Universidad de Guadalajara

Ingeniería en Computación

Compiladores

Practica 2 – Analizador Sintáctico

13-MARZO-2025

Roberto Carlos Rodríguez Rivas

En esta práctica desarrollaremos un analizador sintáctico en Python utilizando la librería PLY (Python Lex-Yacc). Su objetivo es validar la estructura de expresiones matemáticas y sentencias de asignación según una gramática definida. Este es el segundo paso en la construcción de un compilador funcional al final del curso.

Explicación del código

Análisis Sintáctico

El analizador sintáctico se encarga de verificar que una secuencia de tokens generada por el analizador léxico sigue una estructura gramatical válida.

Definición de la Gramática

Para esta práctica, la gramática utilizada es:

<program> -> <assignment> | <assignment> <program>

<assignment> -> <identifier> = <expression> ;

<identifier> -> [a-zA-Z][a-zA-Z0-9_]*

<expression> -> <term> | <term> + <expression> | <term> - <expression>

<term> -> <factor> | <factor> * <term> | <factor> / <term>

<factor> -> <identifier> | <number> | (<expression>)

<number> -> [0-9]+

El analizador sintáctico está construido con ply.yacc. Se definen reglas de producción en funciones donde:

- **p_program**: Define que un programa puede ser una o más asignaciones.
- **p_assignment**: Describe una asignación como identificador = expresión;.
- **p_expression**: Define una expresión como un término o una operación con + o -.
- **p_term**: Describe un término como un factor o una operación con * o /.
- **p_factor**: Determina que un factor puede ser un identificador, un número o una expresión entre paréntesis.

Se incluye una función **p_error** para manejar errores de sintaxis.

Código fuente con comentarios:

```
import ply.lex as lex
import ply.yacc as yacc

# Definición de tokens para el analizador léxico
tokens = (
    'IDENTIFIER', 'NUMBER',
```

```
    'PLUS', 'TIMES', 'DIVIDE', 'EQUALS', 'LPAREN', 'RPAREN', 'SEMICOLON'
)

# Definición de expresiones regulares para los tokens
t_LPAREN    = r'\('
t_RPAREN    = r'\)'
t_PLUS      = r'\+'
t_TIMES     = r'\*'
t_DIVIDE    = r'/'
t_EQUALS    = r'='
t_SEMICOLON = r';'

# Regla para los identificadores (variables)
def t_IDENTIFIER(t):
    r'[a-zA-Z][a-zA-Z0-9_]*'  # Debe comenzar con una letra y puede contener
letras, números y guiones bajos
    return t

# Regla para los números enteros
def t_NUMBER(t):
    r'\d+'  # Coincide con uno o más dígitos
    t.value = int(t.value)  # Convierte el valor a un número entero
    return t

# Ignorar espacios y tabulaciones
t_ignore = ' \t'

# Regla para manejar nuevas líneas
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)  # Incrementa el número de línea

# Manejo de errores en el análisis léxico
def t_error(t):
    print(f"Carácter ilegal '{t.value[0]}'")  # Muestra el carácter no reconocido
    t.lexer.skip(1)  # Salta el carácter no válido

# Construcción del analizador léxico
lexer = lex.lex()

# Definición de la gramática para el analizador sintáctico

def p_program(p):
    '''program : assignment
               | assignment program'''
```

```
    # Un programa puede ser una única asignación o una asignación seguida de otro
programa
    if len(p) == 2:
        p[0] = [p[1]]  # Si es una única asignación, se almacena en una lista
    else:
        p[0] = [p[1]] + p[2]  # Si hay más asignaciones, se concatenan en una
lista

def p_assignment(p):
    'assignment : IDENTIFIER EQUALS expression SEMICOLON'
    # Una asignación tiene la forma: identificador = expresión ;
    p[0] = ('assign', p[1], p[3])  # Se almacena como una tupla ('assign',
identificador, expresión)

def p_expression(p):
    '''expression : term
                  | term PLUS expression'''
    # Una expresión puede ser un término solo o un término seguido de una suma y
otra expresión
    if len(p) == 2:
        p[0] = p[1]  # Si es solo un término, se devuelve como está
    else:
        p[0] = ('+', p[1], p[3])  # Si hay una suma, se almacena como una tupla
('+', término, expresión)

def p_term(p):
    '''term : factor
            | factor TIMES term
            | factor DIVIDE term'''
    # Un término puede ser un factor solo o un factor multiplicado o dividido por
otro término
    if len(p) == 2:
        p[0] = p[1]  # Si es solo un factor, se devuelve directamente
    else:
        p[0] = (p[2], p[1], p[3])  # Si hay multiplicación o división, se
almacena como una tupla (operador, factor, término)

def p_factor(p):
    '''factor : IDENTIFIER
              | NUMBER
              | LPAREN expression RPAREN'''
    # Un factor puede ser un identificador, un número o una expresión entre
paréntesis
    if len(p) == 2:
        p[0] = p[1]  # Si es un identificador o número, se devuelve directamente
```

```
    else:
        p[0] = p[2]  # Si está entre paréntesis, se devuelve la expresión interna

# Manejo de errores en el análisis sintáctico
def p_error(p):
    print("Error de sintaxis")  # Mensaje de error cuando hay una estructura
incorrecta

# Construcción del analizador sintáctico
parser = yacc.yacc()
```

