Assignment 1: Lexical Analysis with Flex

CS 464/5607: Compiler Design

Spring 2026

1 Introduction

In this assignment, you will implement the first phase of a compiler: the **Lexical Analyzer** (or Scanner). Your task is to use the `flex` tool to generate a C scanner that reads source code and breaks it down into a stream of tokens.

You will be provided with a driver program and a list of required tokens. Your job is to define the Regular Expressions (Regex) and rules in a `.l` file to correctly identify these tokens.

2 Understanding Flex (The Fast Lexical Analyzer Generator)

2.1 What is Flex?

Flex (Fast Lexical Analyzer Generator) is a tool used to generate scanners: programs that recognize lexical patterns in text. It reads a specification file (typically with a `.l` extension) containing pairs of Regular Expressions and C code actions, and it outputs a C source file (usually `lex.yy.c`) that implements a scanner for those patterns.

2.2 Why are we using it?

Writing a lexical analyzer by hand involves creating a complex Finite State Machine (FSM) to track state transitions for every character. While possible, this is tedious and error-prone. Flex automates this process. By simply describing *what* patterns you want to match using high-level Regular Expressions, Flex handles the difficult task of generating the optimized, low-level C code to implement the corresponding FSM.

2.3 How Flex Works

When you run the command `flex lexer.l`, the following happens in the background:

1. **Regex Parsing:** Flex reads your rules and parses the Regular Expressions.
2. **NFA Construction:** It converts each Regex into a Nondeterministic Finite Automaton (NFA).
3. **DFA Conversion:** It combines these NFAs and converts them into a single Deterministic Finite Automaton (DFA), which is much faster to execute.
4. **Table Generation:** It generates transition tables and a driver routine (the `yylex()` function) in C.

The resulting file, `lex.yy.c`, is standard C code. When compiled and linked with your driver program, the `yylex()` function simulates the DFA to tokenize the input stream efficiently.

Note: You will need Flex installed on your system to complete this assignment.

3 Provided Files

The following files are provided in the skeleton code. You should not modify the C++ files or headers; your work is focused on the Flex file.

3.1 `driver.cpp`

This is the entry point of the application. Its purpose is to:

1. Open the input file.
2. Call the `yylex()` function (generated by Flex) in a loop.
3. Print the token type and the matched lexeme to standard output using `std::cout`.

3.2 `tokens.h`

This header file defines the `enum` for all the Token IDs (e.g., `TOKEN_INT`, `TOKEN_IF`, `TOKEN_ID`). **Purpose:** These IDs are the "return values" your lexer must send back to the driver whenever it recognizes a pattern.

3.3 `lexer.l` (Your Task)

This is the Flex specification file where you will write your implementation. You must define the patterns (Regex) to recognize keywords, operators, identifiers, and literals.

4 Implementation Guide

4.1 Structure of a Flex File

A Flex file consists of three distinct sections separated by `%%`:

1. **Definitions Section:** Used for C imports and defining reusable Regex macros.
2. **Rules Section:** This is the core of your assignment. Here you define the patterns (using Regex) and the corresponding actions (C code) to execute when a pattern is matched.
3. **User Code Section:** Used for helper C functions (not required for this assignment).

Your goal is to populate the Rules Section to recognize the language constructs.

4.2 Implementation Hints

Here are a few tips to help you structure your `lexer.l` file effectively.

4.2.1 Regex Definitions

To keep your rules clean and readable, you should define reusable Regular Expressions in the **Definitions Section** (the top part of the file). This allows you to construct complex patterns from simpler building blocks.

Example of Regex Definitions:

```
DIGIT        [0-9]
LETTER       [a-zA-Z]
```

4.2.2 Handling Whitespace and Comments

The lexical analyzer should only return meaningful tokens to the parser.

- **Whitespace:** Spaces, tabs, and newlines should be ignored (consumed without returning a token).
- **Comments:** You must implement support for **single-line comments** (starting with `//`). These should also be ignored. Note that multi-line comments are not supported in this assignment.

4.2.3 Token Definitions

Refer to `tokens.h` for the exact names of the token constants you need to return (e.g., `TOKEN_WHILE`, `TOKEN_PLUS`). Your lexer must support all tokens listed in that file.

