**UNIVERSIDAD DE GUADALAJARA**

**CENTRO UNIVERSITARIO DE TONALÁ**

**Logotipo

Descripción generada automáticamente**

**Materia: Traductores de Lenguaje.**

**Tema: Análisis sintáctico de variables y constantes**

**Integrantes:**

// Celestino Martinez Cristopher

// Ramirez Rivas Luis Fernando

// Rivas Tejeda Hector Manuel

**Fecha:** 03 de febrero del 2025

**Lenguaje asignado C**

#include <stdio.h>

// Celestino Martinez Cristopher

// Ramirez Rivas Luis Fernando

// Rivas Tejeda Hector Manuel

int main() {

    // Imprimir nombres completos en orden alfabético

    printf("TRADUCTORES DE LENGUAJES:\n");

    printf("Actividad 2\n");

    int a = 10;  // Asignación de valor a 'a'

    int x = 2;

    int b = 3;

    int y = 3;

    int z = 2;

    int w = 2;

    //Expresiones válidas:

    x + y \* 5

    (a - b) / c

    10 > 5 && 3 <= 4

    (x + y) \* (z - w)

    //Expresiones inválidas:

    x + \* y  // Operador sin término

    10 >  // Comparación sin expresión aritmética

    (a - b  // Paréntesis desbalanceados

    5 / (2 + )  // Paréntesis desbalanceados

    if (a > b) { // Comparación de 'a' y 'b'

        printf("%s\n", saludo); // Imprimir el saludo

    }

    // Imprimir nombres completos en orden alfabético al final

    printf("\nIntegrantes del equipo:\n");

    printf("1. Celestino Martinez Cristopher\n");

    printf("2. Ramirez Rivas Luis Fernando\n");

    printf("3. Rivas Tejeda Hector Manuel\n");

    return 0;

}

   // N�mero hexadecimal mal formado (falta '0x')

    int hex = x12AB;

    //Comentario con una sola barra inclinada (inv�lido)

    Esto no es un comentario v�lido

    return 0;

}

**Declaración de Variables:**

Se definen varias variables enteras:

c

int a = 10; // Asignación de valor a 'a'

int x = 2;

int b = 3;

int y = 3;

int z = 2;

int w = 2;

Aquí, las variables a, b, x, y, z, y w son inicializadas con valores enteros. Estas variables se utilizarán en las expresiones más adelante en el código.

**3. Expresiones Válidas e Inválidas:**

Luego, se presentan expresiones aritméticas y lógicas. Estas expresiones son ejemplos que ilustran tanto expresiones válidas como inválidas.

**Expresiones válidas:**

c

x + y \* 5

(a - b) / c

10 > 5 && 3 <= 4

(x + y) \* (z - w)

* **x + y \* 5**: Suma de x y y \* 5. Primero se realiza la multiplicación y luego la suma.
* **(a - b) / c**: Restamos a y b, y luego el resultado se divide entre c.
* **10 > 5 && 3 <= 4**: Evaluación de una expresión lógica combinando comparaciones. Se evalúa si 10 es mayor que 5 y si 3 es menor o igual a 4.
* **(x + y) \* (z - w)**: Primero se suma x y y, y luego se multiplica por la resta de z y w.

**Expresiones inválidas:**

c

x + \* y // Operador sin término

10 > // Comparación sin expresión aritmética

(a - b // Paréntesis desbalanceados

5 / (2 + ) // Paréntesis desbalanceados

* **x + \* y**: Esta es inválida porque el operador \* está fuera de lugar y no hay un valor después de él, lo que genera un error de sintaxis.
* **10 >**: La comparación está incompleta, falta una expresión después del operador >.
* **(a - b**: Aquí falta el paréntesis de cierre, lo que genera paréntesis desbalanceados.
* **5 / (2 + )**: El operador de adición + no tiene un valor después de él, lo que también genera paréntesis desbalanceados.

**Main**

from lexer import lexer

def main():

    # Leer el archivo de prueba

    with open('C:/Users/user/Desktop/traductores/Lex-main/tests/example.c', 'r') as f:

        code = f.read()

    # Alimentar el lexer

    lexer.input(code)

    # Mostrar tokens generados

    print("Tokens encontrados:")

    while True:

        tok = lexer.token()

        if not tok:

            break

        print(tok)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Explicación del Código**

1. **Importación del Lexer**:

python

from lexer import lexer

Aquí se importa un objeto lexer desde un módulo llamado lexer. Este lexer es probablemente una instancia de un analizador léxico generado por una herramienta como **PLY** (una implementación de Lex y Yacc en Python).

