```
%{
    #include<stdio.h>
    #include<string.h>
    #include<stdlib.h>
    #include<ctype.h>
    #include"lex.yy.c"
    void yyerror(const char *s);
    int yylex();
    int yywrap();
    void add(char);
    void insert_type();
    int search(char *);
        void insert_type();
        void print_tree(struct node*);
        void print_inorder(struct node *);
    void check_declaration(char *);
        void check_return_type(char *);
        int check_types(char *, char *);
        char *get_type(char *);
        struct node* mknode(struct node *left, struct node *right, char *token);

    struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        int line_no;
        } symbol_table[40];

    int count=0;
    int q;
        char type[10];
    extern int countn;
        struct node *head;
        int sem_errors=0;
        int ic_idx=0;
        int temp_var=0;
        int label=0;
        int is_for=0;
        char buff[100];
        char errors[10][100];
        char reserved[10][10] = {"int", "float", "char", "void", "if", "else", "for", "main", "return", "include"};
        char icg[50][100];

        struct node {
                struct node *left;
                struct node *right;
                char *token;
        };

%}

%union { struct var_name {
                        char name[100];
                        struct node* nd;
                } nd_obj;

                struct var_name2 {
                        char name[100];
                        struct node* nd;
                        char type[5];
                } nd_obj2;

                struct var_name3 {
                        char name[100];
                        struct node* nd;
                        char if_body[5];
                        char else_body[5];
                } nd_obj3;
        }
%token VOID
%token <nd_obj> CHARACTER PRINTFF SCANFF INT FLOAT CHAR FOR IF ELSE TRUE FALSE NUMBER FLOAT_NUM ID LE GE EQ NE GT LT AND OR STR ADD MULTIPLY
DIVIDE SUBTRACT UNARY INCLUDE RETURN
%type <nd_obj> headers main body return datatype statement arithmetic relop program else
%type <nd_obj2> init value expression
%type <nd_obj3> condition

%%

program: headers main '(' ')' '{' body return '}' { $2.nd = mknode($6.nd, $7.nd, "main"); $$.nd = mknode($1.nd, $2.nd, "program");
        head = $$.nd;
}
;

headers: headers headers { $$.nd = mknode($1.nd, $2.nd, "headers"); }
| INCLUDE { add('H'); } { $$.nd = mknode(NULL, NULL, $1.name); }
```

```
;

main: datatype ID { add('F'); }
;

datatype: INT { insert_type(); }
| FLOAT { insert_type(); }
| CHAR { insert_type(); }
| VOID { insert_type(); }
;

body: FOR { add('K'); is_for = 1; } '(' statement ';' condition ';' statement ')' '{' body '}' {
        struct node *temp = mknode($6.nd, $8.nd, "CONDITION");
        struct node *temp2 = mknode($4.nd, temp, "CONDITION");
        $$.nd = mknode(temp2, $11.nd, $1.name);
        sprintf(icg[ic_idx++], buff);
        sprintf(icg[ic_idx++], "JUMP to %s\n", $6.if_body);
        sprintf(icg[ic_idx++], "\nLABEL %s:\n", $6.else_body);
}
| IF { add('K'); is_for = 0; } '(' condition ')' { sprintf(icg[ic_idx++], "\nLABEL %s:\n", $4.if_body); } '{' body '}' {
sprintf(icg[ic_idx++], "\nLABEL %s:\n", $4.else_body); } else {
        struct node *iff = mknode($4.nd, $8.nd, $1.name);
        $$.nd = mknode(iff, $11.nd, "if-else");
        sprintf(icg[ic_idx++], "GOTO next\n");
}
| statement ';' { $$.nd = $1.nd; }
| body body { $$.nd = mknode($1.nd, $2.nd, "statements"); }
| PRINTFF { add('K'); } '(' STR ')' ';' { $$.nd = mknode(NULL, NULL, "printf"); }
| SCANFF { add('K'); } '(' STR ',' '&' ID ')' ';' { $$.nd = mknode(NULL, NULL, "scanf"); }
;

else: ELSE { add('K'); } '{' body '}' { $$.nd = mknode(NULL, $4.nd, $1.name); }
| { $$.nd = NULL; }
;

condition: value relop value {
        $$.nd = mknode($1.nd, $3.nd, $2.name);
        if(is_for) {
                sprintf($$.if_body, "L%d", label++);
                sprintf(icg[ic_idx++], "\nLABEL %s:\n", $$.if_body);
                sprintf(icg[ic_idx++], "\nif NOT (%s %s %s) GOTO L%d\n", $1.name, $2.name, $3.name, label);
                sprintf($$.else_body, "L%d", label++);
        } else {
                sprintf(icg[ic_idx++], "\nif (%s %s %s) GOTO L%d else GOTO L%d\n", $1.name, $2.name, $3.name, label, label+1);
                sprintf($$.if_body, "L%d", label++);
                sprintf($$.else_body, "L%d", label++);
        }
}
| TRUE { add('K'); $$.nd = NULL; }
| FALSE { add('K'); $$.nd = NULL; }
| { $$.nd = NULL; }
;

statement: datatype ID { add('V'); } init {
        $2.nd = mknode(NULL, NULL, $2.name);
