```
%{
    #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 printtree(struct node*);
    void printInorder(struct node *);
    struct node* mknode(struct node *left, struct node *right, char *token);

    struct dataType {
        char * id_name;
        char * data_type;
        char * type;
        int line_no;
    } symbolTable[40];
    int count=0;
    int q;
    char type[10];
    extern int countn;
    struct node *head;
    struct node {
        struct node *left;
        struct node *right;
        char *token;
    };
%}

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

%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 expression statement init value arithmetic relop program condition else

%%

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('K'); }
;

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

body: FOR { add('K'); } '(' statement ';' condition ';' statement ')' '{' body '}' { struct node *temp = mknode($6.nd, $8.nd, "CONDITION");
struct node *temp2 = mknode($4.nd, temp, "CONDITION"); $$.nd = mknode(temp2, $11.nd, $1.name); }
| IF { add('K'); } '(' condition ')' '{' body '}' else { struct node *iff = mknode($4.nd, $7.nd, $1.name);      $$.nd = mknode(iff, $9.nd,
"if-else"); }
| 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 { $$.nd = mknode($1.nd, $3.nd, $2.name); }
| TRUE { add('K'); $$.nd = NULL; }
| FALSE { add('K'); $$.nd = NULL; }
| { $$.nd = NULL; }
;
```

```
statement: datatype ID { add('V'); } init { $2.nd = mknode(NULL, NULL, $2.name); $$.nd = mknode($2.nd, $4.nd, "declaration"); }
| ID '=' expression { $1.nd = mknode(NULL, NULL, $1.name); $$.nd = mknode($1.nd, $3.nd, "="); }
| ID relop expression { $1.nd = mknode(NULL, NULL, $1.name); $$.nd = mknode($1.nd, $3.nd, $2.name); }
| ID UNARY { $1.nd = mknode(NULL, NULL, $1.name); $2.nd = mknode(NULL, NULL, $2.name); $$.nd = mknode($1.nd, $2.nd, "ITERATOR"); }
| UNARY ID { $1.nd = mknode(NULL, NULL, $1.name); $2.nd = mknode(NULL, NULL, $2.name); $$.nd = mknode($1.nd, $2.nd, "ITERATOR"); }
;

init: '=' value { $$.nd = $2.nd; }
| { $$.nd = mknode(NULL, NULL, "NULL"); }
;

expression: expression arithmetic expression { $$.nd = mknode($1.nd, $3.nd, $2.name); }
| value { $$.nd = $1.nd; }
;

arithmetic: ADD
| SUBTRACT
| MULTIPLY
| DIVIDE
;

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

value: NUMBER { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| FLOAT_NUM { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| CHARACTER { add('C'); $$.nd = mknode(NULL, NULL, $1.name); }
| ID { $$.nd = mknode(NULL, NULL, $1.name); }
;

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

%%

int main() {
    yyparse();
    printf("\n\n \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", symbolTable[i].id_name, symbolTable[i].data_type, symbolTable[i].type, symbolTable[i].line_no);
        }
        for(i=0;i<count;i++){
                free(symbolTable[i].id_name);
                free(symbolTable[i].type);
        }
        printf("\n\n");
        printf("\t\t\t\t\t\t PHASE 2: SYNTAX ANALYSIS \n\n");
        printtree(head);
        printf("\n\n");
}

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

void add(char c) {
    q=search(yytext);
        if(q==0) {
                if(c=='H') {
                        symbolTable[count].id_name=strdup(yytext);
                        symbolTable[count].data_type=strdup(type);
                        symbolTable[count].line_no=countn;
                        symbolTable[count].type=strdup("Header");
                        count++;
                }
                else if(c=='K') {
                        symbolTable[count].id_name=strdup(yytext);
                        symbolTable[count].data_type=strdup("N/A");
```

```
                        symbolTable[count].line_no=countn;
                        symbolTable[count].type=strdup("Keyword\t");
                        count++;
                }
                else if(c=='V') {
                        symbolTable[count].id_name=strdup(yytext);
                        symbolTable[count].data_type=strdup(type);
                        symbolTable[count].line_no=countn;
                        symbolTable[count].type=strdup("Variable");
                        count++;
                }
                else if(c=='C') {
                        symbolTable[count].id_name=strdup(yytext);
                        symbolTable[count].data_type=strdup("CONST");
                        symbolTable[count].line_no=countn;
                        symbolTable[count].type=strdup("Constant");
                        count++;
                }
    }
}

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 printtree(struct node* tree) {
        printf("\n\n Inorder traversal of the Parse Tree: \n\n");
        printInorder(tree);
        printf("\n\n");
}

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

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

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