Compiler design Lab-3 Task

Name:Arjun N R

SRN: PES2UG22CS910

Lexer.l file

```
%{
   #define YYSTYPE char*
   #include "y.tab.h"
   #include <stdio.h>
   extern void yyerror(const char *); // declare the error handling function

%}

/* Regular definitions */
digit   [0-9]
letter  [a-zA-Z]
id      {letter}({letter}|{digit})*
digits  {digit}+
opFraction   (\.{digits})?
opExponent ([Ee][+-]?{digits})?
number      {digits}{opFraction}{opExponent}
%option yylineno

%%
\/\/(.*) ; // ignore comments
[\t\n] ; // ignore whitespaces
```

```
"int"           {return T_INT;}
"char"                  {return T_CHAR;}
"double"        {return T_DOUBLE;}
"float"                 {return T_FLOAT;}
"while"         {return T_WHILE;}
"if"            {return T_IF;}
"else"          {return T_ELSE;}
"do"            {return T_DO;}
"#include"      {return T_INCLUDE;}
"main"                  {return T_MAIN;}
\".*\"          {yylval=strdup(yytext); return T_STRLITERAL; }
"=="            {return T_EQCOMP;}
"!="            {return T_NOTEQUAL;}
">="            {return T_GREATEREQ;}
"<="            {return T_LESSEREQ;}
"("             {return *yytext;}
")"             {return *yytext;}
"."             {return *yytext;}
","             {return *yytext;}
"{"             {return *yytext;}
"}"             {return *yytext;}
"*"             {return *yytext;}
"+"             {return *yytext;}
";"             {return *yytext;}
"-"             {return *yytext;}
"/"             {return *yytext;}
```

```
"="             {return *yytext;}
">"             {return *yytext;}
"<"             {return *yytext;}
{number}    {
                    yylval=strdup(yytext);  //stores the value of the number to be used later for symbol table insertion
                    return T_NUM;
                }
{id}\.h{return T_HEADER;}  // ending in .h => header file name
{id}            {
                    yylval=strdup(yytext); //stores the identifier to be used later for symbol table insertion
                    return T_ID;
                }
.               {} // anything else => ignore
%%

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

Parser.y

```
%{
        #include "sym_tab.h"
        #include "sym_tab.c" // Including .c for simplicity in this example. In real projects, compile and link.
        #include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h> // For isdigit, etc.

#define YYSTYPE char*

void yyerror(char* s);
int yylex();
extern int yylineno;

table* sym_table; // Global symbol table
int current_scope = 0; // Global scope is 1
int current_type; // To store the current type being declared

int get_type_code(char* type_str); // Function to get type code from string
int get_size_from_type(int type_code); // Function to get size from type code

// To track types during expression evaluation
int E_type, T_type, F_type;

int infer_num_type(char* num_str); // Function to infer type of number literal

%}

%token T_INT T_CHAR T_DOUBLE T_WHILE  T_INC T_DEC   T_OROR T_ANDAND T_EQCOMP T_NOTEQUAL T_GREATEREQ T_LESSEREQ T_LEFTSHIFT
```

```
T_RIGHTSHIFT T_PRINTLN T_STRING  T_FLOAT T_BOOLEAN T_IF T_ELSE
T_STRLITERAL T_DO T_INCLUDE T_HEADER T_MAIN T_ID T_NUM


%start START


%%
START : PROG { printf("Valid syntax\n"); display_symbol_table(sym_table);
YYACCEPT; }
      ;


PROG :  MAIN PROG
         | DECLR ';' PROG
         | ASSGN ';' PROG
         | IF_STMT PROG
         | /*epsilon*/
         ;



DECLR : TYPE { current_type = get_type_code($1); free($1); } LISTVAR
         ;



LISTVAR : LISTVAR ',' VAR
           | VAR
           ;


