

final.y

```
1  %{
2
3  #include<stdio.h>
4  #include<math.h>
5  #include<string.h>
6  #include<stdlib.h>
7  #include <ctype.h>
8  #include "prototype_list.h"
9  #include "constant.h"
10 #include "var_list.h"
11 #include<windows.h>
12
13 extern char lastDataType[10];
14
15 int yylex(void);
16 void yerror(char *s);
17
18 struct PARAMETER *paramHead = NULL;
19 struct PARAMETER *paramTail = NULL;
20
21 struct PARAMETER *callParamHead = NULL;
22 struct PARAMETER *callParamTail = NULL;
23
24 extern char *outBuffer;
25 extern int outBufferSize;
26 extern bool *validityList;
27 extern bool isLastIfValid;
28
29 struct COND{ char *left, *op, *right; };
30 struct ASSIGN{ char *var, *left, *op, *right; };
31
32 struct COND lastCond;
33 struct ASSIGN lastAssign;
34
35 struct FUNC_RESULT {
36     char type[10];
37     double res;
38 } lastFuncRes;
39
40
```

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

```

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, "
user_defined", paramHead,paramTail);
118     //printf("Inside-2\n");
119
120     paramHead = NULL; paramTail = NULL;
121
122     if( doesProtoExists(proto,false) ){
123         printf("\nduplicate proto-type found\n");
124
125         return;
126     }
127
128     insertProto(proto);
129     printf("\nPrototype inserted: ");
130     printProto(proto,true);
131 }
132
133 double doArithOperation(double val1, double val2, char* op){
134     //printf("%lf %lf %s -----kuttar baccha-\n",val1,val2,op);
135     return getValue(val1, val2, op);
136 }
137
138 bool addVariable(char* name,char *type){
139     if( doesVariableExists(name) ){
140         printf("\nVariable %s is already defined\n",name);
141         return false;
142     }
143     insertVariable(name,type,0);
144     return true;
145 }
146
147 void declareVariable(char *name, char *type, double val){
148     if( addVariable(name, type) ){
149         updateVariable(name,val);
150     }
151
152     void assignIfPossible(char *name, double val){
153         if( !doesVariableExists(name) ){
154             printf("\nVariable %s doesn't exist\n",name);
155             return;
156         }
157         updateVariable(name,val);
158     }
159
160     void discardVariable(char *name){
161         if( !doesVariableExists(name) ){
162

```

```

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
175 bool isNumber(char* num){
176     bool count = 0;
177     int i = num[0] == '-' ? 1 : 0;
178
179     for(; i<strlen(num); i++){
180         if( num[i] == '.' && count == 0 ) { count=1; continue; }
181         if( !isdigit(num[i]) ) return false;
182     }
183     return true;
184 }
185
186 bool evaluateCondition(char* leftChar, char* op, char *rightChar){
187
188     lastCond.left = leftChar;
189     lastCond.op = op;
190     lastCond.right = rightChar;
191
192     double left = 0, right = 0;
193
194     if( isNumber(leftChar) ){
195         left = strtod(leftChar,NULL);
196     }
197     else{
198         if( !doesVariableExists(leftChar) ){
199             printf("\nVariable %s doesn't exist. Using default
200 value(0)\n",leftChar);
201         }
202         left = getValueOrDefault(leftChar);
203     }
204
205     if( isNumber(rightChar) ){
206         right = strtod(rightChar,NULL);
207     }
208     else{
209         if( !doesVariableExists(rightChar) ){

```

```

210 value(0)\n",rightChar);
211     }
212     right = getValueOrDefault(rightChar);
213 }
214
215     return isConditionValid(left, op, right);
216 }
217
218 void updateLast(char *left, char *op, char *right){
219     lastAssign.left = left;
220     lastAssign.op = op;
221     lastAssign.right = right;
222 }
223
224 char* dtoc(double val){
225     char* buf = malloc( sizeof(char) * 20 );
226     sprintf(buf, "%lf", val);
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
236         printf("Iterating - %d",counter);
237
238         if( !isNumber(lastCond.left) ){
239             printf(" value: %s",
240 getFormattedValueOrDefault(lastCond.left) );
241         }
242         printf("\n");
243
244         double val1 = 0, val2 = 0;
245         if( isNumber(lastAssign.left) ){
246             sscanf(lastAssign.left, "%lf", &val1);
247         }
248         else{
249             val1 = getValueOrDefault(lastAssign.left);
250         }
251
252         if( isNumber(lastAssign.right) ){

```

```

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

```

```

334         lastFuncRes.res =
335         getOrDefaultResultFromFunction(realName, callParamHead, orig);
336     }
337     else{
338         printf("\nFunction not found\n");
339     }
340
341 }
342 else{
343     printf("\nCalling function ");
344     struct PROTOTYPE* orig = getOriginalProto(temp, false);
345     strcpy( lastFuncRes.type, orig->funcType);
346     printProto( orig , true);
347 }
348 callParamHead = NULL;
349 callParamTail = NULL;
350
351 }
352
353 void assignFromFunction(char* name, char *varType, bool update){
354     double val = 0;
355     if( strcmp("void", lastFuncRes.type, 4) == 0 ){
356         printf("\nInvalid assignment from void to %s\n",
357             lastDataTypes);
358     }
359     val = 0;
360 }
361 else if( strcmp(varType, lastFuncRes.type) == 0 ){
362     val = lastFuncRes.res;
363 }
364 else if( strcmp("int", lastDataTypes, 3) != 0 ){
365     val = lastFuncRes.res;
366 }
367 else if( strcmp("int", lastDataTypes, 3) == 0 ){
368     val = (int)lastFuncRes.res;
369 }
370 else{
371     printf("\nIncompatible assignment. Using default
372     value...\n");
373 }
374
375 if(update){
376     assignIfPossible(name, val);
377 }
378
379 else{
380     declareVariable(name, lastDataTypes, val);

