INTERPRETER REPORT

LUA

Lua is a powerful, efficient, lightweight, embeddable scripting language. It supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description.

Lua combines simple procedural syntax with powerful data description constructs

FEATURES

- Assignment statements
- Arithmetic operators (+,-,/,*)
- Comparison Operators (>, =, <=, == ,!=)
- If Statement
- Nested if statement
- If_else if statement
- While loop
- Single and Multiline comments

CODE EXPLANATION

The whole code is divided in 3 main parts

1. LEXER

- The lexer main work is to take the LUA program as input character stream and convert it into tokens. Every token has a type, textual representation, and position in the original input. The position can help end users of the lexer with debugging.
- This can be used for a variety of purposes. You could apply transformations to the lexemes for simple text processing and manipulation.
- The lexer has functions that can read a number, a string, a variable name, and functions that skip whitespaces, comments (single and multiline) etc.
- The most important function of the lexical analyser is the get_next_token() function, which essentially returns the next token from the input program after advancing through it.
- The `EOF` token is a special token that marks the end of input. It doesn't exist in the source text; we only use it to simplify the parser stage.
- We start with an empty list of tokens, then we go through the string and add tokens as they come.

2. PARSER

- Parsing is the process of breaking down code into individual chunks of code, verifying that all necessary inputs are included in the code, and acting on the instructions dictated by the code
- After the lexical analysis has broken the code into workable chunks, the syntax analysis determines how these chunks relate to each other and builds a model of how the rendering engine should process the code based on this analysis
- it takes the lexer's token as input and returns an Abstract Syntax Tree (AST) of the entire program when the syntax is valid. It returns an error message if the syntax is inconsistent with the syntax (represented in BNF) of the subset of LUA we are working with.
- It has functions which represent non-terminals (like statement list, if statement etc.) of the BNF (discussed below), checking whether the appropriate syntax of those non-terminals has been followed or not and raising an exception if the syntax has not been followed.
- The eat (self, token type) function is very useful in this process. It takes a particular expected token type and gets the next token from the lexer if the current token matches the expected one. If not, it raises an error/exception.
- we create an AST class and its various node classes like If, Compare, BinOp, etc. We create the AST by creating instances of its node classes for each statement and combining them to form a tree.

3. INTERPRETER

- interpreter will make use of our lexer and parser to get the AST of our input expression and then evaluate that AST whichever way we want.
- Interpreters mostly execute one statement at a time. They take an AST returned by a parser and execute it. In that process they typically use some helper constructs like symbol tables, generators, and optimizers.
- It takes the AST of the program as input from parser and executes every single statement in the AST. The Output visible to the end user will be print statements (if any) and a global scope of all the variables in the program and their values after execution of the program.
- The NodeVisitor Class is fundamental class for the interpreter. This class helps in visiting every node of every statement of an AST. The getattr (self, method_name, self.generic_visit) is a built-in method in Python which returns the attribute within 'self' (NodeVisitor / interpreter) with the name "method_name". If no such attribute exists, then self.generic_visit is called. In our case, we are using it to get the method to visit a particular type of node.

CORE IDEA

After reading different blogs we got an idea to use fundamental idea of Recursive-Descent Parsing to parse the tokens and generate an AST. An abstract syntax tree is used as an intermediate representation of an input program. The interpreter/compiler can then do whatever it needs to do with it: optimize, simplify, execute, or something else. We have also used a NodeVisitor class in the interpreter part, which helps in visiting each node of an AST.

DEPENDICIES

To run the python file is the only dependency for the program.

CHALLENGES

we encountered lot of challenges while doing the project

1) we build if else statement while making it nested, we faced a problem that it's not working, else statement was not working in if else, we spend lot of time to debug it still it was out of scope.
2) we also faced a problem that DO statement was not working, program stops working after eating do statement
   solution: after few research we found out that its taking do stmnt as variable, since it was not reserved before ,so we made it as a reserved word.
3) when we were creating a while loop, we faced a problem that atlast end stmnt was getting error
   Problem: we found out that even though we made end stmnt as reserved word, still its taking it as a var it took first to letter "en" as variable then not taking d, at end we were able to resolve this.
4) We tried to write code for nested if else statement but it is giving error for "else" token so we commented else part of the loop.

Reserved keywords used

```
RESERVED_KEYWORDS={
    'if': Token(IF,'if'),
    'elsif': Token(ELSIF, 'elsif'),
    'else': Token(ELSE, 'else'),
    'while': Token(WHILE, 'while'),
    'do':Token(DO, 'do'),                                    #Used with while loop in LUA
    'end':Token(END,'end'),
    'then':Token(THEN,'then'),
}
```

FEATURES

Syntaxes for the features in interpreter

- Assignment statement

```
def assignment_statement(self):
    left = self.variable()
    self.eat(ID)
    print(self.variable())
    token = self.current_token
    self.eat(ASSIGN)
    right = self.expr()
    node = Assign(left, token, right)
    return node
```

- If statement / if else statement / nested if

```
def if_statement(self):
    self.eat(IF)
    condition = self.conditional_statement()
    self.eat(THEN)
    body = self.statement_list()
    rest = self.empty()

    if self.current_token.type == ELSE:
        self.eat(ELSE)
        rest = self.statement_list()

    node = If(condition,body,rest)

    return node
```

- While loop

```
def if_statement(self):
    self.eat(IF)
    condition = self.conditional_statement()
    self.eat(THEN)
    body = self.statement_list()
    rest = self.empty()

    if self.current_token.type == ELSE:
        self.eat(ELSE)
        rest = self.statement_list()

    node = If(condition,body,rest)

    return node
```

Note: outputs are given in separate file

LEARNINGS

1) We got acquainted with the stages of the interpretation process, we already learned about it in the theoretical part, but now we have one in practical sense.
2) we learned how to debug huge programs and how small error in code will ruin your code and reduce quality of it.
3) We learned how BNF is a powerful weapon in terms of crafting parsers and parse trees. We learned how to use/convert BNF functions/methods that can check for syntax errors.
4) We have learned to divide a system into modular systems subcomponents can be very useful. Such as lexer, parser, interpreter etc.
5) We learned the syntax of the LUA programming language and how it is similar/different from normal programming languages such as Python, JavaScript, PHP, C and Go (programming language).
6) We learned what ASTs are and how they differ from Parse Trees.
7) we done interpreter by using python, while doing we also learned python and we also learned how to us some advance features in python like __init__, __str__, __repr__, pass, getattr()
8) we also got an idea of how a programming language works and how it works creating them, also gave us the opportunity to create new programming language in the future especially for those jobs that require programming language as civil engineers, we were inspired to create a new program language in this course.