constant.h

```

#include<stdbool.h>
#ifndef CONSTANT_H
#define CONSTANT_H

//returns true if ch is an arithmatic operator
bool isArithOp(char *ch);

// true if op is a valid conditional operator
bool isCondOpValid(char* op);

// true if condition is valid after performing operation on given parameter
bool isConditionValid(double left, char* op, double right);

// perform arithmatic operations and returns result
double getValue(double left, double right, char *op);

#endif
```

constant.c

```c

#include "constant.h"
#include<stdbool.h>
#include<string.h>
#include<stdio.h>
#include <stddef.h>
#include<stdlib.h>
#include <ctype.h>

int COND_OPERATORS_SIZE = 6;
char COND_OPERATORS[6][5] = {
    "lt","gt","eq","neq","le","ge"
};

int ARITHMATIC_OPERATORS_SIZE = 6;
char ARITHMATIC_OPERATORS[6][5] = {
    "add", "sub", "mul", "div", "dif", "rem"
};

bool isArithOp(char *ch){
    for(int i=0; i<ARITHMATIC_OPERATORS_SIZE; i++){
        if( strcmp(ARITHMATIC_OPERATORS[i], ch) == 0 ) return true;
    }
    return false;
}

bool isCondOpValid(char* op){
    //char arr[6][5] = { "lt","gt","eq","neq","le","ge" };
    for(int i=0; i<COND_OPERATORS_SIZE; i++){
        if(strcmp(COND_OPERATORS[i],op) == 0) return true;
    }
    return false;
}

bool isConditionValid(double left, char* op, double right){
    if( strncmp(op,"lt",2) == 0) return (left < right);
    if( strncmp(op,"gt",2) == 0) return (left > right);
    if( strncmp(op,"eq",2) == 0) return (left == right);

    if( strncmp(op,"le",2) == 0) return (left <= right);
    if( strncmp(op,"ge",2) == 0) return (left >= right);
    if( strncmp(op,"neq",2) == 0) return (left != right); // 2 fine also
}


double getValue(double left, double right, char *op){
    // "add", "sub"
    if( strncmp(op,ARITHMATIC_OPERATORS[0],3) == 0 ) {
        //printf("\nadd %lf %lf\n",left,right);
        return left+right;
    }
    if( strncmp(op,ARITHMATIC_OPERATORS[1],3) == 0 ) return left-right;

```

```
    // "mul", "div"
    if( strncmp(op,ARITHMATIC_OPERATORS[2],3) == 0 ) return left*right;
    if( strncmp(op,ARITHMATIC_OPERATORS[3],3) == 0 ) return left/( right == 0 ? 1 : right) ;

    // "dif", "rem"
    if( strncmp(op,ARITHMATIC_OPERATORS[4],3) == 0 ) return left>=right ? left-right : right-
left;
    if( strncmp(op,ARITHMATIC_OPERATORS[5],3) == 0 ) return ((int)left) % ((int)right);

}

```

var_list.h

```h
#include<stdbool.h>

#ifndef VAR_LIST_H
#define VAR_LIST_H

struct VARIABLE{
    char name[30];
    double value;
    char type[10];
    struct VARIABLE *prev;
    struct VARIABLE *next;
};

// create variable node using passed parameter
struct VARIABLE* createNode(const char *name, char *type, double value);

// insert variable into linked-list
void insertVariable(char *name, char *type, double val);

// return total number of variables
int getTotalVar();

// update variable with given val
void updateVariable(char *name, double val);

// delete variable from linked-list
void deleteVariable(char *name);

// return true if varible exists with passed name
bool doesVariableExists(char *name);

// return variable struct of given name or NULL if not found
struct VARIABLE* getVariable(char* name);

double getValueOrDefault(char* name);

char* getFormattedValueOrDefault(char *name);

// print all variable available when called
void printAll();

void pushValidity(bool val);

bool getCurrentValidity();

bool popValidity();

#endif

