BITS Pilani, Hyderabad Campus
Department of Computer Science and Information Systems
Second Semester, 2024-25
CS F363 Compiler Construction
Lab-5: Flex/Lex tool

1 Objectives

The objectives of this lab sheet are given below.

1. **Mastering State Management:** Students will gain proficiency in using Flex start states (`%s` and `%x`) and the BEGIN directive to create context-sensitive lexical analyzers, enabling them to parse complex language structures.
2. **Contextual Lexical Analysis:** Students will learn to apply the `/` operator for right context matching, enabling them to create Lex programs that recognize tokens based on their surrounding context, as demonstrated in tasks like email validation and distinguishing identifiers from function calls.
3. **Advanced Lex Features:** Students will become familiar with and effectively utilize advanced Flex features such as `REJECT, yyterminate(), yyrestart(), input(), unput(), yymore(),` and `yyless()` to handle complex lexical analysis scenarios including overlapping tokens, input stream manipulation, and multi-part token recognition.
4. **Practical Application of Lex:** Students will apply their knowledge of Lex to build practical tools like a simplified Markdown to HTML converter and a C-style comment remover, demonstrating their ability to use Lex for real-world text processing tasks.
5. **Regular Expression Design:** Students will reinforce their understanding of regular expression design and learn to create complex regular expressions for pattern matching in various contexts, including email addresses, C identifiers, and Markdown syntax.

2 State definitions in Flex

Flex (Fast Lexical Analyzer) is a tool used for lexical analysis, generating a scanner that processes input according to specified patterns and actions. The BEGIN directive plays a crucial role in controlling how the scanner processes input by allowing dynamic switching between different states. Start States in Flex

The start states allow the scanner to recognize different modes of input processing. By default, the scanner operates in the initial state, but we can define custom start states using Flex's `%s` (inclusive) and `%x` (exclusive) directives.

- Inclusive Start States (`%s`): Tokens in these states are recognized along with the rules in the default state.
- Exclusive Start States (`%x`): Tokens in these states are only recognized when the scanner is explicitly placed in that state.

Using BEGIN to Control States:

The BEGIN directive is used to switch between different start states dynamically.

- `BEGIN STATE_NAME;` transitions the scanner into the specified state.
- `BEGIN 0;` resets the scanner to the initial state.

Example 1 The following code counts and prints the number of single-line and multi-line comments (not nested comments) in a `C` file.

```
%{
#include <stdio.h>
int s_comment=0;
int m_comment=0;
%}

%x COMMENT

%%
"//"(.)*\n {s_comment++; }
"/*"       { BEGIN(COMMENT); } /* Enter COMMENT state */
<COMMENT>. { }
<COMMENT>"*/"   { m_comment++; BEGIN(0); } /* Exit COMMENT state */
<COMMENT>\n     { /* Handle new lines inside comments */ }
.|\n       {}
%%

int main() {
   yyin=fopen("input.c", "r");
   yylex();
   printf("No. of Single-line comments = %d\n", s_comment);
   printf("No. of Multi-line comments = %d\n", m_comment);
   return 0;
}
```

The content of `input.c`:

```
//this program just adds two numbers;
#include <stdio.h>

int main() {

   int a, b, c;
   /* This is a
      multi-line comment */

   a = b+ c; //addition of b and c is stored in c

   printf("%d", a); //print the value of a;

   /* done with
   the code
   and just stop it */

  return 0;
}
```

Task 1 (a) Change the state `COMMENT` type from `%x` to `%s` and observe the output.

(b) Write a Flex program that takes a C program from a file and generates a new file with the same content as the input C file, except comments and empty lines.

The output for the above `input.c` should be as follows:

```
#include <stdio.h>
int main() {
    int a, b, c;
    a = b+ c;
    printf("%d", a);
    return 0;
}
```

3 Matching a Right Context

Lex provides a powerful mechanism for matching regular expressions based on the surrounding context. This is achieved using the / operator, where r_1/r_2 means "match regular expression r_1 only if it is followed by regular expression r_2". Think of r_2 as a "right context" or "lookahead" assertion. Crucially, r_2 itself is not part of the matched lexeme; it only influences whether r_1 is matched.

