Lab 3: Compiler Construction
Parser with Flex and Bison
Technical Report

Manmandha Rao Kuttum (2025MCS2101)
Jithendra Gannavarapu (2025MCS2743)

December 24, 2025

Abstract

This report documents the design and implementation of a compiler for a C-like imperative programming language. The project utilizes **Flex** for lexical analysis and **Bison** for parsing to generate an Abstract Syntax Tree (AST). Key features include integer arithmetic, variable declarations, nested block scopes, and structured control flow. The report details the resolution of the "Dangling Else" ambiguity, the implementation of "Panic Mode" error handling, and the recursive grammar design used to support complex nesting.

1 Overview

The objective of this lab was to build a robust front-end compiler capable of verifying the syntax and semantics of a custom programming language. The compiler reads source code from standard input or files, verifies lexical and syntactic correctness, and constructs a hierarchical AST representation.

The project repository is maintained at:
`https://github.com/JithendraGannavarapu/CornerStoneProject/tree/main/lab3`

2 Description of the Designed Language

The language is designed to mimic the fundamental structure of C, focusing on integer arithmetic and structured control flow.

2.1 Key Features

- **Data Types**: Strictly Integers (literals and variables).
- **Declarations**: Explicit typing using the `var` keyword (e.g., `var x = 10;`).
- **Control Flow**:
  - `if (condition) { ... } else { ... }`
  - `while (condition) { ... }`

- **Scoping**: Nested block scopes { ... } are supported, allowing for complex, recursive structures.
- **Comments**: Standard single-line comments using `//`.

3 Internal Structure and Workflow

The compiler functions as a two-stage pipeline where the **Lexer** (Lexical Analyzer) and **Parser** (Syntax Analyzer) work in tandem to transform raw text into a structured Abstract Syntax Tree (AST). This section details the internal mechanics of this interaction.

3.1 Data Flow Architecture

The interaction between the Lexer and Parser is driven by the pull-model architecture of `yyparse()`.

1. **Controller**: The Parser (`yyparse()`) controls the flow. It requests the next token only when it needs one to complete a grammar rule.
2. **Provider**: The Lexer (`yylex()`) scans the input stream and returns a single token ID (e.g., `INTEGER`, `VAR`).
3. **Data Channel**: The actual content of the token (like the value `10` or name `"x"`) is passed through a global union variable called `yylval`.

3.2 Internal Code Structure

3.2.1 1. The Global Union (`yylval`)

To allow the Lexer to pass different data types (int, string, or AST Node) to the Parser, we defined a C `union` in Bison. This is the critical "bridge" between the two components.

```
%union {
    int ival;              /* For integer literals (e.g., 10) */
    char* sval;            /* For identifiers (e.g., "count") */
    struct ASTNode* nval; /* For AST Nodes (expressions, statements) */
}
```

Listing 1: The Data Bridge (Bison Definition)

3.2.2 2. The Lexer's Role (`lexer.l`)

The Lexer calculates the value and places it into the bridge before returning the token type.

```
[0-9]+ {
    yylval.ival = atoi(yytext); /* 1. Convert text to integer */
    return INTEGER;             /* 2. Notify Parser we found an INTEGER */
}

[a-zA-Z_][a-zA-Z0-9_]* {
    yylval.sval = strdup(yytext); /* 1. Copy string to heap */
    return IDENTIFIER;            /* 2. Notify Parser we found an ID */
}
```

Listing 2: Lexer Populating the Bridge

3.2.3 3. The Parser's Role (parser.y)

The Parser reads from the bridge using the `$` notation. `$1` refers to the first component's value, `$2` the second, and so on.

```
/* Rule: term -> term PLUS factor */
term: term PLUS factor {
    /* $$ = Output Node
       $1 = Left Child (term)
       $3 = Right Child (factor) */
    $$ = create_binop("+", $1, $3);
}
```

Listing 3: Parser Consuming Data

3.3 Execution Lifecycle

The internal workflow for processing a single statement like `var x = 10;` proceeds as follows:

1. **Initialization**: `main()` calls `yyparse()`.
2. **Token Request 1**: `yyparse` calls `yylex`. Lexer finds `var`, returns `VAR`.
3. **Token Request 2**: `yyparse` calls `yylex`. Lexer finds `x`.
   - Action: Sets `yylval.sval = "x"`.
   - Return: `IDENTIFIER`.
4. **Token Request 3**: `yyparse` calls `yylex`. Lexer finds `=`, returns `ASSIGN`.
5. **Token Request 4**: `yyparse` calls `yylex`. Lexer finds `10`.
   - Action: Sets `yylval.ival = 10`.
   - Return: `INTEGER`.
6. **Reduction**: The Parser sees the pattern `VAR IDENTIFIER ASSIGN INTEGER` matches a grammar rule. It executes the semantic action to create a `VAR_DECL` AST node using the data in `yylval`.

