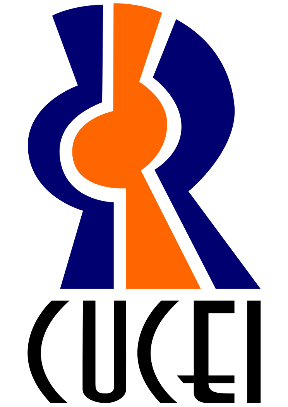
Seminario de Solución de Problemas de Traductores de Lenguajes II

Centro Universitario de Ciencias Exactas e Ingenierías





Actividad 1:

Analizador Léxico

Alumno: Ortiz Macías Guillermo

Sección: D06

Profesor: Ramos Barajas, Armando.

**Introducción**

Esta tarea busca desarrollar un analizar léxico que sea capaz de detectar tokens y clasificarlos. El programa recorre el código ingresado dentro de un archivo y clasifica cada uno de los tokens tales como while, if, elso o bien operadores relacionales y matemáticos.

**Metodología**

El analizador se realizó en el lenguaje C++. El programa funciona mediante un ciclo principal el cual va a recorrer todo el código que ingresó el usuario, formando palabras y buscando tokens. El programa lee caracter por caracter del código dentro del archivo y busca si es un token de un solo caracter (+, -, \*, /, =, etc), de no serlo concatena el caracter a una palabra que se va formando en cada iteración. Cuando se encuentra un token se analiza la palabra ya formada en pasadas iteraciones y después se analiza el token y se clasifica. También toma en cuenta que puede ser un token de dos caracteres (!=, ==, etc). De esta forma el código va analizando y clasificando de forma secuencial los tokens del código.

El código en c++ del analizador es el siguiente:

#include <regex>

#include "LexicalAnalyzer.h"

using namespace std;

LexicalAnalyzer::LexicalAnalyzer(string c) {

code = c;

quoteFound = false;

multipleCharKeyWordFound = false;

index = 0;

currentAnalyzingWord = "";

resultText = "";

keyWordsSize = 28;

}

string LexicalAnalyzer::analyzeCode() {

analyze();

return resultText;

}

void LexicalAnalyzer::analyze(){

for(index = 0; index < code.length(); index++){

if(multipleCharKeyWordFound){

multipleCharKeyWordFound = false;

continue;

}

if(!quoteFound){

if(index + 1 < code.length()){

if(code[index] == '!' && code[index+1] == '='){

twoCharKeyWord("!=");

continue;

}else if(code[index] == '=' && code[index+1] == '='){

twoCharKeyWord("==");

continue;

}else if(code[index] == '<' && code[index+1] == '='){

twoCharKeyWord("<=");

continue;

}else if(code[index] == '>' && code[index+1] == '='){

twoCharKeyWord(">=");

continue;

}else if(code[index] == '&' && code[index+1] == '&'){

twoCharKeyWord("&&");

continue;

}else if(code[index] == '|' && code[index+1] == '|'){

twoCharKeyWord("||");

continue;

}

if(code[index] == ' '){

analyzeWord();

}else if(code[index] == '\n'){

analyzeWord();

}

else if(code[index] == '\t'){

continue;

}

else if(code[index] == '\"'){

quotedText();

}

else if(wordInKeyWords(To\_String(code[index]))){

textInKeyWords();

}

else{

currentAnalyzingWord = currentAnalyzingWord + code[index];

}

}

}else{

currentAnalyzingWord = currentAnalyzingWord + code[index];

if(code[index] == '\"'){

quoteFound = false;

analyzeWord();

}

}

}

}

void LexicalAnalyzer::twoCharKeyWord(string keyWord){

analyzeWord();

currentAnalyzingWord = keyWord;

analyzeWord();

multipleCharKeyWordFound = true;

}

void LexicalAnalyzer::analyzeWord(){

if(regexCompare(regexOneOrMoreSpaces, currentAnalyzingWord)){

currentAnalyzingWord = "";

}

if(currentAnalyzingWord != ""){

currentAnalyzingWord = leftStrip(currentAnalyzingWord);

resultText = resultText + currentAnalyzingWord + " " + lookForKeyWord() + "\n";

currentAnalyzingWord = "";

}

}

void LexicalAnalyzer::quotedText(){

quoteFound = true;

analyzeWord();

currentAnalyzingWord = code[index];

}

bool LexicalAnalyzer::wordInKeyWords(string word){

for(int i(0); i < keyWordsSize; i++){

if(keyWords[i] == word){

return true;

}

}

return false;

}

void LexicalAnalyzer::textInKeyWords(){

analyzeWord();

currentAnalyzingWord = code[index];

analyzeWord();

}

bool LexicalAnalyzer::regexCompare(string myRegex, string text){

return regex\_match(text, regex(myRegex));

}

string LexicalAnalyzer::leftStrip(string str){

size\_t first = str.find\_first\_not\_of(' ');

size\_t last = str.find\_last\_not\_of(' ');

return str.substr(first, (last-first+1));

