**UNIVERSIDAD DE GUADALAJARA**

**CENTRO UNIVERSITARIO DE TONALÁ**

**Logotipo

Descripción generada automáticamente**

**Materia: Traductores de Lenguaje.**

**Tema: Análisis sintáctico de estructuras de control**

**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");

 //   codigo\_valido = """

if (x > 0) { x = x + 1; } else { x = x - 1; }

while (y < 10) { y = y + 1; }

for (i = 0; i < 10; i = i + 1) { suma = suma + i; }

// Casos inválidos

    if (x > 0 { x = x + 1; }  // Paréntesis desbalanceados

    while x < 10) { y = y + 1; }  // Paréntesis faltantes

    for i = 0; i < 10; i = i + 1) { suma = suma + i; }  // Paréntesis faltantes

    // 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;

}

Este código en C tiene como objetivo imprimir ejemplos de estructuras de control válidas e inválidas en un lenguaje de programación, y al final, imprimir los nombres de los integrantes del equipo.

1. **Impresión de encabezado**: Al inicio, el código imprime el título "TRADUCTORES DE LENGUAJES" y el nombre de la actividad "Actividad 2".
2. **Ejemplos de estructuras de control válidas**:
   * Se muestran tres ejemplos de estructuras de control que están correctamente formadas:
     + Un condicional if con su bloque else.
     + Un bucle while con una condición válida.
     + Un bucle for con inicialización, condición y actualización válidas.
3. **Ejemplos de estructuras de control inválidas**:
   * Luego, se muestran tres ejemplos de estructuras de control con errores sintácticos. Los errores se comentan junto a las líneas de código:
     + El if tiene paréntesis desbalanceados.
     + El while tiene paréntesis faltantes.
     + El for también tiene paréntesis faltantes.
4. **Impresión de los nombres del equipo**: Después de mostrar los ejemplos, el código imprime los nombres de los tres miembros del equipo en orden alfabético.

**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**:

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**:

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**:

lexer.input(code)

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

* + **Mostrar los tokens generados**:

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

Copy

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

# Definir los tokens

tokens = (

    'IF', 'ELSE', 'WHILE', 'FOR', 'IDENTIFICADOR', 'NUMERO', 'MAS', 'MENOS', 'PARIZQ', 'PARDER', 'LLAVEIZQ', 'LLAVEDER', 'PUNTOYCOMA',

)

# Expresiones regulares de los tokens

t\_IF = r'if'

t\_ELSE = r'else'

t\_WHILE = r'while'

t\_FOR = r'for'

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

t\_NUMERO = r'\d+'

t\_MAS = r'\+'

t\_MENOS = r'-'

t\_PARIZQ = r'\('

t\_PARDER = r'\)'

t\_LLAVEIZQ = r'\{'

t\_LLAVEDER = r'\}'

t\_PUNTOYCOMA = r';'

# Ignorar espacios en blanco

t\_ignore = ' \t'

# Manejo de errores léxicos

def t\_error(t):

    print(f"Error de estructura de control: {t.value} en la línea {t.lineno}")

    t.lexer.skip(1)

# Construir el lexer

lexer = lex.lex()

# Definir las reglas del parser

def p\_estructura\_control(p):

    '''estructura\_control : sentencia\_if

                          | sentencia\_while

                          | sentencia\_for'''

    p[0] = p[1]

def p\_sentencia\_if(p):

    '''sentencia\_if : IF PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER

                    | IF PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER ELSE LLAVEIZQ bloque LLAVEDER'''

    if len(p) == 8:

        p[0] = ('if', p[3], p[6])

    else:

        p[0] = ('if', p[3], p[6], p[10])

def p\_sentencia\_while(p):

    '''sentencia\_while : WHILE PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER'''

    p[0] = ('while', p[3], p[6])

def p\_sentencia\_for(p):

    '''sentencia\_for : FOR PARIZQ asignacion PUNTOYCOMA expresion PUNTOYCOMA expresion PARDER LLAVEIZQ bloque LLAVEDER'''

    p[0] = ('for', p[3], p[5], p[7], p[10])

def p\_bloque(p):

    '''bloque : sentencia

             | bloque sentencia'''

    if len(p) == 2:

        p[0] = [p[1]]

    else:

        p[0] = p[1] + [p[2]]

def p\_asignacion(p):

    '''asignacion : IDENTIFICADOR MAS NUMERO

                  | IDENTIFICADOR MENOS NUMERO'''

    p[0] = (p[1], p[2], p[3])

def p\_expresion(p):

    '''expresion : IDENTIFICADOR

                 | NUMERO'''

    p[0] = p[1]

def p\_sentencia(p):

    '''sentencia : asignacion PUNTOYCOMA'''

    p[0] = p[1]

def p\_error(p):

    print(f"Error de estructura de control: '{p.value}' en la línea {p.lineno}" if p else "Error de estructura de control en la entrada.")

