AUTOMATA THEORY & COMPILER SESS.
LECTURE 01

SESSION OVERVIEW

- **Topic**: Introduction to Flex and Bison

  Goal: Set up environment and implement a basic lexical and syntax analyzer

- **By the end of this session, students will:**
  - Understand what Flex and Bison are
  - Set up Flex and Bison on Windows (Code::Blocks + MinGW)
  - Write and run a simple program that parses and evaluates arithmetic assignments

WHAT IS COMPILER?

A **compiler** is a special program that translates a high-level programming language (like C, Java) into **machine code** (low-level language) that a computer can understand.

- Example: int a = b + c; → converted into assembly/machine code

PHASES OF A COMPILER

Phase	Description
1. Lexical Analysis	Breaks code into **tokens** (keywords, identifiers) using patterns
2. Syntax Analysis	Checks grammar using **parsing** (derives parse tree)
3. Semantic Analysis	Checks meaning (e.g., type checks)
4. Intermediate Code Generation	Creates platform-independent code
5. Optimization	Improves performance, removes redundancy
6. Code Generation	Converts into target language (e.g., x86)
7. Code Linking & Assembly	Final binary or executable

WHAT IS FLEX?

- **Flex** stands for **Fast Lexical Analyzer Generator**.
- It generates a **scanner** (lexer) from .l files.
- It breaks input text into **tokens** using **regular expressions**.
- Think of it like "reading code word by word.“
- Example:
  - [0-9]+ → NUMBER
  - [a-zA-Z_][a-zA-Z0-9_]* → IDENTIFIER

WHAT IS BISON?

- **Bison** is a **parser generator**.
- It processes the tokens provided by Flex and applies **grammar rules**.
- It helps us understand **structure and hierarchy** in code.
- Example:
  - expression → expression + expression

REGULAR EXPRESSION (REGEX)

- A compact way to describe **patterns** in text.
- Used in **lexical analysis** to match identifiers, numbers, etc.

Pattern	Matches
[0-9]+	Numbers
[a-zA-Z_][a-zA-Z0-9_]*	Identifiers
[\t\n]+	Whitespace

COMMON REGEX SYMBOLS USED IN LEXICAL ANALYSIS

Symbol	Meaning	Example	Matches
[]	Matches **any one character** inside the brackets	[abc]	'a', 'b', or 'c'
- (inside [])	Denotes a **range of characters**	[0-9]	Any digit from 0 to 9
+	Matches **one or more** of the previous pattern	[0-9]+	5, 123, 007
*	Matches **zero or more** of the previous pattern	[a-z]*	"", hello, abcde
.	Matches **any single character** (except newline)	.	a, 1, #, etc.
^ (inside [])	**Negates** the character class	[^0-9]	Any non-digit character
?	Matches **zero or one** of the previous pattern	[0-9]?	"", 5
()	Groups expressions	(ab)+	ab, abab, ababab
|	Logical OR (either this or that)	a|b	'a' or ‘b’ but not both at the same time

EXAMPLE: A SIMPLE STATEMENT PARSER

Let's say the user writes the following code:

x = 5 + 3;

Step 1: Lexical Analysis using Flex

Step 2: Syntax Analysis using Bison

STEP 1: LEXICAL ANALYSIS USING FLEX

Flex reads the input **character by character** and breaks it into **tokens** using regular expressions.

Input	Token Type	Value
x	IDENTIFIER	"x"
=	SYMBOL	'='
5	NUMBER	5
+	SYMBOL	'+'
3	NUMBER	3
;	SYMBOL	';'

STEP 1: LEXICAL ANALYSIS USING FLEX

This is handled by rules in scanner.l, like:

- [0-9]+ → return NUMBER;
- [a-zA-Z_][a-zA-Z0-9_]* → return IDENTIFIER;
- "+" → return '+';

STEP 2: SYNTAX ANALYSIS USING BISON

Bison takes these tokens and matches them against grammar rules to check if the structure is valid.

The rule:

statement:
    IDENTIFIER '=' expression ‘;’

For expression:
expression:
    NUMBER | expression '+' expression

So, x = 5 + 3; matches:
    IDENTIFIER '=' (NUMBER '+' NUMBER) ';’
    → OK!

Output:
    Assign 8 to x

SUMMARY

- **Flex**: Fast lexical analyzer generator (tokenizes input text)
- **Bison**: GNU parser generator (parses structured input)
- Flex ≈ Lex (Lexical analysis)
- Bison ≈ Yacc (Yet Another Compiler Compiler)
- Used in compiler front-end (lexical + syntactic analysis)

SYSTEM REQUIREMENTS

- Code::Blocks IDE (with MinGW)
- win_flex & win_bison (Windows ports of Flex & Bison)
- Basic understanding of C

ENVIRONMENT SETUP

- **Install Code::Blocks** (with MinGW)
- **Download win_flex_bison** (e.g., https://github.com/lexxmark/winflexbison/releases)
- **Extract the ZIP**, copy these files to C:\ :
  - win_flex.exe
  - win_bison.exe
- **Add path to Environment Variables**:
  - Control Panel → System → Environment Variables
  - Edit Path → Add path to folder containing win_flex.exe and win_bison.exe

VERIFY INSTALLATION (COMMAND PROMPT)

```
win_flex --version
win_bison --version
```

Both should show version info

CREATING THE PROJECT

- Open Code::Blocks → File → New → **Console Application**
- Choose **C language**
- Project name: Week01FlexBison (Anything else)
- Save in desired location (E:\ATC Sessional\Week01FlexBison)

CREATE INPUT FILES

Create two new files inside the project directory:

- scanner.l → the Flex file (lexer)
- parser.y → the Bison file (parser)

Do not compile these files directly — we'll use Flex and Bison manually.

