Instituto Politécnico Nacional

Escuela Superior de Cómputo

Unidad de Aprendizaje

Compiladores

Práctica 4: Analizador sintáctico LL(1)

Reporte Práctico

Profa.: Cecilia Albortante Morate

Nombres de los integrantes:

- **García Gasca Axel Gabriel**
- **Taboada Montiel Enrique**

Grupo: 5CM4

Fecha de Entrega: 04 de diciembre del 2024

Introducción

Un **analizador sintáctico LL(1)** es una herramienta utilizada en la teoría de lenguajes formales y compiladores para analizar si una secuencia de tokens (producida por un analizador léxico) cumple con las reglas de una gramática específica. Este tipo de analizador pertenece a la familia de los analizadores predictivos y es ideal para gramáticas libres de contexto que cumplen ciertas condiciones (gramáticas LL(1)).

Introducción al Analizador Sintáctico LL(1)

Un **analizador sintáctico LL(1)** es una herramienta utilizada en la teoría de lenguajes formales y compiladores para analizar si una secuencia de tokens (producida por un analizador léxico) cumple con las reglas de una gramática específica. Este tipo de analizador pertenece a la familia de los analizadores predictivos y es ideal para gramáticas libres de contexto que cumplen ciertas condiciones (gramáticas LL(1)).

¿Qué significa LL(1)?

- **L**: Lectura de izquierda a derecha del flujo de entrada.
- **L**: Derivación más a la izquierda de la gramática.
- **1**: Un solo token de mirada hacia adelante (*lookahead*) para tomar decisiones durante el análisis.

Esto significa que el analizador solo necesita mirar un token por adelantado para decidir cómo avanzar en la construcción de la estructura del lenguaje.

Introducción al Analizador Sintáctico LL(1)

Un **analizador sintáctico LL(1)** es una herramienta utilizada en la teoría de lenguajes formales y compiladores para analizar si una secuencia de tokens (producida por un analizador léxico) cumple con las reglas de una gramática específica. Este tipo de analizador pertenece a la familia de los analizadores predictivos y es ideal para gramáticas libres de contexto que cumplen ciertas condiciones (gramáticas LL(1)).

¿Qué significa LL(1)?

- **L**: Lectura de izquierda a derecha del flujo de entrada.
- **L**: Derivación más a la izquierda de la gramática.

- **1**: Un solo token de mirada hacia adelante (*lookahead*) para tomar decisiones durante el análisis.

Esto significa que el analizador solo necesita mirar un token por adelantado para decidir cómo avanzar en la construcción de la estructura del lenguaje.

¿Cómo funciona un analizador LL(1)?

1. **Gramática y tabla LL(1):**
    - El analizador se basa en una gramática libre de contexto, que describe las reglas del lenguaje.
    - Estas reglas se organizan en una tabla LL(1), donde:
        - Las filas representan los **no terminales** de la gramática.
        - Las columnas representan los **terminales** y el símbolo de fin de entrada.
        - Cada celda indica la producción que debe aplicarse en un contexto específico.
2. **Pila:**
    - La pila es la estructura principal donde se gestionan los símbolos que deben procesarse.
    - Inicialmente, contiene el símbolo de fin de entrada ($) y el no terminal inicial de la gramática.
3. **Proceso iterativo:**
    - Se compara el tope de la pila con el token actual de entrada:
        - Si coinciden (ambos son terminales), se consume el token y se elimina de la pila.
        - Si el tope es un no terminal, se consulta la tabla LL(1) para determinar qué regla de producción aplicará.
        - Si ocurre un error (el símbolo no coincide o no hay una regla válida), el análisis termina y se reporta un fallo.
    - El análisis concluye exitosamente cuando la pila está vacía y se han consumido todos los tokens.

Descripción del código

Es una representación de las reglas de producción de la gramática en forma de tabla LL(1). Cada fila corresponde a un no terminal

Cada columna corresponde a un terminar que puede aparecer como entrada y finalmente las producciones se definen como listas de símbolos que reemplazaran el no terminal.

```
tablaLL1 = {
    'E': {'numero': ['T', 'Ep'], 'identificador': ['T', 'Ep'], '(': ['T', 'Ep']},
    'Ep': {'+': ['+', 'T', 'Ep'], '-': ['-', 'T', 'Ep'], ')': [''], '$': ['']},
    'T': {'numero': ['F', 'Tp'], 'identificador': ['F', 'Tp'], '(': ['F', 'Tp']},
    'Tp': {'+': [''], '-': [''], '*': ['*', 'F', 'Tp'], '/': ['/', 'F', 'Tp'], ')': [''], '$': ['']},
    'F': {'numero': ['numero'], 'identificador': ['identificador'], '(': ['(', 'E', ')']}
}
```

Inicialización de la pila

```
pila = ['$', 'E']
num = 0
```

La comparación entre x y a se ejecuta de la siguiente forma:

```
if(x=='$' or x=='numero' or x=='identificador' or x=='(' or x==')' or x=='+' or x=='-' or x=='*' or x=='/'):
    if x == a:
        pila.pop()
        num += 1
        if pila.__len__() > 0:
            x = pila[-1]
    else:
        if(a == 'identificador' or a == 'numero'):
            print(f'Error: se esperaba un numero o identificador después de {tokens[num-1][1]}')
        else:
            print(f'Error: se esperaba un operador después de {tokens[num-1][1]}')
        return False
```

Caso donde x es un NO terminal

```
                return False
        else:
            try:
                pila.pop()
                for i in range(tablaLL1[x][a].__len__()):
                    if(tablaLL1[x][a][tablaLL1[x][a].__len__()-1-i] != ''):
                        pila.append(tablaLL1[x][a][tablaLL1[x][a].__len__()-1-i])
            except KeyError:
                if(a == 'identificador' or a == 'numero'):
                    print(f'Error: se esperaba un operador después de {tokens[num-1][1]}')
                else:
                    print(f'Error: se esperaba un número o identificador después de {tokens[num-1][1]}')
                return False
    return True
```

Ahora tenemos la parte de la tokenización

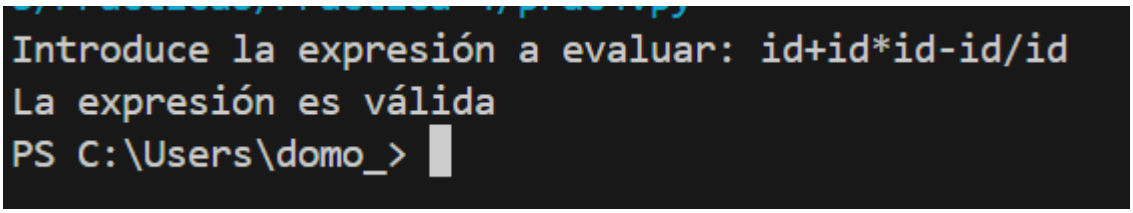
```
# devolver tokens
def tokenize(expression):
    # Patrón de tokenización
    token_pattern = re.compile(r'\d+|[a-zA-Z_]\w*|[()+*/-]')
    tokens = token_pattern.findall(expression)

    # Clasificación de tokens
    categorized_tokens = []
    for token in tokens:
        if token.isdigit():
            categorized_tokens.append(['numero', token])
        elif re.match(r'^[a-zA-Z_]\w*$', token):
            categorized_tokens.append(['identificador', token])
        else:
            categorized_tokens.append(['operador', token])

    # indicar fin de cadena
    categorized_tokens.append(['$', '$'])

    return categorized_tokens
```

Y finalmente tenemos la ejecución

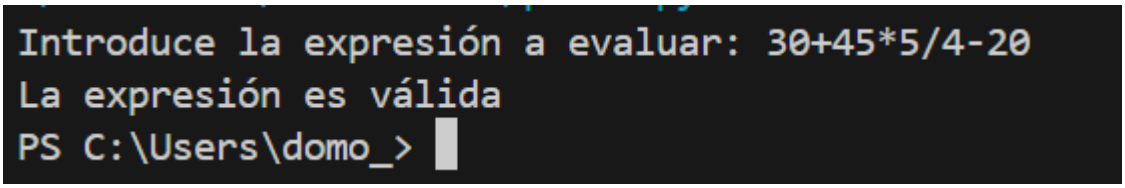
```
# ejecucion
expression = input("Introduce la expresión a evaluar: ")
tokens = tokenize(expression)

# analizar si son validas
if(analizadorLL1(tokens)):
    print("La expresión es válida")
else:
    print("La expresión no es válida")
```

Resultados

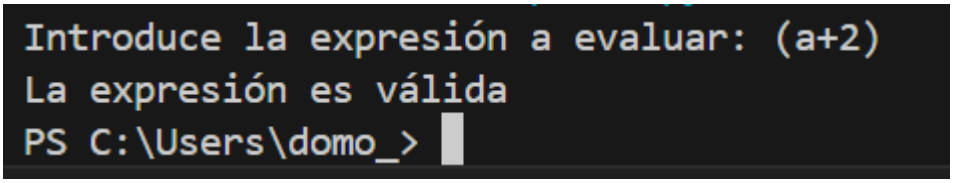
Primeramente, ingresamos la siguiente cadena válida y el resultado fue este:



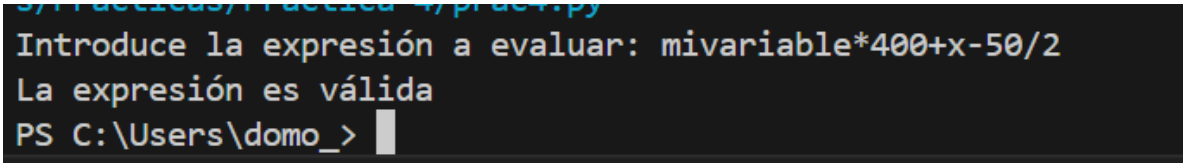
Probamos con otra:



Otra:



Finalmente otra válida:



Ahora probamos resultados que no son válidos