CD Lab-8 Assignment

Name: Ketan Goud
Reg No: 220905260
Section: D D2
Roll no: 39

Q1. Design the recursive descent parser to parse C program with variable declaration and decision statements with error reporting of grammar 7.1.
Program -> main() { declarations statement_list }
Declarations -> data_type identifier_list; declarations | E
data_type -> int | char
identifier_list -> id | id, identifier_list | id[number]| id[number], identifier_list
statement_list -> statement statement_list | E
statement -> assign_stat; | decision_stat
assign_stat -> id=expn
expn -> simple_expn eprime
eprime -> relop simple_expn | E
simple_expn -> term seprime
seprime -> addop term seprime | E
term -> factor tprime
tprime -> mulop factor tprime | E
factor -> id | num
decision_stat -> if (expn) { statement_list } dprime
dprime -> else { statement_list } | E
relop -> == | != | >= | <= | > | <
addop -> + | -
mulop -> * | / | %

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

// Function declarations
void program();
void declarations();
void data_type();
void identifier_list();
void statement_list();
void statement();
void assign_stat();
void expn();
void eprime();
void simple_expn();
void seprime();
void term();
void tprime();
void factor();
void decision_stat();
void dprime();
void relop();
void addop();
void mulop();
struct token getNextToken();
void error(const char *msg);
```

```
struct token
{
   char token[50];
   int row;
   int col;
   int index;
   char type[20];
};
struct token table[200];
struct symbol
{
   int sno;
   char lexeme[50];
   char datatype[20];
   int size;
};
struct symbol symtable[20];
int symind = 0;
struct function
{
   int sno;
   char function[20];
   char returntype[20];
   char parameters[50];
   int num;
};
struct function functable[20];
int funcind = 0;

const char *keywords[] = {
   "int", "float", "double", "char", "if", "else", "for", "return",
   "while", "void", "switch", "case", "break", "continue",
   "default", "struct", "union", "enum", "long", "short", "const",
   "sizeof", "printf", "scanf"};

#define NUM_KEYWORDS 24

int iskeyword(char *word)
{
   for (int i = 0; i < NUM_KEYWORDS; i++)
   {
      if (strcmp(word, keywords[i]) == 0)
      {
         return 1;
      }
   }
   return 0;
}

int tokenexists(char *word, int ind)
{
   for (int i = 0; i < ind; i++)
   {
      if (strcmp(word, table[i].token) == 0)
      {
         return table[i].index;
```

```
        }
    }
    return -1;
}

int getdatatypesize(char *word)
{
    if (strcmp(word, "int") == 0)
        return 4;
    else if (strcmp(word, "float") == 0)
        return 4;
    else if (strcmp(word, "double") == 0)
        return 8;
    else if (strcmp(word, "char") == 0)
        return 1;
    else
        return 0;
}

void addtotable(char *lexeme, char *datatype, int arraysize)
{
    if (iskeyword(lexeme))
    {
        return;
    }

    int size = getdatatypesize(datatype) * arraysize;
    for (int i = 0; i < symind; i++)
    {
        if (strcmp(lexeme, symtable[i].lexeme) == 0)
        {
            return;
        }
    }
    symtable[symind].sno = symind + 1;
    strcpy(symtable[symind].lexeme, lexeme);
    strcpy(symtable[symind].datatype, datatype);
    symtable[symind].size = size;
    symind++;
}


void addtofunctable(char *function, char *returntype, char *parameters, int num)
{
    for (int i = 0; i < funcind; i++)
    {
        if (strcmp(function, functable[i].function) == 0)
        {
            return;
        }
    }
    functable[funcind].sno = funcind + 1;
    strcpy(functable[funcind].function, function);
    strcpy(functable[funcind].returntype, returntype);
    strcpy(functable[funcind].parameters, parameters);
    functable[funcind].num = num;
    funcind++;
```

