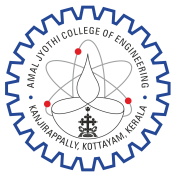
AMAL JYOTHI COLLEGE OF ENGINEERING

(AUTONOMOUS)

KANJIRAPALLY



AMAL JYOTHI
COLLEGE OF ENGINEERING
AUTONOMOUS
KANJIRAPPALLY

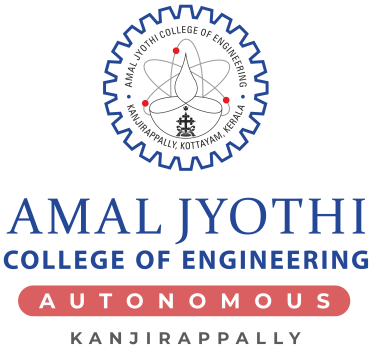
CSL411 – COMPILER LAB RECORD

Department of Computer Science & Engineering

October-2025

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AMAL JYOTHI COLLEGE OF ENGINEERING (AUTONOMOUS), KANJIRAPALLY



Name: .....

Semester: ..... **Branch:** .....

Batch: ..... **Roll No:** .....

University Reg. No: .....

Certified that this is a bonafide record of practical work done in **CSL411 – COMPILER LAB**

Laboratory by .....

during the year **2025–2026.**

Head of the Department

Faculty in Charge

Internal Examiner

External Examiner

Cut here

Cut here

DEPARTMENT OF
COMPUTER SCIENCE
& ENGINEERING

NBA NATIONAL BOARD OF ACCREDITATION

Accredited by NAAC NATIONAL ASSESSMENT AND ACCREDITATION COUNCIL

- Approved by: AICTE
- Affiliated to: APJ Abdul Kalam Technological University, Kerala

Vision of the Department

The Computer Science and Engineering department is committed to continually improve the educational environment in order to develop professionals with strong technical and research backgrounds.

Mission of the Department

To provide quality education in both theoretical and applied foundations of Computer Science & Engineering.
Create highly skilled Computer Engineers, capable of doing research and also develop solutions for the betterment of the nation.
Inculcate professional and ethical values among students.
Support society by participating in and encouraging technology transfer.

Programme Educational Objectives

PE01 Be successfully employed in computing profession as well as multidisciplinary domains in supportive and leadership roles.
PE02 Participate in life-long learning through successful completion of advanced degrees, continuing education, certifications and/or other professional development.
PE03 Promote design, research, product implementation and services in the field of Computer Science and Engineering through strong technical, communication and entrepreneurial skills.

Programme Outcomes

B Tech Computer Science and Engineering Graduates will be able to:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. Design/development of solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Programme Specific Outcomes

PS01 Apply Engineering knowledge to analyze, design and develop computing solutions by employing modern computer languages, environments and platforms that can solve complex problems.
PS02 Anticipate the changing direction of computational technology, evaluate it and communicate the likely utility of that for building software systems that would perform tasks related to industry, research and education.
PS03 Inculcate the knowledge of Engineering and Management principles to manage projects effectively and create innovative career paths.

Cut here

Cut here

CONTENTS

Date: 18-07-25

EXPERIMENT - 1

IMPLEMENTATION OF LEXICAL ANALYZER USING C

AIM

To design and implement a lexical analyzer using C language.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start.

Step 2: File Handling

Step 2:1. Open the input file named `"input.txt"` in read mode.

Step 2:2. If the file cannot be opened, display the message `"File not found!"` and terminate the program.

Step 3: Initialize necessary variables:

- A character `c`
- A string `str` for identifiers/keywords

- An integer `num` for numbers
- A line counter `lineno`
- A string index `i`
- A list of C language keywords

Step 4: While the end of the file has not been reached, repeat the following steps:

Step 4:1. Read one character from the file and store it in `c`.

Step 4:2. If `c` is a digit:

- Read the complete number.
- Print it.
- Push back the non-digit character.

Step 4:3. If `c` is a letter or underscore:

- Read the full alphanumeric string.
- Check if it is a keyword.
- Print it as a keyword or identifier accordingly.

Step 4:4. If `c` is an operator (`+, -, *, %, /, &, |, =, <, >`):

- Print it as an operator.

Step 4:5. If `c` is a space or tab:

- Skip it.

Step 4:6. If `c` is a newline character:

- Increment the line counter.

Step 4:7. Otherwise:

- Print c as a special symbol.

Step 5: After all characters have been processed:

- Close the input file.

Step 6: Stop.

PROGRAM

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
int i,j;
void main(){
FILE *f1;
char c, str[20];
int lineno=0, num=0, i=0;
char *keywords[] = {"auto", "break", "case", "char", "const", "continue",
                    "default", "do", "double", "else", "enum", "extern",
                    "float", "for", "goto", "if", "int", "long", "register",
                    "return", "short", "signed", "sizeof", "static", "struct",
                    "switch", "typedef", "union", "unsigned", "void", "volatile",
                    "while"};

f1 = fopen("input.txt", "r");
if(f1 == NULL){
```

```
    printf("File not found!\n");
    return;
}
while((c=getc(f1)) != EOF){
    if(isdigit(c))
    {
        num=c-48;
        c=getc(f1);
        while(isdigit(c)){
        num=num*10+(c-48);
        c=getc(f1);
        }
        printf("%d is a number \n", num);
        ungetc(c,f1);
    }
    else if(isalpha(c) || c == '_'){
        str[i++] = c;
        c=getc(f1);
        while(isalnum(c) || c == '_'){
            str[i++] = c;
            c=getc(f1);
        }
        str[i++] = '\0';
        int isKeyword = 0;
        for( j=0; j<sizeof(keywords)/sizeof(keywords[0]); j++)
```

```
        {
            if(strcmp(keywords[j], str) == 0){
                printf("%s is a keyword \n", str);
                isKeyword = 1;
                break;
            }
        }
        if(!isKeyword){
            printf("%s is an identifier \n", str);
        }
        ungetc(c,f1);
        i = 0;
    }else if(c == '+' || c == '-' || c == '*' || c == '%' || c == '/' ||
             c == '&' || c == '|' || c == '=' || c == '<' || c == '>'){
        printf("%c is an operator \n", c);
    }else if(c == ' ' || c == '\t'){
        lineno++;
    }else if(c == '\n'){
        lineno++;
    }else{
        printf("%c is a special symbol \n", c);
    }
}
fclose(f1);
}
```

INPUT: text.txt

```
#include<stdio.h>
void main(){
 int x = 4;
 float y = 30
  if(x > y){
  printf("Sum = %d",x+y)
 }
}
```

OUTPUT

```
# is a special symbol
include is an identifier
< is an operator
stdio is an identifier
. is a special symbol
h is an identifier
> is an operator
void is a keyword
main is an identifier
( is a special symbol
) is a special symbol
```

```
{ is a special symbol
int is a keyword
x is an identifier
= is an operator
4 is a number
, is a special symbol
y is an identifier
= is an operator
10 is a number
; is a special symbol
if is a keyword
( is a special symbol
x is an identifier
> is an operator
y is an identifier
) is a special symbol
{ is a special symbol
printf is an identifier
( is a special symbol
" is a special symbol
Sum is an identifier
= is an operator
% is an operator
d is an identifier
" is a special symbol
```

```
, is a special symbol
x is an identifier
+ is an operator
y is an identifier
) is a special symbol
} is a special symbol
} is a special symbol
Process returned 0 (0x0)   execution time : 0.149 s
Press any key to continue.
```

RESULT

The program to implement Lexical analyzer using C has been successfully executed and output verified thus CO1 has been achieved.

Date: 23-07-25

EXPERIMENT - 2

LEXICAL ANALYZER USING LEX TOOL

AIM

To design and implement a lexical analyzer using Lex Tool

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start

Step 2: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

Step 3: Lex programs follow this basic structure:definitions %% rules %% user_subroutines

Step 4: Include necessary headers in the definitions part.

Step 5: Define patterns using regular expressions and associate them with actions in the Rules part:

Step 5:.1 For the input matching `[0-9]+`, print “it is a number”.

Step 5:.2 For the input matching `"if"|"else"|"while"|"int"`, print "it is a keyword".

Step 5:.3 For the input matching `[_a-zA-Z][a-zA-Z0-9]*`, print "it is an identifier".

Step 5:.4 For the input matching `">"|"<"|"="|">="|"<="|"+"|"-"|"*"|"/"`, print "it is an operator".

Step 5:.5 For the input matching `"'"|":"|";"|","|"("|")"`, print "it is a special character".

Step 6: In the last section, include the `main()` function which calls `yylex()`:

Step 6:.1 `yylex()` scans the input.

Step 6:.2 Matches input against the regular expressions defined.

Step 6:.3 Executes corresponding action when a match is found.

Step 6:.4 Stores the matched string in `yytext`.

Step 7: Stop

PROGRAM