# Construir el parser

parser = yacc.yacc()

# Función para analizar código

def analizar\_codigo(codigo):

    parser.parse(codigo)

tokens += list(reserved.values())

# Expresiones regulares para tokens simples

t\_PLUS = r'\+'

t\_MINUS = r'-'

t\_TIMES = r'\\*'

t\_DIVIDE = r'/'

t\_MODULO = r'%'

t\_ASSIGN = r'='

t\_LT = r'<'

t\_GT = r'>'

t\_LE = r'<='

t\_GE = r'>='

t\_EQ = r'=='

t\_NE = r'!='

t\_AND = r'&&'

t\_OR = r'\|\|'

t\_NOT = r'!'

t\_LPAREN = r'\('

t\_RPAREN = r'\)'

t\_LBRACE = r'\{'

t\_RBRACE = r'\}'

t\_LBRACKET = r'\['

t\_RBRACKET = r'\]'

t\_COMMA = r','

t\_SEMICOLON = r';'

t\_DOT = r'\.'

# Manejo de caracteres no reconocidos

def t\_CHARACTER\_UNKNOWN(t):

    r'[^\w\s\+\-\\*/\(\)\[\]\{\},;=<>!\&\|\%\']'

    print(f"Error léxico: Carácter no reconocido '{t.value[0]}' en línea {t.lexer.lineno}")

    t.lexer.skip(1)  # Saltar el carácter no reconocido

# Manejo de identificadores mal formados

def t\_IDENTIFIER(t):

    r'[a-zA-Z\_][a-zA-Z0-9\_]\*'

    if t.value[0].isdigit():

        print(f"Error léxico: Identificador mal formado '{t.value}' en línea {t.lexer.lineno}")

        t.value = ""  # Limpiar el valor del token

    else:

        t.type = reserved.get(t.value, 'IDENTIFIER')

    return t

# Manejo de números flotantes incorrectos

def t\_FLOAT(t):

    r'\d+\.\.\d+'

    print(f"Error léxico: Número flotante incorrecto '{t.value}' en línea {t.lexer.lineno}")

    t.value = ""  # Limpiar el valor del token

    return t

# Manejo de cadenas mal formadas

def t\_STRING(t):

    r'"([^"\\]|\\.)\*"'

    if t.value.count('"') % 2 == 1:  # Verifica si hay una comilla de cierre

        print(f"Error léxico: Cadena mal formada en línea {t.lexer.lineno}")

        t.value = ""  # Limpiar el valor del token

    else:

        t.value = t.value[1:-1]  # Remover comillas

    return t

# Manejo de comentarios mal cerrados

def t\_COMMENT\_SHARP(t):

    r'\#.\*'

    pass

def t\_COMMENT(t):

    r'//.\*|/\\*[\s\S]\*?\\*/'

    if t.value.startswith('/\*') and not t.value.endswith('\*/'):

        print(f"Error léxico: Comentario mal cerrado en línea {t.lexer.lineno}")

    pass

# Manejo de caracteres especiales no válidos (@)

def t\_INVALID\_CHAR(t):

    r'@'

    print(f"Error léxico: Uso de un carácter especial no válido '@' en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de números con letras en medio

def t\_INVALID\_NUMBER(t):

    r'\d+[a-zA-Z]\d+'

    print(f"Error léxico: Uso de un número con una letra en medio '{t.value}' en línea {t.lexer.lineno}")

    t.value = ""  # Limpiar el valor del token

    return t

# Manejo de operadores desconocidos ($)

def t\_UNKNOWN\_OPERATOR(t):

    r'\$'

    print(f"Error léxico: Operador desconocido '$' en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de punto y coma faltante

def t\_SEMICOLON\_MISSING(t):

    r'(?<=\w)\s\*$'

    print(f"Error léxico: Falta punto y coma al final de la declaración en línea {t.lexer.lineno}")

    return t

# Manejo de uso incorrecto de secuencias de escape

def t\_INVALID\_ESCAPE\_SEQUENCE(t):

