

constant.h

```
1 //include<stdbool.h>
2 #ifndef CONSTANT_H
3 #define CONSTANT_H
4
5 //returns true if ch is an arithmetic operator
6 bool isArithOp(char *ch);
7
8 // true if op is a valid conditional operator
9 bool isCondOpValid(char* op);
10
11 // true if condition is valid after performing operation on given parameter
12 bool isConditionValid(double left, char* op, double right);
13
14 // perform arithmetic operations and returns result
15 double getValue(double left, double right, char *op);
16
17 #endif
```

constant.c

```
1 #include "constant.h"
2 #include<stdbool.h>
3 #include<string.h>
4 #include<stdio.h>
5 #include <stddef.h>
6 #include<stdlib.h>
7 #include <cctype.h>
8
9
10 int COND_OPERATORS_SIZE = 6;
11 char COND_OPERATORS[6][5] = {
12     "lt", "gt", "eq", "neq", "le", "ge"
13 };
14
15 int ARITHMATIC_OPERATORS_SIZE = 6;
16 char ARITHMATIC_OPERATORS[6][5] = {
17     "add", "sub", "mul", "div", "dif", "rem"
18 };
19
20 bool isArithOp(char *ch){
21     for(int i=0; i<ARITHMATIC_OPERATORS_SIZE; i++){
22         if( strcmp(ARITHMATIC_OPERATORS[i], ch) == 0 ) return true;
23     }
24     return false;
25 }
26
27 bool isCondOpValid(char* op){
28     //char arr[6][5] = { "lt", "gt", "eq", "neq", "le", "ge" };
29     for(int i=0; i<COND_OPERATORS_SIZE; i++){
30         if(strcmp(COND_OPERATORS[i],op) == 0) return true;
31     }
32     return false;
33 }
34
35 bool isConditionValid(double left, char* op, double right){
36     if( strncmp(op,"lt",2) == 0) return (left < right);
37     if( strncmp(op,"gt",2) == 0) return (left > right);
38     if( strncmp(op,"eq",2) == 0) return (left == right);
39
40     if( strncmp(op,"le",2) == 0) return (left <= right);
41     if( strncmp(op,"ge",2) == 0) return (left >= right);
42     if( strncmp(op,"neq",2) == 0) return (left != right); // 2 fine also
43 }
44
45
46 double getValue(double left, double right, char *op){
47     // "add", "sub"
48     if( strncmp(op,ARITHMATIC_OPERATORS[0],3) == 0 ) {
49         //printf("\nadd %lf %lf\n",left,right);
50         return left+right;
51     }
52     if( strncmp(op,ARITHMATIC_OPERATORS[1],3) == 0 ) return left-right;
53 }
```

```
54 // "mul", "div"
55 if( strncmp(op,ARITHMATIC_OPERATORS[2],3) == 0 ) return left*right;
56 if( strncmp(op,ARITHMATIC_OPERATORS[3],3) == 0 ) return left/( right == 0 ? 1 : right) ;
57
58 // "dif", "rem"
59 if( strncmp(op,ARITHMATIC_OPERATORS[4],3) == 0 ) return left>=right ? left-right : right-
left;
60 if( strncmp(op,ARITHMATIC_OPERATORS[5],3) == 0 ) return ((int)left) % ((int)right);
61
62 }
63 }
```

var_list.h

```
1 #include<stdbool.h>
2
3 #ifndef VAR_LIST_H
4 #define VAR_LIST_H
5
6 struct VARIABLE{
7     char name[30];
8     double value;
9     char type[10];
10    struct VARIABLE *prev;
11    struct VARIABLE *next;
12};
13
14 // create variable node using passed parameter
15 struct VARIABLE* createNode(const char *name, char *type, double value);
16
17 // insert variable into linked-list
18 void insertVariable(char *name, char *type, double val);
19
20 // return total number of variables
21 int getTotalVar();
22
23 // update variable with given val
24 void updateVariable(char *name, double val);
25
26 // delete variable from linked-list
27 void deleteVariable(char *name);
28
29 // return true if variable exists with passed name
30 bool doesVariableExists(char *name);
31
32 // return variable struct of given name or NULL if not found
33 struct VARIABLE* getVariable(char* name);
34
35 double getValueOrDefault(char* name);
36
37 char* getFormattedValueOrDefault(char *name);
38
39 // print all variable available when called
40 void printAll();
41
42 void pushValidity(bool val);
43
44 bool getCurrentValidity();
45
46 bool popValidity();
47
48 #endif
49
```

var_list.c

```
1 #include<stdio.h>
2 #include <stddef.h>
3 #include<string.h>
4 #include<stdlib.h>
5 #include "var_list.h"
6 #include <stdbool.h>
7
8 const int KEYS_SIZE = 25;
9 char KEYS[50][25] = {
10     "void", "int", "double", "float", "justInCase",
11     "println", "discard", "till", "import",
12     "static", "void", "entryPoint",
13     "lt", "gt", "eq", "neq", "le", "ge",
14     "add", "sub", "mul", "div", "dif", "rem"
15 };
16
17 struct VARIABLE *head = NULL;
18 struct VARIABLE *tail = NULL;
19
20 struct VARIABLE* createNode(const char *name, char *type, double value) {
21     struct VARIABLE *newNode = (struct VARIABLE*) malloc(sizeof(struct VARIABLE));
22     if(!newNode) {
23         printf("Memory allocation failed.\n");
24         return NULL;
25     }
26
27     strncpy(newNode->name, name, sizeof(newNode->name) - 1);
28     newNode->value = value;
29     strncpy(newNode->type, type, sizeof(newNode->type)-1);
30     newNode->prev = NULL;
31     newNode->next = NULL;
32     return newNode;
33 }
34
35 void insertVariable(char *name, char *type, double val) {
36
37     for(int i=0; i<KEYS_SIZE; i++){
38         if(strcmp(KEYS[i],name) == 0){
39             printf("Keyword '%s' can't be variable\n",name);
40             return;
41         }
42     }
43
44     struct VARIABLE *var = createNode(name,type,val);
45
46     if( strcmp("int",type,3) == 0 ){ // integer
47         val = (double)( (int)val ); // ignoring after decimal
48     }
49
50     if(tail == NULL) {
51         head = var;
52         tail = var;
53     }
```

```

54     else {
55         var->prev = tail;
56         tail->next = var;
57         tail = var;
58     }
59 }
60
61 int getTotalVar(){
62     struct VARIABLE *ptr;
63     int count=0;
64     ptr = head;
65     while(ptr != NULL){
66         count++;
67         ptr = ptr->next;
68     }
69     return count;
70 }
71
72 void updateVariable(char *name, double val){
73
74     struct VARIABLE *ptr;
75     ptr = head;
76     while(ptr != NULL){
77         if( strcmp(ptr->name,name) == 0 ){
78
79             if( strncmp("int",ptr->type,3) == 0 ) val = (int)val;
80
81             ptr->value = val;
82             break;
83         }
84         ptr = ptr->next;
85     }
86 }
87
88 void deleteVariable(char *name){
89     struct VARIABLE *ptr;
90     ptr = head;
91     while(ptr != NULL){
92         if( strcmp(ptr->name, name) == 0 ){
93             // first delete
94             if(ptr == head){
95                 //single node
96                 if(ptr == tail) tail = NULL;
97                 head = ptr->next;
98             }
99             else if(ptr == tail){ // last delete
100                 tail = tail->prev;
101                 tail->next = NULL;
102             }
103             else{
104                 ptr->prev->next = ptr->next;
105                 ptr->next->prev = ptr->prev;
106             }
107         }
108         ptr = ptr->next;
109     }