How it Works:

(a) **Matching** r_1: Flex attempts to match the regular expression r_1 in the input stream.

(b) **Context Check** (r_2): If r_1 is matched, Lex then checks if the input immediately following the match for r_1 matches the regular expression r_2.

(c) **Successful Match**: If both r_1 and r_2 match (with r_2 following r_1), the match is considered successful. The returned lexeme is the input portion that matched only r_1.

(d) **Failed Match**: If r_1 matches but r_2 does not immediately follow, the match for r_1 is rejected, and Lex continues scanning the input.

(e) **Longest Match**: Flex's longest match rule still applies. When considering multiple possible matches involving contextual expressions, Flex will prefer the longest match for the concatenation of r_1 and r_2. However, the returned lexeme is still only the part matched by r_1.

Example 2 The following code demonstrates the concept of matching the right context.

```
%{
#include<stdio.h>
%}
%%
a+/b { printf("T_AB: %s\n", yytext); }
a+/c { printf("T_AC: %s\n", yytext); }
b {printf("T_B: %s\n", yytext); }
c {printf("T_C: %s\n", yytext); }
. ECHO;
\n return 1;
%%
```

```
int main()
{
printf("Enter a string :");
yylex();
return 0;
}
```

The breakdown of the Lex output for the input `aaaabbc`:

```
aaaa: The rule a+/b matches aaaa. The output is:  T_AB: aaaa
b: The rule b matches the character "b". The output is: T_B: b
b: The rule b matches the character "b". The output is: T_B: b
c: The rule c matches the character"c". The output is: T_C: c

Therefore, the complete output will be:
T_AB: aaaa
T_B: b
T_B: b
T_C: c
```

The output for the input `aaaacbc` is

```
T_AC: aaaa
T_C: c
T_B: b
T_C: c
```

Task 2 Write a Lex program that identifies valid email addresses. A simplified version could require the format `username@domain.com`. The username and domain can contain alphanumeric characters, periods, underscores, and hyphens. The program should print "Valid Email" if the input matches the pattern and "Invalid Email" otherwise. Use the '/' operator to ensure the '.com' (or other top-level domain TLD) is present but not included in the matched lexeme.

Test Case	**Explanation**
user.name@example.com	Valid format with a dot in the username.
user_name-123@domain-name.org	Valid format with underscore and hyphen.
test.email@sub.domain.net	Valid format with subdomain.
simple@example.edu	Simple valid email with '.edu' TLD.
user+alias@some-domain.com	'+' is allowed in the username.

Table 1: Valid Email Test Cases

4 Some additional features of Flex/lex

4.1 REJECT

By default, Flex (Fast Lexical Analyzer) processes input by recognizing the longest matching token and consuming it before moving on to the next match. However, sometimes, we need to detect overlapping tokens—meaning that a single segment of input should be matched by multiple rules.

This is where the REJECT action comes into play. When a rule executes REJECT, Flex:

Test Case	**Explanation**
username@domain	Missing TLD (e.g., ‘.com‘).
user@domain.toolong	Unsupported TLD (not ‘com‘, ‘org‘, ‘net‘, or ‘edu‘).
user@@domain.com	Double ‘@‘ is invalid.
@domain.com	Missing username.
user.name@.com	Domain cannot start with a dot.
user@domain_com	Underscore (_) is not allowed in domain names.
user@domain.	TLD cannot be empty.
user@-domain.com	Domain name cannot start with a hyphen.

Table 2: Invalid Email Test Cases

- “Puts back” the matched text as if it was never consumed.
- Searches for the next best possible match for the same text.

This is particularly useful when you want to find all occurrences of a token within a string, even if they overlap.

Look at the following two examples and observe the difference.

