

**Project Report**

**Course Code : CSE332**

**Course Title : Compiler Design Lab**

**Submitted To:**

**Name : Tapasy Rabeya**

**Designation : Senior Lecturer**

**Department : CSE**

**University : Daffodil International University**

**Submitted By:**

**Name : Md Iftekharul Alam Name : Maloy Kishor Paul Name : Tanvir Ahmed**

**Id : 222-15-6095 Id : 222-15-6095 Id : 222-15-6378**

**Section : 62\_C**

**Department : CSE**

**Semester : Summer 2025**

**University : Daffodil International University**

**Date of Submission : 17/08/2025**

**Table of Contents**

1. Introduction …................................................................................................................................... 3
2. What is a Compiler? ....................................................................................................................................... 3
3. Phases of a Compiler ....................................................................................................................................... 4  
   3.1 Lexical Analysis ....................................................................................................................................... 4  
   3.2 Syntax Analysis ....................................................................................................................................... 4  
   3.3 Semantic Analysis ........................................................................................................................................5  
   3.4 Intermediate Code Generation ....................................................................................................................................... 5  
   3.5 Code Optimization ....................................................................................................................................... 6  
   3.6 Code Generation ....................................................................................................................................... 6
4. Lexical Analysis in Detail ....................................................................................................................................... 6
5. Regular Expressions in Relation to Lexical Analysis ....................................................................................................................................... 7
6. Explanation of Custom Regular Expression for Variable Declaration ....................................................................................................................................... 8
7. Finite Automata in Relation to Lexical Analysis ....................................................................................................................................... 8
8. Code Implementation ....................................................................................................................................... 9
9. File Handling in C: Working and Advantages ..................................................................................................................................... 21
10. Sample Input (input.txt) ..................................................................................................................................... 22
11. Sample Output (output.txt and Terminal) and Explanation ..................................................................................................................................... 26
12. Conclusion ..................................................................................................................................... 28
13. References ..................................................................................................................................... 28

**1. Introduction**

Compiler design is a fundamental aspect of computer science that involves translating high-level programming languages into machine-readable code. This lab report focuses on the implementation of a lexical analyzer, which is the first phase of a compiler. The project involves creating a lexical analyzer in C that reads a large C code file (input.txt), tokenizes it, validates variable declarations using a custom regular expression, generates a symbol table, and outputs results to output.txt. Additionally, it includes an interactive validator for checking variable validity.

The objective is to demonstrate understanding of compiler phases, with emphasis on lexical analysis, regular expressions, and finite automata. The system is tested on a Tourist Management System code, incorporating various tokens, comments, and both valid and invalid identifiers.

This report elaborates on the theoretical concepts and practical implementation, ensuring comprehensive coverage for evaluation.

**2. What is a Compiler?**

A compiler is a specialized software program that translates source code written in a high-level programming language (such as C, Java, or Python) into low-level machine code or an intermediate form that can be executed by a computer's hardware. Unlike interpreters, which execute code line by line, compilers process the entire program at once, detecting errors early and producing an executable file.

**Key Functions of a Compiler:**

* **Error Detection:** Identifies syntax, semantic, and logical errors during compilation.
* **Optimization:** Improves code efficiency by reducing execution time and memory usage.
* **Portability:** Generates machine-specific code, allowing programs to run on different architectures.
* **Translation:** Converts human-readable code into binary instructions understandable by the CPU.

Compilers are essential in software development as they bridge the gap between programmers and hardware. For example, the GCC (GNU Compiler Collection) is a popular compiler for C and C++ languages.

**Advantages:**

* Faster execution of compiled programs.
* Better error handling compared to interpreters.
* Enables code reuse and modular programming.

**Disadvantages:**

* Compilation time can be lengthy for large programs.
* Debugging compiled code is more challenging.

In this project, the lexical analyzer acts as the entry point of the compiler, processing the source code to produce tokens for further phases.

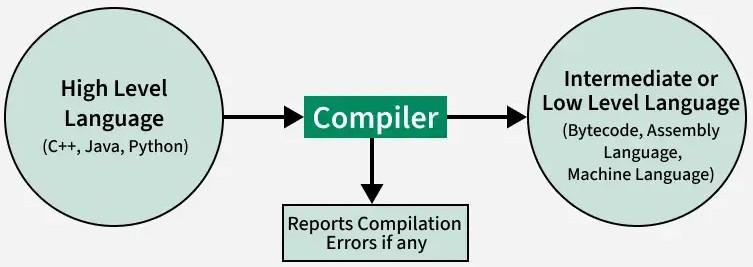


Figure 1. Compiler

**3. Phases of a Compiler**

The compilation process is divided into several phases, each transforming the source code step by step. These phases are grouped into two parts: the front-end (analysis) and back-end (synthesis). The working procedure involves sequential execution, where the output of one phase serves as input to the next. If errors occur in any phase, compilation halts. Below, each phase is explained concisely with an integrated example based on the C expression: **position = initial + rate \* 60** .This arithmetic expression involves assignment, addition, and multiplication, providing a medium-complexity example to illustrate the phases effectively.

**3.1. Lexical Analysis**

Lexical analysis reads the source code as a character stream and groups it into tokens (keywords, identifiers, operators, literals, symbols) using regular expressions and finite automata. It skips whitespace and comments, populates the symbol table with identifiers, and detects errors like invalid characters. The output is a token stream for the next phase.

**Example**: For **position = initial + rate \* 60** the lexer produces tokens: <identifier, position>, <operator, =>, <identifier, initial>, <operator, +>, <identifier, rate>, <operator, \*>, <literal, 60>, <symbol, ;>. The lexer stores "position", "initial", and "rate" in the symbol table, verifies no invalid characters exist, and passes the token stream to the syntax analyzer.

**3.2. Syntax Analysis**

Syntax analysis checks the token stream against the language’s grammar rules, constructing a parse tree or abstract syntax tree (AST). It ensures proper structure (valid expressions, correct punctuation) using top-down or bottom-up parsing, reporting syntax errors like missing semicolons.

**Example**: For the expression, the parser builds a parse tree with a root "AssignmentStmt (=)" node: left child "Identifier: position", right child "Expression: initial + (rate \* 60)". The subexpression "rate \* 60" is a "MultiplicativeExpr" node with higher precedence, ensuring the multiplication is grouped correctly, and the semicolon terminates the statement. Errors like a missing semicolon would be flagged here.

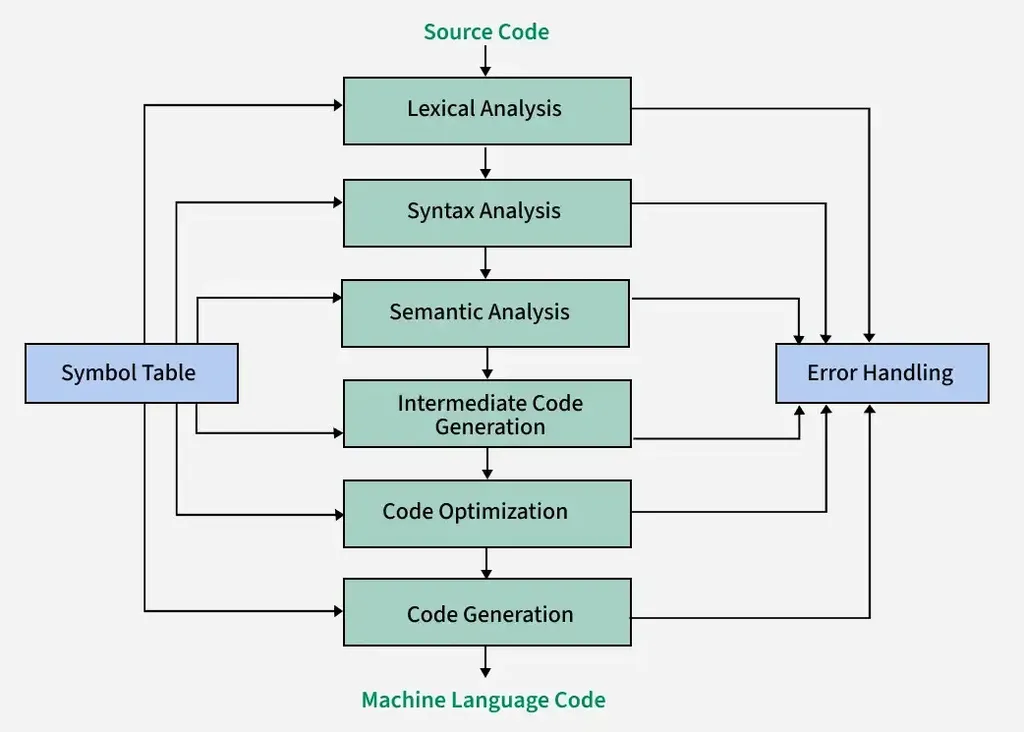


Figure 2. Phases of a compiler

**3.3. Semantic Analysis**

Semantic analysis verifies the program’s logical correctness, checking type consistency, variable declarations, and scope. It uses the symbol table to ensure variables are declared and operations are valid, reporting errors like type mismatches or undeclared variables.

**Example**: The semantic analyzer confirms "position", "initial", and "rate" are declared (e.g., as integers or floats in the symbol table). It verifies that rate \* 60 yields a numeric result (integer or float), initial + (result) is compatible, and the assignment to "position" matches its type. Errors like using an undeclared "initial" or assigning a string to "position" would be reported.

**3.4. Intermediate Code Generation**

This phase translates the parse tree into a machine-independent intermediate representation (IR), such as three-address code (TAC), using temporary variables for simple operations. It simplifies complex expressions and control structures for easier optimization.

**Example**: For position = initial + rate \* 60;, the TAC is:

t1 = rate \* 60

t2 = initial + t1

position = t2

The multiplication is computed first (respecting precedence), stored in t1, added to "initial" in t2, and assigned to "position". The symbol table provides type information to ensure numeric operations.

**3.5. Code Optimization**

Code optimization enhances the IR’s efficiency by reducing execution time or memory usage through techniques like constant folding, common subexpression elimination, or redundant code removal, preserving program semantics.

**Example**: The TAC is optimized:

t1 = rate \* 60

position = initial + t1

Constant folding could compute rate \* 60 if "rate" were a constant, but since it’s a variable, the optimizer removes the redundant temporary t2, combining the addition and assignment. The symbol table confirms variable types for safe optimization.

**3.6. Code Generation**

Code generation produces target machine code or assembly, handling register allocation, instruction selection, and memory management. It uses the symbol table for variable addresses, generating platform-specific instructions.

**Example**: The optimized TAC becomes assembly (using generic registers R1, R2, R3):

text

mov R1, [rate] ; load rate

mov R2, 60 ; load constant 60

mul R1, R2 ; R1 = rate \* 60

mov R2, [initial] ; load initial

add R1, R2 ; R1 = initial + (rate \* 60)

mov R3, [position]

mov [R3], R1 ; position = R1

Registers R1 and R2 compute the expression, R3 handles the address of "position", and the symbol table provides memory offsets, ensuring efficient code.

**4. Lexical Analysis in Detail**

Lexical analysis, also known as scanning or tokenization, is the first phase of the compiler. It reads the source code character by character from left to right and groups them into meaningful sequences called tokens. Tokens include keywords (e.g., int, if), identifiers (variables), constants (numbers), operators (+, =), and symbols (; ,).

**Role in Compiler:**

* Removes whitespace and comments.
* Identifies errors like invalid characters.
* Outputs a stream of tokens for syntax analysis.

**How it Works:**

The lexical analyzer uses patterns defined by regular expressions to match sequences. It employs finite automata to recognize these patterns efficiently. For example, in the code int sum = a + b;, tokens are: int (keyword), sum (identifier), = (operator), a (identifier), + (operator), b (identifier), ; (symbol).

**Advantages:**

* Simplifies subsequent phases by providing structured input.
* Improves compilation speed by early error detection.

**Challenges:**

* Handling ambiguities (e.g., distinguishing keywords from identifiers).
* Managing large source files efficiently.

In this project, the lexical analyzer tokenizes the input.txt file, categorizes tokens (keywords, identifiers, etc.), skips comments, and validates identifiers using a custom regex.

**5. Regular Expressions in Relation to Lexical Analysis**

Regular expressions (regex) are formal notations for describing patterns in strings, crucial for lexical analysis. In compilers, regex defines the structure of tokens, allowing the lexer to match and extract them from source code.

**Explanation of Regular Expressions:**

A regex is a sequence of characters defining a search pattern. Basic elements:

* **Literals:** Match exact characters (a matches 'a').
* **Metacharacters:** . (any char), \* (zero or more), + (one or more), ? (zero or one), | (or), [] (character class), ^ (start), $ (end).
* **Examples:** [a-z]+ matches lowercase words; \d{2}-\d{2}-\d{4} matches dates like 16-08-2025.

In lexical analysis, regex specifies token patterns:

* **Identifiers:** [a-zA-Z\_][a-zA-Z0-9\_]\*
* **Numbers:** \d+
* **Keywords:** Exact matches like int|float.

The lexer converts regex to finite automata for efficient scanning.

**Relation to Lexical Analysis:**

Regex enables pattern-based token recognition, making the lexer flexible for different languages. Tools like LEX use regex to generate scanners automatically.

**Advantages in Compilers:**

* Concise token definition.
* Easy to modify for language changes.

**6. Explanation of Custom Regular Expression for Variable Declaration**

In this project, variables must follow a custom regex for validation during lexical analysis:

* **Pattern:** ^([#@!]?[a-z]{4,7}(?<!([a-z])\2{2})[0-9]{2,4}(?<!([0-9])\2{2})@r)$
* **Breakdown:**
  + **Optional leading:** [#@!]? – May start with #, @, or ! (optional).
  + **Letters:** [a-z]{4,7} – 4 to 7 lowercase letters.
  + **No more than two consecutive same letters:** Using negative lookbehind (?<!([a-z])\2{2}).
  + **Digits:** [0-9]{2,4} – 2 to 4 digits (no negatives, as digits are 0-9).
  + **No more than two consecutive same digits:** (?<!([0-9])\2{2}).
  + **Ends with:** @r – Must end exactly with "@r".

**Why This Regex?**

It enforces strict naming conventions for variables in the Tourist Management System, ensuring uniqueness and readability. In the code, it's implemented in isValidIdentifier\_Advanced() function, which iterates through the string to match the pattern.

**Example Valid:** @abcd12@r (prefix @, letters abcd – no three same, digits 12 – no three same, ends @r). Invalid: @aaaa111@r (aaaa has four 'a's, 111 has three '1's).

This regex integrates with lexical analysis to filter valid identifiers for the symbol table.

**7. Finite Automata in Relation to Lexical Analysis**

Finite Automata (FA) are abstract machines that recognize regular languages, directly linked to regex in lexical analysis.

**Explanation of Finite Automata:**

FA consists of:

* **States (Q):** Finite set, including start and accept states.
* **Alphabet (Σ):** Input symbols.
* **Transition Function (δ):** Maps state-symbol to next state.
* **Start State (q0).**
* **Accept States (F).**

Types:

* **Deterministic FA (DFA):** One transition per symbol.
* **Non-deterministic FA (NFA):** Multiple transitions, epsilon moves.

In compilers, regex is converted to NFA, then DFA for efficient token recognition.

**Relation to Lexical Analysis:**

The lexer uses DFA to scan input: Start at q0, transition on characters; accept state indicates token. For example, regex for identifiers [a-zA-Z\_][a-zA-Z0-9\_]\* becomes a DFA with states for letter/digit sequences.

Advantages:

* Fast O(n) scanning.
* Handles complex patterns.

In our project, the custom regex could be modeled as a DFA for validation, though implemented procedurally.

**8. Code Implementation**

The lexical analyzer is written in C, reading input.txt, tokenizing, validating identifiers, building symbol table, and writing to output.txt.

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <stdbool.h>

#define MAX\_LINE 1000

#define MAX\_TOKENS\_PER\_LINE 200

// Symbol Table Struct

typedef struct {

char name[100]; // Identifier name

char type[100]; // Data type (int, float, etc.)

char value[100]; // Value or "-" if uninitialized

int line; // Line number declared

} Symbol;

Symbol symbolTable[1000];

int symbolCount = 0;

//Valid/Invalid Identifiers

char validIdentifiers[1000][100];

int validIdentifiersCount = 0;

char invalidIdentifiers[1000][100];

int invalidIdentifiersCount = 0;

//Other Tokens

char othersFound[1000][100];

int othersCount = 0;

// Keyword List

const char \*keywords[] = {

"int", "float", "char", "double", "return", "if", "else", "for", "while",

"void", "do", "switch", "case", "default", "break", "continue", "struct",

"typedef", "include", "define", "unsigned", "const", "static", "long", "short", "signed"

};

int keywordCount = sizeof(keywords) / sizeof(keywords[0]);

//Multi-character Operators

const char \*multiCharOps[] = {

"++", "--", "==", "!=", "<=", ">=", "&&", "||", "+=", "-=", "\*=", "/="

};

// Utility Functions

int isKeyword(const char \*word) {

for (int i = 0; i < keywordCount; i++) {

if (strcmp(word, keywords[i]) == 0)

return 1;

}

return 0;

}

int isMultiCharOp(const char \*word) {

for (int i = 0; i < sizeof(multiCharOps) / sizeof(multiCharOps[0]); i++) {

if (strcmp(word, multiCharOps[i]) == 0)

return 1;

}

return 0;

}

int isOperatorChar(char ch) {

return strchr("+-\*/%=<>!&|^~", ch) != NULL;

}

int isOperatorString(const char \*str) {

if (strlen(str) == 1 && isOperatorChar(str[0])) return 1;

if (isMultiCharOp(str)) return 1;

return 0;

}

int isBracket(char ch) {

return strchr("(){}[]", ch) != NULL;

}

int isSeparator(char ch) {

return strchr(",;:", ch) != NULL;

}

int isSpecialSymbol(char ch) {

return strchr("#.", ch) != NULL;

}

// Checks if token is a valid data type token (for declaration parsing)

int isDataTypeToken(const char \*token) {

const char \*dataTypes[] = {

"int", "float", "char", "double", "void",

"unsigned", "const", "static", "long", "short", "signed"

};

int count = sizeof(dataTypes) / sizeof(dataTypes[0]);

for (int i = 0; i < count; i++) {

if (strcmp(token, dataTypes[i]) == 0)

return 1;

}

return 0;

}

//Advanced Custom Regex Matcher

// Pattern:

// - Optional leading '#', '@', or '!'

// - 4 to 7 lowercase letters (a-z), no more than two consecutive same letters

// - 2 to 4 digits (0-9), no negative, no more than two consecutive same digits

// - Ends with "@r"

bool isValidIdentifier\_Advanced(const char \*str) {

int i = 0, len = strlen(str);

if (len < 8 || len > 14) return false; // Min: 4 letters + 2 digits + @r = 8, Max: 1 + 7 letters + 4 digits + @r = 14

// Optional leading '#', '@', or '!'

if (str[i] == '#' || str[i] == '@' || str[i] == '!') i++;

// 4 to 7 lowercase letters (a-z), no more than two consecutive same letters

int letterCount = 0;

char prev\_letter = '\0';

int consec\_letter = 0;

while (i < len && islower(str[i])) {

letterCount++;

if (letterCount == 1) {

prev\_letter = str[i];

consec\_letter = 1;

} else {

if (str[i] == prev\_letter) {

consec\_letter++;

if (consec\_letter > 2) return false;

} else {

consec\_letter = 1;

prev\_letter = str[i];

}

}

i++;

}

if (letterCount < 4 || letterCount > 7) return false;

// 2 to 4 digits (0-9), no more than two consecutive same digits

int digitCount = 0;

char prev\_digit = '\0';

int consec\_digit = 0;

while (i < len && isdigit(str[i])) {

digitCount++;

if (digitCount == 1) {

prev\_digit = str[i];

consec\_digit = 1;

} else {

if (str[i] == prev\_digit) {

consec\_digit++;

if (consec\_digit > 2) return false;

} else {

consec\_digit = 1;

prev\_digit = str[i];

}

}

i++;

}

if (digitCount < 2 || digitCount > 4) return false;

// Must end with "@r"

if (i + 1 >= len || strncmp(&str[i], "@r", 2) != 0) return false;

// Ensure we consumed the entire string

return i + 2 == len;

}

// Symbol Table Functions

int alreadyInSymbolTable(const char \*name) {

for (int i = 0; i < symbolCount; i++) {

if (strcmp(symbolTable[i].name, name) == 0)

return 1;

}

return 0;

}

void addToSymbolTable(const char \*type, const char \*name, const char \*value, int line) {

if (!alreadyInSymbolTable(name)) {

strncpy(symbolTable[symbolCount].type, type, sizeof(symbolTable[symbolCount].type)-1);

strncpy(symbolTable[symbolCount].name, name, sizeof(symbolTable[symbolCount].name)-1);

strncpy(symbolTable[symbolCount].value, value, sizeof(symbolTable[symbolCount].value)-1);

symbolTable[symbolCount].line = line;

symbolCount++;

}

}

// Tokenization Helper

int tokenizeLine(char \*line, char tokens[][100]) {

int tokenIndex = 0;

int i = 0;

int len = strlen(line);

while (i < len) {

// Skip whitespace

if (isspace(line[i])) {

i++;

continue;

}

// Handle multi-char operators (check first 2 chars)

if (i + 1 < len) {

char twoChar[3] = {line[i], line[i+1], '\0'};

if (isMultiCharOp(twoChar)) {

strcpy(tokens[tokenIndex++], twoChar);

i += 2;

continue;

}

}

// Single char operators, separators, brackets

if (isOperatorChar(line[i]) || isSeparator(line[i]) || isBracket(line[i]) || isSpecialSymbol(line[i])) {

char oneChar[2] = {line[i], '\0'};

strcpy(tokens[tokenIndex++], oneChar);

i++;

continue;

}

// Otherwise read a word/identifier/number literal

int start = i;

if (isalpha(line[i]) || line[i] == '\_' || line[i] == '#' || line[i] == '@' || line[i] == '!') {

// Identifier or keyword

while (i < len && (isalnum(line[i]) || line[i] == '\_' || line[i] == '@' || line[i] == '!')) i++;

} else if (isdigit(line[i])) {

// Number literal (integer or float)

while (i < len && (isdigit(line[i]) || line[i] == '.')) i++;

} else if (line[i] == '\"') {

// String literal - read until closing quote

i++; // skip opening quote

while (i < len && line[i] != '\"') i++;

if (i < len) i++; // skip closing quote

} else if (line[i] == '\'') {

// Character literal

i++; // skip opening quote

if (i < len && line[i] != '\'') i++; // skip char

if (i < len && line[i] == '\'') i++; // skip closing quote

} else {

// Unknown single char token

char oneChar[2] = {line[i], '\0'};

strcpy(tokens[tokenIndex++], oneChar);

i++;

continue;

}

int length = i - start;

if (length > 0) {

strncpy(tokens[tokenIndex], &line[start], length);

tokens[tokenIndex][length] = '\0';

tokenIndex++;

}

}

return tokenIndex;

}

// ==================== Processing Declarations ====================

void processDeclarationTokens(char tokens[][100], int startIndex, int tokenCount, const char \*fullType, int lineno) {

int i = startIndex;

while (i < tokenCount) {

// Skip commas

if (strcmp(tokens[i], ",") == 0) {

i++;

continue;

}

// End if semicolon

if (strcmp(tokens[i], ";") == 0) break;

char varName[100] = "";

char varValue[100] = "-";

// The next token should be a potential identifier (variable name)

if (isValidIdentifier\_Advanced(tokens[i])) {

strcpy(varName, tokens[i]);

// Add to valid identifiers

int found = 0;

for (int k = 0; k < validIdentifiersCount; k++) {

if (strcmp(validIdentifiers[k], varName) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(validIdentifiers[validIdentifiersCount++], varName);

}

i++;

// Check if initialization: =

if (i < tokenCount && strcmp(tokens[i], "=") == 0) {

i++;

if (i < tokenCount) {

strcpy(varValue, tokens[i]);

i++;

}

}

addToSymbolTable(fullType, varName, varValue, lineno);

} else {

// Only add to invalid identifiers if it could be a variable name

if (!isKeyword(tokens[i]) && !isOperatorString(tokens[i]) && !isBracket(tokens[i][0]) &&

!isSeparator(tokens[i][0]) && !isSpecialSymbol(tokens[i][0]) && !isdigit(tokens[i][0]) &&

tokens[i][0] != '"' && tokens[i][0] != '\'') {

int found = 0;

for (int k = 0; k < invalidIdentifiersCount; k++) {

if (strcmp(invalidIdentifiers[k], tokens[i]) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(invalidIdentifiers[invalidIdentifiersCount++], tokens[i]);

}

}

i++;

}

}

}

// ==================== Main Lexical Analyzer ====================

void processFile(FILE \*input, FILE \*output) {

char line[MAX\_LINE];

int lineno = 0;

// To accumulate all tokens for token categories output

char keywordsFound[1000][100];

int keywordsCount = 0;

char numericsFound[1000][100];

int numericsCount = 0;

char stringLiteralsFound[1000][1000];

int stringLiteralsCount = 0;

char multiCharOpsFound[1000][10];

int multiCharOpsCount = 0;

char operatorsFound[1000][10];

int operatorsCount = 0;

char separatorsFound[1000][10];

int separatorsCount = 0;

char bracketsFound[1000][10];

int bracketsCount = 0;

char specialSymbolsFound[1000][10];

int specialSymbolsCount = 0;

// Multi-line comment handling

int insideComment = 0;

while (fgets(line, sizeof(line), input)) {

lineno++;

// Remove newline at end

line[strcspn(line, "\n")] = 0;

// Handle multi-line comments /\* ... \*/

if (insideComment) {

char \*endComment = strstr(line, "\*/");

if (endComment) {

insideComment = 0;

memmove(line, endComment + 2, strlen(endComment + 2) + 1);

} else {

continue;

}

}

char \*startComment = strstr(line, "/\*");

if (startComment) {

insideComment = 1;

\*startComment = '\0';

}

// Remove single-line comments (//)

char \*comment = strstr(line, "//");

if (comment) \*comment = '\0';

// Tokenize line

char tokens[MAX\_TOKENS\_PER\_LINE][100];

int tokenCount = tokenizeLine(line, tokens);

if (tokenCount == 0) continue;

// Check if this line starts with data type tokens for declaration

int dataTypeTokensLen = 0;

char dataTypeBuffer[100] = "";

int i;

bool isFunctionDecl = false;

for (i = 0; i < tokenCount; i++) {

if (isDataTypeToken(tokens[i])) {

if (dataTypeTokensLen > 0) strcat(dataTypeBuffer, " ");

strcat(dataTypeBuffer, tokens[i]);

dataTypeTokensLen++;

} else if (dataTypeTokensLen > 0 && i + 1 < tokenCount && strcmp(tokens[i+1], "(") == 0) {

// Function declaration detected

isFunctionDecl = true;

break;

} else {

break;

}

}

// Process declarations (variables or functions)

if (dataTypeTokensLen > 0 && dataTypeTokensLen < tokenCount) {

if (isFunctionDecl) {

// Handle function declaration

if (isValidIdentifier\_Advanced(tokens[i])) {

int found = 0;

for (int k = 0; k < validIdentifiersCount; k++) {

if (strcmp(validIdentifiers[k], tokens[i]) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(validIdentifiers[validIdentifiersCount++], tokens[i]);

}

addToSymbolTable(dataTypeBuffer, tokens[i], "-", lineno);