```

```

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
393 %union{
394     double num;
395     char *name;
396     bool myBool;
397 }
398
399 %error-verbose
400 %token ENTRY_POINT END_POINT
401 %token DATA_TYPE FUNC_TYPE VAR NUMBER ARITH_OPE DISCARD
402 %token FUNC_NAME
403
404 %token OUTPUT OUTPUT_VC OUTPUT_SEP OUTPUT_END
405 %token JUST_IN_CASE COND_OPE OR AND VAR_CON TILL ELSE
406
407 // defining token type
408 // %type<TYPE> ID1 ID2
409
410 %type<name> DATA_TYPE VAR FUNC_TYPE FUNC_NAME ARITH_OPE DISCARD
411 %type<name> OUTPUT_VC OUTPUT_SEP
412 %type<name> VAR_CON COND_OPE
413 %type<name> func_call
414 %type<num> NUMBER arith_exp
415 %type<myBool> many_logic_cond logic_cond
416
417

```

```

418 %%
419 program: many_proto_type ENTRY_POINT block END_POINT { printf("
420 program executed"); }
421 ;
422 many_proto_type:
423 | many_proto_type FUNC_TYPE FUNC_NAME '(' many_type ',' {
424 | processProto($2,$3);
425 }
426
427 many_type:
428 | FUNC_TYPE many_type { initProto($1); }
429 | FUNC_TYPE ',' many_type { initProto($1); }
430 ;
431
432 block: {}
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
454 others: VAR {
455 | declareVariable($1, lastDataType, 0);
456 }
457 | VAR ',' others {
458 | declareVariable($1, lastDataType, 0);
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 {
469 | assignFromFunction($1,lastDataType,false);
470 }
471 | VAR '=' func_call ',' others {
472 | assignFromFunction($1,lastDataType,false);
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 {
482 | declareVariable($1, lastDataType, $3);
483 }
484 | VAR '=' arith_exp ',' others {
485 | declareVariable($1, lastDataType, $3);
486 }
487 ;
488
489 var_assign: VAR '=' NUMBER ',' {
490 | assignIfPossible($1,$3);
491 }
492 | VAR '=' VAR ',' {
493 | if( !doesVariableExists($3) ){
494 | printf("\nVariable %s doesn't exist\n", $3);
495 }
496 | else{
497 | double val = getValueOrDefault($3);
498 | assignIfPossible($1, val);
499 }
500 }
501 | VAR '=' func_call ',' {
502 | continueToAssignFromFunction($1,true);

```

```

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
521 if_sec: if_condition '{' block END_POINT else_block { printf("\nIf-
522 | else processed\n"); }
523 ;
524
525 if_condition: '(' many_logic_cond ')' { pushValidity($2); }
526 ;
527
528 else_block: ELSE '{' block END_POINT { popValidity(); }
529 | { popValidity(); }
530 ;
531
532 loop_sec: '(' logic_cond ')' '{' loop_updater END_POINT {
533 |     printf("\nLoop matched\n");
534 |     startLoop();
535 | }
536 ;
537
538 loop_updater: VAR '=' arith_exp ':' '{
539 |     lastAssign.var = $1;
540 |     //printf("=== %s %s %s ===", lastCond.left, lastCond.op,
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 | }
547 | many_logic_cond AND logic_cond { $$ = $1 && $3; isLastIfValid=
548 | $$; }
549 ;
550
551 logic_cond: VAR_CON COND_OPE VAR_CON {
552 |     $$ = evaluateCondition($1, $2, $3);
553 |     //printf("-----s %s %s %d\n", $1, $2, $3, $$);
554 | }
555 ;
556
557 arith_exp: VAR ARITH_OPE VAR {
558 |     double val1 = getValueOrDefault($1);
559 |     double val2 = getValueOrDefault($3);
560 |     updateLast($1, $2, $3);
561 |     $$ = doArithOperation(val1, val2, $2);
562 | }
563 | VAR ARITH_OPE NUMBER {
564 |     double val1 = getValueOrDefault($1);
565 |     updateLast($1, $2, dtoc($3) );
566 |     $$ = doArithOperation(val1, $3, $2);
567 | }
568 | NUMBER ARITH_OPE VAR {
569 |     double val2 = getValueOrDefault($3);
570 |     updateLast( dtoc($1), $2, $3);
571 |     $$ = doArithOperation($1, val2, $2);
572 | }
573 | NUMBER ARITH_OPE NUMBER {
574 |     updateLast( dtoc($1), $2, dtoc($3) );
575 |     $$ = doArithOperation($1, $3, $2);
576 | }
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
592     initializeLibraryFunction();
593
594     freopen("input.txt", "r", stdin);
595     //freopen("input2.txt", "r", stdin);
596     freopen("output.txt", "w", stdout); // output in file
597     yyparse();
598     printAll();
599     printf("\nPrinting all prototype\n");
600     printAllProto();
601     printAllLibraryFunction();
602     printAllImports();
603     return 0;
604 }
605
```