Department of CSE

Laboratory Manual	
Course:	B.Tech.
Year & Semester:	IV – I
Class:	CSE
Subject:	**Compiler Design Lab Manual**
Regulation:	R22

BALAJI INSTITUTE OF TECHNOLOGY AND SCIENCE (AUTONOMOUS)

B.Tech (Department of Computer Science & Engineering)

COMPILER DESIGN LAB

Course Outcomes:

- Design, develop, and implement a compiler for any language.
- Use lex and yacc tools for developing a scanner and a parser.
- Design and implement LL and LR parsers.
- Implement Type checking.
- Implement storage allocation strategies.

.

List of Experiments:

1. Implementation of symbol table.
2. Develop a lexical analyzer to recognize a few patterns inc (ex. Identifiers, constants, comments, operators etc.)
3. Implementation of lexical analyzer using lex tool.
4. Generate yacc specification for a few syntactic categories.
   a) Program to recognize a valid arithmetic expression that uses operator +,-, * and /. Program to recognize a valid variable which starts with a letter followed by any number digits.
   b) Implementation of calculator using lex and yacc.
5. Convert the bnf rules into yacc form and write code to generate abstract syntax tree.
6. Implement type checking
7. Implement any one storage allocation strategies (heap, stack, static)
8. Write a lex program to count the number of words and number of lines in a given file or program.
9. Write a ‘C’ program to implement lexical analyzer using c program.
10. write recursive descent parser for the grammar
    E->E+T E->T
    T->T*F T->F F->(E)/id.
11. write recursive descent parser for the grammar
    S->(L) S->a
    L->L,S L->S
12. Write a C program to calculate first function for the grammar E->E+T E->T T->T*F T->F F->(E)/id
13. Write a YACC program to implement a top down parser for the given grammar.
14. Write a YACC program to evaluate algebraic expression.

ISO 9001:2015 Certified Institution
Estd.:2001
Balaji Institute of Technology & Science
Laknepally (V), Narsampet (M), Warangal District - 506 331, Telangana State, India
(AUTONOMOUS)
Accredited by NBA (UG - CE, ME, ECE & CSE) **& NAAC A+ Grade**
(Affiliated to JNT University, Hyderabad and Approved by AICTE, New Delhi)
BITS AUTONOMOUS **www.bitswgl.ac.in, email: principal@bitswgl.ac.in, Ph:98660 50044, Fax: 08718-230521**

Department of Computer Science & Engineering

COMPILER DESIGN

LAB MANUAL

No.	List of Experiments
1	Implementation of Symbol Table
2	Develop a Lexical Analyzer
3	Implementation of Lexical Analyzer Using Lex Tool
4	Generate YACC Specification for Syntactic Categories
5	Convert BNF Rules into YACC Form and Generate AST
6	Implement Type Checking
7	Implement Storage Allocation Strategies
8	Lex Program to Count Words and Lines
9	Implement Lexical Analyzer in C
10	Recursive Descent Parser for Arithmetic Grammar
11	Recursive Descent Parser for Custom Grammar
12	C Program to Calculate FIRST Set for Given Grammar
13	YACC Program for Top-Down Parser
14	YACC Program to Evaluate Algebraic Expressions

1. Implementation of Symbol Table

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define the structure for a symbol table entry
typedef struct Symbol {
    char name[50];
    char type[20];
    int memory_location;
    struct Symbol* next;
} Symbol;

// Symbol Table: Using a simple linked list
Symbol* symbolTable = NULL;
int memoryCounter = 0;  // To simulate memory allocation

// Function to insert into the symbol table
void insertSymbol(char* name, char* type) {
    Symbol* newSymbol = (Symbol*)malloc(sizeof(Symbol));
    strcpy(newSymbol->name, name);
    strcpy(newSymbol->type, type);
    newSymbol->memory_location = memoryCounter++;
    newSymbol->next = symbolTable;
    symbolTable = newSymbol;
}

// Function to search for a symbol
Symbol* searchSymbol(char* name) {
    Symbol* current = symbolTable;
    while (current != NULL) {
        if (strcmp(current->name, name) == 0)
            return current;
        current = current->next;
    }
    return NULL;
}

// Function to display the symbol table
void displaySymbolTable() {
    printf("Symbol Table:\n");
    printf("-----------------------------------------------\n");
    printf("| Name       | Type       | Memory Location  |\n");
    printf("-----------------------------------------------\n");
    Symbol* current = symbolTable;
    while (current != NULL) {
        printf("| %-10s | %-9s | %-15d |\n", current->name, current->type,
current->memory_location);
        current = current->next;
```