1. **Función main**:  
   La función main es el punto de entrada del programa. Aquí se realiza lo siguiente:
   * **Lectura del archivo de código fuente**:

python

with open('C:/Users/user/Desktop/traductores/Lex-main/tests/example.c', 'r') as f:

code = f.read()

Se abre el archivo example.c en modo lectura ('r') y se lee todo su contenido en la variable code.

* + **Alimentar el lexer**:

python

lexer.input(code)

El contenido del archivo (code) se pasa al lexer para que lo procese.

* + **Mostrar los tokens generados**:

python

print("Tokens encontrados:")

while True:

tok = lexer.token()

if not tok:

break

print(tok)

Se entra en un bucle infinito donde se llama repetidamente a lexer.token() para obtener los tokens generados por el lexer. Cuando no hay más tokens (tok es None), el bucle se detiene. Cada token se imprime en la consola.

1. **Ejecución del script**:

python

if \_\_name\_\_ == '\_\_main\_\_':

main()

Este bloque asegura que la función main se ejecute solo cuando el script se ejecuta directamente (no cuando se importa como módulo).

**¿Qué hace este código?**

Este script toma un archivo de código fuente en C (example.c) y lo procesa con un lexer para extraer los tokens. Los tokens son las unidades básicas del lenguaje, como palabras clave, identificadores, números, operadores, etc. El lexer identifica estos tokens según las reglas definidas en su implementación.

**Lexer**

import ply.lex as lex

import ply.yacc as yacc

# Definición de los tokens

tokens = (

    'IDENTIFICADOR', 'NUMERO', 'MAS', 'MENOS', 'POR', 'DIVIDIDO',

    'PARIZQ', 'PARDER', 'IGUAL', 'DIFERENTE', 'MAYOR', 'MENOR',

    'MAYORIGUAL', 'MENORIGUAL', 'AND', 'OR', 'COMA'

)

# Expresiones regulares para los tokens

t\_MAS = r'\+'

t\_MENOS = r'-'

t\_POR = r'\\*'

t\_DIVIDIDO = r'/'

t\_IGUAL = r'=='

t\_DIFERENTE = r'!='

t\_MAYOR = r'>'

t\_MENOR = r'<'

t\_MAYORIGUAL = r'>='

t\_MENORIGUAL = r'<='

t\_AND = r'&&'

t\_OR = r'\|\|'

t\_PARIZQ = r'\('

t\_PARDER = r'\)'

t\_COMA = r','

# Identificadores (variables)

t\_IDENTIFICADOR = r'[a-zA-Z\_][a-zA-Z0-9\_]\*'

# Números (enteros)

t\_NUMERO = r'\d+'

# Ignorar espacios en blanco y saltos de línea

t\_ignore = ' \t\n'

# Manejo de errores

def t\_error(t):

    print(f"Error léxico: {t.value} en la línea {t.lineno}, columna {t.lexpos}")

    t.lexer.skip(1)

# Construir el lexer

lexer = lex.lex()

# Definir la precedencia de los operadores (para el análisis correcto de la aritmética)

precedence = (

    ('left', 'MAYOR', 'MENOR', 'IGUAL', 'DIFERENTE', 'MAYORIGUAL', 'MENORIGUAL'),

    ('left', 'AND', 'OR'),

    ('left', 'MAS', 'MENOS'),

    ('left', 'POR', 'DIVIDIDO'),

)

# Regla de la gramática para las expresiones aritméticas y lógicas

def p\_expresion(p):

    '''expresion : expresion\_aritmetica

                 | expresion\_logica'''

    p[0] = p[1]

def p\_expresion\_aritmetica(p):

    '''expresion\_aritmetica : termino

                            | expresion\_aritmetica MAS termino

                            | expresion\_aritmetica MENOS termino'''

    if len(p) == 2:

        p[0] = p[1]

    elif p[2] == '+':

        p[0] = p[1] + p[3]

    elif p[2] == '-':

        p[0] = p[1] - p[3]

def p\_termino(p):

    '''termino : factor

              | termino POR factor

              | termino DIVIDIDO factor'''

    if len(p) == 2:

        p[0] = p[1]

    elif p[2] == '\*':

        p[0] = p[1] \* p[3]

    elif p[2] == '/':

        p[0] = p[1] / p[3]

def p\_factor(p):

    '''factor : NUMERO

              | IDENTIFICADOR

              | PARIZQ expresion\_aritmetica PARDER'''

    if len(p) == 2:

        p[0] = int(p[1]) if isinstance(p[1], str) and p[1].isdigit() else p[1]

    elif len(p) == 4:

        p[0] = p[2]

def p\_expresion\_logica(p):