```

var_list.c

```c
#include<stdio.h>
#include <stddef.h>
#include<string.h>
#include<stdlib.h>
#include "var_list.h"
#include <stdbool.h>

const int KEYS_SIZE = 25;
char KEYS[50][25] = {
    "void", "int","double","float","justInCase",
    "println","discard","till","import",
    "static","void","entryPoint",
    "lt","gt","eq","neq","le","ge",
    "add", "sub", "mul", "div", "dif", "rem"
};

struct VARIABLE *head = NULL;
struct VARIABLE *tail = NULL;

struct VARIABLE* createNode(const char *name, char *type, double value) {
    struct VARIABLE *newNode = (struct VARIABLE*) malloc(sizeof(struct VARIABLE));
    if(!newNode) {
        printf("Memory allocation failed.\n");
        return NULL;
    }

    strncpy(newNode->name, name, sizeof(newNode->name) - 1);
    newNode->value = value;
    strncpy(newNode->type,type,sizeof(newNode->type)-1);
    newNode->prev = NULL;
    newNode->next = NULL;
    return newNode;
}

void insertVariable(char *name, char *type, double val) {

    for(int i=0; i<KEYS_SIZE; i++){
        if(strcmp(KEYS[i],name) == 0){
            printf("Keyword '%s' can't be variable\n",name);
            return;
        }
    }

    struct VARIABLE *var = createNode(name,type,val);

    if( strncmp("int",type,3) == 0 ){ // integer
        val = (double)( (int)val ); // ignoring after decimal
    }

    if(tail == NULL) {
        head = var;
        tail = var;
    }
```

```
    else {
        var->prev = tail;
        tail->next = var;
        tail = var;
    }
}

int getTotalVar(){
    struct VARIABLE *ptr;
    int count=0;
    ptr = head;
    while(ptr != NULL){
        count++;
        ptr = ptr->next;
    }
    return count;
}

void updateVariable(char *name, double val){

    struct VARIABLE *ptr;
    ptr = head;
    while(ptr != NULL){
        if( strcmp(ptr->name,name) == 0 ){

            if( strncmp("int",ptr->type,3) == 0 ) val = (int)val;

            ptr->value = val;
            break;
        }
        ptr = ptr->next;
    }
}

void deleteVariable(char *name){
    struct VARIABLE *ptr;
    ptr = head;
    while(ptr != NULL){
        if( strcmp(ptr->name, name) == 0 ){
            // first delete
            if(ptr == head){
                //single node
                if(ptr == tail) tail = NULL;
                head = ptr->next;
            }
            else if(ptr == tail){ // last delete
                tail = tail->prev;
                tail->next = NULL;
            }
            else{
                ptr->prev->next = ptr->next;
                ptr->next->prev = ptr->prev;
            }
        }
        ptr = ptr->next;
    }
```

```
}

bool doesVariableExists(char *name){
    struct VARIABLE *ptr = head;

    while (ptr != NULL)
    {
        if( strcmp(name,ptr->name) == 0 ) return true;
        ptr = ptr->next;
    }
    return false;
}

struct VARIABLE* getVariable(char* name){
    //printf("printing from inner\n");
    //printAll();
    struct VARIABLE *ptr = head;
    while (ptr != NULL)
    {
        //printf("(%s %ld),",ptr->name,ptr->value);
        if( strcmp(name,ptr->name) == 0 ){
            //printf("\n value returning %s %lf\n",ptr->name,ptr->value);
            return ptr;
        }
        ptr = ptr->next;
    }
    return NULL;
}

double getValueOrDefault(char* name){
    struct VARIABLE *ptr = head;
    while (ptr != NULL)
    {
        if( strcmp(name,ptr->name) == 0 ){
            return ptr->value;
        }
        ptr = ptr->next;
    }
    return 0;
}

char* getFormattedValueOrDefault(char *name){
    struct VARIABLE* var = getVariable(name);
    if(var == NULL){
        return "0";
    }

    char *arr = (char *) malloc(20);

    if ( strcmp(var->type,"int") == 0){
        int num = (int)(var->value);
        sprintf(arr, "%d", num);
        return arr;
    }

    double num = (var->value);
```

```
    sprintf(arr, "%lf", num);
    return arr;
}

void printAll(){
    printf("\n");
    if(head == NULL) return;


    printf("----------Printing all variables----------\n");

    struct VARIABLE *ptr;
    ptr = head;
    while(ptr != NULL){
        if ( strcmp(ptr->type,"int") == 0){
            printf("%s(%s) %d -> ",ptr->name,ptr->type,(int)ptr->value);
        }
        else{
            printf("%s(%s) %lf -> ",ptr->name,ptr->type,ptr->value);
        }
        ptr = ptr->next;
    }
    printf("\n\n");
}


bool *validityList = NULL;
int validIndex = 0;

void pushValidity(bool val){
    validIndex++;
    validityList = (bool*) realloc(validityList,validIndex);
    validityList[validIndex-1] = val;
}

bool getCurrentValidity(){
    if(validIndex <= 0) return true;
    return validityList[validIndex-1];
}

bool popValidity(){
    bool val = validityList[validIndex];
    validIndex--;
    return val;
}

```

prototype_list.h

```c
#include<stdbool.h>

#ifndef PROTOTYPE_LIST_H
#define PROTOTYPE_LIST_H

struct PARAMETER{
    char type[10];
    double value;
    struct PARAMETER *next;
    struct PARAMETER *prev;

};

struct PROTOTYPE{
    char funcType[10];
    char funcName[30];
    char libraryName[30];

    struct PARAMETER *paramsHead;
    struct PARAMETER *paramsTail;
    struct PROTOTYPE *prev;
    struct PROTOTYPE *next;
};

// insert import name from full import line
void insertImport(char fulImp[20]);

// returns true if imp is found
bool isImportImported(char *imp);

// prints all included imports
void printAllImports();

// create and returns PARAMETER after creating using type and value
struct PARAMETER* createParameter(const char *type,double value);

// inserts parameter to passed head and tail after creating using type and val
void insertParameter( struct PARAMETER **head, struct PARAMETER **tail, char *type, double val);

// creates proto-type and save it in the list
struct PROTOTYPE* createProto(char *type, char *name, char *libraryName, struct PARAMETER *paramsHead, struct PARAMETER *paramsTail );

// insert proto type to library proto-type list
void insertLibraryProto(struct PROTOTYPE* var);

// prints all library function
void printAllLibraryFunction();

// insert user defined proto-type to list
void insertProto(struct PROTOTYPE* var);

