INDEX

Name : Gauri Sanjay Mahakal Class Roll No.: 04

Branch : CSE Year / Semester : 3rd year 6th Sem

Name of Subject : Compiler Design

S.No.	Name of Experiment	Page No.	Date	Remarks
01	write a Lex Program to Reg Recognise valid strings using Regular expression: (a/b)*abb	1 - 4	21/01/23	
02	write a Lex program to recognise tokens 1) identifier 2) Integer Constant 3) Real Constants	5 - 9	28/01/23	
03	Source code is a text file. constants used in the program are strings. write a program to read integer constant as a string and convert it represented in Decimal, octal & Hexadecimal. it may be positive or Negative	10 - 14	04/02/23	
04	write a Program to eliminate left Recursive from the grammar.	15 - 19	11/02/23	
05	Design recursive descent parser for grammar E → +EE|-EE|a|b	20 - 22	18/02/23	
06	Design a Program for basic calculator using YACC OR BISON	23 - 25	11/03/23	

INDEX

Name : Gauri S. Mahakal Class Roll No.: 04

Branch : CSE Year / Semester : 3rd year 6th sem

Name of Subject : Compiler Design .

S.No.	Name of Experiment	Page No.	Date	Remarks
07	Design a program to evaluate postfix expressions	26-29	18/03/23	
08	write a c Program for a selection sort generate equivalent 3-address code	30-33	1/04/23	
09	write a c Program to optimize the 3-address code generated.	34-36	1/4/23	
10	write a c program for selection sort & generates a object code for three Address code.	37-38	1/4/23	

Practical No : 1

Aim : Write a Lex Program to recognize valid strings using Regular Expression : (a|b)* abb

Test case :

Valid Strings : ababbb, abb, bbbabb

Invalid strings : baba, bba, bbb, ababba.

Tools : Software :- 1. Flex Software (Compiler)

specifications :- windows 10, 64 bit, exe

Hardware :- 2. Computer system

specifications :- Windows 10, 500 GiB HDD / 250 GiB SSD i3 processor.

Theory :-

Regular Expresion :-

The lexical analyzer needs to scan and identify only a finite set of valid string / token / lexeme that belong to the language in hand. Regular Expression is an important notation for specifying pattern Each pattern matches a set of strings. so regular expression serve as a names for a set of strings

Regular expression can also be described as a sequence of pattern that define a string. Regular expression are used to match character combinations in strings. string Searching algorithm used this pattern to find the operations on a string.

ELLENT

Operations :-

The various operations on languages are

1) Union of two languages L and M is written as
LUM = { s/s is in L or s is in M }

2) Concatenation of two language L and M is written as
LM = { st/s is in L and t is in M }

3] Kleane closure of a language L is written as L^*
L^* = zero or more occurence of language L.

flex software is compiler which is used to compile lex code. we will apply some regular expression to fulfill given condition

Program:-

```
%{
#include<stdio.h>
%}
%%%
(a|b)* abb    printf("valid string : %s\n", yytext);
              printf("Invalid string : %s\n", yytext);
%%%
int main(){
  yylex();
  return 0;
}
```

Error and Remedial action :

1) lex programs must follow a specific syntax . and any deviation from that syntax can result in a syntax Error.

Action : Rewrite the code with correct syntax

2) could not Locate lex . dak
Action : reinstalled the flex software.

Result :-

Conclusion :- By executing this program The lex program is recognize given string by using regular expression . Implemented successfully and all test cases varified.

Cource outcome attained and mapping with Program Outcome :

Cource outcome	Program outcome
	2 . Problem analysis
CO-1 : Generate scanner and parser from formal specifications	3 . Design Development of solution.
	4 . Modern tool usage .

1. 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

2. 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.

3. 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.

Result :-

```
abababb
Valid string: abababb
abb
Valid string: abb
bbbabb
Valid string: bbbabb
baba
bba
bbb
ababba
```

Practical No 2

Aim:- Write a LEX Program to recognize tokens
1) Identifiers
2) Integer Constant (Decimal, octal, Hexadecimal)
3) Real Constants (Floating point and Exponential Format)

Test Cases :

a. Integer Constant Ex. +15, -5, 0123, 0X1A
b. Floating point format Ex. +0.5, 236.0, .567, -26.8
c. Exponential format Ex. 1.6e-19, -0.5E+2, +1.7E4, .26e-7

Tools :

Sr	Tools	specification	Quantity
1	Lex Compiler	- Windows10, 64 bit	-
2	Computer System	- Window10, 500GB HDD /250GB SSD i3 processor	1

Theory :- a) Regular Expression for Decimal constant

Rules : 1. must have at least one digit
2. may be +ve or -ve
3. No spaces, comma, period (.) are allowed.

Regular Expression :- (+|-)?(Digit)+

b) Regular Expression for Real Constant (Floating point Notation)

1. Must have atleast one digit 2. may be +ve or -ve
3. Must contain decimal point 4. atleast are digit after decimal point.
5) No (-) (,) (.)