Example 3 Without REJECT:

```
Ex.1
%{
#include <stdio.h>
int npink=0;
int npin=0;
int nink=0;
%}
%%
pink {npink++; }
pin  {npin++; }
ink  {nink++; }
\n  return;
%%
main()
{
yylex();
printf("number of occurrences of PINK %d\n", npink);
printf("number of occurrences of PIN %d\n", npin);
printf("number of occurrences of INK %d\n", nink);
}
```

Example 4 With REJECT:

```
Ex.1
%{
#include <stdio.h>
int npink=0;
int npin=0;
int nink=0;
%}
%%
pink {npink++; REJECT; }
```

```
pin  {npin++; REJECT;}
ink  {nink++; REJECT;}
\n  return;
%%
main()
{
yylex();
printf("number of occurrences of PINK %d\n", npink);
printf("number of occurrences of PIN %d\n", npin);
printf("number of occurrences of INK %d\n", nink);
}
```

4.2 yyterminate()

The function `yyterminate()` can be used in place of a return statement inside an action. When called, it terminates the scanner and returns 0 to the caller of `yylex()`, indicating that lexing is complete.

By default, Flex calls `yyterminate()` when an end-of-file (`EOF`) is encountered, so explicit use is typically optional unless required for early termination.

Example 5 Illustration of the use of `yyterminate()`

```
%{
#include <stdio.h>
%}
%%
[a-z]+ {
    printf("Lower case detected:\n");
    ECHO;
    printf("\nBegin yyterminate\n");
    yyterminate();
    printf("End of yyterminate\n"); // This will not execute
}
[a-zA-Z]+ {
    printf("Mixed case detected:\n");
    ECHO;
}
%%
int main() {
    yylex();
    printf("Lexing complete. Exiting...\n");
    return 0;
}
```

Run the above example and observe the outputs for the following input strings:

```
1. BitsPilani
2. bitspilani
```

4.3 yyrestart()

In Flex, the function `yyrestart(FILE *fp)` is used to reset the input stream for lexical analysis. Instructs the scanner to switch to a new input file, effectively restarting scanning from the beginning of the given file.

Key Features of yyrestart():

1. Allows scanning from a different file without restarting the entire program.
2. Useful for dynamically handling multiple input sources.
3. Ensures the buffer is reset, preventing residual data from affecting the new scan.

Example 6 The following example illustrates the use of `yyrestart()`; reading the first input from the terminal and the second input from a file.

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

%%

[a-z]+ { printf("\nLower case token found= ");
    ECHO;
    return 0;
}

[A-Z][a-zA-Z]* {  ECHO; }

%%

int main() {
    // First lexical analysis (reads from standard input)
    printf("Starting first lex operation (taking input from terminal):\n");
    printf("**********************************\n");
    printf("Enter a string : ");
    yylex();
    printf("\nEnd of first yylex operation.\n");
    printf("**********************************");

    // Open a new file for second lexical analysis
    FILE *fp = fopen("input.txt", "r");
    if (!fp) {
        printf("Error: Could not open input.txt\n");
        return 1;
    }

    printf("\nSwitching input to 'input.txt'...\n");
    yyrestart(fp); // Restart lexer with new input file

    printf("\nStarting second lex operation from 'input.txt':\n");
    yylex(); // Process the file

    printf("\nEnd of second lex operation.\n");

    fclose(fp); // Close the file
    return 0;
}
```

Content of `input.txt`:

```
Welcome To Compiler Construction course
Lab sheet 4: Flex tool
BPHC Hyd 2024-2025 Second Sem
```

Example 7 The following example illustrates the use of `yyrestart()`; reading the input from two flies one after another.

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

%%

[a-z]+ { printf("\nLower case token found= ");
    ECHO;
    return 0;
}

[A-Z][a-zA-Z]* {  ECHO; }

%%

int main() {
    // First lexical analysis (reads from input1.txt)
    FILE *fp1 = fopen("input1.txt", "r");
    if (!fp1) {
        printf("Error: Could not open input.txt\n");
        return 1;
    }

    yyin=fp1;
    yylex();

    printf("\nSwitching input to 'input2.txt'...\n");


    printf("\nStarting second lex operation from 'input.txt':\n");

    FILE *fp2 = fopen("input2.txt", "r");
    if (!fp2) {
        printf("Error: Cannot open file2.txt\n");
        return 1;
    }

    printf("\nSwitching to file2.txt...\n");
    yyrestart(fp2); // Set input to input2.txt
    yylex(); // Process input2.txt


    printf("\nEnd of second lex operation.\n");
    fclose(fp2); // Close the file
    return 0;
}
```

Content of `input1.txt`:

```
Welcome To Compiler Construction course
Lab sheet 4: Flex tool
BPHC Hyd 2024-2025 Second Sem
```

Content of `input2.txt`:

```
Input File Two
Welcome To Compiler Construction course
Lab sheet 4: Flex tool
BPHC Hyd 2024-2025 Second Sem
```

Task 3 Change the line `yyrestart(fp2);` to `yyin = fp2;` and observe the output.