```

```
// returns actual prototype from function call by user, isLibrary true to check library function, false to check user-defined
struct PROTOTYPE* getOriginalProto(struct PROTOTYPE* proto, bool isLibrary);

// checks if passed proto-type exists
bool doesProtoExists(struct PROTOTYPE* proto, bool isLibrary);

// prints all user-defined proto-type
void printAllProto();

// prints proto-type in formatted way, reverse to indicate the parameter order
void printProto(struct PROTOTYPE *ptr, bool reverse);

// returns result after performing library function
double getLibrayFunctionResult(char* name, struct PARAMETER* params);

// for adding library function
void initializeLibraryFunction();

#endif
```

prototype_list.c

```c
#include<stdio.h>
#include <stddef.h>
#include<string.h>
#include<stdlib.h>
#include<math.h>
#include "prototype_list.h"
#include <stdbool.h>

struct PROTOTYPE *libraryProtoHead = NULL;
struct PROTOTYPE *libraryProtoTail = NULL;

struct PROTOTYPE *protoHead = NULL;
struct PROTOTYPE *protoTail = NULL;


char **imports = 0;
int totalImport = 0;


void insertImport(char fulImp[20]){
    char *imp = malloc(14*sizeof(char));

    strncpy(imp,fulImp+7, strlen(fulImp) - 8 );

    if( isImportImported(imp) ){
        printf("\nImport already exists\n");
        return;
    }

    totalImport++;
    imports = (char **) realloc( imports, totalImport * 20 * sizeof(char) );
    imports[totalImport-1] = calloc(20, sizeof(char));

    strcpy(imports[totalImport-1],imp);
    printf("\nImported %s\n",imp);
    free(imp);
}

bool isImportImported(char *imp){
    for(int i=0; i<totalImport; i++){
        if( strcmp(imports[i], imp)  == 0) return true;
    }
    return false;
}


void printAllImports(){
    printf("\nPrinting all imports:\n");
    for(int i=0; i<totalImport; i++){
        printf("%s\n",imports[i]);
    }
}

struct PARAMETER* createParameter(const char *type,double value) {

    struct PARAMETER *node = (struct PARAMETER*) malloc(sizeof(struct PARAMETER));

    if(!node) { printf("Memory allocation failed.\n"); return NULL; }

    strncpy(node->type, type, sizeof(node->type) - 1);
    node->value = value;

    node->prev = NULL;
    node->next = NULL;
    return node;
}


void insertParameter( struct PARAMETER **head, struct PARAMETER **tail, char *type, double val){

    struct PARAMETER *var = createParameter(type,val);

    if(*tail == NULL) {
        *head = var;
        *tail = var;
    }
    else {
        var->prev = (*tail);
        (*tail)->next = var;
        (*tail) = var;
    }
}


struct PROTOTYPE* createProto(char *type, char *name, char *libraryName,
        struct PARAMETER *paramsHead, struct PARAMETER *paramsTail ) {

    struct PROTOTYPE *newNode = (struct PROTOTYPE*) malloc(sizeof(struct PROTOTYPE));
    if(!newNode) {
```

```
        printf("Memory allocation failed.\n");
        return NULL;
    }

    //printf("Inside-3 %s \n",type);
    strncpy(newNode->funcType, type, sizeof(newNode->funcType) - 1);

    //printf("Inside-4\n");
    strncpy(newNode->funcName, name ,sizeof(newNode->funcName)-1);

    strncpy(newNode->libraryName, libraryName ,sizeof(newNode->libraryName)
-1);

    //printf("Inside-5\n");
    newNode->paramsHead = paramsHead;
    newNode->paramsTail = paramsTail;

    newNode->prev = NULL;
    newNode->next = NULL;

    //printf("Inside - 6\n");

    // printf("inside create: ");
    // printProto(newNode);

    return newNode;
}

void insertLibraryProto(struct PROTOTYPE* var) {

    if(libraryProtoTail == NULL) {
        libraryProtoHead = var;
        libraryProtoTail = var;
    }
    else {
        var->prev = libraryProtoTail;
        libraryProtoTail->next = var;
        libraryProtoTail = var;
    }
}

void printAllLibraryFunction(){
    printf("\nAll library functions are:\n");
    struct PROTOTYPE *ptr = libraryProtoHead;

    while( ptr != NULL ){
        printProto(ptr, false);
        ptr = ptr->next;
    }
}

void insertProto(struct PROTOTYPE* var) {

    if(protoTail == NULL) {
        protoHead = var;
        protoTail = var;
    }
    else {
        var->prev = protoTail;
        protoTail->next = var;
        protoTail = var;
    }
}


struct PROTOTYPE* getOriginalProto(struct PROTOTYPE* proto, bool isLibrary){

    struct PROTOTYPE *ptr = isLibrary ? libraryProtoHead : protoHead;
