Experiment 1: Lexical Analyzer + Symbol Table (LEX)

Aim:

Develop a lexical analyzer using LEX to recognize identifiers, constants, comments, operators, and build a symbol table while recognizing identifiers.

Simple Algorithm:

1. Read source code input.
2. Ignore whitespace and comments.
3. Recognize tokens: keywords, identifiers, constants, operators, punctuations.
4. When an identifier is found, check if it is a keyword; if not, insert into the symbol table.
5. After scanning complete, display the symbol table.

Code (lexer .l):

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

struct Symbol {
   char name[20];
   char type[15];
   char kind[15];
   char scope[15];
   char memloc[15];
   char extra[30];
} symtab[100];

int count = 0;

int is_keyword(const char *word) {
   return (strcmp(word, "int") == 0 || strcmp(word, "float") == 0);
}

void insert_symbol(char *name, char *type, char *kind, char *scope, char *memloc, char *extra) {
   strcpy(symtab[count].name, name);
   strcpy(symtab[count].type, type);
   strcpy(symtab[count].kind, kind);
   strcpy(symtab[count].scope, scope);
   strcpy(symtab[count].memloc, memloc);
   strcpy(symtab[count].extra, extra);
   count++;
}

void display_symtab() {
   printf("\n%-15s %-12s %-12s %-10s %-15s %-20s\n",
          "Name", "Type", "Kind", "Scope", "Mem_Loc", "Extra_Info");
   printf("-------------------------------------------------------------------------------------\n");
   for (int i = 0; i < count; i++) {
      printf("%-15s %-12s %-12s %-10s %-15s %-20s\n",
```

```
            symtab[i].name, symtab[i].type, symtab[i].kind,
            symtab[i].scope, symtab[i].memloc, symtab[i].extra);
    }
}
%}

%%
int|float             { insert_symbol(yytext, "Keyword", "datatype", "global", "-", "reserved word"); }
[a-zA-Z_][a-zA-Z0-9_]* {
                        if (!is_keyword(yytext)) {
                            insert_symbol(yytext, "Identifier", "variable", "local", "0x100", "size=4 bytes");
                        }
                      }
[0-9]+                 { insert_symbol(yytext, "Constant", "literal", "local", "-", "integer constant"); }
[ \t\n]+               ;  /* Ignore spaces and newlines */
.                     ;
%%

int main() {
    printf("Enter the code press Ctrl+Z then Enter to end:\\n");
    yylex();
    display_symtab();
    return 0;
}

int yywrap() {
    return 1;
}
```

How to run (Windows - WinFlexBison):

1. win_flex.exe ex1.l
2. Compile lex.yy.c using MinGW: gcc lex.yy.c -o ex1.exe
3. Run: ex1.exe sample_input.c OR run ex1.exe and type code, finish with Ctrl+Z + Enter.

Sample Input:

```
int a;
a = 10;
float b;
x1 = a + 5;
```

Sample Output (example):

```
Enter the code press Ctrl+Z then Enter to end:
int a;
a = 10;
float b;
x1 = a + 5
```

Name	Type	Kind	Scope	Mem_Loc	Extra_Info
int	Keyword	datatype	global	-	reserved word
a	Identifier	variable	local	0x100	size=4 bytes
10	Constant	literal	local	-	integer constant
float	Keyword	datatype	global	-	reserved word
b	Identifier	variable	local	0x100	size=4 bytes
x1	Identifier	variable	local	0x100	size=4 bytes
a	Identifier	variable	local	0x100	size=4 bytes
5	Constant	literal	local	-	integer constant

Experiment 2: Lexical Analyzer (token classification with line numbers)

Aim:

Implement a lexical analyzer using LEX that prints tokens (keywords, identifiers, constants, operators, punctuation) with line numbers.

Simple Algorithm:

1. Read input file line by line.
2. On encountering a newline, increment line counter.
3. Match keywords, identifiers, constants, operators, punctuation and print them with the current line number.
4. Ignore whitespace.
5. Continue until EOF.

Code (lexer .l):