```

```

110 }
111
112 bool doesVariableExists(char *name){
113     struct VARIABLE *ptr = head;
114
115     while (ptr != NULL)
116     {
117         if( strcmp(name,ptr->name) == 0 ) return true;
118         ptr = ptr->next;
119     }
120     return false;
121 }
122
123 struct VARIABLE* getVariable(char* name){
124     //printf("printing from inner\n");
125     //printAll();
126     struct VARIABLE *ptr = head;
127     while (ptr != NULL)
128     {
129         //printf("(%s %ld),",ptr->name,ptr->value);
130         if( strcmp(name,ptr->name) == 0 ){
131             //printf("\n value returning %s %lf\n",ptr->name,ptr->value);
132             return ptr;
133         }
134         ptr = ptr->next;
135     }
136     return NULL;
137 }
138
139 double getValueOrDefault(char* name){
140     struct VARIABLE *ptr = head;
141     while (ptr != NULL)
142     {
143         if( strcmp(name,ptr->name) == 0 ){
144             return ptr->value;
145         }
146         ptr = ptr->next;
147     }
148     return 0;
149 }
150
151 char* getFormattedValueOrDefault(char *name){
152     struct VARIABLE* var = getVariable(name);
153     if(var == NULL){
154         return "0";
155     }
156
157     char *arr = (char *) malloc(20);
158
159     if ( strcmp(var->type,"int") == 0){
160         int num = (int)(var->value);
161         sprintf(arr, "%d", num);
162         return arr;
163     }
164
165     double num = (var->value);

```

```

166     sprintf(arr, "%lf", num);
167     return arr;
168 }
169
170 void printAll(){
171     printf("\n");
172     if(head == NULL) return;
173
174     printf("-----Printing all variables-----\n");
175
176     struct VARIABLE *ptr;
177     ptr = head;
178     while(ptr != NULL){
179         if( strcmp(ptr->type,"int") == 0){
180             printf("%s(%s) %d -> ",ptr->name,ptr->type,(int)ptr->value);
181         }
182         else{
183             printf("%s(%s) %lf -> ",ptr->name,ptr->type,ptr->value);
184         }
185         ptr = ptr->next;
186     }
187     printf("\n\n");
188 }
189
190
191
192 bool *validityList = NULL;
193 int validIndex = 0;
194
195 void pushValidity(bool val){
196     validIndex++;
197     validityList = (bool*) realloc(validityList,validIndex);
198     validityList[validIndex-1] = val;
199 }
200
201 bool getCurrentValidity(){
202     if(validIndex <= 0) return true;
203     return validityList[validIndex-1];
204 }
205
206 bool popValidity(){
207     bool val = validityList[validIndex];
208     validIndex--;
209     return val;
210 }
211

```

prototype_list.h

```
1 #include<stdbool.h>
2
3 #ifndef PROTOTYPE_LIST_H
4 #define PROTOTYPE_LIST_H
5
6 struct PARAMETER{
7     char type[10];
8     double value;
9     struct PARAMETER *next;
10    struct PARAMETER *prev;
11
12 };
13
14 struct PROTOTYPE{
15     char funcType[10];
16     char funcName[30];
17     char libraryName[30];
18
19     struct PARAMETER *paramsHead;
20     struct PARAMETER *paramsTail;
21     struct PROTOTYPE *prev;
22     struct PROTOTYPE *next;
23 };
24
25 // insert import name from full import line
26 void insertImport(char fulImp[20]);
27
28 // returns true if imp is found
29 bool isImportImported(char *imp);
30
31 // prints all included imports
32 void printAllImports();
33
34 // create and returns PARAMETER after creating using type and value
35 struct PARAMETER* createParameter(const char *type,double value);
36
37 // inserts parameter to passed head and tail after creating using type and val
38 void insertParameter( struct PARAMETER **head, struct PARAMETER **tail, char *type, double val );
39
40 // creates proto-type and save it in the list
41 struct PROTOTYPE* createProto(char *type, char *name, char *libraryName, struct PARAMETER *paramsHead, struct PARAMETER *paramsTail );
42
43 // insert proto type to library proto-type list
44 void insertLibraryProto(struct PROTOTYPE* var);
45
46 // prints all library function
47 void printAllLibraryFunction();
48
49 // insert user defined proto-type to list
50 void insertProto(struct PROTOTYPE* var);
```

```
52 // returns actual prototype from function call by user, isLibrary true to check library
53 // function, false to check user-defined
54 struct PROTOTYPE* getOriginalProto(struct PROTOTYPE* proto, bool isLibrary);
55
56 // checks if passed proto-type exists
57 bool doesProtoExists(struct PROTOTYPE* proto, bool isLibrary);
58
59 // prints all user-defined proto-type
60 void printAllProto();
61
62 // prints proto-type in formatted way, reverse to indicate the parameter order
63 void printProto(struct PROTOTYPE *ptr, bool reverse);
64
65 // returns result after performing library function
66 double getLibrayFunctionResult(char* name, struct PARAMETER* params);
67
68 // for adding library function
69 void initializeLibraryFunction();
70 #endif
```

