

Cornerstone Project

LAB 3

Parser with Flex and Bison

Technical Report

Submitted by: Monit Vaghela (2025MCS2117)
Anunay Sharma (2025MCS2095)

Submitted to: Prof. Kolin Paul

Date: December 24, 2025

1 Problem Description

This assignment involves the design and implementation of a parser for a simple imperative programming language using Flex and Bison. The parser performs lexical analysis to tokenize source code and syntactic analysis to validate program structure and construct an Abstract Syntax Tree (AST).

Key Objectives:

- Implement a lexical analyzer using Flex for tokenization.
- Define a context-free grammar using Bison for syntax validation.
- Construct an Abstract Syntax Tree to represent program structure.
- Support variable declarations, assignments, arithmetic expressions, control flow (if-else), and loops (while).
- Enforce variable declaration before use through symbol table management.
- Provide clear error messages and ensure robust error handling.

2 Language Description

A minimal imperative, C-like language supporting lexical analysis, parsing, and AST construction.

2.1 Supported Constructs

Variable Declarations

```
var x;  
var y = 5;
```

Assignments

```
x = x + 1;
```

Expressions

- Arithmetic: +, -, *, /
- Relational: <, >, <=, >=
- Equality: ==, !=
- Unary: +, -
- Parentheses for grouping: (x + 2) * 3

Conditionals

```
if (x > 0)  
    x = x - 1;  
else  
    x = 0;
```

Loops

```
while (i < 10)  
    i = i + 1;
```

Blocks (introduce new scope)

```
{  
    var x = 1;  
    x = x + 1;  
}
```

Literals Integer literals: 10

3 Lexer Rules and Tokenization Strategy

The lexical analyzer is responsible for converting the input source code into a stream of tokens that can be consumed by the parser.

3.1 Keywords

The following reserved keywords are recognized by the lexer:

- `var`
- `if`
- `else`
- `while`

Keywords are matched before identifiers to avoid ambiguity.

3.2 Identifiers

Identifiers follow conventional programming language rules:

$$\text{IDENTIFIER} \rightarrow [a-zA-Z_][a-zA-Z0-9_]*$$

Identifiers are used as variable names and are validated against the symbol table during parsing.

3.3 Literals

Only integer literals are supported:

$$\text{INTEGER} \rightarrow [0-9]^+$$

3.4 Operators and Delimiters

The lexer recognizes the following operators:

- Arithmetic: `+`, `-`, `*`, `/`
- Relational: `<`, `>`, `<=`, `>=`
- Equality: `==`, `!=`
- Assignment: `=`

Delimiters include parentheses `()`, braces `{ }`, and the semicolon `;`.

3.5 Tokenization Strategy

Whitespace and comments are ignored. Each token carries semantic information using `yylval`, and line numbers are tracked using `yylineno` to enable precise error reporting.

4 Grammar Design and Operator Precedence

4.1 Grammar Overview

The grammar is context-free and unambiguous. It is structured hierarchically to enforce operator precedence without relying on precedence declarations.

At the highest level:

$$\text{program} \rightarrow \text{stmt_list}$$

Statements include variable declarations, assignments, conditional statements, loops, blocks, and empty statements.

4.2 Operator Precedence

Operator precedence is enforced through layered non-terminals as shown below (from highest to lowest precedence):

Level	Operators
Primary	literals, identifiers, parentheses
Unary	+, -
Multiplicative	*, /
Additive	+, -
Relational	<, >, <=, >=
Equality	==, !=

All binary operators are left-associative, which is achieved using left-recursive grammar rules.

5 AST Structure and Construction

5.1 AST Design

The Abstract Syntax Tree (AST) represents the logical structure of the program independently of its concrete syntax. Each AST node contains:

- A node type identifying the construct
- Pointers to child nodes
- Source line number information

5.2 AST Node Categories

The AST includes the following categories of nodes:

- Block nodes for scoped statement lists
- Variable declaration nodes with optional initialization
- Assignment nodes
- Conditional nodes for `if` and `if-else`
- Loop nodes for `while`
- Expression nodes for unary and binary operations

5.3 Construction Strategy

AST nodes are constructed during parsing using semantic actions. Each grammar rule produces an AST node that directly corresponds to the parsed construct. Due to the grammar structure, expression trees naturally encode operator precedence.

