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

Iván Illán Barraya Javier Monescillo Buitrón

21 de marzo de 2019

# Índice

# 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 el 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
  - Creación del EBNF del lenguaje
  - Construcción de la gramática con Cup
- Diseño del Analizador Semántico

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 que será desarrollada en futuras entregas se tratará el análisis sintáctico y el análisis semántico.

# 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.

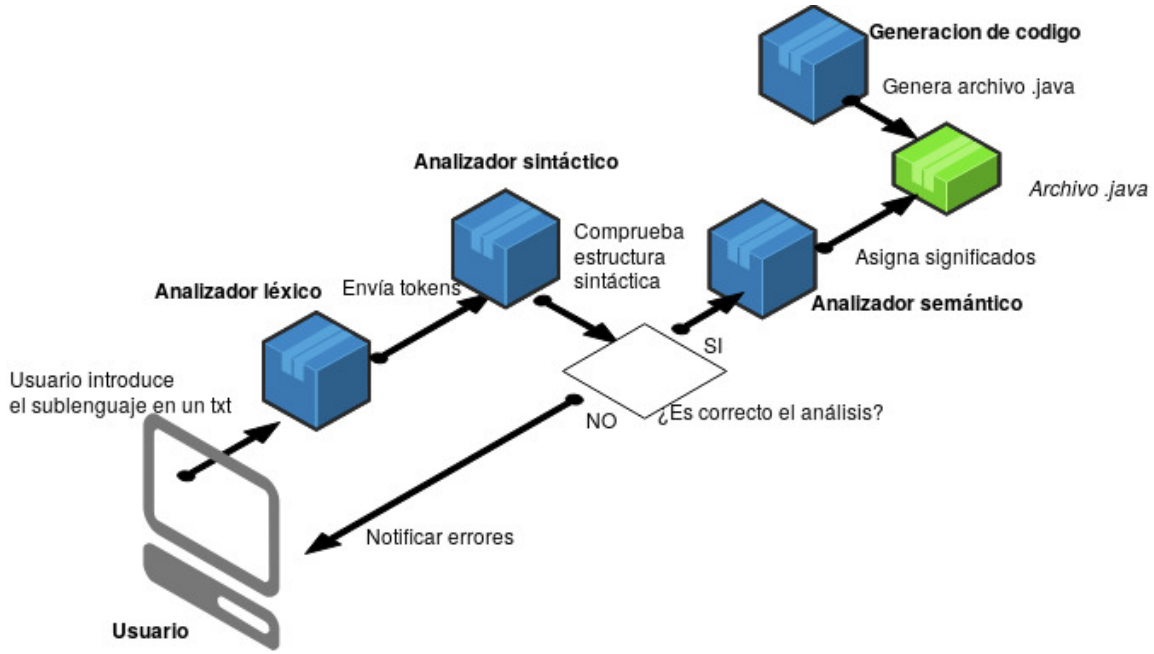


Diagrama general

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 o boolean |
| Instrucción de asignación | type_var a = b; \| type_var a = 0; \| c = b; |
| Decremento | var–; |
| Incremento | var++; |
| Bucles | for( int i = 0; i<hola.length; i++) /* Ops*/ |
| Llamadas a métodos | hola = calcularCosas(); \| metodoVoid(); |
| Retorno de valores | return x; |
| Secuencia de instrucciones | a = b;<br>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 | = | = |
| Negación | ! | ! |
| Lógicos binarios | && | &·& \| \|·\| |
| Incrementos | ++ | +·+ \| -·- |
| 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]* |
| Valor nulo | null | n·u·l·l |

# 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

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

%%

%class AnalizadorLexico
%unicode
%cup
%cupdebug

%line
%column

//Declaraciones

%{

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"_""-"]*
LOGICOS_BINARIOS = "&&" | "||";
ARIT = "*" | "/" | "+" | "-";
RELACIONALES = "<" | "<=" | ">" | ">=" | "==" | "!=";
ASIGNACION = "="
INCREMENT = "++" | "--";
NUM = 0 | [1-9][0-9]*