VAR: T_ID '=' EXPR {
```

```
            if (check_symbol_table(sym_table, $1)) {
                fprintf(stderr, "Error: Redeclaration of variable '%s' at line %d\n", $1, yylineno);
            } else {
                int size = get_size_from_type(current_type);
                symbol* sym = init_symbol($1, size, current_type, yylineno, current_scope);
                insert_into_table(sym_table, sym);
                insert_value_to_name(sym_table, $1, $3); // Assign initial value from EXPR
            }
            free($1);
            free($3); // Free EXPR's value
                        }
    | T_ID              {
            if (check_symbol_table(sym_table, $1)) {
                fprintf(stderr, "Error: Redeclaration of variable '%s' at line %d\n", $1, yylineno);
            } else {
                int size = get_size_from_type(current_type);
                symbol* sym = init_symbol($1, size, current_type, yylineno, current_scope);
                insert_into_table(sym_table, sym);
            }
            free($1);
                        }


//assign type here to be returned to the declaration grammar
```

```
TYPE : T_INT { $$ = strdup("int"); }
     | T_FLOAT { $$ = strdup("float"); }
     | T_DOUBLE { $$ = strdup("double"); }
     | T_CHAR { $$ = strdup("char"); }
     ;

/* Grammar for assignment */
ASSGN : T_ID '=' EXPR     {
          if (!check_symbol_table(sym_table, $1)) {
             fprintf(stderr, "Error: Undeclared variable '%s' at line %d\n", $1,
yylineno);
          } else {
             insert_value_to_name(sym_table, $1, $3);
          }
          free($1);
          free($3); // Free EXPR's value
                    }
       ;

EXPR : EXPR REL_OP E   { /* For relational operators - to be implemented later if
needed */ }
     | E          { $$ = strdup($1); free($1); E_type = E_type; } // Pass value and
type of E up
     ;

E : E '+' T   {
          char* val_e = $1;
```

```
        char* val_t = $3;
        double num_e, num_t, result_double;
        char result_str[50];
        int result_type;

        if ((E_type == INT_TYPE || E_type == FLOAT_TYPE || E_type ==
DOUBLE_TYPE) &&
            (T_type == INT_TYPE || T_type == FLOAT_TYPE || T_type ==
DOUBLE_TYPE)) {

            num_e = atof(val_e);
            num_t = atof(val_t);
            result_double = num_e + num_t;

            result_type = (E_type == DOUBLE_TYPE || T_type ==
DOUBLE_TYPE) ? DOUBLE_TYPE :
                    (E_type == FLOAT_TYPE || T_type == FLOAT_TYPE) ?
FLOAT_TYPE : INT_TYPE;
            E_type = result_type; // Set E's type

            if (result_type == INT_TYPE) {
               sprintf(result_str, "%d", (int)result_double);
            } else if (result_type == FLOAT_TYPE) {
               sprintf(result_str, "%.6f", (float)result_double);
            } else { // DOUBLE_TYPE
               sprintf(result_str, "%lf", result_double);
            }
```

```
            $$ = strdup(result_str);
        } else {
            yyerror("Type mismatch in addition");
            $$ = strdup("0");
            E_type = INT_TYPE; // Default error type
        }
        free(val_e);
        free(val_t);
    }
  | E '-' T  {
        char* val_e = $1;
        char* val_t = $3;
        double num_e, num_t, result_double;
        char result_str[50];
        int result_type;

        if ((E_type == INT_TYPE || E_type == FLOAT_TYPE || E_type ==
DOUBLE_TYPE) &&
            (T_type == INT_TYPE || T_type == FLOAT_TYPE || T_type ==
DOUBLE_TYPE)) {

            num_e = atof(val_e);
            num_t = atof(val_t);
            result_double = num_e - num_t;
            result_type = (E_type == DOUBLE_TYPE || T_type ==
DOUBLE_TYPE) ? DOUBLE_TYPE :
```

