COMPILER DESIGN LAB

MAULANA ABUL KALAM AZAD UNIVERSITY OF TECHNOLOGY

COMPILER DESIGN LAB NOTEBOOK

Name: Nima Dorjee Lama
Roll No.: 10000220031
Regn No.: 201000100210010
Subject: Compiler Design Lab
Subject Code: PECIT691A
Stream: B.TECH in Information Technology
Session: 2022-23

Sl. No.	Date Created	Topic	Page no.	Date submitted	Signature & Remarks
1.	10-05-23	Design a lexical analyzer	3-5	26-05-23	
2.	11-05-23	Program to find a comment.	6-7	26-05-23	
3.	12-05-23	Program to recognize strings under ‘a’, ‘a*b+’, ‘abb’.	8-10	26-05-23	
4.	13-05-23	Program to check the validity of an identifier.	11-12	26-05-23	
5.	14-05-23	Program to simulate lexical analyzer for validating operators.	13-15	26-05-23	
6.	15-05-23	Lexical analyzer using JLex, flex, etc.	15-16	26-05-23	
7.	16-05-23	Program to implement the functionalities of predictive parser.	17-22	26-05-23	
8.	17-05-23	Program for constructing of LL(1) parsing.	23-28	26-05-23	
		Program for constructing of recursive descent parsing.			
9.	18-05-23	Program to implement LALR parsing.	29-31	26-05-23	
10.	19-05-23	Program to implement operator precedence parsing	32-37	26-05-23	
		Program to implement semantic rules to calculate the expression that takes an expression with digits, + and * and computes the value.			
11.	20-05-23	Convert the BNF rules into YACC form and write code to generate abstract syntax tree for the mini language specified in (1).	38-43	26-05-23	
12.	21-05-23	Program to generate machine code from abstract syntax tree generated by the parser.	44-48	26-05-23	

EXPERIMENT – 1

1. Design a lexical analyzer for given language and the lexical analyzer should ignore redundant spaces, tabs, and new lines. It should also ignore comments. Although the syntax specification states that identifiers can be arbitrarily long. you may restrict the length to some reasonable value. Simulate the same in C language.

   Algorithm:

   1. Read the input Expression
   2. Check whether input is alphabet or digits then store it as identifier
   3. If the input is is operator store it as symbol
   4. Check the input for keywords

Code:
```
#include <string.h>
#include <conio.h>
#include <ctype.h>
#include <stdio.h>
void main()
{
    FILE *f1;
    char c, str[10];
    int lineno = 1, num = 0, i = 0;
    clrscr();
    printf("\nEnter the c program\n");
    f1 = fopen("input.txt", "w");
    while ((c = getchar()) != EOF)
        putc(c, f1);
    fclose(f1);
    f1 = fopen("input.txt", "r");
    while ((c = getc(f1)) != EOF) // TO READ THE GIVEN FILE
    {
        if (isdigit(c)) // TO RECOGNIZE NUMBERS
        {
            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)) // TO RECOGNIZE KEYWORDS AND
IDENTIFIERS
        {
            str[i++] = c;
            c = getc(f1);
            while (isdigit(c) || isalpha(c) || c == '_' || c ==
'$')
            {
                str[i++] = c;
                c = getc(f1);
            }
            str[i++] = '\0';
            if (strcmp("for", str) == 0 || strcmp("while", str)
== 0 || strcmp("do", str) == 0 ||
                strcmp("int", str) == 0 || strcmp("float", str)
== 0 || strcmp("char", str) == 0 ||
                strcmp("double", str) == 0 || strcmp("static",
str) == 0 ||
                strcmp("switch", str) == 0 || strcmp("case",
str) == 0) // TYPE 32 KEYWORDS
                printf("%s is a keyword\n", str);
            Blog - https : //
anilkumarprathipati.wordpress.com/ 6
                           else printf("%s is a identifier\n",
str);
            ungetc(c, f1);
            i = 0;
        }

        else if (c == ' ' || c == '\t') // TO IGNORE THE SPACE
            printf("\n");
        else if (c == '\n') // TO COUNT LINE NUMBER
            lineno++;
        else // TO FIND SPECIAL SYMBOL
```

```
            printf("%c is a special symbol\n", c);
    }
    printf("Total no. of lines are: %d\n", lineno);
    fclose(f1);
    getch();
}
```

Output:

```
Enter the c program
int main()
{
int a=10,20;
charch;
float f;
}^Z
The numbers in the program are: 10 20
The keywords and identifiersare:
int is  a  keyword
main is an identifier
int is a keyword
a is an identifier
char is a keyword
ch is an identifier
float is a keyword
f is an identifier
Special characters are ( ) { = , ; ; ; }
Total no. of lines are:5
```