%%
"null" {return symbol(sym.NULL, new String(yytext()));}
"return" {return symbol(sym.RETURN, 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.BEGIN_METODOS, new String(yytext()));}
"true" {return symbol(sym.TRUE, new String(yytext()));}
"false" {return symbol(sym.FALSE, new String(yytext()));}
{ARIT} {return symbol(sym.ARIT, 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()));}
{INCREMENT} {return symbol(sym.INCREMENT, new String(yytext()));}
{LOGICOS_BINARIOS} {return symbol(sym.LOGICOS_B, new String(yytext()));}
"!" {return symbol(sym.LOGICOS_U, new String(yytext()));}
"{" {return symbol(sym.LL_OP, new String(yytext()));}
"}" {return symbol(sym.LL_CL, new String(yytext()));}
"(" {return symbol(sym.PAR_OP, new String(yytext()));}
")" {return symbol(sym.PAR_CL, new String(yytext()));}
"," {return symbol(sym.COMA, new String(yytext()));}
"." {return symbol(sym.PUNTO, new String(yytext()));}
{ID} {return symbol(sym.ID, new String(yytext()));}
{COMENTARIO} {/*Ignoramos comentarios*/}
{ESPACIOBLANCO} {}
{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 y Semántico en CUP

```
package teoria_automatas;

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) {
report_error(message, info);
System.exit(1);
}

public void syntax_error(Symbol s){
System.out.println("Error recuperable de sintaxis: "+s.value+" Linea "+(s.left+1)+"
    columna "+(s.right+1) );
}

public void unrecovered_syntax_error(Symbol s) throws java.lang.Exception{
System.out.println("Error no recuperable de sintaxis: "+s.value+" Linea
    "+(s.left+1)+" columna "+(s.right+1) );
}

:}

action code {:

:}

non terminal programa, dec_metodos, metodos, dec_args, cuerpo_metodo, dec_for,
    dec_ops, type_boolean, for_st_aux, extra;
non terminal dec_asigs_ops, type_methods, type_vars,dec_for_loop, dec_asigs,
    dec_asigs_aux, dec_call,for_st_arrays, dec_exp;
non terminal dec_return, cuerpo_for, for_st_1, for_st_2, for_st_3, init_for,
    dec_call_args, dec_asignaciones, dec_call_ops, type_ops;

terminal String LL_OP, LL_CL, FOR ,ID, RETURN, PUNTOCOMA, NUM, PUNTO, NULL, OR,
    DISTINTO;
terminal String RELACIONALES, ASIGNACION, INCREMENT, PAR_OP, PAR_CL, COMA, POR,
    DIV, MENOS, MAS;
terminal String BEGIN_METODOS, LOGICOS_B, LOGICOS_U, BOOLEAN, TRUE, FALSE, INT,
    VOID;

precedence left MENOS;
precedence left MAS;
```

```
precedence left POR;
precedence right DIV;
precedence right RELACIONALES;
precedence right DISTINTO;
precedence right LOGICOS_B;
precedence right OR;
precedence right LOGICOS_U;


programa ::= dec_metodos ;


dec_metodos ::= metodos | metodos dec_metodos;

metodos ::= BEGIN_METODOS type_methods ID PAR_OP dec_args PAR_CL LL_OP
    cuerpo_metodo LL_CL;



dec_args ::= type_vars ID | type_vars ID COMA dec_args | ;


cuerpo_metodo ::= dec_for_loop dec_call_ops dec_return;

dec_for_loop ::= dec_for | ;


dec_for ::= FOR PAR_OP init_for PAR_CL LL_OP cuerpo_for dec_for LL_CL
| FOR PAR_OP init_for PAR_CL LL_OP cuerpo_for LL_CL ;


init_for ::= for_st_1 PUNTOCOMA for_st_2 PUNTOCOMA for_st_3 ;

for_st_1 ::= type_vars ID ASIGNACION for_st_aux | ID ASIGNACION for_st_aux ;

for_st_aux ::= ID | NUM;

for_st_2 ::= ID RELACIONALES for_st_arrays;

for_st_arrays ::= ID PUNTO ID | ID;

for_st_3 ::= ID INCREMENT;


cuerpo_for ::= type_ops;


dec_asigs_ops ::= dec_asigs dec_asigs_ops | ;


dec_asigs ::= ;

dec_asignaciones ::= ;
```

```
dec_call_ops ::= dec_call dec_call_ops | ;

dec_call ::= ID ASIGNACION ID PAR_OP dec_call_args PAR_CL PUNTOCOMA
| PAR_OP dec_call_args PAR_CL PUNTOCOMA
;


dec_call_args ::= ID | ID COMA dec_call_args | ;

dec_return ::= RETURN dec_ops | ;



dec_ops ::= dec_exp PUNTOCOMA ;

type_ops ::= dec_exp PUNTOCOMA | ;

dec_exp ::= NUM
| ID
| type_boolean
| LOGICOS_U dec_exp
| dec_exp MAS dec_exp
| dec_exp POR dec_exp
| dec_exp DIV dec_exp
| dec_exp MENOS dec_exp
| dec_exp RELACIONALES dec_exp
| dec_exp LOGICOS_B dec_exp
| dec_exp DISTINTO dec_exp
| dec_exp OR dec_exp
| PAR_OP dec_exp PAR_CL;



type_methods ::= INT | BOOLEAN | VOID;

type_vars ::= INT | BOOLEAN ;

dec_asigs_aux ::= ID | NUM | type_boolean;

type_boolean ::= TRUE | FALSE;
```

## 6.3. Control de errores sintácticos

Para controlar los errores sintácticos en CUP se han utilizado producciones de error. Las producciones de error utilizan un símbolo terminal error que pertenece a la clase *Symbol* propia de CUP.

De tal manera que cuando se produce una reducción al mismo, se invoca a una rutina de error asociada a este símbolo. En esta rutina de error, se invoca al método report_error de la clase *Parser*.

# 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