SYSTEM SOFTWARE AND COMPILER DESIGN LAB
(Effective from the academic year 2025-2026)
SEMESTER – V

Course Code	**21CSL56**	**CIE Marks**	**50 Marks**
Number of Lecture Hours/Week	**02**	**SEE Marks**	**50 Marks**
Total Number of Lecture Hours	**30**	**Exam Hours**	**03**

CREDITS – 01

Course Objectives:

1. To make students familiar with Lexical Analysis and Syntax Analysis phases of Compiler
2. Design and implement programs on these phases using LEX & YACC tools and/or C/C++/Java

No.	List of Experiments:
	PART-A
	Execute the following programs using LEX:
1.	a.Program to count the number of characters, words, spaces and lines in a given input file.
	b.Program to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.
2.	a.Program to recognize a valid arithmetic expression and to recognize the identifiers and operators present. Print them separately.
	b.Write a LEX program to scan reserved words and identifiers of C language
	Execute the following programs using YACC:
3.	Program to evaluate an arithmetic expression involving operators +, -, * and /.
4.	Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.
5.	a.Program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using the grammar (anbn, n>= 0).
	b.Program to recognize the grammar (anb, n>= 10).
	PART B
6.	Design, develop and implement program to construct Predictive / LL(1)Parsing Table for the grammar rules: $A \rightarrow aBa, B \rightarrow bB\|\varepsilon$. Use this table to parsethe sentence: abba$
7.	Design, develop and implement program to demonstrate Shift Reduce Parsing technique for the grammar rules: $E \rightarrow E + T\|TT \rightarrow T * F\|F$, F$\rightarrow (E)\|id$ and parse the sentence: id + id * id.
8.	Design, develop and implement syntax-directed definition of “if E then S1” and “if E then S1 else S2”
9.	Write a yacc program that accepts a regular expression as input and produce its parse tree as output.
10.	Design, develop and implement a program to generate the machine code usingTriples for the statement A = -B * (C +D) whose intermediate code in three-addressform: ***T1 = -B*** ***T2 = C + D T3*** ***= T1 + T2 A =*** ***T3***

CO#	Course Outcomes
1.	Demonstrate an understanding of compiler design principles and the usage of practical tools
2.	Develop and implement the programs using lex and yacc tools
3.	Debug and troubleshoot issues effectively.
4.	Analyze the data and interpret the results.
5.	Prepare a well-organized laboratory report.

COURSE ARTICULATION MATRIX

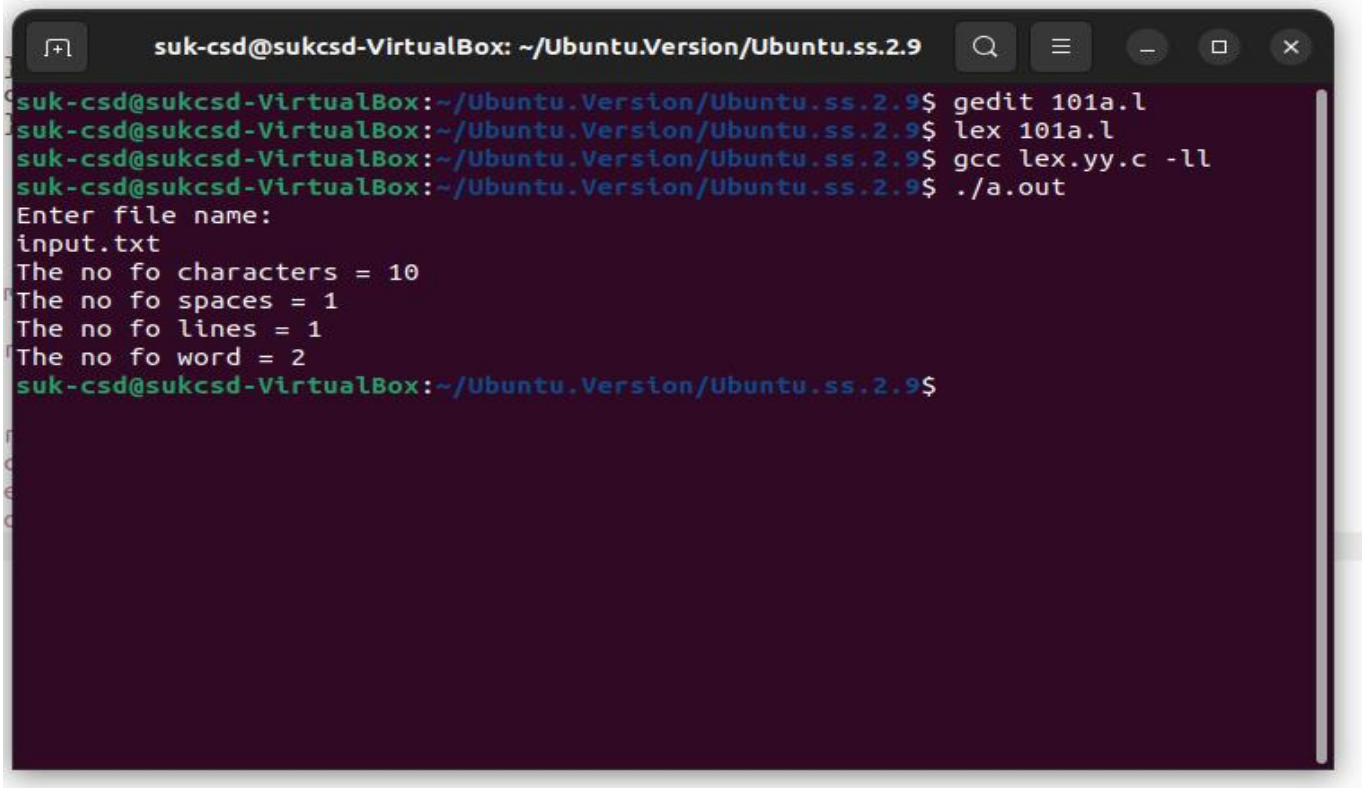
CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO 10	PO1 1	PO12	PSO 1	PSO 2	PS O3
CO1	3	2	1	-	-	-	-	-	-	-	-	2	1	3	3
CO2	1	1	3	-	3	-	-	-	-	-	-	2	1	3	3
CO3	2	3	-	-	1	-	-	-	-	-	-	-	1	2	3
CO4	1	2	2	-	-	-	-	-	-	-	-	-	1	1	3
CO5	1	-	-	-	-	-	-	-	-	3	-	-	1	-	3
AVG	**1.6**	**2**	**2**	**-**	**2**	**-**	**-**	**-**	**-**	3	**-**	2	**1**	**2.25**	**3**

PART A

1. a. Program to count the number of characters, words, spaces and lines in a given input file.

```
%{
      int cc=0,wc=0,sc=0,lc=0;
%}
%%
[   ]{sc++;cc++;}
[ ^  \t\n ]+
{wc++;cc+=yyleng;}
[  \n ]  {lc++;cc++;}
[  \t  ]  {sc+=8;cc++;}
%%
int main()
{
char fname[20];
printf(“Enter a filename:\n”);
scanf(“%s”,fname);
 yyin=fopen(fname,"r");
yylex();
fclose(yyin);
printf("The no.of characters = %d\n",cc);
printf("The no.of spaces = %d\n",sc);
printf("The no.of words = %d\n",wc);
}
```

Output:-



1b. Program to count the numbers of comment lines in a given C program. Also eliminate them and copy the resulting program into separate file.

```
%{
int com=0;
%}

%%
“/*”[^ \n]+”*/” {com++;fprintf(yyout,” “);}
%%
int main()
{
      printf(“write a c program\n”);
      yyout=fopen(“output”,”w”);
      yylex();
      printf(“comment=%d\n”,com);
      fclose(yyout);
      return 0;
}
```

Output:-

```
[student@localhost ~]# lex Prg1B.l
[student@localhost ~]# cc lex.yy.c  -ll
[student@localhost ~]# ./a.out
```

Write a c program

```
#include<stdio.h>
int main()
{
// Simple Program
Printf(" Hii ");
}
Ctrl +d
Comment=1
```

2. a. Program to recognize a valid arithmetic expression and to recognize the identifiers and operators present print them separately.

```
%{
intopdcount=0,oprcount=0,i=0,j=0,k=0;
char OPD[20],OPR [20];
%}

OPERATOR [+\-\*\/]
OPERANDS [a-zA-Z0-9]+

%%
{OPERATOR} {oprcount++;opr[i++]=yytext[0];opr[i++]=' ';}
{OPERANDS} {opdcount++;for(j=0;j<yyleng;j++) opd[k++]=yytext[j];
opd[k++]='';}
\n {return 0;}
. ;
%%

int main()
{
        yylex();
        IF((OPDCOUNT-OPRCOUNT)!=1)
                printf("Not valid expression\n");
        else
        {
```

```
            printf("Valid\n");
            printf("the operands = %s and count = %d\n",opd,opdcount);
            printf("The operators = %s and count = %d\n",opr,oprcount);
           }
      return 1;
   }
```