EXPERIMENT – 2

2. Write a C program to identify whether a given line is a comment or not.
   Algorithm:
   1. Read the input string.
   2. Check whether the string is starting with ‘/’ and check next character is ‘/’ or’*’.
   3. If condition satisfies print comment.
   4. Else not a comment

Code:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char text[30];
    int i = 2, a = 0;
    clrscr();

    printf("\nWrite a C Program to Identify Whether a Given
Line is a Commentor Not.");
    printf("\n\nEnter Text : ");
    gets(text);
    if (text[0] == '/')
    {
        if (text[1] == '/')
        {
            printf("\nIt is a Comment.");
        }
        else if (text[1] == '*')
        {
            for (i = 2; i <= 30; i++)
            {
                if (text[i] == '*' && text[i + 1] == '/')
                {
                    printf("\nIt is a Comment.");
                    a = 1;
                    break;
                }
```

```c
                else
                {
                    continue;
                }
            }
            if (a == 0)
            {
                printf("\nIt is Not a Comment.");
            }
            else
            {
                printf("\nIt is Not a Comment.");
            }
        }
    }
    else
    {
        printf("\nIt is Not a Comment.");
    }
    getch();
}
```

Output:

```
Write a C Program to Identify Whether a Given Line is a Commentor
Not.

Enter Text : It is a comment

It is Not a Comment.
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }

Write a C Program to Identify Whether a Given Line is a Commentor
Not.

Enter Text : // It is a comment.

It is a Comment.
```

EXPERIMENT – 3

1. Write a C program to recognize strings under ‘a’, ‘a*b+’, ‘abb’.

   Algorithm:

   a. By using transition diagram we verify input of the state.
   b. If the state recognize the given pattern rule.
   c. Then print string is accepted under a*/ a*b+/ abb.
   d. Else print string not accepted

```
Code:
#include<stdio.h>
#include<conio.h>
#include<string.h>
char s[20],stack[20];
void main()
{
    char m[5][6][3]={"tb"," "," ","tb"," "," "," ","+tb"," ","
","n","n","fc"," "," ","fc"," "," "," ","n","*fc","
a         ","n","n","i"," "," ","(e)"," "," "};
    int
size[5][6]={2,0,0,2,0,0,0,3,0,0,1,1,2,0,0,2,0,0,0,1,3,0,1,1,1,0
,0,3,0,0};
    int i,j,k,n,str1,str2;
    clrscr();
    printf("\n Enter the input string: ");
    scanf("%s",s);
    strcat(s,"$");
    n=strlen(s);
    stack[0]='$';
    stack[1]='e';
    i=1;
    j=0;
    printf("\nStack     Input\n");
    printf("__________________\n");
    while((stack[i]!='$')&&(s[j]!='$'))
    {
        if(stack[i]==s[j])
        {
            i--;
            j++;
        }
        switch(stack[i])
        {
            case 'e': str1=0;
            break;
            case 'b': str1=1;
            break;
            case 't': str1=2;
            break;
            case 'c': str1=3;
            break;
            case 'f': str1=4;
            break;
        }
        switch(s[j])
        {
            case 'i': str2=0;
            break;
            case '+': str2=1;
            break;
            case '*': str2=2;
            break;
            case '(': str2=3;
            break;
            case ')': str2=4;
            break;
            case '$': str2=5;
            break;
        }
        if(m[str1][str2][0]=='\0')
        {
            printf("\nERROR");
            exit(0);
        }
        else if(m[str1][str2][0]=='n')
            i--;
        else if(m[str1][str2][0]=='i')
            stack[i]='i';
        else
        {
            for(k=size[str1][str2]-1;k>=0;k--)
            {
                stack[i]=m[str1][str2][k];
                i++;
            }
            i--;
        }

        for(k=0;k<=i;k++)
            printf(" %c",stack[k]);
        printf("       ");
        for(k=j;k<=n;k++)
            printf("%c",s[k]);
        printf(" \n ");
    }
    printf("\n SUCCESS");
    getch();
}
```

Output:

```
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }

 Enter the input string: i*i+i

Stack     Input
__________________
 $ b t        i*i+i$
  $ b c f       i*i+i$
  $ b c i       i*i+i$
  $ b c f *       *i+i$
  $ b c i       i+i$
  $ b       +i$
  $ b t +       +i$
  $ b c f       i$
  $ b c i       i$
  $ b       $

