C. U. Shah University, Wadhwan City

Translator Design (4TE07TDE1)

7th SEMESTER B. Tech

TDE

C. U. SHAH COLLEGE OF
ENGINEERING & TECHNOLOGY

यशो सर्वदा वर्धमानम्

C. U. Shah College of
Engineering & Technology
Wadhwan City - 363030

Faculty of Technology & Engineering

C. U. Shah College of Engineering & Technology

C. U. SHAH COLLEGE OF ENGINEERING & TECHNOLOGY
यशो सर्वदा वर्धमानम्

(Managed By Wardhman Bharti Trust)

Accredited by NAAC with B Grade

WADWAN CITY – 363030

C U SHAH UNIVERSITY
VARDHMAN BHARTI TRUST
ESTD - 2013
EMPOWERING YOUR DESTINATION

Certificate

This is to certify that

Mr/Ms. ____

Enrollment No. ______________________*Of* __ Semester

B. Tech Courses in Computer Engineering

has satisfactory completed his/her Term work in

____

with in four Walls of Laboratory / Drawing Hall of This College during the year 2025

Staff In charge.

Date of Submission : __________ Head of The Department

List Of Practical

Translator Design (4TE07TDE1)

Student Name :

Student Enrollment No. :

Sr. No	Date	Practical	Page No	Marks	Sign of Faculty
01		Write a C Program to Scan and Count the number of characters, words, and lines in a file.			
02		Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.			
03		To Write a C program to develop a lexical analyzer(without flex) to recognize few Keywords, words, numbers in C			
04		To Write a program to recognize few Keywords, words, numbers for c using flex			
05		To write a program for implementing a Lexical analyser using LEX tool.			
06		To write a program for recognizing a valid arithmetic expression that uses operator +, –, * and / using YACC (Yet Another Compiler-Compiler).			
07		To write a C program to check whether the type of all variables in an expression are valid or not.			
08		To write a C Program to Generate Machine Code from the Abstract Syntax Tree using the specified machine instruction formats.			
09		To write a C Program to Generate Machine Code given assembly code.			

PRACTICAL 1

AIM:

Write a C Program to Scan and Count the number of characters, words, and lines in a file.

INTRODUCTION:

Text file is required to execute the program. You can create dummy text file, name test.txt, read the file for the experiment

Write c program to read a dummy text files which contains the paragraphs, sentences and words. C program will counts the lines, words and characters.

ALGORITHM:

1. Start
2. Read the input file/text
3. Initialize the counters for characters, words, lines to zero 4. Scan the characters, words, lines
5. increment the respective counters
6. Display the counts
7. End

PROGRAM:

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

int main() {
    FILE *file;
    char filename[100];
    char ch;
    int characters = 0, words = 0, lines = 0;
    int inWord = 0;

    // Ask user for filename
    printf("Enter the filename: ");
    scanf("%s", filename);

    // Open file in read mode
    file = fopen(filename, "r");

    // Error handling
    if (file == NULL) {
        printf("Could not open file %s\n", filename);
        return 1;
    }

    // Read character by character
    while ((ch = fgetc(file)) != EOF) {
        characters++;

        // Count lines
        if (ch == '\n')
            lines++;

        // Count words
        if (isspace(ch)) {
            inWord = 0;
        } else if (inWord == 0) {
            inWord = 1;
            words++;
        }
    }

    // Close the file
    fclose(file);

    // Display results
    printf("\nFile: %s\n", filename);
    printf("Characters: %d\n", characters);
    printf("Words     : %d\n", words);
    printf("Lines     : %d\n", lines);

    return 0;
}
```

INPUT:

In dummy.txt

```
These are few sentences in
 mini Language
```

OUTPUT:

```
No of characters: 35
No of words : 7
No of lines : 2
```

RESULT:

QUESTIONS:

1. What is Compiler?

2. List various language Translators.

3. Is it necessary to translate a HLL program? Explain.

4. List out the phases of a compiler?

5. Which phase of the compiler is called an optional phase? why?

PRACTICAL 2

AIM:

Write a C Program to implement NFAs that recognize identifiers, constants, and operators of the mini language.

INTRODUCTION:

simulates NFAs to recognize whether a given token is an identifier, constant, or operator, and prints "Yes" if it is recognized by one of the NFAs and "No" otherwise.

ALGORITHM:

1. Start
2. Design the NFA (N) to recognize Identifiers, Constants, and Operators
3. Read the input string w give it as input to the NFA
4. NFA processes the input and outputs “Yes” if w b Є L(N), “No” otherwise
5. Display the output
6. End

PROGRAM:

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

int isIdentifier(char *token) {
    if (!isalpha(token[0])) return 0;

    for (int i = 1; token[i] != '\0'; i++) {
        if (!isalnum(token[i]))
            return 0;
    }
    return 1;
}

int isConstant(char *token) {
    for (int i = 0; token[i] != '\0'; i++) {
        if (!isdigit(token[i]))
            return 0;
    }
    return 1;
}
```

```
int isOperator(char *token) {
    const char *operators[] = {
        "+", "-", "*", "/", "=", "==", "!=", "<", ">", "<=", ">="
    };
    int count = sizeof(operators) / sizeof(operators[0]);

    for (int i = 0; i < count; i++) {
        if (strcmp(token, operators[i]) == 0)
            return 1;
    }
    return 0;
}

int main() {
    char token[100];

    printf("Enter the Identifier input: ");
    scanf("%s", token);

    if (isIdentifier(token) || isConstant(token) || isOperator(token)) {
        printf("Yes\n");
    } else {
        printf("No\n");
    }

    return 0;
}
```

