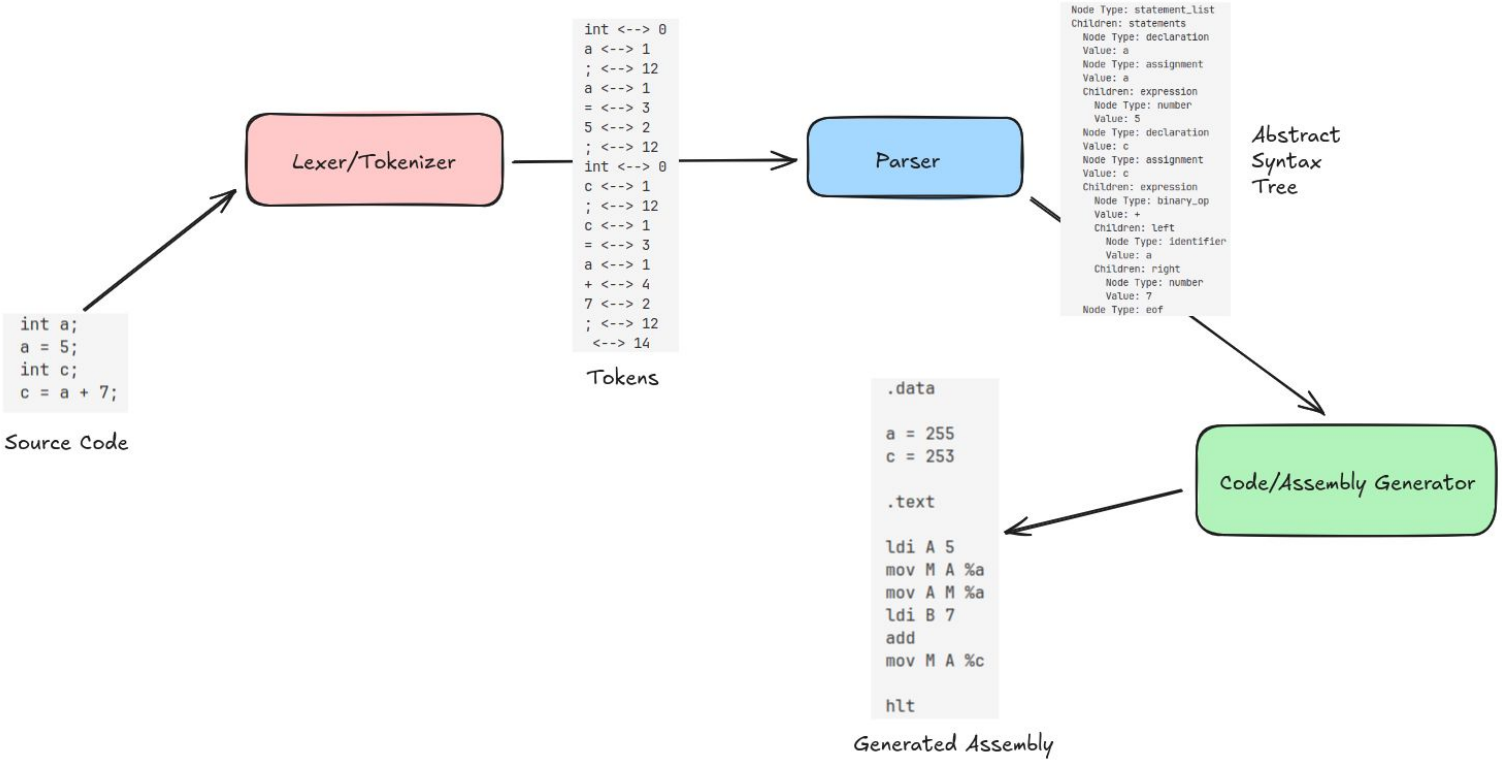
Simple Lang

Assignment overview & explanation

Maharshi Basu

Complete Compiler Flow



Lexer/Tokenizer

`Token` Structure:

- Represents a lexical unit with two attributes:
  - `text`: The string representation of the token.
  - `type`: The classification (e.g., `TOKEN_IF`, `TOKEN_NUMBER`).

`getNextToken(std::ifstream &fileStream)`:

- Reads characters from the input file to generate the next token.
- **Token Types**:
  - **Keywords**: E.g., `if`, `int`.
  - **Identifiers**: Alphanumeric strings not matching keywords.
  - **Numbers**: Sequences of digits.
  - **Operators**: E.g., `+`, `-`, `=`, `==`.
  - **Special Symbols**: E.g., `(`, `)`, `{`, `}`, `;`.
- **Process**:
  - Skips whitespace.
  - Reads and classifies alphanumeric strings, numbers, or single-character symbols.
  - Uses `fileStream.unget()` to backtrack when a character doesn't match the current token type.

`tokenizeFile(const std::string filePath)`:

- Opens the input file and checks for errors.
- Iteratively calls `getNextToken()` to tokenize the entire file.
- Returns a vector containing all tokens.

Parser

Parsing Functions:

- **Program Parsing**: `parseProgram` handles the root `<program>` grammar, which consists of a `<statement_list>`.
- **Statements**: Supports `<declaration>`, `<assignment>`, and `<conditional>` parsing.

Expression Handling:

- Parses arithmetic expressions (`+`, `-`) and operands (`<identifier>`, `<number>`).
- Recursively constructs AST nodes for binary operations.

Conditionals:

- Parses `if` statements with conditions like `a == b` and block structures enclosed by `{}`.

Error Handling:

- Validates expected tokens like `(`, `{`, `;`, and handles unexpected tokens with meaningful error messages.

AST Structure:

- Nodes have a `type`, optional `value`, and `children` containing further parsed nodes.

Code Generation

Declaration Handling:

- The `generateDeclaration` function adds variables to a symbol table and assigns them memory addresses.
- Variables are stored in decreasing memory addresses starting from 255.

Expression Handling:

- The `generateExpression` function evaluates numeric literals, variables, and binary operations (`+`, `-`).
- Binary operations involve loading operands into registers (`A` and `B`) and performing operations like `add` or `sub`.

Assignment Handling:

- The `generateAssignment` function generates code to evaluate the right-hand side expression and store the result in the variable's memory address.

contd.

Code Generation

Condition Handling:

- The `generateCondition` function compares two expressions using the `==` operator and prepares the code for a conditional jump.

Conditional Statements:

- The `generateIf` function processes `if` statements.
- It uses unique labels to handle branching and executes a list of statements in the body of the `if`.

Program-Level Code Generation:

- The `generateProgram` function processes the AST's top-level statements, generating `.data` and `.text` sections.
- Handles declarations first to populate the symbol table and then processes assignments and conditionals.

Limitations

- **Indentation Handling in If-Blocks**:

  The compiler has limitations in parsing indentation levels within if-blocks. Specifically, it misinterprets the closing curly brace (`}`) of an if-block as the end of the program.

- **Limited Support for Expressions**:

  The compiler currently supports only simple binary operations. Complex expressions, such as `a = b + c - 4;`, are not supported and will result in a parsing error.

- **Equality-Only Conditions in If-Statements**:

  The compiler supports only equality (`==`) conditions in if-statements. Other relational operators like `<`, `>`, `<=`, or `>=` are not yet implemented.

- **Lack of Else Clauses**:

  Else clauses are not supported in conditional statements, limiting the compiler to handling only the `if` branch.

Conclusion

This project involved creating a compiler capable of translating a simple programming language into assembly code for an 8-bit CPU. It covers the end-to-end process, from parsing to code generation, providing valuable insights into compiler design and assembly language.