 SUCCESS
```

EXPERIMENT – 4

1. Write a C program to test whether a given identifier is valid or not.
   Algorithm:
   - Read the given input string.
   - Check the initial character of the string is numerical or any special character except '_' then print it is not a valid
   - identifier.
   - Otherwise print it as valid identifier if remaining characters of string doesn't contains any special characters
   - except '_'.

Code:
```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
char string[25];
int count=0,flag;
printf("Enter any string: ");
gets(string);
if(
(string[0]>='a'&&string[0]<='z')||(string[0]>='A'&&string[0]<='Z')
||(string[0]=='_'))
{
    for(int i=1;i<=strlen(string);i++)
    {
        if((string[i]>='a'&& string[i]<='z')||(string[i]>='A' &&
string[i]<='Z')||(string[i]>='0'&& string[i]<='9')||(string[i]=='-
'))
    {
    count++;
    }
       }
    if(count==strlen(string))
    {
      flag=0;
    }
}
else
{
    flag=1;
}
if(flag==1)
    printf("%s is not valid identifier",string);
else
    printf("%s is valid identifier",string);
return 0;
}
```

Output:

```
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }
Enter any string: 10
10 is not valid identifier
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }
Enter any string: _google
_google is valid identifier
```

EXPERIMENT – 5

1. Write a C program to simulate lexical analyzer for validating operators.
   Algorithm:
   - Read the given input.
   - If the given input matches with any operator symbol.
   - Then display in terms of words of the particular symbol.
   - Else print not a operator.

Code:
```
#include <stdio.h>
#include <string.h>
int main()
{
    char arithmetic[5] = {'+', '-', '*', '/', '%'};
    char relational[4] = {'<', '>', '!', '='};
    char bitwise[5] = {'&', '^', '~', '|'};
    char str[2] = {' ', ' '};
    printf("Enter value to be identified: ");
    scanf("%s", &str);
    int i;
    if (((str[0] == '&' || str[0] == '|') && str[0] == str[1]) ||
(str[0] == '!' && str[1] == '\0'))
    {
        printf("\nIt is Logical operator");
    }
    for (i = 0; i < 4; i++)
    {
        if (str[0] == relational[i] && (str[1] == '=' || str[1] ==
'\0'))
        {
            printf("\n It is relational Operator");
            break;
        }
    }
    for (i = 0; i < 4; i++)
    {
        if ((str[0] == bitwise[i] && str[1] == '\0') || ((str[0]
== '<' || str[0] == '>') && str[1] == str[0]))
        {
            printf("\n It is Bitwise Operator");
            break;
        }
    }
    if (str[0] == '?' && str[1] == ':')
        printf("\nIt  is  ternary operator");
    for (i = 0; i < 5; i++)
    {
        if ((str[0] == '+' || str[0] == '-') && str[0] == str[1])
        {
            printf("\nIt is unary operator");
            break;
        }
        else if ((str[0] == arithmetic[i] && str[1] == '=') ||
(str[0] == '=' && str[1] == ' '))
        {
            printf("\nIt is Assignment operator");
            break;
        }
        else if (str[0] == arithmetic[i] && str[1] == '\0')
        {
            printf("\nIt is arithmetic operator");
            break;
        }
    }
    return 0;
}
```

Output:

```
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }
Enter value to be identified: +

It is arithmetic operator
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }
```

```
Enter value to be identified: <

 It is relational Operator
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }
Enter value to be identified: &

 It is Bitwise Operator
```

EXPERIMENT – 6

1. Implement the lexical analyzer using JLex, flex or other lexical analyzer generating tools.

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

Code:

```
// Program name as “lexicalfile.l”
%{
#include<stdio.h>
%}
delim [\t]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
num {digit}+(\.{digit}+)?(E[+|-]?{digit}+)?
%%
ws {printf("no action");}
if|else|then {printf("%s is a keyword",yytext);} // TYPE 32
KEYWORDS
{id} {printf("%s is a identifier",yytext);}
{num} {printf(" it is a number");}
"<" {printf("it is a relational operator less than");}
"<=" {printf("it is a relational operator less than or equal");}
">" {printf("it is a relational operator greater than");}
">=" {printf("it is a relational operator greater than");}
```

```
"==" {printf("it is a relational operator equal");}
"<>" {printf("it is a relational operator not equal");}
%%
main()
{
yylex();
}}
```

Output:

```
lexlexicalfile.l
cc lex.yy.c -ll
if
if is a keyword
number
number is a identifier
254
It is a number
<>
it is a relational operator not equal
^ZEnter value to be identified: <

 It is relational Operator
PS C:\Users\Deepesh\Desktop> cd "c:\Users\Deepesh\Desktop\" ; if
($?) { gcc cd.c -o cd } ; if ($?) { .\cd }
Enter value to be identified: &