Regular Expression :- (+|-)?(Digit)*.(Digit)+

In lexical analysis. the source code is scanned and broken down into these individual tokens, which are then passed on the next stage of the compilation. or interpretation process.
Some of them can be described as follows :-

1. Identifier :- ^[a-z A-Z_] [a-z0 A-z 0-9_]* $

Here,
'^' :- matches the beginning of the string
'[a-z A-Z_]' :- matches any uppercase letter lowercase letter, or underscore.
'[a-z A-z, 0-9_]' :- matches zero or more occurences of any uppercase letter. lowercase letter, digit or underscore.
'$' - matches the end of string.

2. Integer constant :- Regular expressions for decimal constant can be -
a. must have at least one digit.
b. may be have +ve or -ve
c. No space and commas, period (.) are allowed
(+/-)? (Digit)+

3. Real Constant :- This will includes two categories
i) floating point :- must ~~const~~ contain decimal point
ii) Exponential point :- must contain mantissa and Exponent Regular expression for real constant is seperated by 'E' or 'e'
exponent [eE][+-]? {digit}+
{digit}+ "." {digit}* {exponent}? | {digit}+ {exponent}

Page No.
Date :

Program :

```
%  option noyywrap
% }
#include<stdio.h>
%}
digit                [0-9]
lex_digit            [0-9a-fA-F]
integer              {digit}+ | {alpha}
octal                0[0-7]*
hexadecimal          0[xX]{hex_digit}+
Float_num            {digit}+\.{digit}+
exp                  [eE]
alpha                [a-zA-Z]
%%
{integer} {printf("Integer Constant : %s\n", yytext);}
{octal} {printf("Octal Constant : %s\n", yytext);}
{hexadecimal} {printf("Hexadecimal constant : %s\n", yytext)}
{Float_num} {printf("Floating point Constant :%s\n", yytext);}
.[a-zA-Z_][a-zA-Z0-9_]* {printf("valid identifier");}
[^\=0-9][0-9]*?.[0-9]+{exp}[^\-0-9][0-9]*
    {printf("Exponential Constant");}
%%
int main(){
    yylex();
  return 0;
}
```

Error(s) and Remedial action :-

Error :- Integer underflow :- occurs when input Program contains the integer constant i.e to large in absolute value to be represented as decimal integer.

Remedial Action :- We can check length of input string before connecting it to an integer and raise an error if length for given data types.

Error :- Syntactical error [Missing symbol or operator]
Remedial action :- Rewrite code by correcting the error with right syntax.

Result :-

Attached to the back side of this page.

Conclusion :- we successfully able to learnt the concept of lexical analysis like how it breaks the source code into different types of tokens and also implemented lex program to recognize tokens

1] Identifier
2] Real Constant
3) Integer constant

Course outcome attained and mapping with program outcome :-

Course outcome	Program outcome
CO-1 : Generate scanner and parsar from formal specifications	2. Problem Analysis 3. Design / Development of Solutions. 5. Modern tool usage.

3. Design / Development of solution :- Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate considerations for the public health and safety and the cultural, societal and environmental considerations.

Result :-

```
C:\Lex>flex pr23.l

C:\Lex>gcc lex.yy.c -o pr23.exe

C:\Lex>pr23.exe
2.3e3
Exponential constant
2.3E3
Exponential constant
1.6e-19
Exponential constant
-0.5E2
Exponential constant
1.7E4
Exponential constant
```

```
C:\Lex>flex pr23.l

C:\Lex>gcc lex.yy.c -o pr23.exe

C:\Lex>pr23.exe
+15
Integer constant: 15

-5
Integer constant: 5

0x1A
Hexadecimal constant: 0x1A

o123
Octal constant: o123

0.5
Floating point constant: 0.5

236.0
Floating point constant: 236.0

-26.89
Floating point constant: 26.89

1.6E-19
Exponential constant
```

```
C:\Lex>flex pr23.l

C:\Lex>gcc lex.yy.c -o pr23.exe

C:\Lex>pr23.exe
gfg
Valid identifier
23fg
Invalid identifier
&*gdg
Invalid identifier
```

Practical No : 3

Aim : Source code is text file. constants used in program are strings. Write a program to read integer constants as a string and convert it to decimal number. Integer constants are represented in Decimal, octal and hexadecimal. It may few positive or negative.

Test Cases :

a. Decimal constant : +15, -5, 123
b. octal Constant : +0123, 0117, -0777
c. ~~Hexa~~ Exponential
format Ex :- 1.6e-19, -0.5E+2, +1.7E4, .26e-7

Tools :- Software : Codblock editor

Hardware :- Computer System
Specifications :- Windows 10, 500 GB HDD / 250 GB SSD, i3 processor

Theory :- Integer constants :-

An integer constants can be a decimal, octal, or hexadecimal constant. Constants refer to fixed values that the program may not alter during it's execution. ~~The~~ constants can be of any of the basic data types like an integer constant, a floating constant, a character constant or a string literal.

