UNIVERSIDAD CENTROAMERICANA "JOSÉ SIMEÓN CAÑAS"

FACULTAD DE INGENIERÍA Y ARQUITECTURA

Teoría de lenguajes de programación

Catedrático:

Ing. Jaime Clímaco

Integrantes:

García Arévalo, José Enrique 00093619
Hernández García, Rodrigo Anibal 00050519
López Henríquez, Guillermo Daniel 00026018
Cruz Cader, Miguel Alejandro 00019018
Escobar Hernandez, Luis Gustavo 00091318
Alejandro Enrique Rivera Vasquez 00011218

Índice:

Introducción:

En esta documentación, abordamos las fases posteriores al componente central del proyecto de compiladores, centrándonos en el análisis semántico y la generación de código intermedio. Con la fase principal completada, donde se implementó un analizador sintáctico y un manejador de errores, ahora nos sumergimos en aspectos clave para la coherencia y optimización del código.

Exploramos la integración del analizador semántico, encargado de detectar errores en la estructura semántica del código, y la generación de código intermedio, una etapa esencial para simplificar la traducción al código objeto final. Este documento ofrece una visión concisa de nuestras decisiones y estrategias implementadas en estas fases adicionales, complementando el trabajo previo.
Con el compromiso del equipo en mente, presentamos esta documentación como un recurso ágil para comprender nuestras contribuciones y evaluaciones en el desarrollo del compilador.

Un parser y un lexer son componentes esenciales en el proceso de construcción de compiladores y analizadores de lenguajes de programación. Estas dos partes trabajan en conjunto para entender y procesar el código fuente escrito por un programador.

Lexer (Analizador Léxico)

El analizador léxico, también conocido como lexer o scanner, es la primera fase del proceso de análisis de un compilador. Su función principal es dividir el código fuente en piezas más pequeñas llamadas tokens. Estos tokens son las unidades léxicas básicas que constituyen el lenguaje de programación. Algunos ejemplos de tokens son palabras clave, identificadores, operadores y constantes.
El lexer utiliza expresiones regulares y reglas de reconocimiento para identificar y clasificar los diferentes tipos de tokens presentes en el código fuente. Una vez que ha analizado el código, pasa la secuencia de tokens al parser.

Parser (Analizador Sintáctico)

El analizador sintáctico, o parser, es la segunda fase del proceso de compilación. Su tarea principal es analizar la estructura del código fuente y construir un árbol de sintaxis abstracta (Abstract Syntax Tree, AST). Este árbol representa la estructura gramatical del programa.
El parser utiliza reglas gramaticales definidas para el lenguaje y verifica si la secuencia de tokens generada por el lexer sigue estas reglas. Si encuentra errores sintácticos, informa sobre ellos. Si la secuencia es válida, construye el AST que luego se utiliza en las fases posteriores del compilador, como el análisis semántico y la generación de código intermedio.

Documentación:

La gramática formal realizada a continuación se define como una cuádrupla G = (N, Σ, P, S), donde cada elemento representa un aspecto crucial del lenguaje de programación que se está analizando. Esta gramática es esencial para comprender la estructura y las reglas que rigen el lenguaje, permitiendo tanto su análisis como su procesamiento por parte de compiladores o intérpretes. A continuación, se detalla cada componente de la gramática:

Conjunto de Símbolos No Terminales (N): Estos son los símbolos que representan las unidades abstractas o las construcciones gramaticales del lenguaje. En esta gramática, los no terminales incluyen elementos como 'S' (símbolo inicial), 'LIB', 'PARAM', 'FBODY', entre otros. Cada uno de estos no terminales juega un papel específico en la definición de la estructura del lenguaje, como la declaración de funciones, bloques de código, parámetros, etc.

Conjunto de Símbolos Terminales (Σ): Estos símbolos son los elementos básicos del lenguaje, como palabras clave, operadores y otros tokens que no se pueden descomponer en componentes más simples. En esta gramática, los terminales incluyen 'int', 'float', 'if', 'while', operadores como '+', '-', y otros elementos básicos del lenguaje.

Conjunto de Reglas de Producción (P): Estas reglas definen cómo se pueden combinar los símbolos no terminales y terminales para formar estructuras válidas en el lenguaje. Por ejemplo, una regla como S -> int identificador OPENNUMFUN S define cómo se puede formar una declaración de función que retorna un entero.

Símbolo Inicial (S): Este es el punto de partida de cualquier derivación en la gramática. Todas las construcciones válidas en el lenguaje deben comenzar con este símbolo. En esta gramática, 'S' es el símbolo inicial, lo que indica que todas las derivaciones comienzan desde este punto.

La gramática también incluye los conjuntos **FIRST** y **FOLLOW**, que son fundamentales en el análisis sintáctico, especialmente en el diseño de analizadores sintácticos predictivos. Estos conjuntos ayudan a determinar qué regla de producción se debe aplicar en función del símbolo de entrada actual y el contexto.

Conjuntos FIRST: Contienen los símbolos terminales que pueden aparecer al inicio de las cadenas derivadas de un no terminal. Por ejemplo, FIRST(S) incluye 'int', 'float', 'if', etc., lo que significa que estas son las posibles entradas con las que puede comenzar una cadena derivada de 'S'.

Conjuntos FOLLOW: Contienen los símbolos terminales que pueden aparecer inmediatamente después de un no terminal en alguna "sentencia" derivada. Por ejemplo, FOLLOW(S) incluye 'eof', indicando que el final de archivo puede seguir inmediatamente después de una construcción iniciada por 'S'.

Gramática formal

G = (N, Σ, P, S)

Conjunto de No Terminales (N)

no_terminales = ['S', 'LIB', 'LIB2', 'LIBEND', 'PARAM', 'ADDPARAM', 'NUMRETURN', 'ARITMETIC', 'SUMA2', 'RETURNNUM', 'GHANDLER', 'CHARFRETURN', 'RETURNCHAR', 'FLOATN', 'DECIMAL', 'THANDLER', 'OPENNUMFUN', 'OPENCHARFUN', 'OPENVOID', 'FBODY', 'VOIDRETURN', 'RETURNVOID', 'CONDITIONSHANDLER', 'LOGICSIMBOLS', 'NOTHANDLER', 'ELSEXTENTION', 'PRINTCONT', 'FUNCUSE', 'USEPARAM']

Conjunto de Terminales (Σ)

terminales = ['hashtoken', 'int', 'char', 'float', 'keyword', 'comentario', 'identificador', 'LPAREN', 'RPAREN', 'asignacion', 'coma', 'finInstruccion', 'inicioBloque', 'finBloque', 'else', 'printf', 'aumentarvar', 'reducirvar', 'single_quote', 'and', 'or', 'lesser_than', 'greater_than', 'not', 'NUMBER', 'dot', 'eof', 'cadena']