```
}

void get_token(FILE *f1, FILE *f2)
{
   char word[20];
   char datatype[20];
   char number[20];
   char string[30];
   char returntype[20];
   char parameters[30] = "";
   char c, next;
   int i = 0;
   int row = 1, col = 1;
   int cur_row, cur_col;
   int isfunc = 0;
   int arraysize = 1;
   int ind = 0;

   while ((c = getc(f1)) != EOF)
   {
      if (isspace(c))
      {
         if (c == '\n')
         {
            row++;
            col = 1;
         }
         else
         {
            col++;
         }
         continue;
      }
      cur_col = col;
      cur_row = row;

      if (c == '+' || c == '-' || c == '*' || c == '/' || c == '%')
      {
         table[ind].index = ind + 1;
         table[ind].row = cur_row;
         table[ind].col = cur_col;
         table[ind].token[0] = c;
         table[ind].token[1] = '\0';
         strcpy(table[ind].type, "Arithmetic Op");
         fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
         ind++;
         col++;
         continue;
      }

      if (c == '=' || c == '>' || c == '<' || c == '!')
      {
         next = getc(f1);
         if (next == '=')
         {
            col++;
```

```
            table[ind].index = ind + 1;
            table[ind].row = cur_row;
            table[ind].col = cur_col;
            table[ind].token[0] = c;
            table[ind].token[1] = next;
            table[ind].token[2] = '\0';
            strcpy(table[ind].type, "Relational Op");
            fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
            ind++;
            col++;
         }
         else
         {
            ungetc(next, f1);
            if (c == '=')
            {
               table[ind].index = ind + 1;
               table[ind].row = cur_row;
               table[ind].col = cur_col;
               table[ind].token[0] = c;
               table[ind].token[1] = '\0';
               strcpy(table[ind].type, "Assignment Op");
               fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
               ind++;
               col++;
            }
            else
            {
               table[ind].index = ind + 1;
               table[ind].row = cur_row;
               table[ind].col = cur_col;
               table[ind].token[0] = c;
               table[ind].token[1] = '\0';
               strcpy(table[ind].type, "Relational Op");
               fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
               ind++;
               col++;
            }
         }
         continue;
      }

      if (isdigit(c))
      {
         i = 0;
         number[i] = c;
         i++;
         col++;
         c = getc(f1);
         while (isdigit(c) || c == '.')
         {
            number[i] = c;
            i++;
            c = getc(f1);
```

```
            }
            number[i] = '\0';
            ungetc(c, f1);
            strcpy(table[ind].token, number);
            strcpy(table[ind].type, "Numeric");
            table[ind].index = ind + 1;
            table[ind].row = cur_row;
            table[ind].col = cur_col;
            fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
            ind++;
            col += i;
            continue;
         }

         if (isalpha(c) || c == '_')
         {
            i = 0;
            word[i] = c;
            i++;
            c = getc(f1);
            col++;

            while (isalnum(c) || c == '_')
            {
               word[i] = c;
               i++;
               c = getc(f1);
            }
            word[i] = '\0';
            ungetc(c, f1);

            strcpy(table[ind].token, word);
            table[ind].index = ind + 1;
            table[ind].row = cur_row;
            table[ind].col = cur_col;

            if (iskeyword(word))
            {
               strcpy(table[ind].type, "Keyword");
               strcpy(datatype,word);
            }
            else
            {
               strcpy(table[ind].type, "Identifier");
            }

            fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n",
                  table[ind].index, table[ind].token, table[ind].row, table[ind].col, table[ind].type);
            ind++;
            col += i;
            c = getc(f1);
            if (c == '[')
            {
               table[ind].index = ind + 1;
               table[ind].row = cur_row;
               table[ind].col = col;
```