INPUT:

Input: Enter the Identifier input: sum

Output: Yes

Input: Enter a Constant input: 4567

Output: Yes

Input: Enter an operator input: +

Output: Yes

OUTPUT:

```
YES

YES

YES
```

RESULT:

QUESTIONS:

1. What is a Preprocessor and what is its role in compilation?
2. Which language is both compiled and interpreted?
3. List out the languages that are interpreted? 4. Explain the working of a NFA?
5. When do you prefer to design an NFA to DFA?

PRACTICAL 3

AIM

To Write a C program to develop a lexical analyzer(without flex) to recognize few Keywords, words, numbers in C

INTRODUCTION:

C program that acts as a simple lexical analyzer to recognize:

- Keywords in C (like int, if, while, etc.)
- Identifiers (words that are not keywords but follow variable naming rules)
- Numbers (integer constants)

ALGORITHM:

1. Write a array string/words for keywords(if,else)
2. Write a new function to identify the keyword by comparing the input
3. Write a new function to identify the numbers
4. w

PROGRAM:

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

#define MAX 100

// A few common C keywords
const char *keywords[] = { "int", "float", "char", "if", "else", "while",
"return" };

int keywordCount = sizeof(keywords) / sizeof(keywords[0]);

int isKeyword(char *str) {
    for (int i = 0; i < keywordCount; i++) {
        if (strcmp(str, keywords[i]) == 0)
            return 1;
    }
    return 0;
}

int isNumber(char *str) {
    for (int i = 0; str[i] != '\0'; i++) {
        if (!isdigit(str[i]))
```

```
            return 0;
    }
    return 1;
}

int isIdentifier(char *str) {
    if (!isalpha(str[0]) && str[0] != '_')
        return 0;
    for (int i = 1; str[i] != '\0'; i++) {
        if (!isalnum(str[i]) && str[i] != '_')
            return 0;
    }
    return 1;
}

void analyze(char *input) {
    char token[MAX];
    int i = 0, j = 0;

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

        // Word (could be keyword or identifier)
        if (isalpha(input[i]) || input[i] == '_') {
            j = 0;
            while (isalnum(input[i]) || input[i] == '_') {
                token[j++] = input[i++];
            }
            token[j] = '\0';

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

        // Number
        else if (isdigit(input[i])) {
            j = 0;
            while (isdigit(input[i])) {
                token[j++] = input[i++];
            }
            token[j] = '\0';
            printf("Number     : %s\n", token);
        }

        // Skip unrecognized symbols
        else {
            i++;
        }
    }
}

int main() {
    char input[MAX];

    printf("Enter a line of C code:\n");
    fgets(input, MAX, stdin);

    printf("\nLexical Analysis Output:\n");
    analyze(input);

    return 0;
}
```

INPUT:

```
int x = 11;
```

OUTPUT:

```
Keyword   : int

Identifier : x

Number    : 100
```

RESULT:

QUESTIONS:

PRACTICAL 4

AIM:

To Write a program to recognize few Keywords, words, numbers for c using flex

INTRODUCTION:

ALGORITHM:

Input : LEX specification files for the token

Output : Produces the source code for the Lexical Analyzer with the name lex.yy.c
and displays the tokens from an input file.

1. Start
2. Open a file in text editor
3. Create a Lex specifications file to accept keywords, identifiers, constants, operators and relational operators in the following format.
   a) %{

      Definition of constant /header
      files %}

   b) Regular
      Expressions %%

      Transition
      rules %%

   c) Auxiliary Procedure (main( ) function)
4. Save file with **.l** extension e.g. **mylex.l**
5. Call lex tool on the terminal e.g. [root@localhost]# lex mylex.l. This lex tool will convert

   ".l" file into ".c" language code file i.e., **lex.yy.c**
6. Compile the file lex.yy.c using C / C++ compiler. e.g. **gcc lex.yy.c**. After compilation the file lex.yy.c, the output file is in **a.out**
7. Run the file a.out giving an input(text/file) e.g. **./a.out**.
8. Upon processing, the sequence of tokens will be displayed as output.
9. Stop

PROGRAM:

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

%%
"int"        { printf("Keyword     : %s\n", yytext); }
"float"      { printf("Keyword     : %s\n", yytext); }
"if"         { printf("Keyword     : %s\n", yytext); }
"while"      { printf("Keyword     : %s\n", yytext); }
"return"     { printf("Keyword     : %s\n", yytext); }

[0-9]+       { printf("Number      : %s\n", yytext); }

[a-zA-Z_][a-zA-Z0-9_]* { printf("Identifier  : %s\n", yytext); }

[ \t\n]      ; // Ignore whitespace

.            ; // Ignore everything else
%%

int main() {

    printf("Enter C code (Ctrl+D to end input):\n");
    yylex();

      return 0;
}

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

INPUT:

In command prompt

```
flax myflax.l

ggcc lex.yy.c

run a.exe
```

```
int x = 100;
```

```
if (x > 10) return x;
```

OUTPUT:

```
Keyword    : int

Identifier  : x

Number     : 100

Keyword    : if

Identifier  : x

Number     : 10

Keyword    : return

Identifier  : x
```

RESULT:

QUESTIONS:

1. What is are the functions of a Scanner?

2. What is Token?

3. What is lexeme, Pattern?

4. What is purpose of Lex?

5. What are the other tools used in Lexical Analysis?

PRACTICAL 5

AIM:

To write a program for implementing a Lexical analyser using LEX tool.

INTRODUCTION:

ALGORITHM:

1. Read the input string.
2. Check whether the string is identifier/ keyword /symbol by using the rules of identifier and keywords using LEX Tool

PROGRAM:

```
%{
/* program to recognize a c program */ int COMMENT=0;
%}


identifier [a-zA-Z][a-zA-Z0-9]*

%%

#.* { printf("\n %s is a PREPROCESSOR DIRECTIVE", yytext);}
int | float |
char | double |
while | for |
do | if |
break | continue |
void | switch |
case | long
| struct | const |
typedef | return |
else |
goto  { printf("\n \t %s is a KEYWORD", yytext);}

"/*" {COMMENT = 1;}


"*/" {COMMENT = 0;}

{identifier}\( {if(!COMMENT) printf("\n\n FUNCTION\n\t %s", yytext); }
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}

\} {if(!COMMENT) printf("\n BLOCK ENDS");}


{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s
IDENTIFIER",yytext);}

\".*\" {if(!COMMENT) printf("\n\t%s is a STRING",yytext);}


[0-9]+ {if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}

\)(\;)? {if(!COMMENT) printf("\n\t");ECHO;printf("\n");}


\(    ECHO;

=     {if(!COMMENT)printf("\n\t%s is an ASSIGNMENT OPERATOR",yytext);}


\<= | \>= | \< | == |
\>    {if(!COMMENT) printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%


int main(int argc,char **argv) {
      if (argc > 1) {
            FILE *file;
            file = fopen(argv[1],"r");
            if(!file)
            {
                  printf("could not open %s \n",argv[1]); exit(0);
            }
            yyin = file;
      }
      yylex();
      printf("\n\n");
      return 0;
}
int yywrap() {
      return 0;
}
```

INPUT:

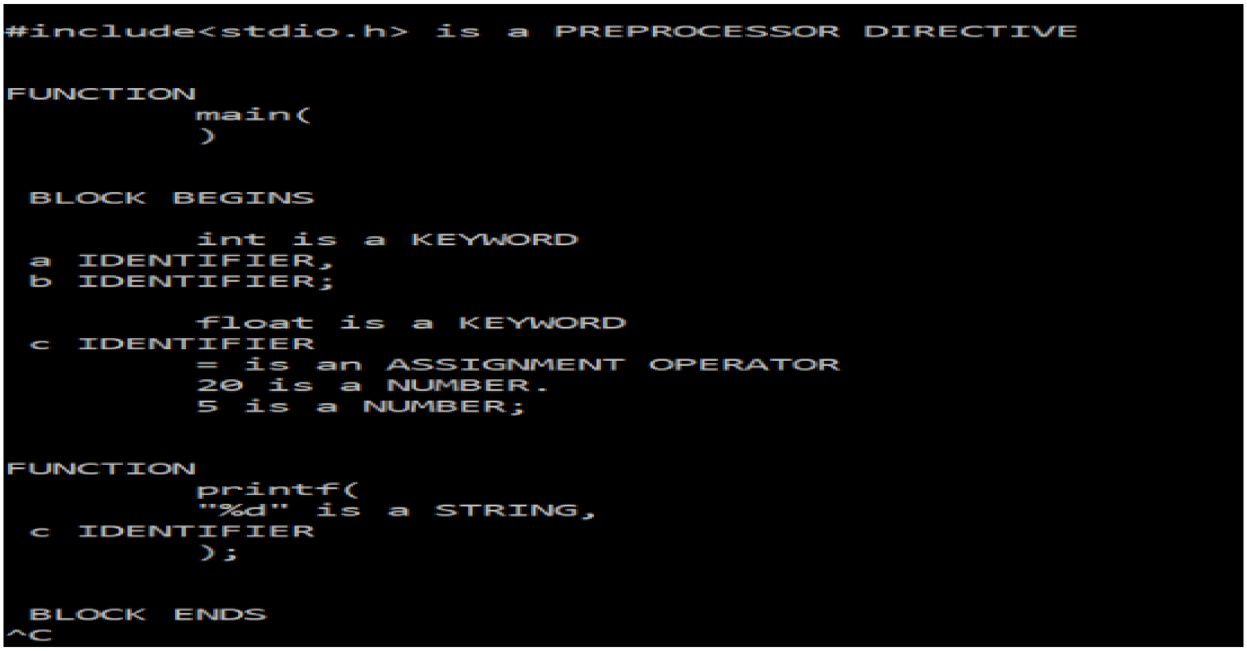
input.txt file