In this practical we used ~~Deci~~ Integer Constants

EXCELLENT

are represented in Decimal, octal and hexadecimal it can be positive integer or Not negative.

Octal :- Integer constant represented in Decimal
octal : A prefix specifies the base or radix : 0 for octal representation. and it specifies a base : 8 for octal.

Hexadecimal :- Integer constant represented in Decimal
Hexadecimal constant :- A prefix specifies a base or radix : "0x" OR "0X" for hexadecimal constants. and Base for Hexadecimal is 16.

Default Integer constants concides (base 10) Decimal.

Program :-

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

int main()
{
char num[20];
int decimal;
Printf("Enter an integer constant (in
decimal, octal, or hexadecimal format): ");
scanf("%s", num);

int is_negative = 0;
if(num[0]=='-')
{
  is_negative = 1;
memmove(num, num+1, strlen(num));
}
int base = 10;
if(num[0] == '0')}
 if(num[1]=='x' || num[1]=='X')
  {
    base = 16;
memmove(num, num+2, strlen(num));
 }
else}
    base = 8;
  }
}
```

```
decimal = 0;
int i = 0;
        while(num[i] != '\0')
        {
          int digit;
          if (isdigit(num[i])) {
          digit = num[i] - '0';
          }
        else if( isalpha(num[i]))
          {
          digit = toupper(num[i]) - 'A' + 10;
          }
        else
          { printf(" Invalid character : %c\n",
                    num[i]);
          return 1;
          }
    if(digit >= base)
     {    printf(" Invalid digit for base %d : %c\n",
                    base, num[i]);
          return 1;
        }
    decimal = decimal * base + digit;
          i++;
     }
if( is_negative)
{  decimal = - decimal;
   }
printf(" The decimal equivalent of %s is %d\n",
     num, decimal);
   }  return 0;
```

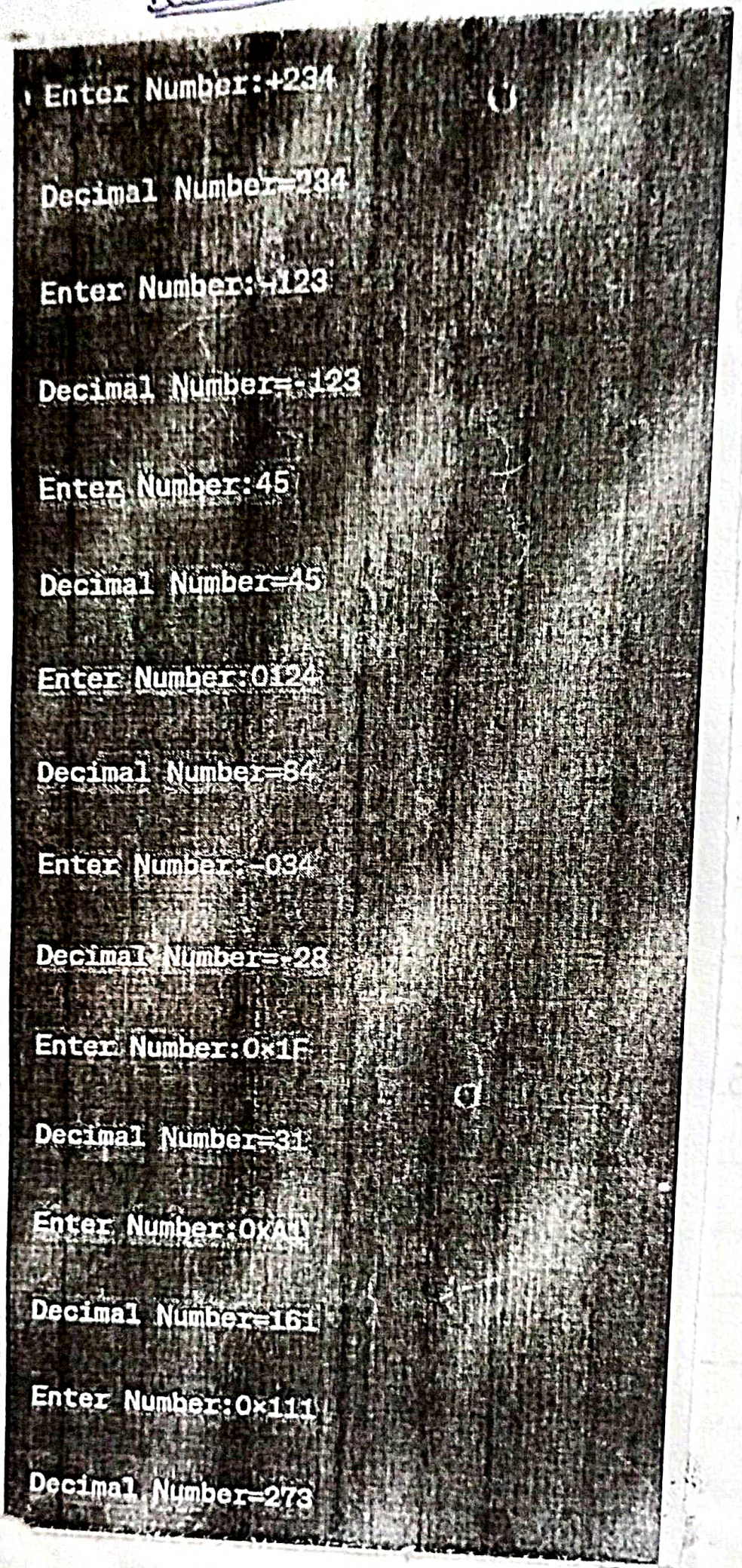
Errors and Remedials Action

- **Error** :- lexical error occurs when input program contains characters that do not match any of defined Rules. lexer will not able to identify input as valid integer

- **Remedial** :- Added printf statement.

Result : Result is attached to back side of this page.

Result :-


Enter Number:+234
Decimal Number=234
Enter Number:-123
Decimal Number=-123
Enter Number:45
Decimal Number=45
Enter Number:0124
Decimal Number=84
Enter Number:-034
Decimal Number=-28
Enter Number:0x1F
Decimal Number=31
Enter Number:0xA1
Decimal Number=161
Enter Number:0x111
Decimal Number=273

Practical No 4

Aim :- Write a program to eliminate left recursion from the grammar.

Test case :-
1. E : E+E | E-E | id
2. S : (L) | a
3. L : L,S | S

Tools :-

1. lex compiler C language Compiler)
   specifications :- any GDB compiler (MinGW)

2. Computer system
   specifications :- windows 10, 150 GB HDD / 8GB RAM, i3 processor.
   Quantity : 1

Theory :-

A Grammar G(V, T, P, S) is left recursive if it has a production in the form

$$A \rightarrow A\alpha \mid \beta$$

The above grammar is left recursive because the left of production is occuring at a First position on the right side of production. It can elimindle left recursion by replacing a pair of production with

