Compiler Construction → Project Phase 1
Name: Ali Ahmad
Roll.NO: L1F22BSCS0800
Section: G5

Objective

Phase 1 of the Compiler Construction project will consist of the design and implementation of a lexical analyzer (**URDU-INSPIRED MINICPPX LEXICAL ANALYZER**) in Flex to a Mini C++-like language. This lexical analyzer will identify and categorize different tokens that are identifiers, numbers, (integers, floats, and exponential forms), strings, and characters. It also recognizes a special group of **keywords** like agar, warna, jabtak, likho, lo, kaam, wapas, shuru, khatam, pura, dosra, lafz, jab, ruk and jarri, and **operators** =, +, -, *or/ /, ==, AND, OR and **punctuations** such as {, } (,),, semicolon, semicolon, colon.

The URDU-INSPIRED MINICPPX LEXICAL ANALYZER also records line numbers of every token and offers extensive error handling, which reports invalid tokens and their line number. The project focuses on originality and uniqueness and each student is free to establish his/her own set of keywords, operators, and punctuations that will design and implement his /her personal one. The final objective is to generate coherent and significant tokenized output, which would reflect a viable grasp of the concepts of lexical analysis in compiler production.

Project Description

You are required to implement a **scanner (**URDU-INSPIRED MINICPPX LEXICAL ANALYZER**)** using **Flex** that performs the following tasks:

1. **Tokenization**

- Recognize and classify tokens into:

Identifiers

Identifier Example	Regex Used in Code	Meaning in C	Notes
`_var1`	`_[a-zA-Z0-9_]+`	Variable names starting with _	In C, identifiers can start with letters or underscore.
`myVar , count123`	`[a-zA-Z][a-zA-Z0-9_]*`	Normal variable names Variable with numbers after first letter	Matches typical C variable naming rules. Digits cannot be first character in C identifiers

Numbers (integers, floats, exponential forms)

Number Example	Regex Used in Code	Meaning in C	Notes
`123`	`[0-9]+`	Integer constant	Matches C integer literals
`12.34`	`[0-9]+\.[0-9]+`	Floating-point constant	Matches C float literals

1.23e4	[0-9]+\.[0-9]+e[0-9]+	Exponential / scientific notation	Matches C float with exponent
5e6	[0-9]+e[0-9]+	Exponential	Same as C scientific notation

Keywords (defined individually by each student, see below)

Keyword	**Regex Used in Code**	**Meaning in C**	**Notes**
agar	"agar"	if	Conditional statement
warna	"warna"	else	Else block
jabtak	"jabtak"	while	Loop
likho	"likho"	printf / output	Custom output function
lo	"lo"	Assignment / operation start	Custom usage
kaam	"kaam"	Function start	Custom usage
wapas	"wapas"	return	Return statement
shuru	"shuru"	{ start block	Begin block
khatam	"khatam"	} end block	End block
pura	"pura"	End program / end main	Custom
dosra	"dosra"	else if	Conditional alternate
lafz	"lafz"	string / word	Custom keyword
jab	"jab"	for	Loop start
ruk	"ruk"	break	Loop break
jarri	"jarri"	continue	Loop continue

Operators

Operator	**Regex Used in Code**	**Meaning in C**	**Notes**
=	"="	Assignment	Assign value to variable
==	"=="	Equality check	Compare two values
+	"+"	Addition	Arithmetic operator
-	"-"	Subtraction	Arithmetic operator
*	"*"	Multiplication	Arithmetic / pointer operator
/	"/"	Division	Arithmetic operator
&&	"&&"	Logical AND	Boolean operator
`		`	`"
!	"!"	Logical NOT	Boolean operator
++	"++"	Increment	Increase by 1
--	"--"	Decrement	Decrease by 1
&	"&"	Bitwise AND / Address	Logical AND or address-of
`	`	`"	"`
^	"^"	Bitwise XOR	Exclusive OR
~	"~"	Bitwise NOT	Complement operator
<<	"<<"	Left shift	Shift bits left
>>	">>"	Right shift	Shift bits right
+=	"+="	Add and assign	Shorthand operator
-=	"-="	Subtract and assign	Shorthand operator
*=	"*="	Multiply and assign	Shorthand operator
/=	"/="	Divide and assign	Shorthand operator
%	"%"	Modulus	Remainder operator
%=	"%="	Modulus assign	Shorthand operator
&=	"&="	Bitwise AND assign	Shorthand

`	=`	`"	="`
^=	"^="	Bitwise XOR assign	Shorthand
<<=	"<<="	Left shift assign	Shorthand
>>=	">>="	Right shift assign	Shorthand

Punctuations

Punctuation	Regex Used in Code	Meaning in C	Notes
;	";"	Semicolon	End statement
,	","	Comma	Separate items / arguments
(	"("	Left parenthesis	Start expression / function args
)	")"	Right parenthesis	End expression / function args
{	"{"	Left brace	Start block
}	"}"	Right brace	End block
[	"["	Left bracket	Array start
]	"]"	Right bracket	Array end
:	":"	Colon	Labels / ternary operator
.	"."	Dot	Member access / structure
->	"->"	Arrow	Pointer to structure member

Strings and Characters

Example	Regex Used in Code	Meaning in C	Notes
"Hello World"	"[^\\n"]*"	String literal	Double-quoted sequence
"Sample Text"	"[^\\n"]*"	String literal	Same as above
'a'	\'[^\\'\n]\'	Character literal	Single character
'Z'	\'[^\\'\n]\'	Character literal	Single character
'\\n'	\'[^\\'\n]\\[^\\'\n]\'	Escape character	Special characters

Each token must be printed in the format:
`Line <n>: <TOKEN_TYPE> → <lexeme>`

2. **Line Tracking**
   Report the line number for each token.

Source Code Line	Token Output

`agar x == 10;`	`Line 1: KEYWORD → agar`
	`Line 1: IDENTIFIER → x`
	`Line 1: EQUAL_OPERATOR → ==`
	`Line 1: INTEGER → 10`
	`Line 1: SEMICOLON`- Punctuation `→ ;`

`lo count += 5;`	`Line 2: KEYWORD → lo`
	`Line 2: IDENTIFIER → count`
	`Line 2: ADD_ASSIGN_OPERATOR → +=`
	`Line 2: INTEGER → 5`
	`Line 2: SEMICOLON`- Punctuation `→ ;`

3. **Error Handling**
    - Invalid tokens must be reported as errors with their line numbers.
    - Example:
    - `Line 7: ERROR → @salary (invalid identifier)`

Source Code Line	**Token Output**
@salary = 5000;	Line 7: ERROR → @salary (invalid identifier - cannot start with special characters) Line 7: ASSIGNMENT_OPERATOR → = Line 7: INTEGER → 5000 Line 7: SEMICOLON → ;
likho "Hello World	Line 10: KEYWORD → likho Line 10: ERROR → "Hello World (unterminated string)

4. **Uniqueness Requirement**
    To make sure that the originality of the individual is achieved, each student has to develop his/her own set of keywords, operators and punctuations, and write a unique Mini C++ test program. That prevents submissions of the same work and it shows an individual grasp of lexical analysis.

In this project:

- **Keywords (15 chosen):**
  `agar`, `warna`, `jabtak`, `likho`, `lo`, `kaam`, `wapas`, `shuru`, `khatam`, `pura`, `dosra`, `lafz`, `jab`, `ruk`, `jarri`
- **Operators (examples):**
  `=`, `+`, `+=`, `&&`, `||`
- **Punctuations (examples):**
  `{`, `}`, `(`, `)`, `;`, `,`
- A **unique Mini C++ test program** (minimum 20 lines) has been written using these chosen keywords, operators, and punctuations. This program demonstrates **variable declaration, arithmetic operations, loops, conditionals, input/output, and error-free execution**.

```
shuru lo

  kaam a = 5;

  kaam b = 15;

  kaam result = 0;

  agar a > b lo

    likho "a is greater than b";

  khatam

  warna

    likho "b is greater than or equal to a";
    khatam

    kaam count = 0;

    jabtak count < 5 lo

        likho "Count: ";

        likho count;

        count++;

    khatam

    kaam arr[3] = {10, 20, 30};

    kaam i = 0;

    jab i < 3 lo

        result += arr[i];

        i++;

    khatam

    likho "Sum of array elements: ";

    likho result;

    kaam ch = 'Z';

    likho "Character is: ";

    likho ch;

    wapas 0;

khatam
```

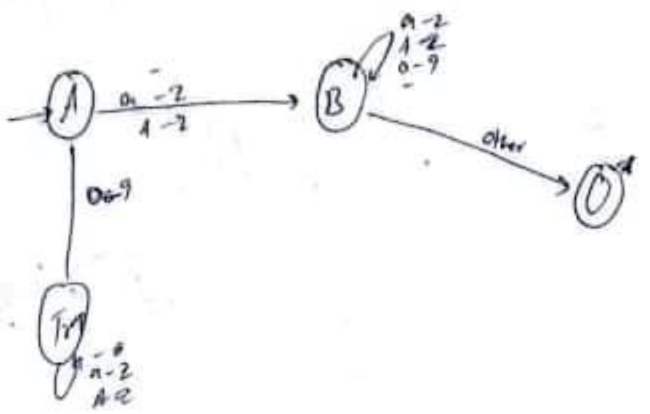
5. **Documentation**

Provide a table of regex definitions for your tokens.

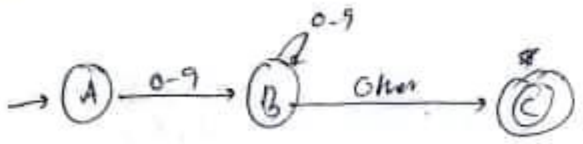
Token Type	**Regex (as used in code)**	**Example**	**Notes / C Equivalent**
Identifier	`[a-zA-Z][a-zA-Z0-9_]* _[a-zA-Z0-9_]+`	`x`, `_var1`, `count123`	Variable or function names; must start with letter or _.
Integer	`[0-9]+`	`10`, `500`	Integer literals in C.
Float	`[0-9]+\.[0-9]+`	`12.34`, `0.5`	Floating-point numbers in C.
Exponential	`[0-9]+e[0-9]+ [0-9]+\.[0-9]+e[0-9]+`	`1e5`, `2.5e3`	Scientific notation in C.
Keyword	`"agar" "warna" "jabtak" "likho" "lo" "kaam" "wapas" "shuru" "khatam" "pura" "dosra" "lafz" "jab" "ruk" "jarri"`	`agar`, `likho`	Custom keywords designed for Mini C++. Equivalent to `if`, `while`, `printf`, `return`, etc.
Assignment Operator	`=`	`x = 5`	Assigns value in C.
Arithmetic Operators	`+`, `-`, `*`, `/`	`x + y`	Standard arithmetic operations.
Compound Assignment	`+=`, `-=`, `*=`, `/=`	`sum += arr[i]`	Shorthand arithmetic operations.
Logical Operators	&&, `		, !`
Comparison Operators	`==`, `!=`, `<`, `>`, `<=`, `>=`	`x == y`	Standard relational operators in C.
Bitwise Operators	&, `	, ^, ~, <<, >>`	`a & b`
Punctuations	`{`, `}`, `(`, `)`, `;`, `,`, `[`, `]`, `:`, `.`	`{`, `;`, `[`	Separate statements, define blocks, array access, etc.
String	`"[^\\n"]*"`	`"Hello World"`	Double-quoted sequence.
Character	`'[^\\'\n]' '[^\\'\n]\\[^\\'\n]'`	`'a'`, `'\n'`	Single character or escape sequence.

Draw finite automata (FA) diagrams for Identifiers and Numbers (hand-drawn or digital).

