Assignment 1: Lexical Analysis with Flex

CS 464/5607: Compiler Design

Spring 2026

1 Introduction

This assignment has two parts. In the first part(see section3), 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. In the second part of the assignment, you will design Non-Deterministic and Deterministic Finite Automata. (see section 6)

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.

2.4 Installation

You will need Flex installed on your system to complete this assignment. Follow the instructions below for your specific operating system:

2.4.1 Windows

Windows does not have native support for Flex. You must install it manually or use a Linux subsystem.

- **Option 1 (Recommended):** Install WSL (Windows Subsystem for Linux) and follow the Linux instructions below.
- **Option 2 (Native):** Follow the step-by-step guide at GeeksforGeeks to install Flex using MinGW or GnuWin32:
  `https://www.geeksforgeeks.org/installation-guide/how-to-install-flex-on-windows/`

2.4.2 Linux (Ubuntu/Debian)

Open your terminal and run the following command to install Flex:

```
sudo apt-get update
sudo apt-get install flex
```

2.4.3 macOS

macOS users can easily install Flex using the Homebrew package manager.

1. Open your Terminal.
2. Run the following command:
   ```
   brew install flex
   ```
3. **Note:** macOS often comes with an older version of Flex pre-installed. Using Homebrew ensures you have a recent version compatible with our assignments.

2.5 Reference Documentation

For detailed information on Flex syntax and features, refer to the Flex documentation:

`https://www.cs.virginia.edu/~cr4bd/flex-manual/`

3 PART A - Designing a Lexer

The first part of your assignment is to design the lexer. For this, a driver program and a list of required tokens is provided. Your job is to define the Regular Expressions (Regex) and rules in a `.l` file to correctly identify these tokens.

The following files are provided in the skeleton code. Please **do not modify** the C++ files or headers; your work should focus 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);
```

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.

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 PART - B: Designing Finite Automata

In addition to implementing the lexical analyzer in Flex, you must strengthen your theoretical understanding of Finite Automata. In this section, you are required to design Non-Deterministic Finite Automata (NFA) and Deterministic Finite Automata (DFA) for the given languages.

6.1 DFA Design Challenges

Design a **DFA** for the following languages over the alphabet $\Sigma = \{0, 1\}$. Ensure your DFA has the minimum number of states possible.

1. **Question 1:** The set of all strings where the number of 0's is divisible by 3 **AND** the number of 1's is even.
2. **Question 2:** The set of all strings that represent a binary number divisible by 5. (Read from left to right, e.g., $101_2 = 5$).
3. **Question 3:** The set of all strings that do **not** contain the substring "110".
4. **Question 4:** The set of all strings such that every block of five consecutive symbols contains at least two 0's.

6.2 NFA Design Challenges

Design an **NFA** for the following languages over the alphabet $\Sigma = \{a, b, c\}$. You should utilize non-determinism to make your design concise.

1. **Question 5:** The set of all strings where the 3^{rd} symbol from the **end** is 'a'. (e.g., "bb**a**bc", "a**a**aa").
2. **Question 6:** The set of all strings that contain either "abc" or "cba" as a substring.
3. **Question 7:** The set of all strings where the first and the last character are different.
4. **Question 8:** The set of all strings of length at least 3 where the symbol at index i is the same as the symbol at index $i + 2$ for some valid i. (i.e., there exists a repetition with exactly one character in between, like "aba" or "aca").

6.3 Tools and Instructions

You may use the online tool **Finite State Machine Designer** by Evan Wallace:

`https://madebyevan.com/fsm/`

How to use the tool:

- **Add a State:** Double-click on the canvas.
- **Add a Transition:** Shift-drag from one state to another.
- **Mark Accepting State:** Double-click on an existing state.
- **Export:** Once your diagram is complete, click the "Export" button and select **LaTeX**. Copy the generated code into your final submission document.

6.4 Submission Format for Exercises

Create a single PDF document named `<rollnumber>.pdf` containing the diagrams for all 8 questions below. You may use Overleaf or any LaTeX editor to compile the exported code from the tool.

7 Submission Requirements

7.1 Evaluation Criteria

Passing every test case, whether visible or hidden, will grant **5 marks**. Only fully functional automata will receive full credit, i.e., (5 mark for each)

7.2 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.

7.3 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 roll number.
Example:

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

`27100289_PA1.zip`

Good Luck!