OBJECTIVES:

- ✓ To create a custom programming language with its own syntax, data types, and operators.
- ✓ To write a Flex program to tokenize source code in the custom language.
- ✓ To configure tokenization rules for keywords, identifiers, constants, and operators.
- ✓ To enhance the Flex program to recognize and handle comments.
- ✓ To implement error detection for undefined variables and syntax issues.
- ✓ To define the custom language's syntax and structure.
- ✓ To provide sample code in the custom language and tokenize it.
- ✓ To count and report the occurrences of keywords, identifiers, constants, and operators.
- ✓ To describe error handling mechanisms in the Flex program.

INTRODUCTION:

In the realm of programming languages, the development of a custom language is a creative and insightful endeavor. This lab explores the creation of a custom programming language, including the design of identifiers and data types, the handling of operators, and the incorporation of loops and directives. Additionally, it implements a lexer using Flex (Lex) to tokenize a source file and effectively detects single-line and multi-line comments.

SOURCE CODE:

```
%{
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int lineno=1;
FILE* output_file;

int keyword_count = 0;
int identifier_count = 0;
int special_symbol_count = 0;
int constant_count = 0;
int operator_count = 0;

char current_type[50];
char variables[100][32];
char constants[100][32];
static int var_len = 0;
static int const_len = 0;

char identifier[100][32];
char datatype[100][32];
union VALUE {
        int INT;
        float FLOAT;
        char CHAR;
        char STRING[100];
};
int choose[100];
static int value_len = 0;
union VALUE value[100];

char* trimWhitespace(char* str) {
   int start = 0, end = strlen(str) - 1;
```

```
    while (isspace(str[start])) {
        start++;
    }

    while ((end >= start) && isspace(str[end])) {
        end--;
    }

    str[end + 1] = '\0';

    return str + start;
}
%}

%option noyywrap

cKeywords((block)|(select)|(option)|(var)|(const)|(return)|(exit)|(resume)|(option)|(true)|(false)|(jump)|(size)|(static)|(try)|(catch)|(not))

%%

[\n]                                                                    { lineno++; }
[ \t] {}

(@).*                                                                   { fprintf(output_file, "%s\t- SINGLE_LINE_COMMENT\n", yytext); }
"<*"([^*]|[\r\n]|(\*+([^*>]|[\r\n])))*"*>"                              { fprintf(output_file, "%s\t- MULTILINE_COMMENT\n", yytext); }

"<*".* {
    printf("Error at Line %d: Unclosed Comment: %s\n", lineno, yytext);
    exit(1);
}

"*>".* {
    printf("Error at Line %d: Unexpected Closing Comment: %s\n", lineno, yytext);
    exit(1);
}

([#][" "]*(define)[" "]*([A-Za-z]+)[" "]*[A-Za-z0-9\.]+) { fprintf(output_file, "%s\t- MACRO\n", yytext); }

\[\[[ ]*[-+]?[0-9]+([ ]*,[ ]*[-+]?[0-9]+)*[ ]*\]\]                      { fprintf(output_file, "%s\t- INTEGER_ARRAY\n", yytext); constant_count++; }
\[\[[ ]*[-+]?[0-9]*\.[0-9]+([ ]*,[ ]*[-+]?[0-9]*\.[0-9]+)*[ ]*\]\]     { fprintf(output_file, "%s\t- FLOAT_ARRAY\n", yytext); constant_count++; }
\[\[[ ]*('[^\']')([ ]*,[ ]*('[^\']'))*[ ]*\]\]                          { fprintf(output_file, "%s\t- CHARACTER_ARRAY\n", yytext); constant_count++; }
\[\[[ ]*(\"[^\"]*\")([ ]*,[ ]*(\"[^\"]*\"))*[ ]*\]\]                    { fprintf(output_file, "%s\t- STRING_ARRAY\n", yytext); constant_count++; }


var|const|assign {
    strcpy(current_type, yytext);
}

[a-zA-Z_][a-zA-Z0-9_]*[ ]*[=][ ]*([-+]?[0-9]+) {
```

```
char *token = strtok(yytext, "=");
token = trimWhitespace(token);

if (token != NULL) {
   for (int i = 1; i <= var_len; i++) {
           if (strcmp(variables[i], token) == 0) {
                   if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                   printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                           exit(1);
                   }
           }
       }

   for (int i = 1; i <= const_len; i++) {
      if (strcmp(constants[i], token) == 0) {
