Compiler Design Laboratory Record

Institution: Vaagdevi College of Engineering, Warangal

Department: Computer Science Engineering

Course: Compiler Design (CD)

Table of Contents

---

1. Lex program to count keywords, numbers, and words

Objective: Write a Lex program to count the number of keywords, numbers, and words in a given input string.

Lex Program Code:

```
%{
#include <stdio.h>
int key = 0, coord = 0, num = 0, word = 0;
%}
%%
if|else { key++; printf("%s is a keyword\n", yytext); }
[0-9]+ { num++; printf("%s is a number\n", yytext); }
[a-zA-Z]+ { word++; printf("%s is a word\n", yytext); }
.|\n ;
```

```
%%
int yywrap()
{
return 1;
}
int main()
{
printf("Enter a String: ");
yylex();
printf("\nKeywords: %d\nNumbers: %d\nWords: %d\n", key, num, word);
return 0;
}
```

Output:

```
Enter a String: Vaagdevi 1 College 2 of Engineering if 3
Vaagdevi is a word
1 is a number
College is a word
2 is a number
of is a word
Engineering is a word
if is a keyword
3 is a number

Keywords: 1
Numbers: 3
Words: 5
```

---

2. Lex program to count vowels and consonants

Objective: Write a Lex program to count the number of vowels and consonants in a given input string.

Lex Program Code:

```
%{
#include <stdio.h>
int vowel = 0, consonant = 0;
%}
%%
[aeiouAEIOU] { vowel++; }
[a-zA-Z] { consonant++; }
.|\n ;
%%
int yywrap()
{
return 1;
}
int main()
{
```

```
printf("Enter a string: ");
yylex();
printf("\nNo. of vowels: %d\n", vowel);
printf("No. of consonants: %d\n", consonant);
return 0;
}
```

Output:

```
Enter a string: Vaagdevi
No. of vowels: 4
No. of consonants: 4
```

---

3. Lex program to count characters and lines

Objective: Write a Lex program to count the total number of characters and number of lines from a given file.

Lex Program Code:

```
%{
#include <stdio.h>
int n_char = 0;
int n_lines = 0;
%}
%%
. { n_char++; }
\n { n_lines++; n_char++; }
%%
int yywrap()
{
return 1;
}
int main(int argc, char *argv[])
{
yyin = fopen("abc.txt", "r");
yylex();
printf("Number of characters: %d\n", n_char);
printf("Number of lines: %d\n", n_lines);
return 0;
}
```

Output:

```
Number of characters: 36
Number of lines: 4
```

---

4. Lex program to count words and lines in a file

Objective: Write a Lex program to count the number of words and number of lines in a given file or program.

Lex Program Code:

```
%{
#include <stdio.h>
int n_words = 0;
int n_lines = 0;
%}
%%
[a-zA-Z]+ { n_words++; }
\n { n_lines++; }
. ;
%%
int yywrap()
{
return 1;
}
int main()
{
yylex();
printf("Number of words: %d\n", n_words);
printf("Number of lines: %d\n", n_lines);
return 0;
}
```

Output:

```
Vaagdevi
College
of
Engineering

Number of words: 4
Number of lines: 4
```

---

5. Implementation of Symbol Table

Objective: Implement a symbol table using linked list data structure to store variable information including name, type, and memory location.

C Program Code:

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

```
typedef struct Symbol {
char name[50];
char type[20];
int memoryLocation;
struct Symbol* next;
} Symbol;

Symbol* symbolTable = NULL;
int memoryCounter = 0;

void insertSymbol(char* name, char* type) {
Symbol* newSymbol = (Symbol*)malloc(sizeof(Symbol));
strcpy(newSymbol->name, name);
strcpy(newSymbol->type, type);
newSymbol->memoryLocation = memoryCounter++;
newSymbol->next = symbolTable;
symbolTable = newSymbol;
}

Symbol* searchSymbol(char* name) {
Symbol* current = symbolTable;
while(current != NULL) {
if(strcmp(current->name, name) == 0)
return current;
current = current->next;
}
return NULL;
}

void displaySymbolTable() {
Symbol* current = symbolTable;
printf("%-10s%-10s%-15s\n", "Name", "Type", "Memory Location");
printf("----------------------------------------\n");
while(current != NULL) {
printf("%-10s%-10s%-15d\n", current->name, current->type, current->memoryLocation);
current = current->next;
}
}

int main() {
insertSymbol("x", "int");
insertSymbol("y", "float");
insertSymbol("Sum", "function");
```

```
Symbol* s = searchSymbol("y");
if(s != NULL)
    printf("Found Symbol: %s, Type: %s, Memory Location: %d\n", s->name, s->ty
else
    printf("Symbol not found\n");
printf("\nSymbol Table:\n");
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

---

6. Lexical Analyzer for identifiers, constants, comments, and operators

Objective: Develop a lexical analyzer to recognize and classify various patterns in C code including identifiers, constants, comments, and operators.

C Program Code:

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

int iskeyword(char *str) {
char *keywords[] = {"int", "float", "if", "else", "while", "for", "return", "void"};
for(int i = 0; i < 8; i++) {
if(strcmp(str, keywords[i]) == 0)
return 1;
}
return 0;
}

int isOperator(char c) {
return (c == '+' || c == '-' || c == '*' || c == '/' || c == '=' || c == '<' || c == '>');
}

int main() {
FILE *inputFile;
char ch;
char buffer[100];
int i = 0;

```
inputFile = fopen("f1.c", "r");
if(inputFile == NULL) {
    printf("Error: Could not open file\n");
    return 1;
}

while((ch = fgetc(inputFile)) != EOF) {
    if(isalpha(ch)) {
        buffer[i++] = ch;
        while(isalnum(ch = fgetc(inputFile)) && ch != EOF) {
            buffer[i++] = ch;
        }
        buffer[i] = '\0';
        ungetc(ch, inputFile);
        i = 0;

        if(iskeyword(buffer))
            printf("Keyword: %s\n", buffer);
        else
            printf("Identifier: %s\n", buffer);
    }
    else if(isdigit(ch)) {
        buffer[i++] = ch;
        while(isdigit(ch = fgetc(inputFile)) && ch != EOF) {
            buffer[i++] = ch;
        }
        buffer[i] = '\0';
        ungetc(ch, inputFile);
        i = 0;
        printf("Constant: %s\n", buffer);
    }
    else if(isOperator(ch)) {
        printf("Operator: %c\n", ch);
    }
    else if(isspace(ch)) {
        // Skip whitespace
    }
```

```
        else {
            printf("Delimiter/Punctuation: %c\n", ch);
        }
    }

    fclose(inputFile);
    return 0;
```

}

Output:

```
Identifier: Vaagdevi
Identifier: College
Delimiter/Punctuation: .
Constant: 1234
Delimiter/Punctuation: .
Constant: 5678
Operator: =
```

---

7. Storage Allocation Strategies in C (Heap, Stack, Static)

Objective: Implement and demonstrate three storage allocation strategies in C: Static, Stack, and Heap allocation.

C Program Code:

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

int static_global = 42;

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

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

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("\nHeap Allocation:\n");
printf("Heap Variable: %d at %p\n", *heap_var, (void *)heap_var);
free(heap_var);
}

int main() {
printf("Storage Allocation Strategies in C\n");
printf("==================================\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 0x8040020
Static Local: 100 at 0x8040030

Stack Allocation:
Argument: 25 at 0xbfa07180
Local Variable: 10 at 0xbfa0716c

Heap Allocation:
Heap Variable: 99 at 0x9a51008
```

---

8. Lex program to count lines, words, spaces, and characters

Objective: Write a Lex program to count the number of lines, words, spaces, and characters from a given file.

Lex Program Code:

```
%{
#include <stdio.h>
int lc = 0, wc = 0, sc = 0, cc = 0;
%}
%%
\n { lc++; cc += yyleng; }
[ \t] { sc++; cc += yyleng; }
[^ \t\n]+ { wc++; cc += yyleng; }
%%
```

```
int yywrap()
{
return 1;
}
int main(int argc, char *argv[])
{
if(argc == 2) {
yyin = fopen(argv[1], "r");
} else {
yyin = stdin;
}
yylex();
printf("Number of Lines: %d\n", lc);
printf("Number of Spaces: %d\n", sc);
printf("Number of Words: %d\n", wc);
printf("Number of Characters: %d\n", cc);
return 0;
}
```

Output:

```
Number of Lines: 9
Number of Spaces: 0
Number of Words: 19
Number of Characters: 32
```

---

9. Recursive Descent Parser for Grammar E→E+T | E-T; T→T*F | T-F; F→(E) | id

Objective: Implement a recursive descent parser that parses arithmetic expressions according to the given grammar.

C Program Code:

```
#include <stdio.h>
#include <string.h>
#define SUCCESS 1
#define FAILED 0

int E(), Edash(), T(), Tdash(), F();
const char *cursor;
char string[64];

int main() {
puts("Enter the String:");
scanf("%s", string);
cursor = string;
puts("");
puts("Input Action");
puts("--------------------------------------------");
```

```
if(E() && *cursor == '\0') {
    puts("--------------------------------------------");
    puts("String is Successfully Parsed");
    return 0;
}
else {
    puts("--------------------------------------------");
    puts("Error in Parsing String");
    return 1;
}
```

```
}

int E() {
printf("%-16s E→TE'\n", cursor);
if(T()) {
if(Edash())
return SUCCESS;
else
return FAILED;
}
else
return FAILED;
}

int Edash() {
if(*cursor == '+') {
printf("%-16s E' → +TE'\n", cursor);
cursor++;
if(T()) {
if(Edash())
return SUCCESS;
else
return FAILED;
}
else
return FAILED;
}
else {
printf("%-16s E' → ε\n", cursor);
return SUCCESS;
}
}

int T() {
printf("%-16s T → FT'\n", cursor);
if(F()) {
```

```
if(Tdash())
return SUCCESS;
else
return FAILED;
}
else
return FAILED;
}

int Tdash() {
if(cursor == '') {
printf("%-16s T' → *FT'\n", cursor);
cursor++;
if(F()) {
if(Tdash())
return SUCCESS;
else
return FAILED;
}
else
return FAILED;
}
else {
printf("%-16s T' → ε\n", cursor);
return SUCCESS;
}
}

int F() {
if(*cursor == '(') {
printf("%-16s F → (E)\n", cursor);
cursor++;
if(E()) {
if(*cursor == ')') {
cursor++;
return SUCCESS;
}
else
return FAILED;
}
else
return FAILED;
}
else if(*cursor != '\0') {
printf("%-16s F → id\n", cursor);
cursor++;
return SUCCESS;
}
else
return FAILED;
}
```

Output:

```
Enter the String:
i+(i+i)*i
```

Input Action

i+(i+i)*i E→TE'
i+(i+i)*i T→FT'
i+(i+i)*i F→id
+(i+i)*i T'→ε
+(i+i)*i E'→+TE'
+(i+i)*i T→FT'
(i+i)*i F→(E)
(i+i)*i E→TE'
i+i)*i T→FT'
i+i)*i F→id
+i)*i T'→ε
+i)*i E'→+TE'
+i)*i T→FT'
i)*i F→id
)*i T'→ε
)*i E'→ε
)*i [Error in matching]

String is Successfully Parsed

---

10. Recursive Descent Parser for Grammar S→(L) | ε; L→L,S | S

Objective: Implement a recursive descent parser for parsing nested list structures.

C Program Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100

typedef enum {
TOK_A,
TOK_LPAREN,
```

TOK_RPAREN,
TOK_COMMA,
TOK_END,
TOK_INVALID
} TokenType;

char input[MAX_LEN];
int pos = 0;
TokenType currentToken;

void match(TokenType t);
void error();
TokenType getToken();
void S();
void L();
void Lprime();

TokenType getToken() {
while(input[pos] == ' ') pos++;
char c = input[pos];

```
switch(c) {
    case 'a': return TOK_A;
    case '(': return TOK_LPAREN;
    case ')': return TOK_RPAREN;
    case ',': return TOK_COMMA;
    case '\0': return TOK_END;
    default: return TOK_INVALID;
}
```

}

void match(TokenType t) {
if(currentToken == t) {
pos++;
currentToken = getToken();
}
else {
error();
}
}

void error() {
printf("Syntax Error at position %d\n", pos);
exit(1);
}

```
void S() {
if(currentToken == TOK_LPAREN) {
match(TOK_LPAREN);
L();
match(TOK_RPAREN);
}
}

void L() {
S();
Lprime();
}

void Lprime() {
if(currentToken == TOK_COMMA) {
match(TOK_COMMA);
S();
Lprime();
}
}

int main() {
printf("Enter input string: ");
fgets(input, MAX_LEN, stdin);
input[strcspn(input, "\n")] = '\0';
```

```
pos = 0;
currentToken = getToken();
S();

if(currentToken == TOK_END) {
    printf("Parsing Successful!\n");
}
else {
    error();
}

return 0;
```

```
}
```

Output:

```
Enter input string: (a)
Parsing Successful!
```

---

11. Implementation of Lexical Analyzer using Lex Tool

Objective: Implement a complete lexical analyzer using Lex tool to recognize and classify C language tokens.

Lex Program Code:

```
%{
int COMMENT = 0;
%}
identifier [a-zA-Z][a-zA-Z0-9]*
%%
"#".* { printf("In\t%s is a preprocessor directive\n", yytext); }
"int"|"float"|"char"|"double"|"while"|"for"|"struct"|"typedef"|"void"|"if"|"break"|"continu
e"|"switch"|"return"|"else"|"goto"
{ if(!COMMENT) printf("In\t%s is a keyword\n", yytext); }
"/*" { COMMENT = 1; }
"*/" { COMMENT = 0; printf("In\t%s is a COMMENT\n", yytext); }
{identifier}(()? { if(!COMMENT) printf("In\tFUNCTION In\t%s\n", yytext); }
"{" { if(!COMMENT) printf("In\tBLOCK BEGINS\n"); }
"}" { if(!COMMENT) printf("In\tBLOCK ENDS\n"); }
{identifier}(
                                        [0 − 9]∗
)? { if(!COMMENT) printf("In\t%s is IDENTIFIER\n", yytext); }
"." { if(!COMMENT) printf("In\t%s is STRING\n", yytext); }
[0-9]+ { if(!COMMENT) printf("In\t%s is a NUMBER\n", yytext); }
[+-/=<>] { if(!COMMENT) printf("In\t%s is OPERATOR\n", yytext); }
. { ECHO; }
\n ;
%%
int yywrap()
{
return 1;
}
int main(int argc, char **argv)
{
FILE *file;
file = fopen("var.c", "r");
if(!file) {
printf("Could not open the file\n");
exit(0);
}
yyin = file;
yylex();
printf("\n");
return 0;
}
```

Output:

```
file: Var.c
if is a keyword
float is a keyword
Vaagdevi is an identifier
College is an identifier
12 is a number
```

---

12. Abstract Syntax Tree (AST) and Quadruple generation using Lex/Yacc

Objective: Generate abstract syntax tree and quadruple representation for code statements.

Yacc Program Code:

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

struct quad {
char op[5];
char arg1[10];
char arg2[10];
char result[10];
} QUAD[30];

struct stack {
int items[100];
int top;
} stk;

int index = 0, tIndex = 0, StNO, Ind, tInd;
extern int LineNo;

void push(int data) {
stk.top++;
if(stk.top == 100) {
printf("In Stack Overflow\n");
exit(0);
}
stk.items[stk.top] = data;
}

int pop() {
int data;
if(stk.top == -1) {
printf("In Stack Underflow\n");
exit(0);
}
data = stk.items[stk.top];
stk.top--;
return data;
}

void AddQuadruple(char op[5], char arg1[10], char arg2[10], char result[10]) {
strcpy(QUAD[index].op, op);
strcpy(QUAD[index].arg1, arg1);
strcpy(QUAD[index].arg2, arg2);
sprintf(QUAD[index].result, "t%d", index++);
strcpy(result, QUAD[index - 1].result);
}

void yyerror() {
printf("In Error on Line no: %d\n", LineNo);
}
%}

%union {
char var[10];
}

%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST
%left '-' '+'
%left '*' '/'
%%

PROGRAM : MAIN '{' CODE '}' ;
CODE : CODE STATEMENT | STATEMENT ;
STATEMENT : ASSIGNMENT ';' | IFST | WHILEST ;
ASSIGNMENT : VAR '=' EXPR {
strcpy(QUAD[index].op, "=");
strcpy(QUAD[index].arg1, $3);
strcpy(QUAD[index].arg2, "");
strcpy(QUAD[index].result, $1);
```

```
strcpy($$, QUAD[index++].result);
```

```
};

EXPR : EXPR '+' EXPR { AddQuadruple("+", $1, $3, $$); }
```

```
| EXPR '-' EXPR { AddQuadruple("-", $1, $3, $$); }
| EXPR '*' EXPR { AddQuadruple("*", $1, $3, $$); }
| EXPR '/' EXPR { AddQuadruple("/", $1, $3, $$); }
| VAR { strcpy($$, $1); }
| NUM { strcpy($$, $1); };
```

CONDITION : VAR RELOP VAR { AddQuadruple($2, $1, $3, $$); };

%%

---

13. Type Checking Implementation

Objective: Implement type checking to validate variable declarations and expressions.

C Program Code:

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

int main() {
int n, i, flag = 0;
char vari[15], typ[15], b[15], c;

```
printf("Enter the number of variables: ");
scanf("%d", &n);

for(i = 0; i < n; i++) {
    printf("Enter the variable [%d]: ", i);
    scanf("%s", &vari[i]);

    printf("Enter the variable-type [%d] (float - f, int - i): ", i);
    scanf("%s", &typ[i]);

    if(typ[i] == 'f')
        flag = 1;
}

printf("Enter the Expression (end with $): ");
i = 0;
getchar();

while((c = getchar()) != '$') {
    b[i] = c;
    i++;
}
```

```
int k = i;
for(i = 0; i < k; i++) {
    if(b[i] == '/') {
        flag = 1;
        break;
    }
}

for(i = 0; i < n; i++) {
    if(b[0] == vari[i]) {
        if(typ[i] == 'f') {
            printf("In the datatype is correctly defined....!\n");
            break;
        }
        else {
            printf("In Identifier %c must be a float type...!\n", vari[i]);
            break;
        }
    }
}

return 0;
```

}

Output:

```
Enter the number of Variables: 4
Enter the variable [0]: A
Enter the variable-type [0] (float - f, int - i): i
Enter the variable [1]: B
Enter the variable-type [1] (float - f, int - i): i
Enter the variable [2]: C
Enter the variable-type [2] (float - f, int - i): f
Enter the variable [3]: D
Enter the variable-type [3] (float - f, int - i): i
Enter the Expression (end with $): A=B*C/D $
Identifier A must be a float type...!
```

14. Lexical Analyzer in C

Objective: Implement a lexical analyzer in C to tokenize source code.

C Program Code:

```
#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 == '_') {
        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)) {
            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)) {
            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;
```

}

Output:

```
Lexical Analysis 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>
<SYMBOL, ;>
<KEYWORD, if>
<SYMBOL, (>
<IDENTIFIER, x>
<SYMBOL, >>
<NUMBER, 0>
<SYMBOL, )>
<SYMBOL, {>
<KEYWORD, return>
<IDENTIFIER, x>
<SYMBOL, ;>
<SYMBOL, }>
<SYMBOL, }>
```

---

15. Yacc Specification for Syntactic Categories (Arithmetic Expression)

Objective: Write a Yacc program to validate and evaluate arithmetic expressions.

Yacc Program Code:

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

%token num let
%left '+' '-'
%left '*' '/'

%%
Stmt: Stmt '\n'
{
printf("In... Valid Expression\n");
exit(0);
}
| expr
{
printf("In... Invalid...\n");
exit(0);
}
;
```

```
expr: num
| let
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')'
;

%%

int main() {
printf("Enter an Expression: ");
yyparse();
return 0;
}

int yylex() {
int c;
while((c = getchar()) == ' ' || c == '\t');
if(isdigit(c))
return num;
if(isalpha(c))
return let;
return c;
}

void yyerror(char *s) {
printf("%s\n", s);
}
```

Output:

```
Enter an Expression: a+b
In... Valid Expression
```

---

16. Identifier Validation using Lex/Yacc

Objective: Write a program using Lex and Yacc to validate identifiers.

Lex Program Code:

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

%%
[a-zA-Z][a-zA-Z_0-9]* return letter;
[0-9] return digit;
. return yytext[0];
\n return 0;
%%
```

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

Yacc Program Code:

```
%{
#include <stdio.h>
int valid = 1;
%}

%token digit letter

%%
Start: letter S
;

S: letter S
| digit S
| ;

%%

int yyerror() {
printf("It's not an Identifier!\n");
valid = 0;
return 0;
}

int main() {
printf("Enter a name to test for Identifier: ");
yyparse();
if(valid) {
printf("It is a valid Identifier!\n");
}
return 0;
}
```

Output:

```
Enter a name to test for Identifier: abc
It is a valid Identifier!
```

---

17. Calculator Implementation using Lex and Yacc

Objective: Implement a calculator using Lex and Yacc to evaluate arithmetic expressions.

Yacc Program Code:

```
%{
#include <stdio.h>
int flag = 0;
%}

%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

%%

Arithmetic_Expression: E '\n'
{
printf("\nResult = %d\n", $1);
return 0;
}
;

E: E '+' E { $$ = $1 + $3; }

| E '-' E { $$ = $1 - $3; }

| E '*' E { $$ = $1 * $3; }

| E '/' E { $$ = $1 / $3; }

| E '%' E { $$ = $1 % $3; }

| '(' E ')' { $$ = $2; }

| NUMBER { $$ = $1; }
;

%%

void main() {
printf("In Enter Any Arithmetic Expression which can have Operations\n");
printf("Addition, Subtraction, Multiplication, Division, Modulus,\n");
printf("and Round Brackets:\n");
yyparse();
if(flag == 0) {
printf("In Entered arithmetic expression is valid\n");
}
}

void yyerror() {
printf("In Entered arithmetic expression is Invalid\n");
flag = 1;
}
```

Lex Program Code:

```
%{
#include <stdio.h>
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
[\t] ;
[\n] return 0;
. return yytext[0];
%%

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

Output:

```
Enter Any Arithmetic Expression which can have Operations
Addition, Subtraction, Multiplication, Division, Modulus,
and Round Brackets:
5+6

Result = 11
Entered Arithmetic Expression is valid
```

---

18. Top-down Parser Implementation using Yacc

Objective: Implement a top-down parser using Yacc for parsing arithmetic expressions.

Yacc Program Code:

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

%token ID NUMBER
%left '+' '-'
%left '*' '/' '%'

%%

program: expression '\n' { printf("Result: %d\n", $1); }
;

expression: term
```

```
      | expression '+' term { $$ = $1 + $3; }
      | expression '-' term { $$ = $1 - $3; }
      ;
```

term: factor

```
  | term '*' factor { $$ = $1 * $3; }
  | term '/' factor { $$ = $1 / $3; }
  ;
```

factor: NUMBER
| ID

```
   | '(' expression ')' { $$ = $2; }
   ;
```

%%

int yylex() {
int c;
while((c = getchar()) == ' ' || c == '\t');
if(isdigit(c)) {
yylval.number = c - '0';
while(isdigit(c = getchar())) {
yylval.number = (yylval.number * 10 + (c - '0'));
}
ungetc(c, stdin);
return NUMBER;
}
else if(isalpha(c)) {
return ID;
}
else if(c == '\n') {
return '\n';
}
else if(c != EOF) {
return c;
}
return 0;
}

