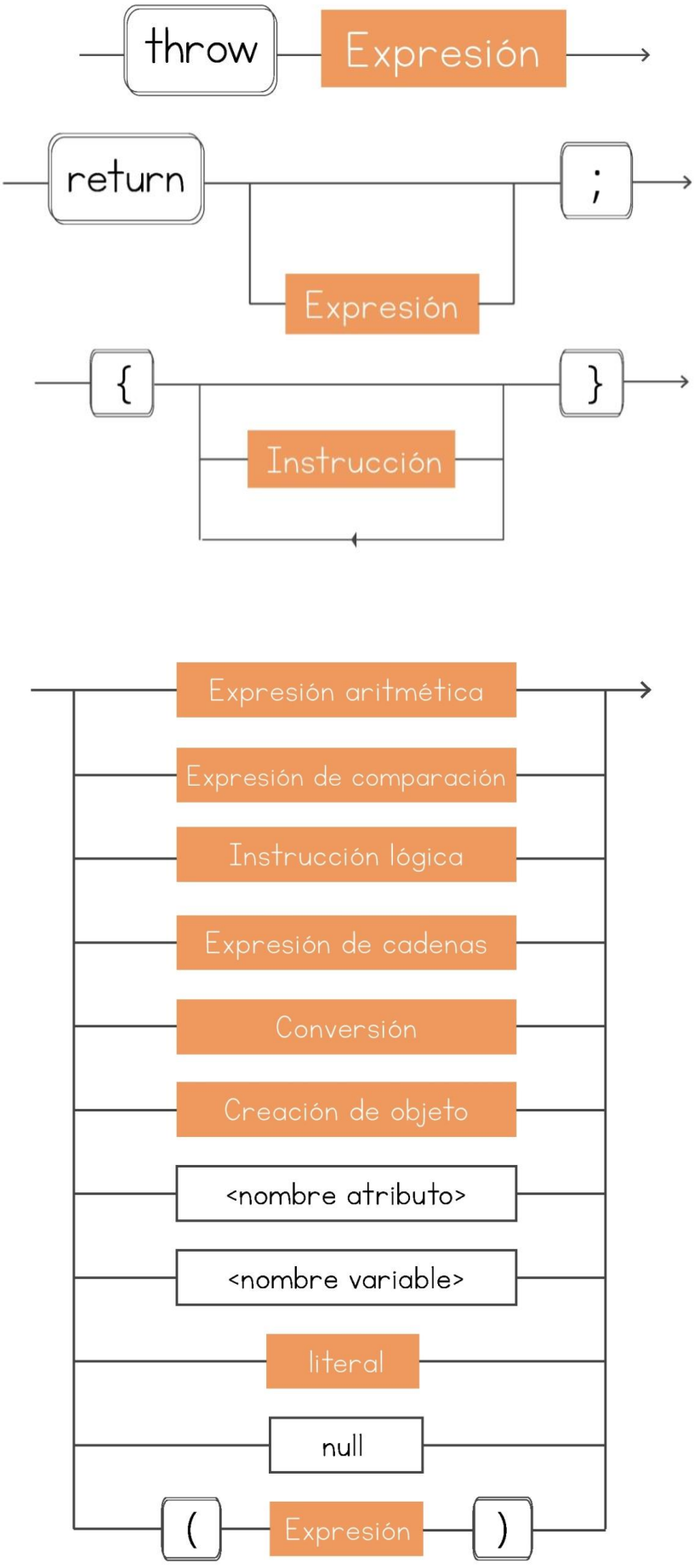

switch
(
Expresión
)
{
}
case
Expresión
:
Instrucción

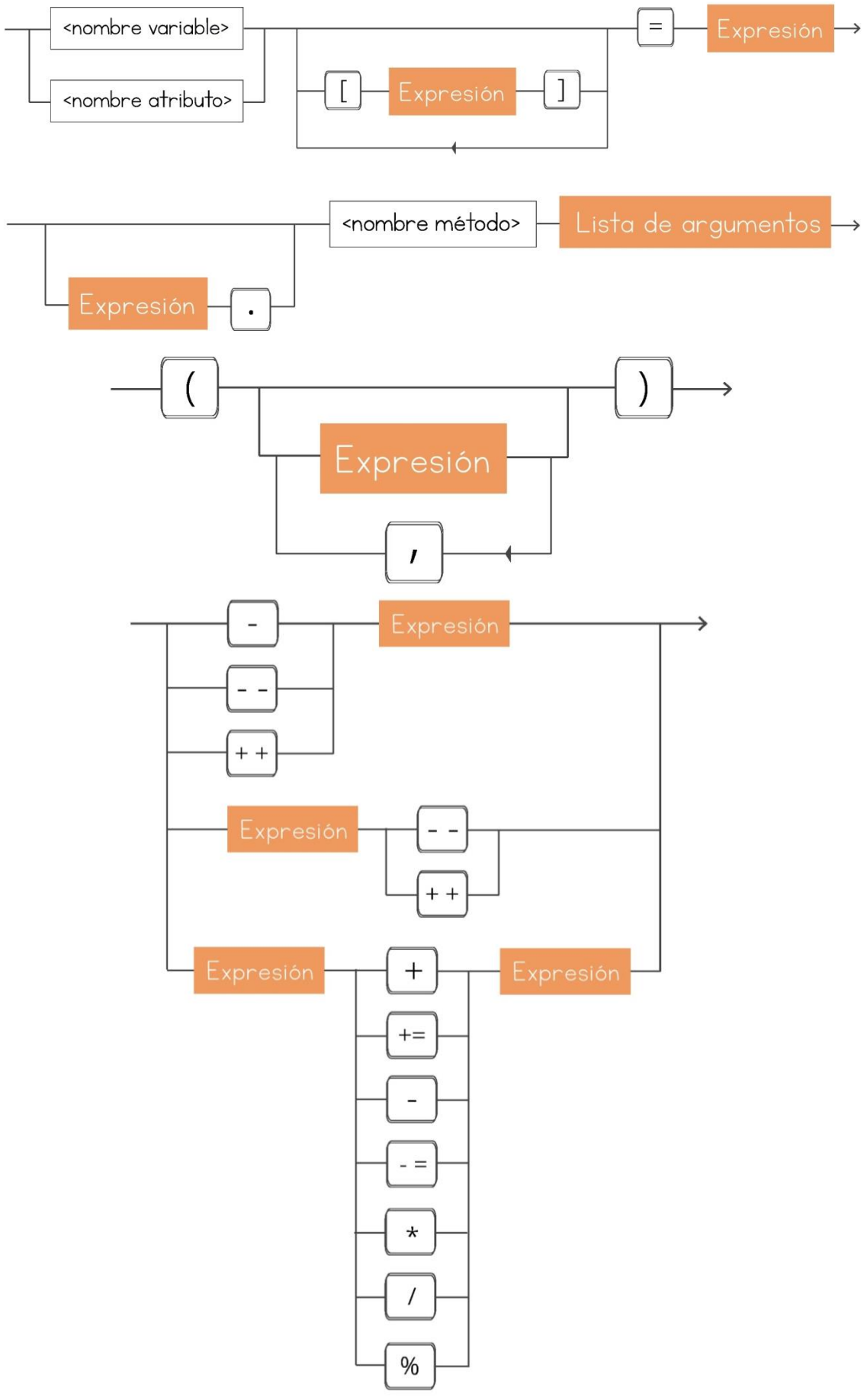

while
(
Expresión
)
Instrucción

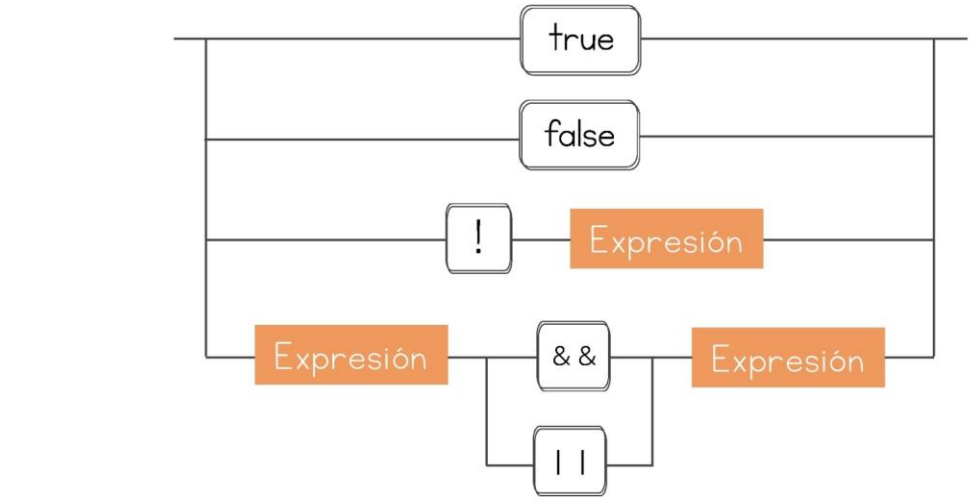

for
(
Expresión
;
Expresión
