Implementar Intérprete I

Profesor: Angel Augusto Agudelo Zapata

Andres Felipe Gordillo Guerrero

1088348241

a.gordillo@utp.edu.co

Inteligencia Artificial

Universidad tecnológica de pereira

2024 - Semestre II

Este proyecto desarrolla un intérprete mejorado para un lenguaje que se asemeja a C, incorporando funcionalidades como la impresión de cadenas, operadores de incremento y decremento, operadores de asignación compuesta, así como la evaluación de expresiones lógicas. A continuación, se ofrece una descripción detallada de las clases, funciones y el flujo de ejecución.

Estructura de Archivos

- `mclex.py`: Define el analizador léxico (Lexer) que tokeniza el código fuente.
- `mcparser.py`: Define el analizador sintáctico (Parser) que genera el AST a partir de los tokens.
- `mcast.py`: Define las clases de nodos AST (Abstract Syntax Tree) para representar la estructura del programa.
- `interpreter.py` (nuevo archivo): Contiene la clase `InterpreteExtendido`, que interpreta el AST y realiza la ejecución del código.

Clases y Métodos

Lexer (Analizador Léxico) - `mclex.py`

La clase `Lexer` es responsable de dividir el código fuente en tokens. Los tokens son unidades básicas del lenguaje, como palabras clave, identificadores, operadores, etc.

Parser (Analizador Sintáctico) - `mcparser.py`

La clase `Parser` toma una lista de tokens y genera un AST. Utiliza reglas gramaticales para construir una representación jerárquica del código que será evaluada por el intérprete.

Nodos AST - `mcast.py`

Los nodos representan estructuras del AST. Algunos de los nodos principales son:

- **ASTNode**: Nodo base para todas las clases.
- **PrintfNode**: Representa una instrucción `printf`, manejando cadenas formateadas.
- **SprintfNode**: Similar a `PrintfNode`, pero devuelve una cadena formateada sin imprimirla.
- **PostIncrementNode**: Representa el operador de incremento postfijo (`x++`).
- **PreIncrementNode**: Representa el operador de incremento prefijo (`++x`).
- **PostDecrementNode**: Representa el operador de decremento postfijo (`x--`).
- **PreDecrementNode**: Representa el operador de decremento prefijo (`--x`).
- **PlusAssignNode**: Representa la operación `+=`.
- **MinusAssignNode**: Representa la operación `-=`.

- **MultAssignNode**: Representa la operación `*=`.
- **DivAssignNode**: Representa la operación `/=`.
- **LogicalAndNode**: Representa la operación lógica `&&`.
- **LogicalOrNode**: Representa la operación lógica `||`.
- **NullNode**: Representa un valor `NULL`.