```
%{
#include <stdio.h>
int line_num = 1;
int printed_line = 0;  // Flag to track if line number printed this line

void print_line_num_if_needed() {
   if (!printed_line) {
      printf("Line %d:\n", line_num);
      printed_line = 1;
   }
}

int yywrap(void) {
   return 1;
}
%}

KEYWORD   int|float|char|void|if|else|while|for|return|printf|scanf|include|getch|main
OPERATOR  \+|\-|\*|\/|=|==|<=|>=|<|>|\(|\)|\{|\}|\[|\]|;|,|\"|\'|#

%%
```

```
\n          { line_num++; printed_line = 0; }
{KEYWORD}    { print_line_num_if_needed(); printf("\t%-12s : Keyword\n", yytext); }
[0-9]+       { print_line_num_if_needed(); printf("\t%-12s : Constant\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* { print_line_num_if_needed(); printf("\t%-12s : Identifier\n", yytext); }

"("          { print_line_num_if_needed(); printf("\t%-12s : open_paren\n", yytext); }
")"          { print_line_num_if_needed(); printf("\t%-12s : close_paren\n", yytext); }
"{"          { print_line_num_if_needed(); printf("\t%-12s : open_brace\n", yytext); }
"}"          { print_line_num_if_needed(); printf("\t%-12s : close_brace\n", yytext); }
";"          { print_line_num_if_needed(); printf("\t%-12s : semicolon\n", yytext); }
","          { print_line_num_if_needed(); printf("\t%-12s : comma\n", yytext); }
"="          { print_line_num_if_needed(); printf("\t%-12s : equal\n", yytext); }
"*"          { print_line_num_if_needed(); printf("\t%-12s : multiply\n", yytext); }
"\""         { print_line_num_if_needed(); printf("\t%-12s : doublequote\n", yytext); }
"'"          { print_line_num_if_needed(); printf("\t%-12s : singlequote\n", yytext); }
"#"          { print_line_num_if_needed(); printf("\t%-12s : preprocessor\n", yytext); }

[ \t]+       ; // ignore whitespace
.            { /* ignore other characters */ }

%%

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <inputfile>\n", argv[0]);
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }
    yyin = file;
    printf("\n--- Lexical Analysis ---\n\n");
    yylex();
    fclose(file);
    return 0;
}
```

How to run (Windows - WinFlexBison):

1. win_flex.exe ex2.l
2. Compile lex.yy.c with MinGW: gcc lex.yy.c -o ex2.exe
3. Run: ex2.exe sample_input.c

Sample Input (sample_input.c):

```
int main() {
    int a = 5;
    a = a + 2;
}
```

Sample Output (example):

```
--- Lexical Analysis ---

Line 1:
   int        : Keyword
   main       : Identifier
   (          : open_paren
   )          : close_paren
   {          : open_brace
Line 2:
   int        : Keyword
   a          : Identifier
   5          : Constant
   ;          : semicolon
Line 3:
   a          : Identifier
   =          : equal
   a          : Identifier
   +          : multiply
   2          : Constant
   ;          : semicolon
Line 4:
   }          : close_brace
```

Experiment 3a: Recognize Arithmetic Expressions (+ - * /) (LEX + YACC)

Aim:

Recognize valid arithmetic expressions using operators +, -, *, /.

Simple Algorithm:

1. Tokenize numbers and operators using Lex.
2. Use Yacc grammar to validate expressions recursively.
3. On success print valid message; on parse error print invalid message.

LEX File (expr.l):

```
%{
#include "y.tab.h"
%}
%%
[0-9]+          { yylval = atoi(yytext); return NUM; }
[+\-*/]         { return yytext[0]; }
\n              { return 0; }
[ \t]           { /* skip whitespace */ }
.               { printf("Invalid character: %s\n", yytext); }
%%
```

YACC File (expr.y):

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token NUM
%%
expr: expr '+' expr
   | expr '-' expr
   | expr '*' expr
   | expr '/' expr
   | NUM
   ;
%%
int main() {
   printf("Enter an expression: ");
   yyparse();
   printf("Valid expression.\n");
   return 0;
}
int yyerror(const char* s) {
   printf("Invalid expression\n");
   return 0;
}
```

How to run (Windows - WinFlexBison):

win_flex expr.l
win_bison -d expr.y
gcc lex.yy.c expr.tab.c -o expr.exe
expr.exe

Sample Input & Output:

Input: 3 + 5 * 2
Output:
Enter an expression:
Valid expression.

Input: 3 + * 2
Output:
Invalid expression

Experiment 3b: Recognize Valid Variable Names (LEX)

Aim:

Recognize variable names that start with a letter and followed by letters/digits.

Simple Algorithm:

1. Use a regular expression [a-zA-Z][a-zA-Z0-9]* to match valid variable names.
2. For inputs that do not match, print invalid message.

LEX File (var.l):

```
%{
#include <stdio.h>
%}
%%
[a-zA-Z][a-zA-Z0-9]*    { printf("Valid variable: %s\n", yytext); }
[^a-zA-Z]([a-zA-Z0-9])* { printf("Invalid variable: %s\n", yytext); }
\n                      { return 0; }
.                       { /* ignore */ }
%%
int main() {
  yylex();
  return 0;
}
```

How to run:

flex var.l
gcc lex.yy.c -o var
./var
Type a token and press Enter.

Sample Input & Output:

Input: var1
Output: Valid variable: var1

Input: 1var
Output: Invalid variable: 1var

Experiment 3c: Recognize Control Structures (LEX + YACC)

Aim:

Recognize C control structures: if, if-else, if-else-if, while, for, switch-case.

Simple Algorithm:

1. Lexical analyzer returns tokens for keywords and punctuation.
2. Yacc grammar recognizes the syntactic patterns for control structures and validates them.
3. On parse success print valid control structure message; otherwise report syntax error.

LEX File (control.l):

```
%{
#include "y.tab.h"
%}
%%
"if"          { return IF; }
"else"        { return ELSE; }
"while"       { return WHILE; }
"for"         { return FOR; }
"switch"      { return SWITCH; }
"case"        { return CASE; }
"default"     { return DEFAULT; }
"break"       { return BREAK; }
[a-zA-Z_][a-zA-Z0-9_]*   { return ID; }
[0-9]+                { return NUM; }
"{"|"}"|"("|")"|":"|";" { return yytext[0]; }
[ \t\n]+              { /* skip whitespace */ }
.             { return yytext[0]; }
%%
```

YACC File (control.y):

```
%{
#include <stdio.h>
%}
%token IF ELSE WHILE FOR SWITCH CASE DEFAULT BREAK ID NUM
%%
stmt: if_stmt
  | while_stmt
  | for_stmt
  | switch_stmt
  ;
if_stmt: IF '(' expr ')' stmt
    | IF '(' expr ')' stmt ELSE stmt
    ;
while_stmt: WHILE '(' expr ')' stmt ;
for_stmt: FOR '(' expr ';' expr ';' expr ')' stmt ;
switch_stmt: SWITCH '(' ID ')' '{' case_list default_stmt '}' ;
case_list: case_list CASE NUM ':' stmt BREAK ';'
      | CASE NUM ':' stmt BREAK ';'
      ;
```

```
default_stmt: DEFAULT ':' stmt BREAK ';'
        | /* empty */
        ;
expr: ID | NUM ;
%%
int main() {
  yyparse();
  printf("Valid control structure.\n");
  return 0;
}
int yyerror(const char *s) {
  printf("Invalid syntax.\n");
  return 0;
}
```

How to run (Linux):

flex control.l
bison -d control.y
gcc lex.yy.c control.tab.c -o control -lfl
./control < inputfile.c

How to run (Windows):

win_flex control.l
win_bison -d control.y
gcc lex.yy.c control.tab.c -o control.exe
control.exe

Sample Inputs & Outputs:

Input: if (x) { }
Output:
Valid control structure.

Input: while (a) { }
Output:
Valid control structure.

Invalid Input: if x) { }
Output:
Invalid syntax.

Experiment 3d: Calculator (LEX + YACC)

Aim:

Implement a simple calculator that evaluates arithmetic expressions using LEX for tokenizing and YACC for parsing and evaluation.

