UNIVERSIDAD NACIONAL DEL ALTIPLANO

FACULTAD DE INGENIERIA MECANICA ELECTRICA, ELECTRONICA Y SISTEMAS

ESCUELA PROFESIONAL DE INGENIERIA DE SISTEMAS

COMPILADORES

INFORME

ANALISADOR LÉXICO

PRESENTADO POR:

CRISTIAN RONY YAGUNO MAMANI

DOCENTE:

PUNO – PERÚ

2018-I

ÍNDICE

Introducción

Dentro de este trabajo se construye una aplicación java usando el ambiente Netbeans 8.1, que efectúe un análisis léxico sobre una entrada ingresada en un componente de texto. Un objeto denominado o AnaLex perteneciente a una clase Lexico, será el encargado de realizar la tarea de analizar lexicamente al texto de entrada. La clase Lexico es generada usando el software SP-PS1, el cual genera también a la clase Automata que está anidada dentro de la clase Lexico.

En el presente trabajo abordaremos sobre la construcción de un programa para realizar el analisis léxico, para ello empezaremos por definir el lenguaje, sus componentes del lenguaje y sus reglas de sintaxis, además del programa, sus funciones y su interfaz gráfica. Este analizador léxico lee caracteres del archivo de entrada, donde se encuentra la cadena a analizar, reconoce lexemas en un cuadro llamado secuencia ,retornando en otro cuadro los tokens , lexemas, la posicion en que fila y columna está ubicado de la tabla de códigos

Este trabajo tiene el objetivo de dar a conocer la lógica y funcionamiento de un analizador léxico , es decir leer el flujo de caracteres de entrada y transformarlo en una secuencia de componentes léxicos que utilizara el analizador sintáctico , además de permitir a nosotros, como alumnos, entender mejor este tema, y de este modo adquirir la idea de la implementación de un programa que ayude en su solución; y dejarlo listo para la siguiente fase de la compilación denominada: Análisis Sintáctico.

1. Analizador Léxico

Según (Gálvez Rojas & Mora Mata, 2005): Se encarga de buscar los componentes léxicos o palabras que componen el programa fuente, según unas reglas o patrones.

Un generador de analizador léxico toma como entrada una especificación con un conjunto de expresiones regulares y acciones correspondientes. Genera un programa (un lexer) que lee la entrada, compara la entrada con las expresiones regulares en el archivo de especificación, y ejecuta la acción correspondiente si una expresión regular coincide. Los Lexers suelen ser el primer paso de front-end en los compiladores, haciendo coincidir palabras clave, comentarios, operadores, etc., y generando un flujo de token de entrada para los analizadores. Lexers también se puede utilizar para muchos otros fines.

a) Simplificación del Diseño

Un diseño sencillo es quizás la ventaja más importante. Separar el análisis léxico del análisis sintáctico a menudo permite simplificar una, otra o ambas fases. Normalmente añadir un analizador léxico permite simplificar notablemente el analizador sintáctico. Aún más, la simplificación obtenida se hace especialmente patente cuando es necesario realizar modificaciones o extensiones al lenguaje inicialmente ideado; en otras palabras, se facilita el mantenimiento del compilador a medida que el lenguaje evoluciona.

b) Eficiencia

La división entre análisis léxico y sintáctico también mejora la eficiencia del compilador. Un analizador léxico independiente permite construir un procesador especializado y potencialmente más eficiente para las funciones. Gran parte del tiempo de compilación se invierte en leer el programa fuente y dividirlo en componentes léxicos. Con técnicas especializadas de manejo de buffers para la lectura de caracteres de entrada y procesamiento de patrones se puede mejorar significativamente el rendimiento de un compilador.

c) Portabilidad

Se mejora la portabilidad del compilador, ya que las peculiaridades del alfabeto de partida, del juego de caracteres base y otras anomalías propias de los dispositivos de entrada pueden limitarse al analizador léxico. La representación de símbolos especiales o no estándares, como ‘8’ en Pascal, se pueden aislar en el analizador léxico.