Ejemplos de ejecución con capturas de pantalla.

- **Ejemplos Validos**

```
PS C:\Users\rober\OneDrive\Escritorio\Compiladores\Practica 2> & "c:/User
e/Escritorio/Compiladores/Practica 2/pruebas.py"
Probando: x = 10;
Resultado: [('assign', 'x', 10)]

Probando: y = x + 5;
Resultado: [('assign', 'y', ('+', 'x', 5))]

Probando: z = (3 + 2) * 4;
Resultado: [('assign', 'z', ('*', ('+', 3, 2), 4))]

Probando: a = b * c + d;
Resultado: [('assign', 'a', ('+', ('*', 'b', 'c'), 'd'))]

Probando: num = 20 / 4;
Resultado: [('assign', 'num', ('/', 20, 4))]

Probando: res = 3 + (5 * (2 + 1));
Resultado: [('assign', 'res', ('+', 3, ('*', 5, ('+', 2, 1))))]

Probando: var1 = 7; var2 = var1 + 3;
Resultado: [('assign', 'var1', 7), ('assign', 'var2', ('+', 'var1', 3))]
```

- **Ejemplos Inválidos**

```
Probando: temp = (x + y) / (z - 1);
Carácter ilegal '-'
Error de sintaxis
Resultado: None

Probando: x = ;
Error de sintaxis
Resultado: None

Probando: y = 3 + * 2;
Error de sintaxis
Resultado: None

Probando: z = (4 + 2;
Error de sintaxis
Resultado: None

Probando: = 5 + 3;
Error de sintaxis
Resultado: None

Probando: num = 10 + ;
Error de sintaxis
Resultado: None

Probando: var 10 = x;
Error de sintaxis
Resultado: None

Probando: x = 2 + 3
Error de sintaxis
Resultado: None
```

Justifique los resultados obtenidos.

Este código implementa un analizador sintáctico en Python utilizando **PLY**, permitiendo validar la estructura de asignaciones y expresiones matemáticas. Los resultados obtenidos confirman que el analizador cumple su función al verificar que las expresiones sigan la sintaxis definida, identificando correctamente asignaciones y operaciones matemáticas.

El analizador fue capaz de procesar expresiones correctamente formadas, detectando errores sintácticos cuando la entrada no cumplía con la gramática especificada. Esto es esencial en el proceso de compilación, ya que proporciona una base estructurada para la generación de código intermedio en futuras fases del compilador.