



Traductores de Lenguajes II
Sección D07

Integrantes:

José Darío Menéndez Gómez | 220286458

Profesor Armando Ramos Barajas

Analizador Léxico

05/02/2023

Contenido

Introducción	3
Objetivo general.....	3
Objetivo particular	3
Desarrollo	4
Autómata.....	4
Tabla de Transiciones	4
Código Fuente	5
analizador_lexico.cpp (main)	5
lexico.h	6
lexico.cpp.....	8
Requisitos funcionales.....	15
Requisitos no funcionales	15
Caja negra.....	15
Caja blanca	16
Conclusiones	17
Bibliografías.....	17
Apéndices	17
Acrónimos	17

Introducción

Un compilador es un programa que traduce el código fuente de un programa de computadora a un lenguaje de máquina, que es entendido por la computadora. Existen dos fases de la compilación: Análisis en el que se realizan 3 tipos de análisis siendo estos el léxico, sintáctico y semántico. Síntesis que es cuando se realiza la traducción del código a lenguaje máquina.

Los tokens son elementos básicos de un lenguaje de programación que se utilizan para formar instrucciones y expresiones. Algunos tokens comunes son los símbolos de puntuación, las palabras clave y las variables. Los tokens en programación sirven para identificar a los elementos que conforman el código fuente de un programa. Los tokens se pueden clasificar en varios grupos, como los símbolos de puntuación, las palabras clave, los identificadores, los operadores y las constantes.

En esta práctica se realizará un analizador léxico mediante el uso del lenguaje de programación C++, para así con la implementación de diversos componentes lograr crear un compilador e interpretar la manera en que este trabaja.

Objetivo general

- Generar un analizador léxico mediante algún lenguaje de programación con al menos 30 tokens.

Objetivo particular

- Simular un compilador, tomando en cuenta únicamente el analizador léxico.
- Interpretar la forma en que un compilador realiza el análisis léxico de un programa.

Autómata

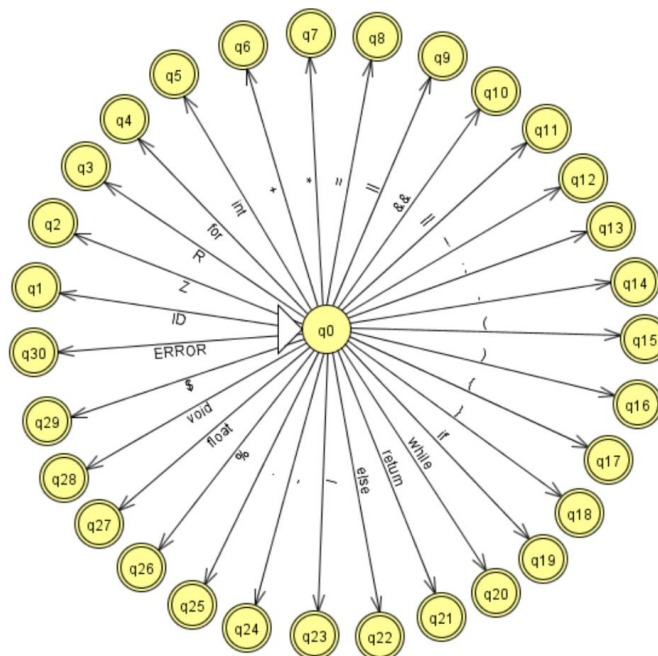


Tabla de Transiciones

[illegible]

Código Fuente

analizador_lexico.cpp (main)

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <wchar.h>
#include <locale.h>
#include <stdlib.h>
#include <fstream>
#include "lexico.h"

using namespace std;

int main(int argc, char* argv[]) {
    fstream file;
    string line;
    file.open("tokens.txt", ios::in);

    if (file.is_open()) {
        while (!file.eof()) {
            getline(file, line);
        }
        file.close();
    }