```
/*comment line*/

#include<stdio.h> main()

{

        int a,b; a=20;

        printf("%d",a);

}
```

Compilation Commands:

```
D:/lab/>flex lexprogram.l

D:/lab/>gcc lex.yy.c

D:/lab/>a.exe input.txt
```

OUTPUT:



```
#include<stdio.h> is a PREPROCESSOR DIRECTIVE


FUNCTION
        main(
        )


 BLOCK BEGINS

        int is a KEYWORD
 a IDENTIFIER,
 b IDENTIFIER;

        float is a KEYWORD
 c IDENTIFIER
        = is an ASSIGNMENT OPERATOR
        20 is a NUMBER.
        5 is a NUMBER;


FUNCTION
        printf(
        "%d" is a STRING,
 c IDENTIFIER
        );


 BLOCK ENDS
^C
```

RESULT:

QUESTIONS:

- What is Lex tool?
- What is yylex() function?

PRACTICAL 6

AIM:

To write a program for recognizing a valid arithmetic expression that uses operator +, –, * and / using YACC (Yet Another Compiler-Compiler).

ALGORITHM:

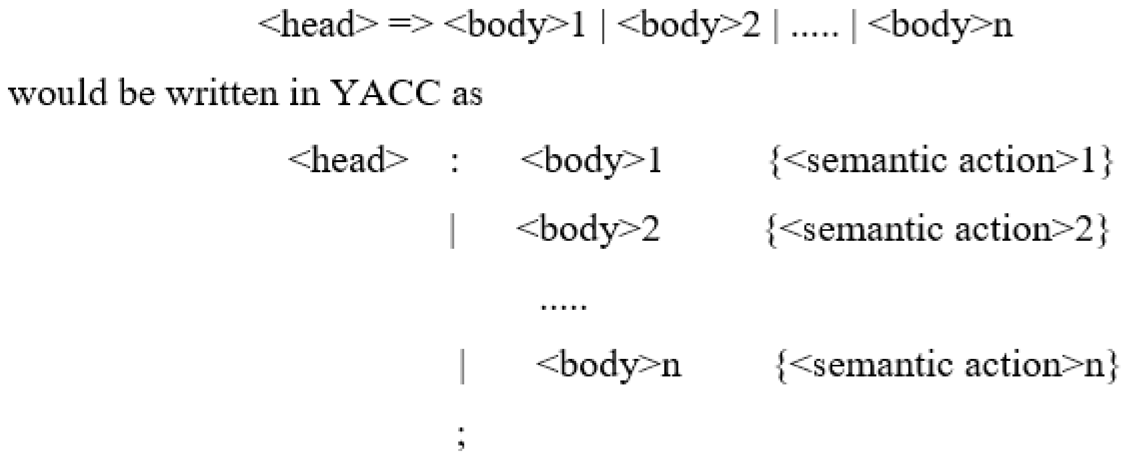
Declarations Part

This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which is delimited by %{ and %}. Any temporary variable used by the second and third sections will be kept in this

Translation Rule Part

After the first %% pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action.

A set of productions:



```
<head> => <body>1 | <body>2 | ..... | <body>n
```

would be written in YACC as