Output:

```
[student@localhost ~]# lex Prg2A.l
[student@localhost ~]# cc lex.yy.c  -ll
[student@localhost ~]# ./a.out

a+b*(e*d)/f*(h-g)
Valid
The operands = a b e d f h g and count = 7
 The operators = + * * / * - and count=6
```

2. b Write a LEX Program to scan reserved word & Identifiers of C Language.

```
%{
#include<stdio.h>
int count=0;
%}
LETTER [a-zA-Z]
DIGIT [0-9]
USCORE [ _ ]

%%
 ({LETTER}|{USCORE})({LETTER}|{DIGIT}|{USCORE})*) {count++;}
 . {}
%%

int main()
{
yyin=fopen("ex","r");
system("cat ex");
yylex();
fclose(yyin);
printf("\n no of identifiers=%d\n",count);
return 0;
```

```
}
```

Output:-

```
[student@localhost~]#lex2B.l
[student@localhost~]#cc lex.yy.c -ll
[student@localhost~]#./a.out
1243_4664
```

Filename

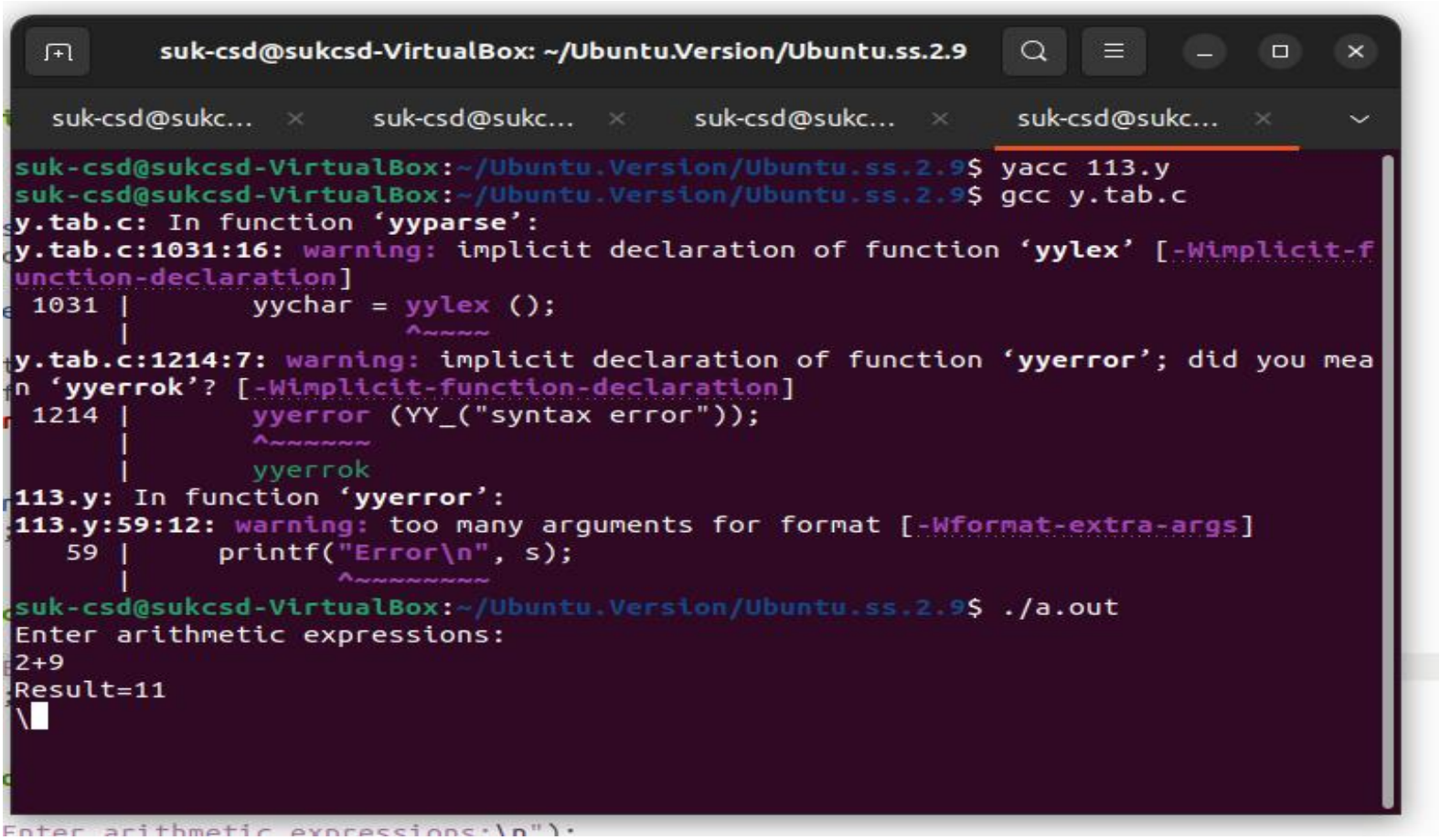
```
a_sdsds
no data
 no. of identifiers = 5
```

3. Write a YACC Program to evaluate an arithmetic expression involving operators +, -, * and /.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#define YYSTYPE double
%}
%token num
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines   :
           | lines exp '\n'    { printf("Result=%g\n", $2); }
            | lines  '\n'
            ;

exp     :
       exp '+' exp        { $$ = $1 + $3; }
       | exp '-' exp      { $$ = $1 - $3; }
       | exp '*' exp      { $$ = $1 * $3; }
       | exp '/' exp
          {
             if ($3 == 0)
                {
                printf("Divide by zero error\n");
                exit(0);
             }
```

```
            $$ = $1 / $3;
          }
       | '(' exp ')'       { $$ = $2; }
       | '-' exp %prec UMINUS { $$ = -$2; }
       | num
       ;

%%
int yylex(void)
{
   int c;
   while ((c = getchar()) == ' ' || c == '\t');
   if (c == '.' || isdigit(c))
 {
       ungetc(c, stdin);
       scanf("%lf", &yylval);
       return num;
   }
   return c;
}
int yyerror(const char *s)
{
   printf("Error\n", s);
   return 0;
}
int main(void)
{
   printf("Enter arithmetic expressions:\n");
   yyparse();
   return 0;
}
```

Output:-



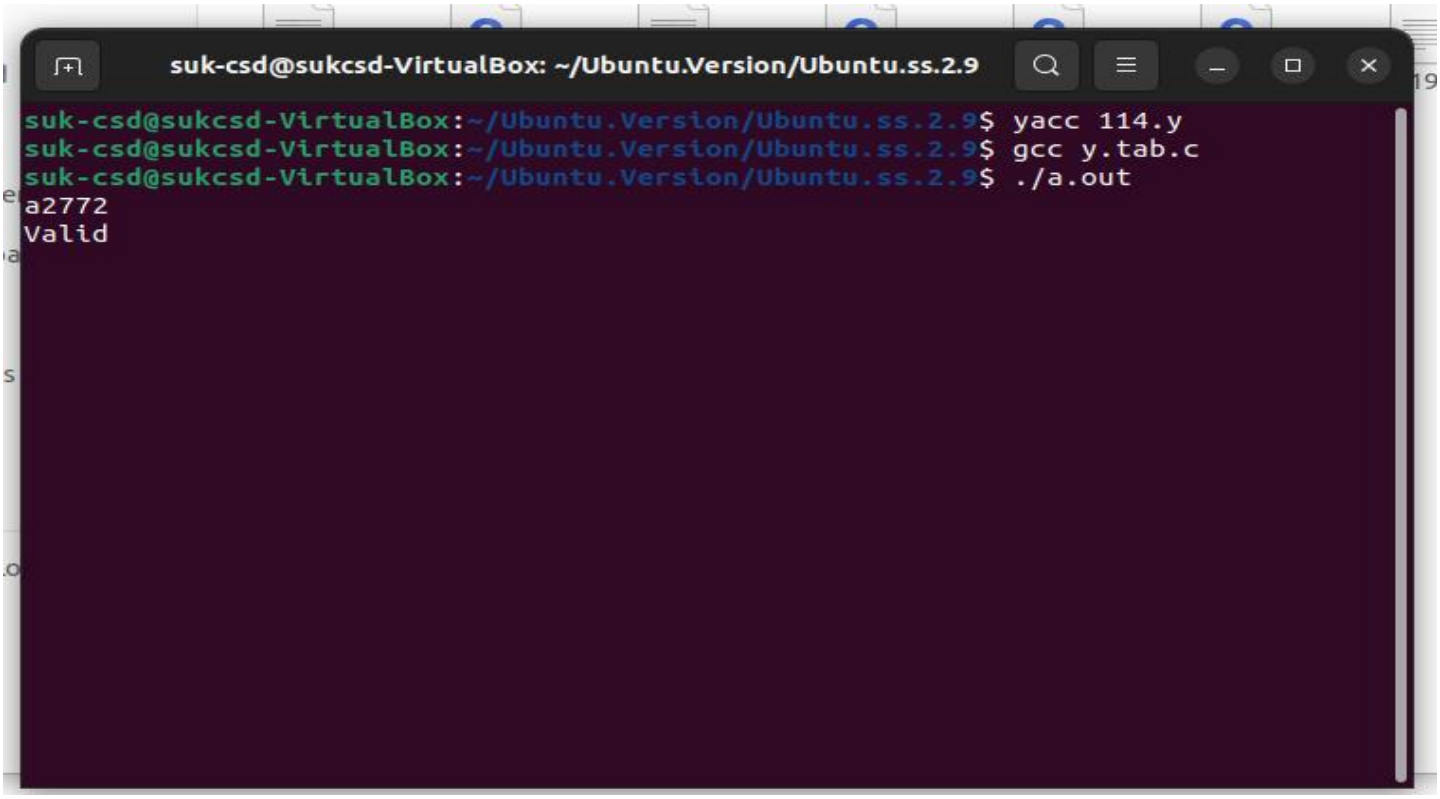
[student@localhost ~]# yacc 3.y
[student@localhost ~]# cc y.tab.c
[student@localhost ~]# ./a.out
4+4
8
[student@localhost~]# ./a.out12+3*(23-
5)
66
[student@localhost ~]# ./a.out
-55+(5*5)
-30

4. Write a YACC Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.