Simple Algorithm:

1. Tokenize numbers and operators using Lex.
2. Use Yacc grammar with precedence rules to evaluate expressions.
3. On each reduction compute value and print result.

LEX File (calc.l):

```
%{
#include "y.tab.h"
%}
%%
[0-9]+      { yylval = atoi(yytext); return NUM; }
[+\-*/]     { return yytext[0]; }
\n          { return 0; }
[ \t]       { }
%%
```

YACC File (calc.y):

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token NUM
%left '+' '-'
%left '*' '/'
%%
expr: expr '+' expr     { printf("= %d\n", $1 + $3); }
   | expr '-' expr      { printf("= %d\n", $1 - $3); }
   | expr '*' expr      { printf("= %d\n", $1 * $3); }
   | expr '/' expr      { printf("= %d\n", $1 / $3); }
   | NUM                { $$ = $1; }
   ;
%%
int main() {
   printf("Enter an expression:\\n");
   yyparse();
   return 0;
}
int yyerror(const char *s) {
   printf("Syntax error: %s\n", s);
   return 0;
}
```

How to run:

flex calc.l
bison -d calc.y
gcc lex.yy.c calc.tab.c -o calc -lfl
./calc
Type expression like: 8 + 2 * 5 and press Enter.

Sample Input & Output:

Input: 8 + 2 * 5
Output:
= 18

Input: 10 / 2
Output:
= 5

Experiment 4: Three-Address Code (TAC) Generation using LEX & YACC

Aim:

Generate three-address code for simple assignment and arithmetic expressions using Lex for tokens and Yacc for parsing and TAC emission.

Simple Algorithm:

1. Lex tokens (identifiers, numbers, operators, semicolon).
2. Yacc expression grammar performs post-order generation: for each binary op generate a new temporary t# and emit a quad (t# = a op b).
3. For assignment emit final 'var = temp' quad.
4. Print TAC sequence.

LEX File (tac.l):

```
%{
#include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z0-9_]*   { yylval.str = strdup(yytext); return ID; }
[0-9]+                   { yylval.str = strdup(yytext); return NUM; }
[=+\-*/;()]              { return yytext[0]; }
[ \t\n]                  { }
%%
```

YACC File (tac.y):

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int temp_count = 0;
char* new_temp() {
  char* temp = malloc(10);
```

```
  sprintf(temp, "t%d", temp_count++);
  return temp;
}
typedef struct {
  char* str;
} YYSTYPE;
#define YYSTYPE YYSTYPE
%}
%token ID NUM
%left '+' '-'
%left '*' '/'
%%
stmt: ID '=' expr ';' {
      printf("%s = %s\n", $1, $3.str);
  }
  ;
expr: expr '+' expr {
      char* temp = new_temp();
      printf("%s = %s + %s\n", temp, $1.str, $3.str);
      $$ = (YYSTYPE){temp};
  }
  | expr '-' expr {
      char* temp = new_temp();
      printf("%s = %s - %s\n", temp, $1.str, $3.str);
      $$ = (YYSTYPE){temp};
  }
  | expr '*' expr {
      char* temp = new_temp();
      printf("%s = %s * %s\n", temp, $1.str, $3.str);
      $$ = (YYSTYPE){temp};
  }
  | expr '/' expr {
      char* temp = new_temp();
      printf("%s = %s / %s\n", temp, $1.str, $3.str);
      $$ = (YYSTYPE){temp};
  }
  | ID {
      $$ = $1;
  }
  | NUM {
      $$ = $1;
  }
  ;
%%
int main() {
  yyparse();
  return 0;
}
int yyerror(const char* s) {
  printf("Syntax Error: %s\n", s);
  return 0;
}
```

How to run:

flex tac.l
bison -d tac.y
gcc lex.yy.c tac.tab.c -o tac -lfl
./tac
Provide an input like: a = b + c * d; and press Enter (or use a file).

Sample Inputs & Outputs:

Input: a = b + c * d;
Output:
t0 = c * d
t1 = b + t0
a = t1

Input: z = x - y / w;
Output:
t2 = y / w
t3 = x - t2
z = t3