```
    }
    printf("-----------------------------------------------\n");
}

int main() {
    // Insert some symbols
    insertSymbol("x", "int");
    insertSymbol("y", "float");
    insertSymbol("sum", "function");

    // Search for symbols
    Symbol* s = searchSymbol("y");
    if (s != NULL) {
        printf("Found symbol: %s, Type: %s, Memory Location: %d\n", s->name,
s->type, s->memory_location);
    } else {
        printf("Symbol not found.\n");
    }

    // Display the symbol table
    displaySymbolTable();

    return 0;
}
```

OUTPUT:

```
Found symbol: y, Type: float, Memory Location: 1
Symbol Table:
-------------------------------------------------
| Name      | Type      | Memory Location  |
-------------------------------------------------
| sum       | function  | 2                |
| y         | float     | 1                |
| x         | int       | 0                |
-------------------------------------------------
```

2. Implementation of Lexical Analyzer Using Lex Tool

```
#include <stdio.h>
  #include <ctype.h>
  #include <string.h>

  #define MAX_TOKEN_SIZE 100

  // List of keywords
  char *keywords[] = {"int", "float", "if", "else", NULL};

  // Function to check if the token is a keyword
  int isKeyword(char *str) {
     for (int i = 0; keywords[i] != NULL; i++) {
        if (strcmp(str, keywords[i]) == 0)
           return 1;
     }
     return 0;
  }

  // Function to recognize tokens
  void getToken(char *str) {
     int i = 0;
     char token[MAX_TOKEN_SIZE];

     while (str[i] != '\0') {
        // Skip whitespace
        if (isspace(str[i])) {
           i++;
           continue;
        }

        // Identifiers/Keywords
        if (isalpha(str[i]) || str[i] == '_') {
           int j = 0;
           while (isalnum(str[i]) || str[i] == '_') {
              token[j++] = str[i++];
           }
```

```
            token[j] = '\0';

            if (isKeyword(token)) {
                printf("Keyword: %s\n", token);
            } else {
                printf("Identifier: %s\n", token);
            }
            continue;
        }

        // Numbers (Integers and Floats)
        if (isdigit(str[i])) {
            int j = 0;
            int hasDot = 0;
            while (isdigit(str[i]) || str[i] == '.') {
                if (str[i] == '.') {
                    hasDot = 1;
                }
                token[j++] = str[i++];
            }
            token[j] = '\0';

            if (hasDot) {
                printf("Float Constant: %s\n", token);
            } else {
                printf("Integer Constant: %s\n", token);
            }
            continue;
        }

        // Operators
        if (strchr("+-*/=", str[i])) {
            printf("Operator: %c\n", str[i]);
            i++;
            continue;
        }

        // Punctuation
        if (strchr("();,", str[i])) {
            printf("Punctuation: %c\n", str[i]);
            i++;
            continue;
        }

        // Unknown token
        printf("Unknown Token: %c\n", str[i]);
        i++;
    }
}
```

```
int main() {
    char code[] = "int main() { int x = 100; float y = 20.5; if (x > y) { x = x +
10; } }";

    printf("Lexical Analysis:\n");
    getToken(code);

    return 0;
}
```

OUTPUT:

```
Keyword: int
Identifier: main
Punctuation: (
Punctuation: )
Punctuation: {
Keyword: int
Identifier: x
Operator: =
Integer Constant: 100
Punctuation: ;
Keyword: float
Identifier: y
Operator: =
Float Constant: 20.5
Punctuation: ;
Keyword: if
Punctuation: (
Identifier: x
Operator: >
Identifier: y
Punctuation: )
Punctuation: {
Identifier: x
Operator: =
Identifier: x
Operator: +
Integer Constant: 10
Punctuation: ;
Punctuation: }
Punctuation: }
```

3. Implementation of Lexical Analyzer Using Lex Tool

a) Lex Specification File (lexer.l)

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

// List of keywords
char *keywords[] = {"int", "float", "if", "else", "for", "while", "return", "break"};
int isKeyword(char *str) {
    for (int i = 0; i < 8; i++) {
        if (strcmp(str, keywords[i]) == 0) return 1;
    }
    return 0;
}
%}

%option noyywrap

// Token definitions
%%
"int"|"float"|"if"|"else"|"for"|"while"|"return"|"break"     { printf("Keyword: %s\n",
yytext); }
[0-9]+\.[0-9]+      { printf("Float Constant: %s\n", yytext); }
[0-9]+              { printf("Integer Constant: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* {
    if (isKeyword(yytext))
        printf("Keyword: %s\n", yytext);
    else
        printf("Identifier: %s\n", yytext);
}
"+"|"-"|"*"|"/"|"="   { printf("Operator: %s\n", yytext); }
";"|"("|")"|"{"|"}"   { printf("Punctuation: %s\n", yytext); }
"//".*                { /* Ignore single-line comments */ }
"/\\*([^*]|\\*+[^*/])*\\*+/" { /* Ignore multi-line comments */ }
```

```
  [ \t\n]+          { /* Ignore whitespaces */ }
  .                 { printf("Unknown Token: %s\n", yytext); }
  %%

  int main(int argc, char **argv) {
      yylex();  // Start scanning
      return 0;
  }
```