```
<head>   :   <body>1      {<semantic action>1}
         |   <body>2      {<semantic action>2}
               .....
         |    <body>n      {<semantic action>n}
         ;
```

1. Open any editor, type the needed yacc program and save it with extension .y (example as arithexp.y).

2. Process the yacc grammar fileusingthe -d optional flag(which informs the yacccommand to create a file that defines the tokens used in addition to the C language source code):

bison -d arithexp.y

Following two files will be get generated.

a. arithexp.tab.c -- The C language source file that the yacc command created for the parser
b. **arithexp.tab.h** -- A header file containing define statements for the tokens used by the parser

3. Compile the c program **arithexp.tab.c**

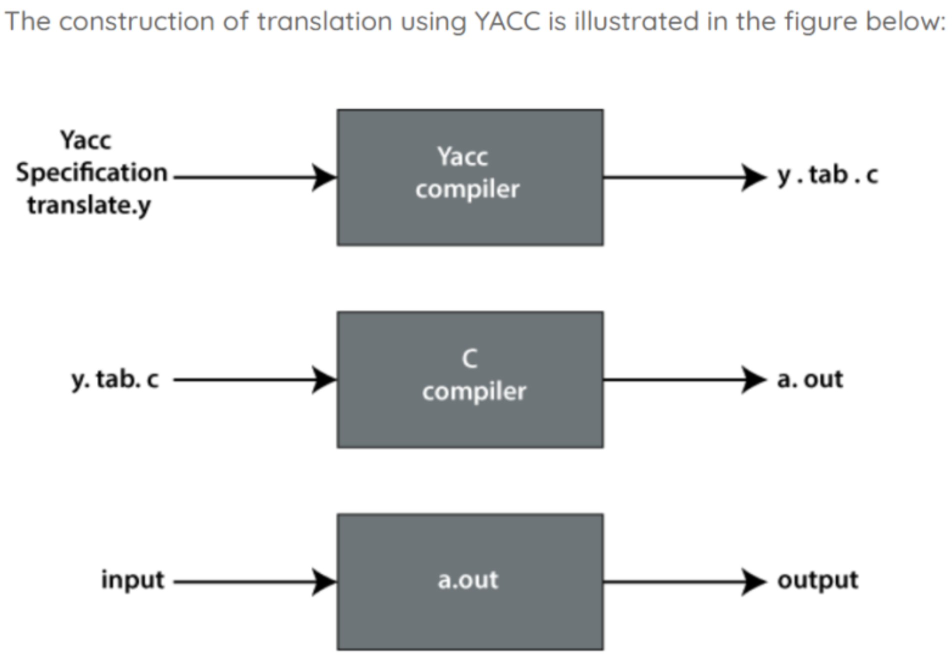
gcc **arithexp.tab.c (Note:** Warning will be generated, Please ignore it**)** Following file will be generated : **a.exe**

4. Run the executable file : a.exe

PROGRAM:

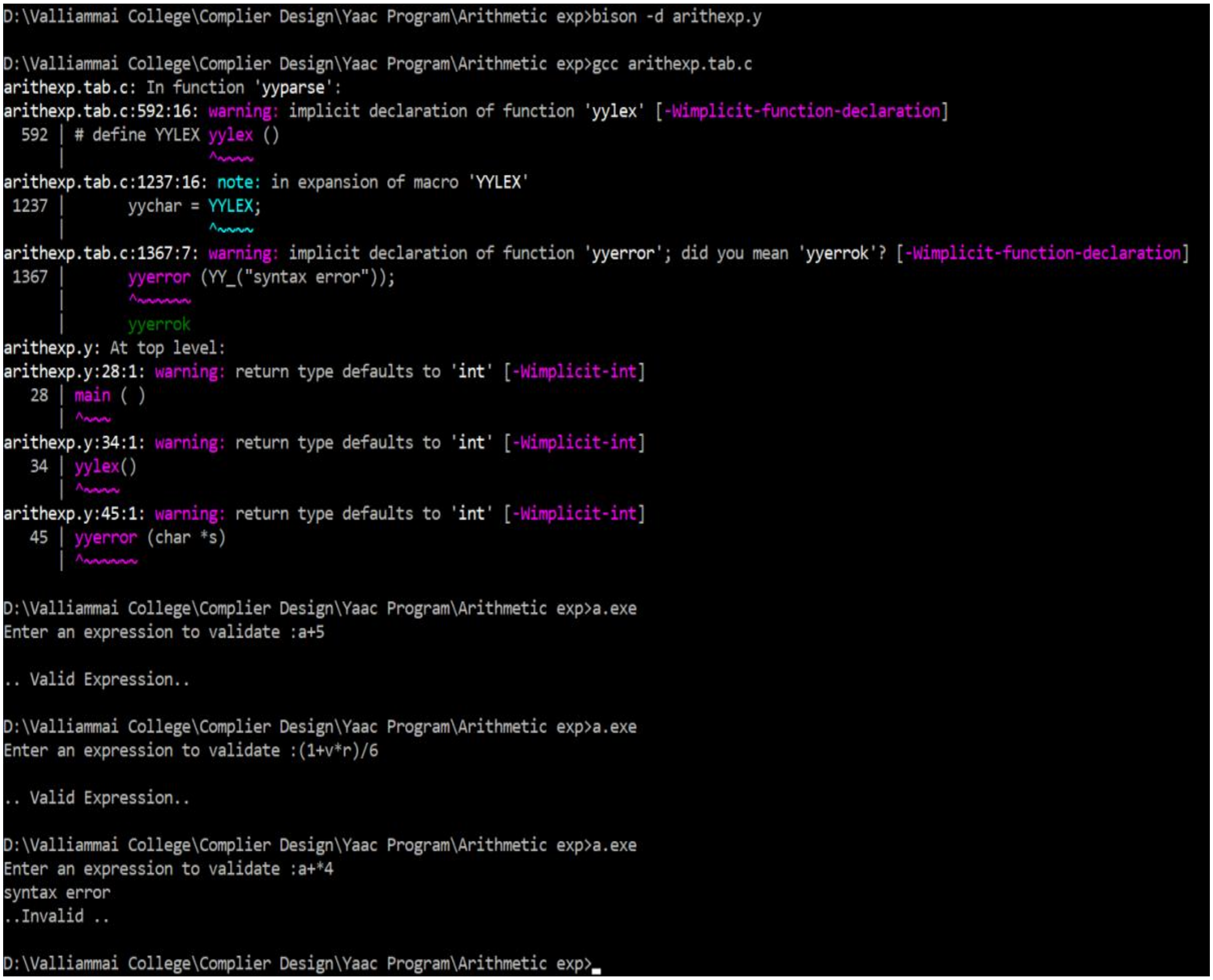
```
%{    #include <stdio.h> #include <ctype.h> #include <stdlib.h>
%}