         if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                   printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                   }
      }
   }

         if (strcmp(current_type, "assign") == 0) {
            for (int i = 1; i <= const_len; i++) {
                   if (strcmp(constants[i], token) == 0) {
                           printf("Error at Line %d: '%s' can't be assigned.\n", lineno, token);
                           exit(1);
                   }
            }

            int i=1;

      for (; i <= value_len; i++) {
                   if (strcmp(identifier[i], token) == 0)
                           break;
            }

            if(i == value_len + 1) {
                   printf("Error at Line %d: %s is not assigned before.\n", lineno, token);
                   exit(1);
            }

            token = strtok(NULL, " \t\n\r\f\v");
      if (token != NULL) {
         fprintf(output_file, "%s - INTEGER_CONSTANT\n", token);
         value[i].INT = atoi(token);
         choose[i] = 1;
            }
   }

   else if (strcmp(current_type, "var") == 0) {
         fprintf(output_file, "%s - IDENTIFIER\n", token);
                   var_len++;
         strcpy(variables[var_len], token);
         value_len++;
         strcpy(identifier[value_len], token);
         strcpy(datatype[value_len], "VAR");
```

```
            token = strtok(NULL, " \t\n\r\f\v");
                        token = trimWhitespace(token);
            if (token != NULL) {
               fprintf(output_file, "%s - INTEGER_CONSTANT\n", token);
               value[value_len].INT = atoi(token);
               choose[value_len] = 1;
            }
         }

         else if (strcmp(current_type, "const") == 0) {
            fprintf(output_file, "%s - IDENTIFIER\n", token);
            const_len++;
            strcpy(constants[const_len], token);
            value_len++;
            strcpy(identifier[value_len], token);
            strcpy(datatype[value_len], "CONST");
            token = strtok(NULL, " \t\n\r\f\v");
                  token = trimWhitespace(token);
            if (token != NULL) {
               fprintf(output_file, "%s - INTEGER_CONSTANT\n", token);
               value[value_len].INT = atoi(token);
               choose[value_len] = 1;
            }
         }
      }
   }

[a-zA-Z_][a-zA-Z0-9_]*[ ]*[=][ ]*([-+]?[0-9]*\.[0-9]+) {
   char *token = strtok(yytext, "=");
   token = trimWhitespace(token);

   if (token != NULL) {
      for (int i = 1; i <= var_len; i++) {
            if (strcmp(variables[i], token) == 0) {
                  if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                  printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                        exit(1);
                  }
            }
         }

      for (int i = 1; i <= const_len; i++) {
         if (strcmp(constants[i], token) == 0) {
            if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                  printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                  }
         }
      }

            if (strcmp(current_type, "assign") == 0) {
               for (int i = 1; i <= const_len; i++) {
                  if (strcmp(constants[i], token) == 0) {
                        printf("Error at Line %d: '%s' can't be assigned.\n", lineno, token);
                        exit(1);
                  }
               }
```

```
            int i=1;

        for (; i <= value_len; i++) {
                    if (strcmp(identifier[i], token) == 0)
                                break;
            }

        if(i == value_len + 1) {
                    printf("Error at Line %d: %s is not assigned before.\n", lineno, token);
                    exit(1);
            }

            token = strtok(NULL, " \t\n\r\f\v");
        if (token != NULL) {
            fprintf(output_file, "%s - FLOATING_CONSTANT\n", token);
            value[i].FLOAT = atof(token);
            choose[i] = 2;
                }
    }

    else if (strcmp(current_type, "var") == 0) {
            fprintf(output_file, "%s - IDENTIFIER\n", token);
