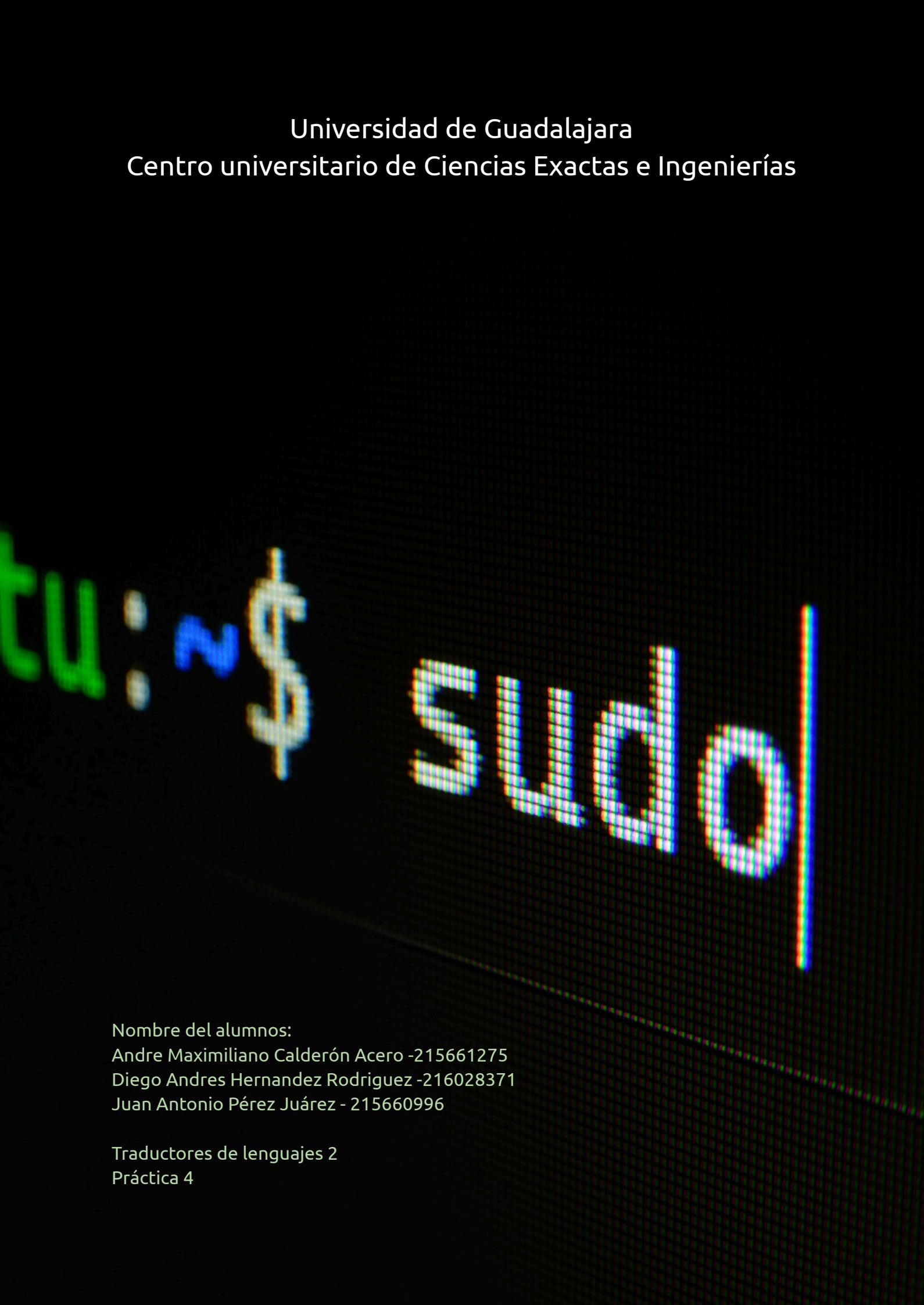
Universidad de Guadalajara
Centro universitario de Ciencias Exactas e Ingenierías



Nombre del alumnos:
Andre Maximiliano Calderón Acero -215661275
Diego Andres Hernandez Rodriguez -216028371
Juan Antonio Pérez Juárez - 215660996

Traductores de lenguajes 2
Práctica 4

Actividad

Analizador sintáctico descendente predictivo con analizador léxico que lea identificadores y números de más de un dígito usando la gramática de la página 360 del libro del dragón.

Agregar operadores de multiplicación y división a la gramática.

Marco Teórico

Un analizador sintáctico descendente predictivo es un tipo de parser que procesa la entrada desde el símbolo inicial de la gramática hacia las hojas, utilizando un enfoque predictivo basado en gramáticas LL(1), lo que evita el retroceso mediante el uso de una tabla de análisis que determina de manera única la producción a aplicar en cada paso. Este analizador trabaja en conjunto con un analizador léxico, el cual se encarga de leer el flujo de caracteres de entrada y agruparlos en tokens como identificadores (definidos por patrones como [a-zA-Z_][a-zA-Z0-9_]*) y números de más de un dígito (enteros o decimales, reconocidos mediante expresiones como [0-9]+(\.[0-9]+)?).

La gramática debe estar diseñada para ser no ambigua y sin recursión por la izquierda, permitiendo así una derivación eficiente. El proceso de análisis sintáctico se basa en comparar los tokens generados por el lexer con las reglas gramaticales almacenadas en la tabla predictiva, expandiendo los no terminales según corresponda y construyendo el árbol de derivación de manera sistemática. Este enfoque es eficiente en tiempo lineal, aunque presenta limitaciones al requerir gramáticas LL(1), lo que puede exigir modificaciones en la definición original del lenguaje. Su aplicación es común en el desarrollo de intérpretes y compiladores para lenguajes con estructuras sintácticas bien definidas y relativamente simples.

1) E → E1 + T	E.nodo = new Nodo('+', E1.nodo, T.nodo)
2) E → E1 - T	E.nodo = new Nodo('-', E1.nodo, T.nodo)
3) E → T	E.nodo = T.nodo
4) T → T1 * F	T.nodo = new Nodo('*', T1.nodo, F.nodo)
5) T → T1 / F	T.nodo = new Nodo('/', T1.nodo, F.nodo)
6) T → F	T.nodo = F.nodo
7) F → (E)	F.nodo = E.nodo
8) F → id	F.nodo = new Hoja(id, id.entrada)
9) F → num	F.nodo = new Hoja(num, num.val)

Código en python de la práctica

Python

```python
import re

class Nodo:
    def __init__(self, operacion, izquierda=None,
derecha=None):
        self.operacion = operacion
        self.izquierda = izquierda
        self.derecha = derecha

class Hoja:
    def __init__(self, tipo, valor):
        self.tipo = tipo  # Puede ser "id" o "num"
        self.valor = valor

# Analizador Léxico
class Lexer:
    def __init__(self, input_text):
        self.tokens = []
        self.index = 0
        self.input_text = input_text
        self.tokenize()

    def tokenize(self):
        patterns = [
            ('NUM', r'\d+'),
            ('ID', r'[a-zA-Z][a-zA-Z0-9]*'),
            ('PLUS', r'\+'),
            ('MINUS', r'-'),
            ('MUL', r'\*'),
            ('DIV', r'/'),
            ('LPAREN', r'\('),
            ('RPAREN', r'\)'),
            ('WHITESPACE', r'\s+'),
        ]
        combined_pattern = '|'.join(f'(?P<{name}>{pattern})'
for name, pattern in patterns)
```

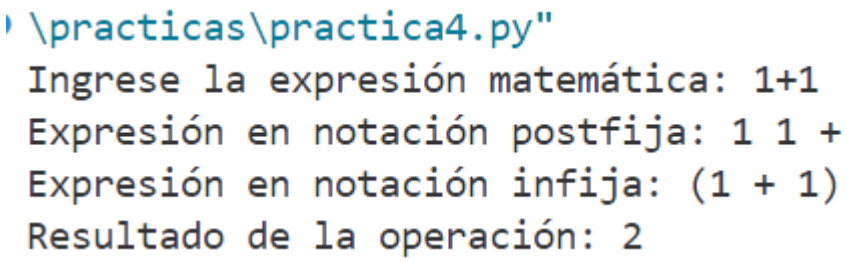
```
        regex = re.compile(combined_pattern)
        for match in regex.finditer(self.input_text):
            kind = match.lastgroup
            value = match.group()
            if kind != 'WHITESPACE':
                self.tokens.append((kind, value))

    def next_token(self):
        if self.index < len(self.tokens):
            tok = self.tokens[self.index]
            self.index += 1
            return tok
        else:
            return ('EOF', 'EOF')

    def peek_token(self):
        if self.index < len(self.tokens):
            return self.tokens[self.index]
        else:
            return ('EOF', 'EOF')

# Analizador Sintáctico
class Parser:
    def __init__(self, lexer):
        self.lexer = lexer
        self.token = self.lexer.next_token()

    def eat(self, token_type):
        if self.token[0] == token_type:
            self.token = self.lexer.next_token()
        else:
            raise SyntaxError(f"Se esperaba {token_type}, pero
se encontró {self.token}")

    def parse(self):
        nodo = self.E()
        if self.token[0] != 'EOF':
            raise SyntaxError("Cadena extra después de la
expresión válida")
```