S: Símbolo inicial S = 0

P: Conjunto de reglas de producción

La gramática descrita en el formato S->alfa es la siguiente:

S -> hashtoken identificador lesser_than LIB
S -> int identificador OPENNUMFUN S
S -> char identificador OPENCHARFUN S
S -> float identificador OPENNUMFUN S
S -> keyword identificador OPENVOID S
S -> comentario S
S -> identificador LPAREN USEPARAM RPAREN finInstruccion S
OPENVOID -> LPAREN PARAM FBODY VOIDRETURN finBloque
OPENNUMFUN -> asignacion RETURNNUM S
OPENNUMFUN -> LPAREN PARAM FBODY NUMRETURN finBloque
FBODY -> comentario FBODY
FBODY -> int identificador asignacion RETURNNUM FBODY
FBODY -> float identificador asignacion RETURNNUM FBODY
FBODY -> char identificador asignacion RETURNCHAR FBODY
FBODY -> identificador LPAREN USEPARAM RPAREN finInstruccion FBODY
FBODY -> if LPAREN CONDITIONSHANDLER RPAREN inicioBloque FBODY finBloque FBODY
FBODY -> while LPAREN CONDITIONSHANDLER RPAREN inicioBloque FBODY finBloque FBODY
FBODY -> else ELSEXTENTION
FBODY -> printf LPAREN PRINTCONT RPAREN finInstruccion FBODY
FBODY -> aumentarvar finInstruccion FBODY
FBODY -> reducirvar finInstruccion FBODY
FBODY -> keyword
FBODY -> finBloque
PRINTCONT -> cadena THANDLER
PRINTCONTPRINTCONT -> identificador THANDLER PRINTCONT
PRINTCONT -> NUMBER THANDLER PRINTCONT
PRINTCONT -> coma PRINTCONT PRINTCONT
PRINTCONT -> RPAREN
ELSEXTENTION -> inicioBloque FBODY finBloque FBODY
ELSEXTENTION -> if FBODY
CONDITIONSHANDLER -> identificador THANDLER LOGICSIMBOLS
CONDITIONSHANDLER -> NUMBER THANDLER LOGICSIMBOLS
CONDITIONSHANDLER -> not NOTHANDLER LOGICSIMBOLS
LOGICSIMBOLS -> and NOTHANDLER LOGICSIMBOLS
LOGICSIMBOLS -> or NOTHANDLER LOGICSIMBOLS
LOGICSIMBOLS -> lesser_than NOTHANDLER LOGICSIMBOLS
LOGICSIMBOLS -> greater_than NOTHANDLER LOGICSIMBOLS
LOGICSIMBOLS -> not NOTHANDLER LOGICSIMBOLS
LOGICSIMBOLS -> RPAREN
NOTHANDLER -> not THANDLER

NOTHANDLER -> identificador THANDLER
NOTHANDLER -> NUMBER THANDLER
OPENCHARFUN -> asignacion RETURNCHAR S
OPENCHARFUN -> LPAREN PARAM CHARFRETURN finBloque
VOIDRETURN -> keyword RETURNVOID
VOIDRETURN -> finBloque
CHARFRETURN -> keyword RETURNCHAR
NUMRETURN -> keyword RETURNNUM
RETURNVOID -> single_quote identificador single_quote finInstruccion
RETURNVOID -> identificador ARITMETIC
RETURNVOID -> LPAREN THANDLER ARITMETIC
RETURNVOID -> NUMBER FLOATN ARITMETIC
RETURNVOID -> finBloque
RETURNCHAR -> single_quote identificador single_quote finInstruccion
RETURNCHAR -> identificador finInstruccion
RETURNCHAR -> LPAREN char RPAREN identificador finInstruccion
RETURNNUM -> identificador ARITMETIC
RETURNNUM -> LPAREN THANDLER ARITMETIC
RETURNNUM -> NUMBER FLOATN ARITMETIC
ARITMETIC -> LPAREN THANDLER ARITMETIC
ARITMETIC -> RPAREN ARITMETIC
ARITMETIC -> PLUS GHANDLER ARITMETIC
ARITMETIC -> MINUS GHANDLER ARITMETIC
ARITMETIC -> TIMES GHANDLER ARITMETIC
ARITMETIC -> DIVIDE GHANDLER ARITMETIC
ARITMETIC -> coma THANDLER ARITMETIC
ARITMETIC -> finInstruccion
GHANDLER -> identificador
GHANDLER -> LPAREN THANDLER
GHANDLER -> NUMBER FLOATN
THANDLER -> identificador
THANDLER -> NUMBER FLOATN
THANDLER -> cadena
PARAM -> int identificador PARAM
PARAM -> char identificador PARAM
PARAM -> float identificador PARAM
PARAM -> RPAREN inicioBloque
PARAM -> coma PARAM
USEPARAM -> identificador USEPARAM
USEPARAM -> NUMBER FLOATN USEPARAM
USEPARAM -> single_quote identificador single_quote USEPARAM
USEPARAM -> coma USEPARAM
USEPARAM -> RPAREN
LIB -> identificador LIB2
LIB2 -> greater_than S
LIB2 -> dot identificador greater_than S
FLOATN -> NUMBER DECIMAL
DECIMAL -> dot NUMBER
DECIMAL -> MINUS
DECIMAL -> PLUS
DECIMAL -> TIMES
DECIMAL -> DIVIDE
DECIMAL -> RPAREN
DECIMAL -> finInstruccion
DECIMAL -> coma
S -> eo

Conjuntos First

FIRST(S) = {'hashtoken', 'int', 'char', 'float', 'keyword', 'comentario', 'identificador', 'printf', 'aumentarvar', 'reducirvar', 'single_quote', 'and', 'or', 'lesser_than', 'greater_than', 'not', 'NUMBER', 'dot', 'eof', 'cadena', 'else', 'while', 'LPAREN', 'RPAREN', 'finInstruccion', 'inicioBloque', 'finBloque'}
FIRST(LIB) = {'identificador'}
FIRST(LIB2) = {'greater_than', 'dot'}
FIRST(LIBEND) = {}
FIRST(PARAM) = {'int', 'char', 'float', 'RPAREN'}
FIRST(ADDPARAM) = {'int', 'char', 'float', 'RPAREN'}
FIRST(NUMRETURN) = {'int', 'char', 'float', 'RPAREN'}
FIRST(ARITMETIC) = {'LPAREN', 'RPAREN', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'coma', 'finInstruccion'}
FIRST(SUMA2) = {'PLUS', 'MINUS'}
FIRST(RETURNNUM) = {'identificador', 'LPAREN', 'NUMBER'}
FIRST(GHANDLER) = {'identificador', 'LPAREN', 'NUMBER', 'finInstruccion', 'RPAREN'}
FIRST(CHARFRETURN) = {'char', 'single_quote', 'identificador', 'finInstruccion'}
FIRST(RETURNCHAR) = {'single_quote', 'identificador', 'LPAREN'}
FIRST(FLOATN) = {'NUMBER'}
FIRST(DECIMAL) = {'dot', 'MINUS', 'PLUS', 'TIMES', 'DIVIDE', 'RPAREN', 'finInstruccion', 'coma'}
FIRST(THANDLER) = {'identificador', 'NUMBER', 'cadena'}
FIRST(OPENNUMFUN) = {'asignacion', 'LPAREN'}
FIRST(OPENCHARFUN) = {'asignacion', 'LPAREN'}