prototype_list.c

```
45 void printAllImports(){
46     printf("\nPrinting all imports:\n");
47     for(int i=0; i<totalImport; i++){
48         printf("%s\n",imports[i]);
49     }
50 }
51 }
52
53 struct PARAMETER* createParameter(const char *type,double value) {
54     struct PARAMETER *node = (struct PARAMETER*) malloc(sizeof(struct
55 PARAMETER));
56
57     if(!node) { printf("Memory allocation failed.\n"); return NULL; }
58
59     strncpy(node->type, type, sizeof(node->type) - 1);
60     node->value = value;
61
62     node->prev = NULL;
63     node->next = NULL;
64     return node;
65 }
66
67 void insertParameter( struct PARAMETER **head, struct PARAMETER **tail, char
68 *type, double val){
69
70     struct PARAMETER *var = createParameter(type,val);
71
72     if(*tail == NULL) {
73         *head = var;
74         *tail = var;
75     }
76     else {
77         var->prev = (*tail);
78         (*tail)->next = var;
79         (*tail) = var;
80     }
81 }
82
83 struct PROTOTYPE* createProto(char *type, char *name, char *libraryName,
84 struct PARAMETER *paramsHead, struct PARAMETER *paramsTail ) {
85
86     struct PROTOTYPE *newNode = (struct PROTOTYPE*) malloc(sizeof(struct
87 PROTOTYPE));
88
89     if(!newNode) {
90
91         bool isImportImported(char *imp){
92             for(int i=0; i<totalImport; i++){
93                 if( strcmp(imports[i], imp) == 0) return true;
94             }
95             return false;
96         }
97
98     }
99 }
```

```

89     printf("Memory allocation failed.\n");
90     return NULL;
91 }
92
93 //printf("Inside-3 %s \n",type);
94 strcpy(newNode->funcType, type, sizeof(newNode->funcType) - 1);
95
96 //printf("Inside-4\n");
97 strcpy(newNode->funcName, name , sizeof(newNode->funcName) -1);
98
99 strcpy(newNode->libraryName, libraryName , sizeof(newNode->libraryName)
-1);
100 //printf("Inside-5\n");
101 newNode->paramsHead = paramsHead;
102 newNode->paramsTail = paramsTail;
103
104 newNode->prev = NULL;
105 newNode->next = NULL;
106
107 //printf("Inside - 6\n");
108
109 // printf("inside create: ");
110 // printProto(newNode);
111
112 return newNode;
113
114 }
115
116 void insertLibraryProto(struct PROTOTYPE* var) {
117
118 if(libraryProtoTail == NULL) {
119     libraryProtoHead = var;
120     libraryProtoTail = var;
121 }
122 else {
123     var->prev = libraryProtoTail;
124     libraryProtoTail->next = var;
125     libraryProtoTail = var;
126 }
127
128
129 void printAllLibraryFunction(){
130     printf("\nAll library functions are:\n");
131     struct PROTOTYPE *ptr = libraryProtoHead;
132
133     while( ptr != NULL ){
134         printProto(ptr, false);
135     }
136 }
137 }
138
139 void insertProto(struct PROTOTYPE* var) {
140
141     if(protoTail == NULL) {
142         protoHead = var;
143         protoTail = var;
144     }
145     else {
146         var->prev = protoTail;
147         protoTail->next = var;
148         protoTail = var;
149     }
150 }
151
152 struct PROTOTYPE* getOriginalProto(struct PROTOTYPE* proto, bool isLibrary){
153
154     struct PROTOTYPE *ptr = isLibrary ? libraryProtoHead : protoHead;
155
156     while (ptr != NULL)
157     {
158         if(isNameSame = strcmp( proto->funcName ,ptr->funcName ) == 0 ?
159             true : false;
160
161         if(isNameSame){
162             struct PARAMETER *param1 = proto->paramsHead;
163             struct PARAMETER *param2 = ptr->paramsTail;
164
165             while ( param1 != NULL && param2 != NULL )
166             {
167                 if( strcmp(param2->type, "any") == 0 ) {}
168                 else if( strcmp(param1->type, param2->type) != 0 ){
169                     break;
170                 }
171             }
172             param1 = param1->next;
173             param2 = param2->next;
174         }
175
176     if(param1 == NULL && param2 == NULL){
177         return ptr;
178     }
179 }
180 }
```

```

181     }
182     ptr = ptr->next;
183     return NULL;
184 }
185 }
186 bool doesProtoExists(struct PROTOTYPE* proto, bool isLibrary){
187     return getOriginalProto(proto,isLibrary) != NULL;
188 }
189 }
190 void printAllProto(){
191     struct PROTOTYPE *ptr = protoHead;
192     while (ptr != NULL)
193     {
194         printProto(ptr,true);
195         ptr = ptr->next;
196     }
197 }
198 }
199 }
200 }
201 // prints data also if reverse is false
202 void printProto(struct PROTOTYPE *ptr, bool reverse){
203     printf("%s %s(%s", ptr->funcType, ptr->funcName);
204     struct PARAMETER* param = reverse ? ptr->paramsTail : ptr->paramsHead;
205     while(param != NULL){
206         printf("%s",param->type);
207         param = reverse ? param->prev : param->next;
208     }
209     printf("\n");
210 }
211 //if(!reverse) printf("'%lf'",param->value);
212 if( (reverse ? param->prev : param->next) != NULL){ printf(" ,");
213 param = reverse ? param->prev : param->next;
214 printf("\n");
215 }
216 double getLibraryFunctionResult(char* name, struct PARAMETER* params){
217     if( strcmp(name, "max") == 0 ){
218         return fmax(params->value, params->next->value);
219     }
220 }
221 double getLibraryFunctionResult(char* name, struct PARAMETER* params){
222     struct PARAMETER *head = NULL, *tail = NULL;
223     if( strcmp(name, "max") == 0 ){
224         return fmax(params->value, params->next->value);
225     }
226 }
227 void initializeLibraryFunction(){
228     if( strcmp(name, "sqrt") == 0 ){
229         if(params->value < 0){
230             printf("\nStay in real world\n");
231             return 0;
232         }
233         return sqrt(params->value);
234     }
235     if( strcmp(name, "scanInt") == 0 ){
236         printf("\nTaking int input 0\n");
237         return 0;
238     }
239     if( strcmp(name, "scan") == 0 ){
240         printf("\nTaking input 0.0\n");
241         return 0;
242     }
243     if( strcmp(name, "toInt") == 0 ){
244         return (int)(params->value);
245     }
246     if( strcmp(name, "toFloat") == 0 ){
247         return (float)(params->value);
248     }
249     if( strcmp(name, "toDouble") == 0 ){
250         return (double)(params->value);
251     }
252     if( strcmp(name, "show") == 0 ){
253         printf("From show function: %f \n",params->value);
254     }
255     if( strcmp(name, "libraryFunction") == 0 ){
256         return 0;
257     }
258     if( strcmp(name, "libraryFunction") == 0 ){
259         printf("From libraryFunction: %f \n",params->value);
260         return 0;
261     }
262 }
263 return 0;
264 }
265 }
266 void initializeLibraryFunction(){
267 struct PARAMETER *head = NULL, *tail = NULL;
268 //scanfInt stdio
269 {
270     head = NULL; tail = NULL;
271 }
272 }
273 }
274 }

```

```

275  struct PROTOTYPE* scanInt = createProto("int", "scanf", "stdio", head,
276      insertLibraryProto(scanInt);
277  }
278
279 // scan stdio
280 {
281     head = NULL; tail = NULL;
282     struct PROTOTYPE* scan = createProto("float", "scanf", "stdio", head, tail);
283     insertLibraryProto(scan);
284 }
285
286 // show stdio
287 {
288     head = NULL; tail = NULL;
289     insertParameter(&head, &tail, "any", -1);
290     struct PROTOTYPE* show = createProto("void", "show", "stdio", head, tail);
291     insertLibraryProto(show);
292 }
293
294 // max math
295 {
296     head = NULL; tail = NULL;
297     insertParameter(&head, &tail, "any", -1);
298     insertParameter(&head, &tail, "any", -1);
299
300     struct PROTOTYPE* max = createProto("float", "max", "math", head, tail);
301     insertLibraryProto(max);
302 }
303
304 // sqrt math
305 {
306     head = NULL; tail = NULL;
307     insertParameter(&head, &tail, "any", -1);
308     struct PROTOTYPE* sqrt = createProto("double", "sqrt", "math", head, tail);
309     insertLibraryProto(sqrt);
310 }
311
312 //ToInt converter
313 {
314     head = NULL; tail = NULL;
315     insertParameter(&head, &tail, "any", -1);
316     struct PROTOTYPE* toInt = createProto("int", "toInt", "converter", head,
317         tail);
318     insertLibraryProto(toInt);
319 }
```