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

```
int main() {
printf("Enter an Expression: ");
return yyparse();
}
```

Output:

```
Enter an Expression: (5+3)*2
Result: 16
```

---

19. Algebraic Expression Evaluation

Objective: Write a program to evaluate algebraic expressions of the form (a*x+b)/(a*x-b).

C Program Code:

```
#include <stdio.h>

int main() {
int a, b, x;
float result;
```

```
printf("Enter the values of a, b, and x: ");
scanf("%d %d %d", &a, &b, &x);

if((a * x - b) == 0) {
    printf("Division by zero not allowed\n");
    return 1;
}

result = (float)(a * b) / (a * x - b);
printf("The result of the expression (a*x+b)/(a*x-b) is: %.2f\n", result);

return 0;
```

```
}
```

Output:

```
Enter the values of a, b and x: 5 4 6
The result of the expression (ax+b)/(ax-b) is: 1.85
```

---

20. FIRST Function Calculation for Grammar

Objective: Calculate the FIRST set for non-terminals in a given grammar.

Grammar:

- E → E+T | E-T | T
- T → T*F | T-F | F
- F → (E) | id

C Program Code:

#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 10

char productions[10][10];
char firstSet[10][MAX];
int n;

int isTerminal(char c) {
return !isupper(c);
}

void findFirst(char symbol, char *result) {
if(isTerminal(symbol)) {
strncat(result, &symbol, 1);
return;
}

```
for(int i = 0; i < n; i++) {
    if(productions[i][0] == symbol) {
        char next = productions[i][2];
        if(isTerminal(next)) {
            if(!strchr(result, next)) {
                strncat(result, &next, 1);
            }
        }
        else {
            char temp[MAX] = "";
            findFirst(next, temp);
            for(int j = 0; j < strlen(temp); j++) {
                if(!strchr(result, temp[j])) {
                    strncat(result, &temp[j], 1);
                }
            }
        }
    }
}
```

}

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

```
printf("Enter productions (e.g., E=E+T or E-T):\n");
for(int i = 0; i < n; i++) {
    scanf("%s", productions[i]);
}

printf("\nFIRST Sets:\n");
for(int i = 0; i < n; i++) {
    char nt = productions[i][0];
    if(firstSet[nt - 'A'][0] == '\0') {
        findFirst(nt, firstSet[nt - 'A']);
    }
    if(firstSet[nt - 'A'][0] != '\0') {
        printf("FIRST (%c) = { ", nt);
        for(int j = 0; j < strlen(firstSet[nt - 'A']); j++) {
            printf("%c", firstSet[nt - 'A'][j]);
        }
        printf(" }\n");
    }
}

return 0;
```

}

Output:

```
Enter number of productions: 5
Enter productions (e.g., E=E+T or E-T):
E=E+T
E=T
T=T*F
T=F
F=(E)

FIRST Sets:
FIRST (E) = { i ( }
FIRST (T) = { i ( }
FIRST (F) = { i ( }
```

Summary

This laboratory record contains comprehensive implementations of various compiler design concepts including:

- Lexical analysis using Lex and custom C programs
- Symbol tables and data structures
- Storage allocation strategies
- Recursive descent parsing
- Yacc specifications for syntax analysis
- Type checking implementations
- Abstract Syntax Tree generation
- Quadruple generation for intermediate code
- FIRST function calculation for grammar analysis
- Calculator implementation using Lex/Yacc

All programs include complete source code, execution procedures, and expected outputs for reference and verification.

Compiled by: Computer Science Engineering Department
Institution: Vaagdevi College of Engineering, Warangal
Date: November 2025