```
                        (E_type == FLOAT_TYPE || T_type == FLOAT_TYPE) ?
FLOAT_TYPE : INT_TYPE;
                E_type = result_type;


                if (result_type == INT_TYPE) {
                   sprintf(result_str, "%d", (int)result_double);
                } else if (result_type == FLOAT_TYPE) {
                   sprintf(result_str, "%.6f", (float)result_double);
                } else { // DOUBLE_TYPE
                   sprintf(result_str, "%lf", result_double);
                }
                $$ = strdup(result_str);
             } else {
                yyerror("Type mismatch in subtraction");
                $$ = strdup("0");
                E_type = INT_TYPE;
             }
             free(val_e);
             free(val_t);
          }
    | T       { $$ = strdup($1); free($1); E_type = T_type; } // Pass value and type
of T up
    ;



T : T '*' F   {
             char* val_t = $1;
```

```
        char* val_f = $3;
        double num_t, num_f, result_double;
        char result_str[50];
        int result_type;

        if ((T_type == INT_TYPE || T_type == FLOAT_TYPE || T_type ==
DOUBLE_TYPE) &&
            (F_type == INT_TYPE || F_type == FLOAT_TYPE || F_type ==
DOUBLE_TYPE)) {

            num_t = atof(val_t);
            num_f = atof(val_f);
            result_double = num_t * num_f;
            result_type = (T_type == DOUBLE_TYPE || F_type == DOUBLE_TYPE)
? DOUBLE_TYPE :
                    (T_type == FLOAT_TYPE || F_type == FLOAT_TYPE) ?
FLOAT_TYPE : INT_TYPE;
            T_type = result_type;

            if (result_type == INT_TYPE) {
               sprintf(result_str, "%d", (int)result_double);
            } else if (result_type == FLOAT_TYPE) {
               sprintf(result_str, "%.6f", (float)result_double);
            } else { // DOUBLE_TYPE
               sprintf(result_str, "%lf", result_double);
            }
            $$ = strdup(result_str);
```

```
            } else {
                yyerror("Type mismatch in multiplication");
                $$ = strdup("0");
                T_type = INT_TYPE;
            }
            free(val_t);
            free(val_f);
        }
    | T '/' F   {
            char* val_t = $1;
            char* val_f = $3;
            double num_t, num_f, result_double;
            char result_str[50];
            int result_type;

            if ((T_type == INT_TYPE || T_type == FLOAT_TYPE || T_type ==
DOUBLE_TYPE) &&
                (F_type == INT_TYPE || F_type == FLOAT_TYPE || F_type ==
DOUBLE_TYPE)) {
                num_t = atof(val_t);
                num_f = atof(val_f);
                if (num_f == 0.0) {
                    yyerror("Division by zero");
                    $$ = strdup("0");
                    T_type = INT_TYPE;
                } else {
                    result_double = num_t / num_f;
```

```
                result_type = (T_type == DOUBLE_TYPE || F_type ==
DOUBLE_TYPE) ? DOUBLE_TYPE :
                        (T_type == FLOAT_TYPE || F_type == FLOAT_TYPE) ?
FLOAT_TYPE : INT_TYPE;
                T_type = result_type;
                if (result_type == INT_TYPE) {
                   sprintf(result_str, "%d", (int)result_double);
                } else if (result_type == FLOAT_TYPE) {
                   sprintf(result_str, "%.6f", (float)result_double);
                } else { // DOUBLE_TYPE
                   sprintf(result_str, "%lf", result_double);
                }
                $$ = strdup(result_str);
             }
          } else {
             yyerror("Type mismatch in division");
             $$ = strdup("0");
             T_type = INT_TYPE;
          }
          free(val_t);
          free(val_f);
       }
   | F       { $$ = strdup($1); free($1); T_type = F_type; } // Pass value and type
of F up
   ;
```