```
%{
#include<stdio.h>
#include<ctype.h>
int yylex(void);
```

```
int yyerror(const char *s);
%}
%token letter digit
%%
id: letter other '\n' { printf("Valid\n"); }
  ;
other: other letter
     | other digit
     | /*empty*/
     ;
%%
int main()
{
   yyparse();
   return 0;
}
int yyerror(const char *s)
{
   printf("Error\n");
   return 0;
}

int yylex(void)
{
   int c;
   while((c=getchar())==' ' || c=='\t');
   if(isalpha(c)) return letter;
   if(isdigit(c)) return digit;
   return c;
}
```

Output:-



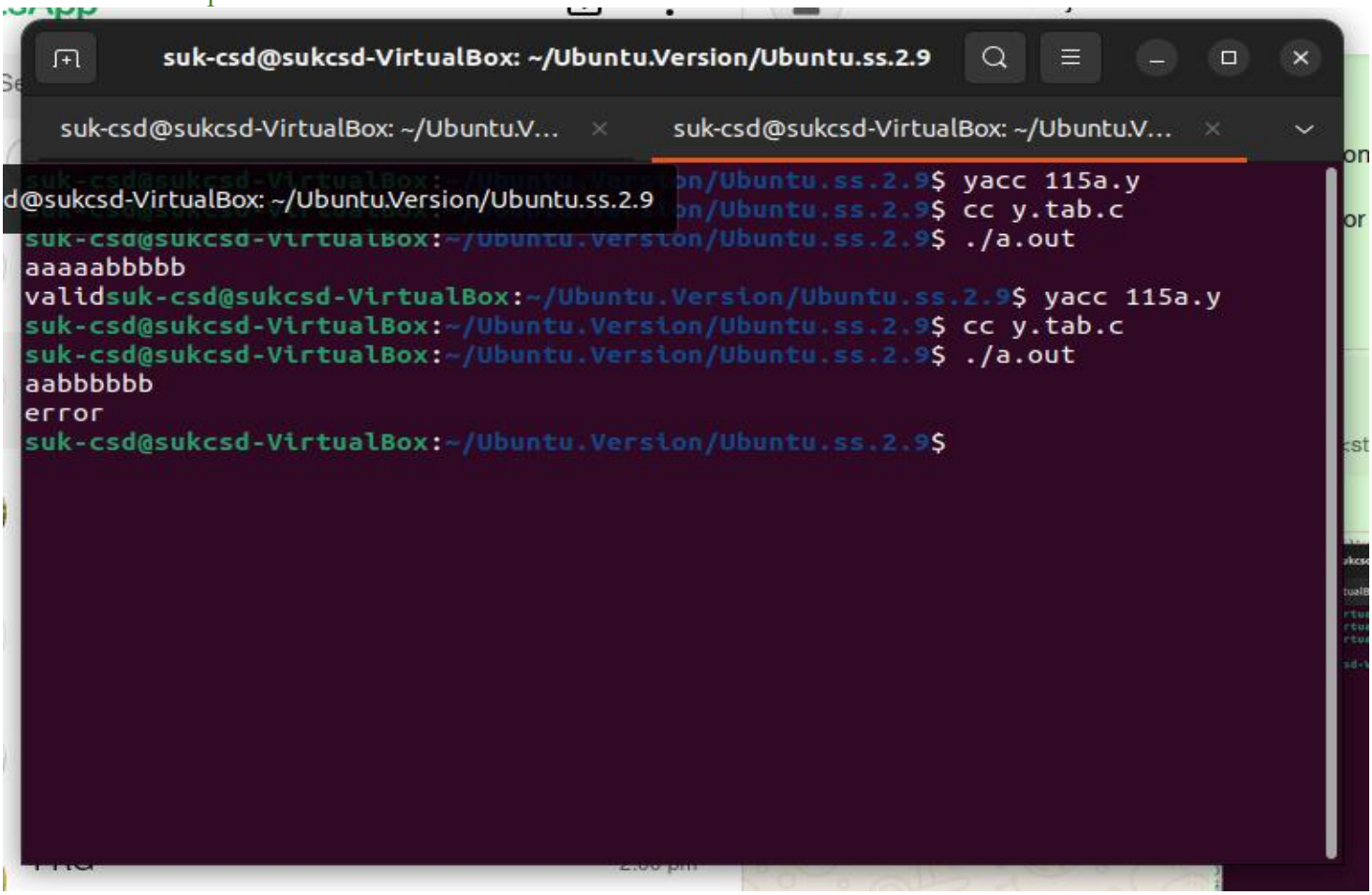
[student@localhost ~]# yacc 4.y
[student@localhost ~]# cc y.tab.c
[student@localhost ~]# ./a.out
A2772
Valid
67676
Error

5. a Write a YACC Program to recognize the grammar ($a^n b$, n>= 10).

```
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
int yylex(void);
int yyerror(const char *s);
%}
%token ta tb
```