    while (ptr != NULL)
    {
        bool isNameSame = strcmp( proto->funcName ,ptr->funcName) == 0 ?
true : false;

        if(isNameSame){

            struct PARAMETER *param1 = proto->paramsHead;
            struct PARAMETER *param2 = ptr->paramsTail;

            while ( param1 != NULL && param2 != NULL )
            {
                if( strcmp(param2->type, "any") == 0) {}
                else if( strcmp(param1->type, param2->type)  != 0){
                    break;
                }

                param1 = param1->next;
                param2 = param2->prev;
            }

            if(param1 == NULL && param2 == NULL){
                return ptr;
            }

        }
```

```
        ptr = ptr->next;
    }
    return NULL;
}

bool doesProtoExists(struct PROTOTYPE* proto, bool isLibrary){
    return getOriginalProto(proto,isLibrary) != NULL;
}


void printAllProto(){
    struct PROTOTYPE *ptr = protoHead;

    while (ptr != NULL)
    {
        printProto(ptr,true);
        ptr = ptr->next;
    }
}

// prints data also if reverse is false
void printProto(struct PROTOTYPE *ptr, bool reverse){

    printf("%s %s(",ptr->funcType, ptr->funcName);

    struct PARAMETER* param = reverse ? ptr->paramsTail : ptr->paramsHead;

    while(param != NULL){
        printf("%s",param->type);

        //if(!reverse) printf("'%lf'",param->value);

        if( (reverse ? param->prev : param->next) != NULL){ printf(","); }

        param = reverse ? param->prev : param->next;
    }
    printf(")\n");

}

double getLibrayFunctionResult(char* name, struct PARAMETER* params){

    if( strcmp(name, "max") == 0 ){
        return fmax(params->value, params->next->value);
    }

    if( strcmp(name, "sqrt") == 0 ){
        if(params->value < 0){
            printf("\nStay in real world\n");
            return 0;
        }
        return sqrt(params->value);
    }

    if( strcmp(name, "scanInt") == 0 ){
        printf("\nTaking int input 0\n");
        return 0;
    }

    if( strcmp(name, "scan") == 0 ){
        printf("\nTaking input 0.0\n");
        return 0;
    }


    if( strcmp(name, "toInt") == 0 ){
        return (int)(params->value);
    }

    if( strcmp(name, "toFloat") == 0 ){
        return (float)(params->value);
    }

    if( strcmp(name, "toDouble") == 0 ){
        return (double)(params->value);
    }

    if( strcmp(name, "show") == 0 ){
        printf("From show function: %lf \n",params->value);
        return 0;
    }

    return 0;
}

void initializeLibraryFunction(){

    struct PARAMETER *head = NULL, *tail = NULL;


    //scanInt stdio
    {
    head = NULL; tail = NULL;
```

```
    struct PROTOTYPE* scanInt = createProto("int","scanInt","stdio",head,
tail);
    insertLibraryProto(scanInt);
    }

    //scan stdio
    {
    head = NULL; tail = NULL;
    struct PROTOTYPE* scan = createProto("float","scan","stdio",head,tail);
    insertLibraryProto(scan);
    }

    // show stdio
    {
    head = NULL; tail = NULL;
    insertParameter(&head,&tail,"any",-1);
    struct PROTOTYPE* show = createProto("void","show","stdio",head,tail);
    insertLibraryProto(show);
    }

    // max math
    {
    head = NULL; tail = NULL;
    insertParameter(&head,&tail,"any",-1);
    insertParameter(&head,&tail,"any",-1);

    struct PROTOTYPE* max = createProto("float","max","math",head,tail);
    insertLibraryProto(max);
    }


    // sqrt math
    {
    head = NULL; tail = NULL;
    insertParameter(&head,&tail,"any",-1);
    struct PROTOTYPE* sqrt = createProto("double","sqrt","math",head,tail);
    insertLibraryProto(sqrt);
    }

    //toInt converter
    {
    head = NULL; tail = NULL;
    insertParameter(&head,&tail,"any",-1);
    struct PROTOTYPE* toInt = createProto("int","toInt","converter",head,
tail);
    insertLibraryProto(toInt);
    }


    //toFloat converter
    {
    head = NULL; tail = NULL;
    insertParameter(&head,&tail,"any",-1);
    struct PROTOTYPE* toFloat = createProto("float","toFloat","converter",
head,tail);
    insertLibraryProto(toFloat);
    }

    //toDouble converter
    {
    head = NULL; tail = NULL;
    insertParameter(&head,&tail,"any",-1);
    struct PROTOTYPE* toDouble = createProto("double","toDouble","converter"
,head,tail);
    insertLibraryProto(toDouble);
    }

}