6 Error Handling Strategy

6.1 Syntax Errors

Syntax errors are detected by the parser. Detailed error messages are enabled to provide informative diagnostics. Each syntax error includes the line number where the error occurred.

6.2 Semantic Errors

Semantic analysis is performed during parsing and includes checks for:

- Redclaration of variables in the same scope
- Assignment to undeclared variables
- Usage of undeclared identifiers in expressions

Semantic errors are reported immediately with descriptive messages, and parsing continues where possible.

7 Limitations and Possible Extensions

7.1 Limitations

The current implementation has several limitations:

- Only integer data types are supported

- No function definitions or function calls
- No logical operators such as `&&` or `||`
- No array or composite data structures

7.2 Possible Extensions

Future enhancements could include:

- Support for multiple data types
- Additional loop constructs such as `for`
- Logical and boolean expressions
- Function definitions and calls
- Intermediate code generation and optimization passes

8 Evaluation and Symbol Table Usage

The evaluator executes the program by traversing the AST and uses the symbol table to manage variable values and scope.

8.1 Key Points

- Uses a symbol table (linked list) to store variables.
- Each entry stores name, value, and scope.
- Variable declarations insert new entries.
- Assignments update existing entries.
- Identifiers fetch values from the symbol table.
- Each block introduces a new scope.
- Scope is entered before execution and exited after.
- Variables of a scope are collected before exit.
- Runtime errors are detected during evaluation.

8.2 Current Limitations

- Loop execution is not implemented.
- Global variables must be declared at the top level.

9 Version Control Activity [GitHub](#)

Git was used to track development progress and collaboration throughout the project.

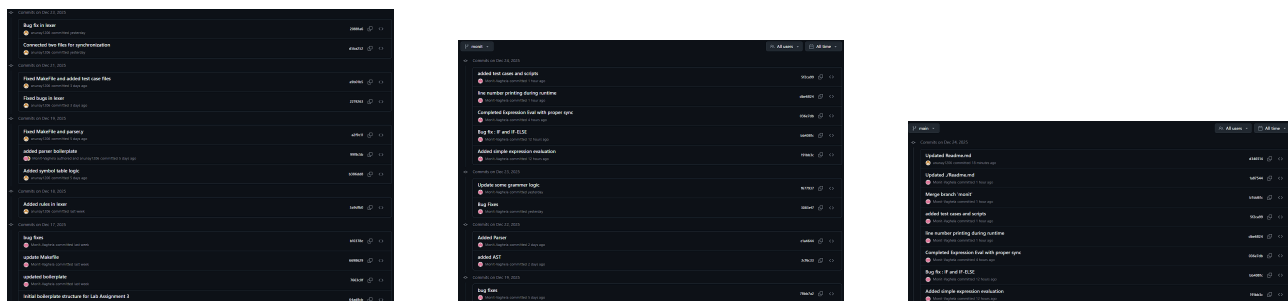


Figure 1: Git activity showing version control usage during development

10 Test Case Validation

The following screenshots demonstrate successful execution of representative test cases.

```
31 >>> tests/valid/test1.var
32 ∨ BLOCK
33 ∨   VAR_DECL(x)
34   |   INT(10)
35 ∨   VAR_DECL(y)
36   |   INT(20)
37 ∨   ASSIGN(x)
38 ∨   BINOP(+)
39   |   IDENT(x)
40   |   IDENT(y)
41 ∨   ASSIGN(x)
42 ∨   BINOP(-)
43   |   INT(0)
44   |   INT(3)
45
46 --- Runtime Values ---
47 Name: y
48 Value: 20
49 Scope: 1
50
51 Name: x
52 Value: -3
53 Scope: 1
54
```

```
>>> tests/valid/test5.var
BLOCK
  VAR_DECL(x)
  |   INT(10)
  ASSIGN(x)
  |   BINOP(<)
  |   IDENT(x)
  |   INT(20)

--- Runtime Values ---
Name: x
Value: 1
Scope: 1
```

```
>>> tests/invalid/test10.var
Error at line 2: redeclaration of variable 'x'
Parsing failed
Variable already exists
```

```
>>> tests/invalid/test5.var
Error at line 1: assignment to undeclared variable 'x'
Parsing failed
Variable not declared
```

Figure 2: Representative test case outputs