```
%%
S  : T '\n'    { printf("valid"); exit(0); }
  ;
T  :
| ta T tb
   ;
%%
int yylex(void)
{
    int c;
    while ((c = getchar()) == ' ' );
    if (c == 'a')
        return ta;
    if (c == 'b')
        return tb;
return c;
}
int main(void)
{
    yyparse();
    return 0;
}
int yyerror(const char *s)
{
    printf("error\n");
    return 0;
}
```

Output:-



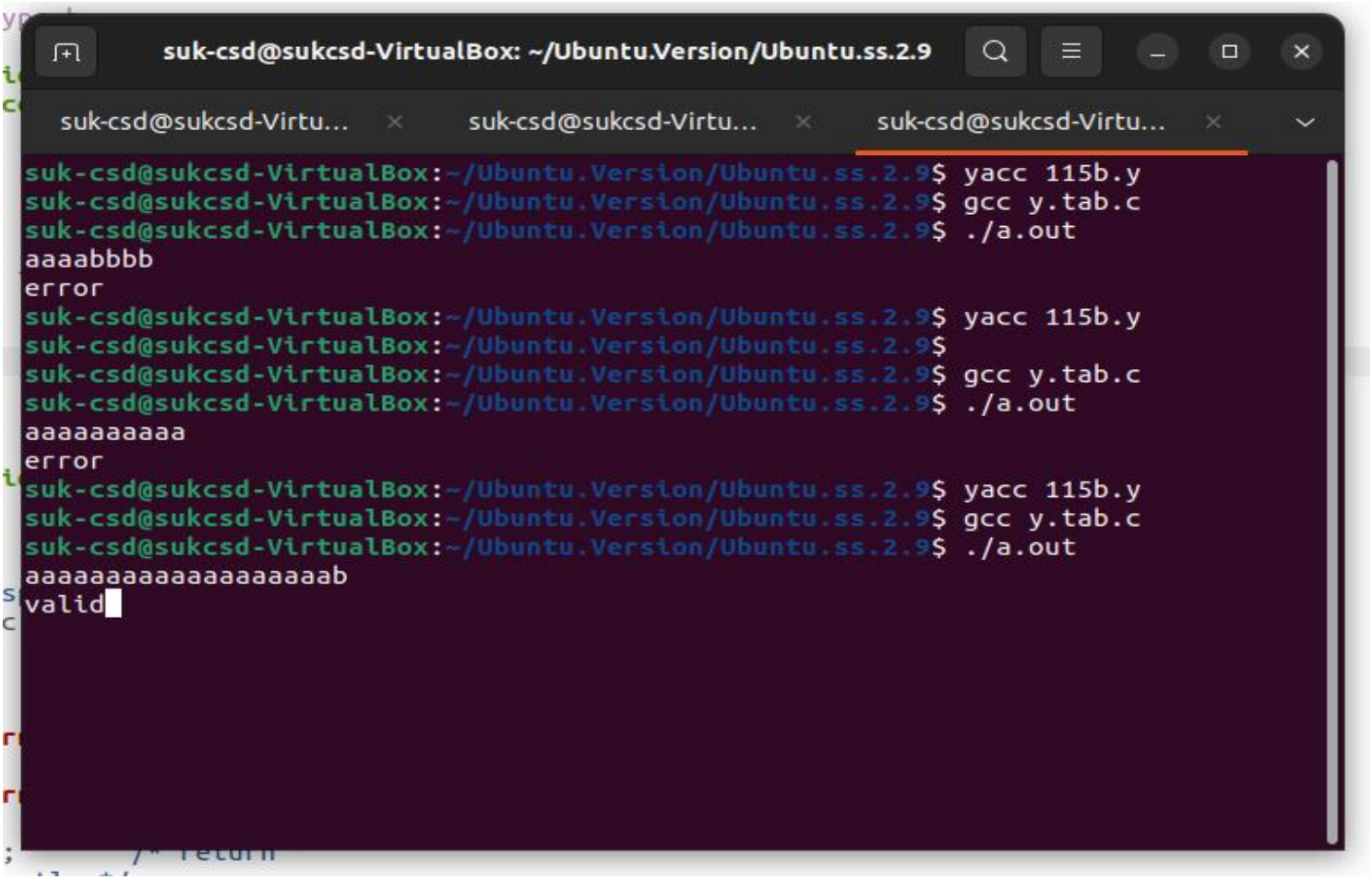
[student@localhost ~]# yacc 5a.y
[student@localhost ~]# cc y.tab.c
[student@localhost ~]# ./a.out
aaaaaaaaaab
Valid
aaaaaaaaaabbbbbb
Error

5. b YACC Program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using the grammar ($a^n b^n$, n>= 0).

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex(void);
int yyerror(const char *s);
%}
```

```
%token ta tb
%%
 V:|ta ta ta ta ta ta ta ta ta p'\n' { printf("valid"); }
;
 p:ta p
 |tb
 ;
%%
int yylex(void)
{
    int c;
    while ((c = getchar()) == ' ')   ;
    if (c == 'a')
        return ta;
    if (c == 'b')
        return tb;
    return c;
}
int main()
{
    yyparse();
    return 0;
}
int yyerror(const char *s)
{
    printf("error\n");
    return 0;
}
```

Output:-



[student@localhost ~]# yacc 5B.y
[student@localhost ~]# cc y.tab.c
[student@localhost ~]# ./a.out
aaab
In valid
[student@localhost ~]# ./a.out
aaaaabbbbb Valid

PART B

6. Design, develop and implement program to construct Predictive / LL(1) Parsing Table for the grammar rules:

A ->aBa ,
B ->bB |Ɛ . Use this table to parse the sentence: abba$

#include <stdio.h>

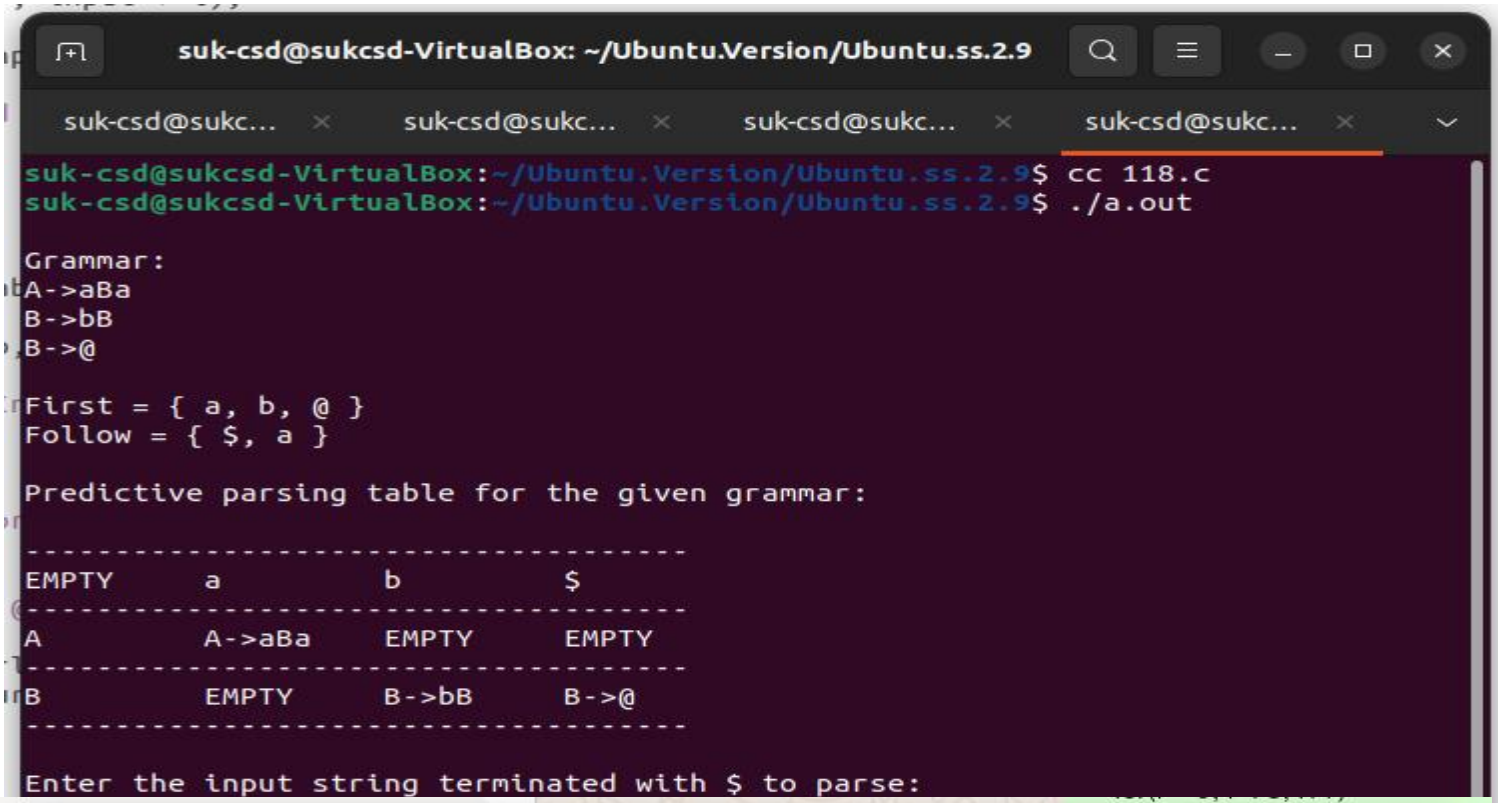
```
#include <stdlib.h>
#include <string.h>

char prod[3][10] = { "A->aBa", "B->bB", "B->@" };
char table[3][4][10];
char stack[20], input[20], curp[20];
int top = -1;

void push(char c)
{
   stack[++top] = c;
}

void pop()
{
   top--;
}

void display()
{
   for(int i = top; i >= 0; i--)
      printf("%c", stack[i]);
}

int col(char c)
{
   if(c == 'a') return 1;
   if(c == 'b') return 2;
   if(c == '$') return 3;
   return 0;
}

int row(char c)
{
```