d) Patrones Complejos

Otra razón por la que se separan los dos análisis es para que el analizador léxico se centre en el reconocimiento de componentes básicos complejos. Por ejemplo en Fortran, existe el siguiente par de proposiciones muy similares sintácticamente, pero de significado bien distinto:

DO5I = 2.5 (Asignación del valor 2.5 a la variable DO5I)

DO 5 I = 2, 5 (Bucle que se repite para I = 2, 3, 4 y 5)

1.1. Token, Patrón y Lexema

1.1.1. Patrón: es una expresión regular.

1.1.2. Token: es la categoría léxica asociada a un patrón. Cada token se convierte en un número o código identificador único. En algunos casos, cada número tiene asociada información adicional necesaria para las fases posteriores de la etapa de análisis. El

concepto de token coincide directamente con el concepto de terminal desde el punto de vista de la gramática utilizada por el analizador sintáctico.

1.1.3. Lexema: Es cada secuencia de caracteres concreta que encaja con un patrón. Por ejemplo: “8", “23" y “50" son algunos lexemas que encajan con el patrón (‘0'|’1'|’2'| ... |’9’) +. El número de lexemas que puede encajar con un patrón puede ser finito o infinito, p.ej. en el patrón ‘W’ ‘H’‘I’‘L’‘E’ sólo encaja el lexema “WHILE”.

1.2. Componentes Léxicos

En general, no basta con saber la categoría a la que pertenece un componente, en muchos casos es necesaria cierta información adicional. Por ejemplo, sería necesario conocer el valor de un entero o el nombre del identificador. Utilizamos los atributos de los componentes para guardar esta información.

Un último concepto que nos será útil es el de lexema: la secuencia concreta de caracteres que corresponde a un componente léxico.

Por ejemplo, en la sentencia altura=2; hay cuatro componentes léxicos, cada uno de ellos de una categoría léxica distinta:

Categoría léxica	Lexema	Atributos
identificador	`altura`	—
asignación	=	—
entero	2	valor: 2
terminador	;	—

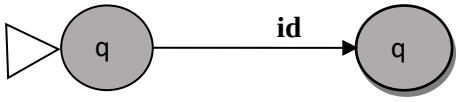
El lenguaje del programa analizador léxico tiene los siguientes tipos de componentes léxicos o tokens:

a) Identificadores, que sólo son nombres de variables y están compuestos por una única letra minúscula de rango: a. . . z.
b) Constantes numéricas de un sólo dígito, de rango: 0. . . 9.
c) Operadores: +, -, *, / y %.
d) Símbolo de asignación: = (igual).
e) Paréntesis: “(”y “)”.
f) Separador de sentencias: “;” (punto y coma).
g) Indicadores de principio y fin de bloque: “{” y “}”.
h) Palabras reservadas, están formadas por una letra mayúscula. Tan sólo son tres: R (lectura), W (escritura) y M (programa principal).

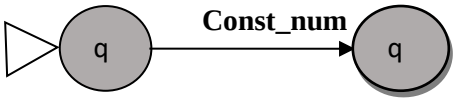
1.3. Diagrama de Estado de los Componentes Léxicos:

Realizaremos a cabo el diagrama de estados de cada componente léxico:

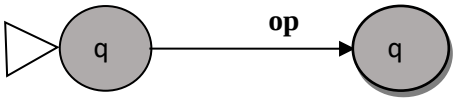
- Para el token identificadores: <id>:
  <id> ::= **a**|**b**|**c**…|**z**



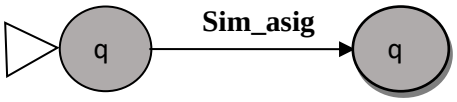
- Para el token constantes numéricas: <const_num>:
  <const_num> ::= **0**|**1**|**2**…|**9**