        int t = check_types($1.name, $4.type);
        if(t>0) {
                if(t == 1) {
                        struct node *temp = mknode(NULL, $4.nd, "floattoint");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 2) {
                        struct node *temp = mknode(NULL, $4.nd, "inttofloat");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 3) {
                        struct node *temp = mknode(NULL, $4.nd, "chartoint");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 4) {
                        struct node *temp = mknode(NULL, $4.nd, "inttochar");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else if(t == 5) {
                        struct node *temp = mknode(NULL, $4.nd, "chartofloat");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
                else{
                        struct node *temp = mknode(NULL, $4.nd, "floattochar");
                        $$.nd = mknode($2.nd, temp, "declaration");
                }
        }
        else {
                $$.nd = mknode($2.nd, $4.nd, "declaration");
        }
```

```
        sprintf(icg[ic_idx++], "%s = %s\n", $2.name, $4.name);
}
| ID { check_declaration($1.name); } '=' expression {
        $1.nd = mknode(NULL, NULL, $1.name);
        char *id_type = get_type($1.name);
        if(strcmp(id_type, $4.type)) {
                if(!strcmp(id_type, "int")) {
                        if(!strcmp($4.type, "float")){
                                struct node *temp = mknode(NULL, $4.nd, "floattoint");
                                $$.nd = mknode($1.nd, temp, "=");
                        }
                        else{
                                struct node *temp = mknode(NULL, $4.nd, "chartoint");
                                $$.nd = mknode($1.nd, temp, "=");
                        }

                }
                else if(!strcmp(id_type, "float")) {
                        if(!strcmp($4.type, "int")){
                                struct node *temp = mknode(NULL, $4.nd, "inttofloat");
                                $$.nd = mknode($1.nd, temp, "=");
                        }
                        else{
                                struct node *temp = mknode(NULL, $4.nd, "chartofloat");
                                $$.nd = mknode($1.nd, temp, "=");
                        }

                }
                else{
                        if(!strcmp($4.type, "int")){
                                struct node *temp = mknode(NULL, $4.nd, "inttochar");
                                $$.nd = mknode($1.nd, temp, "=");
                        }
                        else{
                                struct node *temp = mknode(NULL, $4.nd, "floattochar");
                                $$.nd = mknode($1.nd, temp, "=");
                        }
                }
        }
        else {
                $$.nd = mknode($1.nd, $4.nd, "=");
        }
        sprintf(icg[ic_idx++], "%s = %s\n", $1.name, $4.name);
}
| ID { check_declaration($1.name); } relop expression { $1.nd = mknode(NULL, NULL, $1.name); $$.nd = mknode($1.nd, $4.nd, $3.name); }
| ID { check_declaration($1.name); } UNARY {
        $1.nd = mknode(NULL, NULL, $1.name);
        $3.nd = mknode(NULL, NULL, $3.name);
        $$.nd = mknode($1.nd, $3.nd, "ITERATOR");
        if(!strcmp($3.name, "++")) {
                sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $1.name, $1.name, temp_var++);
        }
        else {
                sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $1.name, $1.name, temp_var++);
        }
}
| UNARY ID {
        check_declaration($2.name);
        $1.nd = mknode(NULL, NULL, $1.name);
        $2.nd = mknode(NULL, NULL, $2.name);
        $$.nd = mknode($1.nd, $2.nd, "ITERATOR");
        if(!strcmp($1.name, "++")) {
                sprintf(buff, "t%d = %s + 1\n%s = t%d\n", temp_var, $2.name, $2.name, temp_var++);
        }
        else {
                sprintf(buff, "t%d = %s - 1\n%s = t%d\n", temp_var, $2.name, $2.name, temp_var++);

        }
}
;

init: '=' value { $$.nd = $2.nd; sprintf($$.type, $2.type); strcpy($$.name, $2.name); }
| { sprintf($$.type, "null"); $$.nd = mknode(NULL, NULL, "NULL"); strcpy($$.name, "NULL"); }
;

expression: expression arithmetic expression {
        if(!strcmp($1.type, $3.type)) {
                sprintf($$.type, $1.type);
                $$.nd = mknode($1.nd, $3.nd, $2.name);
        }
        else {
                if(!strcmp($1.type, "int") && !strcmp($3.type, "float")) {
                        struct node *temp = mknode(NULL, $1.nd, "inttofloat");