final.1

```
45 const int MAXIMUM_VARIABLE_LENGTH = 200;
46
47 int pushState(int id){
48     stackSize++;
49     stack[stackSize] = id;
50     return id;
51 }
52
53 int popState(){
54     if(stackSize > 0) stackSize--;
55     return stack[stackSize];
56 }
57
58 void inc(int index){
59     tempCounter[index]++;
60 }
61
62 void initSingleComment(){
63     commentCounter++;
64     printf("\nSingle line comment: %s\n",yytext);
65 }
66
67 void appendToBuffer(char ch){
68     comment_buffer = (char*) realloc(comment_buffer, buffer_length + 1);
69     buffer_length += sprintf(comment_buffer + buffer_length, "%c", ch);
70 }
71
72 void appendTextToBuffer(char* ch){
73
74     if( strcmp(ch,"/*",2) == 0) comment_depth++;
75     if( strcmp(ch,"*/",2) == 0) comment_depth--;
76     comment_buffer = (char*) realloc(comment_buffer, buffer_length +
77     strlen(ch));
78     buffer_length += sprintf(comment_buffer + buffer_length, ch);
79 }
80
81 void initMultiComment(){
82     comment_depth = 1;
83     comment_buffer = (char*) malloc(3);
84
85     memset(comment_buffer, 0, 3);
86     strcat(comment_buffer, "/");
87 }
88
89 void resetBuffer(){
90     free(comment_buffer);
91 }
```

```

91     comment_buffer = NULL;
92     buffer_length = 0;
93 }
94
95 void process_comment() {
96     commentCounter++;
97     printf("\nCaptured comment: %s\n\n", comment_buffer);
98     //printf("Multi-line comment found\n");
99 }
100
101 void stopProgram(char *error){
102     printf("%s\n", error);
103     free(comment_buffer);
104     exit(1);
105 }
106
107 void checkForEnd(){
108     if (comment_depth == 0) {
109         BEGIN(INITIAL);
110         process_comment();
111         resetBuffer();
112     }
113     else if(comment_depth < 0){
114         resetBuffer();
115         stopProgram("Invalid comment found");
116     }
117 }
118
119 void initMain(){
120     canDeclareHeader = false;
121     //printf("execution started\n");
122 }
123
124 void processHeader(){
125     if(!canDeclareHeader){
126         stopProgram("Header must be at top");
127     }
128 }
129
130
131 void initVarSec(char *temp){
132     char *type = removeRedundant(temp);
133     strcpy(lastDataType,type);
134     canDeclareHeader = false;
135 }
136
137
138     void sendNumber(char *text){
139         double num = 0;
140         sscanf(text, "%lf", &num);
141         yyval.num = num;
142         //printf("======%lf=====, num);
143     }
144
145     void initOutBuffer(){
146         outBuffer = (char*) malloc(1);
147         memset(outBuffer, 0, 1);
148         outBufferSize = 1;
149     }
150
151     char* removeLpRp(char *val){
152         char *op = malloc(14*sizeof(char));
153         char *temp = strdup(val);
154         strcpy(op,temp+1, strlen(temp) - 2 );
155         return op;
156     }
157
158 }
159
160 %X COMMENT
161 %X MAIN
162 %X VAR_SEC
163 %X IF_SEC
164 %X ELSE_SEC
165 %X IGNORE_SEC
166 %X DISCARD_SEC
167 %X LOOP_SEC
168 %X LOOP_BODY_SEC
169 %X OUT_SEC
170 %X PROTO_SEC
171 %X FUNC_SEC
172 %X FUNC_SEC_VAR
173
174 DQ \
175 NUMBER [-]?([0-9]+("." [0-9]+)[-]?([0-9]+)|("." [0-9]+))
176 VARIABLE [a-zA-Z][a-zA-Z0-9]*
177 VAR_NUM {NUMBER} {VARIABLE}
178 COND_OP ('<lt>' | '<gt>' | '<eq>' | '<neq>' | '<le>' | '<ge>')
179 ARITH_OP ('<add>' | '<sub>' | '<mul>' | '<div>' | '<diff>' | '<rem>')
180 HEADER "import" [a-zA-Z]+";
181
182
183 OUT_START "println([" *
184 OUT_BODY_CONST {DQ}[\\"]*{DQ}
```

```

185 OUT_BODY_VAR {VAR_NUM}
186 OUT_SEP [ ]*( ";" " ")[
187 OUT_END [ ]*")[" ];
188
189 MAIN_START "static void entryPoint"[ ]*(" [ ]*")" [ ]*{"[ ]*
190 MAIN_END "}";
191 SINGLE_LINE_COMMENT ("//").*(\n)?
192
193 VARIABLE_ONLY [ ]*{VARIABLE}*[" ,"]
194 VARIABLE_VALUE [ ]*{VARIABLE}[ "(" "[ ]*{NUMBER}[,]
195 VARIABLE_VALUE_ASSIGN_CONST [ ]*{VARIABLE}[ "(" "[ ]*{NUMBER};]
196 VARIABLE_VALUE_ASSIGN_VAR [ ]*{VARIABLE}[ "(" "[ ]*{VARIABLE};]
197 VARIABLE_VALUE_ASSIGN_VAR [ ]*{VARIABLE}[ "(" "[ ]*{VARIABLE};]
198 VARIABLE_VALUE_ASSIGN_CALC [ ]*{VARIABLE}[ "(" "[ ]*{VAR_NUM}[ ]
199 VARIABLE_VALUE_ASSIGN_CALC_LAST [ ]*{VARIABLE}[ "(" [=")][ ]*{VAR_NUM}[ ]*
{ARITH_OP}[ ]*{VAR_NUM}[ ],
200 VARIABLE_VALUE_ASSIGN_CALC_LAST [ ]*{VARIABLE}[ ]*{VAR_NUM}[ ]*
{ARITH_OP}[ ]*{VAR_NUM}[ ];
201 DISCARD_START "discard "
202
203 VARIABLE_ONLY_LAST [ ]*[a-zA-Z][a-zA-Z0-9]*[" ;"]
204 VARIABLE_VALUE_LAST [ ]*[a-zA-Z][a-zA-Z0-9]*[" =")][ ]*{NUMBER}();
205 VAR_SPACE [ ]*
206
207 DATA_TYPE ("int"|"float"|"double")[ ]
208 FUNC_TYPE ("int"|"float"|"double"|"void")
209 FUNC_TYPE ("int"|"float"|"double"|"void")
210
211 IF "justInCase"[ ]
212 ELSE "otherwise"
213 IF_BODY_START "{"
214 IF_BODY_END [ ]*
215 IF_SPACE [ ]*
216
217 IGNORE_LEFT_BRACE "{"
218 IGNORE_RIGHT_BRACE "}"
219
220 FUNC_START "@'{VARIABLE}'"
221 LOOP_START "till"
222 LOOP_START_BRACE "{"
223 LOOP_OTHERS [^}]\\n]*\\n
224 LOOP_END {IF_BODY_END}
225
226 NEW_LINE_AND_TAB [ \\n\\t]*
227
228 %
229
230 {SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
231 /* { initMultiComment(); BEGIN( pushState(COMMENT) ); }
232
233
234 <COMMENT>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
235 <MAIN>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
236 <VAR_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
237 <IF_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
238 <ELSE_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
239 <IGNORE_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
240 <DISCARD_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
241 <LOOP_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
242 <LOOP_BODY_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
243 <OUT_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
244 <PROTO_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
245 <FUNC_SEC>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
246 <FUNC_SEC_VAR>{SINGLE_LINE_COMMENT} { inc(1); initSingleComment(); }
247
248 <MAIN>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
249 <VAR_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
250 <IF_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
251 <ELSE_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
252 <IGNORE_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
253 <DISCARD_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
254 <LOOP_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
255 <LOOP_BODY_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
256 <OUT_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
257 <PROTO_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
258 <FUNC_SEC>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
259 <FUNC_SEC_VAR>/* { initMultiComment(); BEGIN( pushState(COMMENT) );
260
261 <COMMENT>/* { appendTextToBuffer( /* );
262 <COMMENT>/* { appendTextToBuffer( /* );
263 <COMMENT>/* { appendTextToBuffer( /* );
264 <COMMENT>/* { appendTextToBuffer( /* ); checkForEnd();
265 if(comment_depth == 0){ inc(2); BEGIN( popState() ); }
266
267
268 <COMMENT>{NEW_LINE_AND_TAB}* { appendTextToBuffer( /* ); checkForEnd();
269 <COMMENT> . { appendToBuffer(yytext[0]); }
270 {HEADER} { insertImport(yytext); inc(0); processHeader(); }
271
272 {FUNC_TYPE} {
273 //printf("m_type: %s-\n",yytext);
274 BEGIN( pushState(PROTO_SEC) );
275 char *type = removeRedundant(yytext);
276