 It is Bitwise Operator
```

EXPERIMENT – 7

1. Write a C program for implementing the functionalities of predictive parser for the mini language specified in Note 1.

   Algorithm:

   - Read the input string.
   - By using the FIRST AND FOLLOW values.
   - Verify the FIRST of non terminal and insert the production in the FIRST value
   - If we have any @ terms in FIRST then insert the productions in FOLLOW values
   - Constructing the predictive parser table

Code:

```c
#include <stdio.h>
#include <string.h>
#define SIZE 30
char st[100];
int top = -1;
int s1(char), ip(char);
int n1, n2;
char nt[SIZE], t[SIZE];
/*Function to return variable index*/
int s1(char c)
{
    int i;
    for (i = 0; i < n1; i++)
    {
        if (c == nt[i])
            return i;
    }
}
/*Function to return terminal index*/
int ip(char c)
{
    int i;
    for (i = 0; i < n2; i++)
    {
        if (c == t[i])
            return i;
    }
```

```
}
void push(char c)
{
    top++;
    st[top] = c;
    return;
}
void pop()
{
    top--;
    return;
}
main()
{
    char table[SIZE][SIZE][10], input[SIZE];
    int x, f, s, i, j, u;
    printf("Enter the number of variables:");
    scanf("%d", &n1);
    printf("\nUse single capital letters for variables\n");
    for (i = 0; i < n1; i++)
    {
        printf("Enter the %d nonterminal:", i + 1);
        scanf("%c", &nt[i]);
    }
    printf("Enter the number of terminals:");
    scanf("%d", &n2);
    printf("\nUse single small letters for terminals\n");
    for (i = 0; i < n2; i++)
    {
        printf("Enter the %d terminal:", i + 1);
        scanf("%c", &t[i]);
    }
    /*Reading the parsing table*/
    printf("Please enter only right sides of productions\n");
    printf("Use symbol n to denote no entry and e to epsilon\n");
    for (i = 0; i < n1; i++)
    {
        for (j = 0; j < n2; j++)
        {
            printf("\nEnter the entry for %c under %c:", nt[i], t[j]);
            scanf("%s", table[i][j]);
        }
    }
    /*Printing the parsing taable*/
    for (i = 0; i < n2; i++)
        printf("\t%c", t[i]);
    printf("\n--------------------------------------------------------------------------------------------------------
\n");
    for (i = 0; i < n1; i++)
    {
        printf("\n%c|\t", nt[i]);
        for (j = 0; j < n2; j++)
        {
            if (!strcmp(table[i][j], "n"))
                printf("\t");
            else
                printf("%s\t", table[i][j]);
        }

        printf("\n");
    }
    printf("Enter the input:");
    scanf("%s", input);
    /*Initialising the stack*/
    top++;
    st[top] = '$';
    top++;
    st[top] = nt[0];
    printf("STACK content INPUT content PRODUCTION used\n");-----------------------
    printf("
--\n");
    i = 0;
    printf("$%c\t\t\t%s\n\n", st[top], input);
    while (st[top] != '$')
    {
        x = 0;
        f = s1(st[top]);
        s = ip(input[i]);
        if (!strcmp(table[f][s], "n"))
        {
            printf("'String not accepted");
        }
        else if (!strcmp(table[f][s], "e"))
        {
            pop();
        }
        else if (st[top] == input[i])
        {
            x = 1;
            pop();
            i++;
        }
        else
        {
            pop();
            for (j = strlen(table[f][s]) - 1; j > 0; j--)
            {
                {
                    push(table[f][s][j]);
                }
            }
            for (u = 0; u <= top; u++)
                printf("%c", st[u]);
            printf("\t\t\t");
            for (u = i; input[u] != '\0'; u++)
                printf("%c", input[u]);
            printf("\t\t\t");
            if (x == 0)
                printf("%c->%s\n\n", nt[f], table[f][s]);
            printf("'\n\n");
        }
        printf("\n\nThus string is accepted");
    }
}
```

Output:

```
Enter the number of variables:5
Use single capital letters for the variables
Enter the 1 non terminal:E
Enter the 2 non terminal:A
Enter the 3 non terminal:T
Enter the 4 non terminal:B
Enter the 5 non terminal:F
Enter the number of terminals:6
Use only single small letter for the terminals
Enter the 1 terminal:+
Enter the 2 terminal:*
Enter the 3 terminal:(
Enter the 4 terminal:)
Enter the 5 terminal:i
Enter the 6 terminal:$
Please enter only the right sides of productions.
Use symbol n to denote noentry and e to epsilon
Enter the entry for E under $: n
Enter the entry for E under +: n
Enter the entry for E under *: n
Enter the entry for E under (: TA
Enter the entry for E under ): n
Enter the entry for E under i: TA
Enter the entry for A under +: +TA
Enter the entry for A under *: n
Enter the entry for A under (: n
Enter the entry for A under ): e
Enter the entry for A under i: n
Enter the entry for A under $: e
Enter the entry for T under +: n
Enter the entry for T under *: n
Enter the entry for T under (: FB
Enter the entry for T under ): n
Enter the entry for T under i: FB
Enter the entry for T under $: n
Enter the entry for B under +: e
Enter the entry for B under *: *FB
Enter the entry for B under (: n
Enter the entry for B under ): e
Enter the entry for B under i: n
Enter the entry for B under $: e
Enter the entry for F under +: n
Enter the entry for F under *: n
Enter the entry for F under (: (E)
Enter the entry for F under ): n
Enter the entry for F under i: i
Enter the entry for F under $: n
+ * ( ) i $
-------------------------------------------------------------------
-----------
E| TA TA
A| +TA e e
T| FB FB
B| e *FB e e
F| (E) i
Enter the input: i+i*i$
Stack content Input content Production used
-------------------------------------------------------------------
-----------------
$E i+i*i$
```