```
DIGIT [0-9]
ID [_a-zA-Z][a-zA-Z0-9]*
KEYWORD "if"|"else"|"while"|"int"
OP ">"|"<"|"="|">="|"<="|"+"|"-"|"*"|"/"
SPECIALCHARACTER "'"|":"|";"|"("|")"
%%
```

```
{KEYWORD}+ {printf("%s is a keyword\n",yytext);}
{DIGIT}+ {printf("%s is a integer\n",yytext);}
[_a-zA-Z][a-zA-Z0-9]*+ {printf("%s is a an identifier\n",yytext);}
{OP}+ {printf("%s is an operater\n",yytext);}
{SPECIALCHARACTER}+ {printf("%s is A SPECIAL CHARACTER\n",yytext);}
%%

void main()
{
printf("Enter the string\n");
yylex();
}

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

OUTPUT

```
user-HP-Pavilion-Notebook% lex lex.l
user-HP-Pavilion-Notebook% gcc lex.yy.c -o MR
user-HP-Pavilion-Notebook% ./MR
Enter the string:
if(a10=0)
if is a keyword
( is a special character
a10 is an identifier
= is an operator
0 is a digit
) is a special character
```

RESULT

Program for implementation of Lexical Analyzer using Lex tool has been executed successfully and CO1 has achieved.

Date: 23-07-25

EXPERIMENT - 3.1

CHECK IF A NUMBER IS ODD OR EVEN USING LEX TOOL

AIM

To write a lex program to check if a given number is odd or even using LEX tool.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start

Step 2: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

Step 3: Lex programs follow this basic structure:definitions %% rules %%user_subroutines

Step 4: Include necessary headers in the definitions part.

Step 5: Define patterns using regular expressions and associate them with actions in the Rules part:

Step 5:.1 For the regular expression [0-9]*[02468] print it is an even number.

Step 5:.2 For the regular expression [0-9]*[1357] print it is an odd number

Step 6: In the last section, include the `main()` function which calls `yylex()`:

Step 6:.1 `yylex()` scans the input.

Step 6:.2 Matches input against the regular expressions defined.

Step 6:.3 Executes corresponding action when a match is found.

Step 6:.4 Stores the matched string in `yytext`.

Step 7: Stop

PROGRAM

```
%%

["a"|"e"|"i"]     { printf("Even number: %s\n", yytext); }
[0-9]*[13579]     { printf("Odd number: %s\n", yytext); }

%%

int main() {
    yylex();

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

```
}
```

OUTPUT

```
student@P53:~/cs29$ lex lex2.l
student@P53:~/cs29$ gcc lex.yy.c
student@P53:~/cs29$ ./a.out
3457
Odd number: 3457

890
Even number: 890
```

RESULT

The program to check if a number is even or odd has been executed successfully and CO1 has achieved.

Date: 23-07-25

EXPERIMENT - 3.2

NUMBER OF VOWELS AND CONSONANTS

AIM

To write a lex program to find out the total number of vowels and consonants from the given input string using LEX tool.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start

Step 2: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

Step 3: Lex programs follow this basic structure:definitions %% rules %%user_subroutines

Step 4: Include necessary headers in the definitions part.

Step 5: Define patterns using regular expressions and associate them with actions in the Rules part:

Step 5:.1 For the input matching [aeiouAEIOU] increment vowel count by 1

Step 5:.2 For all others increment the consonant count by 1

Step 6: In the last section, include the `main()` function which calls `yylex()`:

Step 6:.1 `yylex()` scans the input.

Step 6:.2 Matches input against the regular expressions defined.

Step 6:.3 Executes corresponding action when a match is found.

Step 6:.4 Stores the matched string in `yytext`.

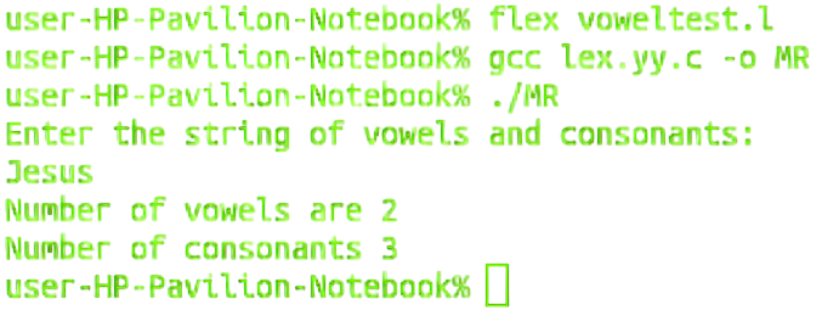
Step 7: Stop

PROGRAM

```
%{
int vow_count=0;
int const_count=0;
%}
vowel [aeiouAEIOU]
consonant [a-zA-Z]
%%
{vowel} {vow_count++;}
[a-zA-Z] {const_count++;}
%%

int main(){
printf("Enter string of vowels and consonant:\n");
```

```
yylex();
printf("No of vowels are: %d\n", vow_count);
printf("No of consonants are: %d\n", const_count);
return 0;
}

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

OUTPUT



RESULT

The program to find out total number of vowels and consonants has been executed successfully and CO1 has achieved.

Date: 23-07-25

EXPERIMENT - 3.3

EMAIL VALIDATION USING LEX

AIM

To write a lex program to check whether an email is valid or not.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start

Step 2: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

Step 3: Lex programs follow this basic structure:definitions %% rules %%user_subroutines

Step 4: Include necessary headers in the definitions part.

Step 5: Define patterns using regular expressions and associate them with actions in the Rules part:

Step 5:.1 For the regular expression [a-z0-9]+@[a-z]+.[a-z]+ ,print its an email

Step 6: In the last section, include the `main()` function which calls `yylex()`:

Step 6:.1 `yylex()` scans the input.

Step 6:.2 Matches input against the regular expressions defined.

Step 6:.3 Executes corresponding action when a match is found.

Step 6:.4 Stores the matched string in `yytext`.

Step 7: Stop

PROGRAM

```
%%
[a-z0-9]+@[a-z]+.[a-z]+   { printf("Email found: %s\n", yytext); }

%%

int main() {
    printf("Enter text with emails:\n");
    yylex();
    return 0;
}

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

OUTPUT

