Title: GW-BASIC *Programming language(C++)*

Parser Module

PRESENTED BY : SUVETHA MANIKANDAN
DATE:30 JULY 2025

Project Overview

- Building a **GW-BASIC interpreter** in modern **C++17/20**
- Part of a larger 64-bit OS project
- Parser handles syntax analysis and AST generation
- Modular structure: lexer, parser, AST, runtime

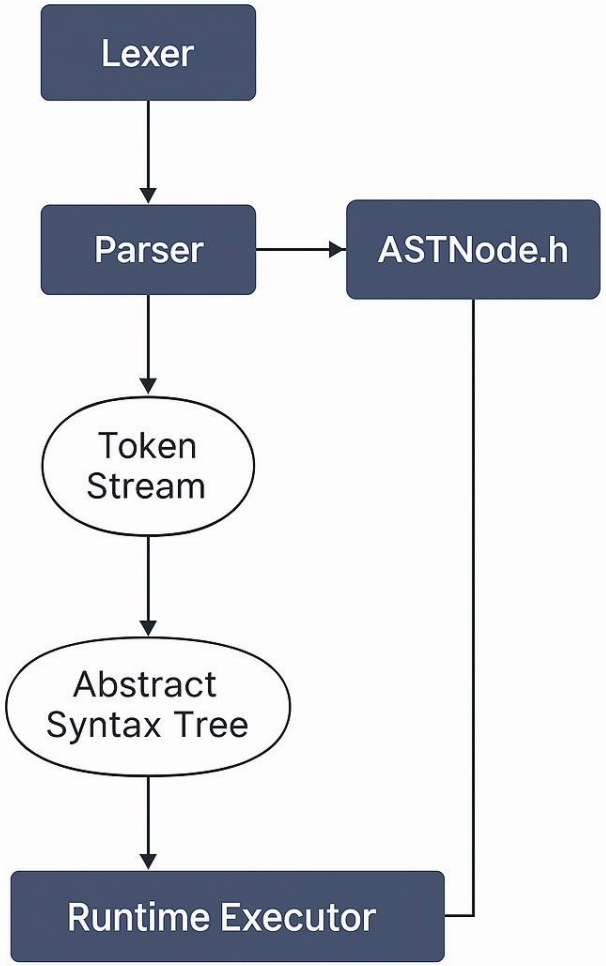
Introduction

- GW-BASIC is a classic BASIC interpreter originally used on DOS systems.This project aims to rebuild a modern GW-BASIC interpreter using C++17/20, designed to run in a 64-bit environment.
- The parser module is a core component that processes tokenized source code and generates an Abstract Syntax Tree (AST).
- This AST is then used by the interpreter or runtime engine to execute programs.
- The parser is built using a recursive descent strategy and supports most major GW-BASIC constructs like PRINT, LET, IF, FOR, etc.

Role of the Parser

- Converts token stream (from lexer) into an Abstract Syntax Tree (AST)
- Implements recursive descent parsing
- Supports major GW-BASIC statements:
- PRINT, LET, IF, FOR, WHILE, DO, GOTO, etc.