B) Compile and Run

```
lex lexer.l
    gcc lex.yy.c -o lexer -ll
    ./lexer
```

C) Sample Input

```
int main() {
   int a = 10;
   float b = 20.5;
   // This is a comment
   if (a > b) {
      return a + b;
   }
}
```

D) Output

```
    Keyword: int
    Identifier: main
    Punctuation: (
    Punctuation: )
    Punctuation: {
    Keyword: int
    Identifier: a
    Operator: =
    Integer Constant: 10
    Punctuation: ;
    Keyword: float
    Identifier: b
    Operator: =
    Float Constant: 20.5
    Punctuation: ;
    Keyword: if
    Punctuation: (
    Identifier: a
    Operator: >
    Identifier: b
    Punctuation: )
    Punctuation: {
    Keyword: return
    Identifier: a
    Operator: +
    Identifier: b
    Punctuation: ;
```

Punctuation: }
Punctuation: }

4. Implementation of Lexical Analyzer Using Lex Tool

A)YACC Specification

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

int yylex();
void yyerror(const char *s);
%}

%token <num> NUMBER
%token IDENTIFIER
%token IF ELSE
%token PLUS MINUS TIMES DIVIDE
%token ASSIGN
%token LPAREN RPAREN
%token SEMICOLON
%token GT LT EQ NE

%left PLUS MINUS
%left TIMES DIVIDE
%left GT LT EQ NE

%%

program:
    statements
;

statements:
    statements statement
    |
;

statement:
```

```
    IDENTIFIER ASSIGN expression SEMICOLON { printf("Assignment: %s
= %d\n", $1, $3); }
    |
    IF expression THEN statements ELSE statements { printf("If statement
executed\n"); }
    |
    IDENTIFIER ASSIGN expression SEMICOLON { printf("Assignment: %s
= %d\n", $1, $3); }
;

expression:
    NUMBER { $$ = $1; }
    |
    IDENTIFIER { $$ = 0; }  // Simplified for demonstration
    |
    expression PLUS expression { $$ = $1 + $3; }
    |
    expression MINUS expression { $$ = $1 - $3; }
    |
    expression TIMES expression { $$ = $1 * $3; }
    |
    expression DIVIDE expression { $$ = $1 / $3; }
    |
    LPAREN expression RPAREN { $$ = $2; }
;

%%

int main() {
    printf("Enter your code:\n");
    yyparse();
    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

B)Lex Specification

```
%{
#include "y.tab.h"
%}

%option noyywrap

%%

[0-9]+              { yylval.num = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.identifier = strdup(yytext); return IDENTIFIER; }
"if"                { return IF; }
```

```
"else"            { return ELSE; }
"+"               { return PLUS; }
"-"               { return MINUS; }
"*"               { return TIMES; }
"/"               { return DIVIDE; }
"="               { return ASSIGN; }
">"               { return GT; }
"<"               { return LT; }
"=="              { return EQ; }
"!="              { return NE; }
"("               { return LPAREN; }
")"               { return RPAREN; }
";"               { return SEMICOLON; }
[ \t\n]+          { /* Ignore whitespace */ }
.                 { return yytext[0]; }

%%

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

C)Compile and Run

```
lex lexer.l
yacc -d syntax.y
gcc lex.yy.c y.tab.c -o parser -ll -ly
./parser
```

Sample Input

```
x = 10;
y = x + 5;
if y > 10 then
   z = y * 2;
else
   z = y - 3;
```

D)Sample Output

```
Assignment: x = 10
Assignment: y = 15
If statement executed
Assignment: z = 30
```

5. Convert BNF Rules into YACC Form and Generate AST

a) simple BNF for arithmetic expressions:

<expr> ::= <expr> + <term>
| <expr> - <term>
| <term>

<term> ::= <term> * <factor>
| <term> / <factor>
| <factor>

<factor> ::= (<expr>)
| NUMBER

b). YACC (Bison) File – parser.y

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

typedef enum { NODE_NUM, NODE_OP } NodeType;

typedef struct ASTNode {
   NodeType type;
   union {
      int value;  // for numbers
      struct {
         char op;
         struct ASTNode *left;
         struct ASTNode *right;
      } op;
   };
} ASTNode;

ASTNode* new_num_node(int value);
```

```
ASTNode* new_op_node(char op, ASTNode* left, ASTNode* right);
void print_ast(ASTNode* node, int indent);
void free_ast(ASTNode* node);

int yylex();
void yyerror(const char *s);
%}

%union {
   int num;
   ASTNode* node;
}

%token <num> NUMBER
%token PLUS MINUS TIMES DIVIDE LPAREN RPAREN

%type <node> expr term factor

%left PLUS MINUS
%left TIMES DIVIDE

%%

expr: expr PLUS term      { $$ = new_op_node('+', $1, $3); }
   | expr MINUS term     { $$ = new_op_node('-', $1, $3); }
   | term                { $$ = $1; }
   ;

term: term TIMES factor   { $$ = new_op_node('*', $1, $3); }
   | term DIVIDE factor  { $$ = new_op_node('/', $1, $3); }
   | factor              { $$ = $1; }
   ;

factor: LPAREN expr RPAREN { $$ = $2; }
     | NUMBER             { $$ = new_num_node($1); }
     ;

%%

void yyerror(const char *s) {
   fprintf(stderr, "Error: %s\n", s);
}

ASTNode* new_num_node(int value) {
   ASTNode* node = malloc(sizeof(ASTNode));
   node->type = NODE_NUM;
   node->value = value;
   return node;
}
```

```
ASTNode* new_op_node(char op, ASTNode* left, ASTNode* right) {
   ASTNode* node = malloc(sizeof(ASTNode));
   node->type = NODE_OP;
   node->op.op = op;
   node->op.left = left;
   node->op.right = right;
   return node;
}

void print_ast(ASTNode* node, int indent) {
   if (!node) return;
   for (int i = 0; i < indent; ++i) printf("  ");
   if (node->type == NODE_NUM) {
      printf("Number(%d)\n", node->value);
   } else {
      printf("Op('%c')\n", node->op.op);
      print_ast(node->op.left, indent + 1);
      print_ast(node->op.right, indent + 1);
   }
}

void free_ast(ASTNode* node) {
   if (!node) return;
   if (node->type == NODE_OP) {
      free_ast(node->op.left);
      free_ast(node->op.right);
   }
   free(node);
}

int main() {
   ASTNode* ast;
   if (yyparse(&ast) == 0) {
      printf("AST:\n");
      print_ast(ast, 0);
      free_ast(ast);
   }
   return 0;
}
```

c. Flex (Lexer) File – lexer.l

```
%{
#include "y.tab.h"
%}

%%

[0-9]+      { yylval.num = atoi(yytext); return NUMBER; }
"+"         { return PLUS; }
```

```
"-"         { return MINUS; }
"*"         { return TIMES; }
"/"         { return DIVIDE; }
"("         { return LPAREN; }
")"         { return RPAREN; }
[ \t\n]+    { /* ignore whitespace */ }
.           { return yytext[0]; }

%%
```

d. Build & Run Instructions

Compile:

```
bison -d parser.y       # generates parser.tab.c and parser.tab.h
flex lexer.l            # generates lex.yy.c
gcc -o calc parser.tab.c lex.yy.c –lfl
```

Run:

```
./calc
```

Then input an expression like:

```
3 + 4 * (2 - 1)
```

Output (AST):

```
    AST:
Op('+')
  Number(3)
  Op('*')
    Number(4)
    Op('-')
      Number(2)
      Number(1)
```

6. Implement Type Checking

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

typedef enum { NODE_NUM, NODE_OP } NodeType;
typedef enum { TYPE_INT, TYPE_ERROR } Type;

typedef struct ASTNode {
   NodeType type;
   Type data_type;
   union {
      int value;  // For numbers
      struct {
         char op;
         struct ASTNode *left;
         struct ASTNode *right;
      } op;
   };
} ASTNode;

