Technical Report: Lab 2B - Abstract Syntax Tree Parser

Author: Jaskaran Singh

Date: December 2025

1. Introduction

This report details the design and implementation of a parser for a C-like programming language. The system utilizes **Flex** for lexical analysis and **Bison** for syntax analysis to generate an Abstract Syntax Tree (AST). Beyond basic parsing, the project implements error recovery, support for function definitions, and a semantic analysis phase to ensure variables are declared before use.

2. Language Features

The implemented language supports the following constructs:

- **Variable Management:** Declarations with optional initializers and assignment statements.
- **Control Flow:** `if-else` conditional branching and `while` loops.
- **Functions:** Definition of functions using the `func` keyword, `return` statements, and function calls.
- **Expressions:** Complex arithmetic with operator precedence and logical comparisons.
- **Comments:** Support for both single-line (`//`) and multi-line (`/* */`) comments.

3. Design Methodology

3.1 Lexical Analysis (Flex)

The lexer transforms the raw input stream into a sequence of tokens.

- **Regular Expressions:** Used to identify keywords, identifiers, and literals.
- **Whitespace & Comments:** Handled at the lexical level. The lexer is configured to silently discard these tokens to simplify the grammar.
- **Token Priority:** Keywords such as `if` and `while` are matched before general identifiers to prevent misidentification.

3.2 Syntax Analysis (Bison)

The parser uses a Context-Free Grammar (CFG) to validate the structure of the token stream.

- **Stratified Grammar:** To enforce mathematical operator precedence (PEMDAS) without ambiguity, the grammar is stratified into levels: *Expression → Equality → Comparison → Term → Factor → Primary*.
- **Error Recovery:** The grammar includes the `error` token within statement rules. This allows the parser to skip malformed lines and synchronize at the next semicolon, enabling the detection of multiple errors in a single run.

3.3 Abstract Syntax Tree (AST) Structure

The AST is built recursively during the reduction phase of parsing. Each node is an `ASTNode` structure containing:

- **Type:** Identifies the node (e.g., `NODE_IF`, `NODE_BIN_OP`).
- **Pointers:** A "multi-child" system using `left`, `right`, and `else_branch`.
- **Next Pointer:** A sibling pointer used to maintain a linked list of statements (Script support).

4. Semantic Analysis

While the parser confirms the **structure** is correct, it does not naturally know if a variable exists. We implemented a **Symbol Table** and a post-parse traversal:

1. **Collection:** As the AST is traversed, `NODE_VAR_DECL` nodes add the variable identifier to the symbol table.
2. **Validation:** When a `NODE_VAR` or `NODE_ASSIGN` node is encountered, the symbol table is queried.
3. **Enforcement:** If a variable is used without a prior declaration, a **Semantic Error** is raised, and execution halts.

5. Grammar Specification (BNF)

The following represents the core formal grammar rules implemented:

```
<program>        ::= <statement_list>
<statement_list> ::= <statement> | <statement> <statement_list>
<statement>      ::= <var_decl> | <assignment> | <if_stmt> | <while_stmt>
                   | <func_def> | <return_stmt> | <block> | error ';'

<expression>     ::= <equality>
<equality>       ::= <comparison> ( '==' | '!=' ) <comparison> | <comparison>
<comparison>     ::= <term> ( '<' | '>' | '<=' | '>=' ) <term> | <term>
<term>           ::= <factor> ( '+' | '-' ) <factor> | <factor>
<factor>         ::= <unary> ( '*' | '/' ) <unary> | <unary>
<unary>          ::= '-' <primary> | <primary>
<primary>        ::= TOK_NUM | TOK_ID | '(' <expression> ')' | <func_call>
```

6. Test Results and Analysis

6.1 Valid Test Cases

The parser was tested against 15 valid scripts (e.g., `arithmetic.txt`, `full_program.txt`). In all cases, the parser generated a correct, indented AST mirroring the logical nesting of the source code.

6.2 Invalid Test Cases

A suite of 15 invalid test cases was categorized into three failure tiers to verify the robustness of the compiler's error-handling pipeline.

A. Lexical Errors (Lexer Level)

The lexer was tested with unauthorized characters and malformed tokens.

- **Case Study (11_invalid_char.txt):** Input containing symbols like @ or ^.
- **Result:** The lexer's "catch-all" rule successfully identified the character, printed an "Unexpected character" message, and terminated to prevent corrupted tokens from reaching the parser.

B. Syntax Errors (Parser Level)

Using Bison's error token, the parser was tested for its ability to "sync" after a failure.

- **Case Study (05_bad_assignment.txt):** Input like x = ;.
- **Result:** The parser identified a syntax error, triggered the error recovery routine, and skipped to the next ;. This allowed the parser to continue validating the rest of the file instead of crashing on the first mistake.

C. Semantic Errors (Symbol Table Level)

These tests verified the context-sensitive logic implemented in main.c.

- Case Study (01_undef_var.txt): ```c
  x = 10;
  var x = 5;
- **Result:** While the syntax is technically valid (Variable = Number), the **Semantic Analyzer** identified that x was used at line 1 but not declared until line 2. The program successfully caught this and threw: SEMANTIC ERROR: Variable 'x' used before declaration.

6.3 Sample test cases provided by mentor

A suite of 15 test cases which were provided in Lab3-Testcases.pdf

7. Conclusion

The Cornerstone Parser successfully meets all functional and non-functional requirements. By implementing optional extensions such as function support and error recovery, the tool provides a robust foundation for a full compiler backend. The addition of a semantic check ensures that the resulting AST is not only syntactically valid but also logically sound according to the language rules.