GWBASIC Parser Module

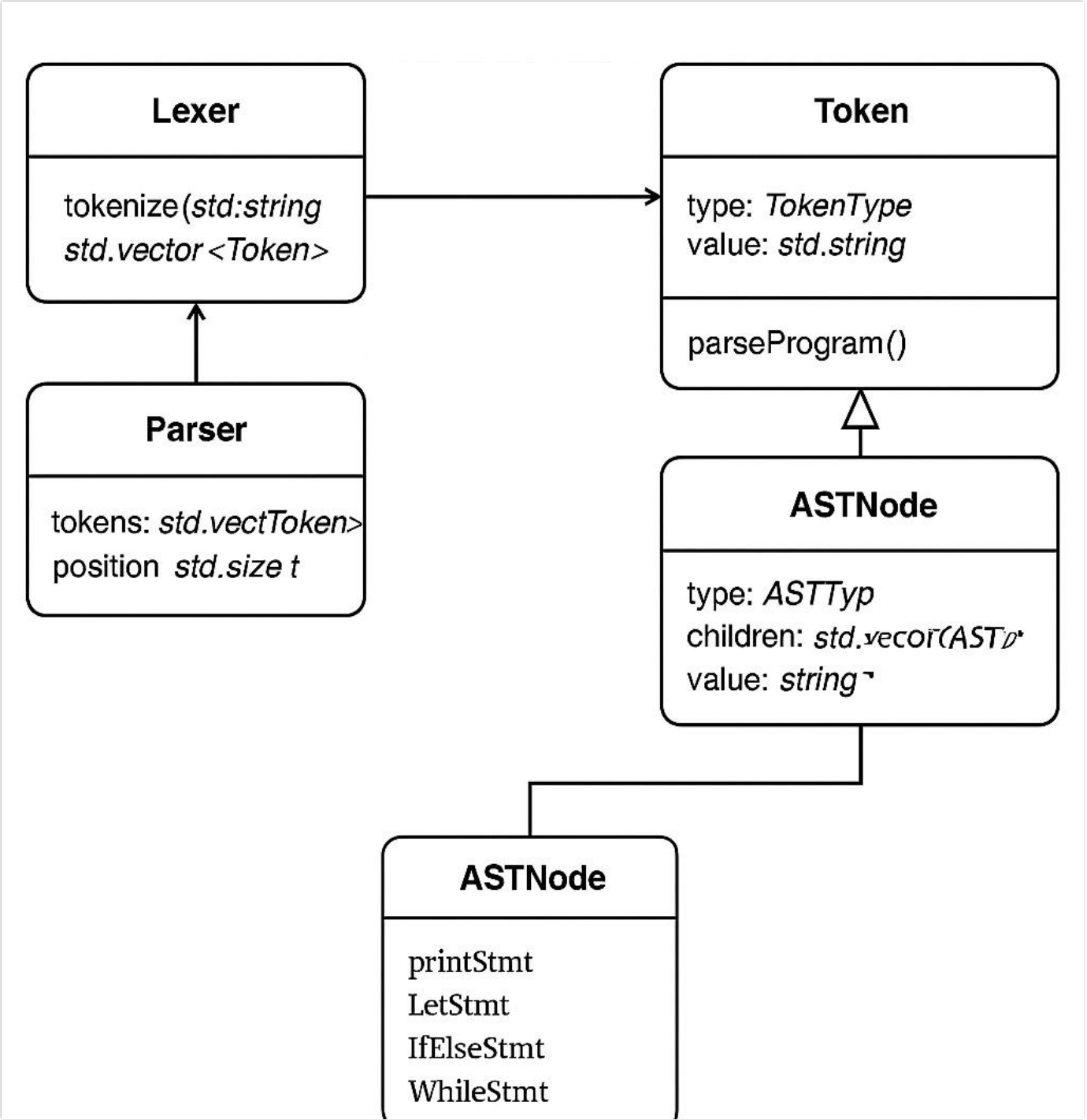


- 🔷 LexerInput: std::string (e.g., "10 PRINT \"HELLO\"").Output: std::vector<Token> – List of tokens with type and value.
- ParserInput: std::vector<Token>Output: ASTNode* – Builds the Abstract Syntax Tree using recursive descent.Uses ASTNode.h to define node types like PrintStmt, LetStmt, etc.
- Token Stream → Abstract Syntax TreeIntermediate phase where raw tokens are converted into a hierarchical structure.
- Runtime ExecutorInput: ASTNode*Executes code like PRINT, LET, IF, FOR, etc.Output: Runtime behavior: printed output, variable updates, and control flow.

Key Parser Components

- parser.h - Function declarations for parsing logic.
- parser.cpp - Recursive descent parsing implementation.
- ASTNode.h - AST node types and structure definitions.
- main.cpp - Testing and integration entry point.