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

%%


Stmt : Stmt '\n'

{ printf ("\n.. Valid Expression.. \n"); exit(0);
}
|     expr
|     error '\n'
{ printf ("\n..Invalid ..\n"); exit(0);
} ;

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

%%

main ( ) {
      printf ("Enter an expression to validate :" );
      yyparse( );
}
yylex() {
      int ch;
      while ( ( ch = getchar() ) == ' ' );
      if ( isdigit(ch) )
      return num; // return token num
      if ( isalpha(ch) )
            return let; // return token let
      return ch;
}
yyerror (char *s) {

      printf ( "%s", s );
}
```

INPUT:



The construction of translation using YACC is illustrated in the figure below:

OUTPUT:



```
D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>bison -d arithexp.y

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>gcc arithexp.tab.c
arithexp.tab.c: In function 'yyparse':
arithexp.tab.c:592:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
  592 | # define YYLEX yylex ()
      |                ^~~~~
arithexp.tab.c:1237:16: note: in expansion of macro 'YYLEX'
 1237 |       yychar = YYLEX;
      |                ^~~~~
arithexp.tab.c:1367:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
 1367 |       yyerror (YY_("syntax error"));
      |       ^~~~~~~
      |       yyerrok
arithexp.y: At top level:
arithexp.y:28:1: warning: return type defaults to 'int' [-Wimplicit-int]
   28 | main ( )
      | ^~~~
arithexp.y:34:1: warning: return type defaults to 'int' [-Wimplicit-int]
   34 | yylex()
      | ^~~~~
arithexp.y:45:1: warning: return type defaults to 'int' [-Wimplicit-int]
   45 | yyerror (char *s)
      | ^~~~~~~

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>a.exe
Enter an expression to validate :a+5

.. Valid Expression..

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>a.exe
Enter an expression to validate :(1+v*r)/6

.. Valid Expression..

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>a.exe
Enter an expression to validate :a+*4
syntax error
..Invalid ..

D:\Valliammai College\Complier Design\Yaac Program\Arithmetic exp>_
```

RESULT:

QUESTIONS:

- What is Yaac? Why do use Yaac?

- What is difference between Yaac and flex

PRACTICAL 7

AIM:

To write a C program to check whether the type of all variables in an expression are valid or not.

INTRODUCTION:

1. Open any editor, type the needed C program and save as **typecheck.c**

2. Compile C language source **gcc typecheck.c**

Following file will be generated : **a.exe**

3. Run the executable file : **a.exe**

ALGORITHM:

Step1: Track the global scope type information (e.g. classes and their members)

Step2: Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.

Step3: If type found correct, do the operation

Step4: Type mismatches, semantic error will be notified

PROGRAM:

```
#include<stdio.h>

void main()