ASTNode* new_num_node(int value);
ASTNode* new_op_node(char op, ASTNode* left, ASTNode* right);
void print_ast(ASTNode* node, int indent);
void free_ast(ASTNode* node);
Type check_types(ASTNode* node);

int yylex();
void yyerror(const char *s);

ASTNode* root; // Global AST root
%}

%union {
```

```
    int num;
    ASTNode* node;
}

%token <num> NUMBER
%token PLUS MINUS TIMES DIVIDE LPAREN RPAREN

%type <node> expr term factor

%left PLUS MINUS
%left TIMES DIVIDE

%%

expr: expr PLUS term       { $$ = new_op_node('+', $1, $3); }
    | expr MINUS term      { $$ = new_op_node('-', $1, $3); }
    | term                 { $$ = $1; }
    ;

term: term TIMES factor    { $$ = new_op_node('*', $1, $3); }
    | term DIVIDE factor   { $$ = new_op_node('/', $1, $3); }
    | factor               { $$ = $1; }
    ;

factor: LPAREN expr RPAREN { $$ = $2; }
      | NUMBER             { $$ = new_num_node($1); }
      ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

ASTNode* new_num_node(int value) {
    ASTNode* node = malloc(sizeof(ASTNode));
    node->type = NODE_NUM;
    node->value = value;
    node->data_type = TYPE_INT;
    return node;
}

ASTNode* new_op_node(char op, ASTNode* left, ASTNode* right) {
    ASTNode* node = malloc(sizeof(ASTNode));
    node->type = NODE_OP;
    node->op.op = op;
    node->op.left = left;
    node->op.right = right;
    node->data_type = TYPE_ERROR; // temporary until type-checked
    return node;
```

```
}

void print_ast(ASTNode* node, int indent) {
    if (!node) return;
    for (int i = 0; i < indent; i++) printf("  ");
    if (node->type == NODE_NUM) {
        printf("Number(%d)\n", node->value);
    } else {
        printf("Op('%c')\n", node->op.op);
        print_ast(node->op.left, indent + 1);
        print_ast(node->op.right, indent + 1);
    }
}

Type check_types(ASTNode* node) {
    if (!node) return TYPE_ERROR;

    if (node->type == NODE_NUM) {
        node->data_type = TYPE_INT;
        return TYPE_INT;
    }

    Type left = check_types(node->op.left);
    Type right = check_types(node->op.right);

    if (left != TYPE_INT || right != TYPE_INT) {
        fprintf(stderr, "Type error: incompatible types for '%c'\n", node->op.op);
        node->data_type = TYPE_ERROR;
        return TYPE_ERROR;
    }

    node->data_type = TYPE_INT;
    return TYPE_INT;
}

void free_ast(ASTNode* node) {
    if (!node) return;
    if (node->type == NODE_OP) {
        free_ast(node->op.left);
        free_ast(node->op.right);
    }
    free(node);
}

int main() {
    printf("Enter an expression:\n");
    if (yyparse() == 0) {
        printf("\nAST:\n");
        print_ast(root, 0);
```

```
        printf("\nType Checking:\n");
        if (check_types(root) == TYPE_INT) {
            printf("✅ Type check passed: Expression is of type INT.\n");
        } else {
            printf("❌ Type check failed.\n");
        }

        free_ast(root);
    }
    return 0;
}
```

Build & Run Instructions
Compile:
bison -d parser.y
flex lexer.l
gcc -o calc parser.tab.c lex.yy.c –lfl

Run:
./calc

Input Example:
3 + 4 * (5 - 2)

Output:
Enter an expression:

AST:
Op('+')
  Number(3)
  Op('*')
    Number(4)
    Op('-')
      Number(5)
      Number(2)

Type Checking:
✅ Type check passed: Expression is of type INT.

7. Implement Storage Allocation Strategies

```
#include <stdio.h>
#include <stdlib.h>

/// ---------- Static Allocation ----------
int static_global = 42; // allocated at compile time (static)

void static_allocation_demo() {
    static int static_local = 100; // static local variable
    printf("Static Allocation:\n");
    printf("Global: %d at %p\n", static_global, (void*)&static_global);
    printf("Static Local: %d at %p\n\n", static_local, (void*)&static_local);
}