                        sprintf($$.type, $3.type);
                        $$.nd = mknode(temp, $3.nd, $2.name);
```

```
                }
                else if(!strcmp($1.type, "float") && !strcmp($3.type, "int")) {
                        struct node *temp = mknode(NULL, $3.nd, "inttofloat");
                        sprintf($$.type, $1.type);
                        $$.nd = mknode($1.nd, temp, $2.name);
                }
                else if(!strcmp($1.type, "int") && !strcmp($3.type, "char")) {
                        struct node *temp = mknode(NULL, $3.nd, "chartoint");
                        sprintf($$.type, $1.type);
                        $$.nd = mknode($1.nd, temp, $2.name);
                }
                else if(!strcmp($1.type, "char") && !strcmp($3.type, "int")) {
                        struct node *temp = mknode(NULL, $1.nd, "chartoint");
                        sprintf($$.type, $3.type);
                        $$.nd = mknode(temp, $3.nd, $2.name);
                }
                else if(!strcmp($1.type, "float") && !strcmp($3.type, "char")) {
                        struct node *temp = mknode(NULL, $3.nd, "chartofloat");
                        sprintf($$.type, $1.type);
                        $$.nd = mknode($1.nd, temp, $2.name);
                }
                else {
                        struct node *temp = mknode(NULL, $1.nd, "chartofloat");
                        sprintf($$.type, $3.type);
                        $$.nd = mknode(temp, $3.nd, $2.name);
                }
        }
        sprintf($$.name, "t%d", temp_var);
        temp_var++;
        sprintf(icg[ic_idx++], "%s = %s %s %s\n",  $$.name, $1.name, $2.name, $3.name);
}
| value { strcpy($$.name, $1.name); sprintf($$.type, $1.type); $$.nd = $1.nd; }
;

arithmetic: ADD
| SUBTRACT
| MULTIPLY
| DIVIDE
;

relop: LT
| GT
| LE
| GE
| EQ
| NE
;

value: NUMBER { strcpy($$.name, $1.name); sprintf($$.type, "int"); add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| FLOAT_NUM { strcpy($$.name, $1.name); sprintf($$.type, "float"); add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| CHARACTER { strcpy($$.name, $1.name); sprintf($$.type, "char"); add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| ID { strcpy($$.name, $1.name); char *id_type = get_type($1.name); sprintf($$.type, id_type); check_declaration($1.name); $$.nd =
mknode(NULL, NULL, $1.name); }
;

return: RETURN { add('K'); } value ';' { check_return_type($3.name); $1.nd = mknode(NULL, NULL, "return"); $$.nd = mknode($1.nd, $3.nd,
"RETURN"); }
| { $$.nd = NULL; }
;

%%

int main() {
    yyparse();
    printf("\n\n");
        printf("\t\t\t\t\t\t\t\t PHASE 1: LEXICAL ANALYSIS \n\n");
        printf("\nSYMBOL   DATATYPE   TYPE   LINE NUMBER \n");
        printf("_____________________________________\n\n");
        int i=0;
        for(i=0; i<count; i++) {
                printf("%s\t%s\t%s\t%d\t\n", symbol_table[i].id_name, symbol_table[i].data_type, symbol_table[i].type,
symbol_table[i].line_no);
        }
        for(i=0;i<count;i++) {
                free(symbol_table[i].id_name);
                free(symbol_table[i].type);
        }
        printf("\n\n");
        printf("\t\t\t\t\t\t\t\t PHASE 2: SYNTAX ANALYSIS \n\n");
        print_tree(head);
        printf("\n\n\n\n");
        printf("\t\t\t\t\t\t\t\t PHASE 3: SEMANTIC ANALYSIS \n\n");
        if(sem_errors>0) {
                printf("Semantic analysis completed with %d errors\n", sem_errors);
                for(int i=0; i<sem_errors; i++){
```

```
                        printf("\t - %s", errors[i]);
                }
        } else {
                printf("Semantic analysis completed with no errors");
        }
        printf("\n\n");
        printf("\t\t\t\t\t\t\t   PHASE 4: INTERMEDIATE CODE GENERATION \n\n");
        for(int i=0; i<ic_idx; i++){
                printf("%s", icg[i]);
        }
        printf("\n\n");
}

int search(char *type) {
        int i;
        for(i=count-1; i>=0; i--) {
                if(strcmp(symbol_table[i].id_name, type)==0) {
                        return -1;
                        break;
                }
        }
        return 0;
}

void check_declaration(char *c) {
    q = search(c);
    if(!q) {
        sprintf(errors[sem_errors], "Line %d: Variable \"%s\" not declared before usage!\n", countn+1, c);
                sem_errors++;
    }
}

void check_return_type(char *value) {
        char *main_datatype = get_type("main");
        char *return_datatype = get_type(value);
        if((!strcmp(main_datatype, "int") && !strcmp(return_datatype, "CONST")) || !strcmp(main_datatype, return_datatype)){