```
student@P53:~/cs29$ lex lex4.l
student@P53:~/cs29$ gcc lex.yy.c
student@P53:~/cs29$ ./a.out
Enter text with emails:
hiiiyess@gmail.com
Email found: hiiiyess@gmail.com
```

RESULT

The program to find whether the input given is an email has been executed successfully and CO1 has achieved.

Date: 30-07-25

EXPERIMENT - 3.4

COUNT NUMBER OF PRINTF AND SCANF USING LEX

AIM

To write a lex program that identifies printf and scanf statements in a c file and replaces them with writef and readf respectively using the LEX tool.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start.

Step 2: A `lex` program contains three sections:

Step 3: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

Step 4: The `lex` program follows this basic structure: `definitions %% rules %% user_subroutines`

Step 5: In the Definitions Section:

Step 5:1. Include the header file `<stdio.h>`.

Step 5:2. Declare two global counters: `printf_count` and `scanf_count`, initialized to 0.

Step 6: In the Rules Section:

Step 6:1. If the input matches `"printf"`:

- Print `"writef"`.
- Increment `printf_count`.

Step 6:2. If the input matches `"scanf"`:

- Print `"readf"`.
- Increment `scanf_count`.

Step 6:3. For any other character (including newline):

- Echo the character using `ECHO`.

Step 7: In the User Subroutines Section:

Step 7:1. Define `main()` function that calls `yylex()`.

Step 7:2. Open the file `input.c` for reading.

Step 7:3. Set the lex input source to the file using `yyin`.

Step 7:4. Call `yylex()` to scan the input.

Step 7:5. After scanning, close the file.

Step 7:6. Print the total number of replacements for `printf` and `scanf`.

Step 8: Define `yywrap()` function that returns `1`.

Step 9: **Stop.**

PROGRAM

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

int printf_count = 0;
int scanf_count = 0;
%}

%%
"printf"    { printf("writef"); printf_count++; }
"scanf"     { printf("readf");  scanf_count++; }
.|\n        { ECHO; }
%%

int main(int argc, char **argv)
{
    FILE *fp = fopen("input.c", "r");

    if (!fp) {
        perror("Error opening file");
        return 1;
    }

    yyin = fp;      // Set Lex input source
```

```
    yylex();        // Start scanning

    fclose(fp);

    printf("\nTotal printf statements replaced: %d\n", printf_count);
    printf("Total scanf statements replaced: %d\n", scanf_count);
    return 0;
}

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

INPUT: input.c

```
#include<stdio.h>
int main(){
int a,b;
printf("Enter two numbers:");
scanf("%d",&a);
scanf("%d",&b);
printf("The sum is %d",a+b);
}
```

OUTPUT

```
#include<stdio.h>
int main(){
int a,b;
writef("Enter two numbers:");
readf("%d",&a);
readf("%d,&b);
writef("The sum is %d",a+b);
}

Total scanf: 2
Total printf: 2
user-HP-Pavilion-Notebook%
```

RESULT

The program that identifies printf and scanf statements in a c file and replaces them with writef and readf respectively using the LEX tool has been executed successfully and CO1 has achieved.

Date: 30-07-25

EXPERIMENT - 3.5

VALIDATE IF STATEMENT USING LEX

AIM

To write a `lex` program that verifies whether a given input contains a *valid `if` statement* with *proper condition* and *block structure* using the `lex` tool.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start.

Step 2: Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

Step 3: The `lex` program follows the structure: definitions %% rules %% user-subroutines

Step 4: In the **definitions** section:

Step 4:1. Include header files `<stdio.h>` and `<string.h>`.

Step 4:2. Declare and initialize three integer flags:

- `found_if = 0` → To check if `if` keyword is found.
- `found_condition = 0` → To check if a condition inside parentheses `( )` is found.
- `found_block = 0` → To check if a block inside curly braces { } is found.

Step 5: In the **rules** section:

Step 5:1. If the input matches `"if"`, set `found_if = 1`.

Step 5:2. If the input matches the condition pattern `"(...)"` and `found_if` is already set, set `found_condition = 1`.

Step 5:3. If input matches block pattern {"[]"}, *set* `found_block = 1`.

`Step 5:3. For any other characters (including newline), ignore them.`

Step 6: In the **user subroutines** section:

Step 6:1. Define `main()` function.

Step 6:2. Call `yylex()` to start scanning input.

Step 6:3. After scanning:

- If `found_if`, `found_condition`, and `found_block` are all set, print `"Valid if statement"`.
- Otherwise, print `"Invalid if statement"`.

Step 6:4. Return success.

Step 7: Define `yywrap()` function to return `1`.

Step 8: Stop.

PROGRAM

```
%%

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

int found_if = 0;
int found_condition = 0;
int found_block = 0;

%%
"if" { found_if = 1; }
"(" [^()]* ")" { if (found_if) found_condition = 1; }
"{" [^{}]* "}" { if (found_if && found_condition) found_block = 1; }
.|\n /* Ignore other characters */ {}
%%

int yywrap() {
    return 1;
}

int main() {
    yylex();
```

```
    if (found_if && found_condition && found_block) {
        printf("Valid if statement\n");
    } else {
        printf("Invalid if statement\n");
    }

    return 0;
}
```

OUTPUT

```
student@P37 ~/Downloads$ lex prgm6.l
student@P37 ~/Downloads$ gcc lex.yy.c
student@P37 ~/Downloads$ ./a.out
Enter an if statement:
if(a>b){
        printf("a is the greatest");
}else{
        printf("b is the greatest");
}
Valid if statement.
student@P37 ~/Downloads$
```

RESULT

The program for validating the structure of an if statement using LEX tool has been executed successfully and CO1 has achieved.

Date: 12-09-25

EXPERIMENT - 4.1

RECOGNIZE VALID IDENTIFIER USING YACC

AIM

Generate a YACC specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start the program.

Step 2: Display a prompt to the user to enter an identifier.

Step 3: Read the input from the user.

Step 4: The lexical analyzer (Lex) scans the input character by character.

Step 5: Classify the input using the following rules:

Step 5.1: If the input starts with a letter (a–z, A–Z) or underscore (_), and is followed by zero or more letters, digits, or underscores, it is recognized as a valid `IDENTIFIER` token.

Step 5.2: Otherwise, treat the input as invalid and return it as a single character token.

Step 6: Pass the generated token(s) to the parser (Yacc).

Step 7: The parser checks the structure of the input:

Step 7.1: If the input is a valid `IDENTIFIER`, print "Valid identifier".

Step 7.2: If the input is empty or not a valid identifier, print "Invalid identifier".

Step 8: If a syntax error is detected, call the `yyerror()` function and display an appropriate error message.

Step 9: Stop.

PROGRAM

Iden.l

```
%{
#include "y.tab.h"
#include <string.h>
%}
%%
[a-zA-Z_][a-zA-Z0-9_]* { return IDENTIFIER; }
. { return yytext[0]; }
%%
int yywrap() {
```

```
    return 1;
}
```

Iden.y

```
%{
#include <stdio.h>
void yyerror(const char *s);
int yylex(void);
%}
%token IDENTIFIER
%%
input:
IDENTIFIER { printf("Valid identifier\n"); }
| /* empty */ { printf("Invalid identifier\n"); }
;
%%
void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
int main() {
    printf("Enter an identifier: ");
    yyparse();
    return 0;
}
```

OUTPUT

```
user-HP-Pavilion-Notebook% gedit calc.l
user-HP-Pavilion-Notebook% gedit calc.y
user-HP-Pavilion-Notebook% flex calc.l
user-HP-Pavilion-Notebook% bison -dy calc.y
user-HP-Pavilion-Notebook% gcc lex.yy.c y.tab.c -o MR
user-HP-Pavilion-Notebook% ./MR
Enter an expression (e.g., (5 + 3) * 2):
4+3
Operand: 4
Operand: 3
Result: 7
user-HP-Pavilion-Notebook%
```

RESULT

Program to recognize a valid identifier which starts with a letter followed by any number of letters or digits using Yacc has been executed successfully and CO1 has achieved.

Date: 12-09-25

EXPERIMENT - 4.2

RECOGNIZE VALID ARITHMETIC EXPRESSION USING YACC

AIM

To generate a YACC specification to recognize a valid arithmetic expression that uses operators +, , *, / and parenthesis.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start the program.

Step 2: Read the input arithmetic expression from the user.

Step 3: The lexical analyzer scans the input character by character.

Step 4: Identify tokens such as numbers (`DIGIT`), identifiers (`LETTER`), operators (`+`, `-`, `*`, `/`), and parentheses (`(`, `)`).

Step 5: Ignore any whitespace or invalid characters.

Step 6: Send the sequence of tokens to the parser.

Step 7: The parser checks the tokens using predefined grammar rules (`expr`) and precedence declarations (`%left`, `%right`) for valid expressions.

Step 8: If the tokens follow the correct grammatical structure, the parser accepts the input.

Step 9: If any mismatch occurs, the `yyerror()` function is called and a syntax error is reported.

Step 10: Display "Valid Arithmetic Expression" if accepted; otherwise, display "Invalid Arithmetic Expression."

Step 11: Stop the program.

PROGRAM

`arithmetic.l`