    Lexico lexico(line);
    cout << "[Simbolo]\t\t[Token]" << endl;

    while (lexico.simbolo.compare("$") != 0) {
        lexico.sigSimbolo();

        cout << lexico.simbolo << "\t\t\t" << lexico.tipoCad(lexico.estado) << endl;
    }

    cout << "===== ";
    cin.get();

    return 0;
}
```

```

lexico.h
#ifndef LEXICO_H
#define LEXICO_H

#include <iostream>
#include <string>

using namespace std;

class SimboloTipo
{
public:
    static const int ERROR = -1;
    static const int IDENTIFICADOR = 0;
    static const int ENTERO = 1;
    static const int REAL = 2;
    static const int FOR = 3;
    static const int TIPOENTERO = 4;
    static const int OPSUMA = 5;
    static const int OPMUL = 6;
    static const int OPRELAC = 7;
    static const int OPOR = 8;
    static const int OPAND = 9;
    static const int OPNOT = 10;
    static const int OPIGUALDAD = 11;
    static const int PUNTOYCOMA = 12;
    static const int COMA = 13;
    static const int PARENTABIERTO = 14;
    static const int PARENTCERRADO = 15;
    static const int CORCHETEABIERTO = 16;
    static const int CORCHETECERRADO = 17;
    static const int ASIGNACION = 18;
    static const int IF = 19;
    static const int WHILE = 20;
    static const int RETURN = 21;
    static const int ELSE = 22;
    static const int FIN = 23;
    static const int OPDIV = 24;
    static const int OPRESTA = 25;
    static const int PUNTO = 26;
    static const int OPMODULO = 27;
    static const int TIPOREAL = 28;
    static const int TIPONULO = 29;
    static const int DO = 30;
};

class Lexico
{
public:
    string simbolo;
    int estado;

    Lexico(string fuente);
    Lexico();

    void entrada(string fuente);
    string tipoCad(int tipo);

```

```
    int sigSimbolo();
    bool terminado();

private:
    string fuente;
    int ind;
    bool continua;
    char c;
    int cont;
    string aux;

    char sigCaracter();
    void sigEstado(int estado);
    void aceptacion(int estado);
    bool esLetra(char c);
    bool esDigito(char c);
    bool esEspacio(char c);
    void retroceso();
};

#endif
```

lexico.cpp

```
#include "lexico.h"
```

```
Lexico::Lexico(string fuente)
{
    ind = 0;
    this->fuente = fuente;
}
```

```
Lexico::Lexico()
{
    ind = 0;
}
```

```
string Lexico::tipoCad(int tipo)
{
    string cad = "";

    switch (tipo)
    {
        case SimboloTipo::IDENTIFICADOR:
            cad = "Identificador";
            break;

        case SimboloTipo::ENTERO:
            cad = "Numero Entero";
            break;

        case SimboloTipo::REAL:
            cad = "Numero Real";
            break;

        case SimboloTipo::FOR:
            cad = "Palabra Reservada";
            break;

        case SimboloTipo::TIPOENTERO:
            cad = "Tipo Entero";
            break;

        case SimboloTipo::TIPOREAL:
            cad = "Tipo Real";
            break;

        case SimboloTipo::TIPONULO:
            cad = "Tipo Retorno Nulo";
            break;

        case SimboloTipo::OPSUMA:
            cad = "Operador Suma";
            break;

        case SimboloTipo::OPRESTA:
            cad = "Operador Resta";
            break;

        case SimboloTipo::OPMUL:
            cad = "Operador Multiplicacion";
```



```
        break;

case SimboloTipo::OPDIV:
    cad = "Operador Division";
    break;

case SimboloTipo::OPMODULO:
    cad = "Operador Modulo";
    break;

case SimboloTipo::OPRELAC:
    cad = "Operador Relacional";
    break;

case SimboloTipo::OPOR:
    cad = "Operador OR";
    break;

case SimboloTipo::OPAND:
    cad = "Operador AND";
    break;

case SimboloTipo::OPNOT:
    cad = "Operador NOT";
    break;

case SimboloTipo::OPIGUALDAD:
    cad = "Operador de Igualdad";
    break;

case SimboloTipo::PUNTO:
    cad = "Punto";
    break;

case SimboloTipo::PUNTOYCOMA:
    cad = "Punto y Coma";
    break;

case SimboloTipo::COMA:
    cad = "Coma";
    break;

case SimboloTipo::PARENTABIERTO:
    cad = "Parentesis abierto";
    break;