InterpreteExtendido (Intérprete) - #mccinterp.py

```
from mclex import Lexer  # Cambia MCLexer por Lexer
from mcparser import Parser
from mcast import (ASTNode, PrintfNode, SprintfNode,
                   PostIncrementNode, PreIncrementNode,
                   PlusAssignNode, MinusAssignNode,
                   MultAssignNode, DivAssignNode,
                   LogicalAndNode, LogicalOrNode, NullNode)

# Definir la clase InterpreteExtendido directamente, sin necesidad de
importarla
class InterpreteExtendido:

    def __init__(self):
        # Inicializa un diccionario para las variables
        self.env = {}

    def visit(self, node):
        """Método principal de recorrido del AST que delega a los
métodos específicos de cada nodo."""
        method_name = f'visit_{type(node).__name__}'
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        """Método genérico para nodos no manejados explícitamente."""
        raise Exception(f"No visit method for {type(node).__name__}")

    def visit_PrintfNode(self, node):
        # Simulación básica de `printf`, reemplazando códigos de escape
        formatted_string = node.string.replace('\\n',
'\n').replace('\\t', '\t')
        print(formatted_string % self.visit(node.expr))

    def visit_SprintfNode(self, node):
        # `sprintf` simula la construcción de una cadena sin imprimir
```

```
        formatted_string = node.string.replace('\\n',
'\n').replace('\\t', '\t')
        return formatted_string % self.visit(node.expr)

    def visit_PostIncrementNode(self, node):
        # Guarda el valor actual, incrementa después
        value = self.visit(node.expr)
        self.env[node.expr.name] = value + 1
        return value

    def visit_PreIncrementNode(self, node):
        # Incrementa antes y devuelve el nuevo valor
        value = self.visit(node.expr) + 1
        self.env[node.expr.name] = value
        return value

    def visit_PostDecrementNode(self, node):
        # Guarda el valor actual, decrementa después
        value = self.visit(node.expr)
        self.env[node.expr.name] = value - 1
        return value

    def visit_PreDecrementNode(self, node):
        # Decrementa antes y devuelve el nuevo valor
        value = self.visit(node.expr) - 1
        self.env[node.expr.name] = value
        return value

    def visit_PlusAssignNode(self, node):
        # `x += value`
        current_value = self.env[node.identifier.name]
        self.env[node.identifier.name] = current_value +
self.visit(node.expr)
        return self.env[node.identifier.name]

    def visit_MinusAssignNode(self, node):
        # `x -= value`
        current_value = self.env[node.identifier.name]
        self.env[node.identifier.name] = current_value -
self.visit(node.expr)
        return self.env[node.identifier.name]

    def visit_MultAssignNode(self, node):
```

```
        # `x *= value`
        current_value = self.env[node.identifier.name]
        self.env[node.identifier.name] = current_value *
self.visit(node.expr)
        return self.env[node.identifier.name]

    def visit_DivAssignNode(self, node):
        # `x /= value`
        current_value = self.env[node.identifier.name]
        if self.visit(node.expr) == 0:
            raise ZeroDivisionError("Division by zero in '/='
operator.")
        self.env[node.identifier.name] = current_value /
self.visit(node.expr)
        return self.env[node.identifier.name]

    def visit_LogicalAndNode(self, node):
        # Evaluación de circuito-corto para `&&`
        left = self.visit(node.left)
        if not left:
            return False
        return self.visit(node.right)

    def visit_LogicalOrNode(self, node):
        # Evaluación de circuito-corto para `||`
        left = self.visit(node.left)
        if left:
            return True
        return self.visit(node.right)

    def visit_NullNode(self, node):
        # Maneja `NULL` como un valor especial
        return None

    def visit_Identifier(self, node):
        # Permite el uso de NULL en expresiones
        value = self.env.get(node.name)
        if value is None:
            raise NameError(f"Variable '{node.name}' is not defined.")
        return value

    # Otras visitas a nodos aquí...
```

```
# Ejecución principal de ejemplo:
if __name__ == '__main__':
    lexer = Lexer()  # Instancia de Lexer en lugar de MCLexer
    parser = Parser()
    code = """
    // Prueba de printf con códigos de escape
printf("Hola\\nMundo\\tCon\\tTabulacion");

    // Prueba de operadores ++ y -- en sus formas prefijas y postfijas
int a = 5;
int b = a++;    // b = 5, a = 6 (postfijo)
int c = ++a;    // c = 7, a = 7 (prefijo)
int d = a--;    // d = 7, a = 6 (postfijo)
int e = --a;    // e = 5, a = 5 (prefijo)
printf("Valores: b=%d, c=%d, d=%d, e=%d\\n", b, c, d, e);

    // Prueba de operadores de asignación compuesta
int x = 10;
x += 5;    // x = 15
x -= 3;    // x = 12
x *= 2;    // x = 24
x /= 4;    // x = 6
printf("Resultado de x: %d\\n", x);

    // Prueba de evaluación de circuito-corto
int y = 0;
int z = 10;
int res1 = (y != 0) && (z /= y);  // res1 = 0 (corto-circuito)
int res2 = (y == 0) || (z /= 2);  // res2 = 1, z = 5 (corto-circuito)
printf("Resultado de z: %d, res1: %d, res2: %d\\n", z, res1, res2);

    // Prueba de valor NULL
int* p = NULL;
if (p == NULL) {
    printf("Puntero es NULL\\n");
}
    """

    # Tokeniza el código y muestra los tokens generados para depuración
    tokens = lexer.tokenize(code)
    for token in tokens:
        print(token)
```

```
    ast = parser.parse(tokens)
    interpreter = InterpreteExtendido()

    if ast is not None:
        interpreter.visit(ast)
    else:
        print("El AST es None. El parseo falló.")
```