```
%{
#include "y.tab.h"
#include <stdlib.h>
#include <string.h>
%}
%%
[0-9]+ { yylval = atoi(yytext); return DIGIT; }
[_a-zA-Z][_a-zA-Z0-9]* { return LETTER; }
"+" { return '+'; }
"-" { return '-'; }
```

```
"*" { return '*'; }
"/" { return '/'; }
"(" { return '('; }
")" { return ')'; }
[\t ]+ ; /* ignore whitespace */
\n { return 0; }
. { return yytext[0]; }
%%
int yywrap() {
    return 1;
}
```

Arithmetic.y

```
%{
#include <stdio.h>
#include <stdlib.h>
void yyerror(char *s);
int yylex(void);
%}
%token DIGIT LETTER
%left '+' '-'
%left '*' '/'
%right UMINUS
```

```
%%
input:
expr { printf("Valid Arithmetic Expression\n"); exit(0); }
;
expr:
expr '+' expr { /* addition */ }
| expr '-' expr { /* subtraction */ }
| expr '*' expr { /* multiplication */ }
| expr '/' expr { /* division */ }
| '-' expr %prec UMINUS { /* unary minus */ }
| '(' expr ')' { /* parentheses */ }
| LETTER { /* identifier */ }
| DIGIT { /* number */ }
;
%%
int main() {
    printf("Enter the expression:\n");
    yyparse();
    printf("Parsing completed\n");
    return 0;
}
void yyerror(char *msg) {
    printf("Invalid Arithmetic Expression\n");
    exit(1);
}
```

OUTPUT

```
user-HP-Pavilion-Notebook% gedit calc.l
user-HP-Pavilion-Notebook% gedit calc.y
user-HP-Pavilion-Notebook% flex calc.l
user-HP-Pavilion-Notebook% bison -dy calc y
user-HP-Pavilion-Notebook% gcc lex.yy.c y.tab.c -o MR
user-HP-Pavilion-Notebook% ./MR
Enter an expression (e.g., (5 + 3) * 2):
4+3
Operand: 4
Operand: 3
Result: 7
user-HP-Pavilion-Notebook%
```

RESULT

Program to generate a YACC specification to recognize a valid arithmetic expression that uses operators +, , *, / and parenthesis.using Yacc has been executed successfully and CO1 has achieved.

Date: 12-09-25

EXPERIMENT - 4.3

SIMPLE CALCULATOR USING LEX & YACC

AIM

To write a program to implement a simple calculator using Lex and Yacc.

COURSE OUTCOME

CO1: Implement lexical analyzer and syntax analyzer using the tool LEX and YACC.

ALGORITHM

Step 1: Start the program.

Step 2: Include necessary header files in the Lex and Yacc files.

Step 3: In the Lex file:

Step 3.1: Recognize numbers as tokens.

Step 3.2: Ignore whitespace characters.

Step 3.3: Return operator characters to Yacc.

Step 4: In the Yacc file:

Step 4.1: Define grammar rules for expressions.

Step 4.2: Handle addition, subtraction, multiplication, and division.

Step 4.3: Check for division by zero and print an error message if it occurs.

Step 4.4: Print the computed result.

Step 5: Implement `yyerror()` to handle syntax errors.

Step 6: Implement `main()` to start parsing expressions.

Step 7: End the program.

PROGRAM

`calculator.l`

```
%{
#include "y.tab.h"
#include <stdlib.h>
%}
[0-9]+ { yylval = atoi(yytext); return NUMBER; }
[ \t\n]+ ;
. { return yytext[0]; }
%%
int yywrap() {
    return 1;
}
```

`calculator.y`

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

```
#include <stdlib.h>
int yylex(void);
void yyerror(const char *msg);
%}
%token NUMBER
%%
calc:
exp '\n' { printf("Result = %d\n", $1); }
;
exp:
NUMBER { $$ = $1; }
| exp '+' exp { $$ = $1 + $3; }
| exp '-' exp { $$ = $1 - $3; }
| exp '*' exp { $$ = $1 * $3; }
| exp '/' exp {
    if ($3 == 0) {
        yyerror("Division by zero");
        $$ = 0;
    } else {
        $$ = $1 / $3;
    }
}
;
%%
int yyerror(const char *msg) {
```

```
    printf("Error: %s\n", msg);
    return 0;
}
int main() {
    printf("Enter an expression (with +, -, *, /):\n");
    yyparse();
    return 0;
}
```

OUTPUT

```
student@P31:~$ lex calc.l
student@P31:~$ yacc -d calc.y
student@P31:~$ gcc lex.yy.c y.tab.c
student@P31:~$ ./a.out
Enter expressions (Ctrl+D to exit):
1+2
Result:3
Enter expression (Ctrl+D to exit ):
5*2
Result:10
```

RESULT

The program has been successfully executed and the output verified. Thus CO1 has been achieved.

Date: 02-08-25

EXPERIMENT - 5

DESIGN OF A DETERMINISTIC FINITE AUTOMATA ACCEPTS A LANGUAGE

AIM

To implementa program to accept strings ending in '01' using DFA.

COURSE OUTCOME

CO3: Design NFA and DFA for a problem and write programs to perform operations on it.

ALGORITHM

Step 1: Start.

Step 2: Declare and initialize variables:

Step 2:.1 Declare a character array `str[max]` to store the input string.

Step 2:.2 Declare a character variable `f` and initialize it to `'a'` (initial state).

Step 3: Prompt the user to enter a binary string and read it using `scanf()`.

Step 4: Traverse each character of the input string using a `for` loop until the null character is encountered:

Step 4:.1 Use a `switch` statement on the current state variable `f`.

Step 4:.2 For each case:

Step 4:.2.1 `Case 'a':`

- If input character is `'0'`, move to state `'b'`.
- Else, stay in state `'a'`.

Step 4:.2.2 `Case 'b':`

- If input character is `'0'`, stay in state `'b'`.
- Else, move to state `'c'`.

Step 4:.2.3 `Case 'c':`

- If input character is `'0'`, move to state `'b'`.
- Else, move to state `'a'`.

Step 4:.2.4 `Default:`

- Print "Invalid Input".

Step 5: After the loop completes:

Step 5:.1 If final state `f` is `'c'`, print "The string is accepted."

Step 5:.2 Else, print "The string is not accepted."

Step 6: Stop.

PROGRAM

```
#include<stdio.h>
#define max 100
int main(){
```

```
char str[max],f = 'a';
printf("Enter a string: ");
scanf("%s",str);
for(int i=0;str[i]!='\0';i++)
{
    switch(f){
        case 'a':
            if(str[i]== '0'){
                f = 'b';
            }
            else
                f= 'a';
            break;
        case 'b':
            if(str[i]== '0'){
                f = 'b';
            }
            else
                f= 'c';
            break;
        case 'c':
            if(str[i]== '0'){
                f = 'b';
            }
            else
```

```
                    f= 'a';
                break;
            default:
                printf("Invalid Input\n");
                break;
        }
    }
    if(f == 'c'){
        printf("The string %s is accepted\n",str);
    }
    else{
        printf("The string %s  is not accepted\n",str);
    }
}
```

OUTPUT

```
student@P33:~/cs29$ gcc dfa.c
student@P33:~/cs29$ ./a.out
Enter a string: 0111101
The string 0111101 is accepted
student@P33:~/cs29$ ./a.out
Enter a string: 01010
The string 01010  is not accepted
student@P33:~/cs29$
```

RESULT

The program implements a DFA to accept binary strings ending in ’01’. After processing the string, it prints the acceptance status. The program has been executed successfully and CO3 has been achieved.

Date: 02-08-25

EXPERIMENT - 6

E-CLOSURE OF GIVEN NFA WITH E-TRANSITION

AIM

Write a program to find ε – closure of all states of any given NFA with ε transition.

COURSE OUTCOME

CO3: Design NFA and DFA for a problem and write programs to perform operations on it.

ALGORITHM

Step 1: Start.

Step 2: Initialize Data Structures

Step 2:1. Initialize a 2D array `transitions[i][3]` to store the transitions of the NFA:

- `transitions[i][0]` = current state
- `transitions[i][1]` = input symbol
- `transitions[i][2]` = destination state

Step 2:2. Initialize a list `closure[]` to store the ε-closure states.

Step 3: Define Helper Functions

Step 3:1. Define the function `is_in_closure(state)` that checks if the state is already in `closure[]`.

Step 3:2. Define the function `add_to_closure(state)`:

- If the state is not in `closure[]`, add it.

Step 4: Define the Recursive Function `find_epsilon_closure(state)`

Step 4:1. Add the current state to `closure[]`.

Step 4:2. For each transition:

- If the transition is from the current state and the input is ε (represented as `"e"`), recursively call `find_epsilon_closure()` on the destination state if it is not already in the closure.

Step 5: Input and Execution

Step 5:1. Open the input file `input1.dat` and read the transitions into the `transitions[][]` array.

Step 5:2. Prompt the user to enter a start state.

Step 5:3. Initialize the closure list and call `find_epsilon_closure(start_state)`.

Step 6: Output

Step 6:1. Print all the states in the computed ε-closure.

Step 7: Stop.

PROGRAM

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

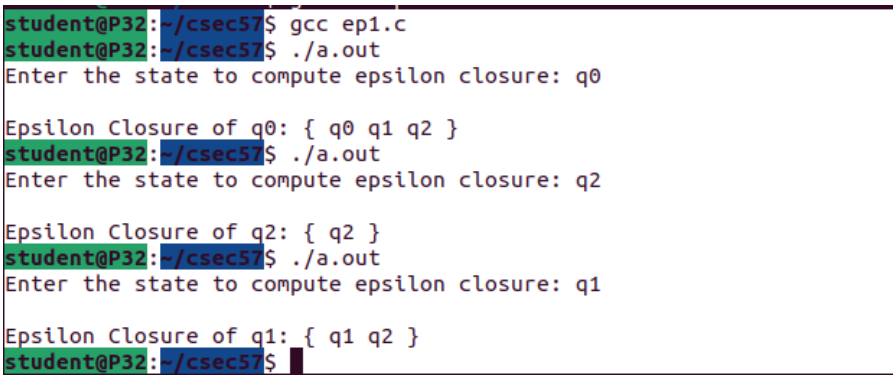
```
#define max 100
int main(){
    char str[max],f = 'a';
    printf("Enter a string: ");
    scanf("\%s",str);
    for(int i=0;str[i]!='\0';i++)
    {
        switch(f){
            case 'a':
                if(str[i]== '0'){
                f = 'b';
                }
                else
                    f= 'a';
                break;
            case 'b':
                if(str[i]== '0'){
                f = 'b';
                }
                else
                    f= 'c';
                break;
            case 'c':
                if(str[i]== '0'){
                f = 'b';
```

```
                }
                else
                    f= 'a';
                break;
            default:
                printf("Invalid Input\n");
                break;
        }
    }
    if(f == 'c'){
        printf("The string \%s is accepted\n",str);
    }
    else{
        printf("The string \%s  is not accepted\n",str);
    }
}
```

Input: input1.dat

```
state1  input   state2
  q0      x       q0
  q0      e       q1
  q1      y       q1
  q1      e       q2
  q2      z       q2
```

OUTPUT



RESULT

The program to find ε – closure of all states of any given NFA with ε transition has been executed successfully and CO3 has been achieved.

Date: 19-09-25

EXPERIMENT - 7

RECURSIVE DESCENT PARSER

AIM

Design and implement a recursive descent parser for a given grammar.

COURSE OUTCOME

CO4: Design and Implement Top-Down parsers.

Grammar

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid id$$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$

ALGORITHM

Step 1: Initialize

Step 1:1. Read the input string from the user and store it in `input`.

Step 1:2. Initialize index `i = 0`.

Step 1:3. Set global variable `error = 0`.

Step 2: Start Parsing

Step 2:1. Call function `E()`.

Step 2:2. After `E()` returns, check if:

Step 2:2.1. All characters have been consumed (`i == strlen(input)`).

Step 2:2.2. `error == 0`.

Step 2:3. If both are true, accept the input; otherwise, reject it.

Step 3: Procedure E()

Step 3:1. Call `T()`.

Step 3:2. Call `Eprime()`.

Step 4: Procedure Eprime()

Step 4:1. If current character is `'+'`:

Step 4:1.1. Increment `i`.

Step 4:1.2. Call `T()`.

Step 4:1.3. Call `Eprime()` recursively.

Step 4:2. Else, return (epsilon production).

Step 5: Procedure T()

Step 5:1. Call `F()`.

Step 5:2. Call `Tprime()`.

Step 6: Procedure Tprime()

Step 6:1. If current character is `'*'`:

Step 6:1.1. Increment `i`.

Step 6:1.2. Call `F()`.

Step 6:1.3. Call `Tprime()` recursively.

Step 6:2. Else, return (epsilon production).

Step 7: Procedure F()

Step 7:1. If current character is an alphabet (i.e., `id`):

- Increment `i`.

Step 7:2. Else if it is `'('`:

Step 7:2.1. Increment `i`.

Step 7:2.2. Call `E()`.

Step 7:2.3. If current character is `')'`, increment `i`.

Step 7:2.4. Else, set `error = 1`.

Step 7:3. Else, set `error = 1`.

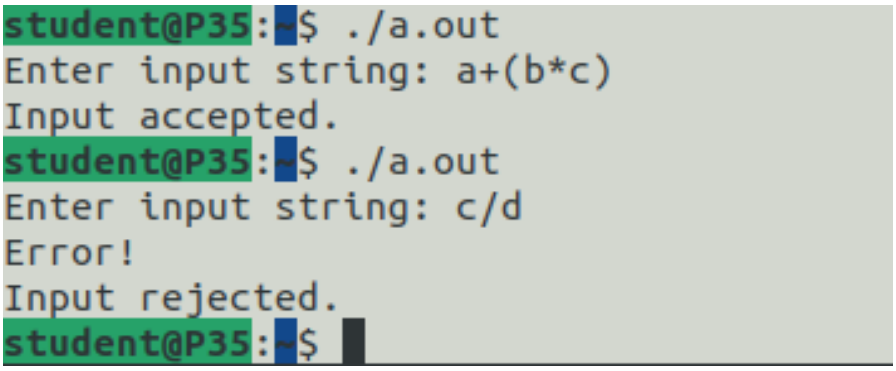
Step 8: End.

PROGRAM

```
#include<stdio.h>
#define max 100
int main(){
    char str[max],f = 'a';
    printf("Enter a string: ");
    scanf("%s",str);
    for(int i=0;str[i]!='\0';i++)
    {
        switch(f){
            case 'a':
                if(str[i]== '0'){
                    f = 'b';
                }
                else
                    f= 'a';
                break;
            case 'b':
                if(str[i]== '0'){
                    f = 'b';
                }
                else
                    f= 'c';
                break;
```

```
            case 'c':
                if(str[i]== '0'){
                    f = 'b';
                }
                else
                    f= 'a';
                break;
            default:
                printf("Invalid Input\n");
                break;
        }
    }
    if(f == 'c'){
        printf("The string %s is accepted\n",str);
    }
    else{
        printf("The string %s  is not accepted\n",str);
    }
}
```

OUTPUT



RESULT

The program to design and implement a recursive descent parser has been executed successfully and CO4 has been achieved.

Date: 19-09-25

EXPERIMENT - 8

SHIFT REDUCE PARSER

AIM

Design and implement a shift reduce parser for a given grammar.

COURSE OUTCOME

CO5: Design and Implement Bottom-Up parsers.

ALGORITHM

Step 1: Start.

Step 2: Display the grammar:

- $E \rightarrow E + E$
- $E \rightarrow E * E$
- $E \rightarrow (E)$
- $E \rightarrow id$

Step 3: Prompt the user to enter an input string.

Step 4: Read the input string and store it in a character array.

Step 5: Initialize variables and define:

- `a[]` – input string
- `stk[]` – parsing stack
- `act[]` – stores current action (”SHIFT-¿”)

Step 6: Display table headers: Stack, Input, Action.

Step 7: Loop over the input string:

Step 7:.1 If the next two characters are ”id”:

- Shift ”id” to the stack.
- Mark ”id” in input as processed.
- Display the current stack, input, and action.
- Call the `check()` function to apply reduction rules.

Step 7:.2 Else:

- Shift the current symbol to the stack.
- Mark the symbol as processed.
- Display the current stack, input, and action.
- Call the `check()` function.

Step 8: In the `check()` function, apply reduction rules:

Step 8:.1 Rule 1: Replace `id` with `E`.

Step 8:.2 Rule 2: Replace `E+E` with `E`.

Step 8:.3 Rule 3: Replace `E*E` with `E`.

Step 8:.4 Rule 4: Replace `(E)` with `E`.

Step 8:.5 After each reduction, update stack and index accordingly.

Step 9: After processing the input:

Step 9:.1 If the stack contains only `E` and the input is empty, the string is accepted.

Step 9:.2 Else, the string is rejected.

Step 10: Stop.

PROGRAM

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

int k = 0, z = 0, i = 0, j = 0, c = 0;
char a[16], ac[20], stk[15], act[10];

void check();
void printResult(int accepted);

int main() {
    // Grammar definitions
    puts("\nGRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("Enter input string: ");

    fgets(a, sizeof(a), stdin);
    a[strcspn(a, "\n")] = '\0';
    c = strlen(a);
```

```
strcpy(act, "SHIFT->");

puts("stack \t input \t action");

for (k = 0, i = 0; j < c; k++, i++, j++) {
    if (a[j] == 'i' && a[j + 1] == 'd') {
        stk[i] = a[j];
        stk[i + 1] = a[j + 1];
        stk[i + 2] = '\0';
        a[j] = ' ';
        a[j + 1] = ' ';
        printf("\n$%s\t%s$\t%sid", stk, a, act);
        check();
    } else {
        stk[i] = a[j];
        stk[i + 1] = '\0';
        a[j] = ' ';
        printf("\n$%s\t%s$\t%ssymbol", stk, a, act);
        check();
    }
}

if (stk[0] == 'E' && stk[1] == '\0' && a[0] == ' ') {
    printResult(1);  // Accepted
}
```

```
    else if (stk[0] == 'E' && a[0] == ' ') {
        printResult(1);  // Accepted
    } else {
        printResult(0);  // Rejected
    }

    return 0;
}

void check() {
    strcpy(ac, "REDUCE TO E");

    // Rule 1: E -> id
    for (z = 0; z < c; z++) {
        if (stk[z] == 'i' && stk[z + 1] == 'd') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            j++;
        }
    }

    // Rule 2: E -> E + E
    for (z = 0; z < c; z++) {
        if (stk[z] == 'E' && stk[z + 1] == '+' && stk[z + 2] == 'E') {
```

```
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i - 2;
        }
    }

    // Rule 3: E -> E * E
    for (z = 0; z < c; z++) {
        if (stk[z] == 'E' && stk[z + 1] == '*' && stk[z + 2] == 'E') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i - 2;
        }
    }

    // Rule 4: E -> ( E )
    for (z = 0; z < c; z++) {
        if (stk[z] == '(' && stk[z + 1] == 'E' && stk[z + 2] == ')') {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
```

```
            printf("\n$%s\t%s$\t%s", stk, a, ac);
            i = i - 2;
        }
    }
}

void printResult(int accepted) {
    if (accepted) {
        printf("\nInput string is ACCEPTED.");
    } else {
        printf("\nInput string is REJECTED.");
    }
}
```

OUTPUT

```
student@P35:~$ ./a.out

GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
Enter input string:
(id*id)+id
stack    input   action

$(       id*id)+id$     SHIFT->symbol
$(id       *id)+id$     SHIFT->id
$(E        *id)+id$     REDUCE TO E
$(E*        id)+id$     SHIFT->symbol
$(E*id        )+id$     SHIFT->id
$(E*E         )+id$     REDUCE TO E
$(E           )+id$     REDUCE TO E
$(E)           +id$     SHIFT->symbol
$E             +id$     REDUCE TO E
$E+             id$     SHIFT->symbol
$E+id             $     SHIFT->id
$E+E              $     REDUCE TO E
$E                $     REDUCE TO E
Input string is ACCEPTED.student@P35:~$ ./a.out
```

RESULT

The program to design and implement a shift reduce parser has been executed successfully and CO5 has been achieved.

Date: 24-09-25

EXPERIMENT - 9

FIRST AND FOLLOW

AIM

Write a program to find First and Follow of any given grammar.

COURSE OUTCOME

CO6: Implement intermediate code for expressions.

ALGORITHM

Step 1: Start.

Step 2: Declare variables and arrays:

- `a[10][10]` to store the grammar productions.
- `f[10]` to store FIRST or FOLLOW results.
- Integers for counters and flags.

Step 3: Read the number of productions `n` from the user.

Step 4: Read `n` grammar productions from the user and store in `a[][]`.

Step 5: Repeat the following steps until the user chooses to exit:

Step 5:.1 Reset result index `m = 0`.

Step 5:.2 Prompt the user to enter a grammar symbol (non-terminal) whose FIRST and FOLLOW sets are to be computed.

Step 5:.3 Call the function `first(c)` to compute FIRST set of the symbol:

Step 5:.3.1 If the symbol is a terminal, add it to the result.

Step 5:.3.2 If the symbol is a non-terminal:

- Check each production where it appears on the left-hand side.
- If the right-hand side starts with:
  - `$`, add `$` to FIRST.
  - A terminal, add the terminal to FIRST.
  - A non-terminal, recursively call `first()`.

Step 5:.4 Display the computed FIRST set.

Step 5:.5 Reset result index and buffer.

Step 5:.6 Call the function `follow(c)` to compute FOLLOW set of the symbol:

Step 5:.6.1 If the symbol is the start symbol (first production), add `$` to FOLLOW.

Step 5:.6.2 For every production, scan the right-hand side:

- If the symbol is followed by another symbol:
  - Call `first()` on the next symbol.
- If the symbol is at the end of a production and is not the same as the left-hand side:
  - Recursively call `follow()` on the left-hand side symbol.

Step 5:.7 Display the computed FOLLOW set.

Step 5:.8 Ask the user if they want to continue (yes = 1 / no = 0).

Step 6: Stop.

PROGRAM

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

int n, m= 0, p, i= 0,j= 0;
char a[10][10], f[10];

void follow(char c);
void first(char c);
int main(){
    int i, z;
    char c,ch;
    printf("Enter the no of productions:\n");
    scanf("%d",&n);
    printf("Enter the productions:\n");
    for(i=0;i<n;i++)
    scanf("%s%c",a[i],&ch);
```

```
    do{
        m=0;
        printf("Enter element whose FIRST and FOLLOW is to be found:\n");
        scanf("%c",&c);

        first(c);
        printf("First(%c) = {",c);
        for(i=0;i<m;i++)
            printf("\t%c",f[i]);
        printf("}\n");
        strcpy(f," ");
        m=0;
        follow(c);
        printf("Follow(%c)={",c);
        for(i=0;i<m;i++)
            printf("\t%c",f[i]);
        printf("}\n");
        printf("Continue (0/1)?");
        scanf("%d%c",&z,&ch);
    }while(z==1);
    return 0;
}
void first(char c){
    int k;
```

```
    if(!isupper(c))
    f[m++]=c;
    for(k=0;k<n;k++){
        if(a[k][0] == c){
            if(a[k][2] == '$')
                f[m++] = '$';
            else if(islower(a[k][2]))
                f[m++] = a[k][2];
            else
                first(a[k][2]);
        }
    }
}
void follow(char c){
    if(a[0][0] == c)
    f[m++] = '$';
    for(i=0;i<n;i++){
        for(j=2;j<strlen(a[i]);j++){
            if(a[i][j] == c){
                if(a[i][j+1] !='\0')
                    first(a[i][j+1]);
                if(a[i][j+1] == '\0' && c != a[i][0])
                    follow(a[i][0]);
            }
        }
```