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

1. left Recursion can be eliminated by introducing new non-terminal A such that.

ELLENT

This process can also defined as elimination of direct left recursion.

$A \rightarrow A\alpha \mid \beta$	Removal of left Recursion →	$A \rightarrow \beta A'$ $A' \rightarrow \alpha A' \mid \epsilon$

left recursion Grammar

2. Eliminate indirect left recursion :-

step one describes a rules to eliminate direct left recurssion from a production. To elimindte left recursion from an entire grammar may be more difficult because of indirect left recursion for eg :-

$A \rightarrow B x y \mid x$

$B \rightarrow CD$

$C \rightarrow A \mid c$

$D \rightarrow d$

for example :-

$E \rightarrow E+T \mid T$

Compairing it with $A \rightarrow A\alpha \mid \beta$

$A = E \quad \alpha = +T \quad \beta = T$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$ $\qquad$ $E \rightarrow TE'$

$A' \rightarrow \alpha A' \mid \epsilon$ $\qquad$ $E' \rightarrow +TE' \mid \epsilon$

C compilee will compiled the code which used for eliminate left recursion.

CELLENT

Program :-

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

#define SIZE.20

int main()
{
 chas pro[SIZE], alpha[size], beta[SizE];
 int  non-terminal, i, j, index=3;
 printf(" Entet the production as E→E|A:");
  Scanf("%s", pro);
 non-terminal = pro[0];
 if(non_terminal == pro[index])
  {
   foe(;=++index. j=0; pro[i]!='|'; i++; j++)
    {
       alpha[j] = pro[i];
       if(PEO[i+1]==0){
          printf("This Grammar CAN'T BE
                  REDUCED \n");
                exit(0);
          }
    }
alpha[i] = '\0';
if(PEO[++i];=0)
 {
   foe (j=i, j=0; pro[j]; = '\0'; i++j++)}
          beta[i] = pro[j];
    }
```

```
        beta[i] = '\0';
    printf("\n Grammar without left Recursion :\n\n");
printf("%c -> %s %c'\n", non-terminal, beta,
                        non-terminal);
printf("%c' -> %s %c' | #\n", non-terminal, alpha,
                        non-terminal);
    }
  else
    printf("This Grammar CAN'T be REDUCED.\n");
}
else
    Printf("\n this grammar is not LEFT REDUCE RECURSION.\n");
}
```

Error and Remedial action :-

Error :- Buffers oweeflow, Program consumers max length of production rule and max no. of production rules are fixed.

Remedial action :- Program could dynamically allocate for rules and symbols. Extra space between the grammar may affect the output the given code.
To get rid ouss this. we can delete the extra space and rewrite.

Result : Result is attached to the back side of this page.

Conclusion :- By implementing this practical we learnt about the How to eliminate left recursion from the given Grammar. and understand the concept of elimination of left recursion from Grammar.

Course outcome attained and mapping with program outcome.

Cource outcome	Program outcome
CO-2: Generate top-down and Bottom-up parsing tables using predicative parsing SLR and LR parsing techniques.	PO-2 : Problem analysis PO-3 : Design / Development of solutions PO-1 - Engineering knowledge.

