Exercise – 9: Generate the Intermediate code (Quadruples) from the given lexical and syntactical specifaction a programming language using Lex and Yacc tools.

Aim: To develop the semantic routine to generate the intermediate code.

Theory:

The intermediate code bridges the syntax and semantics of the source code with the machine code. Some of the popular intermediate codes in the perspective of compiler are quadruple, triple, indirect triple, postfix expression, syntax tree. The standardized intermediate code will enable the programming languages to be ported to many computer platform. Java byte code and Common Intermediate Language (CIL) from Microsoft are the examples of the standard intermediate code. In order to generate the intermediate code, semantic actions (implemented in an programming languages) must be written to convert the programming language constructs into the appropriate intermediate code. From the intermediate code, the machine code could be generated by mapping the intermediate code to the machine code.

Steps:

1. Identify the language constructs.
2. Write the syntactic specification and present to the parser
    a. (A lex and Yacc based parser may be used)
3. Compile the lexical and syntactic specifcation of the selected language constructs such as arithemetic expression, control statement, declarative statement, etc.
4. Attach the semantic routines for each grammar rule so that the intermediate code could be generated.
5. Read the source program.
6. Parse the program.
7. Convert the source code into intermediate code.

Modules:

- Lexical module
- Syntactic module
- Semantic routines module
- Output module

In this exercise, we will highlight the generation of intermediate code for the arithemetic expession. Only the modules related to the intermediate code generation are highlighted. For complete details refer to **Chapter 6** and **Appendix A.**

The **data structure** for the quadruple is as follows:

```
typedef struct Quad{
        char *label;
        int operator;
        Attr *operand1;
        Attr * operand2;
        Attr *result;
        struct Quad *nextQuad;
}Quad;
```

The prototype declarations for the addition of a quadruple to the intermediate code is as follows:

```
void addCode(Quad *quadListHeader, char *label, int operator,
             Attr operand1, Attr operand2, Attr result);
```

The initialization of the code data structure is done as follows:

```
void createQuadList(Quad **quadListHeader)
{
	*quadListHeader=(Quad *)malloc(sizeof(Quad));
	(*quadListHeader)->label=NULL;
	(*quadListHeader)->operator=-1;
	(*quadListHeader)->operand1=NULL;
	(*quadListHeader)->operand2=NULL;
	(*quadListHeader)->result=NULL;
	(*quadListHeader)->nextQuad=NULL;
}
```

The routines for the implementation of the additon of quadruple is as follows:

```
void	addCode(Quad	*quadListHeader,char	*label,int	operator,char
*operand1,char *operand2,char *result)
{
	Quad *temp,*ptr;
	temp=quadListHeader;
	while(temp->nextQuad!=NULL)
		temp=temp->nextQuad;
	ptr=(Quad*)malloc(sizeof(Quad));
	ptr->label=(char *)malloc(strlen(label)+1);
	ptr->operand1=(char *)malloc(strlen(operand1)+1);
	ptr->operand2=(char *)malloc(strlen(operand2)+1);
	ptr->result=(char *)malloc(strlen(result)+1);
	ptr->nextQuad=NULL;
	strcpy(ptr->label,label);
```

```
	if(strcmp(label," ")!=0)
		strcpy(label," ");
	ptr->operator=operator;
	strcpy(ptr->operand1,operand1);
	strcpy(ptr->operand2,operand2);
	strcpy(ptr->result,result);
	temp->nextQuad=ptr;
}
```

Whenever an expression is encountered, a temporary varialbe is created. The module for creating the tempory variable is as follows:

```
void createTemp(char temp[])
{
	static int i=0;
	char no[5];
	i++;
	itoa(i,no);
	strcpy(temp,"t");
	strcat(temp,no);
}
```

Syntactic specification and semantic actions for the generation of quadurple (intermediate code is as follows:

```
assignStmt: assignExpr _semicolon
	;
Expr  : Expr _plus Expr {
			createTemp($$);
```

```
                addCode(quadTable,labelpending,PLUS,$1,$3,$$);
               }
| Expr _minus Expr{
                createTemp($$);
                addCode(quadTable,labelpending,MINUS,$1,$3,$$);
               }

| Expr _mul Expr {
                createTemp($$);
                addCode(quadTable,labelpending,MUL,$1,$3,$$);
                    }
| Expr _div Expr {
                createTemp($$);
                addCode(quadTable,labelpending,DIV,$1,$3,$$);
              }
| Expr _modulo Expr {
                   createTemp($$);
                   addCode(quadTable,labelpending,MOD,$1,$3,$$);
                 }
| _uminus Expr {
               createTemp($$);
               addCode(quadTable,labelpending,UMINUS,$2," ", $$);
              }
| Expr _lt Expr {
                createTemp($$);
                addCode(quadTable,labelpending,LT,$1,$3,$$);
             }

| Expr _le Expr {
```

```
                createTemp($$);
                addCode(quadTable,labelpending,LE,$1,$3,$$);
             }
| Expr _ge Expr {
                createTemp($$);
                addCode(quadTable,labelpending,GE,$1,$3,$$);
             }
| Expr _gt Expr {
                createTemp($$);
                addCode(quadTable,labelpending,GT,$1,$3,$$);
             }
| Expr _dequal Expr {
                   createTemp($$);
                   addCode(quadTable,labelpending,EQ,$1,$3,$$);
                 }
| Expr _unequal Expr  {
                   createTemp($$);
                   addCode(quadTable,labelpending,NE,$1,$3,$$);
                   }
| Expr _or Expr {
                createTemp($$);
                addCode(quadTable,labelpending,OR,$1,$3,$$);
             }
| Expr _and Expr {
                createTemp($$);
                addCode(quadTable,labelpending,AND,$1,$3,$$);
              }
| _leftp  Expr _rightp {strcpy($$,$2);}
| _id
```

```
            {
                  if(findSymbolHash($1) == NULL)
                  {
                        printf("%s: %d:Error %s: Undeclared
                  Identifier\n",srcFileName,lineNo-1,$1);
                        errCount++;
                  }
                  strcpy($$,$1);
            }
      | _num
            {
                  itoa($1,str);
                  strcpy($$,str);
            }
        ;
```

The routines to display the quadruples is as follows:

```
void printCode(Quad *quadListHeader)
{
      Quad *temp;
      temp=quadListHeader->nextQuad;
      printf(" THE TABLE OF QUADRUPLES ARE\n\n");
      printf("LABEL\tOPER\tOP1\tOP2\tRES\n\n");
      while(temp!=NULL)
      {
            if(strcmp(temp->label," "))
                  printf("%s:\t",temp->label);
```

```
            else
                  printf("\t");
            printf("%s\t%s\t%s\t%s\n",ops[temp->operator],temp->operand1,
      temp->operand2,temp->result);
            temp=temp->nextQuad;
      }
}
```

Output

The input code segment is:

```
noOfYears = 10
interestRate = 0.12
principalAmount = 1000
totalAmount =  principalAmount + interestRate * noOfYears
```

The set of quadruples generated are:

```
LABEL OPER  OP1   OP2   RES

      =     10          noOfYears
      =     0.12        interestRate
      =     1000        principalAmount
      *     interestRate      noOfYears   t1
      +     principalAmount   t1    t2
      =     t2          totalAmount
```

Exericses for the students:

Write down the semantic routines for generating the intermediate code (quadruple) for a Boolean expression.