    r'\\[^abfnrtv\\\'"]'

    print(f"Error léxico: Uso incorrecto de secuencias de escape en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de comillas simples en lugar de dobles en cadenas

def t\_SINGLE\_QUOTE\_IN\_STRING(t):

    r"'([^'\\]|\\.)\*'"

    print(f"Error léxico: Uso de comillas simples en lugar de dobles en cadena de texto en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de números hexadecimales mal formados

def t\_INVALID\_HEX(t):

    r'\b[^0x0-9A-Fa-f]\d+[A-Fa-f0-9]{2,}\b'

    print(f"Error léxico: Número hexadecimal mal formado '{t.value}' en línea {t.lexer.lineno}")

    t.value = ""  # Limpiar el valor del token

    return t

# Manejo de comentarios con una sola barra (inválido)

def t\_INVALID\_SINGLE\_SLASH\_COMMENT(t):

    r'[^/]/.\*'

    print(f"Error léxico: Comentario con una sola barra inclinada (inválido) en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Ignorar espacios y tabulaciones

t\_ignore = ' \t'

# Manejo de nuevas líneas

def t\_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)

# Manejo de errores mejorado

def t\_error(t):

    print(f"Error léxico: Carácter no reconocido '{t.value[0]}' en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Construir el lexer

lexer = lex.lex()

# Ignorar espacios en blanco

t\_ignore = ' \t'

# Manejo de errores léxicos

def t\_error(t):

    print(f"Error de estructura de control: {t.value} en la línea {t.lineno}")

    t.lexer.skip(1)

# Construir el lexer

lexer = lex.lex()

# Definir las reglas del parser

def p\_estructura\_control(p):

    '''estructura\_control : sentencia\_if

                          | sentencia\_while

                          | sentencia\_for'''

    p[0] = p[1]

def p\_sentencia\_if(p):

    '''sentencia\_if : IF PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER

                    | IF PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER ELSE LLAVEIZQ bloque LLAVEDER'''

    if len(p) == 8:

        p[0] = ('if', p[3], p[6])

    else:

        p[0] = ('if', p[3], p[6], p[10])

def p\_sentencia\_while(p):

    '''sentencia\_while : WHILE PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER'''

    p[0] = ('while', p[3], p[6])

def p\_sentencia\_for(p):

    '''sentencia\_for : FOR PARIZQ asignacion PUNTOYCOMA expresion PUNTOYCOMA expresion PARDER LLAVEIZQ bloque LLAVEDER'''

    p[0] = ('for', p[3], p[5], p[7], p[10])

def p\_bloque(p):

    '''bloque : sentencia

             | bloque sentencia'''

    if len(p) == 2:

        p[0] = [p[1]]

    else:

        p[0] = p[1] + [p[2]]

def p\_asignacion(p):

    '''asignacion : IDENTIFICADOR MAS NUMERO

                  | IDENTIFICADOR MENOS NUMERO'''

    p[0] = (p[1], p[2], p[3])

def p\_expresion(p):

    '''expresion : IDENTIFICADOR

                 | NUMERO'''

    p[0] = p[1]

def p\_sentencia(p):

    '''sentencia : asignacion PUNTOYCOMA'''

    p[0] = p[1]

def p\_error(p):

    print(f"Error de estructura de control: '{p.value}' en la línea {p.lineno}" if p else "Error de estructura de control en la entrada.")

# Construir el parser

parser = yacc.yacc()

# Función para analizar código

def analizar\_codigo(codigo):

    parser.parse(codigo)

    return t

# Manejo de números flotantes incorrectos

def t\_FLOAT(t):

    r'\d+\.\.\d+'

    print(f"Error léxico: Número flotante incorrecto '{t.value}' en línea {t.lexer.lineno}")

    t.value = ""  # Limpiar el valor del token

    return t

# Manejo de cadenas mal formadas

def t\_STRING(t):

    r'"([^"\\]|\\.)\*"'

    if t.value.count('"') % 2 == 1:  # Verifica si hay una comilla de cierre

        print(f"Error léxico: Cadena mal formada en línea {t.lexer.lineno}")

        t.value = ""  # Limpiar el valor del token

    else:

        t.value = t.value[1:-1]  # Remover comillas

    return t

# Manejo de comentarios mal cerrados

def t\_COMMENT\_SHARP(t):

    r'\#.\*'

    pass

def t\_COMMENT(t):

    r'//.\*|/\\*[\s\S]\*?\\*/'