case SimboloTipo::PARENTCERRADO:
    cad = "Parentesis cerrado";
    break;

case SimboloTipo::CORCHETEABIERTO:
    cad = "Corchete Abierto";
    break;

case SimboloTipo::CORCHETECERRADO:
    cad = "Corchete Cerrado";
    break;
```

```

        case SimboloTipo::ASIGNACION:
            cad = "Operador de Asignacion";
            break;

        case SimboloTipo::IF:
            cad = "Palabra Reservada";
            break;

        case SimboloTipo::WHILE:
            cad = "Palabra Reservada";
            break;

        case SimboloTipo::RETURN:
            cad = "Palabra Reservada";
            break;

        case SimboloTipo::ELSE:
            cad = "Palabra Reservada";
            break;

        case SimboloTipo::DO:
            cad = "Palabra Reservada";
            break;

        case SimboloTipo::FIN:
            cad = "Fin deCodigo";
            break;
    }

    return cad;
}

int Lexico::sigSimbolo()
{
    estado = 0;
    continua = true;
    simbolo = "";
    cont = 0;
    //aux = simbolo;
    while (continua)
    {
        c = sigCaracter();

        switch (estado)
        {
            case 0:
                if (esLetra(c)) {
                    estado = 1;
                    simbolo += c;
                }
                if (esDigito(c)) {
                    estado = 2;
                    simbolo += c;
                }
                if (c == '+') {
                    aceptacion(5);
                }
                if (c == '-') {

```

```

        aceptacion(25);
    }
    if (c == '*') aceptacion(6);
    if (c == '/') aceptacion(24);
    if (c == '%') aceptacion(27);
    if (c == '<' || c == '>') {
        estado = 5;
        simbolo += c;
    }
    if (c == '|') {
        estado = 6;
        simbolo += c;
    }
    if (c == '&') {
        estado = 7;
        simbolo += c;
    }
    if (c == '!') {
        estado = 8;
        simbolo += c;
    }
    if (c == '.') aceptacion(26);
    if (c == ';') aceptacion(12);
    if (c == ',') aceptacion(13);
    if (c == '(') aceptacion(14);
    if (c == ')') aceptacion(15);
    if (c == '{') aceptacion(16);
    if (c == '}') aceptacion(17);
    if (c == '=') {
        estado = 9;
        simbolo += c;
    }
    else {
        if (c == '$') aceptacion(23);
    }
    break;
case 1:
    if (esLetra(c) || esDigito(c)) {
        estado = 1;
        simbolo += c;
    }
    else if (!esLetra(c) || !esDigito(c))
    {
        aux = simbolo;
        if (aux == "int") {
            aceptacion(4);
        }
        else {
            aceptacion(0);
        }
        if (aux == "float") {
            aceptacion(28);
        }
        if (aux == "void") {
            aceptacion(29);
        }
    }

```

```

        if (aux == "if") {
            aceptacion(19);
        }
        if (aux == "while") {
            aceptacion(20);
        }
        if (aux == "return") {
            aceptacion(21);
        }
        if (aux == "else") {
            aceptacion(22);
        }
        if (aux == "do") {
            aceptacion(30);
        }
        if (aux == "for") {
            aceptacion(3);
        }
    }
    break;

case 2:
    if (esDigito(c)) {
        estado = 2;
        simbolo += c;
    }
    else if (c == '.') {
        estado = 3;
        simbolo += c;
    }
    else if (c != '.' || !esDigito(c)) {
        aceptacion(1);
    }
    break;

case 3:
    if (esDigito(c)) {
        estado = 4;
        simbolo += c;
    }
    break;

case 4:
    if (esDigito(c)) {
        estado = 4;
        simbolo += c;
    }
    else {
        aceptacion(2);
    }
    break;

case 5:
    if (c != '=') {
        aceptacion(7);
    }
    else if (c == '=') {
        aceptacion(7);
    }
    break;

```