;
Expresión
)
Instrucción

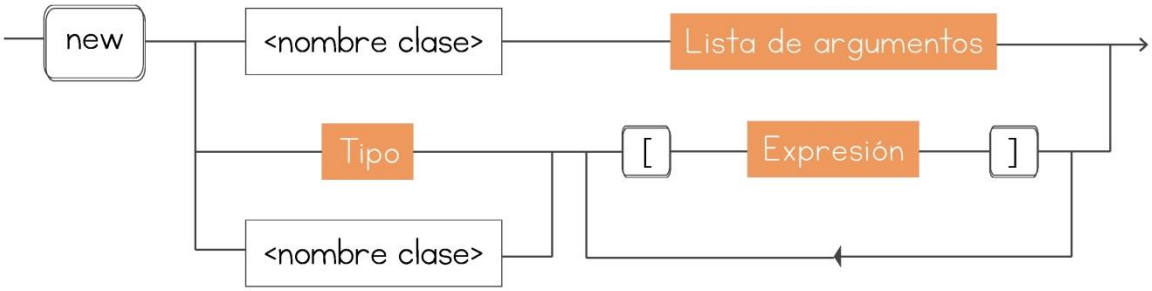

break
;

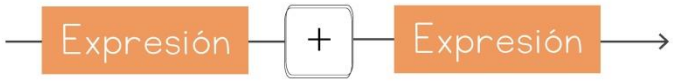

try
Bloque de instrucciones
catch
(
Exception
<nombre variable>
)
Bloque de instrucciones

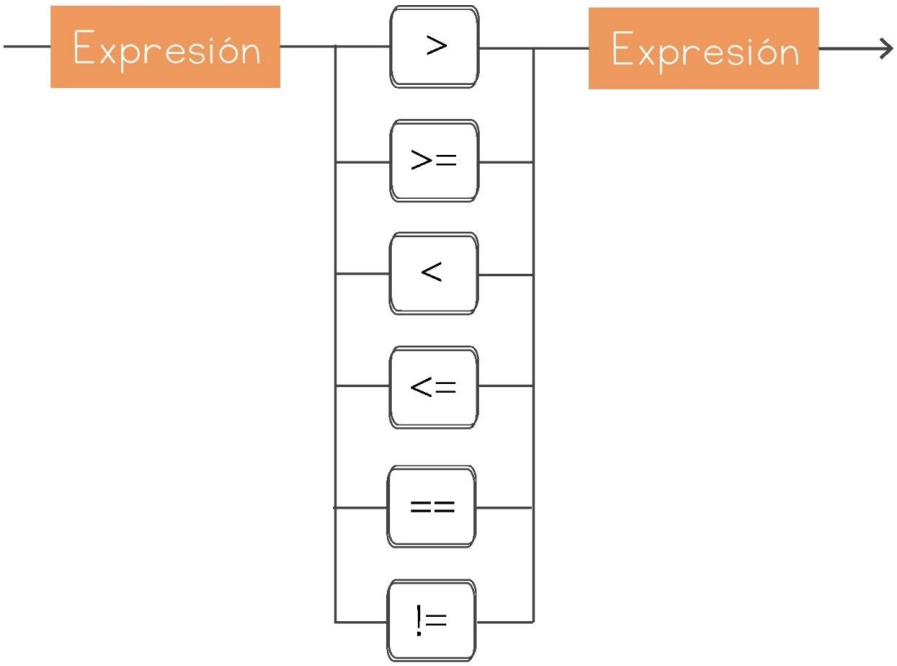

throw
Expresión
return
;
Expresión
{
}
Instrucción
Expresión aritmética
Expresión de comparación
Instrucción lógica
Expresión de cadenas
Conversión
Creación de objeto
<nombre atributo>
<nombre variable>
literal
null
(
Expresión
)