FIRST(OPENVOID) = {'asignacion', 'LPAREN'}
FIRST(FBODY) = {'comentario', 'int', 'float', 'char', 'identificador', 'if', 'while', 'else', 'printf', 'aumentarvar', 'reducirvar', 'keyword', 'finBloque', 'finInstruccion'}
FIRST(VOIDRETURN) = {'single_quote', 'identificador', 'LPAREN', 'finBloque', 'finInstruccion'}
FIRST(RETURNVOID) = {'single_quote', 'identificador', 'LPAREN', 'NUMBER', 'finBloque', 'finInstruccion'}
FIRST(CONDITIONSHANDLER) = {'identificador', 'NUMBER', 'not', 'RPAREN'}
FIRST(LOGICSIMBOLS) = {'and', 'or', 'lesser_than', 'greater_than', 'not', 'RPAREN'}
FIRST(NOTHANDLER) = {'not', 'identificador', 'NUMBER'}
FIRST(ELSEXTENTION) = {'inicioBloque', 'if'}
FIRST(PRINTCONT) = {'cadena', 'identificador', 'NUMBER', 'RPAREN'}
FIRST(FUNCUSE) = {'identificador', 'LPAREN'}
FIRST(USEPARAM) = {'identificador', 'NUMBER', 'single_quote', 'coma', 'RPAREN'}

Conjuntos Follow

FOLLOW(S) = {'eof'}
FOLLOW(LIB) = {'greater_than', 'dot', 'identificador', 'eof'}
FOLLOW(LIB2) = {'eof'}
FOLLOW(LIBEND) = {'eof'}
FOLLOW(PARAM) = {'RPAREN'}
FOLLOW(ADDPARAM) = {'RPAREN'}
FOLLOW(NUMRETURN) = {'finBloque'}
FOLLOW(ARITMETIC) = {'finInstruccion', 'RPAREN'}
FOLLOW(SUMA2) = {'finInstruccion', 'RPAREN'}
FOLLOW(RETURNNUM) = {'finInstruccion', 'RPAREN'}
FOLLOW(GHANDLER) = {'finInstruccion', 'RPAREN'}
FOLLOW(CHARFRETURN) = {'finInstruccion', 'RPAREN'}
FOLLOW(RETURNCHAR) = {'finInstruccion', 'RPAREN'}
FOLLOW(FLOATN) = {'PLUS', 'MINUS', 'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(DECIMAL) = {'PLUS', 'MINUS', 'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(THANDLER) = {'PLUS', 'MINUS', 'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(OPENNUMFUN) = {'PLUS', 'MINUS', 'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(OPENCHARFUN) = {'PLUS', 'MINUS', 'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(OPENVOID) = {'PLUS', 'MINUS', 'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(FBODY) = {'else', 'finBloque'}
FOLLOW(VOIDRETURN) = {'else', 'finBloque'}
FOLLOW(RETURNVOID) = {'else', 'finBloque'}
FOLLOW(CONDITIONSHANDLER) = {'RPAREN', 'and', 'or', 'lesser_than', 'greater_than', 'not', 'finInstruccion', 'finBloque', 'else', 'coma'}
FOLLOW(LOGICSIMBOLS) = {'identificador', 'NUMBER', 'not', 'RPAREN', 'and', 'or', 'lesser_than', 'greater_than', 'finInstruccion', 'else', 'coma'}
FOLLOW(NOTHANDLER) = {'identificador', 'NUMBER', 'RPAREN', 'and', 'or', 'lesser_than', 'greater_than', 'finInstruccion', 'else', 'coma'}
FOLLOW(ELSEXTENTION) = {'identificador', 'NUMBER', 'LPAREN', 'cadena', 'else', 'finBloque', 'RPAREN', 'and', 'or', 'lesser_than', 'greater_than', 'not', 'finInstruccion', 'coma'}
FOLLOW(PRINTCONT) = {'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(FUNCUSE) = {'RPAREN', 'finInstruccion', 'coma'}
FOLLOW(USEPARAM) = {'RPAREN', 'finInstruccion', 'coma'}