                    var_len++;
            strcpy(variables[var_len], token);
            value_len++;
            strcpy(identifier[value_len], token);
            strcpy(datatype[value_len], "VAR");
            token = strtok(NULL, " \t\n\r\f\v");
                    token = trimWhitespace(token);
            if (token != NULL) {
                fprintf(output_file, "%s - FLOATING_CONSTANT\n", token);
                value[value_len].FLOAT = atof(token);
                choose[value_len] = 2;
            }
    }

    else if (strcmp(current_type, "const") == 0) {
        fprintf(output_file, "%s - IDENTIFIER\n", token);
        const_len++;
        strcpy(constants[const_len], token);
        value_len++;
        strcpy(identifier[value_len], token);
        strcpy(datatype[value_len], "CONST");
        token = strtok(NULL, " \t\n\r\f\v");
            token = trimWhitespace(token);
        if (token != NULL) {
            fprintf(output_file, "%s - FLOATING_CONSTANT\n", token);
            value[value_len].FLOAT = atof(token);
            choose[value_len] = 2;
        }
    }
  }
}

[a-zA-Z_][a-zA-Z0-9_]*[ ]*[=][ ]*(\'[^\']\') {
```

```
char *token = strtok(yytext, "=");
token = trimWhitespace(token);

if (token != NULL) {
   for (int i = 1; i <= var_len; i++) {
           if (strcmp(variables[i], token) == 0) {
                   if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                   printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                           exit(1);
                   }
           }
       }

   for (int i = 1; i <= const_len; i++) {
      if (strcmp(constants[i], token) == 0) {
         if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                   printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                   }
      }
   }

         if (strcmp(current_type, "assign") == 0) {
            for (int i = 1; i <= const_len; i++) {
                   if (strcmp(constants[i], token) == 0) {
                           printf("Error at Line %d: '%s' can't be assigned.\n", lineno, token);
                           exit(1);
                   }
            }

            int i=1;

      for (; i <= value_len; i++) {
                   if (strcmp(identifier[i], token) == 0)
                           break;
            }

      if(i == value_len + 1) {
                   printf("Error at Line %d: %s is not assigned before.\n", lineno, token);
                   exit(1);
            }

            token = strtok(NULL, " \t\n\r\f\v");
      if (token != NULL) {
         fprintf(output_file, "%s - CHARACTER_CONSTANT\n", token);
         value[i].CHAR = token[1];
         choose[i] = 3;
            }
   }

   else if (strcmp(current_type, "var") == 0) {
         fprintf(output_file, "%s - IDENTIFIER\n", token);
                   var_len++;
         strcpy(variables[var_len], token);
         value_len++;
         strcpy(identifier[value_len], token);
         strcpy(datatype[value_len], "VAR");
```

```
            token = strtok(NULL, " \t\n\r\f\v");
                        token = trimWhitespace(token);
            if (token != NULL) {
               fprintf(output_file, "%s - CHARACTER_CONSTANT\n", token);
               value[value_len].CHAR = token[1];
               choose[value_len] = 3;
            }
      }

      else if (strcmp(current_type, "const") == 0) {
         fprintf(output_file, "%s - IDENTIFIER\n", token);
         const_len++;
         strcpy(constants[const_len], token);
         value_len++;
         strcpy(identifier[value_len], token);
         strcpy(datatype[value_len], "CONST");
         token = strtok(NULL, " \t\n\r\f\v");
               token = trimWhitespace(token);
         if (token != NULL) {
            fprintf(output_file, "%s - CHARACTER_CONSTANT\n", token);
            value[value_len].CHAR = token[1];
            choose[value_len] = 3;
         }
      }
   }
}


[a-zA-Z_][a-zA-Z0-9_]*[ ]*[=][ ]*(\"[^\"]*\") {
   char *token = strtok(yytext, "=");
   token = trimWhitespace(token);