```
F : '(' EXPR ')' { $$ = strdup($2); free($2); F_type = E_type; } // Type of F is type of EXPR
   | T_ID       {
                symbol* sym = get_symbol_entry(sym_table, $1);
                if (sym == NULL) {
                   yyerror("Undeclared variable in expression");
                   $$ = strdup("0");
                   F_type = INT_TYPE; //Default error type
                } else if (sym->val == NULL) {
                   yyerror("Variable not initialized in expression");
                   $$ = strdup("0");
                   F_type = sym->type; //Or default to INT_TYPE?
                }
                else {
                   $$ = strdup(sym->val);
                   F_type = sym->type;
                }
                free($1);
               }
   | T_NUM      {
                $$ = strdup($1);
                F_type = infer_num_type($1); // Infer type of number literal
                free($1);
               }
   | T_STRLITERAL { $$ = strdup($1); free($1); F_type = CHAR_TYPE; /* or STRING_TYPE if you define one */ } // Treat string literal as char type for now
   ;
```

```
REL_OP :   T_LESSEREQ
           | T_GREATEREQ
           | '<'
           | '>'
           | T_EQCOMP
           | T_NOTEQUAL
           ;

// Grammar for IF statement
IF_STMT : T_IF '(' EXPR ')' STMT     {
          printf("IF statement parsed.\n");
          free($3); // Free EXPR's value
        }
        | T_IF '(' EXPR ')' STMT T_ELSE STMT {
          printf("IF-ELSE statement parsed.\n");
          free($3); // Free EXPR's value
        }
        ;



/* Grammar for main function */
MAIN : TYPE T_MAIN '(' EMPTY_LISTVAR ')' '{'
       { current_scope++; } // Increment scope when entering MAIN's body
       STMT
       { current_scope--; } // Decrement scope when exiting MAIN's body
```

```
    '}'
   ;

EMPTY_LISTVAR : LISTVAR
              |
              ;

STMT : STMT_NO_BLOCK STMT
     | BLOCK STMT
     | IF_STMT STMT
     | /*epsilon*/
     ;


STMT_NO_BLOCK : DECLR ';'
     | ASSGN ';'
     ;

BLOCK : '{'
       { current_scope++; } // Increment scope when entering BLOCK
       STMT
       { current_scope--; } // Decrement scope when exiting BLOCK
     '}'
     ;

COND : EXPR
```

```
      | ASSGN
      ;


%%


/* error handling function */
void yyerror(char* s)
{
         printf("Error :%s at line %d\n",s,yylineno);
}

int get_type_code(char* type_str) {
   if (strcmp(type_str, "int") == 0) return INT_TYPE;
   if (strcmp(type_str, "char") == 0) return CHAR_TYPE;
   if (strcmp(type_str, "float") == 0) return FLOAT_TYPE;
   if (strcmp(type_str, "double") == 0) return DOUBLE_TYPE;
   return 0; // Unknown type
}

int get_size_from_type(int type_code) {
   switch (type_code) {
      case CHAR_TYPE: return 1;
      case INT_TYPE: return 2;
      case FLOAT_TYPE: return 4;
```

```
        case DOUBLE_TYPE: return 8;
        default: return 0; // Unknown size
    }
}

// Infer type of number literal (T_NUM)
int infer_num_type(char* num_str) {
    int is_float = 0;
    for (int i = 0; num_str[i] != '\0'; i++) {
        if (num_str[i] == '.' || tolower(num_str[i]) == 'e') {
            is_float = 1;
            break;
        }
    }
    if (is_float) {
        return FLOAT_TYPE; // Or DOUBLE_TYPE if you want to default to double for floating point literals
    } else {
        return INT_TYPE;
    }
}


int main(int argc, char *argv[]) {
    // If an input file is provided as a command line argument, open it.
    if (argc > 1) {
        FILE *fp = freopen(argv[1], "r", stdin);
```