```
$A i+i*i$ E->TA
$AB i+i*i$ T->FB
$AB i+i*i$ F->i
$A +i*i$
$ +i*i$ B->e
$AT +i*i$ A->+TA
$A i*i$
$AB i*i$ T->FB
$AB i*i$ F->i
$A *i$
$ABF *i$ B->*FB
$AB i$
$AB i$ F->i
$A $
$ $ B->e
$ A->e
Thus string is accepted
```

EXPERIMENT – 8

a) Write a C program for constructing of LL(1) parsing.

Algorithm:

- Read the input string.
- Using predictive parsing table parse the given input using stack .
- If stack [i] matches with token input string pop the token else shift it repeat the process until it reaches to $.

Code:

```
#include <stdio.h>
#include <string.h>
char str[25], st[25], *temp, v, ch, ch1;
char t[5][6][10] = {"$", "$", "TX", "TX", "$", "$",
                    "+TX", "$", "$", "$", "e", "e",
                    "$", "$", "FY", "FY", "$", "$",
                    "e", "*FY", "$", "$", "e", "e",
                    "$", "$", "i", "(E)", "$", "$"};
int i, k, n, top = -1, r, c, m, flag = 0;
void push(char t)
{
    top++;
    st[top] = t;
}
char pop()
{
    ch1 = st[top];
    top--;
    return ch1;
}
main()
{
    printf("enter the string:\n");
    scanf("%s", str);
    n = strlen(str);
    str[n++] = '$';
    i = 0;
    push('$');
    push('E');
    printf("stack\tinput\toperation\n");
    while (i < n)
    {
        for (k = 0; k <= top; k++)
            printf("%c", st[k]);
        printf("\t");
        for (k = i; k < n; k++)
            printf("%c", str[k]);
        printf("\t");
        if (flag == 1)
            printf("pop");
        if (flag == 2)
            printf("%c->%s", ch, t[r][c]);
        if (str[i] == st[top])
        {
            flag = 1;
            ch = pop();
            i++;
        }
        else
        {
            flag = 2;
            if (st[top] == 'E')
                r = 0;
            else if (st[top] == 'X')
                r = 1;
            else if (st[top] == 'T')
                r = 2;
            else if (st[top] == 'Y')
                r = 3;
            else if (st[top] == 'F')
                r = 4;
            else
                break;
            if (str[i] == '+')
                c = 0;
            else if (str[i] == '*')
                c = 1;
            else if (str[i] == 'i')
                c = 2;
            else if (str[i] == '(')
                c = 3;
            else if (str[i] == ')')
                c = 4;
            else if (str[i] == '$')
```

```
                c = 5;
            else
                break;
            if (strcmp(t[r][c], "$") == 0)
                break;
            ch = pop();
            temp = t[r][c];
            m = strlen(temp);
            if (strcmp(t[r][c], "e") != 0)
            {
                for (k = m - 1; k >= 0; k--)
                    push(temp[k]);
            }
        }
        printf("\n");
    }
    if (i == n)
        printf("\nparsed successfully");
    else
        printf("\nnot parsed");
}
```

Output:

```
1)
Enter any String(Append with $)i+i*i$
Stack Input Output
$E i+i*i$
$HT i+i*i$ E->TH
$HUF i+i*i$ T->FU
$HUi i+i*i$ F->i
$HU +i*i$ POP
$H +i*i$ U->ε
$HT+ +i*i$ H->+TH
$HT i*i$ POP
$HUF i*i$ T->FU
$HUi i*i$ F->i
$HU *i$ POP
$HUF* *i$ U->*FU
$HUF i$ POP
$HUi i$ F->i
$HU $ POP
$H $ U->ε
$ $ H->ε
Given String is accepted
```

```
2)
Enter any String(Append with $)i+i**i$
Stack Input Output
$E i+i**i$
$HT i+i**i$ E->TH
$HUF i+i**i$ T->FU
$HUii+i**i$ F->i
$HU +i**i$ POP
$H +i**i$ U->ε
$HT+ +i**i$ H->+TH
$HT i**i$ POP
$HUF i**i$ T->FU
$HUi i**i$ F->i
$HU **i$ POP
$HUF* **i$ U->*FU
$HUF *i$ POP
$HU$ *i$ F->$
Syntax Error
Given String is not accepted
```

b) Write a C program for constructing recursive descent parsing.

Alogrithm:

- Read the input string.
- Write procedures for the non terminals
- Verify the next token equals to non terminals if it satisfies match the non terminal.
- If the input string does not match print error.

Code:
```
#include <stdio.h>
#include <string.h>
char input[10];
int i = 0, error = 0;
void E();
void T();
voidEprime();
voidTprime();
void F();
void main()
{
    clrscr();
    printf("Enter an arithmetic expression :\n");
    gets(input);
    E();
    if (strlen(input) == i && error == 0)
        printf("\nAccepted..!!!");
    else
        printf("\nRejected..!!!");
    getch();
}
void E()
{
    T();
    Eprime();
}
voidEprime()
{
    if (input[i] == '+')
    {
        i++;
        T();
        Eprime();
    }
}
void T()
{
    F();
    Tprime();
}
voidTprime()
{
    if (input[i] == '*')
    {
        i++;
        F();
        Tprime();
    }
}
void F()
{
    if (input[i] == '(')
    {
        i++;
        E();
        if (input[i] == ')')
            i++;
    }
    else if (isalpha(input[i]))
    {
        i++;
        while (isalnum(input[i]) || input[i] == '_')
            i++;
    }
    else
        error = 1;
}
```

Output:

```
1)
Enter an arithmetic expression :
sum+month*interest
Accepted..!!!
2)
Enter an arithmetic expression :
sum+avg*+interest
Rejected..!!!
```

EXPERIMENT – 9

1. Write a C program to implement LL(1) parsing.

   Algorithm:

   - Read the input string.
   - Push the input symbol with its state symbols in to the stack by referring lookaheads
   - We perform shift and reduce actions to parse the grammar.
   - Parsing is completed when we reach $ symbol.

Code:

```
{%
#nclude<stdio.h>
#include<conio.h>
intyylex(void);
%}
%token ID
%start line
%%
line:expr '\n', {printf("%d",S1);}
expr:expr'+'term {SS=S1+S3;}
 |term
term:term'*'factor {SS=S1+S3;}
|factor
factor:'('expr')' {SS=S2;}
|ID
%%
yylex()
{
char c[10],i;
gets(c);
if(isdigit(c))
{
yylval=c;
return ID;
}
return c;
}
```

Output:

```
$vi lalr.y
$yacc –v lalr.y
```

```
$vi y.output
y.output contains the ouput
1 line : expr '\n'
2 expr : expr '+' term
 3 | term
4 term : term '*' factor
 5 | factor
6 factor : '(' expr ')'
 7 | ID
^L
state 0
 $accept : . line $end (0)
ID shift 1
 '(' shift 2
 . error
line goto 3
exprgoto 4
term goto 5
state 1
factor : ID . (7)
 . reduce 7
state 2
factor : '(' . expr ')' (6)
ID shift 1
 '(' shift 2
 . error
exprgoto 7
term goto 5
factor goto 6
state 3
 $accept : line . $end (0)
 $end accept
state 4
line :expr . '\n' (1)
expr :expr . '+' term (2)
'\n' shift 8
 '+' shift 9
 . error
state 5
expr : term . (3)
term : term . '*' factor (4)
 '*' shift 10
 '\n' reduce 3
 '+' reduce 3
 ')' reduce 3
state 6
```

```
term : factor . (5)
 . reduce 5
state 7
expr :expr . '+' term (2)
factor : '(' expr . ')' (6)
'+' shift 9
 ')' shift 11
 . error
state 8
line :expr '\n' . (1)
 . reduce 1
state 9
expr :expr '+' . term (2)
ID shift 1
 '(' shift 2
 . error
term goto 12
factor goto 6
state 10
term : term '*' . factor (4)
ID shift 1
 '(' shift 2
 . error
factor goto 13
state 11
factor : '(' expr ')' . (6)
 . reduce 6
state 12
expr :expr '+' term . (2)
term : term . '*' factor (4)
'*' shift 10
 '\n' reduce 2
 '+' reduce 2
 ')' reduce 2
state 13
term : term '*' factor . (4)
 . reduce 4
8 terminals, 5 nonterminals
8 grammar rules, 14 states
```

EXPERIMENT – 10

a) Write a C program to implement operator precedence parsing.

Algorithm:

- Read the input string.
- Push the input symbol with its state symbols in to the stack by referring lookaheads
- We perform shift and reduce actions to parse the grammar.
- Parsing is completed when we reach $ symbol.

Code:

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char stack[20], ip[20], opt[10][10][1], ter[10];
    int i, j, k, n, top = 0, row, col;
    clrscr();
    for (i = 0; i < 10; i++)
    {
        stack[i] = NULL;
        ip[i] = NULL;
        for (j = 0; j < 10; j++)
        {
            opt[i][j][1] = NULL;
        }
    }
    printf("Enter the no.of terminals:");
    scanf("%d", &n);
    printf("\nEnter the terminals:");
    scanf("%s", ter);
    printf("\nEnter the table values:\n");
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("Enter the value for %c %c:", ter[i], ter[j]);
            scanf("%s", opt[i][j]);
        }
    }
    printf("\nOPERATOR PRECEDENCE TABLE:\n");
    for (i = 0; i < n; i++)
    {
        printf("\t%c", ter[i]);
    }
    printf("\n_________________________________________");
    printf("\n");
    for (i = 0; i < n; i++)
    {
        printf("\n%c |", ter[i]);
        for (j = 0; j < n; j++)
        {
            printf("\t%c", opt[i][j][0]);
        }
    }
    stack[top] = '$';
    printf("\n\nEnter the input string(append with $):");
    scanf("%s", ip);
    i = 0;
    printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
    printf("\n%s\t\t\t%s\t\t\t", stack, ip);
    while (i <= strlen(ip))
    {
        for (k = 0; k < n; k++)
        {
            if (stack[top] == ter[k])
                row = k;
            if (ip[i] == ter[k])
                col = k;
        }
        if ((stack[top] == '$') && (ip[i] == '$'))
        {
            printf("String is ACCEPTED");
            break;
        }
        else if ((opt[row][col][0] == '<') || (opt[row][col][0] ==
'='))
```

```
        {
            stack[++top] = opt[row][col][0];
            stack[++top] = ip[i];
            ip[i] =  '  ';
            printf("Shift %c", ip[i]);
            i++;
        }
        else
        {
            if (opt[row][col][0] == '>')
            {
                while (stack[top] != '<')
                {
                    --top;
                }
                top = top - 1;
                printf("Reduce");
            }
            else
            {
                printf("\nString is not accepted");
                break;
            }
        }
        printf("\n");
        printf("%s\t\t\t%s\t\t\t", stack, ip);
    }
    getch();
}
```

Output:

```
Enter the no.of terminals:4
Enter the terminals:i+*$
Enter the table values:
Enter the value for i i:
-
Enter the value for i +:>
Enter the value for i *:>
Enter the value for i $:>
Enter the value for + i:<
Enter the value for + +:>
Enter the value for + *:<
Enter the value for + $:>
Enter the value for * i:<
Enter the value for * +:>
Enter the value for * *:>
Enter the value for * $:>
Enter the value for $ i:<
Enter the value for $ +:<
Enter the value for $ *:<
Enter the value for $ $:-
OPERATOR PRECEDENCE TABLE:
 i + * $
_________________________________________
i | - > > >
+ | < > < >
* | < > > >
$ | < < < -
Enter the input string(append with $):i+i*i$
STACK INPUT STRING ACTION
$ i+i*i$ Shift
$<i +i*i$ Reduce
$<i +i*i$ Shift
$<+ i*i$ Shift
$<+<i *i$ Reduce
$<+<i *i$ Shift
$<+<* i$ Shift
$<+<*<i $ Reduce
$<+<*<i $ Reduce
$<+<*<i $ Reduce
$<+<*<i $ String is ACCEPTED
```

b) Write a C program to implement Program semantic rules to calculate the expression that takes an expression with digits, + and * and computes the value.

Algorithm:

- Reading an input file
- Calculate the sum or multiplication of given expression.
- Using expression rule print the result of the given values.

Code:

```
<parser.l>
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.dval=atof(yytext);
return DIGIT;
}
\n|. return yytext[0];
%%
<parser.y>
%{
/*This YACC specification file generates the LALR parser for the
program
considered in experiment 4.*/
#include<stdio.h>
%}
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
line: expr '\n' {
38
printf("%g\n",$1);
}
;
expr: expr '+' term {$$=$1 + $3 ;}
| term
;
term: term '*' factor {$$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' {$$=$2 ;}
| DIGIT
;
%%
```

```
int main()
{
yyparse();
}
yyerror(char *s)
{
printf("%s",s);
}
```

Output:

```
$lex parser.l
$yacc –d parser.y
$cc lex.yy.c y.tab.c –ll –lm
$./a.out
2+3
5.0000
```

EXPERIMENT – 11

1. Convert The BNF rules into Yacc form and write code to generate abstract syntax tree.

   Algorithm:

   - Reading an input file line by line.
   - Convert it in to abstract syntax tree using three address code.
   - Represent three address code in the form of quadruple tabular form.