   if (token != NULL) {
      for (int i = 1; i <= var_len; i++) {
               if (strcmp(variables[i], token) == 0) {
                        if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                        printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                                 exit(1);
                        }
               }
            }

      for (int i = 1; i <= const_len; i++) {
         if (strcmp(constants[i], token) == 0) {
            if(strcmp(current_type, "var") == 0 || strcmp(current_type, "const") == 0) {
                        printf("Error at Line %d: '%s' has been defined before.\n", lineno, token);
                        }
         }
      }

            if (strcmp(current_type, "assign") == 0) {
               for (int i = 1; i <= const_len; i++) {
                        if (strcmp(constants[i], token) == 0) {
                                 printf("Error at Line %d: '%s' can't be assigned.\n", lineno, token);
                                 exit(1);
                        }
```

```
            }

            int i=1;

        for (; i <= value_len; i++) {
                    if (strcmp(identifier[i], token) == 0)
                                break;
            }

        if(i == value_len + 1) {
                    printf("Error at Line %d: %s is not assigned before.\n", lineno, token);
                    exit(1);
            }

            token = strtok(NULL, " \t\n\r\f\v");
        if (token != NULL) {
            fprintf(output_file, "%s - STRING_LITERAL\n", token);
            strcpy(value[i].STRING, token);
            choose[i] = 4;
                }
        }

        else if (strcmp(current_type, "var") == 0) {
            fprintf(output_file, "%s - IDENTIFIER\n", token);
                    var_len++;
            strcpy(variables[var_len], token);
            value_len++;
            strcpy(identifier[value_len], token);
            strcpy(datatype[value_len], "VAR");
            token = strtok(NULL, " \t\n\r\f\v");
                    token = trimWhitespace(token);
            if (token != NULL) {
                fprintf(output_file, "%s - STRING_LITERAL\n", token);
                        strcpy(value[value_len].STRING, token);
                choose[value_len] = 4;
            }
        }

        else if (strcmp(current_type, "const") == 0) {
            fprintf(output_file, "%s - IDENTIFIER\n", token);
            const_len++;
            strcpy(constants[const_len], token);
            value_len++;
            strcpy(identifier[value_len], token);
            strcpy(datatype[value_len], "CONST");
            token = strtok(NULL, " \t\n\r\f\v");
                token = trimWhitespace(token);
            if (token != NULL) {
                fprintf(output_file, "%s - STRING_LITERAL\n", token);
                strcpy(value[value_len].STRING, token);
                choose[value_len] = 4;
            }
        }
    }
}
```

```
[a-zA-Z_][a-zA-Z0-9_]*\[[0-9]+\]\[[0-9]+\] {
   printf("Error at Line %d: Multi-dimensional arrays are not allowed: %s\n", lineno, yytext);
   exit(1);
}

[a-zA-Z_][a-zA-Z0-9_]*\[[0-9]+\](\[[0-9]+\])? {
   char baseIdentifier[100];

   if (sscanf(yytext, "%[^[]", baseIdentifier) == 1) {
      int varDefined = 0;

      for (int i = 1; i <= var_len; i++) {
         if (strcmp(variables[i], baseIdentifier) == 0) {
            if (strcmp(current_type, "const") == 0) {
               printf("Error at Line %d: '%s' has been defined as a constant before.\n", lineno, baseIdentifier);
               exit(1);
            }
                        else if (strcmp(current_type, "var") == 0) {
               printf("Warning: '%s' has been defined as a variable before.\n", baseIdentifier);
               varDefined = 1;
               break;
            }
         }
      }

      for (int i = 1; i <= const_len; i++) {
         if (strcmp(constants[i], baseIdentifier) == 0) {
            if (strcmp(current_type, "var") == 0) {
               printf("Error at Line %d: '%s' has been defined as a constant before.\n", lineno, baseIdentifier);