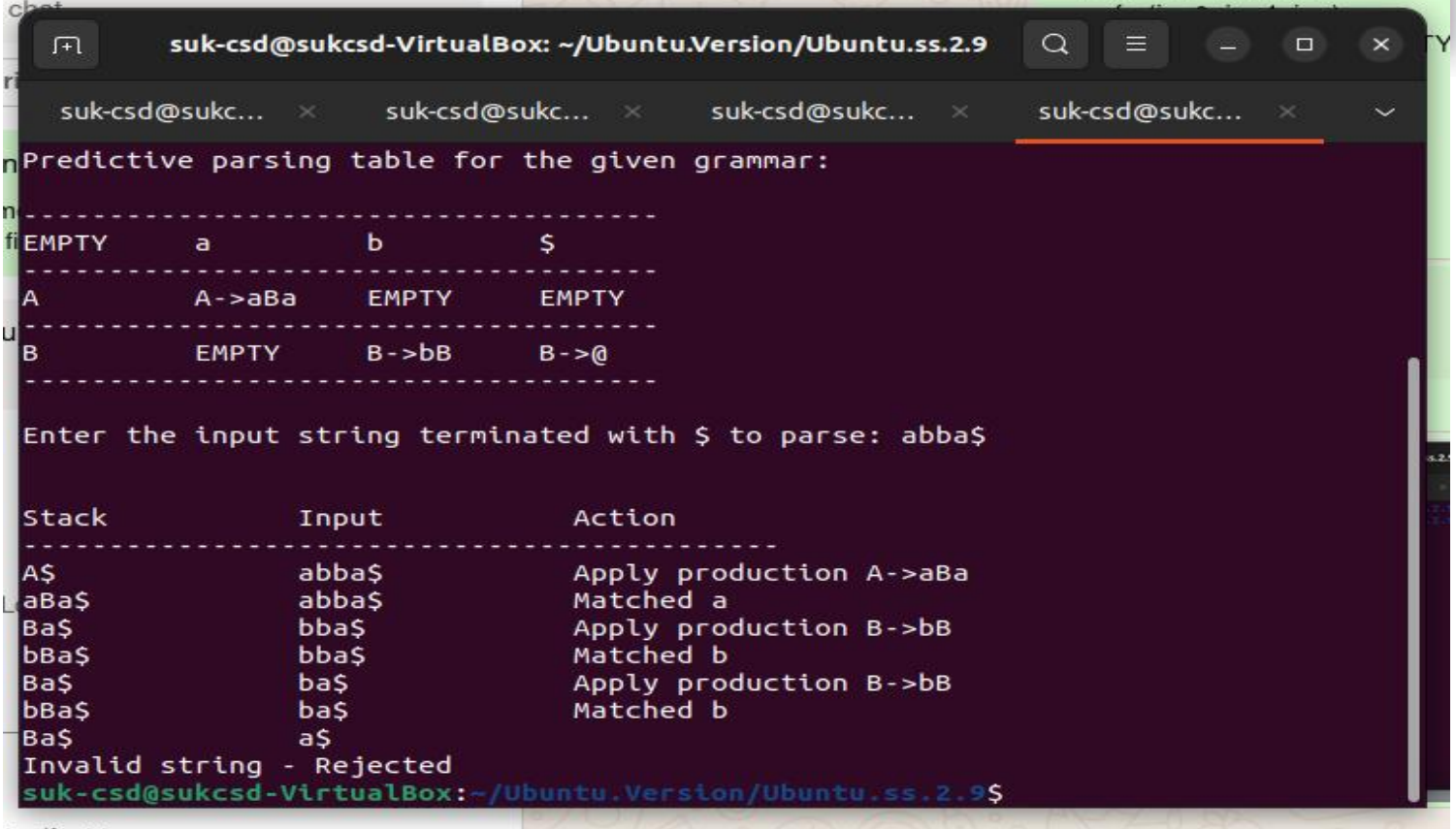
```
    if(c == 'A') return 1;
    if(c == 'B') return 2;
    return 0;
}

int main()
{
    int i, j;

    for(i = 0; i < 3; i++)
        for(j = 0; j < 4; j++)
            strcpy(table[i][j], "EMPTY");

    strcpy(table[0][1], "a");
    strcpy(table[0][2], "b");
    strcpy(table[0][3], "$");

    strcpy(table[1][0], "A");
    strcpy(table[2][0], "B");

    strcpy(table[1][1], "A->aBa");
    strcpy(table[2][2], "B->bB");
    strcpy(table[2][3], "B->@");

    printf("\nGrammar:\n");
    for(i = 0; i < 3; i++)
        printf("%s\n", prod[i]);

    printf("\nFirst = { a, b, @ }");
    printf("\nFollow = { $, a }");

    printf("\n\nPredictive parsing table for the given grammar:\n");
    printf("\n------------------------------------\n");
```

```
for(i = 0; i < 3; i++)
{
   for(j = 0; j < 4; j++)
   {
      printf("%-10s", table[i][j]);
      if(j == 3)
         printf("\n-----------------------------------\n");
   }
}

printf("\nEnter the input string terminated with $ to parse: ");
scanf("%s", input);

push('$');
push('A');
i = 0;

printf("\n\nStack\t\tInput\t\tAction\n");
printf("-----------------------------------------\n");

while(stack[top] != '$')
{
   display();
   printf("\t\t%s\t\t", input + i);

   if(stack[top] == input[i])
   {
      printf("Matched %c\n", input[i]);
      pop();
      i++;
   }
   else
   {
      strcpy(curp, table[row(stack[top])][col(input[i])]);
```

```
        if(!strcmp(curp, "EMPTY"))
        {
           printf("\nInvalid string - Rejected\n");
           exit(0);
        }

        printf("Apply production %s\n", curp);
        pop();

        if(curp[3] != '@')
        {
           for(j = strlen(curp) - 1; j >= 3; j--)
              push(curp[j]);
        }
     }
  }

  if(input[i] == '$')
     printf("\nValid string - Accepted\n");
  else
     printf("\nInvalid string - Rejected\n");
  return 0;
}
```

OUTPUT:


suk-csd@sukcsd-VirtualBox: ~/Ubuntu.Version/Ubuntu.ss.2.9
suk-csd@sukc...
suk-csd@sukc...
suk-csd@sukc...
suk-csd@sukc...
suk-csd@sukcsd-VirtualBox:~/Ubuntu.Version/Ubuntu.ss.2.9$ cc 118.c
suk-csd@sukcsd-VirtualBox:~/Ubuntu.Version/Ubuntu.ss.2.9$ ./a.out
Grammar:
A->aBa
B->bB
B->@
First = { a, b, @ }
Follow = { $, a }
Predictive parsing table for the given grammar:
EMPTY a b $
A A->aBa EMPTY EMPTY
B EMPTY B->bB B->@
Enter the input string terminated with $ to parse:


suk-csd@sukcsd-VirtualBox: ~/Ubuntu.Version/Ubuntu.ss.2.9
suk-csd@sukc...
suk-csd@sukc...
suk-csd@sukc...
suk-csd@sukc...
Predictive parsing table for the given grammar:
EMPTY a b $
A A->aBa EMPTY EMPTY
B EMPTY B->bB B->@
Enter the input string terminated with $ to parse: abba$
Stack Input Action
A$ abba$ Apply production A->aBa
aBa$ abba$ Matched a
Ba$ bba$ Apply production B->bB
bBa$ bba$ Matched b
Ba$ ba$ Apply production B->bB
bBa$ ba$ Matched b
Ba$ a$
Invalid string - Rejected
suk-csd@sukcsd-VirtualBox:~/Ubuntu.Version/Ubuntu.ss.2.9$