```
        table[ind].token[0] = '[';
        table[ind].token[1] = '\0';
        strcpy(table[ind].type, "Parenthesis");

        fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n",
              table[ind].index, table[ind].token, table[ind].row, table[ind].col, table[ind].type);
        ind++;
        col++;
        int arraysize=1;
        if (fscanf(f1, "%d", &arraysize) == 1)
        {
           table[ind].index = ind + 1;
           table[ind].row = cur_row;
           table[ind].col = col;
           sprintf(table[ind].token, "%d", arraysize);
           strcpy(table[ind].type, "Numeric");

           fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n",
                 table[ind].index, table[ind].token, table[ind].row, table[ind].col, table[ind].type);
           ind++;
           col++;
        }
        c = fgetc(f1);
        if (c == ']')
        {
           table[ind].index = ind + 1;
           table[ind].row = cur_row;
           table[ind].col = col;
           table[ind].token[0] = ']';
           table[ind].token[1] = '\0';
           strcpy(table[ind].type, "Parenthesis");
           if (datatype[0] != '\0')
           {
              addtotable(word, datatype, arraysize);
           }
           fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n",
                 table[ind].index, table[ind].token, table[ind].row, table[ind].col, table[ind].type);
           ind++;
           col++;
        }
        else
        {
           ungetc(c, f1);
        }
     }
     else
     {
        ungetc(c, f1);
     }
     if (datatype[0] != '\0' )
     {
        addtotable(word, datatype, arraysize);
     }

     continue;
  }
```

```
        if (c == '"')
        {
           i = 0;
           string[i] = c;
           i++;
           col++;
           c = getc(f1);
           while (c != '"')
           {
              string[i] = c;
              i++;
              c = getc(f1);
           }
           string[i] = '"';
           i++;
           string[i] = '\0';
           strcpy(table[ind].token, string);
           strcpy(table[ind].type, "String Literal");
           table[ind].index = ind + 1;
           table[ind].row = cur_row;
           table[ind].col = cur_col;
           fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
           ind++;
           col += i;
           continue;
        }

        if (c == '(' || c == ')' || c == '{' || c == '}')
        {
           table[ind].index = ind + 1;
           table[ind].row = cur_row;
           table[ind].col = cur_col;
           table[ind].token[0] = c;
           table[ind].token[1] = '\0';
           strcpy(table[ind].type, "Parenthesis");
           fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
           ind++;
           col++;
           continue;
        }
        if (c == ',')
        {
           table[ind].index = ind + 1;
           table[ind].row = cur_row;
           table[ind].col = cur_col;
           table[ind].token[0] = c;
           table[ind].token[1] = '\0';
           strcpy(table[ind].type, "comma");
           fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
           ind++;
           col++;
        }

        if (c == ';')
```

```
        {
            table[ind].index = ind + 1;
            table[ind].row = cur_row;
            table[ind].col = cur_col;
            table[ind].token[0] = c;
            table[ind].token[1] = '\0';
            strcpy(table[ind].type, "semi colon");
            fprintf(f2, "<%d, '%s', %d, %d, '%s'>\n", table[ind].index, table[ind].token, table[ind].row,
table[ind].col, table[ind].type);
            ind++;
            col++;
            continue;
        }
    }
}

int current_index = 1;
struct token currentToken;
struct token getNextToken()
{
    return table[current_index++];
}
void error(const char *msg)
{
    printf("Syntax Error: %s at token '%s'\n", msg, currentToken.token);
    exit(1);
}

// <Program> -> main() { <declarations> <statement_list> }
void program()
{
    currentToken = getNextToken();
    if (strcmp(currentToken.token, "main") == 0)
    {
        currentToken = getNextToken();
        if (strcmp(currentToken.token, "(") == 0)
        {
            currentToken = getNextToken();
            if (strcmp(currentToken.token, ")") == 0)
            {
                printf("parsed main()\n");
                currentToken = getNextToken();
                if (strcmp(currentToken.token, "{") == 0)
                {
                    currentToken = getNextToken();
                    declarations();
                    statement_list();
                    if (strcmp(currentToken.token, "}") == 0)
                    {
                        printf("Parsing successful!\n");
                        return;
                    }
                    else
                        error("Expected '}'");
                }
                else
                    error("Expected '{'");
```