ELLENT

Result :-

```
Enter the productions: E->E+E|T
The productions after eliminating Left
    Recursion are:
E->+EE'
E'->TE'
E->ε

Enter the productions: S : ( L ) | a

 L : L , S | S
The productions after eliminating Left
    Recursion are:
S -> ( L )
    | a
 L -> ( L ) L'
    | a L'
L' -> , S L'
    | ε
```

Experiment No 5

Aim :- Design Recursive descent parser for the grammar $E \rightarrow +EE \mid -EE \mid a \mid b$

Tools :-

Sr No	Tools	specification	Qty
1	C compiler	TURBO, GCC, Dev	1
2	Computer system	windows, 4GB RAM 1TB HDD	1

Theory :- A recursive descent parser is a top-down parsing technique where a parser tree is built from top- and constructed down to leaves this type of parser starts with highest level of grammar & recursively expands non-terminals until a terminal symbol is reached.

- To implement this grammar using recursive descent parser, we need to follow steps to implement parser :-

1. Analyze grammar, identify terminals, non-terminals production rules
2. write function To Parse each non-terminal
3. use tokenizer to convert input string into list of tokens
4. pass list tokens as input, function should return whether string can be derived from grammar or not

Program :-

```
#include<stdio.h>
#include<stdlibool.h>
#include<string.h>
bool parser_E(char *input, int *index);
bool parser_EE(char *input, int *index);
bool parser_T(char *input, int *index);
}
if(input[*index]=='a'||input[*index]
==b'){}
(*index)++;
return true;
} else if(input[*index]=='*'||input[*in
-dex]=='-'){(*index)++;
return parse_EE(input, index)&&
parse-E(input, index);
}
else{return false;
bool parse_EE(char *input, int *index){
if(input[*index]=='+'||index=='-'){
(*index)++;
return parse_T(input index);}
else{return false;}

int main(){
char input[100];
int index=0;
printf("Enter an input string:\n");
scanf("%s", input);
bool success=parse_E(input, &index);
if(success&&index==strlen(input))
```

```
}
  Printf("The input string can be accepted using
        grammar");
    }
else{
    printf("The input string can not be accepted
        from grammar \n");
    }
return 0;
}
```

Error & Remedia action

1. Program assumes input string contains only symbols 'a' 'b' '+' '-' if contain any other symbol, program may produce error

Remedial Action : Program can be manipulated to check each symbol that is terminals.

Conclusion : the program to implemented recursive descent parser for grammar designed successfully.

Course outcome attained and Mapping with Program outcomes :-

Course outcomes	Program outcomes
CO1	PO1 , PO2 , PO3
CO2	PO4 , PO5

ELLENT

Result :

```
Enter an input string:
a
The input string can be accepted using the grammar.

Enter an input string:
b
The input string can be accepted using the grammar.

Enter an input string:
+a+b
The input string can be accepted using the grammar.

Enter an input string:
--a
The input string can be accepted using the grammar.

Enter an input string:
b+
The input string cannot be accepted from grammar.
```

Practical No 06

Aim :- Design a program for basic calculator using YACC or BISON.

Tools :

Sr. No	Tools	Specification	Qty
01	Compiler	YACC or BISON	-
02	Computer system	Windows, 4GB RAM 1 TB HDD	1

Theory :- YACC (yet Another compiler compiler) & BISON are tool that generate parsers (Programs analyze structure of text on code) based on formal desc of language grammar.
- commonly used in development of compilers interpreters, & other language-based s/w

Syntax :-

```
%{
    //declaration section
%}
%%
% token definition
    grammar declaration
%%
int main(){
yyparse();
return 1
}
```

CELLENT

Program :-

```
%{ #include<stdio.h>
#include <lex.yy.c> int yylex();
int yyerror();
%}
%%
%left '+''-';
%left '*''/';
expression;
lexpression '*' expression { $$ = $1 * $3};
    printf(" * :\d\n", $$);};
lexpression '+' expression {$$ = $1+$};
    printf("+ : %d\n",$$); };

| expression '-' expression { $$ = $1- $3};
    printf("-:%d\n",$$);};
| expression '/' expression { $$ = $1 /$3};
printf("/:%d\n",$$);};
%%
    int main() {
    yyparse();
    return 1;
}
```

Error and Remedial Action :-

1. Parsing error for invalid input expression yyerror() will generate error.

Remedial Action :- Print appropriate error message.