/// ---------- Stack Allocation ----------
void stack_allocation_demo(int arg) {
    int local_var = 10; // stack
    printf("Stack Allocation:\n");
    printf("Argument: %d at %p\n", arg, (void*)&arg);
    printf("Local Variable: %d at %p\n\n", local_var, (void*)&local_var);
}

/// ---------- Heap Allocation ----------
void heap_allocation_demo() {
    int *heap_var = (int *)malloc(sizeof(int));
    if (heap_var == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    *heap_var = 99;
    printf("Heap Allocation:\n");
    printf("Heap Variable: %d at %p\n\n", *heap_var, (void*)heap_var);
```

```
    free(heap_var); // don't forget to free!
}

int main() {
    printf("----- Storage Allocation Strategies in C -----\n\n");

    static_allocation_demo();
    stack_allocation_demo(25);
    heap_allocation_demo();

    return 0;
}
```

Output:

```
----- Storage Allocation Strategies in C -----

Static Allocation:
Global: 42 at 0x5567c31df014
Static Local: 100 at 0x5567c31df018

Stack Allocation:
Argument: 25 at 0x7ffc3c1dfe9c
Local Variable: 10 at 0x7ffc3c1dfe98

Heap Allocation:
Heap Variable: 99 at 0x56352a7b12a0
```

8. Lex Program to Count Words and Lines

wordline_counter.l – Lex Program

```
%{
int word_count = 0;
int line_count = 0;
%}

%%

[ \t]+                ;  // ignore whitespace
\n                    { line_count++; }
[a-zA-Z0-9_]+         { word_count++; }
.                     ;  // ignore other characters

%%

int main() {
    yylex();  // call lexer
    printf("Lines: %d\n", line_count);
    printf("Words: %d\n", word_count);
    return 0;
}

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

Build and Run Instructions

Compile:

```
flex wordline_counter.l
gcc -o wordline_counter lex.yy.c –lfl
```

Run:
./wordline_counter

input text
Hello world
This is a test
123_abc
Hit Ctrl+D

Output
Lines: 3
Words: 7

9. Implement Lexical Analyzer in C

C Program: lexical_analyzer.c

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

#define MAX_TOKEN_LEN 100

char* keywords[] = {"int", "float", "if", "else", "while", "return", "void", NULL};

int is_keyword(const char* str) {
    for (int i = 0; keywords[i] != NULL; i++) {
        if (strcmp(str, keywords[i]) == 0) return 1;
    }
    return 0;
}

void analyze(FILE* fp) {
    char ch;
    char token[MAX_TOKEN_LEN];
    int i;

    while ((ch = fgetc(fp)) != EOF) {
        if (isspace(ch)) {
            continue;
        }
        else if (isalpha(ch) || ch == '_') {
            // Identifier or keyword
            i = 0;
            token[i++] = ch;
            while (isalnum(ch = fgetc(fp)) || ch == '_') {
                token[i++] = ch;
```

```
            }
            token[i] = '\0';
            ungetc(ch, fp);

            if (is_keyword(token))
                printf("<KEYWORD, %s>\n", token);
            else
                printf("<IDENTIFIER, %s>\n", token);
        }
        else if (isdigit(ch)) {
            // Number
            i = 0;
            token[i++] = ch;
            while (isdigit(ch = fgetc(fp))) {
                token[i++] = ch;
            }
            token[i] = '\0';
            ungetc(ch, fp);
            printf("<NUMBER, %s>\n", token);
        }
        else if (ispunct(ch)) {
            // Simple symbols and operators
            printf("<SYMBOL, %c>\n", ch);
        }
    }
}

int main() {
    FILE* fp = fopen("input.c", "r");
    if (!fp) {
        perror("Failed to open file");
        return 1;
    }

    printf("Lexical Analysis Output:\n\n");
    analyze(fp);
    fclose(fp);
    return 0;
}
```

Compile & Run