Tabla LL1

FIRST	FOLLOW	Nuntermi nal	harktake n	identific ador	lazzer_th en	int	char	float	keyword	comentar io	LPAREN	RPAREN	finInstru cciona	finBloqu e	asignacio n	if	inicioBlo que	while	else	printf	aumentar var	reducirv ar	cadena	NUMBER	coma	nat	and	or	greater_ than	zinglo_qu ote	FLOATN	PLUS	MINUS	TIMES	DIVIDE	dot	greater	$
{harktakon,if	{$,harktakon	S	S -> harktako	S -> identificador LPAREN L		S -> int identi	S -> char iden	S -> float iden	S -> keyword	S -> comentaria S																												
{LPAREN}	{harktakon,in	OPENVOID									OPENVOID -> LPAREN PARAM FBODY VOIDRETURN finBloque																										
{asignacion,L	{harktakon,in	OPENNUMFUN									OPENNUMFUN -> LPAREN PARAM FBODY NUMRETURN				OPENNUMFUN -> asignacion RETURNNUMS																							
{camentaria,	{keyword,fin	FBODY		FBODY -> identificador LPA		FBODY -> inti	FBODY -> cha	FBODY -> floa	FBODY -> key	FBODY -> camentaria FBODY				FBODY -> finBloque		FBODY -> if LPAREN CONDI		FBODY -> whil	FBODY -> else	FBODY -> prin	FBODY -> aum	FBODY -> reducirvar finInstruccian FBODY														
{cadena,iden	{RPAREN,ca	PRINTCONT		PRINTCONT -> identificador THANDLER PRINTCONT								PRINTCONT -> RPAREN											PRINTCONT ->	PRINTCONT ->	PRINTCONT -> coma PRINTCONT PRINTCONT												
{inicioBloque	{keyword,fin	ELSEXTENTION														ELSEXTENTIO	ELSEXTENTION -> inicioBloque FBODY finBloque FBODY																				
{identificado	{RPAREN}	CONDITIONSHANDLER		CONDITIONSHANDLER -> identificador THANDLER LOGICSIMBOLS																				CONDITIONSHANDLER -> N	CONDITIONSHANDLER -> nat NOTHANDLER LOGICSIMBOLS												
{and,or,lazze	{RPAREN}	LOGICSIMBOLS			LOGICSIMBOLS -> lazzer_than NOTHANDLER LOGICSIMBOLS							LOGICSIMBOLS -> RPAREN														LOGICSIMBO	LOGICSIMBO	LOGICSIMBO	LOGICSIMBOLS -> greater_than NOTHANDLER LOGICSIMBOLS									
{nat,identific	{and,or,lazze	NOTHANDLER		NOTHANDLER -> identificador THANDLER																				NOTHANDLER -> NUMBER T	NOTHANDLER -> nat THANDLER												
{asignacion,L	{harktakon,in	OPENCHARFUN									OPENCHARFUN -> LPAREN PARAM CHARFRETURN finB				OPENCHARFUN -> asignacion RETURNCHAR S																							
{keyword,fin	{finBloque}	VOIDRETURN							VOIDRETURN -> keyword RETURNVOID					VOIDRETURN -> finBloque																							
{keyword}	{finBloque}	CHARFRETURN							CHARFRETURN -> keyword RETURNCHAR																												
{keyword}	{finBloque}	NUMRETURN							NUMRETURN -> keyword RETURNNUM																												
{zinglo_quote	{finBloque}	RETURNVOID		RETURNVOID -> identificador ARITMETIC							RETURNVOID -> LPAREN THANDLER ARIT			RETURNVOID -> finBloque										RETURNVOID -> NUMBER FLOATN ARITMETIC					RETURNVOID -> zinglo_quote identificador zinglo_quote finInstruccian							
{zinglo_quote	{camentaria,	RETURNCHAR		RETURNCHAR -> identificador finInstruccian							RETURNCHAR -> LPAREN char RPAREN identificador finInstruccian																	RETURNCHAR -> zinglo_quote identificador zinglo_quote finInstruccian							
{identificado	{harktakon,in	RETURNNUM		RETURNNUM -> identificador ARITMETIC							RETURNNUM -> LPAREN THANDLER ARITMETIC													RETURNNUM -> NUMBER FLOATN ARITMETIC												
{LPAREN,RP	{finBloque,ke	ARITMETIC									ARITMETIC ->	ARITMETIC ->	ARITMETIC -> finInstruccian												ARITMETIC -> coma THANDLER ARITMETIC					ARITMETIC ->	ARITMETIC ->	ARITMETIC ->	ARITMETIC -> DIVIDE GHANDLER ARITMETIC			
{identificado	{LPAREN,RP	GHANDLER		GHANDLER -> identificador							GHANDLER -> LPAREN THANDLER													GHANDLER -> NUMBER FLOATN												
{identificado	{cadena,iden	THANDLER		THANDLER -> identificador																			THANDLER ->	THANDLER -> NUMBER FLOATN												
{int,char,floa	{camentaria,	PARAM				PARAM -> int	PARAM -> ch	PARAM -> float identificador PARAM				PARAM -> RPAREN inicioBloque													PARAM -> coma PARAM												
{identificado	{RPAREN}	USEPARAM		USEPARAM -> identificador USEPARAM								USEPARAM -> RPAREN												USEPARAM ->	USEPARAM -> coma USEPARAM				USEPARAM -> zinglo_quote identificador zinglo_quote USEPARAM							
{identificado	{$,harktakon	LIB		LIB -> identificador LIB2																																		
{greater_tha	{$,harktakon	LIB2																										LIB2 -> greater_than S							LIB2 -> dot identificador greater		

Main.py

implementa un analizador sintáctico (parser) personalizado para analizar un código fuente específico mediante el uso de una tabla de análisis sintáctico predictivo no recursivo (LL(1)). El propósito principal es identificar y validar la estructura sintáctica del código fuente basándose en las reglas gramaticales predefinidas.

Sobre el archivo en C a ser analizado

Si el editor de código que se va a utilizar es **visual studio code** la ruta al archivo de C debe ir especificada de la siguiente manera:

```
def miParser():
    filename = "proyectoenv/example.c"
    code = readSourceCodeFromFile(filename)
    lexer.input(code)
```

Y si el editor a utilizar es pycharm debe ser de esta otra manera:

```
def miParser():
    filename = "../proyectoenv/example.c"
    code = readSourceCodeFromFile(filename)
    lexer.input(code)
```

Analizador Léxico

El analizador léxico se encarga de dividir el código fuente en tokens, que son las unidades básicas de significado en el lenguaje. Los tokens reconocidos incluyen palabras clave, operadores, identificadores, números, etc.

Definición de Tokens:

aumentarvar
reducirvar
libcall
NUMBER
PLUS
MINUS
TIMES
DIVIDE
LPAREN
while
RPAREN
keyword
printf
inicioBloque
finBloque
finInstruccion
asignacion
comentario
comentario_bloque
cadena
coma
int
char
float
greater_than
lesser_than
single_quote
dot
hashtoken
identificador
if
else
else_if
or
and
not
eof

Reglas de Tokens (Expresiones Regulares):

t_PLUS: +
t_MINUS: -
t_TIMES: *
t_DIVIDE: /
t_LPAREN: (
t_RPAREN:)
t_inicioBloque: {
t_finBloque: }
t_finInstruccion: ;
t_asignacion: =
t_coma: ,
t_eof: $
t_hashtoken: #

Reglas de Tokens (Funciones de Python):

t_aumentarvar: ([a-z]|[A-Z])([a-z]|[A-Z]|\d|)*++
t_reducirvar: ([a-z]|[A-Z])([a-z]|[A-Z]|\d|)*--
t_libcall: <([a-zA-Z_][a-zA-Z0-9_]*)>

t_int: (int)
t_while: (while)
t_or: ||
t_and: &&
t_not: !
t_if: (if)
t_else: (else)
t_else_if: (else if)
t_char: (char)
t_float: (float)
t_printf: (printf)
t_NUMBER: \d+
t_keyword:
(char|return|do|while|for|void)
t_identificador:
([a-z]|[A-Z]|)([a-z]|[A-Z]|\d|)*
t_newline: \n+
t_cadena: "[^"]*"
t_comentario: //.*
t_comentario_bloque: /*(.|\n)**/
t_greater_than: >
t_lesser_than: <
t_single_quote: '
t_dot: .
El analizador léxico se completa con funciones para el manejo de errores y la recuperación de tokens.

identificador
if
else
else_if
or
and
not
eof
Expresiones regulares asociadas a tokens:
t_PLUS = \+
t_MINUS = -
t_TIMES = *
t_DIVIDE = /
t_LPAREN = \(
t_RPAREN = \)
t_inicioBloque = \{
t_finBloque = \}
t_finInstruccion = \;
t_asignacion = \=
t_coma = \,
t_eof = \$
t_hashtoken = \#

Este código es un programa escrito en Python que utiliza la biblioteca ply para implementar un analizador léxico y un simple analizador sintáctico para un lenguaje de programación simplificado. El lenguaje tiene construcciones básicas como variables, funciones, declaraciones condicionales (if), bucles (while), y algunas operaciones aritméticas y lógicas.