               exit(1);
            }
         }
      }

      if (strcmp(current_type, "var") == 0 && !varDefined) {
         var_len++;
         strcpy(variables[var_len], baseIdentifier);
               fprintf(output_file, "%s\t- IDENTIFIER\n", yytext);
      }

            else if (strcmp(current_type, "const") == 0) {
         const_len++;
         strcpy(constants[const_len], baseIdentifier);
               fprintf(output_file, "%s\t- IDENTIFIER\n", yytext);
      }
   }

   else {
      printf("Failed to extract base identifier.\n");
      exit(1);
   }
}

{cKeywords}                                    { fprintf(output_file, "%s\t- KEYWORD\n", yytext); keyword_count++; }
```

```
"("                         { fprintf(output_file, "%s\t- OPENING BRACKETS\n", yytext); special_symbol_count++; }
")"                         { fprintf(output_file, "%s\t- CLOSING BRACKETS\n", yytext); special_symbol_count++; }
"{"                         { fprintf(output_file, "%s\t- OPENING BRACES\n", yytext); special_symbol_count++; }
"}"                         { fprintf(output_file, "%s\t- CLOSING BRACES\n", yytext); special_symbol_count++; }
";"                         { fprintf(output_file, "%s\t- SEMICOLON\n", yytext); special_symbol_count++; }
","                         { fprintf(output_file, "%s\t- COMMA\n", yytext); special_symbol_count++; }
">>"                        { fprintf(output_file, "%s\t- INPUT\n", yytext); special_symbol_count++; }
"<<"                        { fprintf(output_file, "%s\t- OUTPUT\n", yytext); special_symbol_count++; }

"+"                         { fprintf(output_file, "%s\t- PLUS\n", yytext); operator_count++; }
"-"                         { fprintf(output_file, "%s\t- MINUS\n", yytext); operator_count++; }
"*"                         { fprintf(output_file, "%s\t- MULTIPLY\n", yytext); operator_count++; }
"**"                        { fprintf(output_file, "%s\t- POWER\n", yytext); operator_count++; }
"/"                         { fprintf(output_file, "%s\t- DIVIDE\n", yytext); operator_count++; }
"%"                         { fprintf(output_file, "%s\t- MODULO\n", yytext); operator_count++; }
"="                         { fprintf(output_file, "%s\t- ASSIGNMENT\n", yytext); operator_count++; }

"<"                         { fprintf(output_file, "%s\t- LESS_THAN\n", yytext); operator_count++; }
"<="                        { fprintf(output_file, "%s\t- LESS_THAN_OR_EQUAL\n", yytext); operator_count++; }
">"                         { fprintf(output_file, "%s\t- GREATER_THAN\n", yytext); operator_count++; }
">="                        { fprintf(output_file, "%s\t- GREATER_THAN_OR_EQUAL\n", yytext); operator_count++; }
"=="                        { fprintf(output_file, "%s\t- EQUALS\n", yytext); operator_count++; }
"!="                        { fprintf(output_file, "%s\t- NOT_EQUAL\n", yytext); operator_count++; }

"++"                        { fprintf(output_file, "%s\t- INCREMENT\n", yytext); operator_count++; }
"--"                        { fprintf(output_file, "%s\t- DECREMENT\n", yytext); operator_count++; }

"&&"                        { fprintf(output_file, "%s\t- LOGICAL_AND\n", yytext); operator_count++; }
"||"                        { fprintf(output_file, "%s\t- LOGICAL_OR\n", yytext); operator_count++; }
"&"                         { fprintf(output_file, "%s\t- BITWISE_AND\n", yytext); operator_count++; }
"|"                         { fprintf(output_file, "%s\t- BITWISE_OR\n", yytext); operator_count++; }
"^"                         { fprintf(output_file, "%s\t- BITWISE_XOR\n", yytext); operator_count++; }
"!"                         { fprintf(output_file, "%s\t- LOGICAL_NOT\n", yytext); operator_count++; }

[-+]?[0-9]+                             { fprintf(output_file, "%s\t- INTEGER_CONSTANT\n", yytext);
constant_count++; }
[-+]?[0-9]*\.[0-9]+                     { fprintf(output_file, "%s\t- FLOATING_CONSTANT\n", yytext);
constant_count++; }
\'[^\']\'                       { fprintf(output_file, "%s\t- CHARACTER_CONSTANT\n", yytext);
constant_count++; }