```
gcc lexical_analyzer.c -o lexer
./lexer
```

Sample input.c

```
int main() {
    int x = 42;
    float pi = 3.14;
```

```
    if (x > 0) {
        return x;
    }
}
```

Output

```
<KEYWORD, int>
<IDENTIFIER, main>
<SYMBOL, (>
<SYMBOL, )>
<SYMBOL, {>
<KEYWORD, int>
<IDENTIFIER, x>
<SYMBOL, =>
<NUMBER, 42>
<SYMBOL, ;>
<KEYWORD, float>
<IDENTIFIER, pi>
<SYMBOL, =>
<NUMBER, 3>
<SYMBOL, .>
<NUMBER, 14>
...
```

10. Recursive Descent Parser for Arithmetic Grammar

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

char *input;
char lookahead;

// Function declarations
void expr();
void expr_prime();
void term();
void term_prime();
void factor();

void error(const char *msg) {
   printf("Parse error: %s at '%c'\n", msg, lookahead);
   exit(1);
}

void match(char expected) {
   if (lookahead == expected) {
      lookahead = *++input;
   } else {
      char err[50];
      sprintf(err, "Expected '%c'", expected);
      error(err);
   }
}

void expr() {
   term();
   expr_prime();
}
```

```
void expr_prime() {
    if (lookahead == '+') {
        match('+');
        term();
        expr_prime();
    } else if (lookahead == '-') {
        match('-');
        term();
        expr_prime();
    }
    // else ε (do nothing)
}

void term() {
    factor();
    term_prime();
}

void term_prime() {
    if (lookahead == '*') {
        match('*');
        factor();
        term_prime();
    } else if (lookahead == '/') {
        match('/');
        factor();
        term_prime();
    }
    // else ε (do nothing)
}

void factor() {
    if (isdigit(lookahead)) {
        while (isdigit(lookahead)) {
            match(lookahead);  // consume digits
        }
    } else if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else {
        error("Expected number or '('");
    }
}

int main() {
    char buffer[100];
    printf("Enter an arithmetic expression: ");
    fgets(buffer, sizeof(buffer), stdin);
```

```
    input = buffer;
    lookahead = *input;

    expr();

    if (lookahead == '\n' || lookahead == '\0') {
        printf("☑ Parsing successful: valid expression.\n");
    } else {
        error("Unexpected trailing input");
    }

    return 0;
}
```

Valid Input:

Enter an arithmetic expression: 3 + 4 * (5 - 2)
Output:
☑ Parsing successful: valid expression.

Invalid Input:

Enter an arithmetic expression: 3 + * 2
Parse error: Expected number or '(' at '*'

11. Recursive Descent Parser for Custom Grammar

Custom grammar

S → A B
A → a A | ε
B → b B | ε

C Program

```
#include <stdio.h>
#include <stdlib.h>

char *input;
char lookahead;

void error(const char *msg) {
    printf("Parse error: %s at '%c'\n", msg, lookahead);
    exit(1);
}

void match(char expected) {
    if (lookahead == expected)
        lookahead = *++input;
    else {
        char err[50];
        sprintf(err, "Expected '%c'", expected);
        error(err);
    }
}

// Grammar Rules as Functions
void S();
void A();
void B();

void S() {
    A();
    B();
}

void A() {
    while (lookahead == 'a') {
```

```
        match('a');
    }
    // ε
}

void B() {
    while (lookahead == 'b') {
        match('b');
    }
    // ε
}

int main() {
    char buffer[100];
    printf("Enter a string: ");
    fgets(buffer, sizeof(buffer), stdin);
    input = buffer;
    lookahead = *input;

    S();  // Start symbol

    if (lookahead == '\n' || lookahead == '\0') {
        printf("☑ Accepted.\n");
    } else {
        error("Unexpected trailing input");
    }

    return 0;
}
```

Example Input/Output

- `aaabbb` → ✅ Accepted
- `aab` → ✅ Accepted
- `abcc` → ❌ Rejected (trailing `c`)
- `xyz` → ❌ Rejected

12. C Program to Calculate FIRST Set for Given Grammar

FIRST Set

For a non-terminal `A`, `FIRST(A)` is the set of terminals that begin the strings derivable from `A`

`Example`

Grammar:

S → AB

A → a | ε

B → b

FIRST(S) = {a, b}

FIRST(A) = {a, ε}

FIRST(B) = {b}

C Program: FIRST Set Calculator

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>
```

```
#define MAX 10
char production[MAX][MAX];
char firstSet[MAX][MAX];
int numProductions;
int isNonTerminal(char symbol) {
    return isupper(symbol);
}

void findFirst(int prodIndex, char symbol, char* result) {
    if (!isNonTerminal(symbol)) {
        strncat(result, &symbol, 1);  // Terminal
        return;
    }
    for (int i = 0; i < numProductions; i++) {
        if (production[i][0] == symbol) {
            // Rule: A → XYZ
            int j = 2;
            while (production[i][j] != '\0') {
                char temp[10] = "";
                findFirst(i, production[i][j], temp);