```

```

277     yyval.name = strdup( type );
278     return FUNC_TYPE;
279 }
280 <PROTO_SEC>{FUNC_TYPE} {
281     //printf("type: -%s-\n",yytext);
282     char *type = removeRedundant(yytext);
283     yyval.name = strdup( type );
284     return FUNC_TYPE;
285 }
286 <PROTO_SEC>{VARIABLE} {
287     //printf("name: -%s-\n",yytext);
288     yyval.name = strdup(yytext);
289     return FUNC_NAME;
290 }
291 }
292 <PROTO_SEC>," { return *yytext; }
293 <PROTO_SEC>" { return *yytext; }
294 <PROTO_SEC>" { BEGIN( popState() ); return *yytext; }
295 <PROTO_SEC>;" { BEGIN( popState() ); return *yytext; }
296 {MAIN_START} {
297     blockBalance++;
298     //printf("main start %s\n",yytext);
299     initMain();
300     BEGIN( pushState(MAIN) );
301     return ENTRY_POINT;
302 }
303 }
304 <MAIN>{DATA_TYPE} {
305     //printf("main data type %s\n",yytext);
306     initVarSec(yytext);
307     char *type = removeRedundant(yytext);
308     yyval.name = strdup( type );
309     BEGIN( pushState(VAR_SEC) );
310     return DATA_TYPE;
311 }
312 }
313 <MAIN>{OUT_START} {
314     initOutBuffer();
315     BEGIN( pushState(OUT_SEC) );
316     return OUTPUT;
317 }
318 }
319 <MAIN>{IF} {
320     inc(7);
321     BEGIN( pushState(IF_SEC) );
322     BEGIN( pushState(IF_SEC) );
323     return JUST_IN_CASE;
324 }
325 <MAIN>{ELSE} {
326     inc(10);
327     BEGIN( pushState(ELSE_SEC) );
328     return ELSE;
329 }
330 <MAIN>{LOOP_START} {
331     inc(8);
332     BEGIN( pushState(LOOP_SEC) );
333     BEGIN( pushState(TILL) );
334     return TILL;
335 }
336 <MAIN>{FUNC_START} {
337     //printf("----%s----",yytext);
338     BEGIN( pushState(FUNC_SEC) );
339     yyval.name = strdup(yytext);
340     return FUNC_NAME;
341 }
342 }
343 <VAR_SEC>{FUNC_START} {
344     //printf("----%s----",yytext);
345     BEGIN( pushState(FUNC_SEC) );
346     yyval.name = strdup(yytext);
347     yyval.name = strdup(yytext);
348     return FUNC_NAME;
349 }
350 }
351 <FUNC_SEC_VAR>" { return *yytext; }
352 <FUNC_SEC_VAR>" { BEGIN( popState() ); return *yytext; }
353 <FUNC_SEC_VAR>{VAR_NUM} { yyval.name = strdup(yytext); return VAR_CON; }
354 }
355 <FUNC_SEC_VAR>" { return *yytext; }
356 <FUNC_SEC>{VAR_NUM} { yyval.name = strdup(yytext); return VAR_CON; }
357 }
358 <FUNC_SEC>" { return *yytext; }
359 }
360 }
361 <FUNC_SEC>; {
362     BEGIN( popState() );
363     yyval.name = strdup(yytext);
364     return *yytext;
365 }
366 <MAIN>{VARIABLE} {
367     yyval.name = strdup(yytext);
368     BEGIN( pushState(VAR_SEC) );
369     return VAR;
370 }