                return ;
        }
        else {
                sprintf(errors[sem_errors], "Line %d: Return type mismatch\n", countn+1);
                sem_errors++;
        }
}

int check_types(char *type1, char *type2){
        // declaration with no init
        if(!strcmp(type2, "null"))
                return -1;
        // both datatypes are same
        if(!strcmp(type1, type2))
                return 0;
        // both datatypes are different
        if(!strcmp(type1, "int") && !strcmp(type2, "float"))
                return 1;
        if(!strcmp(type1, "float") && !strcmp(type2, "int"))
                return 2;
        if(!strcmp(type1, "int") && !strcmp(type2, "char"))
                return 3;
        if(!strcmp(type1, "char") && !strcmp(type2, "int"))
                return 4;
        if(!strcmp(type1, "float") && !strcmp(type2, "char"))
                return 5;
        if(!strcmp(type1, "char") && !strcmp(type2, "float"))
                return 6;
}

char *get_type(char *var){
        for(int i=0; i<count; i++) {
                // Handle case of use before declaration
                if(!strcmp(symbol_table[i].id_name, var)) {
                        return symbol_table[i].data_type;
                }
        }
}

void add(char c) {
        if(c == 'V'){
                for(int i=0; i<10; i++){
                        if(!strcmp(reserved[i], strdup(yytext))){
                        sprintf(errors[sem_errors], "Line %d: Variable name \"%s\" is a reserved keyword!\n", countn+1, yytext);
                                sem_errors++;
                                return;
                        }
                }
        }
```

```
    q=search(yytext);
        if(!q) {
                if(c == 'H') {
                        symbol_table[count].id_name=strdup(yytext);
                        symbol_table[count].data_type=strdup(type);
                        symbol_table[count].line_no=countn;
                        symbol_table[count].type=strdup("Header");
                        count++;
                }
                else if(c == 'K') {
                        symbol_table[count].id_name=strdup(yytext);
                        symbol_table[count].data_type=strdup("N/A");
                        symbol_table[count].line_no=countn;
                        symbol_table[count].type=strdup("Keyword\t");
                        count++;
                }
                else if(c == 'V') {
                        symbol_table[count].id_name=strdup(yytext);
                        symbol_table[count].data_type=strdup(type);
                        symbol_table[count].line_no=countn;
                        symbol_table[count].type=strdup("Variable");
                        count++;
                }
                else if(c == 'C') {
                        symbol_table[count].id_name=strdup(yytext);
                        symbol_table[count].data_type=strdup("CONST");
                        symbol_table[count].line_no=countn;
                        symbol_table[count].type=strdup("Constant");
                        count++;
                }
                else if(c == 'F') {
                        symbol_table[count].id_name=strdup(yytext);
                        symbol_table[count].data_type=strdup(type);
                        symbol_table[count].line_no=countn;
                        symbol_table[count].type=strdup("Function");
                        count++;
                }
    }
    else if(c == 'V' && q) {
        sprintf(errors[sem_errors], "Line %d: Multiple declarations of \"%s\" not allowed!\n", countn+1, yytext);
                sem_errors++;
    }
}

struct node* mknode(struct node *left, struct node *right, char *token) {
        struct node *newnode = (struct node *)malloc(sizeof(struct node));
        char *newstr = (char *)malloc(strlen(token)+1);
        strcpy(newstr, token);
        newnode->left = left;
        newnode->right = right;
        newnode->token = newstr;
        return(newnode);
}

void print_tree(struct node* tree) {
        printf("\n\nInorder traversal of the Parse Tree is: \n\n");
        print_inorder(tree);
}

void print_inorder(struct node *tree) {
        int i;
        if (tree->left) {
                print_inorder(tree->left);
        }
        printf("%s, ", tree->token);
        if (tree->right) {
                print_inorder(tree->right);
        }
}

void insert_type() {
        strcpy(type, yytext);
}

void yyerror(const char* msg) {
    fprintf(stderr, "%s\n", msg);
}
```