CS 609: Final Project Assignment

Project Title: Building a Python Interpreter

Objective:

In this project, you will create a fully functioning interpreter using Python that can parse and execute a simple programming language. The interpreter should support basic variable assignments, arithmetic operations, and print statements. Your project will help you understand how interpreters work under the hood, including tokenization, parsing, and interpreting abstract syntax trees (AST).

Abstract Syntax Tree (AST):

An **Abstract Syntax Tree (AST)** is a data structure used in compilers and interpreters to represent the hierarchical structure of source code. It abstracts away the specific syntax of the programming language and focuses on its logical structure. Here are some key characteristics and components of an AST:

Key Characteristics of AST:

1. **Node Representation**: Each node in the AST represents a construct occurring in the source code, such as expressions, statements, or declarations.
2. **Hierarchical Structure**: The AST has a tree-like structure where the root node represents the entire program, and child nodes represent sub-expressions or statements. This reflects the nested structure of the code.
3. **Simplification**: The AST simplifies the parsing process by omitting irrelevant details such as parentheses, commas, and other syntactic elements that don't affect the semantics of the code.
4. **Semantic Information**: In addition to representing the structure of the code, the AST can also carry semantic information, such as types and variable scopes.

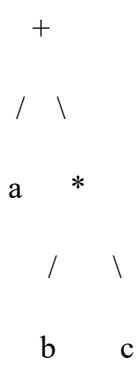
Components of an AST:

1. **Nodes**: Each node typically represents:
    - **Expressions**: Arithmetic operations, function calls, etc. (e.g., addition, multiplication).
    - **Statements**: Control flow statements like if-else, loops, and function definitions.
    - **Declarations**: Variable and function declarations.
2. **Leaf Nodes**: These nodes represent the basic values or identifiers (e.g., numbers, variable names).

3. **Subtrees**: Nodes can have child nodes, forming subtrees that represent more complex expressions or statements.

Example:

For the expression a + b * c, the AST might look like this:



Uses of AST:

- **Code Analysis**: ASTs are used for static analysis, enabling tools to analyze the structure and semantics of code without executing it.
- **Code Transformation**: ASTs facilitate code transformations, such as optimization to another language.
- **Interpretation/Compilation**: In interpreters and compilers, the AST is traversed to generate executable code or perform evaluation.

Requirements:

Your interpreter should be able to:

1. **Handle variable assignments**:
    - Example:
        - let a = 5;
        - let b = a + 3;
2. **Support arithmetic operations**:
    - Addition (+), subtraction (-), multiplication (*), and division (/)
    - Example:
        - let c = a * b;
3. **Print output**:
    - Use the print statement to display variable values.
    - Example:
        - print(a);

4. **Parse multiple statements**:
    - Your program should handle multiple let and print statements, executed sequentially.
5. **Handle whitespace and semicolons**:
    - Statements should be separated by semicolons (;).

Project Structure:

Your project should include the following components:

- **Lexer**: Tokenizes the input string into meaningful symbols like NUMBER, ID, PLUS, MINUS, MUL, etc.
- **Parser**: Constructs the AST from the tokenized input.
- **AST Nodes**: Represent the various constructs of the language (e.g., variable assignments, arithmetic expressions, print statements).
- **Interpreter**: Visits the AST and performs the actual execution (e.g., performing arithmetic operations and printing values).

Milestones:

Milestone 1: Lexical Analysis (Lexer)

- Tokenize the input into meaningful components like identifiers, numbers, operators, and keywords (let, print).
- Sample input:

```
let x = 10 + 5;
let y = x * 2;
print(x);
print(y);
```

- Expected tokens: [LET, ID, ASSIGN, NUMBER, PLUS, NUMBER, SEMI], etc.

Milestone 2: Parsing

- Implement a parser that constructs an Abstract Syntax Tree (AST) from the tokenized input.
- Sample AST:
    - AssignNode(var=x, expr=BinOpNode(left=10, op=PLUS, right=5))
    - PrintNode(expr=VarNode(x))

Milestone 3: Interpretation

- Implement the interpreter to execute the parsed AST.

- Sample execution:

```
15
30
```

Deliverables:

1. **Code**:
    - The complete interpreter implementation (`lexer.py`, `parser.py`, `interpreter.py`, `ast_nodes.py`, etc.).
    - Ensure that your code is well-structured and commented.
2. **Live Demonstration**:
    - Each group will give a live demo of their interpreter during the final project presentation.
    - Demonstrate the interpreter running sample input, showing variable assignments, arithmetic operations, and print statements.
3. **Project Report**:
    - A short report (2-3 pages) that explains the structure of your interpreter, key design decisions, and a summary of how it works.
    - Include a few example programs and their output.
4. **Group Contributions**:
    - Each student must clearly mention their contributions to the project in the report. Specify the parts of the interpreter you worked on, such as the lexer, parser, or interpreter logic.

Assessment Criteria:

Your project will be evaluated based on the following:

1. **Correctness** (40%):
    - Does the interpreter correctly handle the required features (e.g., parsing, arithmetic, print statements)?
2. **Code Quality** (20%):
    - Is the code organized, modular, and well-documented?
3. **Demonstration** (20%):
    - Quality of the live demo, including explanations and examples.
4. **Report** (20%):
    - Clarity of explanations, example programs, and insights into your implementation.

Deadline:

- The project code, report, and demo will be due by **November 30, 2024, 11:59 pm**.