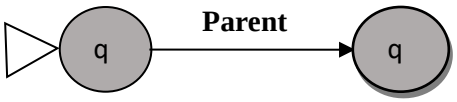
- Para el token operadores: <op>:
  <op> ::= **+** | **-** | ***** | **/** | **%**



- Para el token símbolo de asignación: <simb_asig>:
  <simb_asig> ::= **=**

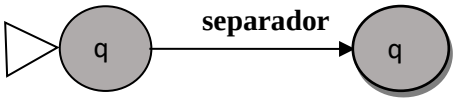


- Para el token paréntesis: <parent>:
  <parent> ::= **(** | **)**



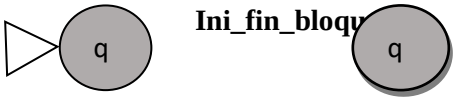
- Para el token separador de sentencias: <separador>:

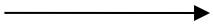
  <separador> ::= **;**



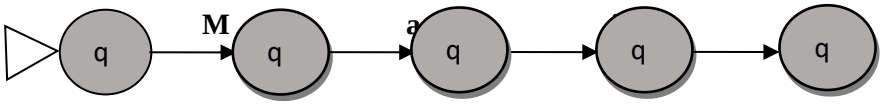
- Para el token indicadores de principio y fin de bloque: <IFB>

  <IFB> ::= **{** | **}**

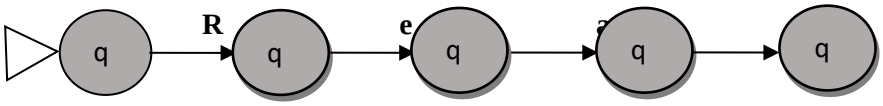




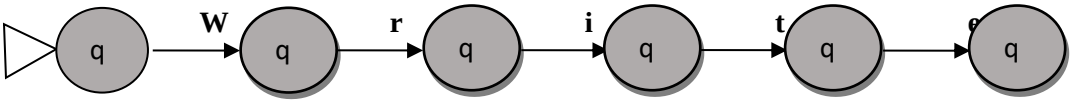
- Para el token de la palabra reservada: programa principal: <M>
  <Main> ::= **Main**



- Para el token de la palabra reservada: lectura: <R>
  <Read> ::= **Read**



- Para el token de la palabra reservada: escritura: <W>
  <Write> ::= **Write**



1.4. Categorías Léxicas

Algunas familias de categorías léxicas típicas de los lenguajes de programación son:

a) **Palabras clave:** Palabras con un significado concreto en el lenguaje. Ejemplos de palabras clave en C son while, if, return. . . Cada palabra clave suele corresponder a una categoría léxica. Habitualmente, las palabras clave son reservadas. Si no lo son, el analizador léxico necesitara información del sintáctico para resolver la ambigüedad.

b) **Identificadores**: Nombres de variables, nombres de función, nombres de tipos definidos por el usuario, etc. Ejemplos de identificadores en C son i, x10, valor_leido. . .

c) **Operadores:** Símbolos que especifican operaciones aritméticas, lógicas, de cadena, etc. Ejemplos de operadores en C son +, *, /, %, ==!=, &&...

d) **Constantes numéricas:** Literales que especifican valores numéricos enteros (en base decimal, octal, hexadecimal...), en coma flotante, etc. Ejemplos de constantes numéricas en C son 928, 0xF6A5, 83.3E+2. . .

e) **Constantes de carácter o de cadena Literales**, que especifican caracteres o cadenas de caracteres. Un ejemplo de literal de cadena en C es "una cadena"; ejemplos de literal de carácter son 'x', '\0'...

f) **Símbolos especiales** Separadores, delimitadores, terminadores, etc. Ejemplos de estos símbolos en C son {, }, ;. . . Suelen pertenecer cada uno a una categoría léxica separada.