} else {

int found = 0;

for (int k = 0; k < invalidIdentifiersCount; k++) {

if (strcmp(invalidIdentifiers[k], tokens[i]) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(invalidIdentifiers[invalidIdentifiersCount++], tokens[i]);

}

}

} else {

// Handle variable declarations

processDeclarationTokens(tokens, dataTypeTokensLen, tokenCount, dataTypeBuffer, lineno);

}

}

// Process all tokens for token categories

for (int t = 0; t < tokenCount; t++) {

char \*token = tokens[t];

// Skip empty tokens

if (strlen(token) == 0) continue;

// Keywords

if (isKeyword(token)) {

int found = 0;

for (int k = 0; k < keywordsCount; k++) {

if (strcmp(keywordsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found && strlen(token) < 100) {

strcpy(keywordsFound[keywordsCount++], token);

}

continue;

}

// Multi-char operators

if (isMultiCharOp(token)) {

int found = 0;

for (int k = 0; k < multiCharOpsCount; k++) {

if (strcmp(multiCharOpsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(multiCharOpsFound[multiCharOpsCount++], token);

}

continue;

}

// Single char operators

if (isOperatorString(token)) {

int found = 0;

for (int k = 0; k < operatorsCount; k++) {

if (strcmp(operatorsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(operatorsFound[operatorsCount++], token);

}

continue;

}

// Separators

if (strlen(token) == 1 && isSeparator(token[0])) {

int found = 0;

for (int k = 0; k < separatorsCount; k++) {

if (strcmp(separatorsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(separatorsFound[separatorsCount++], token);

}

continue;

}

// Brackets

if (strlen(token) == 1 && isBracket(token[0])) {

int found = 0;

for (int k = 0; k < bracketsCount; k++) {

if (strcmp(bracketsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(bracketsFound[bracketsCount++], token);

}

continue;

}

// Special symbols

if (strlen(token) == 1 && isSpecialSymbol(token[0])) {

int found = 0;

for (int k = 0; k < specialSymbolsCount; k++) {

if (strcmp(specialSymbolsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(specialSymbolsFound[specialSymbolsCount++], token);

}

continue;

}

// String literals

if (token[0] == '"' && token[strlen(token)-1] == '"') {

int found = 0;

for (int k = 0; k < stringLiteralsCount; k++) {

if (strcmp(stringLiteralsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(stringLiteralsFound[stringLiteralsCount++], token);

}

continue;

}

// Character literals

if (token[0] == '\'' && token[strlen(token)-1] == '\'') {

int found = 0;

for (int k = 0; k < stringLiteralsCount; k++) {

if (strcmp(stringLiteralsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(stringLiteralsFound[stringLiteralsCount++], token);

}

continue;

}

// Numeric literals

if (isdigit(token[0])) {

int found = 0;

for (int k = 0; k < numericsCount; k++) {

if (strcmp(numericsFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(numericsFound[numericsCount++], token);

}

continue;

}

// Identifiers are only checked in declaration contexts

// Tokens not in any category

int found = 0;

for (int k = 0; k < othersCount; k++) {

if (strcmp(othersFound[k], token) == 0) {

found = 1;

break;

}

}

if (!found) {

strcpy(othersFound[othersCount++], token);

}

}

}

// Print to output.txt

fprintf(output, "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

fprintf(output, "\* LEXICAL ANALYSIS REPORT \*\n");

fprintf(output, "\* Tourist Management System Code \*\n");

fprintf(output, "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

// Print Valid and Invalid Identifiers

fprintf(output, "Valid Variables/Identifiers (Count: %d): [", validIdentifiersCount);

for (int i = 0; i < validIdentifiersCount; i++) {

fprintf(output, "%s%s", validIdentifiers[i], (i == validIdentifiersCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

fprintf(output, "Invalid Variables/Identifiers (Count: %d): [", invalidIdentifiersCount);

for (int i = 0; i < invalidIdentifiersCount; i++) {

fprintf(output, "%s%s", invalidIdentifiers[i], (i == invalidIdentifiersCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Print all tokens by category

fprintf(output, "=========== TOKENS BY CATEGORY ===========\n\n");

// Print Keywords

fprintf(output, "Keywords: [");

for (int i = 0; i < keywordsCount; i++) {

fprintf(output, "%s%s", keywordsFound[i], (i == keywordsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Print Identifiers (valid ones)

fprintf(output, "Identifiers: [");

for (int i = 0; i < validIdentifiersCount; i++) {

fprintf(output, "%s%s", validIdentifiers[i], (i == validIdentifiersCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Print Numeric literals

fprintf(output, "Numeric: [");

for (int i = 0; i < numericsCount; i++) {

fprintf(output, "%s%s", numericsFound[i], (i == numericsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// String literals

fprintf(output, "String Literals: [");

for (int i = 0; i < stringLiteralsCount; i++) {

fprintf(output, "%s%s", stringLiteralsFound[i], (i == stringLiteralsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Multi-char operators

fprintf(output, "Multi-char Operators: [");

for (int i = 0; i < multiCharOpsCount; i++) {

fprintf(output, "%s%s", multiCharOpsFound[i], (i == multiCharOpsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Single char operators

fprintf(output, "Operators: [");

for (int i = 0; i < operatorsCount; i++) {

fprintf(output, "%s%s", operatorsFound[i], (i == operatorsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Separators

fprintf(output, "Separators: [");

for (int i = 0; i < separatorsCount; i++) {

fprintf(output, "%s%s", separatorsFound[i], (i == separatorsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Brackets

fprintf(output, "Brackets: [");

for (int i = 0; i < bracketsCount; i++) {

fprintf(output, "%s%s", bracketsFound[i], (i == bracketsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Special Symbols

fprintf(output, "Special Symbols: [");

for (int i = 0; i < specialSymbolsCount; i++) {

fprintf(output, "%s%s", specialSymbolsFound[i], (i == specialSymbolsCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Other tokens

fprintf(output, "Others: [");

for (int i = 0; i < othersCount; i++) {

fprintf(output, "%s%s", othersFound[i], (i == othersCount -1) ? "" : ", ");

