Compiler Design Lab-4

Abstract Syntax tree generation

Name: Arjun N R

SRN: PES2UG22CS910

Lexer file

```
%{
   #define YYSTYPE char*
   #include <unistd.h>
   #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
"("             {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}            {
                                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 file

```
%{
    #include "abstract_syntax_tree.h"  // Include the header file
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    void yyerror(char* s);
    int yylex();
    extern int yylineno;
    extern FILE* yyin; // Declare the input file stream for the lexer
%}

%union {
    char* text;
    expression_node* exp_node;
}

%token <text> T_ID T_NUM

%type <exp_node> E T F ASSGN START

%start START
```

```
%%

START : ASSGN   { display_exp_tree($1); printf("Valid syntax\n"); YYACCEPT; }

ASSGN : T_ID '=' E  { $$ = init_exp_node("=", init_exp_node($1, NULL, NULL),
$3); }
     ;

E : E '+' T    { $$ = init_exp_node("+", $1, $3); }
  | E '-' T    { $$ = init_exp_node("-", $1, $3); }
  | T          { $$ = $1; }
  ;

T : T '*' F    { $$ = init_exp_node("*", $1, $3); }
  | T '/' F    { $$ = init_exp_node("/", $1, $3); }
  | F          { $$ = $1; }
  ;

F : '(' E ')'   { $$ = $2; }
  | T_ID       { $$ = init_exp_node($1, NULL, NULL); }
  | T_NUM      { $$ = init_exp_node($1, NULL, NULL); }
  ;

%%

void yyerror(char* s) {
```

```
    printf("Error :%s at %d \n", s, yylineno);
}

int main(int argc, char* argv[]) {
    if (argc < 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE* file = fopen(argv[1], "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    yyin = file; // Redirect lexer input to the file
    yyparse();   // Start parsing
    fclose(file); // Close the file after parsing
    return 0;
}
```

Abstract_syntax_tree.h

```
// abstract_syntax_tree.h
#ifndef ABSTRACT_SYNTAX_TREE_H
#define ABSTRACT_SYNTAX_TREE_H
```

```
typedef struct expression_node {
   char* value;              // String to store the node's value (e.g., operator, number, identifier)
   struct expression_node* left;  // Pointer to the left child
   struct expression_node* right; // Pointer to the right child
} expression_node;

expression_node* init_exp_node(char* val, expression_node* left, expression_node* right);
void display_exp_tree(expression_node* exp_node);

#endif
```

Abstract_syntax_tree.c

```
// abstract_syntax_tree.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "abstract_syntax_tree.h"

expression_node* init_exp_node(char* val, expression_node* left, expression_node* right) {
   // Allocate memory for a new AST node
   expression_node* node = (expression_node*)malloc(sizeof(expression_node));
```

```
    if (node == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(1);
    }
    // Copy the value string to ensure persistence
    node->value = strdup(val);
    if (node->value == NULL) {
        fprintf(stderr, "String duplication failed\n");
        free(node);
        exit(1);
    }
    node->left = left;   // Set the left child
    node->right = right; // Set the right child
    return node;
}

void display_exp_tree(expression_node* exp_node) {
    // Traverse the AST in preorder: root, left, right
    if (exp_node != NULL) {
        printf("%s \n", exp_node->value);    // Print the current node's value
        display_exp_tree(exp_node->left);  // Recurse on left child
        display_exp_tree(exp_node->right); // Recurse on right child
    }
}
```

Output:

```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Lab4\Lab 4 (AST Generation)> ./a.exe .\test_input_1.c
=
a
+
-
/
10
5
*
2
7
3
Valid syntax
```

```
PS C:\Users\arjun\Documents\SEM-6\CD\CompilerDesign\Lab4\Lab 4 (AST Generation)> ./a.exe .\test_input_2.c
=
b
-
+
/
c
6.7
12.45
*
a
1234.0
Valid syntax
```