Title: Python Compiler

Team Members: Devanshi Saraf, Kshitij Kumar, Aditi Talpallikar

---

Abstract

The Python Compiler project aims to develop a compiler that translates Python code into machine-executable instructions. The primary objective is to implement fundamental phases of compilation, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. The compiler will support a subset of Python's language features, including variable assignments, conditional statements, loops, function calls, and basic object-oriented programming constructs.

Python is a dynamically typed language with an extensive grammar, consisting of identifiers, keywords, literals, operators, delimiters, and comments. Our compiler will focus on handling essential tokens such as:

- **Identifiers:** Variable and function names.
- **Keywords:** Reserved words like `if`, `else`, `while`, `def`, `return`, etc.
- **Literals:** Numeric values, strings, and boolean literals (`True`, `False`).
- **Operators:** Arithmetic (`+`, `-`, `*`, `/`), relational (`==`, `!=`, `>`), logical (`and`, `or`, `not`), and bitwise (`&`, `|`, `^`).
- **Delimiters:** Parentheses, brackets, braces, commas, colons, indentation-based blocks.

The compilation process will be divided into the following key phases:

Compiler Phases

1. **Lexical Analysis:**
    - Tokenizes the input source code into meaningful units.
    - Handles whitespace, indentation, and comments.
    - Implements symbol table management for identifiers.
2. **Syntax Analysis (Parsing):**
    - Checks the structural correctness of tokens based on Python’s grammar.
    - Constructs a Parse Tree or Abstract Syntax Tree (AST).
    - Implements recursive descent parsing for efficient analysis.
3. **Semantic Analysis:**
    - Performs type checking and ensures correct variable usage.
    - Validates function calls and scope resolution.
    - Implements error handling and reporting.
4. **Intermediate Code Generation:**

    - Converts the AST into an intermediate representation (IR), such as three-address code (TAC) or static single assignment (SSA).
    - Optimizes temporary variable usage and control flow.
5. **Optimization:**
    - Implements basic optimizations like constant folding, loop unrolling, and dead code elimination.
    - Reduces redundant computations and memory usage.
6. **Code Generation:**
    - Converts IR to low-level bytecode or machine code for execution.
    - Generates optimized assembly-level instructions for efficient performance.

Development Timelines and Phases

The development of this compiler is planned over multiple phases, divided as follows:

Phase 1: Research and Planning

- Study Python's grammar and token structure.
- Design compiler architecture and determine toolsets.
- Draft the high-level execution pipeline.

Phase 2: Lexical & Syntax Analysis

- Implement tokenization using a lexical analyzer.
- Develop parsing strategies and AST construction.
- Test basic parsing of Python expressions.

Phase 3: Semantic Analysis & Intermediate Code Generation

- Implement type checking and symbol table management.
- Generate an intermediate representation (IR).
- Develop error handling and debugging mechanisms.

Phase 4: Optimization & Code Generation

- Implement basic optimizations for performance improvement.
- Develop backend code generation targeting bytecode or assembly.
- Integrate compiler components and perform comprehensive testing.

Phase 5: Testing & Finalization

- Conduct extensive testing with various Python programs.

- Optimize compiler efficiency and document findings.
- Prepare final documentation and project presentation.

Expected Output

1. A command-line-based Python compiler capable of processing scripts with fundamental Python constructs.
2. A functional pipeline showcasing lexical analysis, parsing, semantic checks, and code generation.
3. Step-wise breakdown of input code through different compilation stages.
4. Error reporting and debugging assistance for syntax and semantic errors.
5. Generation of optimized intermediate code or direct executable output.
6. Implementation of basic optimization strategies for improving compilation efficiency.
7. Comprehensive documentation on the architecture, functionality, and test results of the compiler.

This project will serve as an educational tool for understanding compiler design principles and practical implementation of language processing techniques. By the end of the development cycle, we aim to deliver a fully functional Python compiler with an efficient execution pipeline.