SAMPLE SCANNER.L

Header Section

```
%{
#include "parser.tab.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}
```

This header is generated by Bison and contains **token definitions** (like NUMBER, IDENTIFIER) and yylval.

Standard libraries for I/O, memory, and string manipulation.

Rules Section

```
%%
[0-9]+       { yylval.ival = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.sval = strdup(yytext); return IDENTIFIER; }
"+"          { return '+'; }
[ \t\n]+     ;   // skip whitespace
.            { return yytext[0]; }
%%
```

SAMPLE SCANNER.L

- yytext: A global char* holding the **matched text** (e.g., "123").
- atoi(yytext): Converts the matched string to an integer.
- yylval.ival: Stores the integer in the **yacc (Bison) semantic value union**.
- return NUMBER: Returns the token NUMBER to the parser.
- strdup(yytext): Duplicates the string (allocates memory and copies text).
- yylval.sval: Stores the string value for the parser to use.
- return IDENTIFIER: Returns IDENTIFIER token to Bison.
- Returns '+' as a character token (Bison treats single-character tokens like +, =, ; directly).
- return yytext[0];: Returns the actual character matched. Useful for operators like =, ;, *, etc. that are **handled directly by their character** in the parser.

SAMPLE PARSER.Y

Header Section

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int yylex(void);
void yyerror(const char *s) { fprintf(stderr, "Error: %s\n", s); }
int yywrap(void) { return 1; }
%}
```

- `int yylex(void);` Declaration of the scanner function (defined in scanner.l).
- `yyerror` Called by Bison to report **syntax errors**.
- `yywrap` Tells the scanner when to stop reading input (1 = no more input).

Declare the Union for Semantic Values

```
%union {
    int ival;
    char* sval;
}
```

Defines a **union** named yylval to carry semantic values between lexer and parser.

Declare Tokens and Types

```
%token <ival> NUMBER
%token <sval> IDENTIFIER
%type <ival> expression
```

- Tells Bison that the NUMBER token carries an int value.
- Tells Bison that IDENTIFIER carries a char* (string).
- The non-terminal expression will evaluate to an int.

Operator Precedence and Associativity

```
%left '+'
```

Defines + as a left-associative operator.

Grammar Rules Section

```
%%
program: /* empty */ | program statement ;
statement: IDENTIFIER '=' expression ';' {
    printf("Assign %d to %s\n", $3, $1); free($1);
};
expression:
    NUMBER                          { $$ = $1; }
  | IDENTIFIER                      { $$ = 0; }
  | expression '+' expression { $$ = $1 + $3; }
%%
int main(void) { return yyparse(); }
```

This is the **start symbol**. It defines a program as: **empty**, or sequence of **statements**. This allows parsing multiple statements.

- $1 = the identifier (e.g., "x")
- $3 = the result of the expression on the right-hand side (e.g., 5)
- free($1);: Frees the dynamically allocated memory from strdup() in the sc

- `NUMBER`: A single number just evaluates to its value.
- `IDENTIFIER`: Treated as 0 for simplicity
- `expression '+' expression`: Evaluates a + b by adding the two sub-expressi

`int main(void)`:

- The entry point of the program.
- yyparse() starts the parsing process, calling yylex() to get tokens and applying grammar rules.

GENERATE OUTPUT FILES

- From CMD (in project folder):
  win_bison -d parser.y → generates parser.tab.c, parser.tab.h
  win_flex scanner.l → generates lex.yy.c

- Add these 2 files to Code::Blocks project:
  - parser.tab.c
  - lex.yy.c

- Delete default main.c

BUILD AND RUN

Input:

x = 5 + 10;

a = 4;

Output:

Assign 15 to x

Assign 4 to a

FUNCTIONS SUMMARY

Function	Role	Defined Where	Called By
yylex()	Returns tokens using regex	Generated by Flex (scanner.l)	Called by yyparse()
yyparse()	Main parser loop	Generated by Bison (parser.y)	Called by main()
yyerror()	Handles syntax errors	Defined in parser.y	Called by yyparse()
yywrap()	Signals end of input	Defined	Called by yylex()

HOW IT WORKS?

```
main() → yyparse()
               ↓
           yylex()
               ↓
     Match token using regex
               ↓
     Return token → yyparse()
               ↓
     Apply grammar → call actions
```

SUMMARY

- Concepts:
  - Lexical vs Syntactic Analysis
  - Tokenizing with Flex
  - Parsing with Bison

- Skills:
  - Installed & configured tools
  - Created and linked lexer/parser
  - Ran a working arithmetic parser