Esta clase es responsable de interpretar el AST y ejecutar el código. Utiliza un enfoque de *visitor pattern* para recorrer el árbol de sintaxis abstracta y ejecutar las instrucciones correspondientes a cada tipo de nodo. La clase mantiene un diccionario `env` para almacenar variables y sus valores.

Métodos de `InterpreteExtendido`:

- **`visit(self, node)`**: Método principal que delega la visita a un método específico según el tipo de nodo.
- **`generic_visit(self, node)`**: Método genérico que se utiliza si no se encuentra un método específico para el nodo.
- **`visit_PrintfNode(self, node)`**: Imprime una cadena formateada, manejando códigos de escape como `\n` y `\t`.
- **`visit_SprintfNode(self, node)`**: Genera una cadena formateada (sin imprimirla) utilizando el formato especificado.
- **`visit_PostIncrementNode(self, node)`**: Realiza un incremento postfijo en el valor de una variable.
- **`visit_PreIncrementNode(self, node)`**: Realiza un incremento prefijo en el valor de una variable.
- **`visit_PostDecrementNode(self, node)`**: Realiza un decremento postfijo en el valor de una variable.
- **`visit_PreDecrementNode(self, node)`**: Realiza un decremento prefijo en el valor de una variable.
- **`visit_PlusAssignNode(self, node)`**: Realiza la operación de asignación con suma (`x += value`).
- **`visit_MinusAssignNode(self, node)`**: Realiza la operación de asignación con resta (`x -= value`).
- **`visit_MultAssignNode(self, node)`**: Realiza la operación de asignación con multiplicación (`x *= value`).
- **`visit_DivAssignNode(self, node)`**: Realiza la operación de asignación con división (`x /= value`), manejando la excepción por división por cero.

- `visit_LogicalAndNode(self, node)`: Realiza la evaluación de circuito corto para el operador lógico `&&`.
- `visit_LogicalOrNode(self, node)`: Realiza la evaluación de circuito corto para el operador lógico `||`.
- `visit_NullNode(self, node)`: Retorna `None` para nodos `NULL`.
- `visit_Identifier(self, node)`: Recupera el valor de una variable desde el entorno (`env`), lanzando un error si la variable no está definida.

Ejecución Principal

El código dentro de la sección `if __name__ == '__main__':` realiza los siguientes pasos:

1. **Tokenización**: El código fuente se tokeniza usando el `Lexer`, y los tokens generados se imprimen para depuración.
2. **Parseo**: Los tokens son procesados por el `Parser` para generar un AST.
3. **Ejecución del AST**: El AST es evaluado por la clase `InterpreteExtendido`, que ejecuta las instrucciones y muestra los resultados de las operaciones en la consola.

Código de Prueba

El código de prueba implementa varias características del lenguaje:

- **`printf` con cadenas formateadas**: Simula la impresión de cadenas con códigos de escape.
- **Operadores de incremento y decremento**: Realiza incrementos y decrementos tanto prefijos como postfijos.
- **Operadores de asignación compuesta**: Usa `+=`, `-=`, `*=`, `/=`.
- **Evaluación de expresiones lógicas**: Simula el comportamiento de los operadores lógicos `&&` y `||` con evaluación de circuito corto.
- **Manejo de `NULL`**: Verifica la manipulación de punteros nulos.

Conclusión

Este intérprete extendido es un ejemplo de cómo construir un sistema de evaluación de código para un lenguaje similar a C, utilizando patrones como el de *visitor* para recorrer el AST y realizar las operaciones correspondientes. Cada parte del proyecto está modularizada para facilitar su extensión y mantenimiento.