Project #1: Build a Parser

CMPSC- 461

FALL-2023

Penn State University

Overview

This project provides hands-on experience with lexical analysis, syntax analysis, and abstract syntax tree (AST) construction, fundamental concepts in compiler construction and programming languages. Students are provided with starter Python classes, each modeling components of a basic interpreter.

Grammar

This language supports a basic set of grammar rules outlined below, defining the valid combinations of tokens and structures in the language.

Tokens

VARIABLE:	Represents variable identifiers and consists of alphabetic characters.
INTEGER:	Represents integer literals.
OPERATOR:	Represents arithmetic operators, specifically +, -, *, /.
ASSIGN:	Represents the assignment operator =.
SEMICOLON:	Represents the semicolon ; used to denote the end of a statement.
PARENTHESIS:	Represents open (and close) parentheses used for grouping.

Grammar Rules

Program → StatementList
StatementList →Statement StatementList | Statement
Statement → AssignmentStatement
AssignmentStatement → VARIABLE ASSIGN Expression SEMICOLON
Expression → (Expression) | Expression OPERATOR Term | Term
Term → INTEGER | VARIABLE

Example

For an input string like a = (5 + 3) * 2;

The sequence of tokens would be as follows:

VARIABLE(a) ASSIGN(=) PARENTHESIS(() INTEGER(5) OPERATOR(+) INTEGER(3) PARENTHESIS()) OPERATOR(*) INTEGER(2) SEMICOLON(;)

And the Abstract Syntax Tree (AST) would be constructed as such:

```
Assignment
        Variable: a
        Operator: *
                Operator: +
                        INTEGER: 5
                        INTEGER: 3
                INTEGER: 2
```

Classes

1. Token Class

This class models individual tokens, representing the smallest units of meaning in the input.

Attributes:
type: Denotes the type of token (e.g., INTEGER, OPERATOR, VARIABLE).
value: Holds the actual string value or content of the token.

2. Lexer Class

tokenize: Converts the input string into a list of tokens.

Attributes:
input: The input string to be analyzed.
position: The current character position in the input string.

3. Node Class

Models a node in the AST, holding the type, value, and children of each node in the tree.

Attributes:
type: Represents the type of the node (e.g. ASSIGNMENT, EXPRESSION).
value: Contains the value or content of the node.
children: A list holding the child nodes of the current node.

4. Parser Class

Converts the sequence of tokens into an AST, revealing the hierarchical structure of the program.

Attributes:
tokens: List of tokens to be parsed.
position: The current position in the list of tokens.

Each class contains several methods for performing tasks like tokenizing input and parsing tokens into an AST.

Implementation

1. Class: Lexer

Function: tokenize

Purpose: To convert the input string, representing a sequence of characters, into a list of Token objects.

Input: A string comprising characters, digits, operators, parentheses, assignment characters, semicolons, and potentially invalid characters.

Output: A list of Token objects, each representing a recognized symbol or keyword in the input string.

Implementation: Students will need to implement the logic to traverse each character in the string, recognize the type of token it represents, and create the corresponding Token object.

2. Class: Parser

Function: consume

Purpose: To read the current token and ensure it is of the expected type, moving the position to the next token.

Input: Expected token type as a string (optional).

Output: The current Token object if it matches the expected type, else it raises a syntax error. Implementation: Students will implement logic to compare the current token's type to the expected type, and progress to the next token if it matches or raise an exception if it doesn't.

Function: peek

Purpose: To examine the type and value of the current token without moving to the next token. Input: None.

Output: The current Token object.

Implementation: Students will implement logic to return the current token without changing the position in the token list.

Function: parse

Purpose: To construct the root of the Abstract Syntax Tree (AST) and iteratively parse each statement in the token list.

Input: None.

Output: The root Node object representing the entire program.

Implementation: Students will implement the logic to iteratively call the appropriate parse functions and construct the overall AST structure.

Function: parse_statement

Purpose: To determine the type of the current statement and delegate to the corresponding parse function.

Input: None.

Output: A Node object representing a parsed statement.

Implementation: Students will need to implement logic to identify the statement type and call the corresponding parse function to construct the Node object for that statement.

Function: parse_assignment

Purpose: To parse assignment statements.

Input: None.

Output: A Node object representing an assignment statement.

Implementation: Students will implement the logic to construct the Node object for an assignment statement, which involves parsing the variable, the assignment operator, and the expression.

Function: parse_expression

Purpose: To parse expressions which involve mathematical operations.

Input: None.

Output: A Node object representing a mathematical expression.

Implementation: Students will construct the Node object for an expression, which could involve multiple terms and operators, and handle the construction of a corresponding subtree for each operation.

Function: parse_term

Purpose: To parse individual terms in an expression, which could be integers, variables, or nested expressions within parentheses.

Input: None.

Output: A Node object representing an individual term in an expression.

Implementation: Students will implement logic to recognize and construct the Node object for a term, whether it's an integer, variable, or a nested expression, by delegating to the parse_expression function when necessary.

Testing

These test cases aim to assess the functionalities of the Lexer and Parser classes. They primarily focus on the capability of these classes to appropriately tokenize input strings and construct Abstract Syntax Trees (AST) based on those tokens.

Run the python tester file by executing the following command to test your parser implementation.

```
Python3 test_parser.py
```

Here is a brief overview of each test case:

Test: Lexer
This test evaluates whether the Lexer class can correctly tokenize a given single line of input code.

Test: Parser Valid Input
This test assesses if the Parser can correctly form an AST from a single line of valid input code, consisting of a straightforward assignment.

Test: Parser Multi Line
This case checks Parser's ability to create an accurate AST for multi-line input, which includes several variable assignments and mathematical operations.

Test: Parser Multiple Operations
This test is intended to verify the Parser's ability to build a correct AST when presented with multiple operations, such as addition, multiplication, and subtraction spread across multiple lines.

Test: Parser Nested Expressions
This test ensures that the Parser can accurately handle and represent nested mathematical expressions within parentheses in the AST.

Test: Parser Invalid Assignment

This test makes sure that the Parser appropriately raises an exception when it encounters invalid assignment, where a number is assigned to a variable.

Test: Parser Invalid Character
This test verifies whether the Lexer raises the correct exception when it comes across an invalid character during the tokenization process.

Test: Parser Invalid Operation
This test guarantees that the Parser raises an exception when dealing with an incomplete operation, where an operator is followed by a closing parenthesis without an operand in between.

Test: Parser Invalid Inputs
This test determines whether the Parser can handle multiple lines of invalid inputs and raise the appropriate exception.

Test: Missing Semicolon
This test ensures that the Parser raises an exception when the input is missing a semicolon, which is necessary to indicate the end of a statement.

Grading of Test Cases

The grading of the test cases in this project is done based on the number of test cases passed. There are several test cases included in this project, and each is aimed at testing different aspects and functionalities of the Lexer, Parser, and Node implementations that the students will complete.

The grading is as follows:

- For every test case passed, the students earn a 10% of the total score.
- The total score available for passing all the test cases is 100 points.
- If a student passes all the test cases, they receive the full 100 points.
- If any test case fails, the 10 points corresponding to that test case are deducted from the total score.