Code:

main.c

```
#include "code_gen.c"
#include <stdio.h>
int main ()
{
    mult_stmt();
}
```

code_gen.c

```
#include <stdio.h>
#include <stdlib.h>
#include <error.h>
#include <string.h>
#include <ctype.h>
#include "lex.h"
#include "lex.c"
#include "name.c"

char    *factor     ( void );
char    *term       ( void );
char    *expression ( void );
void    statement   ( void);
void     mult_stmt  ( void);
void    stmt_list   ( void);
char    *exp1       ( void);
char * reg[6] = {"BL", "BH", "CL", "CH", "DH", "DL"};
char  variables[100][10] = {{'1','1','1','1','1','1','1','1','1','1','\0'}};
int cur_variable = 0;
int ifThen=0, comp=0, loop=0;

void insert_variable(char * var)
{
    int i=0;
    for(i=0; i<cur_variable; i++)
    {
        if(strcmp(var, variables[i])==0)        // check variable exist or not
            return;
    }
```

```
    if(cur_variable == 100)
        return;

    strcpy(variables[cur_variable++], var);     // insert new variable in array
    return;
}

void mult_stmt(){

   int i=0;
   printf("ORG 100h\n");
   while( ! match(EOI) ){
       statement();
       if( match (SEMI) ){
           advance();
       }else{
           fprintf( stderr, "%d: Insert missing semicolon (mult_stmt)\n", yylineno);
           exit(1);
       }
   }
   printf("RET\n");

    for(i=0; i<cur_variable; i++){
        printf("_%s DB ?\n", variables[i]);
    }
    return;
}

void statement()
{
   /*  statements -> expression SEMI  |  expression SEMI statements  */
   char *tempvar;

   if( match( ID )){
       char var[50];
       int id=0;
       while(id<yyleng){
           var[id]=*(yytext+id);
           id++;
       }
       var[yyleng] = '\0';
       advance();
       if(match( ASSIGN )){
           advance();
           tempvar = exp1();
           insert_variable(var);
           printf("MOV _%s, %s\n", var, reg[tempvar[1] - '0']);
           freename(tempvar);
```

```
        }else{
            fprintf( stderr, "%d: Inserting missing assign symbol (statement)\n",
yylineno );
            exit(1);
        }
    }else if( match(IF) ){
        advance();
        tempvar = exp1();
        if( match( THEN )){
            advance();
            int label = ifThen++;
            printf("CMP %s, 0\n", reg[tempvar[1]-'0']);
            printf("JZ ifThen%d\n", label);
            freename(tempvar);
            statement();
            printf("ifThen%d:\n", label);
        }else{
            fprintf( stderr, "%d: Inserting missing then (statement)\n", yylineno );
            exit(1);
        }
    }else if( match(WHILE) ){
        advance();
        int label = loop++;
        printf("loopA%d:\n", label);
        tempvar = exp1();
        if( match( DO )){
            advance();
            printf("CMP %s, 0\n", reg[tempvar[1]-'0']);
            printf("JZ loopB%d\n", label);
            freename(tempvar);
            statement();
            printf("JMP loopA%d\n", label);
            printf("loopB%d:\n", label);
        }else{
            fprintf( stderr, "%d: Inserting missing do (statement)\n", yylineno );
            exit(1);
        }
    }else if( match(BEGIN) ){
        advance();
        stmt_list();
        if( match( END )){
            advance();
        }else{
            fprintf( stderr, "%d: Inserting missing end (statement)\n", yylineno );
            exit(1);
        }
    }else{
        fprintf( stderr, "%d: Inserting missing statement (statement)\n", yylineno );
```

```
        exit(1);
    }
}


char *exp1(){
    // printf("exp1\n");
    char * tempvar;
    char * tempvar1;
    char * tempvar2;
    tempvar = expression();
    if( match(EQ) ){
        advance();
        tempvar1 = expression();
        tempvar2 = newname();
        int label = comp++;
        printf("CMP %s, %s\n", reg[tempvar[1]-'0'], reg[tempvar1[1]-'0']);
        printf("MOV %s, 1\n", reg[tempvar2[1]-'0']);
        printf("JZ COMP%d\n", label);
        printf("MOV %s, 0\n", reg[tempvar2[1]-'0']);
        printf("COMP%d:\n", label);
        freename(tempvar);
        freename(tempvar1);
        return tempvar2;
    }else if( match(LT) ){
        advance();
        tempvar1 = expression();
        tempvar2 = newname();
        int label = comp++;
        printf("CMP %s, %s\n", reg[tempvar[1]-'0'], reg[tempvar1[1]-'0']);
        printf("MOV %s, 1\n", reg[tempvar2[1]-'0']);
        printf("JC COMP%d\n", label);
        printf("MOV %s, 0\n", reg[tempvar2[1]-'0']);
        printf("COMP%d:\n", label);
        freename(tempvar);
        freename(tempvar1);
        return tempvar2;
    }else if( match(GT) ){
        advance();
        tempvar1 = expression();
        tempvar2 = newname();
        int label = comp++;
        printf("CMP %s, %s\n", reg[tempvar[1]-'0'], reg[tempvar1[1]-'0']);
        printf("MOV %s, 0\n", reg[tempvar2[1]-'0']);
        printf("JC COMP%d\n", label);
        printf("JZ COMP%d\n", label);
        printf("MOV %s, 1\n", reg[tempvar2[1]-'0']);
        printf("COMP%d:\n", label);
```

```
        freename(tempvar);
        freename(tempvar1);
        return tempvar2;
    }
    return tempvar;
}

void stmt_list(){
    // printf("stmt_list\n");
    while( ! match(END) ){
        if( ! match(EOI) ){
            statement();
            if( match(SEMI) ){
                advance();
            }else{
                fprintf( stderr, "%d: Insert missing semicolon (stmt_list)\n",
yylineno);
                exit(1);
            }
        }else{
            fprintf( stderr, "%d: Insert missing end (stmt_list)\n", yylineno);
            exit(1);
        }
    }
}


char    *expression()
{
    /* expression -> term expression'
     * expression' -> PLUS term expression' |  epsilon
     */

    char  *tempvar, *tempvar1;

    tempvar = term();
    // printf("%.*s\n", yyleng, yytext);
    while( match( PLUS ) || match( MINUS ))
    {
        if(match( PLUS )){
            advance();
            tempvar1 = term();
            printf("ADD %s, %s\n", reg[tempvar[1] - '0'], reg[tempvar1[1] - '0']);
            freename( tempvar1 );
        }else if(match( MINUS )){
            advance();
            tempvar1 = term();
            printf("SUB %s, %s\n", reg[tempvar[1] - '0'], reg[tempvar1[1] - '0']);
```

```
            freename( tempvar1 );
        }
    }
    return tempvar;
}

char    *term()
{
    char  *tempvar, *tempvar1;

    tempvar = factor();
    while( match( TIMES ) || match( DIV ) )
    {
        if(match(TIMES)){
            advance();
            tempvar1 = factor();
            printf("MOV AL, %s\n", reg[tempvar[1]-'0']);
            printf("MUL %s\n", reg[tempvar1[1]-'0']);
            printf("MOV %s, AL\n", reg[tempvar[1]-'0']);
            freename(tempvar1);
        }else if(match(DIV)){
            advance();
            tempvar1 = factor();
            printf("MOV AL, %s\n", reg[tempvar[1]-'0']);
            printf("DIV %s\n", reg[tempvar1[1]-'0']);
            printf("MOV AH, 0\n");
            printf("MOV %s, AL\n", reg[tempvar[1]-'0']);
            freename(tempvar1);
        }
    }
    return tempvar;
}

char    *factor()
{
    char *tempvar;

    if( match(NUM) || match (ID) )
    {
     /* Print the assignment instruction. The %0.*s conversion is a form of
        * %X.Ys, where X is the field width and Y is the maximum number of
        * characters that will be printed (even if the string is longer). I'm
        * using the %0.*s to print the string because it's not \0 terminated.
        * The field has a default width of 0, but it will grow the size needed
        * to print the string. The ".*" tells printf() to take the maximum-
        * number-of-characters count from the next argument (yyleng).
        */
```

```
        // printf("    %s = %0.*s\n", tempvar = newname(), yyleng, yytext );
        char var[50];
        int id=0;
        while(id<yyleng){
            var[id]=*(yytext+id);
            id++;
        }
        var[yyleng] = '\0';
        if(match(NUM)){
            advance();
            tempvar = newname();
            printf("MOV %s, %s\n", reg[tempvar[1]-'0'], var);
            return tempvar;
        }else if(match(ID)){
            advance();
            tempvar = newname();
            insert_variable(var);
            printf("MOV %s, _%s\n", reg[tempvar[1]-'0'], var);
            return tempvar;
        }
    }
    else if( match(LP) )
    {
        advance();
        tempvar = exp1();
        if( match(RP) ){
            advance();
            return tempvar;
        }
        else{
            fprintf(stderr, "%d: Mismatched parenthesis\n", yylineno );
            exit(1);
        }
    }
    else{
          fprintf( stderr, "%d: Number or identifier expected\n", yylineno );
       exit(1);
    }
}
```

lex.h

```
#define EOI     0   /* End of input         */
#define SEMI        1   /* ;                */
#define PLUS        2   /* +                */
#define MINUS       3   /* -                */
#define TIMES       4   /* *                */
#define DIV         5   /* /                */
#define LP      6   /* (                */
#define RP      7   /* )                */
#define NUM     8   /* Decimal Number or Identifier */
#define ID          9
#define IF          10
#define THEN        11
#define WHILE       12
#define DO          13
#define LT          14
#define GT          15
#define EQ          16
#define ASSIGN          17
#define END         18
#define BEGIN         19

extern char *yytext;        /* in lex.c         */
extern int yyleng;
extern int yylineno;
```

lex.c

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

char* yytext = ""; /* Lexeme (not '\0'
                      terminated)              */
int yyleng   = 0;  /* Lexeme length.           */
int yylineno = 0;  /* Input line number        */

int lex(void){

  static char input_buffer[1024];
  char        *current;

  current = yytext + yyleng; /* Skip current
                                lexeme        */
```

```
 // printf("%s\n", current);

while(1){       /* Get the next one         */
   while(!*current ){
      /* Get new lines, skipping any leading
      * white space on the line,
      * until a nonblank line is found.
      */

      current = input_buffer;
      if(!gets(input_buffer)){
         *current = '\0' ;

         return EOI;
      }
      ++yylineno;
      while(isspace(*current))
         ++current;
   }
   for(; *current; ++current){
      /* Get the next token */
      yytext = current;
      yyleng = 1;
      switch( *current ){
        case ';':
         return SEMI;
        case '+':
         return PLUS;
        case '-':
         return MINUS;
        case '*':
         return TIMES;
        case '/':
         return DIV;
        case '(':
         return LP;
        case ')':
         return RP;
        case '=':
         return EQ;
        case '<':
         return LT;
        case '>':
         return GT;
        case '\n':
        case '\t':
        case ' ' :
         break;
```

```
  default:
   if(*current == ':'){
     ++current;
     ++current;
     yyleng = current - yytext;
     return ASSIGN;
   }
   if(!isalnum(*current))
            fprintf(stderr, "Not alphanumeric <%c>\n", *current);
          else{
            while(isalnum(*current))
                ++current;
            yyleng = current - yytext;
            char subbuff[yyleng+1];
            memcpy( subbuff, yytext, yyleng );
            subbuff[yyleng] = '\0';
            if(strcmp(subbuff, "if") == 0)
            {
                return IF;
            }
            else if(strcmp(subbuff, "then") == 0)
            {
                return THEN;
            }
            else if(strcmp(subbuff, "while") == 0)
            {
                return WHILE;
            }
            else if(strcmp(subbuff, "do") == 0)
            {
                return DO;
            }
            else if(strcmp(subbuff, "begin") == 0)
            {
                return BEGIN;
            }
            else if(strcmp(subbuff, "end") == 0)
            {
                return END;
            }
    else if(isdigit(subbuff[0]))
    {
      return NUM;
    }
            return ID;
          }
     break;
}
```

```
    }
  }
}


static int Lookahead = -1; /* Lookahead token  */

int match(int token){
  /* Return true if "token" matches the
     current lookahead symbol.                */

  if(Lookahead == -1){
     Lookahead = lex();
     // printf("%d\n", Lookahead);
  }

  return token == Lookahead;
}

void advance(void){
/* Advance the lookahead to the next
  input symbol.                               */

   Lookahead = lex();
   // printf("%d\n", Lookahead);
}
```

name.c

```
#include <stdio.h>
#include <stdlib.h>
#include "lex.h"
char  *Names[] = { "t0", "t1", "t2", "t3", "t4", "t5" };
char  **Namep  = Names;

char  *newname()
{
   if( Namep >= &Names[ sizeof(Names)/sizeof(*Names) ] )
   {
       fprintf( stderr, "%d: Expression too complex\n", yylineno );
       exit( 1 );
   }

   return( *Namep++ );
```

```
}

void freename(s)
char    *s;
{
   if( Namep > Names )
   *--Namep = s;
   else
   fprintf(stderr, "%d: (Internal error) Name stack underflow\n",
                               yylineno );
}
```

Grammer

```
mult_stmt ->            statement SEMI mult_stmt |
                        EOF

statement ->            ID ASSIGN exp1 |
                        IF exp1 THEN statement |
                        WHILE exp1 DO statement |
                        BEGIN stmt_list END

exp1 ->                 expression |
                        expression EQ expression |
                        expression LT expression |
                        expression GT expression

stmt_list ->            statement SEMI stmt_list |
                        epsilon


expression ->           term expression'
expression' ->          PLUS term expression' |
                        MINUS term expression' |
                        epsilon

term ->                 factor term'
term' ->                TIMES factor term' |
                        DIV factor term' |
                        epsilon

factor ->               NUM or ID |
                        LP exp1 RP
```

Expression tree of a+b*c+4