}

fprintf(output, "]\n\n");

// Print Symbol Table

fprintf(output, "=========== SYMBOL TABLE ===========\n");

fprintf(output, "---------------------------------------------------------------\n");

fprintf(output, "| Name | DataType | Value | Line |\n");

fprintf(output, "---------------------------------------------------------------\n");

for (int i = 0; i < symbolCount; i++) {

fprintf(output, "| %-15s | %-21s | %-14s | %-4d |\n",

symbolTable[i].name, symbolTable[i].type, symbolTable[i].value, symbolTable[i].line);

}

fprintf(output, "---------------------------------------------------------------\n");

fprintf(output, "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

fprintf(output, "\* END OF REPORT \*\n");

fprintf(output, "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

// Interactive validator for terminal input

void interactiveValidator() {

char input[100];

printf("\n========================================\n");

printf("Variable Declaration Validity Check \n");

printf("========================================\n");

while (1) {

printf("\nDo you want to check a variable name? (Y/N): ");

if (!fgets(input, sizeof(input), stdin)) break;

if (input[0] == 'N' || input[0] == 'n') {

printf("Exiting validation mode.\n");

break;

} else if (input[0] == 'Y' || input[0] == 'y') {

while (1) {

printf("\nEnter variable/identifier name to validate (or N to exit): ");

if (!fgets(input, sizeof(input), stdin)) return;

// Remove newline

input[strcspn(input, "\n")] = 0;

if (input[0] == 'N' || input[0] == 'n') {

printf("Exiting validation mode.\n");

return;

}

printf("\nChecking variable: \"%s\"\n", input);

// Validate using your advanced pattern:

if (isValidIdentifier\_Advanced(input)) {

printf("Valid identifier!\n");

printf("Reason: \n");

int i = 0;

if (input[i] == '#' || input[i] == '@' || input[i] == '!') {

printf(" - Optional leading character (#, @, !): Present (%c)\n", input[i]);

i++;

} else {

printf(" - Optional leading character (#, @, !): Not present\n");

}

int letterCount = 0;

char prev\_letter = '\0';

int consec\_letter = 0;

bool letter\_consec\_ok = true;

while (i < (int)strlen(input) && islower(input[i])) {

letterCount++;

if (letterCount == 1) {

prev\_letter = input[i];

consec\_letter = 1;

} else {

if (input[i] == prev\_letter) {

consec\_letter++;

if (consec\_letter > 2) letter\_consec\_ok = false;

} else {

consec\_letter = 1;

prev\_letter = input[i];

}

}

i++;

}

printf(" - Lowercase letters (a-z) count: %d (required 4-7)\n", letterCount);

printf(" - No more than two consecutive same letters: %s\n", letter\_consec\_ok ? "Yes" : "No");

int digitCount = 0;

char prev\_digit = '\0';

int consec\_digit = 0;

bool digit\_consec\_ok = true;

while (i < (int)strlen(input) && isdigit(input[i])) {

digitCount++;

if (digitCount == 1) {

prev\_digit = input[i];

consec\_digit = 1;

} else {

if (input[i] == prev\_digit) {

consec\_digit++;

if (consec\_digit > 2) digit\_consec\_ok = false;

} else {

consec\_digit = 1;

prev\_digit = input[i];

}

}

i++;

}

printf(" - Digits (0-9) count: %d (required 2-4)\n", digitCount);

printf(" - No more than two consecutive same digits: %s\n", digit\_consec\_ok ? "Yes" : "No");

printf(" - Ends with \"@r\": Yes\n");

} else {

printf("Invalid identifier!\n");

printf("Reason:\n");

int i = 0;

bool has\_prefix = false;

if (input[i] == '#' || input[i] == '@' || input[i] == '!') {

has\_prefix = true;

i++;

}

int letterCount = 0;

char prev\_letter = '\0';

int consec\_letter = 0;

bool letter\_consec\_ok = true;

while (i < (int)strlen(input) && islower(input[i])) {

letterCount++;

if (letterCount == 1) {

prev\_letter = input[i];

consec\_letter = 1;

} else {

if (input[i] == prev\_letter) {

consec\_letter++;

if (consec\_letter > 2) letter\_consec\_ok = false;

} else {

consec\_letter = 1;

prev\_letter = input[i];

}

}

i++;

}

int digitCount = 0;

char prev\_digit = '\0';

int consec\_digit = 0;

bool digit\_consec\_ok = true;

while (i < (int)strlen(input) && isdigit(input[i])) {

digitCount++;

if (digitCount == 1) {

prev\_digit = input[i];

consec\_digit = 1;

} else {

if (input[i] == prev\_digit) {

consec\_digit++;

if (consec\_digit > 2) digit\_consec\_ok = false;

} else {

consec\_digit = 1;

prev\_digit = input[i];

}

}

i++;

}

if (letterCount < 4 || letterCount > 7) {

printf(" - Lowercase letters count not in 4 to 7 (found %d)\n", letterCount);

}

if (!letter\_consec\_ok) {

printf(" - More than two consecutive same letters found\n");

}

if (digitCount < 2 || digitCount > 4) {

printf(" - Digits count not in 2 to 4 (found %d)\n", digitCount);

}

if (!digit\_consec\_ok) {

printf(" - More than two consecutive same digits found\n");