```
            }
            else
                error("Expected ')'");
        }
        else
            error("Expected '('");
    }
    else
        error("Expected 'main'");
}

void declarations()
{
    if (strcmp(currentToken.type, "Keyword") == 0)
    {
        printf("parsed %s ", currentToken.token);
        data_type();
        identifier_list();

        if (strcmp(currentToken.token, ";") == 0)
        {
            printf(";\n");
            currentToken = getNextToken();
            declarations();
        }
        else
        {
            error("Expected ';'");
        }
    }
}

void data_type()
{
    if (strcmp(currentToken.type, "Keyword") == 0 &&
        (strcmp(currentToken.token, "int") == 0 || strcmp(currentToken.token, "char") == 0))
    {
        currentToken = getNextToken();
    }
    else
    {
        error("Expected data type (int/char)");
    }
}

// <identifier_list> -> id | id, <identifier_list> | id[number] | id[number], <identifier_list>
void identifier_list()
{
    if (strcmp(currentToken.type, "Identifier") == 0)
    {
        printf("%s", currentToken.token);
        currentToken = getNextToken();

        if (strcmp(currentToken.token, "[") == 0)
        {
            printf("[");
            currentToken = getNextToken();
```

```
            if (strcmp(currentToken.type, "Numeric") == 0)
            {
                printf("%s", currentToken.token);
                currentToken = getNextToken();

                if (strcmp(currentToken.token, "]") == 0)
                {
                    printf("]");
                    currentToken = getNextToken();
                }
                else
                    error("Expected ']' after array size");
            }
            else
                error("Expected array size inside '[]'");
        }

        if (strcmp(currentToken.token, ",") == 0)
        {
            printf(", ");
            currentToken = getNextToken();
            identifier_list();
        }
    }
    else
        error("Expected identifier in declaration");
}
// <statement_list> -> <statement> <statement_list> | E
void statement_list()
{
    if (strcmp(currentToken.type, "Identifier") == 0 || strcmp(currentToken.token, "if") == 0)
    {
        statement();
        statement_list();
    }
}
// <statement> -> <assign_stat> ; | <decision_stat>
void statement()
{
    if (strcmp(currentToken.type, "Identifier") == 0)
    {
        assign_stat();
        if (strcmp(currentToken.token, ";") == 0)
        {
            currentToken = getNextToken();
        }
        else
            error("Expected ';'");
    }
    else if (strcmp(currentToken.token, "if") == 0)
    {
        decision_stat();
    }
    else
        error("Invalid statement");
}
```

```
// <assign_stat> -> id = <expn>
void assign_stat()
{
   if (strcmp(currentToken.type, "Identifier") == 0)
   {
      printf("parsed %s", currentToken.token);
      currentToken = getNextToken();
      if (strcmp(currentToken.token, "[") == 0)
      {
         printf("[");
         currentToken = getNextToken();
         if (strcmp(currentToken.type, "Numeric") == 0)
         {
            printf("%s", currentToken.token);
            currentToken = getNextToken();
            if (strcmp(currentToken.token, "]") == 0)
            {
               printf("] ");
               currentToken = getNextToken();
            }
            else
               error("Expected ']' in array index");
         }
         else
            error("Expected numeric index in array access");
      }
      if (strcmp(currentToken.token, "=") == 0)
      {
         printf("= ");
         currentToken = getNextToken();
         printf("%s\n", currentToken.token);
         expn();
      }
      else
         error("Expected '='");
   }
   else
      error("Expected identifier");
}
// <expn> -> <simple_expn> <eprime>
void expn()
{
   simple_expn();
   eprime();
}
// <eprime> -> <relop> <simple_expn> | E
void eprime()
{
   if (strcmp(currentToken.type, "Relational Op") == 0)
   {
      relop();
      simple_expn();
   }
}
// <simple_expn> -> <term> <seprime>
void simple_expn()
```

```
{
   term();
   seprime();
}
// <seprime> -> <addop> <term> <seprime> | E
void seprime()
{
   if (strcmp(currentToken.type, "Arithmetic Op") == 0)
   {
      addop();
      term();
      seprime();
   }
}
// <term> -> <factor> <tprime>
void term()
{
   factor();
   tprime();
}
// <tprime> -> <mulop> <factor> <tprime> | E
void tprime()
{
   if (strcmp(currentToken.type, "Arithmetic Op") == 0)
   {
      mulop();
      factor();
      tprime();
   }
}
// <factor> -> id | num
void factor()
{
   if (strcmp(currentToken.type, "Identifier") == 0 || strcmp(currentToken.type, "Numeric") == 0)
   {
      currentToken = getNextToken();
   }
   else
      error("Expected identifier or number");
}
// <decision_stat> -> if ( <expn> ) { <statement_list> } <dprime>
void decision_stat()
{
   if (strcmp(currentToken.token, "if") == 0)
   {
      currentToken = getNextToken();
      if (strcmp(currentToken.token, "(") == 0)
      {
         currentToken = getNextToken();
         expn();
         if (strcmp(currentToken.token, ")") == 0)
         {
            currentToken = getNextToken();
            if (strcmp(currentToken.token, "{") == 0)
            {
               currentToken = getNextToken();
               statement_list();
```

```
                if (strcmp(currentToken.token, "}") == 0)
                {
                    currentToken = getNextToken();
                    dprime();
                }
                else
                    error("Expected '}'");
            }
            else
                error("Expected '{'");
        }
        else
            error("Expected ')'");
    }
    else
        error("Expected '('");
}
else
    error("Expected 'if'");
}
// <dprime> -> else { <statement_list> } | E
void dprime()
{
    if (strcmp(currentToken.token, "else") == 0)
    {
        currentToken = getNextToken();
        if (strcmp(currentToken.token, "{") == 0)
        {
            currentToken = getNextToken();
            statement_list();
            if (strcmp(currentToken.token, "}") == 0)
            {
                currentToken = getNextToken();
            }
            else
                error("Expected '}'");
        }
        else
            error("Expected '{'");
    }
}
// Utility functions for operators
void relop()
{
    currentToken = getNextToken();
}
void addop()
{
    currentToken = getNextToken();
}
void mulop()
{
    currentToken = getNextToken();
}
int main()
{
    FILE *f1 = fopen("s1.txt", "r");
```

```
    FILE *f2 = fopen("s2.txt", "w");
    get_token(f1, f2);
    printf("Symbol Table:\n");
    printf("S.No\tLexeme\tDataType\tSize\n");
    for (int i = 0; i < symind; i++)
    {
        printf("%d\t%s\t%s\t\t%d\n", symtable[i].sno, symtable[i].lexeme, symtable[i].datatype,
symtable[i].size);
    }
    program();
    fclose(f1);
    fclose(f2);
    return 0;
}
```

Sample C code:

```
int main() {
    int a, b[10];
    char c;

    a = 5;
    b[2] = a + 3;

    if (a < 10) {
        c = 10;
    } else {
        c = 20;
    }
}
```

```
cd_d2@prg:~/220905260/Lab 8$ cc q1.c
cd_d2@prg:~/220905260/Lab 8$ ./a.out
Symbol Table:
S.No    Lexeme  DataType        Size
1       main    int             4
2       a       int             4
3       b       int             40
4       c       char            1
parsed main()
parsed int a, b[10];
parsed char c;
parsed a= 5
parsed b[2] = a
parsed c= 10
parsed c= 20
Parsing successful!
cd_d2@prg:~/220905260/Lab 8$
```