```

final.l

```
%{
    #include<stdio.h>
    #include<string.h>
    #include <stdlib.h>
    #include <stdbool.h>
    #include <ctype.h>
    #include "final.tab.h"

    // header, single, multi, var, print, dis, calc_val, if, till else
    int tempCounter[11];

    int blockBalancer = 0;
    int stack[100];
    int stackSize = 0;

    const int MAXIMUM_VAR_SIZE = 10;

    char error[200];

    // extern
    char lastDataType[10];
    char *outBuffer = NULL;
    int outBufferSize = 0;
    bool isLastIfValid = false;
    //extern

    int commentCounter = 0;
    int comment_depth = 0;
    char *comment_buffer = NULL;

    bool canDeclareHeader = true;
    bool isMainBlockFound = false;
    bool isCommingFromMain = false;
    bool isCommingFromVar = false;

    size_t buffer_length = 0;

    int bracketCounter=0;

    char *removeRedundant(char*);
    bool getCurrentValidity();
    void insertImport(char*);

    // variable
    const int MAXIMUM_VARIABLE_LENGTH = 200;

    int pushState(int id){
        stackSize++;
        stack[stackSize] = id;
        return id;
    }

    int popState(){
        if(stackSize > 0) stackSize--;
        return stack[stackSize];
    }

    void inc(int index){
        tempCounter[index]++;
    }

    void initSingleComment(){
        commentCounter++;
        printf("\nSingle line comment: %s\n",yytext);
    }

    void appendToBuffer(char ch){
        comment_buffer = (char*) realloc(comment_buffer, buffer_length + 1);
        buffer_length += sprintf(comment_buffer + buffer_length, "%c", ch);
    }

    void appendTextToBuffer(char* ch){

        if( strncmp(ch,"/*",2) == 0) comment_depth++;
        if( strncmp(ch,"*/",2) == 0) comment_depth--;
        comment_buffer = (char*) realloc(comment_buffer, buffer_length +
strlen(ch) );
        buffer_length += sprintf(comment_buffer + buffer_length, ch);
    }

    void initMultiComment(){
        comment_depth = 1;
        buffer_length = 2;
        comment_buffer = (char*) malloc(3);

        memset(comment_buffer, 0, 3);
        strcat(comment_buffer, "/*");
    }

    void resetBuffer(){
        free(comment_buffer);
```

```
        comment_buffer = NULL;
        buffer_length = 0;
    }

    void process_comment() {
        commentCounter++;
        printf("\nCaptured comment: %s\n\n", comment_buffer);
        //printf("Multi-line comment found\n");
    }

    void stopProgram(char *error){
        printf("%s\n",error);
        free(comment_buffer);
        exit(1);
    }

    void checkForEnd(){
        if (comment_depth == 0) {
            BEGIN(INITIAL);
            process_comment();
            resetBuffer();
        }
        else if(comment_depth < 0){
            resetBuffer();
            stopProgram("Invalid comment found");
        }
    }

    void initMain(){
        canDeclareHeader = false;
        //printf("execution started\n");
    }

    void processHeader(){
        if(!canDeclareHeader){
            stopProgram("Header must be at top");
            return;
        }

    }

    void initVarSec(char *temp){
        char *type = removeRedundant(temp);
        strcpy(lastDataType,type);
        canDeclareHeader = false;
    }

    void sendNumber(char *text){
        double num = 0;
        sscanf(text, "%lf", &num);
        yylval.num = num;
        //printf("======%lf=====",num);
    }

    void initOutBuffer(){
        outBuffer = (char*) malloc(1);
        memset(outBuffer, 0, 1);
        outBufferSize = 1;
    }

    char* removeLpRp(char *val){
        char *op = malloc(14*sizeof(char));
        char *temp = strdup(val);
        strncpy(op,temp+1, strlen(temp) - 2 );
        return op;
    }

%}

%x COMMENT
%x MAIN
%x VAR_SEC
%x IF_SEC
%x ELSE_SEC
%x IGNORE_SEC
%x DISCARD_SEC
%x LOOP_SEC
%x LOOP_BODY_SEC
%x OUT_SEC
%x PROTO_SEC
%x FUNC_SEC
%x FUNC_SEC_VAR

DQ \"
NUMBER [-]?([0-9]+(".")[0-9]+)|[-]?([0-9]+)|("."[0-9]+)
VARIABLE [a-zA-Z][a-zA-Z0-9]*
VAR_NUM ({NUMBER}|{VARIABLE})
COND_OP ("<lt>"|"<gt>"|"<eq>"|"<neq>"|"<le>"|"<ge>")
ARITH_OP ("<add>"|"<sub>"|"<mul>"|"<div>"|"<dif>"|"<rem>")

HEADER "import "[a-zA-Z]+";"

OUT_START "println("[ ]*
OUT_BODY_CONST {DQ}[^\"]*{DQ}
```

```
OUT_BODY_VAR {VAR_NUM}
OUT_SEP [ ]*(","|" )[ ]*