2. JFlex

JFlex es un generador de analizador léxico (también conocido como generador de escáner) para Java, escrito en Java.

Los lectores de JFlex se basan en autómatas finitos deterministas (DFA). Son rápidos, sin costosos retrocesos.

JFlex está diseñado para trabajar junto con el generador de analizadores LALR CUP de Scott Hudson, y la modificación Java de Berkeley Yacc BYacc / J de Bob Jamison. También se puede usar junto con otros generadores de analizadores sintácticos como ANTLR o como una herramienta independiente (Klein, Rowe, & Décamps, 2015).

3. Estructura de un Archivo Jflex

3.1. Opciones y Declaraciones

En esta sección van los paquetes que se van a utilizar, aquí se declaran las directivas y macros, se indica el nombre de la clase, en este caso "%class AnLex", se habilita line y column para obtener la línea y columna de la posición actual del compilador y cup para realizar la integración con el archivo cup.

3.2. Código de Usuario

En esta sección van las directivas o especificaciones para obtener la salida deseada. También se muestra aquí los métodos para encontrar los tokens deceados

3.3. Reglas léxicas

En esta sección del archivo JFlex, es donde se define las reglas para obtener los tokens de la cadena que está leyendo

4. Lenguaje de Desarrollo

Código de archivo Jflex

```
//primera parte

package ejemploal; //paquete en donde se creara el analizador

import static ejemploal.Token.*;

%%

//segunda parte

%class AnLex

%type Token

%full

%char

%line//Activar el contador de lineas, variables yyline

%column//Activar el contador de columna, variable yycolumn
```

```
%ignorecase
F= [int main()]
L=[a-zA-Z_]
D=[0-9]
L_D={L}|{D}
COMA_FLOTANTE=[D]"."[D]
E=" "
WHITE=[ \t\r\n]
%{
public String lexeme;
%}

%%
//tercera parte
{WHITE} {/* ignore */}
"//".* {/* ignore */}
","|"@"|"<<"|">>"|"." {return ESPECIAL;}
"pow"|"sqrt"|"cbrt"|"hypot" {return FUNCIONES_POTENCIA;}
"cos"|"sin"|"tan"|"acos"|"asin"|"atan" {return FUNCIONES_TRIGONOMETRICAS;}
"=" {return ASIGNACION;}
"==" {return IGUAL;}
"+" {return SUMA;}
"*" {return PRODUCTO;}
"-" {return RESTA;}
"^" {return POTENCIA;}
"%" {return MODULO;}
";" {return PUNTOYCOMA;}
"while" {return MIENTRAS;}
"(" {return ABRE_PARENTESIS;}
">" {return MAYOR;}
"<" {return MENOR;}
">=" {return MAYOR_IGUAL;}
```

"<=" {return MENOR_IGUAL;}

"true" {return VERDADERO;}

"false" {return FALSO;}

"||" {return OR;}

"&&" {return AND;}

"==" {return IGUAL_IGUAL;}

"!=" {return NO_IGUAL;}

")" {return CIERRA_PARENTESIS;}

"cout<<" {lexeme=yytext();return MENSAJE;}

"{" {lexeme=yytext();return ABRE_LLAVE;}

"}" {lexeme=yytext();return CIERRA_LLAVE;}

"#include <iostream>"|"include <cmath>"|"include <conio>"|"include <cfloat>"|"include <studio.h>"|"include <time>"|"include <string>" {return LIBRERIAS;}

{L}({L}|{E}|{D})* {lexeme=yytext(); return ID;}

[-+]?{D}+ {lexeme=yytext();return INT;}

{F}* {lexeme=yytext();return FUNCION_PRINCIPAL;}

. {return ERROR;}

El lenguaje de desarrollo que utilizaremos es el LENGUAJE java. A continuación, se presenta las líneas de código para la creación de las ventanas:

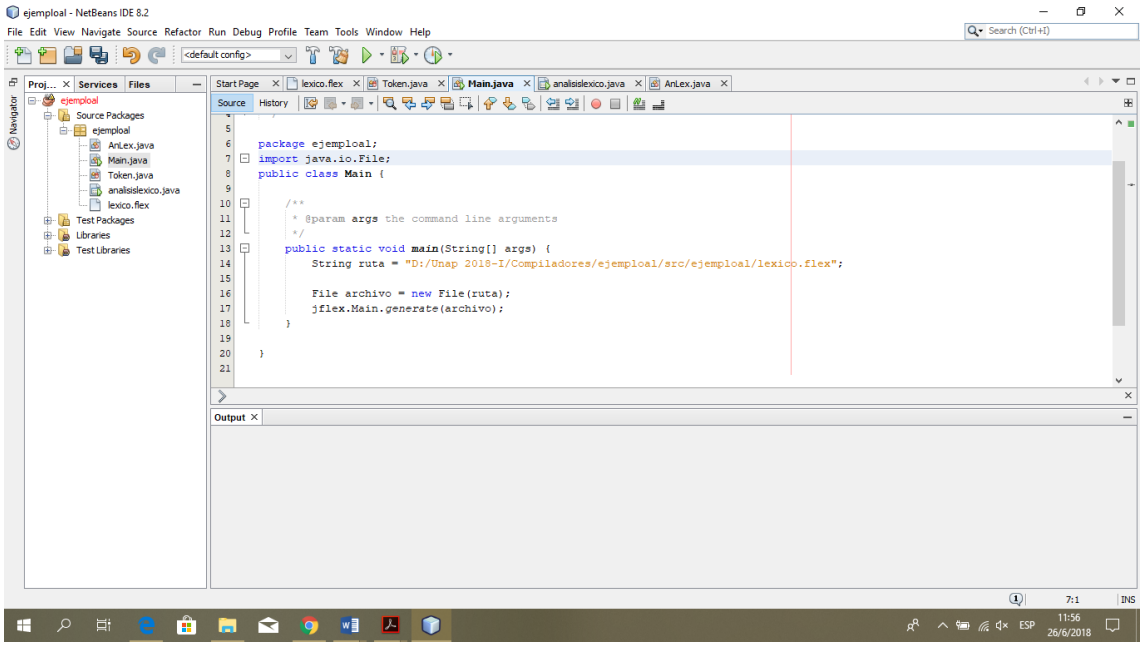


Figura 1: Método main

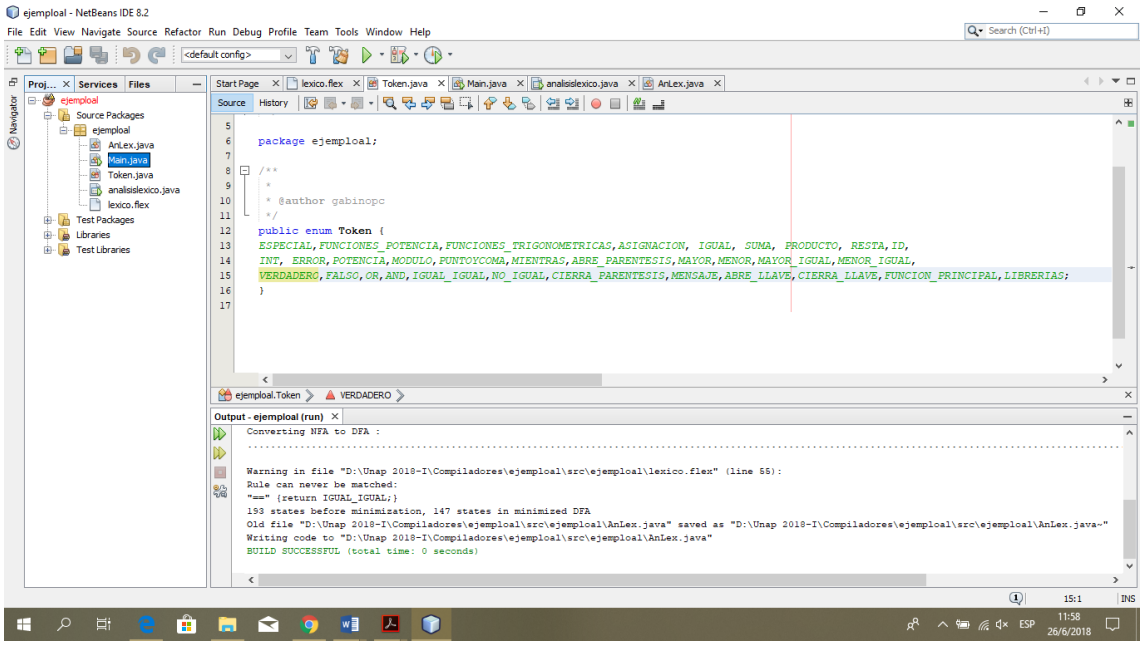


Figura 2: Método token

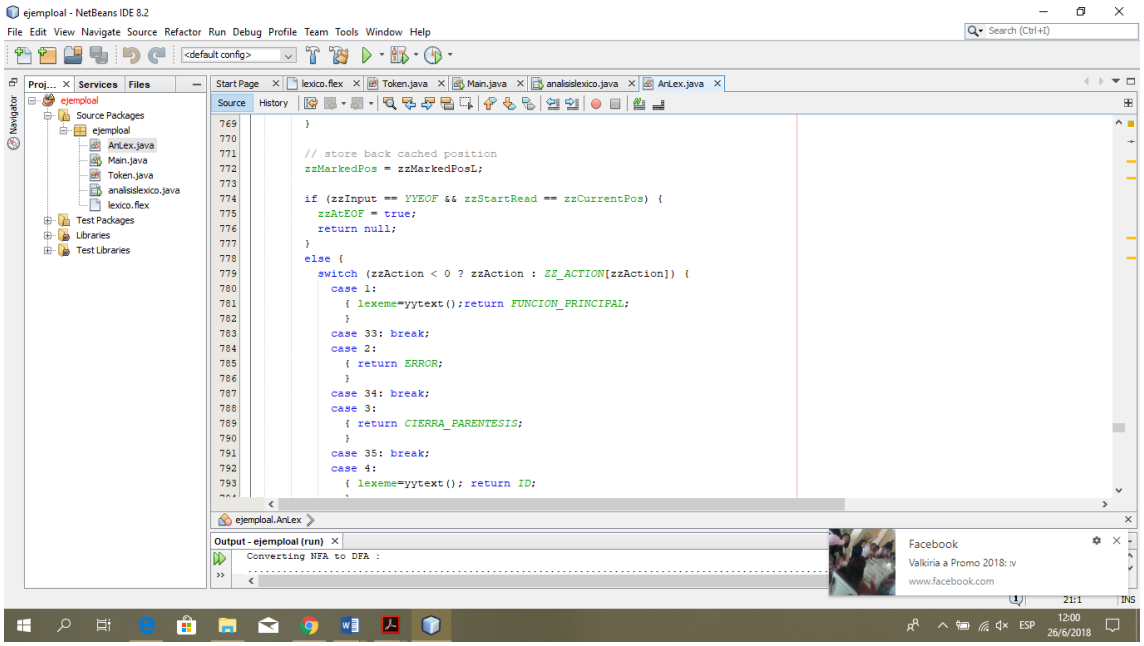