Identifier:- [a-z A-Z _] [a-z A-Z 0-9 _]*

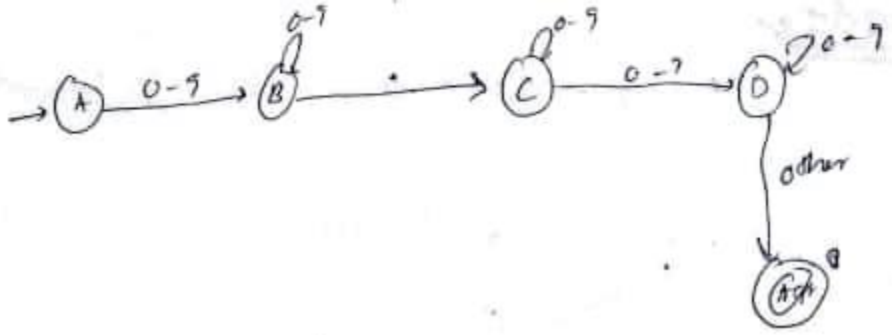


Integer:- (number) [0-9]+



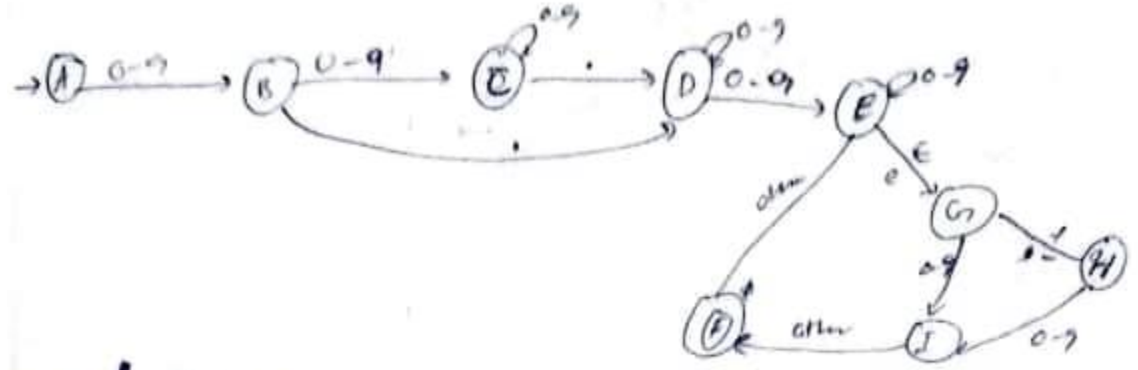
Float:-

[0-9]+ \. [0-9]+



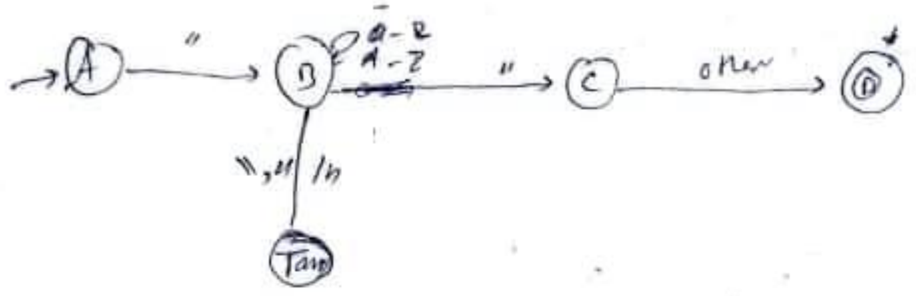
Float with Exponent:-

R.E = [0-9]$^+$ \. [0-9]$^+$ [/E•][+ -]? [0-9])



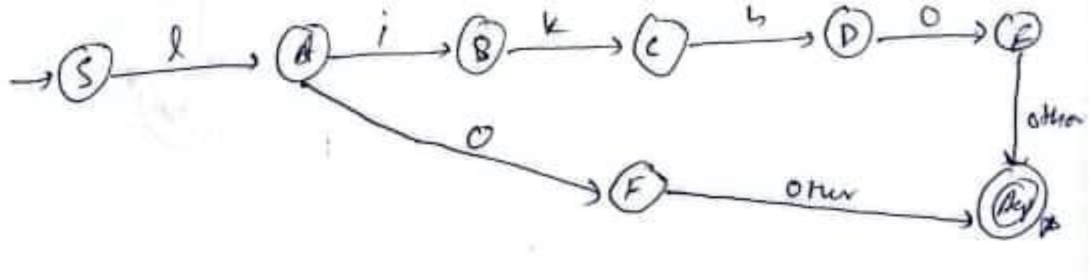
valid String:-

"[^\\n"]*"



keywords:-

I design a FA using the "like", for keywords



Include a brief explanation of your choice of 15 keywords and why you designed them.

Keyword	Reason / Meaning
`agar`	Represents `if` statements; easy to remember and related to conditional flow.
`warna`	Represents `else` branch in conditions.
`jabtak`	Represents `while` loops; controls iteration.
`likho`	Output/print statement; similar to `printf`.
`lo`	Marks the start of a block or operation.
`kaam`	Used for declaring variables or functions.
`wapas`	Return statement in functions.
`shuru`	Marks the beginning of a program or main block.
`khatam`	Marks the end of a block or program.
`pura`	Ends the main execution or program.
`dosra`	Represents `else if` branch.
`lafz`	Denotes strings or textual content.
`jab`	Represents `for` loops; iteration control.
`ruk`	Represents `break` to exit loops early.
`jarri`	Represents `continue` to skip to next iteration.

Explanation:

These 15 keywords are native, culturally motivated (Urdu-like) and are correlated to regular C operations.
They allow the student to write Mini C++ programs using their own language, and to be compatible with the lexical analyzer.

Deliverables

1. **Document (PDF)**
    1. Regex definitions (in table form).
    2. Transition diagrams for Identifiers and Numbers.
    3. Explanation of chosen 15 keywords + operators + punctuations.
2. **Source Code**
    - i. Flex file (minicppx`.l`).
    - ii. Unique sample Mini C++ test program (minimum 20 lines).
3. **Outputs**
    - i. Token output file (`tokens.txt`).
    - ii. Error log (if applicable).

4. **Demo Video (5 minutes max)**

   i. Demonstrate running your scanner on your unique program.
   ii. Explain how your regex works for at least 2 tokens.
   iii. Highlight one error-handling example.

Rubrics (10 Marks)

Criteria	Marks	Description
Regex Definitions & Correctness	2	Regex table is complete, accurate, and handles edge cases.
Flex Implementation (Tokenization)	2	Scanner correctly recognizes tokens and prints them with line numbers.
Error Handling	1	Invalid tokens are reported with line numbers.
Uniqueness & Creativity	3	Student-designed 15 keywords + operators + punctuations, plus unique sample program.
Diagrams (FA for Identifier & Number)	1	Clear, correct diagrams provided.
Demo & Explanation (Video& Viva)	1	Student demonstrates execution and explains regex.

Important Note on Individual Contribution

- Each student must **design their own set of 15 keywords**, **3-5 operators**, and **2-4 punctuations**, **Identifiers**, **Numbers.**
- Each student must create their **own sample test program** (at least 20 lines).
- Identical submissions will not be accepted.

A short viva/demo will verify authorship.