4.4 `input()` and `unput()`

Lex provides built-in functions that help in advanced character handling:

- The `input()` macro reads the next character from the input stream. It's the most fundamental way to consume characters in Lex.

  Return Value: `input()` returns the next character as an integer. If the end of the input stream is reached, it returns 0. It is crucial to check for this end-of-file condition.

  Usage: `input()` is typically used when you need to look ahead in the input stream or when you need to process characters individually beyond what regular expression matching allows.

- `unput(c)`: The `unput()` macro "puts back" a character into the input stream.

  Argument: `unput()` takes a single character (an integer) as an argument.

Example 8 Write a Lex program to read a line of input and print each character separately.

```
%{
#include <stdio.h>
%}
%%
\n return 0;
. {
   printf("Read character by the regular expression: %s\n", yytext);
   printf("Read character by input(): %c\n", input());
   }

%%
int main() {
printf("Enter a string:");
yylex();
return 0;
}
```

Sample input and output for the above program is as follows:

```
$./a.out
Enter a string:abcd
Read character by the regular expression: a
Read character by input(): b
Read character by the regular expression: c
Read character by input(): d
```

Task 4 Observe the output of the above code for input string `abcde`.

Example 9 The following example illustrates the behavior of `input()`.

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

%%

"un" {printf("The unput char is = %s\n", yytext);

   }
[a-z]+ {printf("Lower case token is = %s\n", yytext);
   unput('n');
   unput('u');
   printf("The token after unput is = %s\n", yytext);

   }
[a-zA-Z]+ {printf("\nMixed token is = ");
   ECHO;
   }

. {}
\n return 0;
%%

int main() {
    printf("Enter a string: ");
    yylex();
    return 0;
}
```

The sample input and output for the above program is as follows:

```
$ ./a.out
Enter a string: hello
Lower case token is = hello
The token after unput is = helun
The unput char is = un
```

Important Considerations:

- **Buffering:** Lex uses buffers for input. `unput()` doesn't directly modify the original input source; it puts characters back into Lex's buffer.

- **Limitations:** There's a limit to how many characters you can `unput()`. Pushing too many characters back might lead to buffer overflow.
- **Efficiency:** Excessive use of `input()` and `unput()` can make your Lexer less efficient. Try to design your rules to minimize the need for these functions. Regular expressions are usually the most efficient way to match patterns.

These routines give you fine-grained control over the input stream, allowing you to handle complex lexical analysis tasks that pure regular expressions might not be able to achieve alone. They are powerful tools when used judiciously.

4.5 `yymore()` and `yyless()`

This section describes the `yymore()` and `yyless()` functions in Flex, which provide powerful mechanisms for manipulating the matched text (`yytext`) and the input stream.

4.5.1 `yymore()`: Appending to `yytext`

Normally, each regular expression match *replaces* the contents of `yytext`. `yymore()` changes this behavior by *appending* the newly matched text to the *existing* contents of `yytext`. This is useful for collecting parts of a token matched by different rules or accumulating characters across multiple matches.

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

%}

%%
"BITS" {printf("First match: %s\n", yytext); yymore(); } // Match "part1" and append
"Pilani" {printf("Combined: %s\n", yytext); } // Match "part2" and print the
    combined text
.       { } // Ignore other characters
\n return 0;
%%

int main() {
   printf("Enter a string: ");
   yylex();
   return 0;
}
```

The input and output samples for the above program are as follows:

```
$ ./a.out
Enter a string: BITSPilani
First match: BITS
Combined: BITSPilani
```

Explanation:

1. "BITS" is matched (Rule 1):

   "First match: BITS" is printed. `yymore()` is executed, so "BITS" is stored in `yytext`, and the scanner looks for more input to append.

2. "Pilani" is matched (Rule 2):

   "Combined: BITSPilani" is printed. `yytext` now contains "BITSPilani" because of the `yymore()` call earlier.