```
    }
}
```

OUTPUT

```
student@P30:~$ gcc first_follow.c
student@P30:~$ ./a.out
Enter the no of productions:
5
Enter the productions:
S=AbCd
A=Cf
A=a
C=gE
E=h
Enter element whose FIRST and FOLLOW is to be found:
S
First(S) = {    g       a}
Follow(S)={     $}
Continue (0/1)?1
Enter element whose FIRST and FOLLOW is to be found:
A
First(A) = {    g       a}
Follow(A)={     b}
Continue (0/1)?E
Enter element whose FIRST and FOLLOW is to be found:
First(E) = {    h}
Follow(E)={     d       f}
Continue (0/1)?0
```

RESULT

The program to find FIRST and FOLLOW of any given grammar has been executed successfully and CO6 has been achieved.

Date: 08-10-25

EXPERIMENT - 10

INTERMEDIATE CODE GENERATION

AIM

Implement Intermediate code generation for simple expressions.

COURSE OUTCOME

CO6: Implement intermediate code for expressions.

ALGORITHM

Step 1: Start.

Step 2: Initialize character arrays:

- `stack[]` for holding operands.
- `input[]` for storing postfix expression.
- `input1[]` to read the infix expression from the user.
- `op[]` and `priority[]` to manage the operator stack and their precedence.
- `intermediates[][]` to store the intermediate code (three-address instructions).

Step 3: Read the infix arithmetic expression into `input1[]`.

Step 4: Convert the infix expression to postfix using the Shunting Yard algorithm:

Step 4:.1 Initialize the operator stack with '('.

Step 4:.2 For each character in `input1[]`:

- If it is an operand, append it to `input[]`.
- If it is '(', push it onto `op[]`.
- If it is an operator (+, -, *, /):
  i. Determine its precedence (1 for +/-, 2 for */).
  ii. While the precedence of the top of the stack is greater or equal, pop from the stack to `input[]`.
  iii. Push the current operator to the stack.
- If it is ')', pop operators from the stack to `input[]` until '(' is encountered.

Step 4:.3 After processing all characters, pop remaining operators from the stack to complete the postfix expression.

Step 5: Display the generated postfix expression.

Step 6: Generate intermediate code (three-address code) from the postfix expression:

Step 6:.1 Traverse each character in the postfix expression:

- If it is an operand, push it to the `stack[]`.
- If it is an operator:
  i. Pop the top two operands from the stack.

ii. Form an intermediate instruction: `T = operand1 operator operand2`.

iii. Store it in the `intermediates[][]` array.

iv. Push the temporary result symbol (like A, B, ...) back to the stack.

Step 7: Print all the intermediate instructions.

Step 8: Stop.

PROGRAM

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

int main() {
    char stack[20], input[20], inter = 'A', intermediates[20][20], input1[20],
    op[20];
    int priority[20], priority1 = 0;
    int k = 0, i = 0, j = 0, l = 0;

    printf("Enter the Input arithmetic expression: ");
    scanf("%s", input1);

    printf("\n--- Postfix Conversion ---\n");
```

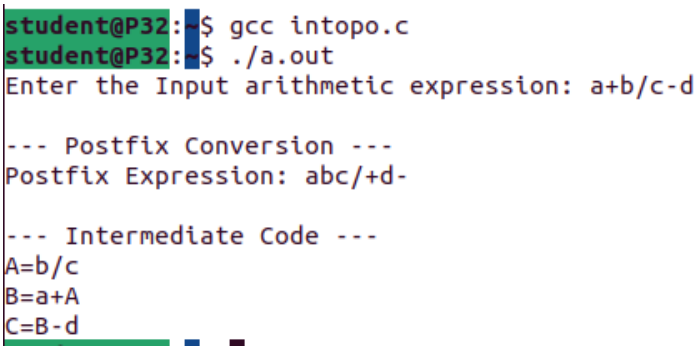
```
// Initialize operator stack
op[0] = '(';
priority[0] = 0;
l = 1;

i = 0;
while (input1[i] != '\0') {
    if (isalnum(input1[i])) {
        input[k++] = input1[i];
    }
    else if (input1[i] == '(') {
        op[l] = '(';
        priority[l] = 0;
        l++;
    }
    else if (input1[i] == '+' || input1[i] == '-' || input1[i] == '*' ||
            input1[i] == '/') {
        if (input1[i] == '*' || input1[i] == '/')
            priority1 = 2;
        else
            priority1 = 1;

        while (priority[l - 1] >= priority1) {
            l--;
            input[k++] = op[l];
```

```
            }
            op[l] = input1[i];
            priority[l++] = priority1;
        }
        else if (input1[i] == ')') {
            while (op[l - 1] != '(') {
                l--;
                input[k++] = op[l];
            }
            l--; // Pop '('
        }
        i++;
    }

    // Pop remaining operators
    while (l > 1) {
        l--;
        input[k++] = op[l];
    }
    input[k] = '\0';

    printf("Postfix Expression: %s\n", input);

    // =========================================================
    // ============= Intermediate Code Generation ============
```

```
// ======================================================


i = 0;

k = 0;

j = 0;


while (input[i] != '\0') {
    if (isalnum(input[i])) {
        stack[k++] = input[i];
    }
    else if (input[i] == '+' || input[i] == '-' || input[i] == '*' ||
            input[i] == '/') {
        intermediates[j][0] = inter;
        intermediates[j][1] = '=';
        intermediates[j][2] = stack[k - 2];
        intermediates[j][3] = input[i];
        intermediates[j][4] = stack[k - 1];
        intermediates[j][5] = '\0';


        // Update stack
        k = k - 2;
        stack[k++] = inter;
        inter++;
        j++;
    }
```

```
        i++;
    }

    printf("\n--- Intermediate Code ---\n");
    for (i = 0; i < j; i++) {
        printf("%s\n", intermediates[i]);
    }

    return 0;
}
```

OUTPUT



RESULT

The program to implement intermediate code generation for simple expressions has been executed successfully and CO6 has been achieved.

Date: 08-10-25

EXPERIMENT - 11

CODE OPTIMIZATION

AIM

Implement Simulation of code optimization Techniques.

COURSE OUTCOME

CO6: Implement intermediate code for expressions.

ALGORITHM

Step 1: Start.

Step 2: Declare required variables:

- `c[100]` to store the input expression.
- `pre[10], post[10]` to hold operand strings.
- `ne[10]` to hold the result as a string.
- Integer variables for indexing and arithmetic: `i, j, k, l, m, n, x, y, z`.

Step 3: Read the input expression as a string into `c`.

Step 4: Initialize `i = 0` to start traversing the expression.

Step 5: Traverse the expression from left to right:

Step 5:.1 If `c[i]` is an operator (`'+'`, `'-'`, `'*'`, `'/'`):

Step 5:.1.1 Set `j = i - 1` and `k = i + 1` to identify operands before and after the operator.

Step 5:.1.2 Check if both operands are digits using `isdigit()`.

Step 5:.1.3 If valid digits:

Step 5:.1.3.1 Extract right operand starting from `c[k]` into `post[]`.

Step 5:.1.3.2 Extract left operand in reverse from `c[j]` into `pre[]`.

Step 5:.1.3.3 Convert both `pre[]` and `post[]` to integers using `atoi()`.

Step 5:.1.3.4 Perform the arithmetic operation based on the operator:

- `+` : addition
- `-` : subtraction
- `*` : multiplication
- `/` : division

Step 5:.1.3.5 Store the result in `ne[]` using `sprintf()`.

Step 5:.1.3.6 Replace the operation and operands in `c[]` with the result:

- Copy the result back into `c[]` at the appropriate index.
- Shift remaining part of the expression forward to maintain the structure.

Step 5:.1.3.7 Update `i` to new position after modification.

Step 6: Continue until end of the string is reached.

Step 7: Print the final result stored in `c[]`, which contains the evaluated expression.

Step 8: Stop.