                int k = 0;
                int epsilonFound = 0;

                while (temp[k] != '\0') {
```

```
            if (temp[k] != 'ε') {
                if (!strchr(result, temp[k]))
                    strncat(result, &temp[k], 1);
            } else {
                epsilonFound = 1;
            }
            k++;
        }

        if (!epsilonFound)
            break;
        else
            j++;
    }

    // All symbols can derive ε
    if (production[i][j] == '\0') {
        if (!strchr(result, 'ε'))
            strcat(result, "ε");
    }
  }
}

void computeAllFirstSets() {
```

```
    for (int i = 0; i < numProductions; i++) {
        char symbol = production[i][0];
        if (firstSet[symbol - 'A'][0] == '\0') {
            findFirst(i, symbol, firstSet[symbol - 'A']);
        }
    }
}

int main() {
    printf("Enter number of productions: ");
    scanf("%d", &numProductions);

    printf("Enter productions (e.g., A->aB or A->ε):\n");
    for (int i = 0; i < numProductions; i++) {
        scanf("%s", production[i]);
    }

    computeAllFirstSets();

    printf("\nFIRST Sets:\n");
    for (int i = 0; i < numProductions; i++) {
        char symbol = production[i][0];
        if (firstSet[symbol - 'A'][0] != '\0') {
            printf("FIRST(%c) = { ", symbol);
            for (int j = 0; firstSet[symbol - 'A'][j] != '\0'; j++) {
```

```
            printf("%c ", firstSet[symbol - 'A'][j]);
        }
        printf("}\n");
    }
}

    return 0;
}
```

Example Input

Enter number of productions: 3

Enter productions:

S->AB

A->a

B->b

Output:

FIRST Sets:

FIRST(S) = { a }

FIRST(A) = { a }

FIRST(B) = { b }

13. YACC Program for Top-Down Parser

Example Grammar

E → T E'
E' → + T E' | - T E' | ε
T → F T'
T' → * F T' | / F T' | ε
F → (E) | NUM

1. parser.y – YACC File (Top-Down Style Parsing)

```
%{

#include <stdio.h>

#include <stdlib.h>

int yylex();

void yyerror(const char* s);

%}

%token NUM

%%

E   : T E1 ;

E1  : '+' T E1

    | '-' T E1
```

```
   | /* ε */ ;
T   : F T1 ;
T1  : '*' F T1
   | '/' F T1
   | /* ε */ ;
F   : '(' E ')'
   | NUM ;
%%
void yyerror(const char* s) {
   fprintf(stderr, "Error: %s\n", s);
}
int main() {
   printf("Enter an arithmetic expression:\n");
   if (yyparse() == 0)
      printf(" ✅ Valid expression (Top-Down parsed).\n");
   return 0;
}
```

2. `lexer.l` – Lex File for Tokenizing

```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUM; }
[ \t\n]     ;  // Ignore whitespace
```

[()+\-*/]   { return yytext[0]; }

.   { printf("Invalid character: %s\n", yytext); }

%%

Compile & Run

Step 1: Build

yacc -d parser.y

lex lexer.l

gcc y.tab.c lex.yy.c -o parser

Step 2: Run

./parser

Example Input:

(3 + 4) * 5 – 2

Output:

✅ Valid expression (Top-Down parsed).

14. YACC Program to Evaluate Algebraic Expressions

Step 1: expr.y — YACC File (Evaluator)

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

int yylex();
void yyerror(const char* s);
%}

%token NUM

%left '+' '-'
%left '*' '/'

%%

S : E { printf("Result = %d\n", $1); }
  ;

E : E '+' E { $$ = $1 + $3; }
  | E '-' E { $$ = $1 - $3; }
  | E '*' E { $$ = $1 * $3; }
  | E '/' E {
      if ($3 == 0) {
          yyerror("Division by zero!");
          YYABORT;
      }
      $$ = $1 / $3;
    }
  | '(' E ')' { $$ = $2; }
  | NUM { $$ = $1; }
```

```
  ;

%%

void yyerror(const char* s) {
   fprintf(stderr, "Error: %s\n", s);
}
```

Step 2: expr.l — Lex File (Tokenizer)

```
%{
#include "y.tab.h"
%}

%%

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

%%
```

Build & Run

```
yacc -d expr.y
lex expr.l
gcc y.tab.c lex.yy.c -o expr
./expr
```

Sample Input

```
3 + 4 * (2 - 1)
```

Output:

```
Result = 7
```