    if t.value.startswith('/\*') and not t.value.endswith('\*/'):

        print(f"Error léxico: Comentario mal cerrado en línea {t.lexer.lineno}")

    pass

# Manejo de caracteres especiales no válidos (@)

def t\_INVALID\_CHAR(t):

    r'@'

    print(f"Error léxico: Uso de un carácter especial no válido '@' en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de números con letras en medio

def t\_INVALID\_NUMBER(t):

    r'\d+[a-zA-Z]\d+'

    print(f"Error léxico: Uso de un número con una letra en medio '{t.value}' en línea {t.lexer.lineno}")

    t.value = ""  # Limpiar el valor del token

    return t

# Manejo de operadores desconocidos ($)

def t\_UNKNOWN\_OPERATOR(t):

    r'\$'

    print(f"Error léxico: Operador desconocido '$' en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de punto y coma faltante

def t\_SEMICOLON\_MISSING(t):

    r'(?<=\w)\s\*$'

    print(f"Error léxico: Falta punto y coma al final de la declaración en línea {t.lexer.lineno}")

    return t

# Manejo de uso incorrecto de secuencias de escape

def t\_INVALID\_ESCAPE\_SEQUENCE(t):

    r'\\[^abfnrtv\\\'"]'

    print(f"Error léxico: Uso incorrecto de secuencias de escape en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de comillas simples en lugar de dobles en cadenas

def t\_SINGLE\_QUOTE\_IN\_STRING(t):

    r"'([^'\\]|\\.)\*'"

    print(f"Error léxico: Uso de comillas simples en lugar de dobles en cadena de texto en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de números hexadecimales mal formados

def t\_INVALID\_HEX(t):

    r'\b[^0x0-9A-Fa-f]\d+[A-Fa-f0-9]{2,}\b'

    print(f"Error léxico: Número hexadecimal mal formado '{t.value}' en línea {t.lexer.lineno}")

    t.value = ""  # Limpiar el valor del token

    return t

# Manejo de comentarios con una sola barra (inválido)

def t\_INVALID\_SINGLE\_SLASH\_COMMENT(t):

    r'[^/]/.\*'

    print(f"Error léxico: Comentario con una sola barra inclinada (inválido) en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Ignorar espacios y tabulaciones

t\_ignore = ' \t'

# Manejo de nuevas líneas

def t\_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)

# Manejo de errores mejorado

def t\_error(t):

    print(f"Error léxico: Carácter no reconocido '{t.value[0]}' en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Construir el lexer

lexer = lex.lex()

    p[0] = (p[1], p[2], p[3])

def p\_expresion(p):

    '''expresion : IDENTIFICADOR

                 | NUMERO'''

    p[0] = p[1]

def p\_sentencia(p):

    '''sentencia : asignacion PUNTOYCOMA'''

    p[0] = p[1]

def p\_error(p):

    print(f"Error de estructura de control: '{p.value}' en la línea {p.lineno}" if p else "Error de estructura de control en la entrada.")

# Construir el parser

parser = yacc.yacc()

# Función para analizar código

def analizar\_codigo(codigo):

    parser.parse(codigo)

    print(f"Error léxico: Uso incorrecto de secuencias de escape en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de comillas simples en lugar de dobles en cadenas

def t\_SINGLE\_QUOTE\_IN\_STRING(t):

    r"'([^'\\]|\\.)\*'"

    print(f"Error léxico: Uso de comillas simples en lugar de dobles en cadena de texto en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Manejo de números hexadecimales mal formados

def t\_INVALID\_HEX(t):

    r'\b[^0x0-9A-Fa-f]\d+[A-Fa-f0-9]{2,}\b'

    print(f"Error léxico: Número hexadecimal mal formado '{t.value}' en línea {t.lexer.lineno}")

    t.value = ""  # Limpiar el valor del token

    return t

# Manejo de comentarios con una sola barra (inválido)

def t\_INVALID\_SINGLE\_SLASH\_COMMENT(t):

    r'[^/]/.\*'

    print(f"Error léxico: Comentario con una sola barra inclinada (inválido) en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Ignorar espacios y tabulaciones

t\_ignore = ' \t'

# Manejo de nuevas líneas

def t\_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)

# Manejo de errores mejorado

def t\_error(t):

    print(f"Error léxico: Carácter no reconocido '{t.value[0]}' en línea {t.lexer.lineno}")

    t.lexer.skip(1)

# Construir el lexer

lexer = lex.lex()

Este código implementa un analizador léxico y sintáctico (usando la biblioteca ply de Python) para reconocer y analizar estructuras de control comunes como if, else, while y for en un lenguaje de programación hipotético. Aquí te explico cómo funciona cada parte del código:

**1. Definición de Tokens:**

Se definen los tokens que el lexer reconocerá. Los tokens son representaciones de las unidades léxicas básicas del lenguaje que estamos analizando.