```
        if (!fp) {
            perror("Error opening file");
            exit(EXIT_FAILURE);
        }
    }
    sym_table = init_table();
    yyparse();
    return 0;
}
```

Symbol_table.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "sym_tab.h"

// Global symbol table instantiation (definition)
table* sym_table;

table* init_table()
{
    table* t = (table*)malloc(sizeof(table));
    if (t == NULL) {
        fprintf(stderr, "Memory allocation failed for symbol table.\n");
        exit(EXIT_FAILURE);
    }
```

```
    t->head = NULL;
    return t;
}

symbol* init_symbol(char* name, int size, int type, int lineno, int scope)
{
    symbol* s = (symbol*)malloc(sizeof(symbol));
    if (s == NULL) {
        fprintf(stderr, "Memory allocation failed for symbol entry.\n");
        exit(EXIT_FAILURE);
    }
    s->name = strdup(name); // Allocate memory and copy name
    s->size = size;
    s->type = type;
    s->val = NULL; // Value will be updated later, initialized to NULL
    s->line = lineno;
    s->scope = scope;
    s->next = NULL;
    return s;
}

void insert_into_table(table* sym_table, symbol* sym_entry)
{
    if (sym_table == NULL || sym_entry == NULL) {
        fprintf(stderr, "Invalid arguments to insert_into_table.\n");
        return;
```

```
    }

    if (sym_table->head == NULL) {
        sym_table->head = sym_entry;
    } else {
        symbol* current = sym_table->head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = sym_entry;
    }
}

int check_symbol_table(table* sym_table, char* name)
{
    return get_symbol_entry(sym_table, name) != NULL; // Reuse
get_symbol_entry
}

symbol* get_symbol_entry(table* sym_table, char* name)
{
    if (sym_table == NULL || name == NULL) {
        return NULL; // Not found or invalid table
    }

    symbol* current = sym_table->head;
    while (current != NULL) {
```

```
        if (strcmp(current->name, name) == 0) {
            return current; // Found, return the symbol entry
        }
        current = current->next;
    }
    return NULL; // Not found
}

void insert_value_to_name(table* sym_table, char* name, char* value)
{
    if (sym_table == NULL || name == NULL) {
        return;
    }
    if (value == NULL) return; // if value is default value return back

    symbol* entry = get_symbol_entry(sym_table, name);
    if (entry != NULL) {
        if (entry->val != NULL) free(entry->val); // Free old value if exists
        entry->val = strdup(value); //strdup allocates memory and copies the string
        return;
    }
    printf("Warning: Variable '%s' not found in symbol table to update value.\n",
name);
}

void display_symbol_table(table* sym_table)
{
```

```
    if (sym_table == NULL) {
        printf("Symbol table is NULL.\n");
        return;
    }

    printf("Symbol Table:\n");
    printf("------------------------------------------------------------------\n");
    printf("Name\tSize\tType\tLine No\tScope\tValue\n");
    printf("------------------------------------------------------------------\n");

    symbol* current = sym_table->head;
    while (current != NULL) {
        char* type_str;
        switch (current->type) {
            case CHAR_TYPE: type_str = "char"; break;
            case INT_TYPE: type_str = "int"; break;
            case FLOAT_TYPE: type_str = "float"; break;
            case DOUBLE_TYPE: type_str = "double"; break;
            default: type_str = "unknown"; break;
        }
        printf("%s\t%d\t%s\t%d\t%d\t%s\n", current->name, current->size,
type_str, current->line, current->scope, current->val != NULL ? current->val :
"~");
        current = current->next;
    }
    printf("------------------------------------------------------------------\n");
}
```

Symbol_table.h

```
#ifndef SYM_TAB_H
#define SYM_TAB_H

#define CHAR_TYPE 1
#define INT_TYPE 2
#define FLOAT_TYPE 3
#define DOUBLE_TYPE 4

typedef struct symbol
{
   char* name;
   int size;
   int type;
   char* val; // Value stored as string for simplicity in this example
   int line;
   int scope;
   struct symbol* next;
} symbol;

typedef struct table
{
   symbol* head;
} table;
```

```
extern table* sym_table; // Declare the global symbol table


table* init_table();
symbol* init_symbol(char* name, int size, int type, int lineno, int scope);
void insert_into_table(table* sym_table, symbol* sym_entry);
void insert_value_to_name(table* sym_table, char* name, char* value);
int check_symbol_table(table* sym_table, char* name);
symbol* get_symbol_entry(table* sym_table, char* name);
void display_symbol_table(table* sym_table);


#endif
```

Output screenshot: