# How to Code in LEX

CS321 : Compiler Lab

16-Jan-2025

LEX (Lexical Analyzer Generator) is a tool used to generate lexical analyzers or scanners. These analyzers identify strings in input text/program based on specified patterns. A sample of input strings and their corresponding output are provided below.

```
//input program
var = 12 + 9;
if (test > 20)
    temp = 0;
else
    while (a < 20)
        temp++;
```

Output:

- Identifier: Var
- Operand: =
- Integer: 12
- Operand: +
- Integer: 9
- Semicolon: ;
- Keyword: if
- ...

A detailed guide on coding in LEX is provided below, covering the structure, syntax, and steps for creating and executing LEX programs.

## 1 Structure of a LEX Program

A LEX program consists of three main sections:

### 1.1 Definition Section

This section is enclosed between %{ and %}. It is used to define C code, including header files, macros, global variables, and functions.

```
%{
#include <stdio.h>
%}
```

### 1.2 Rules Section

Contains the patterns (in regular expression) to be matched and the corresponding actions to be executed. Each rule is written as:

`pattern { action }`

Patterns use regular expressions to describe the input text. Actions are written in C code.

### 1.3 User Code Section

Typically, contains the `main()` function to initiate the lexical analysis process. Additional helper functions can also be included.

## 2 Key Components in a LEX Program

### 2.1 Patterns

Patterns are specified using regular expressions. Common elements include:

- `.` : Matches any single character except a newline.
- `[ ]` : Matches any one character inside the brackets.
- `*` : Matches zero or more occurrences of the preceding character.
- `+` : Matches one or more occurrences of the preceding character.
- `|` : Logical OR (e.g., `cat|dog` matches "cat" or "dog").
- `^` : Matches the beginning of a line.
- `$` : Matches the end of a line.
- `?` : Zero or one copies of the preceding expression

**Some pattern examples:**

| Expression | Matches |
|---|---|
| abc | abc |
| abc* | ab, abc, abcc, abccc, ... |
| abc+ | abc, abcc, abccc, ... |
| a(bc)+ | bc, abcbc, abcbcbc, ... |
| a(bc)? | a, abc |
| [abc] | a, b, c |
| [a-z] | Any letter, a through z |
| [a\-z] | a, -, z |
| [-az] | -, a, z |
| [a-zA-Z]+ | One or more alphabet |
| [0-9]+ | Any number |
| [A-Za-z0-9]+ | One or more alphanumeric characters |
| [ \t\n]+ | Whitespace ( , \t, \n) |
| [^ab] | Anything except a or b |
| [a^b] | a, ^, b |
| [a—b] | a, —, b |
| a—b | a or b |

### 2.2 Actions

Actions are written in C code and executed when a pattern matches. Common actions include:

- displaying output to the user (using `printf()`),
- variable assignments and arithmetic operations,
- function calls.

```
[a-z]+ { printf("Found a word: %s\n", yytext); }
```

### 2.3 Special Variables

- `yytext`: A pointer to the matched string.
- `yyleng`: The length of the matched string.
- `yylineno`: The current line number in the input.

## 3. Example of a LEX Program

### Task: Identify digits and words in an input text.

```
%{
#include <stdio.h>
%}

%%
[0-9]+          { printf("Number: %s\n", yytext); }
[a-zA-Z]+       { printf("Word: %s\n", yytext); }
[ \t\n]+        ;  /* Ignore whitespace */
.               { printf("Unknown character: %s\n", yytext); }
%%

int main() {
    printf("Enter input:\n");
    yylex(); // Call the lexical analyzer
    return 0;
}
```

## 4. Steps to Create and Run a LEX Program

1. Write the LEX file and save it with a `.l` extension (e.g., `scanner.l`).
2. Generate the C code using the `lex` command:

```
lex scanner.l
```

3. Compile the C code using a C compiler:

```
gcc lex.yy.c -o scanner -ll
```

4. Execute the compiled program:

```
./scanner
```

5. Provide input to test the program.

# 5. Tips for Writing Efficient LEX Programs

- Place more specific patterns before general ones to avoid incorrect matches.
- Use patterns like `[ \t\n]+` to handle and ignore whitespace.
- Use `yytext` in actions to debug pattern matching.
- Include a catch-all pattern (`.`) to handle unexpected input.