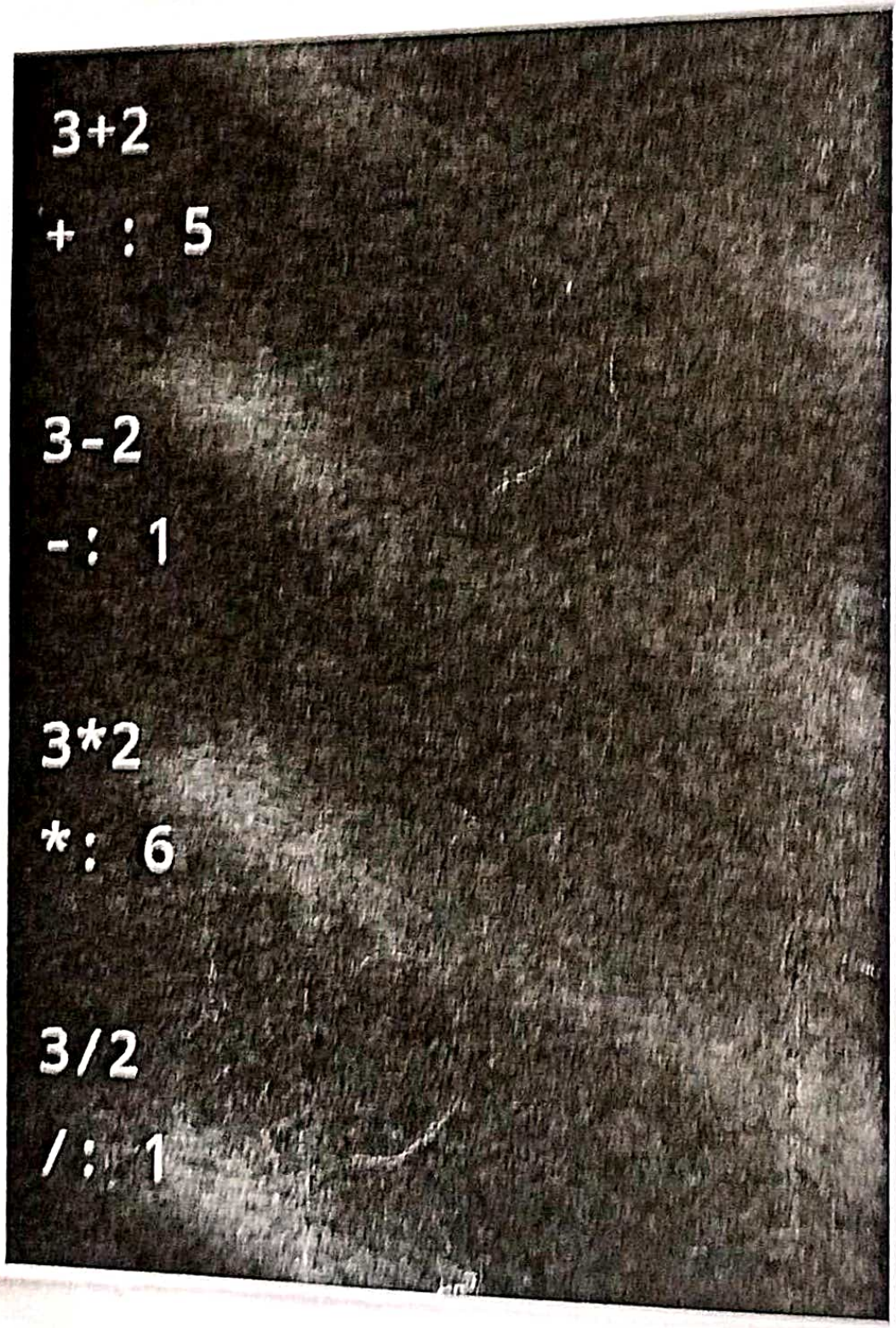
Conclusion :

The program for basic calculator using YACC or BISON designed successfully.

Course outcomes Attained & mapping with Program outcomes :

course outcomes	Program outcomes
CO-1	PO1, PO2, PO3, PO4
CO-2	PO5, PO7

Result :-



Practical NO : 07

Aim : Design a Program to evaluate postfix expression.

Tools :-

SE.NO	Tools	specification	qnty.
1	C compiler	Turbo C, acc, any	--
2	computer system	widows, 4GB RAM	01

Theory :-

Postfix Notation : way of writing arithmatic expressions in which each operation follow its Dpenands. Ex "3+4" ⇒ "34+"

Set steps to evaluate postfix expression : Ex : "3+4"

1. Scan Expression from Left to Right :
   - push 3 onto stack (stack : 3)
   - push 4 on to stack (stack : 3,4)
   - pop 3 and 4 from stack add them togather (4+3 = 7) and push the result (7) back onto stack

Return expression is in top element of stack which is 7.