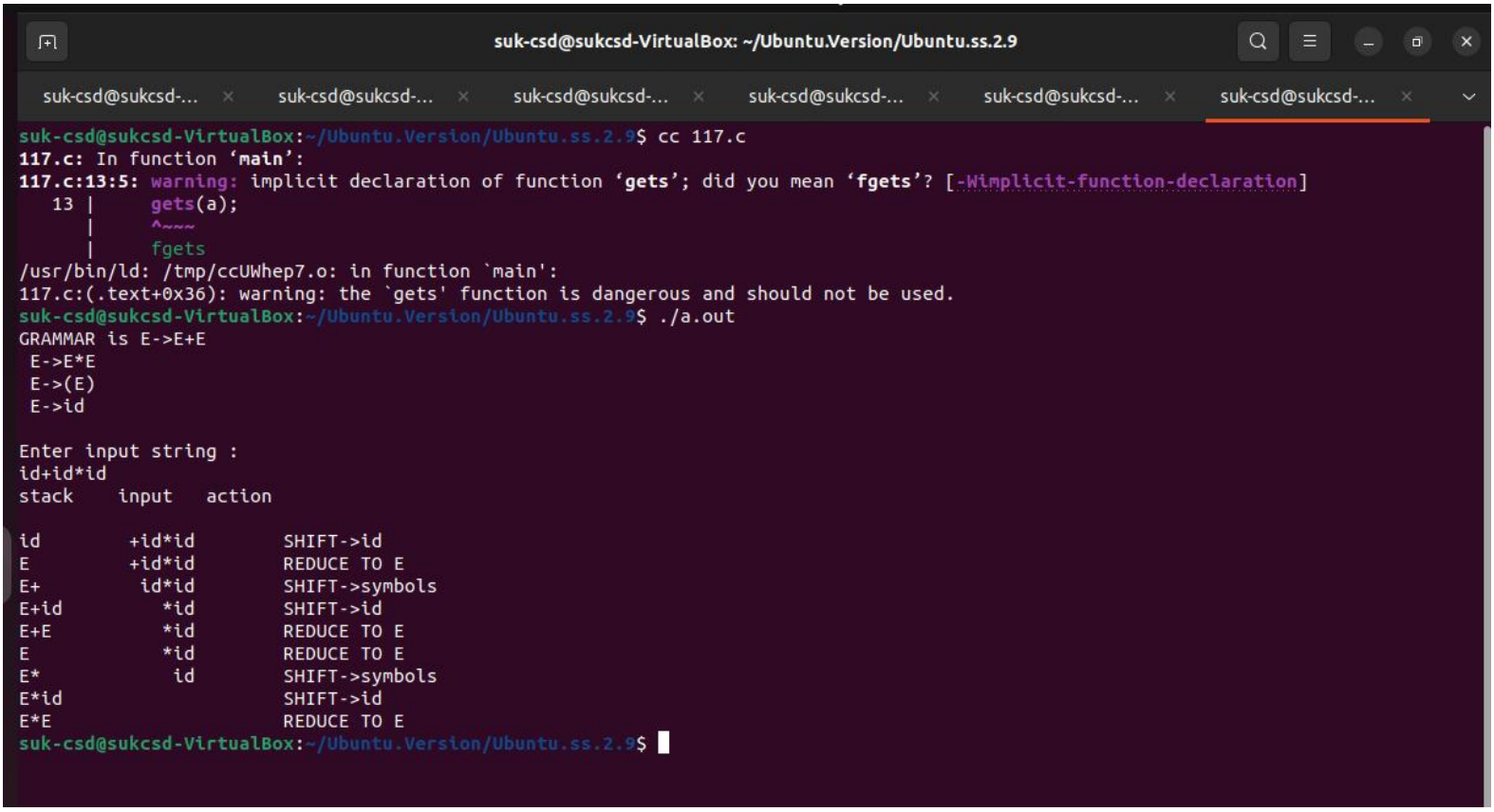
7. Design, develop and implement program to demonstrate Shift Reduce Parsing technique for the grammar rules: $E \rightarrow E + T|TT \rightarrow T * F|F$, F→ $(E)|id$ and parse the sentence: id + id * id.

```
#include <stdio.h>
#include <string.h>
int k = 0, z = 0, i = 0, j = 0, c = 0;
char a[16], ac[20], stk[15], act[10];
void check();
int main()
{
   puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
   puts("\nEnter input string : ");
   gets(a);
   c = strlen(a);
   strcpy(act, "SHIFT->");
   puts("stack \t input \t action");
   for (k = 0, i = 0, j = 0; j < c; k++, i++, j++)
   {
      if (a[j] == 'i' && a[j + 1] == 'd')
      {
         stk[i] = a[j];
         stk[i + 1] = a[j + 1];
         stk[i + 2] = '\0'
         a[j] = ' ';
         a[j + 1] = ' ';
         printf("\n%s\t%s\t%sid", stk, a, act);
         check();
      }
      else
      {
         stk[i] = a[j];
         stk[i + 1] = '\0';
         a[j] = ' ';
         printf("\n%s\t%s\t%ssymbols", stk, a, act);
```

```
            check();
        }
    }
    return 0;
}
void check()
{
    strcpy(ac, "REDUCE TO E")
    for (z = 0; z < c; z++)
    {
        if (stk[z] == 'i' && stk[z + 1] == 'd')
        {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            printf("\n%s\t%s\t%s", stk, a, ac);
            j++;
        }
    }
    for (z = 0; z < c; z++)
    {
        if (stk[z] == 'E' && stk[z + 1] == '+' && stk[z + 2] == 'E')
        {
            stk[z] = 'E';
            stk[z + 1] = '\0';
            stk[z + 2] = '\0';
            printf("\n%s\t%s\t%s", stk, a, ac);
            i = i - 2;
        }

    for (z = 0; z < c; z++)
    {
        if (stk[z] == 'E' && stk[z + 1] == '*' && stk[z + 2] == 'E')
        {
            stk[z] = 'E';
```

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

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

OUTPUT:

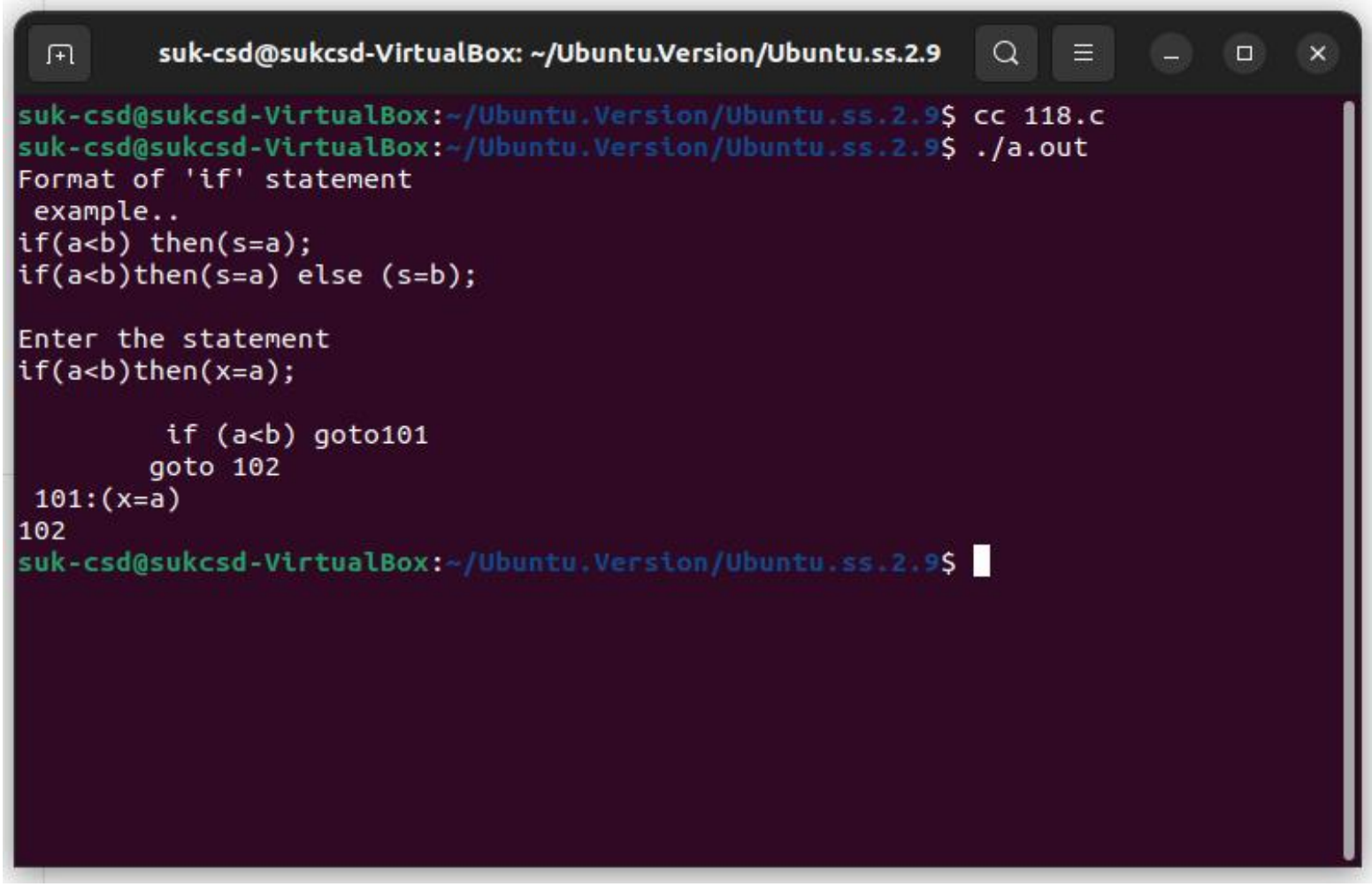


8.Design, develop and implement the syntax-directed definition of "if E then S1" and "if E then S1 else S2"