* IF, ELSE, WHILE, FOR son las palabras clave del lenguaje.
* IDENTIFICADOR es cualquier nombre de variable, función, etc.
* NUMERO es una secuencia de dígitos.
* MAS, MENOS, PARIZQ, PARDER, LLAVEIZQ, LLAVEDER, y PUNTOYCOMA son operadores y delimitadores.

Las expresiones regulares definen cómo detectar cada uno de estos tokens en el texto que se analiza.

**2. Ignorar Espacios en Blanco:**

La línea t\_ignore = ' \t' indica que el lexer debe ignorar los espacios y tabulaciones. Esto significa que estos caracteres no serán considerados para la creación de tokens.

**3. Manejo de Errores Léxicos:**

La función t\_error(t) se ejecuta cuando el lexer encuentra un carácter que no corresponde a ninguno de los tokens definidos. Imprime un mensaje de error indicando que hay un problema con la estructura de control.

**4. Definición de las Reglas del Parser:**

Aquí se definen las reglas de sintaxis del lenguaje utilizando ply.yacc. Estas reglas indican cómo se combinan los tokens para formar estructuras de control completas.

* p\_estructura\_control(p) define que una estructura de control puede ser una sentencia if, while o for.
* p\_sentencia\_if(p) describe la sintaxis de un if que puede tener o no una cláusula else.
* p\_sentencia\_while(p) define la estructura de un while con una condición y un bloque de código.
* p\_sentencia\_for(p) describe la estructura de un for con inicialización, condición, actualización y bloque de código.
* p\_bloque(p) define un bloque de sentencias como una o más sentencias dentro de llaves {}.
* p\_asignacion(p) define cómo se realiza una asignación en el lenguaje (por ejemplo, x + 1 o x - 1).
* p\_expresion(p) define qué es una expresión (un identificador o un número).
* p\_sentencia(p) define que una sentencia es una asignación seguida de un punto y coma ;.

**5. Manejo de Errores Sintácticos:**

La función p\_error(p) se ejecuta cuando el parser encuentra un error de sintaxis. Si se encuentra un error, imprime un mensaje con la línea y el valor donde ocurrió el error.

**Parser**

import ply.yacc as yacc

from lexer import lexer

# Definición de la gramática y reglas del parser

def p\_estructura\_control(p):

    '''estructura\_control : sentencia\_if

                          | sentencia\_while

                          | sentencia\_for'''

    p[0] = p[1]

def p\_sentencia\_if(p):

    '''sentencia\_if : IF PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER

                    | IF PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER ELSE LLAVEIZQ bloque LLAVEDER'''

    if len(p) == 9:

        p[0] = ('if', p[3], p[6])  # ('if', condición, bloque)

    elif len(p) == 13:

        p[0] = ('if', p[3], p[6], p[10])  # ('if', condición, bloque, else\_bloque)

def p\_sentencia\_while(p):

    '''sentencia\_while : WHILE PARIZQ expresion PARDER LLAVEIZQ bloque LLAVEDER'''

    p[0] = ('while', p[3], p[6])  # ('while', condición, bloque)

def p\_sentencia\_for(p):

    '''sentencia\_for : FOR PARIZQ inicializacion PUNTOYCOMA expresion PUNTOYCOMA actualizacion PARDER LLAVEIZQ bloque LLAVEDER'''

    p[0] = ('for', p[3], p[5], p[7], p[10])  # ('for', inicialización, condición, actualización, bloque)

def p\_bloque(p):

    '''bloque : sentencia

             | bloque sentencia'''

    p[0] = p[1:]  # Almacena las sentencias del bloque

def p\_sentencia(p):

    '''sentencia : expresion

                 | estructura\_control

                 | declaracion

                 | asignacion'''

    p[0] = p[1]

def p\_expresion(p):

    '''expresion : IDENTIFICADOR

                 | NUMERO'''

    p[0] = p[1]

def p\_inicializacion(p):

    '''inicializacion : declaracion

                      | asignacion'''

    p[0] = p[1]

def p\_actualizacion(p):

    '''actualizacion : expresion'''

    p[0] = p[1]

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 código

def analizar\_codigo(codigo):

    try:

        parser.parse(codigo)

        print("Código válido")

    except Exception as e:

        print(f"Error de sintaxis: {e}")

1. **t\_CHARACTER\_UNKNOWN**: Detecta cualquier carácter no reconocido.
2. **t\_IDENTIFIER**: Detecta identificadores mal formados (como los que empiezan con un número).