{

  int n, i, k, 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(" %c", & vari[i]);
    printf("Enter the variable-type[%d](float-f,int-i):", i);
    scanf(" %c", & typ[i]);
    if (typ[i] == 'f') flag = 1;
  }

  printf("Enter the Expression(end with $):");
  i = 0;
  getchar();
  while ((c = getchar()) != '$') {
    b[i] = c;
    i++;
  }
  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 (flag == 1) {
        if (typ[i] == 'f') {
          printf("\nthe datatype is correctly defined..!\n");
          break;
        } else {
          printf("Identifier %c must be a float type..!\n", vari[i]);
          break;
        }
      } else {
        printf("\nthe datatype is correctly defined..!\n");
        break;
      }
    }
  }
}
```

INPUT:

OUTPUT:

RESULT:

QUESTIONS:

- What is expression?
- What is typecheck?
- What is abstract syntax tree?
- What is quadruple?
- What is the difference between Triples and Indirect Triple?
- State different forms of Three address statements.
- What are different intermediate code forms?

PRACTICAL 08

AIM:

To write a C Program to Generate Machine Code from the Abstract Syntax Tree using the specified machine instruction formats.

INTRODUCTION:

Generate the Machine code output given instruction formats.

ALGORITHM:

PROGRAM:

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

int label[20];

int no = 0;
int main() {
   FILE * fp1, * fp2;
   char fname[10], op[10], ch;
   char operand1[8], operand2[8], result[8];
   int i = 0, j = 0;
   printf("\n Enter filename of the intermediate code");
   scanf("%s", & fname);
   fp1 = fopen(fname, "r");
   fp2 = fopen("target.txt", "w");
   if (fp1 == NULL || fp2 == NULL) {
      printf("\n Error opening the file");
      exit(0);
   }
   while (!feof(fp1)) {
      fprintf(fp2, "\n");
      fscanf(fp1, "%s", op);
      i++;
      if (check_label(i))
         fprintf(fp2, "\nlabel#%d", i);
      if (strcmp(op, "print") == 0) {
         fscanf(fp1, "%s", result);
         fprintf(fp2, "\n\t OUT %s", result);
      }
      if (strcmp(op, "goto") == 0) {
         fscanf(fp1, "%s %s", operand1, operand2);
         fprintf(fp2, "\n\t JMP %s,label#%s", operand1, operand2);
         label[no++] = atoi(operand2);
      }
      if (strcmp(op, "[]=") == 0) {
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n\t STORE %s[%s],%s", operand1, operand2, result);
      }
      if (strcmp(op, "uminus") == 0) {
         fscanf(fp1, "%s %s", operand1, result);
         fprintf(fp2, "\n\t LOAD -%s,R1", operand1);
         fprintf(fp2, "\n\t STORE R1,%s", result);
      }
      switch (op[0]) {
      case '*':
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t MUL R1,R0");
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;
      case '+':
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t ADD R1,R0");
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;
      case '-':
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t SUB R1,R0");
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;
      case '/':
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t DIV R1,R0");
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;
      case '%':
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n \t LOAD %s,R1", operand2);
         fprintf(fp2, "\n \t DIV R1,R0");
         fprintf(fp2, "\n \t STORE R0,%s", result);
         break;
      case '=':
         fscanf(fp1, "%s %s", operand1, result);
         fprintf(fp2, "\n\t STORE %s %s", operand1, result);
         break;
      case '>':
         j++;
```

```
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n\t JGT %s,label#%s", operand2, result);
         label[no++] = atoi(result);
         break;
      case '<':
         fscanf(fp1, "%s %s %s", operand1, operand2, result);
         fprintf(fp2, "\n \t LOAD %s,R0", operand1);
         fprintf(fp2, "\n\t JLT %s,label#%d", operand2, result);
         label[no++] = atoi(result);
         break;
      }
   }
   fclose(fp2);
   fclose(fp1);
   fp2 = fopen("target.txt", "r");
   if (fp2 == NULL) {
      printf("Error opening the file\n");
      exit(0);
   }
   do {
      ch = fgetc(fp2);
      printf("%c", ch);
   } while (ch != EOF);
   fclose(fp1);
   return 0;
}
int check_label(int k) {
   int i;
   for (i = 0; i < no; i++) {
      if (k == label[i]) return 1;
   }
   return 0;
}
```

INPUT:

In int.txt =

t1 2[]= a 0 1 []=a 1 2 []=a 2 3 *t1 6 t2 +a[2] t2 t3 -a[2] t1 t2 /t3 t2 t2

uminus t2 t2 print t2

goto t2 t3 =t3 99 uminus 25 t2 *t2 t3 t3 uminus t1 t1 +t1 t3 t4 print t4

OUTPUT:

```
Enter filename of the intermediate code: int.txt


STORE  t1, 2
STORE  a[0],
1

STORE  a[1],
2

STORE  a[2],
3

LOAD  t1, R0
LOAD  6, R1
ADD  R1, R0
STORE  R0,
t3

LOAD  a[2],
R0

LOAD t2, R1
ADD  R1,R0
STORE
R0,t3

LOAD
a[t2],R0
LOAD  t1,R1
SUB  R1,R0
STORE
R0,t2

LOAD  t3,R0
LOAD t2,R1
DIV  R1,R0
STORE
R0,t2

LOAD  t2,R1
STORE
R1,t2

 LOAD  t2,R0
JGT  5,

label#11

Label#11: OUT
```

```
t2

JMP
t2,label#13

Label#13: STORE
t3,99

LOAD  25,R1

STORE
R1,t2
LOAD
t2,R0
LOAD
t3,R1

MUL
R1,R0
STORE
R0,t3
LOAD
t1,R1
STORE
R1,t1
LOAD
t1,R0
LOAD
t3,R1 ADD
R1,R0
STORE
R0,t4

 OUT  t4
```

RESULT:

QUESTIONS:

- What is instruction?
- What is abstract syntax Tree

PRACTICAL 09

AIM:

To write a C Program to Generate Machine Code given assembly code.

INTRODUCTION:

simplified version to demonstrate machine code generation using custom instruction formats for a mini-language (a subset of C). This works by:

- Recognizing basic C-like instructions (like a = b + c;)
- Mapping them to predefined machine instructions (in binary or hex)

ALGORITHM:

Simplified Assumptions:

- You define your own instruction set (machine format)
- Only support basic operations: MOV, ADD, SUB
- Operands are registers like R1, R2, R3, etc.

Example Instruction Format (hypothetical 16-bit):

Opcode (4 bits)	Dest (4 bits)	Src1 (4 bits)	Src2 (4 bits)
MOV = 0001	dest	src	0000
ADD = 0010	dest	src1	src2
SUB = 0011	dest	src1	src2

PROGRAM:

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

int getRegisterCode(char *reg) {
    if (reg[0] != 'R') return -1;
    return atoi(&reg[1]);
}
```

```
int getOpcode(char *op) {
    if (strcmp(op, "MOV") == 0) return 1;
    if (strcmp(op, "ADD") == 0) return 2;
    if (strcmp(op, "SUB") == 0) return 3;
    return -1;
}

void toBinary(int value, int bits, char *output) {
    output[bits] = '\0';
    for (int i = bits - 1; i >= 0; i--) {
        output[i] = (value % 2) + '0';
        value /= 2;
    }
}

void generateMachineCode(char *line) {
    char opcode[10], reg1[10], reg2[10], reg3[10];
    int code, r1, r2, r3;

    int count = sscanf(line, "%s %[^,], %[^,], %s", opcode, reg1, reg2,
reg3);

    code = getOpcode(opcode);
    if (code == -1) {
        printf("Unknown instruction: %s\n", opcode);
        return;
    }

    char op_bin[5], r1_bin[5], r2_bin[5], r3_bin[5];
    toBinary(code, 4, op_bin);
    toBinary(getRegisterCode(reg1), 4, r1_bin);
    toBinary(getRegisterCode(reg2), 4, r2_bin);

    if (strcmp(opcode, "MOV") == 0 && count == 3) {
        strcpy(r3_bin, "0000");
    } else if ((strcmp(opcode, "ADD") == 0 || strcmp(opcode, "SUB") == 0)
&& count == 4) {
        toBinary(getRegisterCode(reg3), 4, r3_bin);
    } else {
        printf("Invalid format for %s\n", opcode);
        return;
    }

    printf("Instruction: %s\n", line);
    printf("Machine Code: %s %s %s %s\n\n", op_bin, r1_bin, r2_bin,
r3_bin);
}

int main() {
    char line[100];

    printf("Enter instructions (one per line, type 'end' to stop):\n");

    while (1) {
        printf(">> ");
        fgets(line, sizeof(line), stdin);
        line[strcspn(line, "\n")] = '\0';  // Remove newline

        if (strcmp(line, "end") == 0)
            break;

        generateMachineCode(line);
    }

    return 0;
}
```

INPUT:

MOV R1, R2

ADD R3, R1, R2

SUB R4, R3, R1

OUTPUT:

Instruction: MOV R1, R2

Machine Code: 0001 0001 0010 0000

Instruction: ADD R3, R1, R2

Machine Code: 0010 0011 0001 0010

Instruction: SUB R4, R3, R1

Machine Code: 0011 0100 0011 0001

RESULT:

QUESTIONS:

- What is instruction ? what is instruction format?
- What is machine code?
- What is source and destination register or memory?