Analizador Léxico (ply.lex)
El analizador léxico está implementado utilizando la biblioteca ply.lex. Aquí se definen los tokens y algunas reglas para reconocerlos en el código fuente. Los tokens incluyen palabras clave como int, char, float, operadores aritméticos (+, -, *, /), paréntesis, llaves, punto y coma, etc. También se definen algunas reglas para manejar comentarios de línea y de bloque.

Tabla de Símbolos (TablaSimbolos Class)
Se implementa una clase TablaSimbolos para mantener un registro de las variables en el código. Cada entrada en la tabla contiene información sobre el tipo de variable, su valor, la línea en la que se declaró y su ámbito (global, método/función o instrucción). Se proporcionan funciones para insertar, buscar, actualizar y eliminar entradas en la tabla, así como para imprimir la tabla.

Analizador Sintáctico
El analizador sintáctico es una implementación simple de un analizador descendente recursivo no recursivo (LL) que utiliza una tabla de análisis predictivo para reconocer la estructura del código fuente. La tabla de análisis predictivo se representa como una lista bidimensional llamada tabla. El analizador sintáctico utiliza una pila para realizar el análisis.

Ejecución del Parser:
La función principal es miParser(). Se lee el código fuente desde el archivo "../proyectoenv/example.c". Se realiza el análisis léxico y sintáctico, y al finalizar, se imprime la tabla de símbolos resultante.

El parser sigue el enfoque de análisis descendente predictivo no recursivo utilizando una pila stack. Se implementan mecanismos de manejo de errores y recuperación en caso de encontrar una estructura incorrecta.

```
def miParser():
            filename = "../proyectoenv/example.c"
            code = readSourceCodeFromFile(filename)
            lexer.input(code)
            tok = lexer.token()
            x = stack[-1]  # primer elemento de der a izq
            current_var_type = None
            scope = 0
            while True:
                # Manejo de tabla de simbolos
                if tok.type == "int":
                    current_var_type = "int"
                if tok.type == "char":
                    current_var_type = "char"
                if tok.type == "float":
                    current_var_type = "float"
                if tok.type == "inicioBloque":
                    scope = 1
                if tok.type == "finBloque":
                    scope = 0
                if tok.type == "identificador":
                    if current_var_type is not None:
                        tabla_simbolos.insertar(
                            tok.value, current_var_type, tok.value, tok.lineno, scope
                        )
                        current_var_type = None
                    elif current_var_type is None:
                        tabla_simbolos.insertar(
                            tok.value, tok.type, tok.value, tok.lineno, scope
                        )
```

```
            # Parseo
            if x == tok.type and x == "eof":
                print("Cadena reconocida exitosamente")
                tabla_simbolos.imprimir_tabla()  # Imprimir tabla de símbolos al final
                return  # aceptar
            else:
                if x == tok.type and x != "eof":
                    stack.pop()
                    x = stack[-1]
                    tok = lexer.token()
                if x in tokens and x != tok.type:
                    print("Error detectado")
                    expected_tokens = buscar_token_esperado(x)
                    if len(expected_tokens) == 0:
                        expected_tokens = ['hashToken', 'int', 'char', 'float', 'keyword', 'comentario', 'identificador']
                    print("Error: se esperaba uno de", expected_tokens, "pero se encontró", tok.type)
                    print("En linea:", tok.lineno)
                    tok = recuperar_modo_panico(expected_tokens, tok)
                    if tok is None or tok.type == 'eof':
                        print("No se pudo recuperar del error.")
                        return 0
                    # Reanuda el análisis después de la recuperación
                    x = stack[-1]
                    continue
                if x not in tokens:  # es no terminal
                    print("van entrar a la tabla:")
                    print(x)
                    print(tok.type)
                    celda = buscar_en_tabla(x, tok.type)
                    # Manejo de Errores y Recuperación
                    if celda is None:
                        print("Error detectado")
                        expected_tokens = buscar_token_esperado(x)
                        if len(expected_tokens) == 0:
                            expected_tokens = ['hashToken', 'int', 'char', 'float', 'keyword', 'comentario',
                                               'identificador']
                        print("Error: se esperaba uno de", expected_tokens, "pero se encontró", tok.type)
                        print("En linea:", tok.lineno)
                        print("celda: ", celda)
                        tok = recuperar_modo_panico(expected_tokens, tok)
                        if tok is None or tok.type == 'eof':
                            print("No se pudo recuperar del error.")
                            return 0
                        # Reanuda el análisis después de la recuperación
                        x = stack[-1]
                        continue
                    else:
                        stack.pop()
                        agregar_pila(celda)
                        print(stack)
                        print("/--------------------------------------------------------------------------------/")
                        x = stack[-1]
```

Ejemplo de Uso
Se proporciona un código de ejemplo en el archivo "../proyectoenv/example.c". Este código simula un código fuente con diferentes constructos del lenguaje. Al llamar a la función miParser(), se realiza el análisis y se imprime la tabla de símbolos resultante.

Cadenas de Producción
El programa utiliza cadenas de producción para definir la gramática del lenguaje. Cada cadena de producción en la tabla representa una regla gramatical y contiene información sobre cómo manejar esa regla durante el análisis sintáctico.

Manejo de Errores en el Analizador Sintáctico:
En el analizador sintáctico se implementa un manejo de errores para detectar discrepancias entre la gramática esperada y los tokens encontrados durante el análisis. El manejo de errores se lleva a cabo principalmente en la función miParser().
Puntos Clave:
Error Detectado:

- Cuando se encuentra un token que no coincide con la expectativa, se imprime un mensaje de error indicando el tipo de error, los tokens esperados y el token encontrado.
- El mensaje incluye información útil como la línea donde ocurrió el error.