Program :-

```
#include <stdio.h>
#include <ctype.h>
#define POST FIXSIZE 100
#define MAXSTACK 100
int stack[MAXSTACK];
int top = -1;
void push (int item)
{
 if (top >= MAXSTACK-1) {
 printf(" stack overflow ");
  return; }
 else { top = top+1; stack[top] = item; }}
int pop()
 {
 int item;
 if (top < 0) {
          printf(" stack under flow");
   }
  else { item = stack[top];
       top = top-1; return item; }
 }
void evaluate evalpostfix(char postfix[])
 { int i;
 int val;
 char ch;
 int A, B
 for (i=0; postfix[i] != '\0'); i++){
       ch = postfix[i];
  if(isdigit(ch)) { push(ch-'0'); }
```

```
else if ( ch == '+' || ch = "-" () ch = '*'
        || ch == '/' )
  {
      A = pop ();
      B = pop ();
      switch (ch)
  {
    case '*' :  val = B * A ;  break ;
    case '/' :  val = B / A ;  break ;
    case '+' :  val = B + A ;  break ;
    case '-' :  val = B - A ;  break ;
  }
  push (val) ;
}}
printf("\n Resu(.) of expression : %d \n", pop());
}
int main () { int i ;
              char postfix [ POSTFIXSIZES ] ;
printf (" \n intet postfix expression , \n press right
  parenthesis ')' for end expression :");
for (i=0 ; i <= POSTFIXSIZE - 1 ; i++ ) {
      scanf ("%c ", &postfix [i] );
if (postfix [i] == ')' ) { break ; } }
  EvalPostfix (postfix ) ;
      return 0;
  }
```

Result -

Error and Remedial Action :-

Error : Divide by zero
Remedial : Need to apply proper error message for divide by zero

Conclusion :- the program to evaluate postfix expression designed successfully.

Course outcomes attained & mapping with Program outcomes.

Course Outcomes	Program outcomes
CO-1	PO-1, PO-2, PO-3 PO-4

Result :-

```
Enter postfix expression, press right parenthesis ')' for end expression : 456*+)
Result of expression evaluation : 34

Enter postfix expression, press right parenthesis ')' for end expression: 12345*+*+)
Result of expression evaluation: 47
```

Page No. 30
Date :

Practical No 8

Aim:- Write a c program for selection sort Generate equivalent 3add-code.

Tools :

Sr.No	Tools	specification	@by
1	C compiler	Turbo C, cc, any	-
2	Computer system	windows, 4GB RAM	1

Theory :-

Selection sort :- It's a simple and easy to understand sorting algorithm that works by repeatedly selecting the smallest element from unsorted ~~patter~~ of the list.

It's repeated for the remaining unsorted ~~patter~~ portion of the list until the entire list is sorted. Three address code. It's a type of intermediate code which is easy to generates and can be easily converte to machine code. It makes use of atmost three address code and one operator to represent an expression

Program :-

```
#include<stdio.h>
void selectionsort(int arr[], int n)
{
    int i,j, min_idx;
    for(i=0; i<n-1; i++)
    {
    min_idx=i;
```

```
for (j=i+1 ; j<n ; j++)
  {
   if (arr[j] < arr[min_idx])
        min_idx = j ;

   int temp = arr[min_idx];
   arr[min_idx] = arr[i];
       arr[i] = temp;
  }
}
int main(){
       int arr[] = {64,25,12,22,11};
  int n = sizeof(arr)/sizeof(arr[0]);
   selectionsort(arr,n);
  printf("Sorted Array");
 for(int i=0; i<n ; i++)
  printf("%d", arr[i]);
  return 0;
}
```

3-address Code :-

Initialize variable
i = 0 ;
j = 0 ;
min_idx = 0 ;
i start outer loop
if i >= n-1 go 16

: Start inner loop
if j > n goto 14

```
if arr[j] < arr[min_idx] goto 13
   goto 14

; update min_idx
   L3 : min_idx = j
   L4 : j = j+1
   goto 12
   ; End inner loop

   ; swap arr[min_idx] with arr[i]
   temp = arr[min_idx]
    arr [min_idx] = arr [i]
       arr[ i] = temp

        i = i+1
        j = i
        goto L1
    ; END outer loop
; print sorted array
    L5 ; printf("Sorted Array : ").
     for i=0 to n-1
      Printf (" %d ", arr[i])

{
```

ERRORS and Remedial action :