```

```

371 }
372 <VAR_SEC>{NUMBER} {
373   sendNumber(yytext);
374   return NUMBER;
375 }
376 <VAR_SEC>{ARITH_OP} {
377   yy1val.name = removeLrp(yytext);
378   return ARITH_OPE;
379 }
380 <VAR_SEC>{VARIABLE} {
381   yy1val.name = removeLrp(yytext);
382   return VAR;
383 }
384 <VAR_SEC>; { return *strdup(yytext); }
385 <VAR_SEC>!= { return *strdup(yytext); }
386 <VAR_SEC>!= { return *strdup(yytext); }
387 <VAR_SEC>; {
388   BEGIN( popState() );
389   return *strdup(yytext);
390 }
391 <VAR_SEC>{VAR_SPACE} {}
392 }
393 <MAIN>{DISCARD_START} {
394   //printf("Discard section\n");
395   inc(5);
396   BEGIN( pushState(DISCARD_SEC) );
397   return DISCARD;
398 }
399 }
400 <DISCARD_SEC>{VARIABLE} {
401   //printf("\nFrom discard only:-%s-\n",yytext);
402   yy1val.name = strdup(yytext);
403   return VAR;
404 }
405 <DISCARD_SEC>; { return *yytext; }
406 <DISCARD_SEC>; { BEGIN( popState() ); return *yytext; }
407 <DISCARD_SEC>; { BEGIN( popState() ); return *yytext; }
408 }
409 <OUT_SEC>{OUT_BODY_CONST} {
410   yy1val.name = strdup(yytext);
411   return OUTPUT_VC;
412 }
413 <OUT_SEC>{OUT_BODY_VAR} {
414   yy1val.name = strdup(yytext);
415   return OUTPUT_VC;
416 }
417 }

418 <OUT_SEC>{OUT_SEP} {
419   return OUTPUT_SEP;
420 }
421 <OUT_SEC>{OUT_END} {
422   inc(4);
423   BEGIN( popState() );
424   return OUTPUT_END;
425 }
426 <ELSE_SEC>"("|"")" { return *yytext; }
427 <ELSE_SEC>"%" { return END_POINT; }
428 <ELSE_SEC>"%" { return END_POINT; }
429 <ELSE_SEC>{IF_BODY_START} {
430   //popState(); // popping else section and taking to main section arki
431   popState();
432   bool isLastIfValid = getCurrentValidity();
433   if(isLastIfValid)
434     char *valid = !isLastIfValid ? "True" : "False";
435     printf("\nElse condition is %s\\n",valid);
436   if(isLastIfValid){
437     bracketCounter=1;
438     BEGIN( pushState(IGNORE_SEC) );
439   }
440   else{
441     BEGIN(pushState(MAIN));
442   }
443   }
444   return *yytext;
445 }
446 }
447 <IF_SEC>"("|"")" { return *yytext; }
448 <IF_SEC>"%" { return END_POINT; }
449 <IF_SEC>"%" { return END_POINT; }
450 }
451 <IF_SEC>{COND_OP} { yy1val.name = removeRp(yytext); return COND_OPE; }
452 <IF_SEC>"and" { return AND; }
453 <IF_SEC>"or" { return OR; }
454 <IF_SEC>{VAR_NUM} { yy1val.name = strdup(yytext); return VAR_CON; }
455 <IF_SEC>{IF_BODY_START} {
456   //popState(); // popping if section and taking to main section arki
457   popState();
458   bool test = getCurrentValidity();
459   char *valid = test ? "True" : "False";
460   printf("\nIf condition is %s\\n",valid);
461   if(!test){
462     bracketCounter=1;
463   }
464 }
```

```

465 } pushState(IGNORE_SEC) );
466
467 else{
468     BEGIN(pushState(MAIN));
469 }
470     return *yytext;
471 }
472 <IF_SEC>{NEW_LINE_AND_TAB} {}
473
474 <IGNORE_SEC>{IGNORE_LEFT_BRACE} { bracketCounter++;
475 <IGNORE_SEC>{IGNORE_RIGHT_BRACE} {
476     bracketCounter--;
477     if(bracketCounter == 0){
478         BEGIN( popState() );
479         return END_POINT;
480     }
481     <IGNORE_SEC>[^} ] {};
482     <IGNORE_SEC>[^\} ] {};
483
484     <LOOP_SEC>("(" | ")") { return *yytext; }
485     <LOOP_SEC>"}" { return END_POINT; }
486     <LOOP_SEC>"{" { return END_POINT; }
487     <LOOP_SEC>{COND_OP} { yyival.name = removeLpRp(yytext); return COND_OPE; }
488     <LOOP_SEC>{VAR_NUM} { yyival.name = strdup(yytext); return VAR_CON; }
489
490     <LOOP_SEC>{LOOP_START_BRACE} { BEGIN( pushState(LOOP_BODY_SEC) ); return
491     *yytext; }
492     <LOOP_BODY_SEC>{VARIABLE} {
493         yyival.name = strdup(yytext);
494         BEGIN( pushState(VAR_SEC) );
495         return VAR;
496     }
497     <LOOP_SEC>{NEW_LINE_AND_TAB} {}
498     <LOOP_BODY_SEC>{LOOP_END} {
499         popState();
500         BEGIN( popState() );
501         return END_POINT;
502     }
503     <LOOP_BODY_SEC>{NEW_LINE_AND_TAB} {}
504
505     <MAIN_END> {
506         blockBalancer--;
507         BEGIN( popState() );
508         //printf("\n-----main end--\n");
509         return END_POINT;
510     }

```

```

41     char* removeRedundant (char *source){
42         int len = strlen(source);
43
44         char temp[len+1];
45         int j = 0;
46         for(int i=0; i<len; i++){
47             //printf("%d, ", source[i]);
48             if( source[i] == '\n' || 
49                 source[i] == '\r' || 
50                 source[i] == '\v' || 
51                 source[i] == '\t' || 
52                 source[i] == '\n' || 
53                 source[i] == 32 )|| 
54             )continue;
55
56         temp[j] = source[i];
57         j++;
58     }
59
60     temp[j] = '\0';
61     char *dest;
62     dest = malloc(j*sizeof(char));
63     strcpy(dest,temp);
64     //printf("\n%s\n",dest);
65     return dest;
66 }
67
68 void printAndClearOutBuffer(){
69     printf("\n%$s <-----\n",outBuffer);
70     free(outBuffer);
71     outBufferSize = 0;
72 }
73
74 void continueOutBuffer(char *vc){
75     int i=0, len = strLen(vc) - 1;
76     while( i <= len && vc[i] == ' ' ) i++; // leading space
77     while( len>0 && vc[len] == ' ' ) len--; // trailing space
78     after "-----" len--; // 
79     space
80     char *res;
81
82     if(vc[i] == '\n') { // constant

```

```

83     i++; // trailing quotaion bad dilam
84
85     int size = len-i+1;
86     res = (char *)malloc(size+1);
87     strncpy(res, vc + i, size);
88     res[size] = '\0';
89
90   else{ // variable
91     vc = removeRedundant(vc);
92     struct VARIABLE *var = getVariable(vc);
93
94     if( !doesVariableExists(vc) {
95       printf("\nVariable %s doesn't exist\n",vc);
96
97       char* val = getFormattedValueOrDefault(vc);
98       res = (char *) malloc(20);
99       sprintf(res, "%s ", val);
100      }
101
102      outBufferSize += strlen(res);
103      outBuffer = (char *)realloc(outBuffer,outBufferSize);
104
105      strcat(outBuffer,res);
106    }
107
108  void initProto(char *paramType){
109    //printf("type found: %s\n",paramType);
110    // -1 default value, not needed for proto-type
111    insertParameter(&paramHead, &paramTail, paramType,-1);
112  }
113
114  void processProto(char *funcType, char *funcName){
115    //printf("Inside\n");
116
117    struct PROTOTYPE* proto = createProto(funcType, funcName, "
118 user_defined", paramHead,paramTail);
119    //printf("Inside-2\n");
120
121    paramHead = NULL; paramTail = NULL;
122
123  if( doesProtoExists(proto,false) {
124    printf("\nDuplicate proto-type found\n");
125
126    return;
127
128    insertProto(proto);
129    printf("\nPrototype inserted: ");
130    printProto(proto,true);
131
132  }
133
134  double doArithOperation(double val1, double val2, char* op){
135    //printf("%lf %lf %s -----kuttar baccha-\n",val1,val2,op);
136    return getValve(val1, val2, op);
137
138
139  bool addVariable(char* name,char *type){
140    if( doesVariableExists(name) {
141      printf("\nVariable %s is already defined\n",name);
142      return false;
143    }
144    insertVariable(name,type,0);
145    return true;
146
147
148  void declareVariable(char *name, char *type, double val){
149    if( addVariable(name, type) {
150      updateVariable(name, val);
151    }
152  }
153
154  void assignIfPossible(char *name, double val){
155
156    if( !doesVariableExists(name) {
157      printf("\nVariable %s doesn't exist\n",name);
158      return;
159    }
160    updateVariable(name, val);
161
162
163  }
164
165  void discardVariable(char *name){
166    if( !doesVariableExists(name) {
167

```

```

168     printf("\nCan't discard %s since not found.\n", name);
169     return;
170 }
171 printf("\nDiscarded variable %s\n", name);
172 deleteVariable(name);
173
174 bool isNumber(char* num){
175     bool count = 0;
176     int i = num[0] == '-' ? 1 : 0;
177
178     for(; i<strlen(num); i++){
179         if( num[i] == '.' && count == 0 ) { count=1; continue; }
180         if( !isdigit(num[i]) ) return false;
181     }
182     return true;
183 }
184
185 bool evaluateCondition(char* leftChar, char* op, char *rightChar){
186
187     LastCond.left = leftChar;
188     LastCond.op = op;
189     LastCond.right = rightChar;
190
191     double left = 0, right = 0;
192
193     if( isNumber(leftChar) ){
194         left = strtod(leftChar, NULL);
195     }
196     else{
197         if( !doesVariableExists(leftChar) ){
198             printf("\nVariable %s doesn't exist. Using default
199 value(0)\n", leftChar);
200         }
201         left = getValueOrDefault(leftChar);
202
203     if( isNumber(rightChar) ){
204         right = strtod(rightChar, NULL);
205     }
206     else{
207         if( !doesVariableExists(rightChar) ){
208             printf("\nVariable %s doesn't exist. Using default
209 value(0)\n", right);
210         }
211         right = getValueOrDefault(rightChar);
212     }
213
214     return isConditionValid(left, op, right);
215 }
216
217 void updateLast(char *left, char *op, char *right){
218     lastAssign.left = left;
219     lastAssign.op = op;
220     lastAssign.right = right;
221 }
222
223 char* dtoC(double val){
224     char* buf = malloc(sizeof(char) * 20 );
225     sprintf(buf, "%lf", val);
226
227     return buf;
228 }
229
230 void startLoop(){
231     int counter = 0;
232     printf("\n");
233     while( evaluateCondition( lastCond.left, lastCond.op,
234     lastCond.right ) ){
235         printf("Iterating - %d", counter);
236
237         if( !isNumber(lastCond.left) ){
238             printf(" , value: %s"
239             getFormattedValueOrDefault(lastCond.left) );
240             printf("\n");
241
242             double val1 = 0, val2 = 0;
243             if( isNumber(lastAssign.left) ){
244                 sscanf(lastAssign.left, "%lf" , &val1);
245             }
246             else{
247                 val1 = getValueOrDefault(lastAssign.left);
248             }
249
250         if( isNumber(lastAssign.right) ){

```

```

251     insertParameter(&callParamHead, &callParamTail, type, val);
252
253     sscanf(lastAssign.right, "%lf" , &val12);
254
255     else{
256         val2 = getValueOrDefault(lastAssign.right);
257
258         if( !doesVariableExists(lastAssign.var) ){
259             printf("\nVariable %s doesn't exists. Stopping....\n" ,
260                   lastAssign.var);
261             break;
262         }
263         double res = doAnithOperation(val1, val2, lastAssign.op );
264         updateVariable( lastAssign.var, res );
265         counter++;
266         Sleep(200);
267     }
268     void addTypeToCall(char *name){
269         //printf("Type call %s\n",name);
270         char *type = "double";
271         double val = 0;
272
273         if( isNumber(name) ){
274             type = "double";
275             val = strtod(name,NULL);
276         }
277         else{
278             struct VARIABLE* var = getVariable(name);
279             type = var->type;
280             val = var->value;
281         }
282         if( doesVariableExists(name) ){
283             struct PROTOTYPE* orig = getOriginalProto(temp,true);
284             printf("\nCalling function ");
285             printProto( orig , true);
286         }
287         else{
288             printf("\nVariable %s doesn't exists\n",name);
289         }
290     }
291
292     strcpy( lastFuncs.type, orig->funcType );
293
294     if( !isImported(orig->libraryName) ){
295         printf("Warning - Library %s is not imported\n" ,
296               orig->libraryName);
297     }
298     double getResultFromFunction(
299         char* name, struct PARAMETER* head,
300         struct PROTOTYPE* funcToCall )
301     {
302         return getLibraryFunctionResult(name,head);
303     }
304
305     void processCall(char *funcName){
306         char realName[ strlen(funcName) ];
307         strcpy(realName, funcName + 1);
308         // Copy the string starting from the second character
309         // strcpy(realName, funcName + 1);
310         struct PROTOTYPE* temp = createProto("void",realName, "NULL" );
311         callParamHead=NULL;
312
313         strcpy( lastFuncRes.type, "null" );
314         lastFuncRes.res = 0;
315
316         if( !doesProtoExists(temp,false) ){
317             strcpy( lastFuncRes.type, "null" );
318             lastFuncRes.res = 0;
319
320             // check library function
321             if( doesProtoExists(temp,true) ){
322                 // returns from library function
323                 struct PROTOTYPE* orig = getOriginalProto(temp,true)
324                 printf("\nCalling function ");
325                 printProto( orig , true);
326             }
327         }
328
329         if( !isImported(orig->libraryName) ){
330             printf("Warning - Library %s is not imported\n" ,
331               orig->libraryName);
332         }
333     }

```

```

334     lastFuncRes.res = getOrDefaultResultFromFunction(realName,callParamHead,orig);
335     }
336     else{
337         printf("\nFunction not found\n");
338     }
339 }
340 else{
341     printf("\nCalling function ");
342     struct PROTOTYPE* orig = getOriginalProto(temp, false);
343     strcpy( lastFuncRes.type, orig->funcType);
344     printProto( orig , true);
345 }
346 callParamHead = NULL;
347 callParamTail = NULL;
348 }
349 }

350 void assignFromFunction(char* name, char *varType, bool update){
351     double val = 0;
352     if( strcmp("void",lastFuncRes.type,4) == 0 ){
353         printf("\nInvalid assignment from void to %s\n",
354         lastDataType);
355         val = 0;
356     }
357     else if( strncmp(varType, lastFuncRes.type) == 0 ){
358         val = lastFuncRes.res;
359     }
360     else if( strncmp("int", lastDataType, 3) != 0 ){
361         val = lastFuncRes.res;
362     }
363     else if( strncmp("int", lastDataType, 3) == 0 ){
364         val = (int)lastFuncRes.res;
365     }
366     else{
367         printf("\nIncompatible assignment. Using default
368         value...\n");
369     }
370     if(update){
371         assignIfPossible(name, val);
372     }
373     declareVariable(name, lastDataType, val);
374 }

375     }
376 }
377 }
378 void continueToAssignFromFunction(char *name, bool update){
379
380     if(doesVariableExists(name)){
381         struct VARIABLE* var = getVariable(name);
382         assignFromFunction(name,var->type,update);
383     }
384     else{
385         printf("\nVariable %s doesn't exists");
386     }
387 }
388 }
389 }
390 }
391 %

392 %union{
393     double num;
394     char *name;
395     bool myBool;
396 }
397 }

398 %error-verbose
399 %token ENTRY_POINT END_POINT
400 %token DATA_TYPE FUNC_TYPE VAR NUMBER ARITH_OPE DISCARD
401 %token DATA_TYPE FUNC_TYPE VAR NUMBER ARITH_OPE
402 %token FUNC_NAME
403 %token OUTPUT_OUTPUT_VC_OUTPUT_SEP OUTPUT_END
404 %token JUST_IN_CASE_COND_OPE OR AND VAR_CON TILL ELSE
405 // defining token type
406 //%%type<TYPE> ID1 ID2
407
408 %%type<name> DATA_TYPE VAR FUNC_TYPE FUNC_NAME ARITH_OPE DISCARD
409 %%type<name> OUTPUT_VC_OUTPUT_SEP
410 %%type<name> VAR_CON_COND_OPE
411 %%type<name> func_call
412 %%type<num> NUMBER
413 %%type<myBool> many_logic_cond logic_cond
414 %%type<myBool> declareVariable(name, lastDataType, val);
415
416
417