\"[^\"]*\"                              { fprintf(output_file, "%s\t- STRING_LITERAL\n", yytext);
constant_count++; }


"input"                     { fprintf(output_file, "%s\t- IDENTIFIER\n", yytext); identifier_count++; }
"output"                    { fprintf(output_file, "%s\t- IDENTIFIER\n", yytext); identifier_count++; }


"repeat"([^*]|[\r\n]|(\*+([^*>]|[\r\n])))*"end_repeat" {
   int statement = 0;

   fprintf(output_file, "%s\n- REPEATED_STATEMENT starting at line %d\n", yytext, lineno);

   char *line = strtok(yytext, "\n");
```

```
    while (line != NULL) {
        if (strcmp(line, "") != 0) {
            statement++;
        }

        line = strtok(NULL, "\n");
    }

    fprintf(output_file, "No. of statements: %d\n", statement - 2);
}

"repeat".* {
    printf("Error at Line %d: Unclosed Loop: %s\n", lineno, yytext);
    exit(1);
}

"end_repeat".* {
    printf("Error at Line %d: Unexpected Loop: %s\n", lineno, yytext);
    exit(1);
}

"select"([^*]|[\r\n]|(\*+([^*>]|[\r\n])))*"end_select" {
    int total_options = 0;
    int otherwise_count = 0;

    fprintf(output_file, "\n%s - CONDITIONAL_STATEMENT\n", yytext);

    char *line = strtok(yytext, "\n");
    while (line != NULL) {
        if (strcmp(line, "") != 0) {
            if (strstr(line, "option") != NULL) {
                total_options++;
            } else if (strstr(line, "otherwise") != NULL) {
                otherwise_count++;
            }
        }
        line = strtok(NULL, "\n");
    }

    if (otherwise_count > 1) {
        printf("Error at Line %d: Multiple 'otherwise' statements found within the block\n", lineno);
    }
    else {
        fprintf(output_file, "Total options within the block: %d\n", total_options + otherwise_count);
    }
}

"end_select".* {
    printf("Error at Line %d: Unexpected Condition: %s\n", lineno, yytext);
    exit(1);
}

.                                                   { printf("Error at Line %d: %s", lineno, yytext); exit(1);}

%%
```

```
int main()
{

   output_file = fopen("output.txt", "w");
   if (output_file == NULL) {
      perror("Error opening output file");
      return 1;
   }

   FILE *input_file = fopen("input.txt", "r");
   if (input_file == NULL) {
      perror("Error opening input file");
      return 1;
   }

   yyin = input_file;
   yylex();

   if (keyword_count)
      fprintf(output_file, "\nKeyword Count: %d\n", keyword_count);
   if (identifier_count + value_len)
      fprintf(output_file, "Identifier Count: %d\n", identifier_count + value_len);
   if (special_symbol_count)
      fprintf(output_file, "Special Symbol Count: %d\n", special_symbol_count);
   if (constant_count + value_len)
      fprintf(output_file, "Constant/Literal Count: %d\n", constant_count + value_len);
   if (operator_count)
      fprintf(output_file, "Operator Count: %d\n\n", operator_count);

   if (value_len > 0) {

      fprintf(output_file, "\n\n--SYMBOL TABLE--\n");

      fprintf(output_file, "+--------------------------------+--------------------------------+--------------------------------+\n");
      fprintf(output_file, "|         IDENTIFIER             |           DATATYPE             |           VALUE                |\n");
      fprintf(output_file, "+--------------------------------+--------------------------------+--------------------------------+\n");

      for (int i = 1; i <= value_len; i++) {

         fprintf(output_file, "| %-30s | %-30s | ", identifier[i], datatype[i]);

         if (choose[i] == 1)
            fprintf(output_file, "%-30d |\n", value[i].INT);

         else if (choose[i] == 2)
            fprintf(output_file, "%-30f |\n", value[i].FLOAT);

         else if (choose[i] == 3)
            fprintf(output_file, "%-30c |\n", value[i].CHAR);

         else if (choose[i] == 4)
            fprintf(output_file, "%-30s |\n", value[i].STRING);

         fprintf(output_file, "+--------------------------------+--------------------------------+--------------------------------+\n");
      }
   }
```

```
    fclose(input_file);
    fclose(output_file);

    return 0;
}
```