ERRORS : Not handling dep

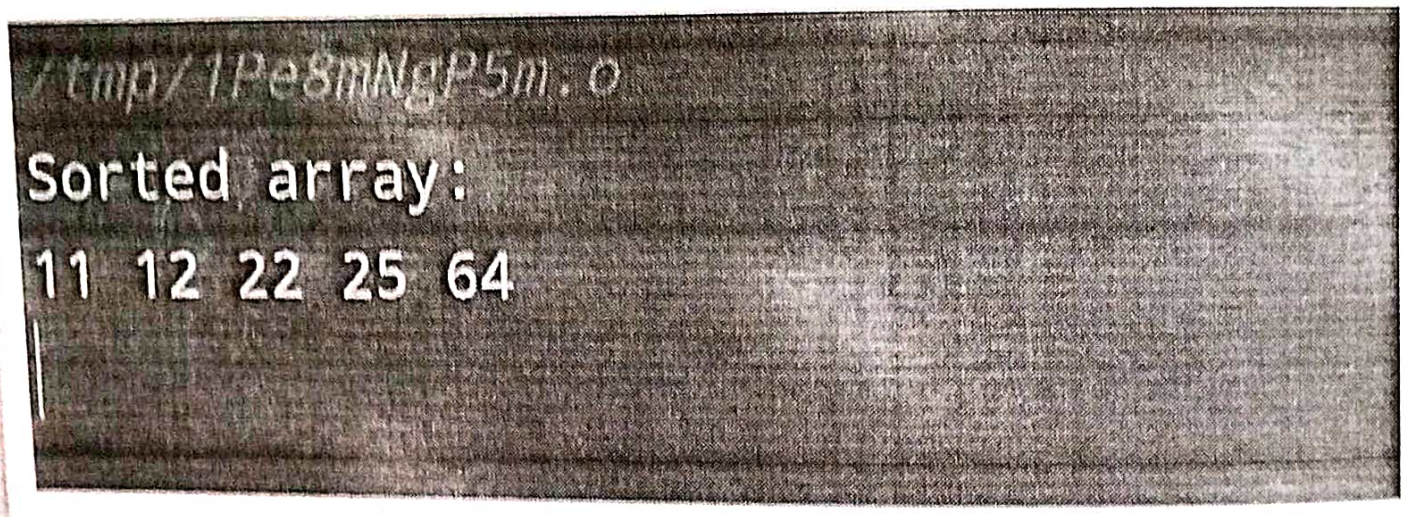
Remedial Action :- we are modify algorithm to keep track of all occurences of the minimum element.

Conclusion : C program for selection sort and its equivalent 3 add code has been generate successfully.

Cource outcome attained & mapping with Program outcomes

Cource outcome	Peogram outcome
co-3 Apply the knowledge YACC to syntax divided translation for generating intermediate code 3-add code	PO2 : Problem Analysis PO3 : Design / Dev. pt of solution.

Result :-



Practical No 09

Aim :- Write a Program in C ~~pcarno~~ program to Optimize the 3-address code generated.

Tools :-

Sr. No	Tools	specification	Qty
1	compiler	Computer (any) TURBO C ~~or~~ C++ OF GCC	
2	computer system	4GB RAM Windows, 1TB HDD	1

Theory :- **3Address code** : It is a type of intermediate code which is easy to generate and can be easily converted to Machine code. It make use of at most 3address and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.

3 Address code

Optimization :- 3Address code is often used as an intermediate representation of code during optimization phases of the comilation process. the 3-address code allows the compiler to analyze the and perform optimization that can improve the performance of the generated code.

PROGRAM :

Optimized Three Address code :

```
1) i = 0
2) j = 0
3) min_indx
4) if i > n-1 goto (23)
5] if j >= n goto (13)
6) t1 = arr [j]
7) t2 = arr [min_ind]
8) if t1 < t2, goto (10)
9) goto (13)
10) min_indx j ;
11) t3 = j+1 ;
12] j = t3
13) goto
14] swap t2 with arr[i]
15] t4 = arr[i]
16] t5 = t2
17) t2 = t4
18) t4 = t5
19) t6 = t1
20) i = t6
21) j = i
22] goto ()
23) printf ("sorted Array");
24) for @i = 0 to n-1 goto (25]
25] printf ("%d", t4)
26] goto (24)
```

~~ERROR~~ ERROR and Remedial Action

Error :- Not handling duplicate elements correctly

Remedial : We can modify algorithm to keep track of all occurence of the minimum element and swap them all to correct position.

Conclusion :-

Program to implement C program for Selection sort, then generating three address code for it, and after this optimizing the three address code and generation of object code implemented successfully.

Course outcome attained and Mapping with PO's :

Course outcome	Program outcomes
CO-4 : Build a code generation using different intermediate and code and optimize the target code.	PO1 : Engineering knowledge PO2 : problem Analyze PO3 : Design development of solution PO5 : Modern tool usage PO6 : individual and team work.

Practical No 10

Aim:- write a c program for selection sort generate and generate objects code for three address code.

Tools :-

SR.NO	Tools	Specification coby
1.	compilers	Turbo C, C++ IDLE
2.	computer system	windows, 4GB RAM 1TB HDD

Theory :-

Three address code is an intermediate code it is used by the optimizing compilers

In 3-address code the given expression is broken down into several separate instructions These instruction can easily translate into assembly language such three address code intruction has at most three operands. It is a combination of assigment and a binary operator.

Program :- object code Generated from optimized 3-address code :

```
1) start :
2) mov i, 0
3) mov j, 0
4) loop outer
5] cmpl n, i, jmp loop
6] cmple n, j, jmp (11)
7) mov a[i][j], t1
```

8] mov t2, arr[mid-ind]
9] comp t1, t2
10] mov min_ind, j
11] add j, 1
(12) mov j, t3
12] mov t4, arr[j]
13] mov t5, t2
14] mov t2, t4
15] mov t4, t5, add j, 1
16] mov t6, j
17] mov i, t6
18] mov j, i
19] loop end
20] loop
21] jmp loop

Conclusion : Program to implement C program for selection sort then generating 3-address code for it, and after this optimization the 3-address code and generation of object code implemented successfully.