Use Cases:

- **Multi-part tokens:** Tokens composed of parts matched by different regular expressions.
- **Collecting text:** Gathering sequences of characters or words.
- **Handling escape sequences:** Matching escape characters and appending the escaped characters to the token.

4.5.2 `yyless(n)`: Trimming and Reputting

`yyless(n)` removes all but the *first* `n` characters from `yytext` and puts the removed characters back onto the input stream. These characters will be rescanned. `yyless()` also updates `yyleng` to `n`.

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

%}

%%
[0-9]+ {
       if (yyleng > 3) {
           printf("Matched number: %s\n", yytext);
           yyless(3); // Keep only the first 3 digits
           printf("First 3 digits: %s\n", yytext);
           printf("Rest of the number will be rescanned.\n");
       } else {
           printf("the last match: %s\n", yytext);
       }
   }
.      ;
\n return 0;
%%

int main() {
   printf("Enter a string: ");
   yylex();
   return 0;
}
```

The input and output samples for the above program are as follows:

```
$ ./a.out
Enter a string: 12345
Matched number: 12345
First 3 digits: 123
Rest of the number will be rescanned.
the last match: 45

$ ./a.out
Enter a string: 1234567
Matched number: 1234567
First 3 digits: 123
Rest of the number will be rescanned.
Matched number: 4567
First 3 digits: 456
Rest of the number will be rescanned.
the last match: 7
```

Explanation for input 12345

1. The rule matches “12345”.
2. `yyleng` is 5.
3. `yyless(3)` is called. “45” is put back onto the input stream. `yytext` becomes “123”, and `yyleng` becomes 3.
4. The `printf` statement prints “123”.
5. The next `yylex()` call rescans “45”.

Use Cases:

- **Lookahead with backtracking:** Inspecting more input before processing a part of it.
- **Handling variable-length tokens:** Processing a fixed-size prefix of variable-length tokens.
- **Error recovery:** Putting back characters after error detection.

5 Exercise Problems

1. Write a Flex program to implement a simple calculator (evaluating from left to right, without operator precedence and associative rules).

```
Input: 1+2*3-4
Answer: 5.00


Input: 2*3/4-2
Answer:-0.50
```

2. Write a Lex program that recognizes C language identifiers. Identifiers start with a letter or underscore and can contain letters, digits, and underscores. However, the program should *only* recognize identifiers that are not followed by a parenthesis (. This can be used to distinguish variable names from function calls in a simple C-like language. Use the / operator for this contextual check.

Input	**Output**
variable_name	Identifier: variable_name
_myVar123	Identifier: _myVar123
x_y_z	Identifier: x_y_z

Table 3: Valid Identifiers Recognized by the Program

Input	**Explanation**
funcName(	Function call (ignored due to '(‘).
123variable	Cannot start with a digit.
main ()	Function name ignored due to '(‘.

Table 4: Inputs Not Recognized as Identifiers

3. Simplified Markdown to HTML Converter:

   Goal: Create a Lex program that converts a simplified Markdown-like text to HTML. Focus on handling headings (H1-H3), bold text, italic text, and paragraphs.

   States: Use states to distinguish between regular text and Markdown formatting. For example, you could have a TEXT state for regular text and states like HEADING, BOLD, and ITALIC to handle the different formatting elements.

   Input:

```
# My Title

This is a paragraph. **This is bold text.** *This is italic text.*

## A Subheading

Another paragraph.
```

Output:

```
<h1>My Title</h1>

<p>This is a paragraph. <b>This is bold text.</b> <i>This is italic
    text.</i></p>

<h2>A Subheading</h2>

<p>Another paragraph.</p>
```

4. Multi-line Comment Removal (even nested ones):

   Goal: Enhance the multi-line comment removal example from the previous response. The improvement is to handle nested multi-line comments correctly. This is a more challenging task.

   States: You'll likely need to use a counter along with states. When you enter a multi-line comment, increment the counter. When you encounter `*/`, decrement the counter. Only when the counter reaches 0 should you exit the COMMENT state. This will allow you to correctly skip nested comments.

   Input:

```
/* Outer comment
 /* Nested comment */
 More text in outer comment
*/
int x = 10; // Some code
```

   Output:

```
int x = 10; // Some code
```

****