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

int extract(char s[], int i, char d[]) {
   int k = 0;
   while (s[i] != '(') i++;
   while (s[i] != ')') d[k++] = s[i++];
   d[k++] = ')';
   d[k] = '\0';
   return i;
}

void gen(char B[], char S1[], char S2[], int e) {
   int Bt = 101, Bf = 102, Sn = 103;
   printf("\n\t if %s goto%d", B, Bt);
```

```
    printf("\n\tgoto %d", Bf);
    printf("\n %d:%s", Bt, S1);
    if (!e) printf("\n%d\n", Bf);
    else {
        printf("\n\t goto %d", Sn);
        printf("\n %d:%s", Bf, S2);
        printf("\n%d\n", Sn);
    }
}

int main() {
    char stmt[60], B[30], S1[30], S2[30];
    int i = 2, flag = 0;

    printf("Format of 'if' statement\n example..\n");
    printf("if(a<b) then(s=a);\n");
    printf("if(a<b)then(s=a) else (s=b);\n\n");
    printf("Enter the statement\n");
    scanf("%[^\n]", stmt);

    i = extract(stmt, i, B) + 4;
    i = extract(stmt, i, S1);

    if (stmt[i + 1] == ';') gen(B, S1, S2, flag);
    else {
        flag = 1;
        i += 4;
        extract(stmt, i, S2);
        gen(B, S1, S2, flag);
    }
    return 0;
}
```

Output:



9. Write a yacc program that accepts a regular expression as input and produce its parse tree as output.

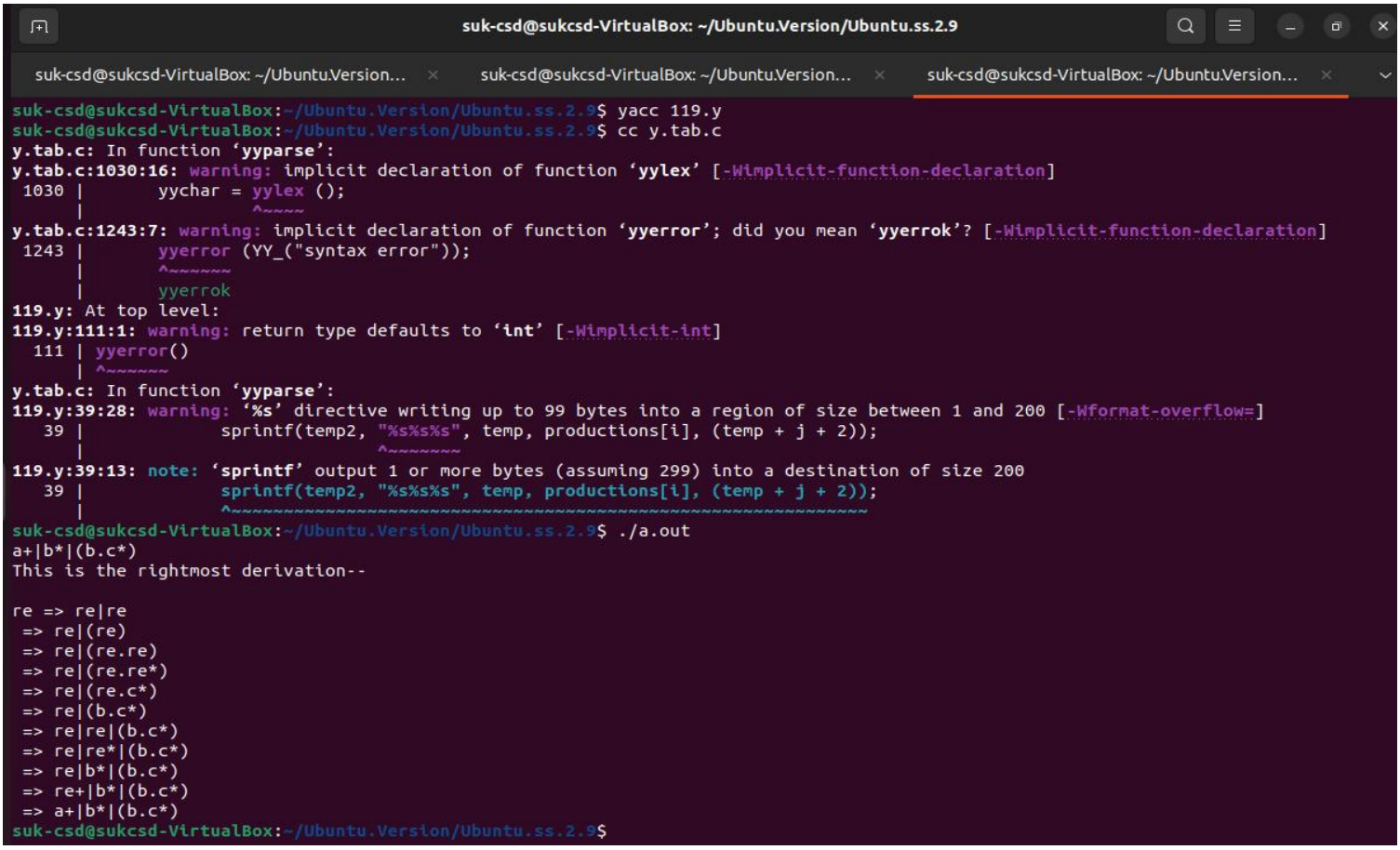
```
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100

int getREindex(const char *);
signed char productions[MAX][MAX];
int count = 0, i, j;
char temp[200], temp2[200];
```

```
%}

%token ALPHABET
%left '|'
%left '.'
%nonassoc '*' '+'

%%
/* rules section */

S : re '\n'
{
   printf("This is the rightmost derivation--\n");
   for (i = count - 1; i >= 0; --i)
   {
      if (i == count - 1)
      {
         printf("\nre => ");
         strcpy(temp, productions[i]);
         printf("%s", productions[i]);
      }
      else
      {
         printf("\n => ");
         j = getREindex(temp);
         temp[j] = '\0';
         sprintf(temp2, "%s%s%s", temp, productions[i], (temp + j + 2));
         printf("%s", temp2);
         strcpy(temp, temp2);
      }
   }
   printf("\n");
   exit(0);
}
;

re : ALPHABET
{
   temp[0] = yylval;
   temp[1] = '\0';
```

```
   strcpy(productions[count++], temp);
}

/* only conditions defined here will be valid, this is the production array */
/* ( re ) adds the (expression) to the production array */
| '(' re ')'
{
   strcpy(productions[count++], "(re)");
}

/* re * */
| re '*'
{
   strcpy(productions[count++], "re*");
}

/* re + */
| re '+'
{
   strcpy(productions[count++], "re+");
}

/* re | re */
| re '|' re
{
   strcpy(productions[count++], "re|re");
}

/* re . re */
| re '.' re
{
   strcpy(productions[count++], "re.re");
}
;
%%

int main(int argc, char **argv)
{
   /*
    Parse and output the rightmost derivation,
```

```
    from which we can get the parse tree
    */
    yyparse();   /* calls the parser */
    return 0;
}

int yylex()
/* calls lex and takes each character as input and feeds ALPHABET */
{
    signed char ch = getchar();
    yylval = ch;

    if (isalpha(ch))
        return ALPHABET;

    return ch;
}

yyerror()
/* Function to alert user of invalid regular expressions */
{
    fprintf(stderr, "Invalid Regular Expression!!\n");
    exit(1);
}

