COMPILER DESIGN LABORATORY **Premises : IIIT Ranchi, BSNL 1st Floor**
Saturday, Feb 10, 2024

Lab Assignment 2 :
Objective : To get familiar with the use of Lex, the Lexical Analyzer generator tool. Write regular expressions for a few basic tokens and generate the lexical analyzer through Lex. Experiment with the generated scanner and perform experiments to understand its working. Augment the scanner to add more functionality.

Practice Problems

Note that after logging into the system, you have to save all your work undertaken in a laboratory session in a separate directory, as you had done in the first lab.

As soon as you have completed a problem call one of us to demonstrate the working. It would be goo for you to create a document recording your findings of each experiment in a separate doc or text file for future use. The commands to be used for today's Lab are given in the **Handout** at the end of this document.

P1. You are given a lex script in the file "**exp1.l**" shared through WhatsApp. Generate a scanner by performing the steps given at the end of this document, and give the executable a suitable name, such as "scan1".

There are several inputs given to you for experimentation, these are also uploaded as files with names, "inp1", "inp2", "inp3", "inp4".

input1	**input2**	**input3**	**input4**
ababcabc	aba b ab bbbba	ababab bbbba	abababbbbba

Run your generated scanner on all the 4 inputs and examine the output generated by the scanner. Explain with reasons the differences, if any, among the 4 outputs.

P2. Perform the change stated below in the file "**exp1.l**" while retaining all the other statements without change. Save the changed script file by a separate name, say, "**exp2.l**"

Existing Code	Changed Code
a*b {printf("Token 1 found \n");} c+ {printf("Token 2 found \n");}	a*b {printf("Token 1 found \n");return 1;} c+ {printf("Token 2 found \n");return 2;}

Now repeat the steps of generating a fresh scanner, give a different name, such scan2 and then use it to recognize tokens in all the 4 input files. Record your findings and explain the cause of the behavior of the scanner.

P3. Perform the change stated below in the file "**exp2.l**" while retaining all the other statements without change. Save the changed script file by a separate name, say, "**exp3.l**".

Existing Code	Changed Code
int main() { yylex(); }	int main() { while(yylex()); }

Redo all the tasks stated in P2 for this script file, "**exp3.l**"

P4. You are permitted to make minimal changes to the script file, "**exp3.l**" and create a new file, "**exp4.l**" such that the output produced by the scanner generated from **exp4.l** is of the form as shown below.

Behavior of scanner generated from "**exp3.l**" on input1 : **ababcabc**	Behavior of scanner generated from "**exp4.l**" on input1 : **ababcabc**
Token 1 found Token 1 found Token 2 found Token 1 found Token 2 found	Token : ab found Token : ab found Token : c found Token : ab found Token : c found

Run your scanner over all the 3 remaining inputs and explain your findings.

P5. If time permits, you can experiment with the functioning of the scanner by creating your own input files and / or modifying the lex script to change the behavior to your liking.

End of Laboratory 2

LABORATORY HANDOUT

Steps to Generate a Lexical Analyzer Using Lex (or Flex)

Step 1 : Run the lex script through lex as shown below.

```
$ lex exp1.l
```

In case there are no errors in the script, lex generates a C program in the file named as "lex.yy.c". You can read the contents of this generated file now or later.

Step 2 : Compile the lex.yy.c program using gcc; you may need to provide the lex library, using the option ll (read as hyphen el el where 'hyphen el' denotes reference to a library and the last 'el' stands for lex). If your environment has already included the lex library, you can skip the "-ll" option.

```
$ gcc lex.yy.c -ll -o scan1
```

On successful compilation the executable code is generated by gcc and it has been named as "scan1" using option "-o" and "scan1" is our executable scanner.

Step 3: For ease of experimentation, you may use relevant features of the shell, such as i/o redirection for using scan1 over the different input files.

```
$ ./scan1 < inp1
```

in which case the output of the scanner is displayed on screen. In case you want to save the output of the scanner in a file, use $ ./scan1 < inp1 > out1

You can easily save the inputs and outputs together in your directory so that your efforts during the lab session is recorded for future.

File i/o using the Shell

A small digression for those who are NOT familiar with shell programming. There are 3 operators used by the shell for i/o redirection (redirecting the standard input, standard output and standard error files to other files chosen by an user rather than their default binding to device files.

Operator	Usage	Semantics
<	$ ./a.out < file1	The input required by a.out will be read from "file1" when the program reads using scanf() or equivalent functions for reading.
> or 1>	$ ./a.out > file2 $ ./a.out 1> file2	The output produced by a.out, using printf() or equivalent, will be written to "file2"
2>	$ ./a.out 2> file3	The error messages produced by a.out, using the standard error channel, will be written to "file3"
>> or 2>>	$ ./a.out >> file4	Same effect as > or 2> except that the data is

	$ ./a.out 2>> file5	appended to the files instead of overwriting.