4.2.4 Handling Errors

A robust lexer must handle unexpected input. If the scanner encounters a character or pattern that does not match any of your defined rules, it should return `T_ERROR`. This acts as a catch-all for invalid characters.

To provide meaningful error messages, Flex provides two global variables:

- `yylineno`: An integer holding the current line number (enabled by `%option yylineno`).
- `yytext`: A string containing the text that matched the current rule.

You can use these variables in your error rule to print exactly where and why the error occurred. For example:

```
printf("Lexical Error at line %d: Unknown char '%s'\n", yylineno, yytext);
```

Note on Order: Since `T_ERROR` is a catch-all, where you place this rule matters significantly. (See Section 3.3).

4.3 Key Lexical Properties

To implement the lexer correctly, you must understand how Flex decides which rule to apply when multiple patterns could potentially match the input.

4.3.1 1. The Longest Match Rule (Maximal Munch)

Flex will always choose the rule that matches the **most characters** in the input stream.

Conceptual Example: Imagine your input is the word `"integer"`.

- You have a rule that matches the keyword `"int"`.
- You have another rule that matches identifiers (words starting with a letter).

Even though the first 3 characters match the keyword rule, Flex sees that the identifier rule can match all 7 characters ("integer"). Because 7 is greater than 3, Flex selects the identifier rule. This ensures that variables with names like `integer` or `format` are not incorrectly split into keywords.

4.3.2 2. Rule Priority (Order Matters)

What happens if two rules match the **exact same number** of characters? In this case, Flex resolves the tie by choosing the rule that appears **first** in the `lexer.l` file.

Conceptual Example: Imagine your input is the word `"int"`.

- The keyword rule matches 3 characters.
- The identifier rule also matches 3 characters.

If the identifier rule is placed *before* the keyword rule in your file, Flex will treat `"int"` as an identifier, which is incorrect. To ensure reserved words are recognized properly, you must order your rules from **specific** (keywords) to **general** (identifiers).

5 Building and Testing

5.1 Building the Project

We use a `Makefile` to automate compilation. To build the lexer, open your terminal in the project directory and run:

```
make
```

This will generate the C code using Flex and compile it into an executable located in the `build/` directory.

5.2 Platform Specifics (Windows vs. macOS/Linux)

The provided build scripts assume a Windows environment by default (looking for `.exe` files). If you are working on **macOS** or **Linux (WSL)**, you must modify two files before starting:

1. **Makefile**: Find the line defining the executable name.
   - Change: `LEXER_EXE = $(BUILD_DIR)/lexer.exe`
   - To: `LEXER_EXE = $(BUILD_DIR)/lexer`
2. **run_tests.py**: Find the configuration variable at the top.
   - Change: `LEXER_EXECUTABLE = "./build/lexer.exe"`
   - To: `LEXER_EXECUTABLE = "./build/lexer"`

5.3 Running the Test Suite

We have provided a Python script to run automated tests.

```
# Build first
make

# Run tests
python run_tests.py
```

The script compares your lexer's output against ”Golden Output” files.

- **60%** of the grade is based on the visible tests provided to you.
- **40%** of the grade is based on **Hidden Test Cases**

5.4 Testing with Custom Inputs

You are encouraged to create your own test cases to debug specific issues.

1. Create a text file (e.g., `my_input.txt`) with some code.
2. Run your built lexer and redirect the input:

On Windows:

```
build\lexer.exe my_input.txt
```

On macOS/Linux:

```
./build/lexer my_input.txt
```

This will print the tokens to the terminal, allowing you to manually verify if your rules are working as expected.

6 Submission Requirements

Please follow the steps below carefully to ensure your assignment is graded correctly.

6.1 Preparing Your Submission

Before zipping your project, you must clean the directory to remove all generated build files and executables.

1. Run the following command:

```
make clean

```

2. Verify that the `build/` directory has been removed.

6.2 Naming Convention

You must compress your project folder into a single **.zip** file. The filename must strictly follow this format:

`<rollnumber>_PA1.zip`

Replace `<rollnumber>` with your actual university roll number.
Example:

- If your roll number is **27100289**, your submission file must be named:

`27100289_PA1.zip`