int getREindex(const char *str)
{
    int i = strlen(str) - 1;
    for (; i >= 0; --i)
    {
        if (str[i] == 'e' && str[i - 1] == 'r')
            return i - 1;
    }
}
```

Output


suk-csd@sukcsd-VirtualBox: ~/Ubuntu.Version/Ubuntu.ss.2.9
suk-csd@sukcsd-VirtualBox: ~/Ubuntu.Version...
suk-csd@sukcsd-VirtualBox: ~/Ubuntu.Version...
suk-csd@sukcsd-VirtualBox: ~/Ubuntu.Version...
suk-csd@sukcsd-VirtualBox:~/Ubuntu.Version/Ubuntu.ss.2.9$ yacc 119.y
suk-csd@sukcsd-VirtualBox:~/Ubuntu.Version/Ubuntu.ss.2.9$ cc y.tab.c
y.tab.c: In function 'yyparse':
y.tab.c:1030:16: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
1030 | yychar = yylex ();
y.tab.c:1243:7: warning: implicit declaration of function 'yyerror'; did you mean 'yyerrok'? [-Wimplicit-function-declaration]
1243 | yyerror (YY_("syntax error"));
yyerrok
119.y: At top level:
119.y:111:1: warning: return type defaults to 'int' [-Wimplicit-int]
111 | yyerror()
y.tab.c: In function 'yyparse':
119.y:39:28: warning: '%s' directive writing up to 99 bytes into a region of size between 1 and 200 [-Wformat-overflow=]
39 | sprintf(temp2, "%s%s%s", temp, productions[i], (temp + j + 2));
119.y:39:13: note: 'sprintf' output 1 or more bytes (assuming 299) into a destination of size 200
39 | sprintf(temp2, "%s%s%s", temp, productions[i], (temp + j + 2));
suk-csd@sukcsd-VirtualBox:~/Ubuntu.Version/Ubuntu.ss.2.9$ ./a.out
a+|b*|(b.c*)
This is the rightmost derivation--
re => re|re
=> re|(re)
=> re|(re.re)
=> re|(re.re*)
=> re|(re.c*)
=> re|(b.c*)
=> re|re|(b.c*)
=> re|re*|(b.c*)
=> re|b*|(b.c*)
=> re+|b*|(b.c*)
=> a+|b*|(b.c*)
suk-csd@sukcsd-VirtualBox:~/Ubuntu.Version/Ubuntu.ss.2.9$

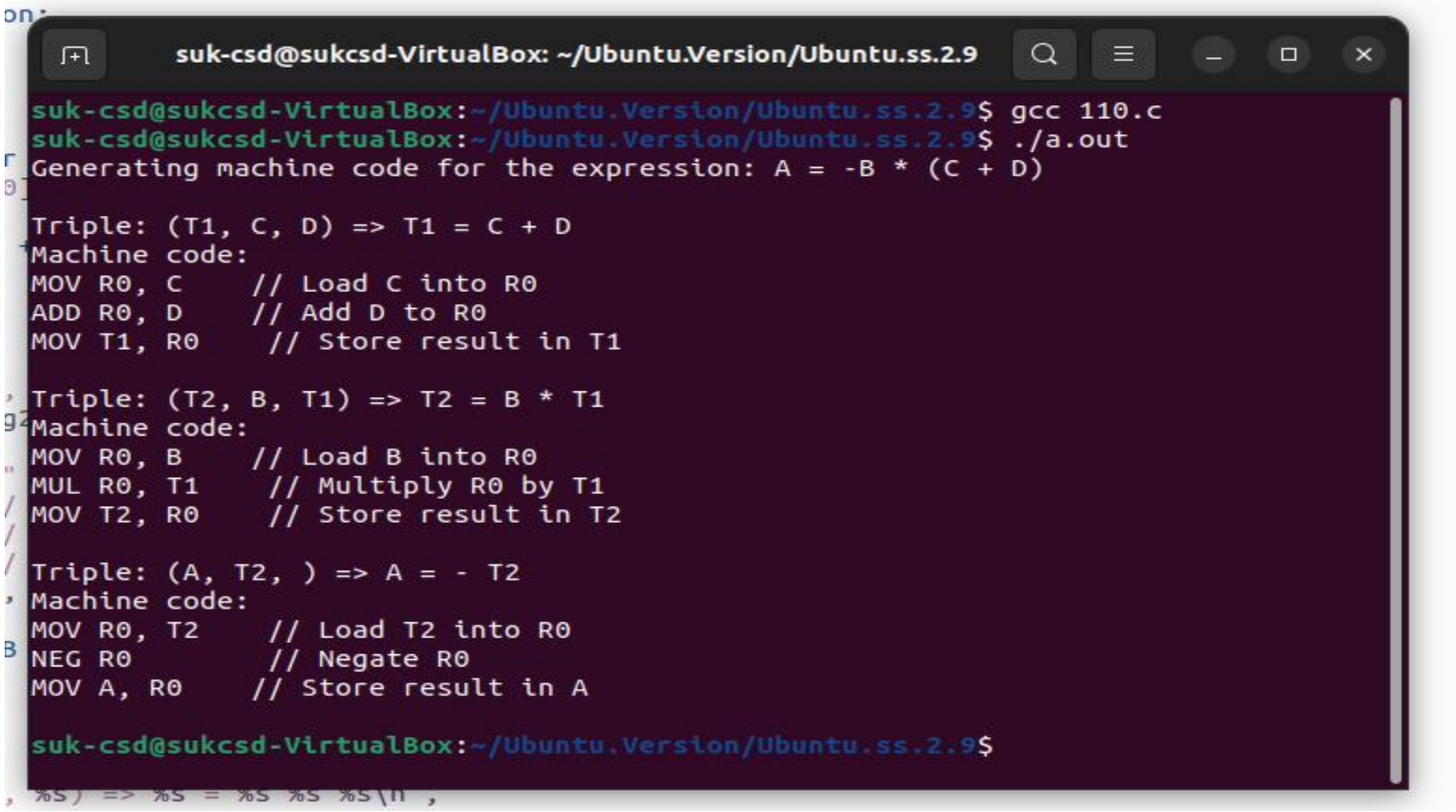
10. Design, develop and implement a program to generate the machine code using Triples for the statement A = -B * (C +D) whose intermediate code in three-address form:

T1 = -B
T2 = C + D
T3 = T1 + T2
A = T3

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

// Function to generate the machine code for the given expression
void generate_machine_code() {

    // Triples representation:
    // T1 = C + D
    // T2 = B * T1
    // A  = -T2

    // Declare variables for the triples
    char result[10], arg1[10], arg2[10], op[3];

    // First triple: T1 = C + D
    sprintf(result, "T1");
    sprintf(arg1, "C");
    sprintf(arg2, "D");
    sprintf(op, "+");
    printf("Triple: (%s, %s, %s) => %s = %s %s %s\n",
           result, arg1, arg2, result, arg1, op, arg2);

    printf("Machine code:\n"
           "MOV R0, %s    // Load %s into R0\n"
           "ADD R0, %s    // Add %s to R0\n"
           "MOV %s, R0    // Store result in %s\n\n",
           arg1, arg1, arg2, arg2, result, result);

    // Second triple: T2 = B * T1
    sprintf(result, "T2");
    sprintf(arg1, "B");
    sprintf(arg2, "T1");
    sprintf(op, "*");
    printf("Triple: (%s, %s, %s) => %s = %s %s %s\n",
           result, arg1, arg2, result, arg1, op, arg2);
```

```
    printf("Machine code:\n"
           "MOV R0, %s    // Load %s into R0\n"
           "MUL R0, %s    // Multiply R0 by %s\n"
           "MOV %s, R0    // Store result in %s\n\n",
           arg1, arg1, arg2, arg2, result, result);

    // Third triple: A = -T2
    sprintf(result, "A");
    sprintf(arg1, "T2");
    sprintf(arg2, " ");
    sprintf(op, "-");
    printf("Triple: (%s, %s, ) => %s = %s %s\n",
           result, arg1, result, op, arg1);

    printf("Machine code:\n"
           "MOV R0, %s    // Load %s into R0\n"
           "NEG R0        // Negate R0\n"
           "MOV %s, R0    // Store result in %s\n\n",
           arg1, arg1, result, result);
}

int main() {
    // Generate the machine code for the expression
    printf("Generating machine code for the expression: A = -B * (C + D)\n\n");
    generate_machine_code();
    return 0;
}
```

OUTPUT:



VIVA QUESTIONS

1. Define system software.
2. What is an Assembler?
3. Explain lex and yacc tools
4. Explain yyleng?
5. What is a Parser?
6. What is the Syntax of a Language?
7. What is the Semantics of a Language?
8. What are tokens?
9. What is the Lexical Analysis?
10. How can we represent a token in a language?
11. How are the tokens recognized?
12. Are Lexical Analysis and Parsing two different Passes?

13. What are the Advantages of using Context-Free grammars?
14. If Context-free grammars can represent every regular expression, why do one needs R.E at all?
15. What is LL(1) parsing?
16. How to parse LL(1) parser
17. How many types of Parsers are there?
18. What is quadruple?
19. What is the difference between Triples and Indirect Triple?
20. State different forms of Three address statements.
21. What are different intermediate code forms?
22. What are the derivation methods to generate a string for the given grammar? What is the output of parse tree?
23. What is a parser and state the Role of it?
24. Types of parsers? Examples to each
25. What are the Tools available for implementation?
26. How do you calculate FIRST(),FOLLOW() sets used in Parsing Table construction?
27. What is are the functions of a Scanner?
28. What are Terminals and non-Terminals in a grammar?
29. What are Ambiguous Grammars?
30. What is bottomup Parsing?