```

        case 6:
            if (c != '|') {
                aceptacion(23);
            }
            else if (c == '|') {
                aceptacion(8);
            }
            break;
        case 7:
            if (c != '&') {
                aceptacion(23);
            }
            else if (c == '&') {
                aceptacion(9);
            }
            break;
        case 8:
            if (c != '=') {
                aceptacion(10);
            }
            else if (c == '=') {
                aceptacion(11);
            }
            break;
        case 9:
            if (c != '=') {
                aceptacion(18);
            }
            else if (c == '=') {
                aceptacion(11);
            }
            break;
    }

}

return estado;
}

```

```

char Lexico::sigCaracter()
{
    if (terminado()) return '$';

    return fuente[ind++];
}

void Lexico::sigEstado(int estado)
{
    this->estado = estado;
    if (estado != 0)
    {
        simbolo += c;
    }
}

```

```

void Lexico::aceptacion(int estado)
{
    sigEstado(estado);
    continua = false;
}

bool Lexico::terminado()
{
    return ind >= fuente.length();
}

bool Lexico::esLetra(char c)
{
    return c >= 'a' && c <= 'z' || c == '_' || c >= 'A' && c <= 'Z';
}

bool Lexico::esDigito(char c)
{
    return isdigit(c);
}

bool Lexico::esEspacio(char c)
{
    return c == ' ' || c == '\t';
}

void Lexico::retroceso()
{
    if (c != '$') ind--;
    continua = false;
}

```

Requisitos funcionales

- Poder captar un mínimo de 30 palabras clave, el analizador léxico.
- Ser capaz de poder crear un autómata con su comportamiento.
- Poder captar la extensión de las palabras clave.
- Estará validado para que reconozca las palabras automáticamente sin confundirse el analizador con otras palabras.

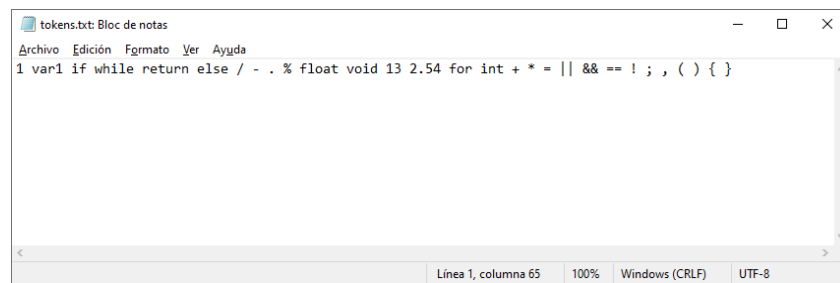
Requisitos no funcionales

- Ser de cómodo uso al usuario.
- Deberá contar con una Interfaz para que el usuario pueda usarlo fácilmente.
- El analizador será codificado en el lenguaje de preferencia del desarrollador (C++).

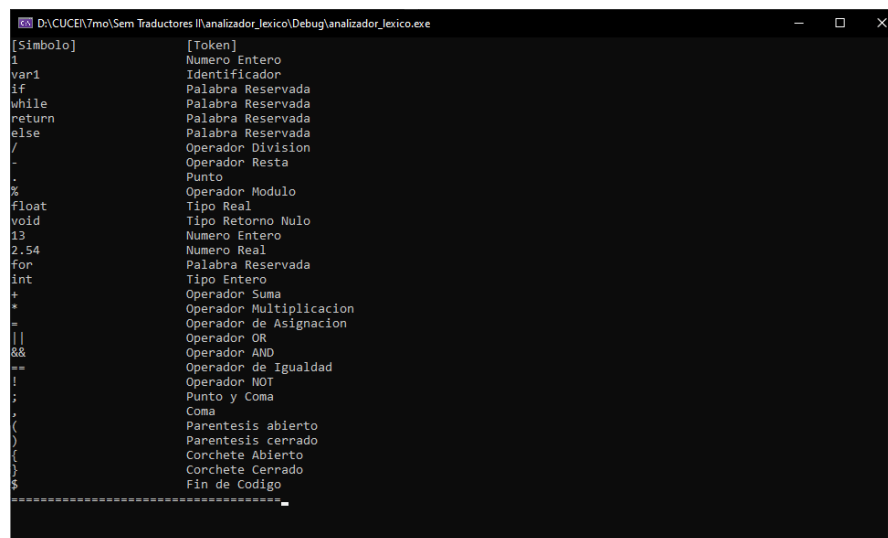
Caja negra

Para la prueba de caja negra tenemos 3 pasos

1. Editar el archivo de texto llamado “tokens.txt”
2. Ejecutar Programa
3. Evaluar Salida