Recuperación de Modo Pánico:

- Se implementa la función recuperar_modo_panico() para buscar el próximo token válido en la secuencia hasta que se encuentra un token de recuperación válido.
- La recuperación de modo pánico ayuda a sincronizar el analizador después de un error y encontrar un estado válido para continuar el análisis.

Reanudar el Análisis:

- Después de la recuperación, se reanuda el análisis con el último no terminal en la pila (x).
- Este enfoque permite continuar el análisis sintáctico incluso después de encontrar errores, mejorando la capacidad del analizador para identificar múltiples errores en una sola ejecución.

Resultados de la Ejecución:

- Cuando se completa el análisis sin errores, se imprime un mensaje indicando que la cadena se reconoció exitosamente.
- Se imprime la tabla de símbolos al final de la ejecución para proporcionar una visión completa de las variables y su estado.

Recuperación Personalizada:

- Se proporciona una recuperación personalizada para varios escenarios, intentando minimizar la interrupción del análisis y continuar con la mayor cantidad de información posible.

Subconjunto de C

Este subconjunto de C está diseñado para ilustrar el funcionamiento del sistema de análisis léxico y sintáctico implementado en Python. Se recomienda su uso como base para entender los conceptos de análisis de código fuente
Se espera que el lexer y parser asociados puedan identificar correctamente los elementos léxicos y sintácticos presentes en este código. La definición precisa de tokens y reglas gramaticales es esencial para un análisis exitoso.

Inclusión de Biblioteca Estándar

- Se realiza la inclusión de la biblioteca estándar <stdio.h>. Esto permite el uso de la función printf para la salida estándar.

Definición de Funciones:

- suma: Esta función toma dos parámetros enteros (a y b) y retorna la suma de ambos.
- imprimir_mayor: La función compara dos enteros (x e y) e imprime un mensaje indicando cuál es mayor.

Función Principal "main":

- La función main es el punto de entrada del programa. Aquí se declaran e inicializan variables de diferentes tipos (int, char, float).
- Se realiza una llamada a la función suma con parámetros específicos, almacenando el resultado en la variable resultado.
- También se invoca la función imprimir_mayor con argumentos concretos.
- Finalmente, el programa retorna 0, indicando una ejecución exitosa al sistema operativo.

```c
#include <stdio.h>

    int suma(int a, int b) {
        return a + b;
    }

    void imprimir_mayor(int x, int y) {
    if (x > y) {
        printf("El número %d es mayor que %d\n", x, y);
    } else {
        printf("El número %d es menor o igual que %d\n", x, y);
    }
    }

    int main() {
    int numero_entero = 10;
    char caracter = 'A';
    float numero_flotante = 5.5;

    // Llamando a la función suma
    int resultado = suma(numero_entero, 20);
    printf("El resultado de la suma es: %d\n", resultado);

    // Llamando a la funcion imprimir_mayor
    imprimir_mayor(8, numero_entero);

    return 0;
    }
```

parsingTable.py

Tabla de Análisis Sintáctico:

La tabla de análisis sintáctico está representada como una lista llamada tabla. Cada entrada en la lista tiene el formato [No Terminal, Terminal, Producciones], donde:

- No Terminal: Representa un símbolo no terminal en la gramática.
- Terminal: Representa un símbolo terminal o token del lenguaje.
- Producciones: Lista que describe cómo se expande o reduce el símbolo no terminal.
- Reglas Principales:

Variables Globales y Declaración de Funciones:

- int, char, float, y funciones sin tipo de retorno (keyword).

Cuerpo de Funciones:

- FBODY que puede contener comentarios, declaraciones de variables, operaciones aritméticas, estructuras de control (if, while), llamadas a funciones (printf), y más.

Manejo de Print:

- Estructura de la función printf y su contenido.

Manejo de Estructuras de Control:

- if, while, incluyendo la posibilidad de tener un bloque else.

Operaciones Aritméticas:

- Operaciones aritméticas básicas.

Manejo de Librerías:

- Inclusión de librerías en el código C.

Manejo de Retornos:

- RETURNVOID, RETURNCHAR, RETURNNUM.

Manejo de Parámetros y Uso de Funciones:

- Estructura de parámetros en funciones y cómo se usan.

Manejo de Operaciones Lógicas:

- Operaciones lógicas en condiciones if.

Auxiliares y Finales de Archivo:

- Números decimales, símbolos aritméticos, fin de archivo (eof).

```python
import nonTerminal as nt

tabla = [
    [nt.S, "hashtoken", ["hashtoken", "identificador", "lesser_than", nt.LIB]],
    # Manejo de variables globales
    # Manejo de variables globales de tipo int
    [nt.S, "int", ["int", "identificador", nt.OPENNUMFUN, nt.S]],
    [nt.S, "char", ["char", "identificador", nt.OPENCHARFUN, nt.S]],
    [nt.S, "float", ["float", "identificador", nt.OPENNUMFUN, nt.S]],
    [nt.S, "keyword", ["keyword", "identificador", nt.OPENVOID, nt.S]],
    [nt.S, "comentario", ["comentario", nt.S]],
    [
        nt.S,
        "identificador",
        ["identificador", "LPAREN", nt.USEPARAM, "RPAREN", "finInstruccion", nt.S],
    ],
    # Apoyo de funcion void
    [nt.OPENVOID, "LPAREN", ["LPAREN", nt.PARAM, nt.FBODY, nt.VOIDRETURN, "finBloque"]],
    # Manejo global scope de numericas
    [nt.OPENNUMFUN, "asignacion", ["asignacion", nt.RETURNNUM, nt.S]],
    [
        nt.OPENNUMFUN,
        "LPAREN",
        ["LPAREN", nt.PARAM, nt.FBODY, nt.NUMRETURN, "finBloque"],
    ],
    [nt.OPENNUMFUN, "finInstruccion", ["finInstruccion", nt.S]],
    # Manejo de variables
    [nt.EMPTYVAR, "asignacion", ["asignacion"]],
    [nt.EMPTYVAR, "finInstrucción", ["finInstrucción", nt.FBODY]],
    # Manejo de comentarios en el body
    [nt.FBODY, "comentario", ["comentario", nt.FBODY]],
    [nt.FBODY, "int", ["int", "identificador", nt.EMPTYVAR, nt.RETURNNUM, nt.FBODY]],
    [
        nt.FBODY,
        "float",
        ["float", "identificador", nt.EMPTYVAR, nt.RETURNNUM, nt.FBODY],
    ],
    [nt.FBODY, "char", ["char", "identificador", nt.EMPTYVAR, nt.RETURNCHAR, nt.FBODY]],
    [
        nt.FBODY,
        "identificador",
        ["identificador", "LPAREN", nt.USEPARAM, "RPAREN", "finInstruccion", nt.FBODY],
    ],
    [
        nt.FBODY,
        "if",
        [
            "if",
            "LPAREN",
            nt.CONDITIONSHANDLER,
            "RPAREN",
            "inicioBloque",
            nt.FBODY,
            "finBloque",
            nt.FBODY,
        ],
    ],
    [
        nt.FBODY,
        "while",
        [
            "while",
            "LPAREN",
            nt.CONDITIONSHANDLER,
            "RPAREN",
            "inicioBloque",
            nt.FBODY,
            "finBloque",
            nt.FBODY,
        ],
    ],
    [nt.FBODY, "else", ["else", nt.ELSEXTENTION]],
    [
        nt.FBODY,
        "printf",
        ["printf", "LPAREN", nt.PRINTCONT, "RPAREN", "finInstruccion", nt.FBODY],
    ],
    [nt.FBODY, "aumentarvar", ["aumentarvar", "finInstruccion", nt.FBODY]],
    [nt.FBODY, "reducirvar", ["reducirvar", "finInstruccion", nt.FBODY]],
    [nt.FBODY, "keyword", []],
    [nt.FBODY, "finBloque", []],
    # Manejo de estructura printf
    [nt.PRINTCONT, "cadena", [nt.THANDLER, nt.PRINTCONT]],
    [nt.PRINTCONT, "identificador", [nt.THANDLER, nt.PRINTCONT]],
    [nt.PRINTCONT, "NUMBER", [nt.THANDLER, nt.PRINTCONT]],
    [nt.PRINTCONT, "coma", ["coma", nt.PRINTCONT, nt.PRINTCONT]],
    [nt.PRINTCONT, "RPAREN", []],
    # Manejo de else
    [
        nt.ELSEXTENTION,
        "inicioBloque",
        ["inicioBloque", nt.FBODY, "finBloque", nt.FBODY],
    ],
    [nt.ELSEXTENTION, "if", [nt.FBODY]],
    # Manejo de condiciones en if
    [nt.CONDITIONSHANDLER, "identificador", [nt.THANDLER, nt.LOGICSIMBOLS]],
    [nt.CONDITIONSHANDLER, "NUMBER", [nt.THANDLER, nt.LOGICSIMBOLS]],
    [nt.CONDITIONSHANDLER, "not", ["not", nt.THANDLER, nt.LOGICSIMBOLS]],
    # Simbolos de if
    [nt.LOGICSIMBOLS, "and", ["and", nt.NOTHANDLER, nt.LOGICSIMBOLS]],
    [nt.LOGICSIMBOLS, "or", ["or", nt.NOTHANDLER, nt.LOGICSIMBOLS]],
    [nt.LOGICSIMBOLS, "lesser_than", ["lesser_than", nt.NOTHANDLER, nt.LOGICSIMBOLS]],
    [nt.LOGICSIMBOLS, "greater_than", ["greater_than", nt.NOTHANDLER, nt.LOGICSIMBOLS]],
    [nt.LOGICSIMBOLS, "not", ["not", nt.NOTHANDLER, nt.LOGICSIMBOLS]],
    [nt.LOGICSIMBOLS, "RPAREN", []],
    # Manejo de variables not
    [nt.NOTHANDLER, "not", ["not", nt.THANDLER]],
    [nt.NOTHANDLER, "identificador", [nt.THANDLER]],
    [nt.NOTHANDLER, "NUMBER", [nt.THANDLER]],
    # Manejo de retorno scope global
    [nt.OPENCHARFUN, "asignacion", ["asignacion", nt.RETURNCHAR, nt.S]],
    [nt.OPENCHARFUN, "LPAREN", ["LPAREN", nt.PARAM, nt.CHARFRETURN, "finBloque"]],
    [nt.OPENNUMFUN, "finInstruccion", ["finInstruccion", nt.S]],
    # Manejo de retornos de tipo char
    [nt.VOIDRETURN, "keyword", ["keyword", nt.RETURNVOID]],
    [nt.VOIDRETURN, "finBloque", []],
    [nt.CHARFRETURN, "keyword", ["keyword", nt.RETURNCHAR]],
    # Manejo de retorno de tipo int
    [nt.NUMRETURN, "keyword", ["keyword", nt.RETURNNUM]],
    # Manejo de retorno de void
    [
        nt.RETURNVOID,
        "single_quote",
        ["single_quote", "identificador", "single_quote", "finInstruccion"],
    ],
    [nt.RETURNVOID, "identificador", ["identificador", nt.ARITMETIC]],
    [nt.RETURNVOID, "LPAREN", ["LPAREN", nt.THANDLER, nt.ARITMETIC]],
    [nt.RETURNVOID, "NUMBER", [nt.FLOATN, nt.ARITMETIC]],
    [nt.RETURNVOID, "finBloque", []],
    # Manejo retorno para funciones char
    [
        nt.RETURNCHAR,
        "single_quote",
        ["single_quote", "identificador", "single_quote", "finInstruccion"],
    ],
    [nt.RETURNCHAR, "identificador", ["identificador", "finInstruccion"]],
    [
        nt.RETURNCHAR,
        "LPAREN",
        ["LPAREN", "char", "RPAREN", "identificador", "finInstruccion"],
    ],
    # Manejo de retornos para funciones int
    [nt.RETURNNUM, "identificador", ["identificador", nt.ARITMETIC]],
    [nt.RETURNNUM, "LPAREN", ["LPAREN", nt.THANDLER, nt.ARITMETIC]],
    [nt.RETURNNUM, "NUMBER", [nt.FLOATN, nt.ARITMETIC]],
    # Manejo de operaciones
    [nt.ARITMETIC, "LPAREN", ["LPAREN", nt.THANDLER, nt.ARITMETIC]],
    [nt.ARITMETIC, "RPAREN", ["RPAREN", nt.ARITMETIC]],
    [nt.ARITMETIC, "PLUS", ["PLUS", nt.GHANDLER, nt.ARITMETIC]],
    [nt.ARITMETIC, "MINUS", ["MINUS", nt.GHANDLER, nt.ARITMETIC]],
    [nt.ARITMETIC, "TIMES", ["TIMES", nt.GHANDLER, nt.ARITMETIC]],
    [nt.ARITMETIC, "DIVIDE", ["DIVIDE", nt.GHANDLER, nt.ARITMETIC]],
    [nt.ARITMETIC, "coma", ["coma", nt.THANDLER, nt.ARITMETIC]],
    [nt.ARITMETIC, "finInstruccion", ["finInstruccion"]],
    # Manejador de parentesis en operaciones aritmeticas
    [nt.GHANDLER, "identificador", ["identificador"]],
    [nt.GHANDLER, "LPAREN", ["LPAREN", nt.THANDLER]],
    [nt.GHANDLER, "NUMBER", [nt.FLOATN]],
    # Manejo de tipos de variables
    [nt.THANDLER, "identificador", ["identificador"]],
    [nt.THANDLER, "NUMBER", [nt.FLOATN]],
    [nt.THANDLER, "cadena", ["cadena"]],
    # Manejo de cantidad de parametros en funciones
    [nt.PARAM, "int", ["int", "identificador", nt.PARAM]],
    [nt.PARAM, "char", ["char", "identificador", nt.PARAM]],
    [nt.PARAM, "float", ["float", "identificador", nt.PARAM]],
    [nt.PARAM, "RPAREN", ["RPAREN", "inicioBloque"]],
    [nt.PARAM, "coma", ["coma", nt.PARAM]],
    # Manejo uso de parametros en funciones
    [nt.USEPARAM, "identificador", ["identificador", nt.USEPARAM]],
    [nt.USEPARAM, "NUMBER", [nt.FLOATN, nt.USEPARAM]],
    [
        nt.USEPARAM,
        "single_quote",
        ["single_quote", "identificador", "single_quote", nt.USEPARAM],
    ],
    [nt.USEPARAM, "coma", ["coma", nt.USEPARAM]],
    [nt.USEPARAM, "RPAREN", []],
    # Apoyo para los prototipos de librias de C
    [nt.LIB, "identificador", ["identificador", nt.LIB2]],
    [nt.LIB2, "greater_than", ["greater_than", nt.S]],
    [nt.LIB2, "dot", ["dot", "identificador", "greater_than", nt.S]],
    # Auxiliares
    [nt.FLOATN, "NUMBER", ["NUMBER", nt.DECIMAL]],
    [nt.DECIMAL, "dot", ["dot", "NUMBER"]],
    [nt.DECIMAL, "MINUS", []],
    [nt.DECIMAL, "PLUS", []],
    [nt.DECIMAL, "TIMES", []],
    [nt.DECIMAL, "DIVIDE", []],
    [nt.DECIMAL, "RPAREN", []],
    [nt.DECIMAL, "finInstruccion", []],
    [nt.DECIMAL, "coma", []],
    # Cadena de fin de archivo
    [nt.S, "eof", ["eof"]],
]
```

