

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 analizador 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, else 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 carácter por carácter del código dentro del archivo y busca si es un token de un solo carácter (+, -, *, /, =, etc), de no serlo concatena el carácter 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"){

```

```

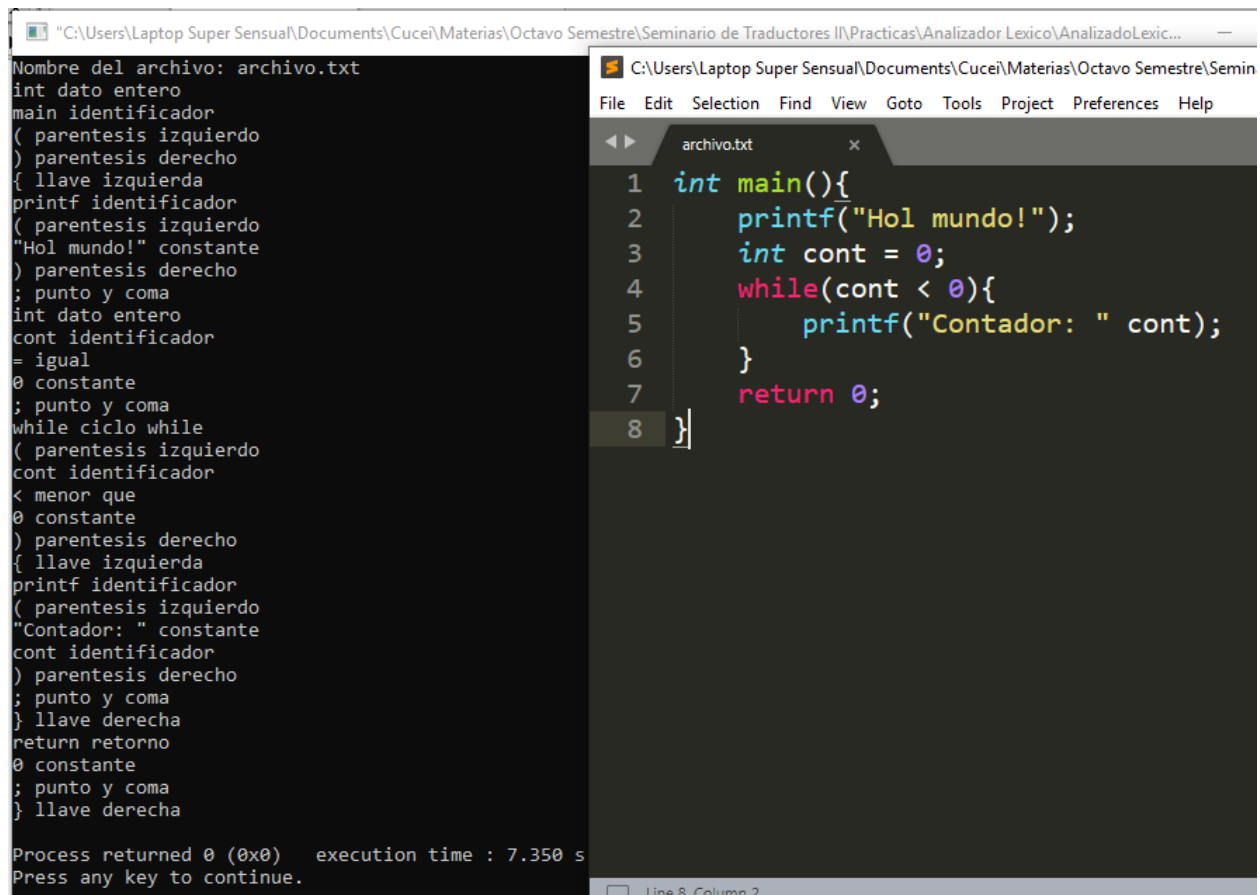
        return "dato entero";
    }
    if(currentAnalyzingWord == "float"){
        return "dato flotante";
    }
    if(currentAnalyzingWord == "char"){
        return "dato caracter";
    }
    if(currentAnalyzingWord == "void"){
        return "dato vacío";
    }
    if(regexCompare(regexNumber, currentAnalyzingWord)){
        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.



The screenshot shows a C++ program in a text editor and its execution output in a console window. The program reads a file named 'archivo.txt' and performs lexical analysis on its contents. The output shows the tokens identified in the file, such as 'int', 'dato', 'entero', 'main', 'identificador', 'parentesis izquierdo', 'parentesis derecho', 'llave izquierda', 'llave derecha', 'printf', 'identificador', 'constante', 'punto y coma', 'while', 'ciclo', 'while', 'menor que', 'return', and 'retorno'. The program also displays the execution time and prompts the user to press any key to continue.

```
Nombre del archivo: archivo.txt
int dato entero
main identificador
( parentesis izquierdo
) parentesis derecho
{ llave izquierda
printf identificador
( parentesis izquierdo
"Hol mundo!" constante
) parentesis derecho
; punto y coma
int dato entero
cont identificador
= igual
0 constante
; punto y coma
while ciclo while
( parentesis izquierdo
cont identificador
< menor que
0 constante
) parentesis derecho
{ llave izquierda
printf identificador
( parentesis izquierdo
"Contador: " constante
cont identificador
) parentesis derecho
; punto y coma
} llave derecha
return retorno
0 constante
; punto y coma
} llave derecha

Process returned 0 (0x0)   execution time : 7.350 s
Press any key to continue.
```

```
1 int main(){
2     printf("Hol mundo!");
3     int cont = 0;
4     while(cont < 0){
5         printf("Contador: " cont);
6     }
7     return 0;
8 }
```

Line 8. Column 2

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.