    '''expresion\_logica : termino\_logico

                       | expresion\_logica AND termino\_logico

                       | expresion\_logica OR termino\_logico'''

    if len(p) == 2:

        p[0] = p[1]

    elif p[2] == '&&':

        p[0] = p[1] and p[3]

    elif p[2] == '||':

        p[0] = p[1] or p[3]

def p\_termino\_logico(p):

    '''termino\_logico : expresion\_aritmetica IGUAL expresion\_aritmetica

                     | expresion\_aritmetica DIFERENTE expresion\_aritmetica

                     | expresion\_aritmetica MAYOR expresion\_aritmetica

                     | expresion\_aritmetica MENOR expresion\_aritmetica

                     | expresion\_aritmetica MAYORIGUAL expresion\_aritmetica

                     | expresion\_aritmetica MENORIGUAL expresion\_aritmetica'''

    if p[2] == '==':

        p[0] = p[1] == p[3]

    elif p[2] == '!=':

        p[0] = p[1] != p[3]

    elif p[2] == '>':

        p[0] = p[1] > p[3]

    elif p[2] == '<':

        p[0] = p[1] < p[3]

    elif p[2] == '>=':

        p[0] = p[1] >= p[3]

    elif p[2] == '<=':

        p[0] = p[1] <= p[3]

# Manejo de errores sintácticos

def p\_error(p):

    if p:

        print(f"Error sintáctico: '{p.value}' en la línea {p.lineno}")

    else:

        print("Error sintáctico en la entrada.")

# Construir el parser

parser = yacc.yacc()

# Función para analizar las expresiones y manejar errores

def analizar\_expresion(expresion):

    try:

        lexer.input(expresion)

        parser.parse(expresion)

        return "Expresión inválida"

    except:

        return "Error en la expresión"

# Ignorar espacios en blanco y saltos de línea

t\_ignore = ' \t\n'

# Manejo de errores

def t\_error(t):

    print(f"Error léxico: {t.value} en la línea {t.lineno}, columna {t.lexpos}")

    t.lexer.skip(1)

# Construir el lexer

lexer = lex.lex()

# Definir la precedencia de los operadores (para el análisis correcto de la aritmética)

precedence = (

    ('left', 'MAYOR', 'MENOR', 'IGUAL', 'DIFERENTE', 'MAYORIGUAL', 'MENORIGUAL'),

    ('left', 'AND', 'OR'),

    ('left', 'MAS', 'MENOS'),

    ('left', 'POR', 'DIVIDIDO'),

)

# Regla de la gramática para las expresiones aritméticas y lógicas

def p\_expresion(p):

    '''expresion : expresion\_aritmetica

                 | expresion\_logica'''

    p[0] = p[1]

def p\_expresion\_aritmetica(p):

    '''expresion\_aritmetica : termino

                            | expresion\_aritmetica MAS termino

                            | expresion\_aritmetica MENOS termino'''

    if len(p) == 2:

        p[0] = p[1]

    elif p[2] == '+':

        p[0] = p[1] + p[3]

    elif p[2] == '-':

        p[0] = p[1] - p[3]

def p\_termino(p):

    '''termino : factor

              | termino POR factor

              | termino DIVIDIDO factor'''

    if len(p) == 2:

        p[0] = p[1]

    elif p[2] == '\*':

        p[0] = p[1] \* p[3]

    elif p[2] == '/':

        p[0] = p[1] / p[3]

def p\_factor(p):

    '''factor : NUMERO

              | IDENTIFICADOR

              | PARIZQ expresion\_aritmetica PARDER'''

    if len(p) == 2:

        p[0] = int(p[1]) if isinstance(p[1], str) and p[1].isdigit() else p[1]

    elif len(p) == 4:

        p[0] = p[2]

def p\_expresion\_logica(p):

    '''expresion\_logica : termino\_logico

                       | expresion\_logica AND termino\_logico

                       | expresion\_logica OR termino\_logico'''

    if len(p) == 2:

        p[0] = p[1]

    elif p[2] == '&&':

        p[0] = p[1] and p[3]

    elif p[2] == '||':

        p[0] = p[1] or p[3]

def p\_termino\_logico(p):

    '''termino\_logico : expresion\_aritmetica IGUAL expresion\_aritmetica

                     | expresion\_aritmetica DIFERENTE expresion\_aritmetica

                     | expresion\_aritmetica MAYOR expresion\_aritmetica

                     | expresion\_aritmetica MENOR expresion\_aritmetica

                     | expresion\_aritmetica MAYORIGUAL expresion\_aritmetica

                     | expresion\_aritmetica MENORIGUAL expresion\_aritmetica'''

    if p[2] == '==':

        p[0] = p[1] == p[3]

    elif p[2] == '!=':

        p[0] = p[1] != p[3]

    elif p[2] == '>':

        p[0] = p[1] > p[3]

    elif p[2] == '<':

        p[0] = p[1] < p[3]

    elif p[2] == '>=':

        p[0] = p[1] >= p[3]

    elif p[2] == '<=':

        p[0] = p[1] <= p[3]

# Manejo de errores sintácticos

def p\_error(p):

    if p:

        print(f"Error sintáctico: '{p.value}' en la línea {p.lineno}")

    else:

        print("Error sintáctico en la entrada.")