LEXER IMPLEMENTATION:

The primary focus of the provided code is the lexer implemented using Flex (Lex). The lexer reads the source code character by character and identifies and categorizes tokens, such as keywords, identifiers, constants, operators, and special symbols. It also handles error scenarios, including redefinitions, inappropriate assignments, and unclosed or unexpected comments or loops.

ID AND TYPE DESIGN:

In our custom programming language, we have the creative freedom to design identifiers (IDs) and data types according to our preferences and project requirements. This flexibility allows us to create a language that aligns with the specific needs of our application.

In the provided code, we have introduced two fundamental data types: 'var' and 'const.' 'Var' represents mutable variables that can be assigned new values, while 'const' represents constants with fixed values. Identifiers can be given any name, following standard rules for variable naming in programming languages.

Sample Input:

```
var v1=45, v2='c'
const c1=76.35, c2="khulna"
var v3=53.64, v4="kuet"
const c3=23, c4='r'
assign v2="dhaka", v4='g'
```

Sample Output:

```
v1 - IDENTIFIER
45 - INTEGER_CONSTANT
,          - COMMA
v2 - IDENTIFIER
'c' - CHARACTER_CONSTANT
c1 - IDENTIFIER
76.35 - FLOATING_CONSTANT
,          - COMMA
c2 - IDENTIFIER
"khulna" - STRING_LITERAL
v3 - IDENTIFIER
53.64 - FLOATING_CONSTANT
,          - COMMA
v4 - IDENTIFIER
"kuet" - STRING_LITERAL
c3 - IDENTIFIER
23 - INTEGER_CONSTANT
,          - COMMA
c4 - IDENTIFIER
'r' - CHARACTER_CONSTANT
"dhaka" - STRING_LITERAL
,          - COMMA
'g' - CHARACTER_CONSTANT
Identifier Count: 8
```

Special Symbol Count: 5
Constant/Literal Count: 8

--SYMBOL TABLE--

IDENTIFIER	DATATYPE	VALUE
v1	VAR	45
v2	VAR	"dhaka"
c1	CONST	76.349998
c2	CONST	"khulna"
v3	VAR	53.639999
v4	VAR	g
c3	CONST	23
c4	CONST	r

OPERATOR HANDLING:

Our custom language incorporates various operators, including arithmetic (+, -, *, /, %), comparison (==, !=, <, <=, >, >=), logical (&&, ||, !), bitwise (&, |, ^), and assignment (=). The lexer efficiently recognizes these operators and categorizes them as 'OPERATOR' tokens.

Input Sample:

```
4 % 9 * 1 / -3 * 7 ** 2
!(4 >= 5) ==  (2 < 6)
```

Output Sample:

```
4          - INTEGER_CONSTANT
%          - MODULO
9          - INTEGER_CONSTANT
*          - MULTIPLY
1          - INTEGER_CONSTANT
/          - DIVIDE
-3         - INTEGER_CONSTANT
*          - MULTIPLY
7          - INTEGER_CONSTANT
**         - POWER
2          - INTEGER_CONSTANT
!          - LOGICAL_NOT
(          - OPENING BRACKETS
4          - INTEGER_CONSTANT
>=         - GREATER_THAN_OR_EQUAL
5          - INTEGER_CONSTANT
)          - CLOSING BRACKETS
==         - EQUALS
(          - OPENING BRACKETS
2          - INTEGER_CONSTANT
```

```
<           - LESS_THAN
6           - INTEGER_CONSTANT
)           - CLOSING BRACKETS

Special Symbol Count: 4
Constant/Literal Count: 10
Operator Count: 9
```

LOOP AND DIRECTIVES:

To enable iterative and conditional control flow, we have designed loop and directive constructs within our language. In the provided code, we implement a 'repeat' to 'end_repeat' construct for loops and a 'select' to 'end_select' construct for conditionals. These constructs are crucial for controlling the flow of our custom language programs.

Input Sample:

```
#define pi 3.1416

var a=0
repeat a!=10
        out<<a
        a++
end_repeat

var b=10
select
        option b<10:
                output<<"Smaller than 10"
        option b>10:
                output<<"Greater than 10"
        otherwise:
                output<<"Smaller than 10"
end_select
```

Output Sample:

```
#define pi 3.1416 - MACRO
a - IDENTIFIER
0 - INTEGER_CONSTANT
repeat a!=10
        out<<a
        a++
end_repeat
- REPEATED_STATEMENT starting at line 4
No. of statements: 2
b - IDENTIFIER
10 - INTEGER_CONSTANT

select
        option b<10:
                output<<"Smaller than 10"
        option b>10:
                output<<"Greater than 10"
        otherwise:
                output<<"Smaller than 10"
end_select - CONDITIONAL_STATEMENT
Total options within the block: 3
```

Identifier Count: 2
Constant/Literal Count: 2

--SYMBOL TABLE--

IDENTIFIER	DATATYPE	VALUE
a	VAR	0
b	VAR	10

COMMENT DETECTION:

The lexer in the code effectively detects both single-line and multi-line comments. Single-line comments are denoted by '@' at the beginning of a line, while multi-line comments are enclosed within '<' and '>'.
In summary, this lab explores the development of a custom programming language with a focus on ID and type design, operator handling, loop and directive constructs, and effective tokenization using Flex (Lex). The design of this custom language's syntax is open to creativity, and it serves as a starting point for creating programming languages tailored to specific applications and needs.