Este código crea un **analizador sintáctico** (parser) utilizando ply.yacc y un lexer previamente definido (importado desde lexer). El objetivo del analizador es verificar la validez sintáctica de un fragmento de código que contiene estructuras de control como if, while, for, asignaciones, declaraciones y expresiones. A continuación te explico cómo funciona el código:

**1. Definición de la Gramática y Reglas del Parser:**

El parser se construye a partir de las reglas de gramática que se definen en las funciones p\_\*:

**a. Reglas para Estructuras de Control:**

* **estructura\_control**: Esta es una regla general que agrupa las sentencias if, while y for. Un bloque de control puede ser cualquiera de estas sentencias.

**b. Sentencias:**

* **sentencia\_if**: Esta regla define las sentencias if y if-else:
  + Si el if tiene una cláusula else, se captura con 13 elementos (if, condición, bloque, else, bloque).
  + Si no tiene else, se captura con 9 elementos (if, condición, bloque).
  + La salida es una tupla con la estructura de la sentencia if: ('if', condición, bloque) o ('if', condición, bloque, else\_bloque).
* **sentencia\_while**: Esta regla define la sentencia while con una condición y un bloque de código a ejecutar. La salida es una tupla ('while', condición, bloque).
* **sentencia\_for**: Define la sentencia for, que incluye:
  + inicialización: puede ser una declaración o una asignación.
  + condición: expresión que se evalúa en cada iteración.
  + actualización: puede ser una expresión de actualización de la variable.
  + bloque: bloque de código a ejecutar dentro del for.

**c. Bloque de Sentencias:**

* **bloque**: Esta regla define un bloque como una secuencia de una o más sentencias. Si hay varias sentencias, se almacenan como una lista.

**d. Sentencia General:**

* **sentencia**: La sentencia puede ser una expresión, una estructura de control, una declaración o una asignación.

**e. Expresiones:**

* **expresion**: Define que una expresión puede ser un IDENTIFICADOR o un NUMERO.

**f. Inicialización y Actualización:**

* **inicializacion**: Es una declaración o asignación que se ejecuta al inicio del for.
* **actualizacion**: Es una expresión que se evalúa al final de cada iteración del for.

**2. Manejo de Errores Sintácticos:**

La función p\_error(p) se encarga de manejar errores sintácticos:

* Si el token en error es conocido (es decir, no es None), imprime un mensaje indicando el token erróneo y la línea en que ocurrió.
* Si no hay información sobre el token erróneo (p es None), se imprime un mensaje genérico de error de sintaxis.

**3. Construcción del Parser:**

* La función yacc.yacc() genera el parser a partir de las reglas de gramática definidas previamente.

**4. Función para Analizar Código:**

La función analizar\_codigo(codigo) recibe el código a analizar. Se intenta analizar el código con el parser. Si todo está bien, imprime un mensaje diciendo "Código válido". Si se encuentra un error de sintaxis, se captura y muestra el error correspondiente.

**Salida**cuser\x5cDesktop\x5ctraductores\x5cHoy\x5cTokkens-y-Exp.-Reguares\x5cTraductores\x5cA6 Análisis sintáctico de estructuras de control\x5clexer.py' ;1b0bcab8-54b3-4838-a72d-14a6073e7568

Analizando código inválido:

Error de estructura de control:

if (x > 0 { x = x + 1; }

while x < 10) { y = y + 1; }

for i = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: 'if' en la línea 1

Error de estructura de control: > 0 { x = x + 1; }

while x < 10) { y = y + 1; }

for i = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: = x + 1; }

while x < 10) { y = y + 1; }

for i = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control:

while x < 10) { y = y + 1; }

for i = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: < 10) { y = y + 1; }

for i = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: = y + 1; }

for i = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control:

for i = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: = 0; i < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: < 10; i = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: = i + 1) { suma = suma + i; }

en la línea 1

Error de estructura de control: = suma + i; }

en la línea 1

Error de estructura de control:

en la línea 1 **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>