final.y

```
%{
    #include<stdio.h>
    #include<math.h>
    #include<string.h>
    #include<stdlib.h>
    #include <ctype.h>
    #include "prototype_list.h"
    #include "constant.h"
    #include "var_list.h"
    #include<windows.h>


    extern char lastDataType[10];

    int yylex(void);
    void yyerror(char *s);

    struct PARAMETER *paramHead = NULL;
    struct PARAMETER *paramTail = NULL;

    struct PARAMETER *callParamHead = NULL;
    struct PARAMETER *callParamTail = NULL;

    extern char *outBuffer;
    extern int outBufferSize;
    extern bool *validityList;
    extern bool isLastIfValid;

    struct COND{ char *left, *op, *right; };
    struct ASSIGN{ char *var, *left, *op, *right; };

    struct COND lastCond;
    struct ASSIGN lastAssign;

    struct FUNC_RESULT {
        char type[10];
        double res;
    } lastFuncRes;


    char* removeRedundant(char *source){
        int len = strlen(source);

        char temp[len+1];
        int j = 0;
        for(int i=0; i<len; i++){
            //printf("%d, ", source[i]);
            if( source[i] == ' ' ||
                source[i] == ',' ||
                source[i] == ';' ||
                source[i] == '\t' ||
                source[i] == '\n' ||
                source[i] == 32
            )continue;

            temp[j] = source[i];
            j++;
        }

        temp[j] = '\0';
        char *dest;
        dest = malloc(j*sizeof(char));
        strcpy(dest,temp);
        //printf("\n%s\n",dest);
        return dest;
    }

    void printAndClearOutBuffer(){
        printf("\n%s < - - - - - - - - - -\n",outBuffer);
        free(outBuffer);
        outBufferSize = 0;
    }

    void continueOutBuffer(char *vc){
        int i=0, len = strlen(vc) - 1;
        while( i <= len && vc[i] == ' ') i++; // leading space
        while( len>=0 && vc[len] == ' ') len--; // trailing space
after "
        while( len>=0 && vc[len] == ',' || vc[len]=='\"') len--; //
space

        char *res;

        if(vc[i] == '\"'){ // constant
```

```
            i++; // trailing quotaion bad dilam

            int size = len-i+1;
            res = (char *)malloc(size+1);
            strncpy(res, vc + i, size);
            res[size] = '\0';
        }
        else{ // variable
            vc = removeRedundant(vc);
            struct VARIABLE *var = getVariable(vc);

            if( !doesVariableExists(vc)) {
                printf("\nVariable %s doesn't exist\n",vc);
            }
            char* val = getFormattedValueOrDefault(vc);
            res = (char *) malloc(20);
            sprintf(res, "%s ", val);
        }

        outBufferSize += strlen(res);
        outBuffer = (char *)realloc(outBuffer,outBufferSize);

        strcat(outBuffer,res);
    }


    void initProto(char *paramType){
        //printf("type found: %s\n",paramType);
        // -1 default value, not needed for proto-type
        insertParameter(&paramHead, &paramTail, paramType,-1);
    }

    void processProto(char *funcType, char *funcName){
        //printf("Inside\n");

        struct PROTOTYPE* proto = createProto(funcType, funcName, "
user_defined", paramHead,paramTail);
        //printf("Inside-2\n");

        paramHead = NULL; paramTail = NULL;

        if( doesProtoExists(proto,false) ){
            printf("\nDuplicate proto-type found\n");
            return;
        }

        insertProto(proto);
        printf("\nPrototype inserted: ");
        printProto(proto,true);

    }


    double doArithOperation(double val1, double val2, char* op){
        //printf("%lf %lf %s ---------kuttar baccha-\n",val1,val2,op);
        return getValue(val1, val2, op);
    }

    bool addVariable(char* name,char *type){
        if( doesVariableExists(name) ){
            printf("\nVariable %s is already defined\n",name);
            return false;
        }
        insertVariable(name,type,0);
        return true;
    }

    void declareVariable(char *name, char *type, double val){
        if( addVariable(name, type) ){
            updateVariable(name,val);
        }
    }

    void assignIfPossible(char *name, double val){

        if( !doesVariableExists(name) ){
            printf("\nVariable %s doesn't exist\n",name);
            return;
        }

        updateVariable(name,val);

    }

    void discardVariable(char *name){
        if( !doesVariableExists(name) ){
```

```
            printf("\nCan't discard %s since not found.\n",name);
            return;
        }
        printf("\nDiscarded variable %s\n",name);
        deleteVariable(name);
    }

    bool isNumber(char* num){
        bool count = 0;
        int i = num[0] == '-' ? 1 : 0;

        for(; i<strlen(num); i++){
            if( num[i] == '.' && count == 0 ) { count=1; continue; }
            if( !isdigit(num[i]) ) return false;
        }
        return true;
    }

    bool evaluateCondition(char* leftChar, char* op, char *rightChar){

        lastCond.left = leftChar;
        lastCond.op = op;
        lastCond.right = rightChar;

        double left = 0, right = 0;

        if( isNumber(leftChar) ){
            left = strtod(leftChar,NULL);
        }
        else{
            if( !doesVariableExists(leftChar) ){
                printf("\nVariable %s doesn't exist. Using default value(0)\n",leftChar);
            }
            left = getValueOrDefault(leftChar);
        }


        if( isNumber(rightChar) ){
            right = strtod(rightChar,NULL);
        }
        else{
            if( !doesVariableExists(rightChar) ){
                printf("\nVariable %s doesn't exist. Using default value(0)\n",rightChar);
            }
            right = getValueOrDefault(rightChar);
        }

        return isConditionValid(left, op, right);
    }

    void updateLast(char *left, char *op, char *right){
        lastAssign.left = left;
        lastAssign.op = op;
        lastAssign.right = right;
    }

    char* dtoc(double val){
        char* buf = malloc( sizeof(char) * 20 );
        sprintf(buf, "%lf", val);
        return buf;
    }

    void startLoop(){
        int counter = 0;
        printf("\n");
        while( evaluateCondition( lastCond.left, lastCond.op, lastCond.right ) ){

            printf("Iterating - %d",counter);

            if( !isNumber(lastCond.left) ){
                printf(", value: %s", getFormattedValueOrDefault(lastCond.left) );
            }
            printf("\n");

            double val1 = 0, val2 = 0;
            if( isNumber(lastAssign.left) ){
                sscanf(lastAssign.left, "%lf", &val1);
            }
            else{
                val1 = getValueOrDefault(lastAssign.left);
            }

            if( isNumber(lastAssign.right) ){
```

```
                sscanf(lastAssign.right, "%lf", &val2);
            }
            else{
                val2 = getValueOrDefault(lastAssign.right);
            }

            if( !doesVariableExists(lastAssign.var) ){
                printf("\nVariable %s doesn't exists. Stopping...\n",
lastAssign.var);
                break;
            }
            double res = doArithOperation(val1, val2, lastAssign.op);

            updateVariable( lastAssign.var, res );
            counter++;
            Sleep(200);
        }
    }

    void addTypeToCall(char *name){
        //printf("Type call %s\n",name);

        char *type = "double";

        double val = 0;

        if( isNumber(name) ){
            type = "double";
            val = strtod(name,NULL);
        }
        else{

            if( doesVariableExists(name) ){
                struct VARIABLE* var = getVariable(name);
                type = var->type;
                val = var->value;
            }
            else{
                printf("\nVariable %s doesn't exists\n",name);
            }

        }

        insertParameter(&callParamHead, &callParamTail, type,val);

    }


    double getOrDefaultResultFromFunction(
        char* name, struct PARAMETER* head,
        struct PROTOTYPE* funcToCall )
    {

        return getLibrayFunctionResult(name,head);
    }

    void processCall(char *funcName){

        char realName[ strlen(funcName) ];

        // Copy the string starting from the second character
        strcpy(realName, funcName + 1);

        struct PROTOTYPE* temp = createProto("void",realName, "NULL",
callParamHead,NULL);

        strcpy( lastFuncRes.type, "null" );
        lastFuncRes.res = 0;


        if( !doesProtoExists(temp,false) ){

            // check library function
            if( doesProtoExists(temp,true) ){
                // returns from library function
                struct PROTOTYPE* orig = getOriginalProto(temp,true);

                printf("\nCalling function ");
                printProto( orig , true);

                if( !isImportImported(orig->libraryName) ){
                    printf("Warning - library %s is not imported\n",
orig->libraryName);
                }

                strcpy( lastFuncRes.type, orig->funcType );
```

```
                lastFuncRes.res = getOrDefaultResultFromFunction(realName,callParamHead,orig);
            }
            else{
                printf("\nFunction not found\n");
            }

        }
        else{
            printf("\nCalling function ");
            struct PROTOTYPE* orig = getOriginalProto(temp,false);
            strcpy( lastFuncRes.type, orig->funcType);
            printProto( orig , true);
        }
        callParamHead = NULL;
        callParamTail = NULL;
    }

    void assignFromFunction(char* name, char *varType, bool update){
        double val = 0;
        if( strncmp("void",lastFuncRes.type,4) == 0 ){
            printf("\nInvalid assignment from void to %s\n", lastDataType);
            val = 0;
        }
        else if( strcmp(varType, lastFuncRes.type) == 0 ){
            val = lastFuncRes.res;
        }
        else if( strncmp("int", lastDataType, 3) != 0 ){
            val = lastFuncRes.res;
        }
        else if( strncmp("int", lastDataType, 3) == 0 ){
            val = (int)lastFuncRes.res;
        }
        else{
            printf("\nImcompatible assignment. Using default value...\n");
        }

        if(update){
            assignIfPossible(name,val);
        }
        else{
            declareVariable(name, lastDataType, val);
        }

    }

    void continueToAssignFromFunction(char *name, bool update){

        if(doesVariableExists(name)){
            struct VARIABLE* var = getVariable(name);
            assignFromFunction(name,var->type,update);
        }
        else{
            printf("\nVariable %s doesn't exists");
        }

    }

%}

%union{
    double num;
    char *name;
    bool myBool;
}

%error-verbose
%token ENTRY_POINT END_POINT
%token DATA_TYPE FUNC_TYPE VAR NUMBER ARITH_OPE DISCARD
%token FUNC_NAME

%token OUTPUT OUTPUT_VC OUTPUT_SEP OUTPUT_END
%token JUST_IN_CASE COND_OPE OR AND VAR_CON TILL ELSE

// defining token type
//%type<TYPE> ID1 ID2

%type<name> DATA_TYPE VAR FUNC_TYPE FUNC_NAME ARITH_OPE DISCARD
%type<name> OUTPUT_VC OUTPUT_SEP
%type<name> VAR_CON COND_OPE
%type<name> func_call
%type<num> NUMBER arith_exp
%type<myBool> many_logic_cond logic_cond
```

```
%%
program: many_proto_type ENTRY_POINT block END_POINT     { printf("
program executed"); }
    ;

many_proto_type:
    | many_proto_type FUNC_TYPE FUNC_NAME '(' many_type ');' {
        processProto($2,$3);
    }

many_type:
    | FUNC_TYPE many_type { initProto($1); }
    | FUNC_TYPE ',' many_type { initProto($1); }
    ;


block: {}
    | JUST_IN_CASE if_sec block {}
    | TILL loop_sec block {}
    | func_call ';' block {}
    | var_declare block {}
    | var_assign block  {}
    | DISCARD discard_var block {}
    | OUTPUT out_sec block {}
    ;

func_call: FUNC_NAME '(' many_var_con ')' {
        processCall($1); $$ = lastFuncRes.type;
    }

many_var_con:
    | many_var_con VAR_CON { addTypeToCall($2); }
    | many_var_con ',' VAR_CON { addTypeToCall($3); }
    ;

var_declare: DATA_TYPE others';'
    ;
others: VAR {
        declareVariable($1, lastDataType, 0);
    }
    | VAR ',' others {
        declareVariable($1, lastDataType, 0);
    }
    | VAR '=' NUMBER {
        declareVariable($1, lastDataType, $3);
    }
    | VAR '=' VAR {
        double val = getValueOrDefault($3);
        declareVariable($1, lastDataType, val);
    }
    | VAR '=' func_call {
        assignFromFunction($1,lastDataType,false);
    }
    | VAR '=' func_call ',' others {
        assignFromFunction($1,lastDataType,false);
    }
    | VAR '=' NUMBER ',' others {
        declareVariable($1, lastDataType, $3);
    }
    | VAR '=' VAR ',' others {
        double val = getValueOrDefault($3);
        declareVariable($1, lastDataType, val);
    }
    | VAR '=' arith_exp {
        declareVariable($1, lastDataType, $3);
    }
    | VAR '=' arith_exp ',' others {
        declareVariable($1, lastDataType, $3);
    }
    ;

var_assign: VAR '=' NUMBER ';'{
        assignIfPossible($1,$3);
    }
    | VAR '=' VAR ';'{
        if( !doesVariableExists($3) ){
            printf("\nVariable %s doesn't exist\n",$3);
        }
        else{
            double val = getValueOrDefault($3);
            assignIfPossible($1, val);
        }
    }
    | VAR '=' func_call ';' {
        continueToAssignFromFunction($1,true);
    }
```

```
    | VAR '=' arith_exp ';'{
        assignIfPossible($1, $3);
    }
    ;

discard_var: VAR ';'{
        discardVariable($1);
    }
    | VAR ',' discard_var {
        discardVariable($1);
    }

out_sec: out_vc OUTPUT_END { printAndClearOutBuffer(); }
    ;
out_vc: OUTPUT_VC { continueOutBuffer($1); }
    | out_vc OUTPUT_SEP OUTPUT_VC { continueOutBuffer($3); }
    ;

if_sec: if_condition '{' block END_POINT else_block { printf("\nIf-else processed\n"); }
    ;

if_condition: '(' many_logic_cond ')' { pushValidity($2); }
    ;

else_block: ELSE '{' block END_POINT { popValidity(); }
    | { popValidity(); }
    ;

loop_sec: '(' logic_cond ')' '{' loop_updater END_POINT {
        printf("\nLoop matched\n");
        startLoop();
    }
    ;

loop_updater: VAR '=' arith_exp ';'{
        lastAssign.var = $1;
        //printf("=== %s %s %s ===",lastCond.left,lastCond.op, lastCond.right);
        //printf("=== %s %s %s %s ===", lastAssign.var, lastAssign.left,lastAssign.op,lastAssign.right);
    }
    ;

many_logic_cond: logic_cond { $$ = $1; isLastIfValid=$$; }
    | many_logic_cond OR logic_cond { $$ = $1 || $3; isLastIfValid=$$; }
    | many_logic_cond AND logic_cond { $$ = $1 && $3; isLastIfValid=$$; }
    ;

logic_cond: VAR_CON COND_OPE VAR_CON {
        $$ = evaluateCondition($1, $2, $3);
        //printf("------%s %s %s %d\n",$1, $2, $3, $$);
    }
    ;

arith_exp: VAR ARITH_OPE VAR {
        double val1 = getValueOrDefault($1);
        double val2 = getValueOrDefault($3);
        updateLast($1, $2, $3);
        $$ = doArithOperation(val1, val2, $2);
    }
    | VAR ARITH_OPE NUMBER {
        double val1 = getValueOrDefault($1);

        updateLast($1, $2, dtoc($3) );
        $$ = doArithOperation(val1, $3, $2);

    }
    | NUMBER ARITH_OPE VAR {
        double val2 = getValueOrDefault($3);

        updateLast( dtoc($1), $2, $3);
        $$ = doArithOperation($1, val2, $2);
    }
    | NUMBER ARITH_OPE NUMBER {
        updateLast( dtoc($1), $2, dtoc($3) );
        $$ = doArithOperation($1, $3, $2);
    }
    ;



%%


void yyerror(char *s)
```

```
{
    fprintf(stderr, "\n%s", s);
}

int main(){

    initializeLibraryFunction();

    freopen("input.txt","r",stdin);
    //freopen("input2.txt","r",stdin);
    freopen("output.txt", "w", stdout); // output in file
    yyparse();
    printAll();
    printf("\nPrinting all prototype\n");
    printAllProto();
    printAllLibraryFunction();
    printAllImports();
    return 0;
}

```