OUT_END [ ]*")"[ ]*;[ ]*

MAIN_START "static void entryPoint"[ ]*"("[ ]*")"[ ]*"{"[ ]*
MAIN_END "}"

SINGLE_LINE_COMMENT ("//").*(\n)?

VARIABLE_ONLY [ ]*{VARIABLE}*[ ]*(,)
VARIABLE_VALUE [ ]*{VARIABLE}[ ]*("=")[ ]*{NUMBER}(,)
VARIABLE_VALUE_ASSIGN_CONST [ ]*{VARIABLE}[ ]*("=")[ ]*{NUMBER};
VARIABLE_VALUE_ASSIGN_VAR [ ]*{VARIABLE}[ ]*("=")[ ]*{VARIABLE};
VARIABLE_VALUE_ASSIGN_CALC [ ]*{VARIABLE}[ ]*("=")[ ]*{VAR_NUM}[ ]*
{ARITH_OP}[ ]*{VAR_NUM}[ ]*,
VARIABLE_VALUE_ASSIGN_CALC_LAST [ ]*{VARIABLE}[ ]*("=")[ ]*{VAR_NUM}[ ]*
{ARITH_OP}[ ]*{VAR_NUM}[ ]*;

DISCARD_START "discard "

VARIABLE_ONLY_LAST [ ]*[a-zA-Z][a-zA-Z0-9]*[ ]*(;)
VARIABLE_VALUE_LAST [ ]*[a-zA-Z][a-zA-Z0-9]*[ ]*("=")[ ]*{NUMBER}(;)

VAR_SPACE [ ]*

DATA_TYPE ("int"|"float"|"double")[ ]
FUNC_TYPE ("int"|"float"|"double"|"void")

IF "justInCase"[ ]*
ELSE "otherwise"
IF_BODY_START "{"
IF_BODY_END [ ]*"}"
IF_SPACE [ ]*

IGNORE_LEFT_BRACE "{"
IGNORE_RIGHT_BRACE "}"

FUNC_START "@"{VARIABLE}

LOOP_START "till"
LOOP_START_BRACE "{"
LOOP_OTHERS [^{}\n]*\n
LOOP_END {IF_BODY_END}

NEW_LINE_AND_TAB [\n\t]*

%%
```

```

{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }


<COMMENT>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<MAIN>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<VAR_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<IF_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<ELSE_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<IGNORE_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<DISCARD_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<LOOP_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<LOOP_BODY_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<OUT_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<PROTO_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<FUNC_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
<FUNC_SEC_VAR>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }

<MAIN>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<VAR_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<IF_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<ELSE_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<IGNORE_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<DISCARD_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<LOOP_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<LOOP_BODY_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<OUT_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<PROTO_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<FUNC_SEC>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }
<FUNC_SEC_VAR>"/*" { initMultiComment(); BEGIN( pushState(COMMENT) ); }

<COMMENT>"/*" { appendTextToBuffer("/*"); }

<COMMENT>"*/" {
    appendTextToBuffer("*/"); checkForEnd();
    if(comment_depth == 0){ inc(2); BEGIN( popState() ); }
}
<COMMENT>{NEW_LINE_AND_TAB}* { appendTextToBuffer("|"); }
<COMMENT>. { appendToBuffer(yytext[0]); }

{HEADER} { insertImport(yytext); inc(0); processHeader();}

{FUNC_TYPE} {
    //printf("m_type: -%s-\n",yytext);
    BEGIN( pushState(PROTO_SEC) );
    char *type = removeRedundant(yytext);
```

```
    yylval.name =  strdup( type );
    return FUNC_TYPE;
}

<PROTO_SEC>{FUNC_TYPE} {
    //printf("type: -%s-\n",yytext);
    char *type = removeRedundant(yytext);
    yylval.name =  strdup( type );
    return FUNC_TYPE;
}
<PROTO_SEC>{VARIABLE} {
    //printf("name: -%s-\n",yytext);
    yylval.name = strdup(yytext);
    return FUNC_NAME;
}

<PROTO_SEC>"," { return *yytext; }
<PROTO_SEC>"(" { return *yytext; }
<PROTO_SEC>");" { BEGIN( popState() ); return *yytext; }

{MAIN_START} {
    blockBalancer++;
    //printf("main start %s\n",yytext);
    initMain();
    BEGIN( pushState(MAIN) );
    return ENTRY_POINT;
}

<MAIN>{DATA_TYPE} {
    //printf("main data type %s\n",yytext);
    initVarSec(yytext);
    char *type = removeRedundant(yytext);
    yylval.name =  strdup( type );
    BEGIN( pushState(VAR_SEC) );
    return DATA_TYPE;
}

<MAIN>{OUT_START} {
    initOutBuffer();
    BEGIN( pushState(OUT_SEC) );
    return OUTPUT;
}

<MAIN>{IF} {
    inc(7);
    BEGIN( pushState(IF_SEC) );
    return JUST_IN_CASE;
}
<MAIN>{ELSE} {
    inc(10);
    BEGIN( pushState(ELSE_SEC) );