SymbolTable.py

La clase TablaSimbolos se encarga de gestionar una tabla de símbolos, que es utilizada para almacenar información sobre los identificadores en un programa. Cada entrada en la tabla representa un símbolo y contiene detalles como el tipo, valor, número de línea y ámbito.

Método __init__(self):

- Inicializa una nueva instancia de la clase.

Atributos:

- simbolos: Diccionario para almacenar símbolos con sus propiedades.

Método insertar(self, identificador, tipo, valor, linea, ambito):

- Inserta un nuevo símbolo en la tabla.

Parámetros:

- identificador: Nombre del símbolo.
- tipo: Tipo de dato del símbolo.
- valor: Valor asignado al símbolo.
- linea: Número de línea en el que se encuentra el símbolo.
- ambito: Nivel de ámbito del símbolo (0: global, 1: método/función, 2: instrucción).

Método buscar(self, identificador):

- Busca un símbolo en la tabla.

Parámetros:

- identificador: Nombre del símbolo a buscar.

Retorna:

- Información del símbolo si se encuentra, None si no existe.

Método actualizar(self, identificador, valor):

- Actualiza el valor de un símbolo existente en la tabla.

Parámetros:

- identificador: Nombre del símbolo a actualizar.
- valor: Nuevo valor asignado al símbolo.

Método eliminar(self, identificador):

- Elimina un símbolo de la tabla.

Parámetros:

- identificador: Nombre del símbolo a eliminar.

Método imprimir_tabla(self):

- Imprime la tabla de símbolos en un formato tabular en la consola.
- Muestra información detallada sobre identificadores, tipos, valores, líneas y ámbitos.

```python
import shutil


# Clase para la tabla de simbolos
class TablaSimbolos:
    def __init__(self):
        self.simbolos = {}

    def insertar(self, identificador, tipo, valor, linea, ambito):
        if identificador not in self.simbolos:
            self.simbolos[identificador] = {
                "tipo": tipo,
                "valor": valor,
                "linea": linea,
                "ambito": ambito,
            }

    def buscar(self, identificador):
        return self.simbolos.get(identificador, None)

    def actualizar(self, identificador, valor):
        if identificador in self.simbolos:
            self.simbolos[identificador]["valor"] = valor

    def eliminar(self, identificador):
        if identificador in self.simbolos:
            del self.simbolos[identificador]

    def imprimir_tabla(self):
        terminal_width = shutil.get_terminal_size().columns
        print("\nTabla de Símbolos:")

        # Distribuir el ancho uniformemente entre las columnas
        column_width = terminal_width // 5

        header = "{:<{}} | {:<{}} | {:<{}} | {:<{}} | {:<{}}".format(
            "Identificador",
            column_width,
            "Tipo",
            column_width,
            "Valor",
            column_width,
            "Línea",
            column_width,
            "Ámbito",
            column_width,
        )
        separator = "-" * (column_width * 6)

        print(header)
        print(separator)

        for identificador, info in self.simbolos.items():
            tipo = info["tipo"]
            valor = info["valor"]
            linea = info["linea"]
            ambito_nivel = info["ambito"]
            ambito = (
                "global"
                if ambito_nivel == 0
                else ("metodo/funcion" if ambito_nivel == 1 else "instruccion")
            )

            row = "{:<{}} | {:<{}} | {:<{}} | {:<{}} | {:<{}}".format(
                identificador,
                column_width,
                tipo,
                column_width,
                valor,
                column_width,
                linea,
                column_width,
                ambito,
                column_width,
            )
            print(row)
```

Ejecución del Programa:

El inicio del programa se lleva a cabo mediante el archivo "main.py". En este contexto, se importa la clase "SymbolTable" para facilitar la gestión de símbolos, y se procede a realizar el análisis léxico del código fuente. Los resultados de este análisis se presentan de manera organizada en una tabla.

Ejecución con Archivo Ejecutable Directo:

- Paso 1: En el administrador de archivos, navegue hasta la ubicación donde se encuentra almacenado el archivo ejecutable principal, denominado "main.exe" (ubicado en la carpeta: AnalizadorLexicoGrafico).

- Paso 2: Ejecute el programa haciendo doble clic en el archivo "main.exe". En caso de que aparezca una ventana emergente, proceda a hacer clic en el botón "Run" o "Ejecutar".

- Paso 3: Para finalizar la ejecución del programa, presione la tecla "Enter". Esto culminará el proceso de manera ordenada.