Figura 3: Método Anlex

5. Resultados

Entradas y salidas

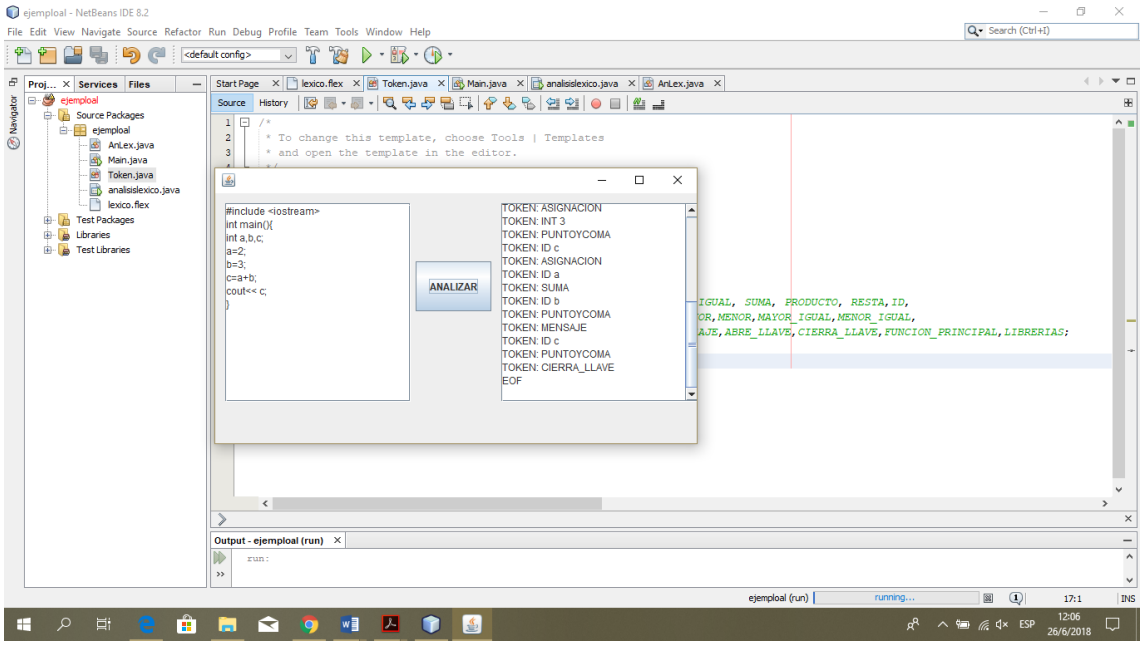


Figura 4: Ejemplo 1

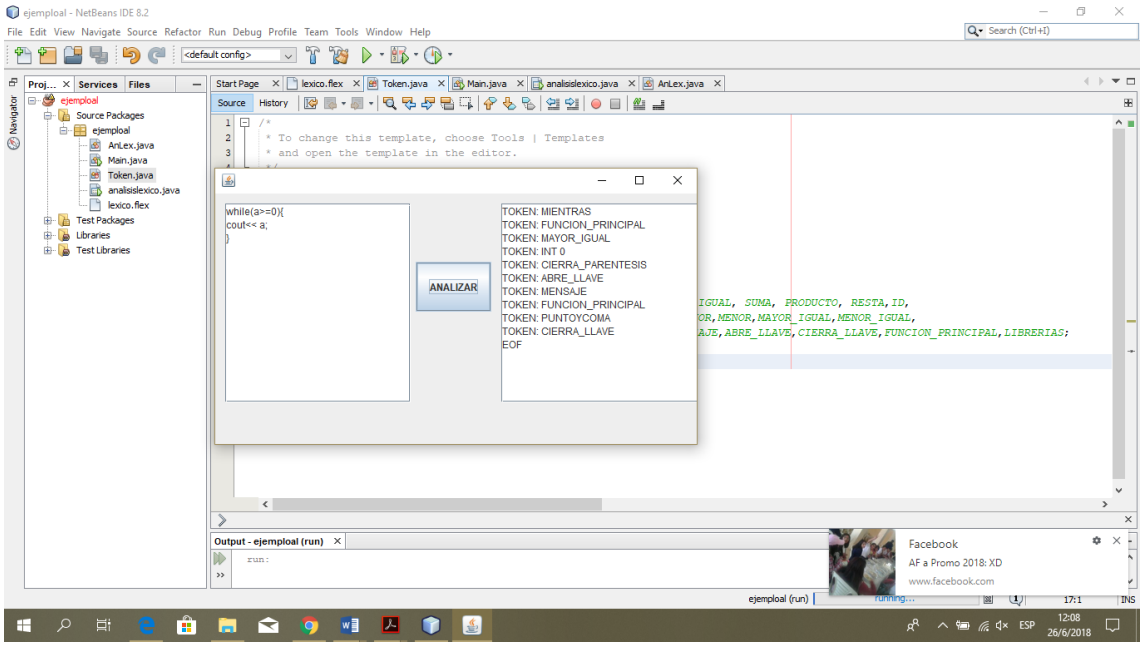