Input Sample:

```
@var v1=45, v2='c'

<* const c1=76.35, c2="khulna"
var v3=53.64, v4="kuet"
const c3=23, c4='r' *>
```

Output Sample:

```
@var v1=45, v2='c'          - SINGLE_LINE_COMMENT
<* const c1=76.35, c2="khulna"
var v3=53.64, v4="kuet"
const c3=23, c4='r' *>      - MULTILINE_COMMENT
```

ERROR HANDLING:

Sample 1:

```
<*
const c1=76.35, c2="khulna"
var v3=53.64, v4="kuet"
const c3=23, c4='r'
```

*Error at Line 1: Unclosed Comment: <**

Sample 2:

```
repeat a!=10
        out<<a
        a++
end_repeat
*>
```

*Error at Line 2: Unexpected Closing Comment: *>*

Sample 3:
```
var array[3] = [["khulna", "dhaka", "sylhet"]
```

Error *at Line 1*: [

Sample 4:
```
var array[3] = ‘k’, ‘d’, ‘s’]]
```

Error at Line 1:]

Sample 5:
```
var array[3][3]
```

Error at Line 1: Multi-dimensional arrays are not allowed: array[3][3]

Sample 6:
```
var b=10
end_select
```

Error at Line 2: Unexpected Condition: end_select

Sample 7:
```
select
        option b<10:
                output<<"Smaller than 10"
        otherwise:
                output<<"Smaller than 10"
        otherwise:
                output<<"Smaller than 5"
end_select
```
Error at Line 1: Multiple 'otherwise' statements found within the block

Sample 8:
```
var a=0
repeat a!=10
        out<<a
        a++
```

Error at Line 2: Unclosed Loop: repeat a!=10

Sample 9:
```
var a=0
end_repeat
```

Error at Line 2: Unexpected Loop: end_repeat

Sample 10:
```
var v1=45, v2='c'
const c1=76.35, c2="khulna"
var v3=53.64, v4="kuet"
const c3=23, c4='r'
var v2="dhaka"
```

Error at Line 5: 'v2' has been defined before.

Sample 11:

```
var v1=45, v2='c'
const c1=76.35, c2="khulna"
var v3=53.64, v4="kuet"
const c3=23, c4='r'
assign c2="dhaka"
```

Error at Line 5: 'c2' can't be assigned.

Sample 12:

```
var v1=45, v2='c'
const c1=76.35, c2="khulna"
var v3=53.64, v4="kuet"
const c3=23, c4='r'
assign v2="dhaka", v5=56
```

Error at Line 5: v5 is not assigned before.

DISCUSSION:

In this lab, our primary goal was to design and implement a lexer using Flex for a custom programming language. The lexer was responsible for breaking down the source code into distinct tokens, categorizing them into keywords, identifiers, constants, operators, and symbols.

The lexer efficiently handled single-line and multi-line comments, ensuring that comments did not interfere with the tokenization process.

An important aspect of our lexer was its error-handling capability. It accurately detected and reported errors, such as redefining variables or constants, which is crucial for code reliability. The lexer prevented constants from being reassigned and identified issues like multi-dimensional arrays and undefined variables.

To manage variables and constants effectively, we implemented a symbol table. This table stored information about identifiers, including data types and values where applicable, simplifying variable and constant tracking.

The lexer generated an "output.txt" file as its output, documenting the identified tokens and their categories. It also provided counts of keywords, identifiers, constants, and operators, offering insights into the source code's composition.

Overall, we successfully achieved the lab's objectives by designing a custom programming language and implementing a lexer using Flex. The symbol table enhanced our understanding of variables and constants in the code.

Key takeaways from this lab include practical experience in lexer development, understanding the importance of error handling, and appreciating the value of a symbol table for managing variables and constants. These skills will be valuable for future compiler and interpreter projects.

CONCLUSION:

This lab was an accessible and informative introduction to lexer development using Flex. We gained valuable insights into syntax design, tokenization, and error handling. By creating an efficient symbol table, we simplified variable and constant management, aiding in code analysis. The "output.txt" file provided a straightforward summary of identified tokens and their categories. This experience has laid a solid foundation for understanding lexical analysis, making it easier to approach similar language-related projects and programming tasks in the future.