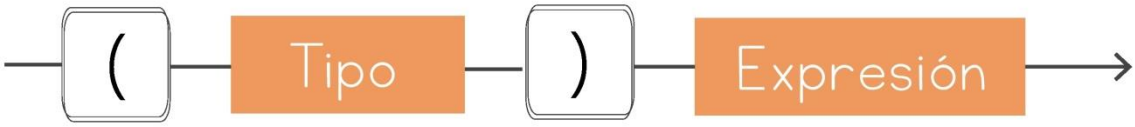

<nombre variable>
<nombre atributo>
=
Expresión
[
Expresión
]
<nombre método>
Lista de argumentos
Expresión
.
(
)
Expresión
,
-
Expresión
- -
+ +
Expresión
- -
+ +
Expresión
+
Expresión
+=
-
- =
*
/
%

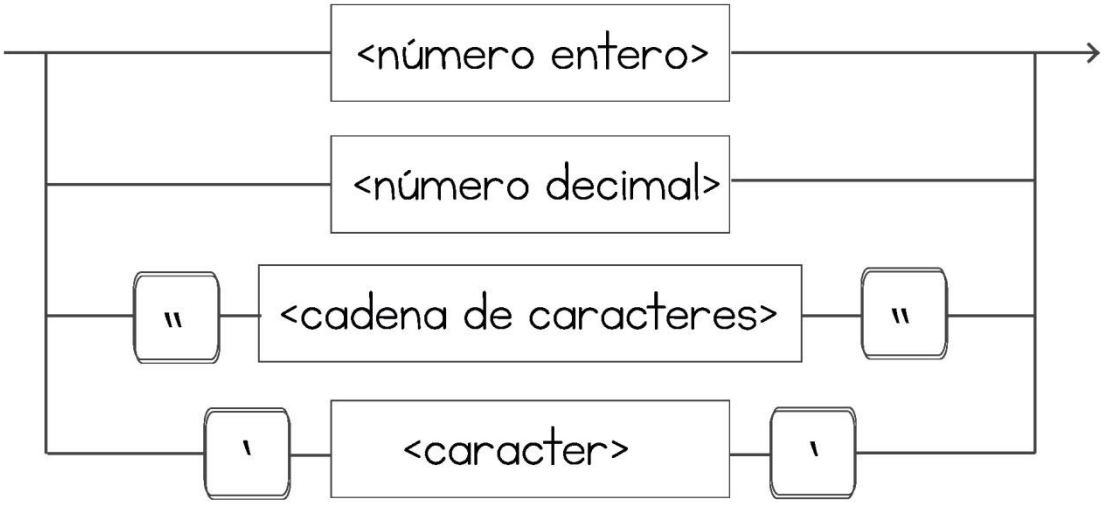

true
false
!
Expresión
Expresión
&&
||
Expresión

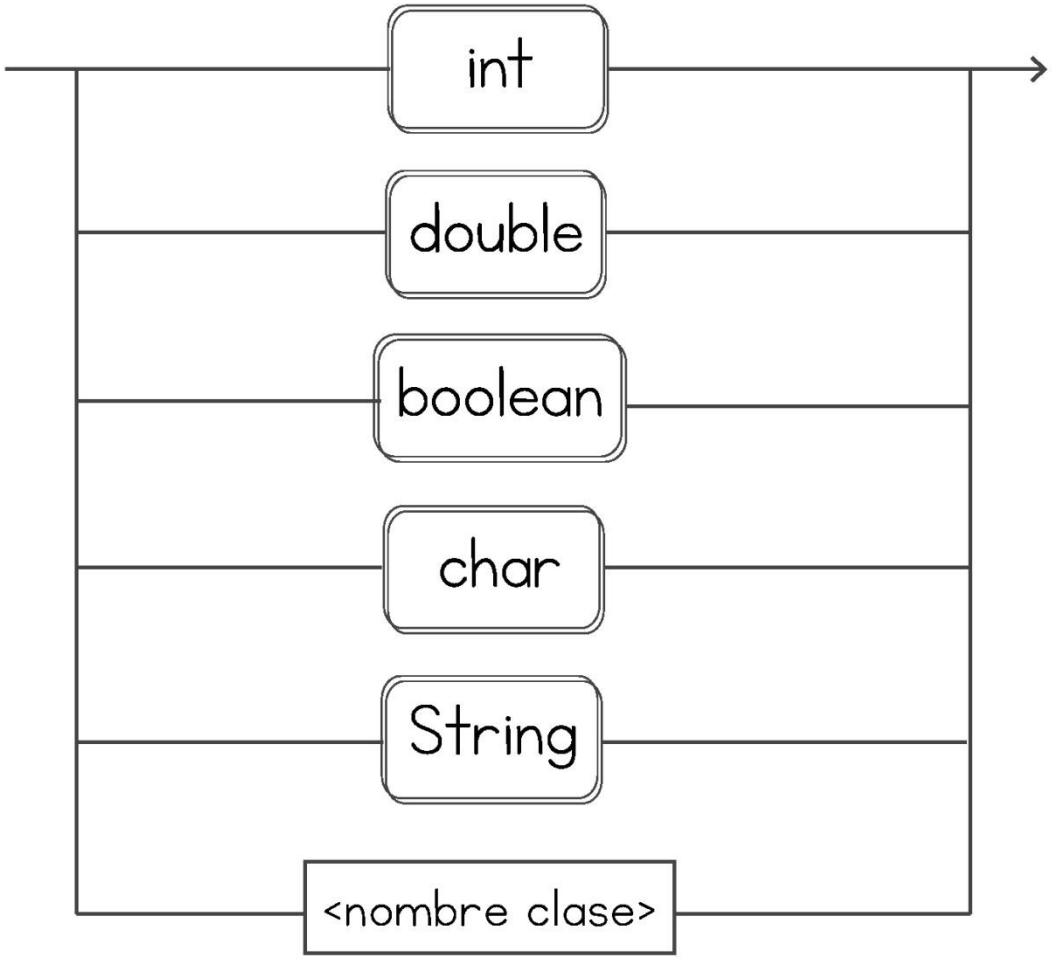

new
<nombre clase>
Lista de argumentos
Tipo
[
Expresión
]
<nombre clase>


Expresión
+
Expresión


Expresión
>
>=
<
<=
==
!=
Expresión


(
Tipo
)
Expresión


<número entero>
<número decimal>
"
<cadena de caracteres>
"
'
<caracter>
'


int
double
boolean
char
String
<nombre clase>

Conclusiones:

Conclusión de Jesús Silva:

Se pudo comprender la segunda etapa de los compiladores que desde mi punto de vista es de las más importantes ya que se encuentra en medio del análisis léxico y el análisis semántico por lo que, para poder avanzar a esta etapa se debe tener dominado el tema de análisis léxico porque incluso en el desarrollo de los analizadores sintácticos tiene mucha importancia ya que son la base para identificar si los componentes de un código fuente son correctos además de indicar por qué no son correctos o qué es lo que le falta para serlo.

Conclusión de Mayra Castillo:

Como conclusión personal comprendí la estructura y sintaxis de las 3 herramientas para la simulación de la etapa del análisis sintáctico, pude aprender las diferencias más notables de cada una y cómo funcionan aunque su función es muy parecida entre ellas.

Conclusión de Patricia Castellanos:

Esta práctica fue cuestión de mucha investigación, para saber cómo desarrollar y utilizar un analizador sintáctico, con algunas respectivas herramientas para su uso, en lo personal trabajamos en equipo y logramos desarrollar el analizador y sus funciones con los tokens y validación de errores entre otras cosas.

Bibliografía:

https://prezi.com/_7e_7aqglspx/java-cc/

http://commpi.blogspot.com/2010/11/herramientas-para-generar-compiladores.html

http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=3D314FD656B6009BD0C9CD66298E966B?doi=10.1.1.611.5399&rep=rep1&type=pdf

http://lenguajesyautomatas1unidad6-4.blogspot.com/2016/06/diagramas-de-sintaxis.html

https://universidad-de-los-andes.gitbooks.io/fundamentos-de-programacion/content/Anexos/A_ElLenguajeJava.html

https://cs.lmu.edu/~ray/notes/javacc/

https://www.javaworld.com/article/2076269/build-your-own-languages-with-javacc.html

http://www.lcc.uma.es/~galvez/ftp/tci/tictema2.pdf

http://cursocompiladoresuaeh.blogspot.com/2010/11/unidad-iii-analizador-sintactico.html