PROGRAM

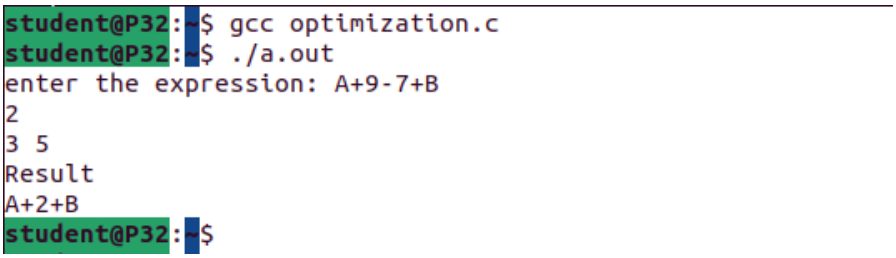
```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

void main()
{
    char c[100],pre[10],post[10],ne[10];
    int i,j,k,l,m,n,x,y,z;

    printf("enter the expression: ");
    scanf("%s",c);

    i=0;
    while(c[i]!='\0')
    {
        if(c[i]=='+' || c[i]=='-' || c[i]=='*' || c[i]=='/')
        {
            j=i-1;
            k=i+1;

            if(isdigit(c[k])&&isdigit(c[j]))
```

```
{
    l=0;
    m=0;
    while(isdigit(c[k]))
    {
        post[l++]=c[k++];
    }
    post[1]='\0';

    while(isdigit(c[j]))
    {
        pre[m++]=c[j--];
    }
    pre[m]='\0';

    x=atoi(pre);
    y=atoi(post);

    switch(c[i])
    {
        case '+':
            z=x+y;
            break;
        case '-':
            z=x-y;
```

```
                break;
            case '*':
                z=x*y;
                break;
            case '/':
                z=x/y;
                break;
        }
        sprintf(ne,"%d",z);
        printf("%s",ne);
        n=0;
        while(ne[n]!='\0')
        {
            c[++j]=ne[n++];
        }
        i=j;
        j++;

        printf("\n%d %d\n",j,k);
        while(c[k]!='\0')
        {
            c[j++]=c[k++];
        }
        c[j]='\0';
    }
```

```
        }
        i++;
    }
    printf("Result\n%s\n",c);
}
```

OUTPUT



RESULT

The program to implement code optimization technique has been executed successfully and CO6 has been achieved.

Date: 15-10-25

EXPERIMENT - 12

BACKEND OF THE COMPILER

AIM

Implement the back end of the compiler.

COURSE OUTCOME

CO6: Implement intermediate code for expressions.

ALGORITHM

Step 1: Start.

Step 2: Declare required variables:

- `c[100]` to store the input expression.
- `pre[10], post[10]` to hold operand strings.
- `ne[10]` to hold the result as a string.
- Integer variables for indexing and arithmetic: `i, j, k, l, m, n, x, y, z`.

Step 3: Read the input expression as a string into `c`.

Step 4: Initialize `i = 0` to start traversing the expression.

Step 5: Traverse the expression from left to right:

Step 5:.1 If `c[i]` is an operator (`'+'`, `'-'`, `'*'`, `'/'`):

Step 5:.1.1 Set `j = i - 1` and `k = i + 1` to identify operands before and after the operator.

Step 5:.1.2 Check if both operands are digits using `isdigit()`.

Step 5:.1.3 If valid digits:

Step 5:.1.3.1 Extract right operand starting from `c[k]` into `post[]`.

Step 5:.1.3.2 Extract left operand in reverse from `c[j]` into `pre[]`.

Step 5:.1.3.3 Convert both `pre[]` and `post[]` to integers using `atoi()`.

Step 5:.1.3.4 Perform the arithmetic operation based on the operator:

- `+` : addition
- `-` : subtraction
- `*` : multiplication
- `/` : division

Step 5:.1.3.5 Store the result in `ne[]` using `sprintf()`.

Step 5:.1.3.6 Replace the operation and operands in `c[]` with the result:

- Copy the result back into `c[]` at the appropriate index.
- Shift remaining part of the expression forward to maintain the structure.

Step 5:.1.3.7 Update `i` to new position after modification.

Step 6: Continue until end of the string is reached.

Step 7: Print the final result stored in `c[]`, which contains the evaluated expression.

Step 8: Stop.

PROGRAM

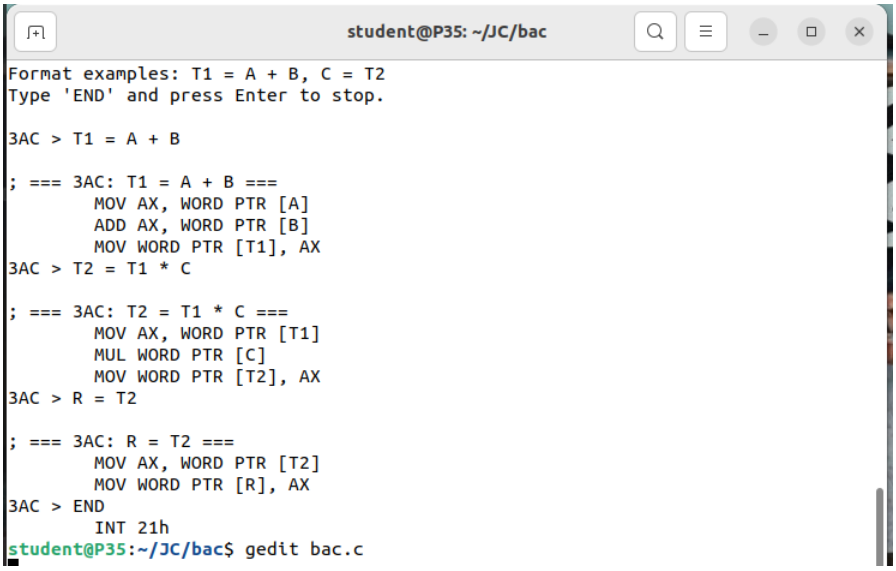
```
#include <stdio.h>
#include <string.h>

#define MAX_INSTRUCTIONS 100

typedef struct {
    char op[5];    // operation: +, -, *, /
    char arg1[10];
    char arg2[10];
    char result[10];
} TACInstruction;

TACInstruction instructions[MAX_INSTRUCTIONS];
int instruction_count = 0;

// Function to read TAC instructions from user input
void readIntermediateCode() {
    printf("Enter number of TAC instructions: ");
    scanf("%d", &instruction_count);
    getchar(); // consume newline
```

```
    printf("Enter TAC instructions in the format:\n");
    printf("result = arg1 op arg2\n");
    printf("For example: t1 = a + b\n\n");

    for (int i = 0; i < instruction_count; i++) {
        char line[50];
        printf("Instruction %d: ", i + 1);
        fgets(line, sizeof(line), stdin);

        // Parse the line
        // Format: result = arg1 op arg2
        sscanf(line, "%s = %s %s %s", instructions[i].result, instructions[i].arg1

        // Remove any trailing newline in result if present
        int len = strlen(instructions[i].result);
        if (instructions[i].result[len - 1] == '\n')
            instructions[i].result[len - 1] = '\0';
    }
}

// Function to generate assembly code from TAC
void generateAssembly() {
    printf("\nAssembly code generated from TAC:\n");

    for (int i = 0; i < instruction_count; i++) {
```

```
        TACInstruction instr = instructions[i];

        if (strcmp(instr.op, "+") == 0) {
            printf("MOV R1, %s\n", instr.arg1);
            printf("ADD R1, %s\n", instr.arg2);
            printf("MOV %s, R1\n\n", instr.result);
        } else if (strcmp(instr.op, "-") == 0) {
            printf("MOV R1, %s\n", instr.arg1);
            printf("SUB R1, %s\n", instr.arg2);
            printf("MOV %s, R1\n\n", instr.result);
        } else if (strcmp(instr.op, "*") == 0) {
            printf("MOV R1, %s\n", instr.arg1);
            printf("MUL R1, %s\n", instr.arg2);
            printf("MOV %s, R1\n\n", instr.result);
        } else if (strcmp(instr.op, "/") == 0) {
            printf("MOV R1, %s\n", instr.arg1);
            printf("DIV R1, %s\n", instr.arg2);
            printf("MOV %s, R1\n\n", instr.result);
        } else {
            printf("Unknown operation: %s\n", instr.op);
        }
    }
}

int main() {
```

```
    readIntermediateCode();
    generateAssembly();
    return 0;
}
```

OUTPUT



RESULT

The program to implement the back end of the compiler. has been executed successfully and CO6 has been achieved.