    return ELSE;
}

<MAIN>{LOOP_START} {
    inc(8);
    BEGIN( pushState(LOOP_SEC) );
    return TILL;
}

<MAIN>{FUNC_START} {
    //printf("----%s----",yytext);
    BEGIN( pushState(FUNC_SEC) );
    yylval.name = strdup(yytext);
    return FUNC_NAME;
}

<VAR_SEC>{FUNC_START} {
    //printf("----%s----",yytext);
    BEGIN( pushState(FUNC_SEC_VAR) );
    yylval.name = strdup(yytext);
    return FUNC_NAME;
}

<FUNC_SEC_VAR>("("|",") { return *yytext; }
<FUNC_SEC_VAR>(")") { BEGIN( popState() ); return *yytext; }
<FUNC_SEC_VAR>{VAR_NUM} { yylval.name = strdup(yytext); return VAR_CON; }


<FUNC_SEC>{VAR_NUM} { yylval.name = strdup(yytext); return VAR_CON; }

<FUNC_SEC>("("|")"|",") {
    return *yytext;
}

<FUNC_SEC>";" {
    BEGIN( popState() );
    return *yytext;
}

<MAIN>{VARIABLE} {
    yylval.name = strdup(yytext);
    BEGIN( pushState(VAR_SEC) );
    return VAR;
```

```
}

<VAR_SEC>{NUMBER} {
    sendNumber(yytext);
    return NUMBER;
}
<VAR_SEC>{ARITH_OP} {
    yylval.name = removeLpRp(yytext);
    return ARITH_OPE;
}
<VAR_SEC>{VARIABLE} {
    yylval.name = strdup(yytext);
    return VAR;
}
<VAR_SEC>"," { return *strdup(yytext); }
<VAR_SEC>"=" { return *strdup(yytext); }
<VAR_SEC>";" {
    BEGIN( popState() );
    return *strdup(yytext);
}
<VAR_SEC>{VAR_SPACE} {}


<MAIN>{DISCARD_START} {
    //printf("Discard section\n");
    inc(5);
    BEGIN( pushState(DISCARD_SEC) );
    return DISCARD;
}

<DISCARD_SEC>{VARIABLE} {
    //printf("\nFrom discard only:-%s-\n",yytext);
    yylval.name = strdup(yytext);
    return VAR;
}
<DISCARD_SEC>"," { return *yytext; }
<DISCARD_SEC>";" { BEGIN( popState() ); return *yytext; }


<OUT_SEC>{OUT_BODY_CONST} {
    yylval.name = strdup(yytext);
    return OUTPUT_VC;
}
<OUT_SEC>{OUT_BODY_VAR} {
    yylval.name = strdup(yytext);
    return OUTPUT_VC;
}
<OUT_SEC>{OUT_SEP} {
    return OUTPUT_SEP;
}
<OUT_SEC>{OUT_END} {
    inc(4);
    BEGIN( popState() );
    return OUTPUT_END;
}

<ELSE_SEC>("("|")") { return *yytext; }
<ELSE_SEC>"}" { return END_POINT; }
<ELSE_SEC>{IF_BODY_START} {
    //popState(); // popping else section and taking to main section arki
    popState();

    bool isLastIfValid = getCurrentValidity();

    char *valid = !isLastIfValid ? "True" : "False";
    printf("\nElse condition is %s\n",valid);
    if(isLastIfValid){
        bracketCounter=1;
        BEGIN( pushState(IGNORE_SEC) );
    }
    else{
        BEGIN(pushState(MAIN));
    }
    return *yytext;
}


<IF_SEC>("("|")") { return *yytext; }
<IF_SEC>"}" { return END_POINT; }

<IF_SEC>{COND_OP} { yylval.name = removeLpRp(yytext); return COND_OPE; }
<IF_SEC>"and" { return AND; }
<IF_SEC>"or" { return OR; }
<IF_SEC>{VAR_NUM} { yylval.name = strdup(yytext); return VAR_CON; }
<IF_SEC>{IF_BODY_START} {
    //popState(); // popping if section and taking to main section arki
    popState();

    bool test = getCurrentValidity();
    char *valid = test ? "True" : "False";
    printf("\nIf condition is %s\n",valid);

    if(!test){
        bracketCounter=1;
```

```
        BEGIN( pushState(IGNORE_SEC) );
    }
    else{
        BEGIN(pushState(MAIN));
    }
    return *yytext;
}
<IF_SEC>{NEW_LINE_AND_TAB} {}

<IGNORE_SEC>{IGNORE_LEFT_BRACE} { bracketCounter++; }
<IGNORE_SEC>{IGNORE_RIGHT_BRACE} {
    bracketCounter--;
    if(bracketCounter == 0){
        BEGIN( popState() );
        return END_POINT;
    }
}
<IGNORE_SEC>[^}] {}


<LOOP_SEC>("("|")") { return *yytext; }
<LOOP_SEC>"}" { return END_POINT; }
<LOOP_SEC>{COND_OP} { yylval.name = removeLpRp(yytext); return COND_OPE; }
<LOOP_SEC>{VAR_NUM} { yylval.name = strdup(yytext); return VAR_CON; }

<LOOP_SEC>{LOOP_START_BRACE} { BEGIN( pushState(LOOP_BODY_SEC) ); return
*yytext; }

<LOOP_BODY_SEC>{VARIABLE} {
    yylval.name = strdup(yytext);
    BEGIN( pushState(VAR_SEC) );
    return VAR;
}
<LOOP_SEC>{NEW_LINE_AND_TAB} {}
<LOOP_BODY_SEC>{LOOP_END} {
    popState();
    BEGIN( popState() );
    return END_POINT;
}

<LOOP_BODY_SEC>{NEW_LINE_AND_TAB} {}

<MAIN>{MAIN_END} {
    blockBalancer--;
    BEGIN( popState() );
    //printf("\n------main end--\n");
    return END_POINT;
}

<MAIN>{NEW_LINE_AND_TAB}* { }
{NEW_LINE_AND_TAB}* { }

%%

int yywrap(){
    return 1;
}
```