```

```

418 %% program: many_proto_type ENTRY_POINT block END_POINT      f printf(" 419
419   program_executed"); } ;
420
421 many_proto_type:
422   | many_proto_type FUNC_TYPE FUNC_NAME `(` many_type `)`; {
423     processProto($2,$3);
424   }
425
426 many_type:
427   | FUNC_TYPE many_type { initProto($1); }
428   | FUNC_TYPE `;` many_type { initProto($1); }
429   | ;
430
431 block: {}
432
433   | JUST_IN_CASE if_sec block {}
434   | TILL_LOOP_SEC block {}
435   | func_call `;` block {}
436   | var_declare block {}
437   | var_assign block {}
438   | DISCARD discard_var block {}
439   | OUTPUT out_sec block {}
440   |
441
442 func_call: FUNC_NAME `(` many_var_con `)` {
443   processCall($1); $$ = lastFuncRes.type;
444
445   }
446 many_var_con:
447   | many_var_con VAR_CON { addTypeToCall($2); }
448   | many_var_con `;` VAR_CON { addTypeToCall($3); }
449   | ;
450
451 var_declare: DATA_TYPE others `;` ;
452
453 others: VAR {
454   declareVariable($1, lastDataType, 0);
455   | VAR `;` others {
456     declareVariable($1, lastDataType, 0);
457   }
458
459   }
460
461   | VAR `=` NUMBER {
462     declareVariable($1, lastDataType, $3);
463   }
464   | VAR `=` VAR {
465     double val = getValueOrDefault($3);
466     declareVariable($1, lastDataType, val);
467   }
468   | VAR `=` func_call `;` others {
469     assignFromFunction($1, lastDataType, false);
470   }
471   | VAR `=` func_call `;` others {
472     assignFromFunction($1, lastDataType, $3);
473   }
474   | VAR `=` NUMBER `;` others {
475     declareVariable($1, lastDataType, $3);
476   }
477   | VAR `=` VAR `;` others {
478     double val = getValueOrDefault($3);
479     declareVariable($1, lastDataType, val);
480   }
481   | VAR `=` arith_exp `;` others {
482     declareVariable($1, lastDataType, $3);
483   }
484   | VAR `=` arith_exp `;` others {
485     declareVariable($1, lastDataType, $3);
486   }
487
488 var_assign: VAR `=` NUMBER `;` {
489   assignIfPossible($1,$3);
490   }
491   | VAR `=` VAR `;` {
492     if( !doesVariableExists($3) ) {
493       printf("\nVariable %s doesn't exist\n", $3);
494     }
495   }
496   | double val = getValueOrDefault($3);
497   | assignIfPossible($1, val);
498   }
499   | VAR `=` func_call `;` {
500     continueToAssignFromFunction($1, true);
501   }
502

```

```

503 | VAR `=' arith_exp `;` {
504 |   assignIfPossible($1, $3);
505 }
506 ;
507
508 discard_var: VAR `:' {
509   discardVariable($1);
510 }
511 | VAR `,' discard_var {
512   discardVariable($1);
513 }
514
515 out_sec: out_vc OUTPUT_END { printAndClearOutBuffer(); }
516 ;
517 out_vc: OUTPUT_VC { continueOutBuffer($1); }
518 | out_vc OUTPUT_SEP OUTPUT_VC { continueOutBuffer($3); }
519 ;
520 if_sec: if_condition {' block END_POINT else_block { printf("\nIf-
521 else processed\n"); }
522 if_condition: (' many_logic_cond ') { pushValidity($2); }
523 ;
524 if_condition: (' many_logic_cond ') { pushValidity($2); }
525 ;
526 else_block: ELSE {' block END_POINT { popValidity(); }
527 | { popValidity(); }
528 ;
529
530 loop_sec: (' logic_cond ') {' loop_updater END_POINT {
531   printf("\nLoop matched\n");
532   startLoop();
533 }
534 ;
535
536 loop_updater: VAR `=' arith_exp `;` {
537   lastAssign.var = $1;
538   //printf("== %s %s ==", lastCond.left, lastCond.op,
539   lastCond.right);
540   //printf("== %s %s %s ==", lastAssign.var,
541   lastAssign.left, lastAssign.op, lastAssign.right);
542 }
543
544 many_logic_cond: logic_cond { $$ = $1; isLastIfValid=$$; }
545 | many_logic_cond OR logic_cond { $$ = $1 || $3; isLastIfValid=$$;
546 | many_logic_cond AND logic_cond { $$ = $1 && $3; isLastIfValid=
547 ;
548
549 logic_cond: VAR_COND_OPE VAR_CON {
550   $$ = evaluateCondition($1, $2, $3);
551   //printf("-----%s %s %d\n",$1, $2, $3, $$);
552 }
553 ;
554
555 arith_exp: VAR_ARITH_OPE VAR {
556   double val1 = getValueOrDefault($1);
557   double val2 = getValueOrDefault($3);
558   updateLast($1, $2, $3);
559   $$ = doArithOperation(val1, val2, $2);
560 }
561 | VAR_ARITH_OPE NUMBER {
562   double val1 = getValueOrDefault($1);
563
564   updateLast($1, $2, dtoc($3));
565   $$ = doArithOperation(val1, $3, $2);
566 }
567 | NUMBER_ARITH_OPE VAR {
568   NUMBER_ARITH_OPE NUMBER {
569   double val2 = getValueOrDefault($3);
570
571   updateLast( dtoc($1), $2, $3);
572   $$ = doArithOperation($1, val2, $2);
573 }
574 | NUMBER_ARITH_OPE NUMBER {
575   updateLast( dtoc($1), $2, dtoc($3));
576   $$ = doArithOperation($1, $3, $2);
577 }
578 ;
579
580
581
582
583
584
585 void yyerror(char *s)

```

```
586 {  
587     fprintf(stderr, "\n%s", s);  
588 }  
589  
590 int main(){  
591     initializeFunction();  
592     freopen("input.txt", "r", stdin);  
593     //freopen("input2.txt", "r", stdin);  
594     freopen("output.txt", "w", stdout); // output in file  
595     yyparse();  
596     printAll();  
597     printf("\nPrinting all prototype\n");  
598     printAllProto();  
599     printAllLibraryFunction();  
600     printAllImports();  
601     return 0;  
602 }  
603  
604 }  
605 }
```

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) 6.000
```

```
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
```