4 Delimiters and File Structure

In Flex (Lexer) and Bison (Parser) files, specific delimiters are used to switch between **C Code**, **Regular Expressions**, and **Grammar Rules**. These markers separate the distinct sections of the code.

4.1 The C Block Delimiter: %{ ... %}

This delimiter is used in the **Definitions Section** at the very top of the file.

- **Meaning**: "Everything inside here is raw C code. Copy it directly to the generated file."
- **Usage**: It is primarily used for `#include` statements and global variable declarations.

```
%{
    #include <stdio.h>
    #include "ast.h"    /* Include C headers here */
%}
```

Listing 4: Usage of C Block Delimiter

4.2 The Section Separator: %%

This is the primary delimiter that divides the file into three distinct sections.

1. **Definitions Section**: Imports and token definitions.
2. **Rules Section**: Regular expressions (Lexer) or Grammar rules (Parser).
3. **User Code Section**: C functions like `main()` or helpers.

- The **first** `%%` marks the end of definitions and the start of rules.
- The **second** `%%` marks the end of rules and the start of raw C code.

4.3 Bison Declarations

In addition to the block delimiters, specific `%` keywords are used for declarations in the parser:

- `%union { ... }`: Defines the semantic value types (int, string, AST node).
- `%token`: Defines terminal symbols coming from the Lexer.
- `%type`: Defines the return type of a non-terminal rule.
- `%left / %right`: Defines operator precedence and associativity.

4.4 Structure Visualization

The following snippet demonstrates how these delimiters structure a real file:

```
/* SECTION 1: DEFINITIONS */
%{
   #include <stdio.h>   /* C Code Block */
%}

%token INTEGER          /* Bison Declaration */

%%                      /* SEPARATOR: Switch to Rules */

/* SECTION 2: RULES */
[0-9]+    { return INTEGER; }

%%                      /* SEPARATOR: Switch to User Code */

/* SECTION 3: USER CODE */
int main() {
    yylex();
}
```

Listing 5: File Structure with Delimiters

5 Lexical Analysis (The Lexer)

The Lexer (`lexer.l`) acts as the "front desk" of the compiler. It transforms the raw stream of input characters into meaningful **Tokens**.

5.1 Tokenization Strategy

We utilized **Flex** to implement the tokenizer. It uses Regular Expressions (Regex) to match patterns and return specific Token IDs to the parser.

- **Keywords**: `var`, `if`, `else`, `while`.
- **Identifiers**: `[a-zA-Z_][a-zA-Z0-9_]*` (Standard C naming conventions).
- **Integers**: `[0-9]+`.
- **Operators**: `+`, `-`, `*`, `/`, `==`, `<=`, etc.

5.2 Design Decision: The `yylval` Union

A critical implementation detail is the data transfer between Lexer and Parser. When the Lexer identifies a value, it must pass the *content* (not just the type) to the Parser. We used the `%union` construct in Bison to define `yylval`:

- `ival` (int): Stores the numeric value of `INTEGER` tokens.
- `sval` (char*): Stores the name of `IDENTIFIER` tokens.
- `nval` (ASTNode*): Used by the parser for non-terminals (e.g., `expression`).

5.3 Error Handling: Panic Mode

To ensure the compiler does not produce invalid executables from garbage input, we implemented a **Panic Mode** strategy.

- **Logic**: A catch-all rule (`.`) at the end of the lexer file detects any unknown character.
- **Action**: It prints an error to `stderr` and calls `exit(1)` immediately. This ensures the operating system receives a failure exit code.

6 Grammar Design & Parsing (The Parser)

The Parser (`parser.y`), generated by **Bison**, defines the grammatical rules and builds the AST.

6.1 Recursive Grammar & Robustness

The grammar is designed recursively to support infinite nesting.

```
statement -> block -> statement_list -> statement
```

This design allows for constructs like `while` loops inside `if` blocks inside other `while` loops. We verified this using a "Stress Test" containing 5 levels of nesting.

6.2 Operator Precedence

To respect standard mathematical order of operations (PEMDAS), precedence is encoded directly into the grammar hierarchy:

1. `expression` (Lowest: `==`, `!=`)
2. `comparison` (`<`, `>`)
3. `term` (`+`, `-`)
4. `factor` (`*`, `/`)
5. `primary` (Highest: `()`, `Integer`)

6.3 Resolving the "Dangling Else" Problem

A classic ambiguity in compiler design is the "Dangling Else".

- **Ambiguity**: `if (x) if (y) A; else B;`
- **Solution**: We relied on Bison's default **Shift** preference. When the parser encounters the `else`, it shifts it onto the stack, binding it to the **nearest** open `if`.

7 AST Structure & Construction

The Abstract Syntax Tree (AST) is the final output of the analysis phase.