# Construir el parser

parser = yacc.yacc()

# Función para analizar las expresiones y manejar errores

def analizar\_expresion(expresion):

    try:

        lexer.input(expresion)

        parser.parse(expresion)

        return "Expresión inválida"

    except:

        return "Error en la expresión"

Este código implementa un analizador léxico y sintáctico utilizando la librería ply en Python para analizar expresiones matemáticas y lógicas simples. Vamos a desglosar el código paso a paso:

**1. Tokens y Expresiones Regulares:**

* **Tokens**: Son los elementos básicos que el lexer (analizador léxico) reconocerá en el código fuente. Estos incluyen operadores aritméticos (MAS, MENOS, POR, DIVIDIDO), operadores lógicos (AND, OR), comparaciones (IGUAL, DIFERENTE, etc.), paréntesis y números, entre otros.

Los tokens se definen en la tupla tokens y se asignan expresiones regulares para que el lexer los identifique. Por ejemplo:

* + MAS = r'\+' reconoce el operador de suma (+).
  + NUMERO = r'\d+' reconoce un número entero.

**2. Reglas del Lexer:**

* **Ignorar espacios en blanco y saltos de línea**: El lexer ignora los espacios en blanco y saltos de línea (definido en t\_ignore).
* **Manejo de errores**: Si el lexer encuentra algo que no coincide con las expresiones regulares, llama a la función t\_error y muestra un mensaje de error indicando el valor problemático.

**3. Precedencia de Operadores:**

* Para evitar ambigüedades en las expresiones, se define la **precedencia** de los operadores aritméticos y lógicos. Esto asegura que, por ejemplo, las multiplicaciones y divisiones se realicen antes que las sumas y restas.

La precedencia se define en la tupla precedence, donde se ordenan los operadores por prioridad (de mayor a menor).

**4. Reglas de la Gramática (Parser):**

* El parser se encarga de interpretar las expresiones sintácticamente, construyendo el árbol de análisis.
  + **expresion**: Esta es la regla principal que puede ser una expresión aritmética o lógica.
  + **expresion\_aritmetica**: Representa las operaciones aritméticas (suma, resta, multiplicación, etc.).
  + **expresion\_logica**: Representa las operaciones lógicas (AND, OR, etc.).
  + **termino y factor**: Son subreglas que definen la forma de los términos y factores de las expresiones aritméticas y lógicas.
  + **termino\_logico**: Define las comparaciones entre expresiones aritméticas, como ==, !=, >, <, etc.

Las funciones que comienzan con p\_ son las reglas de producción que definen la gramática. Por ejemplo:

* + p\_expresion\_aritmetica maneja las expresiones como expresion\_aritmetica + termino.
  + p\_termino maneja la multiplicación y división.

El valor de cada expresión se guarda en p[0], que es el valor de la expresión calculada.

**5. Manejo de Errores Sintácticos:**

* **p\_error**: Si el parser encuentra un error sintáctico, se imprime un mensaje con el token que causó el problema.

**6. Análisis de la Expresión:**

* La función analizar\_expresion toma una cadena de texto (expresión) como entrada. Primero, inicializa el lexer con la expresión, luego pasa esa entrada al parser para que la analice.
* Si se encuentra un error léxico o sintáctico, se captura y se muestra un mensaje.

**Salida**//Expresiones inválidas:

Error sintáctico: '\*' en la línea 1

x + \* y // Expresión inválida

Error sintáctico en la entrada.

10 > // Expresión inválida

(a - b // Error en la expresión

Error sintáctico: ')' en la línea 1

5 / (2 + ) // Expresión inválida

PS C:\Users\user\Desktop\traductores\Hoy\Tokkens-y-Exp.-Reguares\Traductores\A5 Análisis sintáctico de expresiones aritméticas y lógicas> **Github**

[cristopherstrange25/Tokkens-y-Exp.-Reguares: Actividad 2: Definición de tokens y expresiones regulares.](https://github.com/cristopherstrange25/Tokkens-y-Exp.-Reguares/tree/main)

**Video**

<https://youtu.be/Zw6Ms9GDgak?si=nfkU-OzzXlQmzKFO>