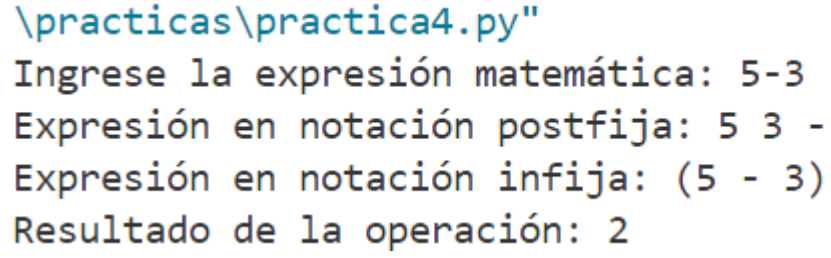
```
        return nodo

    def E(self):
        nodo = self.T()
        while self.token[0] in ('PLUS', 'MINUS'):
            if self.token[0] == 'PLUS':
                self.eat('PLUS')
                nodo = Nodo('+', nodo, self.T())
            elif self.token[0] == 'MINUS':
                self.eat('MINUS')
                nodo = Nodo('-', nodo, self.T())
        return nodo

    def T(self):
        nodo = self.F()
        while self.token[0] in ('MUL', 'DIV'):
            if self.token[0] == 'MUL':
                self.eat('MUL')
                nodo = Nodo('*', nodo, self.F())
            elif self.token[0] == 'DIV':
                self.eat('DIV')
                nodo = Nodo('/', nodo, self.F())
        return nodo

    def F(self):
        if self.token[0] == 'LPAREN':
            self.eat('LPAREN')
            nodo = self.E()
            self.eat('RPAREN')
            return nodo
        elif self.token[0] == 'ID':
            value = self.token[1]
            self.eat('ID')
            return Hoja('id', value)
        elif self.token[0] == 'NUM':
            value = self.token[1]
            self.eat('NUM')
            return Hoja('num', value)
        else:
```

```
            raise SyntaxError("Se esperaba ( o identificador o
número")

# Funciones auxiliares para recorrer el árbol
def imprimir_postorden(nodo):
    if isinstance(nodo, Hoja):
        return f"{nodo.valor}"
    else:
        return f"{imprimir_postorden(nodo.izquierda)}
{imprimir_postorden(nodo.derecha)} {nodo.operacion}"

def imprimir_infijo(nodo):
    if isinstance(nodo, Hoja):
        return f"{nodo.valor}"
    else:
        return f"({imprimir_infijo(nodo.izquierda)}
{nodo.operacion} {imprimir_infijo(nodo.derecha)})"

def evaluar_arbol(nodo):
    if isinstance(nodo, Hoja):
        if nodo.tipo == 'num':
            return int(nodo.valor)
        else:
            raise ValueError("No se puede evaluar un
identificador")
    else:
        izquierda = evaluar_arbol(nodo.izquierda)
        derecha = evaluar_arbol(nodo.derecha)
        if nodo.operacion == '+':
            return izquierda + derecha
        elif nodo.operacion == '-':
            return izquierda - derecha
        elif nodo.operacion == '*':
            return izquierda * derecha
        elif nodo.operacion == '/':
            if derecha == 0:
                raise ZeroDivisionError("División entre cero")
            return izquierda / derecha
```

```python
# Ejemplo de uso
if __name__ == "__main__":
    entrada = input("Ingrese la expresión matemática: ")
    lexer = Lexer(entrada)
    parser = Parser(lexer)

    try:
        arbol = parser.parse()
        print("Expresión en notación postfija:",
imprimir_postorden(arbol))
        print("Expresión en notación infija:",
imprimir_infijo(arbol))
        resultado = evaluar_arbol(arbol)
        print("Resultado de la operación:", resultado)
    except SyntaxError as e:
        print("Error de sintaxis:", e)
    except ValueError as e:
        print("Error de evaluación:", e)
    except ZeroDivisionError as e:
        print("Error de evaluación:", e)
```

Capturas de pantalla





```
\practicas\practica4.py"
Ingrese la expresión matemática: 10*2
Expresión en notación postfija: 10 2 *
Expresión en notación infija: (10 * 2)
Resultado de la operación: 20

\practicas\practica4.py"
Ingrese la expresión matemática: (1+1)*2
Expresión en notación postfija: 1 1 + 2 *
Expresión en notación infija: ((1 + 1) * 2)
Resultado de la operación: 4

Ingrese la expresión matemática: 10*(2+2)/5
Expresión en notación postfija: 10 2 2 + * 5 /
Expresión en notación infija: ((10 * (2 + 2)) / 5)
Resultado de la operación: 8.0
```

Conclusión:

DIEGO ANDRES HERNANDEZ RODRIGUEZ

Con esta práctica estamos a un paso de poder realizar un compilador en forma, capaz de identificar y clasificar correctamente cada uno de los tokens que recibe y poder realizar las operaciones que se necesitan, en este caso podemos ver como interpreta cada uno de los números que son ingresados para realizar la operación aritmética necesaria.

Juan Antonio Pérez Juárez:

Eso de los compiladores, no se me da absolutamente nada, me dan ganas de estrangular a alguien, afortunadamente el libro está bien explicado.

Pero gracias a mis compañeros de equipo que me están cargando en este proyecto, porque si no, no hay manera de poder hacerlo a tiempo. Y el sueño de poder hacer una aportación al kernel de linux cada vez lo veo más lejano.

Andre Maximiliano Calderon Acero

Esta práctica me hizo comprender que los analizadores sintácticos predictivos son un tema muy denso y difícil de entender, si no fuera por mis compañeros y las guías del libro no creo que hubiera podido hacer algo, también es importante comprender que definir una gramática adecuadamente es vital para poder interpretar correctamente una expresión.