Figura 5: Ejemplo 2

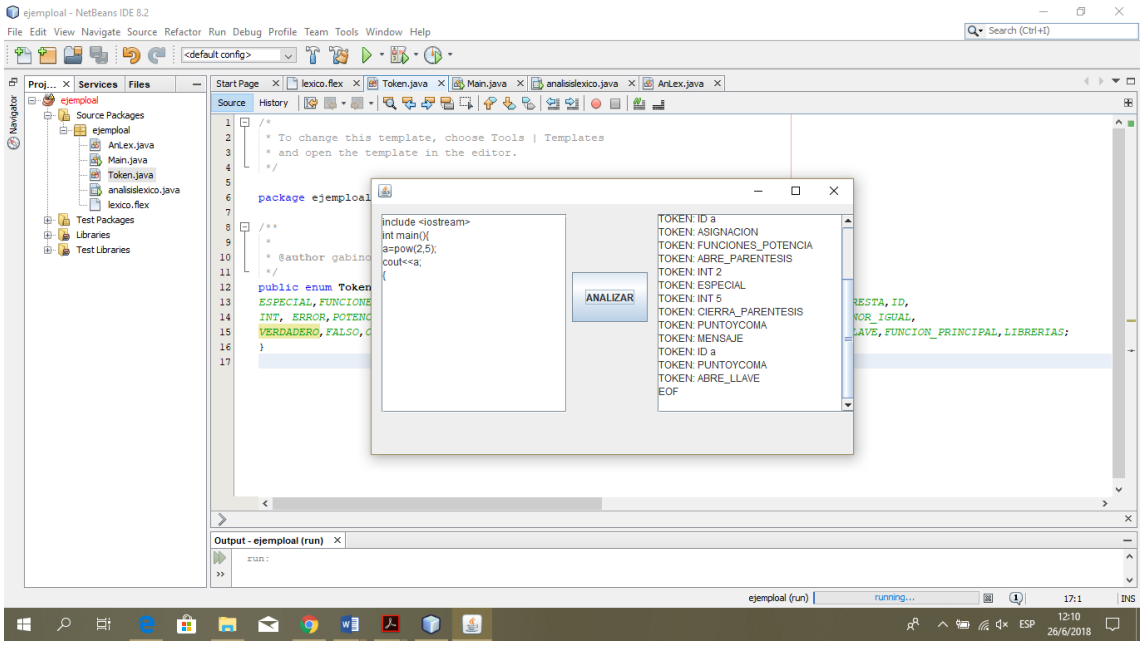


Figura 6: Ejemplo 3

6. Referencias

Gálvez Rojas, S., & Mora Mata, M. Á. (2005). *Traductores y Compiladores con Lex/Yacc, JFlex/cup y JavaCC.* Málaga.

Klein, G., Rowe, S., & Décamps, R. (2015). *The Fast Lexical Analyser Generator* (Vol. Version 1.6.1). Copyright.

http://jflex.de/