```
tokens.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
1 var1 if while return else / - . % float void 13 2.54 for int + * = || && == ! ; , ( ) { }
Línea 1, columna 65 100% Windows (CRLF) UTF-8
```



```
D:\CUCE\7mo\Sem Traductores II\analizador_lexico\Debug\analizador_lexico.exe
[Simbolo] [Token]
1 Numero Entero
var1 Identificador
if Palabra Reservada
while Palabra Reservada
return Palabra Reservada
else Palabra Reservada
/ Operador Division
- Operador Resta
. Punto
% Operador Modulo
float Tipo Real
void Tipo Retorno Nulo
13 Numero Entero
2.54 Numero Real
for Palabra Reservada
int Tipo Entero
+ Operador Suma
* Operador Multiplicacion
= Operador de Asignacion
|| Operador OR
&& Operador AND
== Operador de Igualdad
! Operador NOT
; Punto y Coma
, Coma
( Parentesis abierto
) Parentesis cerrado
{ Corchete Abierto
} Corchete Cerrado
$ Fin deCodigo
=====
```

Caja blanca

Para la prueba de caja blanca se parte de un estado inicial 0 y se evalúa de carácter en carácter del archivo de texto, usando el autómata para determinar los tokens que se están manejando. Este autómata se representa como un switch que evalúa los estados y los acepta o rechaza con la ayuda de funciones que determinan el estado y el carácter actual, deteniendo el análisis de la cadena para determinar el token.

```
int Lexico::sigSimbolo()
{
    estado = 0;
    continua = true;
    simbolo = "";
    cont = 0;
    //aux = simbolo;
    while (continua)
    {
        c = sigCaracter();

        switch (estado) { ... }

    }

    return estado;
}

char Lexico::sigCaracter()
{
    if (terminado()) return '$';

    return fuente[ind++];
}

void Lexico::sigEstado(int estado)
{
    this->estado = estado;
    if (estado != 0)
    {
        simbolo += c;
    }
}

void Lexico::aceptacion(int estado)
{
    sigEstado(estado);
    continua = false;
}
```

```
bool Lexico::terminado()
{
    return ind >= fuente.length();
}

bool Lexico::esLetra(char c)
{
    return c >= 'a' && c <= 'z' || c == '_' || c >= 'A' && c <= 'Z';
}

bool Lexico::esDigito(char c)
{
    return isdigit(c);
}

bool Lexico::esEspacio(char c)
{
    return c == ' ' || c == '\t';
}

void Lexico::retroceso()
{
    if (c != '$') ind--;
    continua = false;
}
```


Conclusiones

Considero que los conocimientos adquiridos en la clase de Teoría de la Computación son esenciales y fundamentales para el desarrollo de los analizadores con los que cuenta un compilador. Como nos pudimos dar cuenta, se manejó un autómata finito no determinista que determina los estados por los que se pasan para la construcción de palabras reservadas de un lenguaje de programación. De la misma forma se desarrollarán los siguientes analizadores, solo que con el manejo de distintos autómatas.

Bibliografías

- Publicado por pmoinformatica.com. (s/f). Requerimientos no funcionales: Ejemplos. Pmoinformatica.com. Recuperado el 04 de febrero de 2023, de <http://www.pmoinformatica.com/2015/05/requerimientos-no-funcionales-ejemplos.html>
- Seguir, S. (s/f). Software caja negra y caja blanca. Slideshare.net. Recuperado el 04 de febrero de 2023, de <https://es.slideshare.net/StudentPc/software-caja-negra-y-caja-blanca>
- Siriwardhana, S. (2020, octubre 22). Tutorial de diagramas de casos de uso (Guía con ejemplos). Blog de Creately. <https://creately.com/blog/es/diagramas/tutorial-diagrama-caso-de-uso/>
- Tokens. (s/f). Zator.com. Recuperado el 04 de febrero de 2023, de https://www.zator.com/Cpp/E3_2.htm

Apéndices

N/A

Acrónimos

N/A