}

string LexicalAnalyzer::lookForKeyWord(){

if(currentAnalyzingWord == "+"){

return "suma";

}

if(currentAnalyzingWord == "-"){

return "resta";

}

if(currentAnalyzingWord == "\*"){

return "multiplicacion";

}

if(currentAnalyzingWord == "/"){

return "division";

}

if(currentAnalyzingWord == "!="){

return "diferente";

}

if(currentAnalyzingWord == "=="){

return "igualdad";

}

if(currentAnalyzingWord == ">"){

return "mayor que";

}

if(currentAnalyzingWord == "<"){

return "menor que";

}

if(currentAnalyzingWord == "<="){

return "menor igual";

}

if(currentAnalyzingWord == ">="){

return "mayor igual";

}

if(currentAnalyzingWord == "="){

return "igual";

}

if(currentAnalyzingWord == "("){

return "parentesis izquierdo";

}

if(currentAnalyzingWord == ")"){

return "parentesis derecho";

}

if(currentAnalyzingWord == "{"){

return "llave izquierda";

}

if(currentAnalyzingWord == "}"){

return "llave derecha";

}

if(currentAnalyzingWord == ":"){

return "dos puntos";

}

if(currentAnalyzingWord == ";"){

return "punto y coma";

}

if(currentAnalyzingWord == ","){

return "coma";

}

if(currentAnalyzingWord == "&&"){

return "and";

}

if(currentAnalyzingWord == "||"){

return "or";

}

if(currentAnalyzingWord == "!"){

return "not";

}

if(currentAnalyzingWord == "while"){

return "ciclo while";

}

if(currentAnalyzingWord == "if"){

return "condicional if";

}

if(currentAnalyzingWord == "else"){

return "condicional else";

}

if(currentAnalyzingWord == "return"){

return "retorno";

}

/\*if(currentAnalyzingWord == "$"){

return "pesos";

}\*/

if(currentAnalyzingWord == "int"){

return "dato entero";

}

if(currentAnalyzingWord == "float"){

return "dato flotante";

}

if(currentAnalyzingWord == "char"){

return "dato caracter";

}

if(currentAnalyzingWord == "void"){

return "dato vacío";

}

if(regexCompare(regexNumber, currentAnalyzingWord)){

return "constante";

}

if(regexCompare(regexQuote, currentAnalyzingWord)){

return "constante";

}

return "identificador";

}

string LexicalAnalyzer::To\_String(char myChar){

string myStr = "";

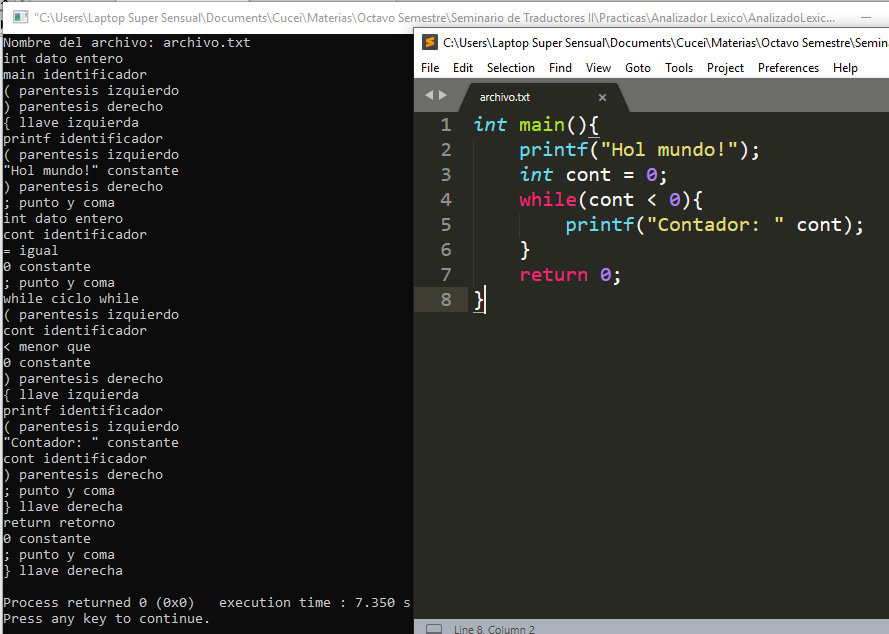
myStr.push\_back(myChar);

return myStr;

}

**Resultados obtenidos.**

El programa pide al usuario un nombre de archivo y realiza el análisis léxico del código.



**Conclusiones**

El análisis léxico es la primera fase de un compilador y puedo ver la utilidad del mismo pues el compilador necesita identificar las distintas partes del código para darles su tratamiento adecuado y también determinar si la sintaxis del mismo es correcta o no. El compilador en una etapa siguiente podría ver errores de sintaxis con sucesiones de tokens no válidas, como un tipo de dato seguido de una constante, sin tener un identificador en medio.