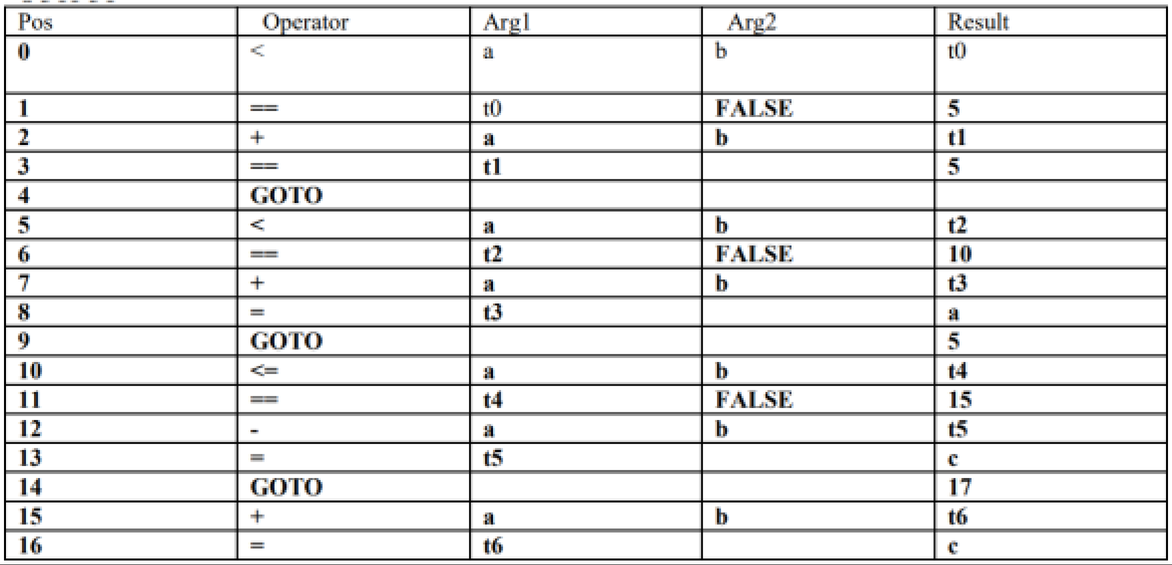
```
Code:
<int.l>
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
< |> |>= |<= |== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
. return yytext[0];
%%
<int.y>
%{
#include<string.h>
#include<stdio.h>
struct quad{
```

```
char  op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
struct stack{
int items[100];
int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
40
extern int LineNo;
%}
%union{
char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
}
;
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
| EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}
| EXPR '*' EXPR {AddQuadruple("*",$1,$3,$$);}
| EXPR '/' EXPR {AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
41
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
```

```
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[]) {
FILE *fp;
int i;
if(argc>1){
fp=fopen(argv[1],"r");
if(!fp) {
printf("\n File not found");
exit(0);
42
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------------------------""\n\t\t Pos Operator
Arg1 Arg2 Result" "\n\t\t
--------------------");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t %s\t
%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t -----------------------");
printf("\n\n");
return 0;
}
void push(int data){
stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{
int data;
if(stk.top==-1){
printf("\n Stack underflow\n");
exit(0);}
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char
result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

Output:

```
$lex int.l
$yacc –d int.y
$gcc lex.yy.c y.tab.c –ll –lm$./a.out test.c
```



Pos	Operator	Arg1	Arg2	Result
0	<	a	b	t0
1	==	t0	FALSE	5
2	+	a	b	t1
3	==	t1		5
4	GOTO			
5	<	a	b	t2
6	==	t2	FALSE	10
7	+	a	b	t3
8	=	t3		a
9	GOTO			5
10	<=	a	b	t4
11	==	t4	FALSE	15
12	-	a	b	t5
13	=	t5		c
14	GOTO			17
15	+	a	b	t6
16	=	t6		c

EXPERIMENT – 12

1. Write a C program to generate machine code from abstract syntax tree generated by the parser. The instruction set specified in Note 2 may be considered as the target code.

   Algorithm:

   - Read input string
   - Consider each input string and convert in to machine code instructions

```
Code:
#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))
{
45
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
46
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;
```

```
47
}
int check_label(int k)
{
int i;
for(i=0;i<no;i++)
{
if(k==label[i])
return 1;
}
return 0;
}
```

Output:

```
$vi 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
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
```