7.1 Node Structure (`ast.h`)

We defined a generic `ASTNode` structure capable of representing any construct:

```
typedef struct ASTNode {
    NodeType type;              // NODE_BINOP, NODE_VAR_DECL, NODE_IF
    char* val_str;              // For Identifiers ("x") or Operators ("+")
    int val_int;                // For Integers (10)
    struct ASTNode *left;       // Left Child
    struct ASTNode *right;      // Right Child
} ASTNode;
```

7.2 Bottom-Up Construction

The AST is built bottom-up using Bison actions. For example:

```
term: term PLUS factor {
    $$ = create_binop("+", $1, $3); // Connects Left ($1) and Right ($3)
}
```

8 System Test Suite

To validate the compiler's robustness, we executed a comprehensive suite of **20 Test Cases** (10 Valid, 10 Invalid). The test script verified that valid programs return Exit Code 0 and invalid programs return Exit Code 1.

8.1 Valid Test Cases (Functional Verification)

#	Test Case	Description	Status
1	`test01_basic`	Simple variable declaration.	Pass
2	`test02_math`	Arithmetic precedence (`2 + 3 * 4`).	Pass
3	`test03_ifelse`	Basic branching logic.	Pass
4	`test04_loop`	`while` loop structure.	Pass
5	`test05_nested`	**Stress Test** (Loop inside Loop inside If).	Pass
6	`test06_scope`	Block scoping { `var z = 5;` }.	Pass
7	`test07_compare`	Relational operators (<, >).	Pass
8	`test08_div_sub`	Subtraction and Division logic.	Pass
9	`test09_init_expr`	Initialization with expressions.	Pass
10	`test10_empty`	Handling empty blocks and comments.	Pass

Table 1: Valid Test Cases

8.2 Invalid Test Cases (Error Handling)

#	Test Case	Description	Status
1	`fail01_undeclared`	Using variable without declaration.	Pass
2	`fail02_double_decl`	Declaring same variable twice.	Pass
3	`fail03_no_semi`	Missing semicolon (Syntax Error).	Pass
4	`fail04_bad_assign`	Assigning to a literal (`10 = x`).	Pass
5	`fail05_paren`	Mismatched parentheses.	Pass
6	`fail06_brace`	Unclosed block braces.	Pass
7	`fail07_bad_char`	Illegal input (`var x = 10 $`).	Pass
8	`fail08_keyword`	Using keyword as identifier (`var if`).	Pass
9	`fail09_missing_op`	Incomplete expression (`10 + ;`).	Pass
10	`fail10_bad_if`	Malformed control flow syntax.	Pass

Table 2: Invalid Test Cases

9 Solutions & Design Decisions

This section summarizes critical technical solutions implemented during the lab.

1. **Makefile Re-structuring**:
   - *Problem*: The `Makefile` was originally inside `src/`, complicating the build process from the root directory.
   - *Solution*: We moved the `Makefile` to the root and used strict paths (e.g., `flex -o src/lex.yy.c src/lexer.l`), ensuring a clean separation of source and build artifacts.
2. **Exit Code Management**:
   - *Problem*: The test script failed to detect errors because `main()` returned 0 even after `yyerror()` was called.

- *Solution*: We modified `yyerror()` to call `exit(1)`. This guarantees that the OS receives a failure signal, allowing the automated test suite to function correctly.

3. **Symbol Table Safety**:

   - *Design*: Implemented a linear symbol table with a `MAX_VARS` limit of 1000.
   - *Safety*: Added explicit bounds checking to prevent Segmentation Faults.

10 Limitations and Future Extensions

Although the current implementation fulfills all functional requirements for a basic compiler front-end, there are several areas where the design could be expanded.

10.1 Limitations

- **Symbol Table Efficiency**: The current symbol table is implemented as a linear array with a fixed limit (`MAX_VARS 1000`). For significantly larger programs, this would be inefficient ($O(n)$ lookup time). A production-grade compiler would use a Hash Table for $O(1)$ lookups.
- **Type System**: The language is strictly limited to the `integer` data type. It does not currently support floating-point numbers, strings, or boolean types.
- **Modular Programming**: The language does not support function definitions or function calls. The entire program is effectively a single `main` body.

10.2 Future Extensions

- **Type Checking**: The `check_symbol` function could be extended to store metadata about variable types, enabling the compiler to reject invalid assignments (e.g., assigning a string to an integer).
- **Code Generation**: The natural next step is to traverse the constructed AST and emit **LLVM IR** or **x86_64 Assembly**. This would transform the tool from a syntax checker into a fully functional compiler that produces executables.
- **Constant Folding**: We could implement an optimization pass during AST construction to pre-calculate constant expressions (e.g., turning `3 + 5` directly into `8`), reducing the runtime overhead of the final program.

11 Conclusion

This project successfully implements a front-end compiler satisfying all functional requirements. By integrating a recursive grammar with a robust AST construction strategy, the compiler handles complex control flows and mathematical expressions accurately. The decision to implement **Panic Mode** error handling ensures that invalid inputs are strictly rejected, providing a reliable foundation for future code generation stages.