**Design Document: GW-BASIC Parser Module**

**1. Overview**

The GW-BASIC parser module converts a sequence of tokens (produced by a lexer) into an Abstract Syntax Tree (AST). This tree represents the syntactic structure of a BASIC program and forms the basis for subsequent interpretation or compilation phases.

**2. Goals**

* Implement a recursive-descent parser for a subset of GW-BASIC
* Support common statements (LET, PRINT, IF/THEN, FOR/NEXT, WHILE/WEND, DO/LOOP, GOSUB, RETURN, etc.)
* Build a well-structured and memory-safe AST hierarchy
* Provide descriptive error handling

**3. Architecture**

**3.1 Key Components**

* Parser: Entry point and main control for parsing logic
* ASTNode.h: Contains all AST node classes
* Token.h: Token structure used for parsing input
* parser.cpp: Implements parsing logic using recursive-descent techniques

**3.2 Parser Flow**

Lexer Output --> [Tokens] --> Parser --> [AST Root (ProgramNode)]

**3.3 Parsing Strategy**

Recursive-descent parsing is used, with each grammar rule implemented as a C++ method:

* parseProgram() -> full program
* parseStatement() -> single statement
* parseExpression() -> arithmetic/comparison expression
* parseTerm(), parseFactor() -> lower precedence rules

**4. AST Structure**

**4.1 Base**

class ASTNode {

public:

virtual ~ASTNode() {}

virtual ASTType type() const = 0;

};

**4.2 Statement Nodes**

* ProgramNode: contains a list of statements
* LetNode, PrintNode, InputNode, IfElseNode, ForNode, NextNode, GotoNode, ReturnNode, StopNode, RemNode, etc.
* CommandNode: generic command for keywords like SYSTEM, DIR, CONT, etc.

**4.3 Expression Nodes**

* BinOpNode: binary operations (+, -, \*, /, <, >, etc.)
* NumberNode, StringNode, IdentNode
* MathFuncNode: built-in math functions (SIN, COS, LOG, etc.)

**5. Token Handling**

* peek() - returns current token without consuming
* get() - consumes and returns the next token
* match() - utility to check and consume a token based on type/value

**6. Parsing Functions**

**6.1 parseProgram()**

* Skips line numbers
* Parses statement list until end-of-line
* Handles multiple statements separated by :

**6.2 parseStatement()**

Dispatches based on the current keyword:

if (match(TokenType::Keyword, "PRINT")) return parsePrint();

if (match(TokenType::Keyword, "LET")) return parseLet();

if (match(TokenType::Keyword, "IF")) return parseIf();

// ... more rules

Supports implicit assignments, e.g. X = 5

**6.3 parseExpression()**

Handles operator precedence and expression trees:

* Binary comparisons (<, >, =, etc.)
* Delegates to parseTerm() and parseFactor()
* Parses math function calls (e.g. SIN(x)) using parseMathFunc()

**7. Error Handling**

* Throws std::runtime\_error on invalid syntax
* Custom messages help trace source of failure
* Uses synchronization patterns (e.g. skip tokens until : or END\_OF\_LINE)

**8. Sample Grammar (Subset)**

program ::= { line }

line ::= [Number] statement { ':' statement }

statement ::= LET | PRINT | IF | FOR | GOSUB | ...

expression ::= term { ('+' | '-' | '<' | '>' | '=') term }

term ::= factor { ('\*' | '/') factor }

factor ::= NUMBER | IDENTIFIER | STRING | '(' expression ')' | mathFunc

**9. Supported Statements**

* LET, PRINT, INPUT
* IF/THEN/ELSE
* FOR/NEXT
* WHILE/WEND
* DO/LOOP [WHILE | UNTIL]
* GOSUB/RETURN
* STOP/END
* DATA/READ
* REM comments
* ON ERROR GOTO
* FIELD definitions
* SYSTEM, CONT, EDIT, CLEAR, DIR, SCREEN, and other shell commands

**10. Testing Strategy**

* **Unit Tests** for all parse\*() methods
* **Integration Tests** using real GW-BASIC programs
* Error Injection: invalid syntax, unknown keywords, unmatched expressions
* Visualization: ASTPrinter or dump tool for output verification

**11. Extensibility**

* Add semantic analysis phase post-AST generation
* Hook in an interpreter or bytecode generator
* Expand AST types for user-defined functions, subroutines

**12. Conclusion**

This parser forms the backbone of a GW-BASIC interpreter/compiler. It cleanly separates syntax handling from semantics and provides structured error diagnostics. It is designed to be extensible, readable, and robust for both educational and hobbyist OS/interpreter development.