**PyCompile Documentation**

-- Introduction

PyCompile is a simple and intuitive compiler for a custom programming language. It takes source code as input, tokenizes it, parses it into a syntax tree, and performs semantic analysis. The project is designed to be educational, demonstrating how a compiler works from source code to execution.

-- Features

- Lexical Analysis: Tokenizes the input source code.

- Syntax Analysis: Parses tokens into a syntax tree.

- Semantic Analysis: Checks for semantic errors.

- Intermediate Representation: Generates an intermediate representation of the code.

- Code Generation: Generates low-level code from the intermediate representation.

- Assembly Generation: Converts low-level code to assembly instructions.

- Virtual Machine: Executes the generated assembly instructions.

- Command-Line Interface: Simple interface to compile and execute source code files.

-- Installation

To install and set up PyCompile, follow these steps:

1. Clone the repository:

```bash

git clone https://github.com/Devansh-46/PyCompile.git

```

2. Navigate to the project directory:

```bash

cd PyCompile

```

3. Install the required dependencies:

```bash

pip install -r requirements.txt

```

-- Usage

To use the compiler, run the `main.py` script with the path to your source code file as an argument:

```bash

python main.py <path\_to\_source\_code>

```

-- Examples

Create a file named `example.src` with the following content:

```python

def add(x, y) {

x + y

}

z = add(5, 3);

print(z);

```

Run the compiler:

```bash

python main.py example.src

```

-- Project Structure

```

PyCompile/

├── main.py

├── lexer.py

├── parser.py

├── semantic\_analyzer.py

├── ir\_generator.py

├── code\_generator.py

├── assembly\_generator.py

├── vm.py

├── node.py

├── mytoken.py

└── tests/

├── test\_lexer.py

├── test\_parser.py

├── test\_semantic\_analyzer.py

├── test\_ir\_generator.py

├── test\_code\_generator.py

├── test\_assembly\_generator.py

└── test\_vm.py

```

- main.py: Entry point for the compiler. Handles reading input files and orchestrating the compilation process.

- lexer.py: Handles lexical analysis, converting source code into tokens.

- parser.py: Handles syntax analysis, converting tokens into a syntax tree.

- semantic\_analyzer.py: Performs semantic checks on the syntax tree.

- ir\_generator.py: Generates an intermediate representation of the code.

- code\_generator.py: Generates low-level code from the intermediate representation.

- assembly\_generator.py: Converts low-level code to assembly instructions.

- vm.py: A virtual machine to execute the generated assembly instructions.

- node.py: Defines the Node class used in the syntax tree.

- mytoken.py: Defines the Token class used in lexical analysis.

- tests/: Contains test cases for various components of the compiler.

-- Classes and Methods

--- Lexer

The `Lexer` class is responsible for tokenizing the input source code.

- Methods:

- `tokenize(input)`: Tokenizes the input source code and returns a list of tokens.

--- Parser

The `Parser` class is responsible for parsing the tokens into a syntax tree.

- Methods:

- `\_\_init\_\_(self, tokens)`: Initializes the parser with a list of tokens.

- `parse(self)`: Parses the tokens and returns the syntax tree.

--- Semantic Analyzer

The `SemanticAnalyzer` class is responsible for performing semantic analysis on the syntax tree.

- Methods:

- `\_\_init\_\_(self, syntax\_tree)`: Initializes the analyzer with the syntax tree.

- `analyze(self)`: Performs semantic analysis on the syntax tree.

--- IR Generator

The `IRGenerator` class is responsible for generating an intermediate representation (IR) of the code.

- Methods:

- `\_\_init\_\_(self, syntax\_tree)`: Initializes the generator with the syntax tree.

- `generate(self)`: Generates the intermediate representation.

--- Code Generator

The `CodeGenerator` class is responsible for generating low-level code from the IR.

- Methods:

- `\_\_init\_\_(self, ir)`: Initializes the generator with the IR.

- `generate(self)`: Generates the low-level code.

--- Assembly Generator

The `AssemblyGenerator` class is responsible for converting low-level code to assembly instructions.

- Methods:

- `\_\_init\_\_(self, low\_level\_code)`: Initializes the generator with the low-level code.

- `generate(self)`: Generates the assembly instructions.

--- Virtual Machine

The `VirtualMachine` class is responsible for executing the generated assembly instructions.

- Methods:

- `\_\_init\_\_(self, assembly\_code)`: Initializes the VM with the assembly code.

- `run(self)`: Executes the assembly code.

--- Node

The `Node` class represents nodes in the syntax tree.

- Attributes:

- `type`: The type of the node (e.g., 'binary\_expression', 'number').

- `value`: The value of the node (e.g., an operator or a number).

- `children`: A list of child nodes.

--- Token

The `Token` class represents tokens in the lexical analysis stage.

- Attributes:

- `type`: The type of the token (e.g., 'NUMBER', 'IDENTIFIER').

- `value`: The value of the token (e.g., '5', 'add').