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;
}

```

input.txt

```
/* importing header */
import math;
import stdio;
import math;


// declaring function prototype
int add(int,int, float);
int scan();
void add(int, float, double);

/*
    multi-line comment
    /* support nested also */
*/


static void entryPoint(){

    // variable declaration
    int a;
    float b,c;

    println(a b c);
    // deleting variables
    discard a,b,c;

    /* variable initialization */
    int a = 10, b;
    int c = a <add> b, d = 10 <sub> a;
    println(a,b,c,d);

    double e = @max(2,3), f;
    println(e,f);

    discard c,d,e, /* valid comment */ f;

    // variable assignment
    a = 100;
    println("a: " a);

    b = a <rem> 11;
    println("b: " b);

    b = @max(a,101);
    println("b: " b);
    discard a,b;

    /* condition */
    float a = 10;
```

```
float b = 100;

justInCase(a <lt> b){
    println("Actually executed since " a "< " b);

    justInCase(10 <lt> 100){
        println("Also executed");
    }

    justInCase(10 <gt> b){
        println("Not executed since false");
    }
    otherwise{
        println("Inside else. Since 10" " is < than " b);
    }
}

println("------------- separator -----------------");

a = 10; int c = 15;
justInCase(5 <lt> 10 and a <lt> c){
    println("5 < 10 and " a " < " c);
}

justInCase(5 <lt> 10 and a <lt> b and 10 <gt> 10 ){
    println("Condition is false");
}
otherwise{
    println("Executed else block");
}

int i = 0;
till( i <lt> 100){
    i = i <add> 15;
}

/* library function */

int n = @scanInt();
println(n);

float f = @scan();
println("Value after scan is: " f);

discard n,f;

int m = 6, n = 4;
int result = m <mul> n;
@show(result);

double mx = @max(2.5,5);
@show(mx);

float rootf = @sqrt(42);
```

```
    println("Square root of 42 is :" rootf "(float)");

    int toFind = -42;
    n = @sqrt(-42);
    println("Square root of " toFind " is :" n "(int)");

    n = @toInt(rootf);
    println(rootf " becomes " n " after toInt" );

    rootf = @toFloat(n);
    println(n " becomes " rootf " after toFloat" );

    rootf = @toDouble(rootf);
    println(rootf " becomes " rootf " after toDouble" );

    int mn = @show(n);

    // user defined function
    int n1, n2;
    float f1;


    int addRes = @add(n1,n1,f1);
    println(addRes);

    @add(m,f1,100);

    int dum = @add(m,f1,100);

    @test();
}
```

output.txt

```
Captured comment: /* importing header */

Imported math

Imported stdio

Import already exists

Single line comment: // declaring function prototype

Prototype inserted: int add(int,int,float)

Prototype inserted: int scan()

Prototype inserted: void add(int,float,double)

Captured comment: /*|    multi-line comment|    /* support nested also */|*/

Single line comment: // variable declaration

0 0.000000 0.000000  < - - - - - - - - - - -

Single line comment: // deleting variables

Discarded variable c

Discarded variable b

Discarded variable a

Captured comment: /* variable initialization */

10 0 10 0  < - - - - - - - - - - -

Calling function float max(any,any)

3.000000 0.000000  < - - - - - - - - - - -

Captured comment: /* valid comment */

Discarded variable f

Discarded variable e

Discarded variable d

Discarded variable c

Single line comment: // variable assignment

a: 100  < - - - - - - - - - - -

b: 1  < - - - - - - - - - - -

Calling function float max(any,any)

b: 101  < - - - - - - - - - - -

Discarded variable b

Discarded variable a

Captured comment: /* condition */

If condition is True

Actually executed since 10.000000 < 100.000000  < - - - - - - - - - - -

If condition is True

Also executed < - - - - - - - - - - -

If-else processed

If condition is False

Else condition is True

Inside else. Since 10 is < than 100.000000  < - - - - - - - - - - -

If-else processed

If-else processed
```

```
------------- separator ----------------- < - - - - - - - - - -

If condition is True

5 < 10 and 10.000000  < 15  < - - - - - - - - - -

If-else processed

If condition is False

Else condition is True

Executed else block < - - - - - - - - - -

If-else processed

Loop matched

Iterating - 0, value: 0
Iterating - 1, value: 15
Iterating - 2, value: 30
Iterating - 3, value: 45
Iterating - 4, value: 60
Iterating - 5, value: 75
Iterating - 6, value: 90

Captured comment: /* library function */


Calling function int scanInt()

Taking int input 0

0  < - - - - - - - - - -

Calling function int scan()

Value after scan is: 0.000000  < - - - - - - - - - -

Discarded variable f

Discarded variable n

Calling function void show(any)
From show function: 24.000000

Calling function float max(any,any)

Calling function void show(any)
From show function: 5.000000

Calling function double sqrt(any)

Square root of 42 is :6.480741 (float) < - - - - - - - - - -

Calling function double sqrt(any)

Stay in real world

Square root of -42  is :0 (int) < - - - - - - - - - -

Calling function int toInt(any)
Warning - library converter is not imported

6.480741  becomes 6  after toInt < - - - - - - - - - -

Calling function float toFloat(any)
Warning - library converter is not imported

6  becomes 6.000000  after toFloat < - - - - - - - - - -

Calling function double toDouble(any)
Warning - library converter is not imported

6.000000  becomes 6.000000  after toDouble < - - - - - - - - - -

Calling function void show(any)
From show function: 6.000000

Invalid assignment from void to int

Single line comment: // user defined function


Calling function int add(int,int,float)

0  < - - - - - - - - - -

Calling function void add(int,float,double)

Calling function void add(int,float,double)

Invalid assignment from void to int

Function not found
program executed
----------Printing all variables----------
a(float) 10.000000 -> b(float) 100.000000 -> c(int) 15 -> i(int) 105 -> n(int) 6 -> m(int) 6 -> result(int) 24 -> mx(double) 5.000000 -> rootf(float) 6.00
```

```
Printing all prototype
int add(int,int,float)
int scan()
void add(int,float,double)

All library functions are:
int scanInt()
float scan()
void show(any)
float max(any,any)
double sqrt(any)
int toInt(any)
float toFloat(any)
double toDouble(any)

Printing all imports:
math
stdio
```