CLASS DIAGRAM



Lexer → Parser → Runtime Flow

1. Lexer

- • Input: std::string (source code line, e.g., "10 PRINT \"HELLO\"")
- • Output: std::vector<Token>
- – Token { TokenType type; std::string value; }

2. Parser

- • Input: std::vector<Token>
- • Output: ASTNode* (Abstract Syntax Tree)
- – ASTNode { ASTType type; std::string value; std::vector<ASTNode*> children; }

3. Runtime Executor

- • Input: ASTNode*
- • Output: Executes the code (e.g., prints output, updates variables)
- – Runtime side effects, no return value (void)

ASTNode.h – Abstract Syntax Tree

- Defines the structure of the AST (Abstract Syntax Tree).
- Key Components:
- enum class ASTType – Lists all supported statement and expression node types.
- class ASTNode – Base class for all AST nodes.
- Members:
- – ASTType type: Enum to identify node kind.
- – std::vector<ASTNode*> children: Child nodes in tree.
- – std::string value: Literal or variable name (if any).
- May include line number, operator, or additional metadata.

parser.h – Parser Declarations

- Declares the Parser class and public parsing functions.
- Key Components:
- • class Parser – Maintains token list and current index.
- • Constructor: Parser(const std::vector<Token>& tokens);
- • Core methods:
- – ASTNode* parseProgram();
- – ASTNode* parseStatement();
- – ASTNode* parseExpression();
- • Parses control flow, expressions, I/O, math functions.
- • Header includes only interfaces and essential includes.

parser.cpp – Parser Implementation

- Implements recursive descent parsing logic.
- Key Functions:
- • parseProgram(): Parses entire BASIC program.
- • parseStatement(): Parses PRINT, LET, IF, FOR, etc.
- • parseExpression(): Handles literals, identifiers, math.
- • match(), expect(): Token consumption helpers.
- • parseIfStatement(), parseForLoop(), parsePrintStatement(), etc.
- • Builds and returns ASTNode trees for execution.
- • Handles basic error reporting (unexpected tokens, missing parts).

Error Handling

Custom syntax error messages with line numbers.

Checks for missing keywords, unbalanced parentheses, invalid tokens.

Graceful failure and fallback in ambiguous cases.

Sample Parsing Flow

- -- Input: 200 PRINT A + B
- Program
- PrintStmt
- BinOp '+'
- Ident A
- Ident B

- -- Input: 210 LET C = A * B
- Program
- LetStmt C
- BinOp '*'
- Ident A
- Ident B

- -- Input: 220 PRINT SQR(C)
- Program
- PrintStmt
- MathFunc SQR
- Ident C

Standard Libraries

- <string> – Used for storing source lines, identifiers, and literals.
- <vector> – Used for storing dynamic lists of tokens and AST child nodes.
- <map> – (If used) for managing variable-value pairs in symbol tables.
- <iostream> – For printing debug output and AST visualization.

C++ *CONCEPTS*

- **Classes & Structs** – ASTNode, Token, and Parser are implemented as C++ classes.
- **Enums** – TokenType, ASTType provide meaningful named constants.
- **Pointers** – Used for dynamic AST node management (ASTNode*).
- **Recursive Functions** – Recursive descent parsing is implemented using recursion.
- **Encapsulation** – Each class manages its own state and logic.
- **Constructor Initialization** – Used in Parser to initialize with token input cleanly.
- **Namespaces (optional)** – May be used to group lexer/parser/runtime logically.

Challenges Faced

- Mapping GW-BASIC's syntax to a clean AST design.
- Handling optional syntax variations (e.g., IF THEN, FOR TO STEP).
- Managing parser state without smart pointers or advanced C++ features.
- Debugging complex expressions with nested operations.
- Ensuring error messages are meaningful and user-friendly.

Conclusion

- The GW-BASIC parser project demonstrates how a classic language can be reinterpreted with modern tools in a structured, modular way. It reinforces core programming concepts like tokenization, recursion, tree structures, and modular design, making it an excellent project for learning language implementation fundamentals.

THANK YOU!