}

if (i + 1 >= (int)strlen(input) || strncmp(&input[i], "@r", 2) != 0) {

printf(" - Does not end with \"@r\"\n");

}

if (!has\_prefix && !(islower(input[0]))) {

printf(" - Must start with optional '#', '@', '!' followed by lowercase letters\n");

}

}

}

} else {

printf("Invalid choice, please type Y or N.\n");

}

}

}

// ==================== Main ====================

int main() {

FILE \*input = fopen("input.txt", "r");

if (!input) {

printf("Error: Could not open input.txt\n");

return 1;

}

FILE \*output = fopen("output.txt", "w");

if (!output) {

printf("Error: Could not open output.txt\n");

fclose(input);

return 1;

}

processFile(input, output);

fclose(input);

fclose(output);

if (invalidIdentifiersCount > 0) {

printf("Invalid identifiers found in input.txt. Please remove or correct them to make the code valid.\n");

}

printf("\n==============================\n");

printf("Lexical analysis completed.\n");

printf("See 'output.txt' for detailed token categories and symbol table.\n");

printf("==============================\n");

interactiveValidator();

return 0;

}

**9. File Handling in C: Working and Advantages**

File handling in C allows programs to read/write persistent data using functions like fopen(), fclose(), fread(), fwrite(), fprintf().

**How it Works:**

* **Open:** FILE \*fp = fopen("file.txt", "r"); (modes: r, w, a).
* **Read/Write:** fscanf(fp, "%s", var); or fprintf(fp, "%s", data);.
* **Close:** fclose(fp);

In the project, fopen("input.txt", "r") reads source code; fopen("output.txt", "w") writes results.

**Advantages:**

* **Data Persistence:** Saves data beyond program execution.
* **Large Data Management:** Handles files too big for memory.
* **Portability:** Works across systems.
* **Reusability:** Data can be shared between programs.

**10. Sample Input (input.txt)**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include <ctype.h>**

**// Single-line comment: This should be skipped by the lexical analyzer**

**/\***

**Line 3**

**Line 4**

**\*/**

**#define maxcount12@r 100**

**// Structure definition with valid identifiers**

**typedef struct {**

**char nameab12@r[20];**

**char gendef34@r[6];**

**int agebc56@r;**

**struct node \*linkxy78@r;**

**} nodegh90@r;**

**nodegh90@r \*startkl12@r = NULL;**

**// Global variables with valid and invalid identifiers**

**int countmn34@r;**

**char typepq56@r[60];**

**char placevw78@r[30];**

**char datexy90@r[20];**

**int amountst12@r;**

**// Invalid identifiers to test error handling**

**int aaaa12@r;**

**int abcd111@r;**

**int abc@r;**

**int abcdefghi12@r;**

**int abcd1@r;**

**int abcd12345@r;**

**int abcd12@s;**

**int Abcd12@r;**

**int abcd-12@r;**

**int \_abcd12@r;**

**// Function declarations with valid identifiers**

**void headingkl34@r();**

**void detailsmn56@r();**

**void interop78@r();**

**void indiast90@r();**

**void receiptvw12@r();**

**void addnodekl34@r(char lname[20], char lgen[6], int lage);**

**void brochurexy56@r();**

**// Main function**

**int main() {**

**int choiceab12@r;**

**int loopbc34@r;**

**float ratecd56@r = 3.14;**

**char msgde78@r[] = "Tourism Management System"; // Valid: 6 letters (m,s,g,d,e), 2 digits, ends with @r**

**int arrayfg90@r[maxcount12@r]; // Valid: 6 letters (a,r,r,a,y,f,g), 2 digits, ends with @r**

**// Test operators**

**choiceab12@r = 5;**

**choiceab12@r++;**

**choiceab12@r--;**

**choiceab12@r += 2;**

**choiceab12@r -= 1;**

**choiceab12@r \*= 3;**

**choiceab12@r /= 2;**

**// Test conditional statements**

**if (choiceab12@r == 5) {**

**printf("Equal\n");**

**} else if (choiceab12@r != 6) {**

**printf("Not equal\n");**

**} else if (choiceab12@r <= 10 && choiceab12@r >= 0) {**

**printf("In range\n");**

**} else if (choiceab12@r > 10 || choiceab12@r < 0) {**

**printf("Out of range\n");**

**}**

**// Test loops**

**for (loopbc34@r = 0; loopbc34@r < 10; loopbc34@r++) {**

**arrayfg90@r[loopbc34@r] = loopbc34@r;**

**}**

**loopbc34@r = 0;**

**while (loopbc34@r < 5) {**

**loopbc34@r++;**

**}**

**loopbc34@r = 0;**

**do {**

**loopbc34@r++;**

**} while (loopbc34@r < 5);**

**// Test switch**

**switch (choiceab12@r) {**

**case 1: printf("One\n"); break;**

**case 2: printf("Two\n"); break;**

**default: printf("Default\n"); break;**

**}**

**// Test additional keywords**

**const int constgh12@r = 5;**

**static int staticij34@r = 0;**

**unsigned int unsigkl56@r = 10;**

**signed int sigmn78@r = -5;**

**long longop90@r = 100;**

**short shortqr12@r = 20;**

**// Additional valid identifiers**

**int @travelmn34@r = 0;**

**int #bookst56@r = 1;**

**int !tourvw78@r = 2;**

**double costxy90@r = 1500.50;**

**// Call functions**

**headingkl34@r();**

**printf("\t\t\t\t1. International Tour Packages\n");**

**printf("\t\t\t\t2. India Tour Packages\n");**

**printf("\t\t\t\tEnter Choice: ");**

**scanf("%d", &choiceab12@r);**

**switch (choiceab12@r) {**

**case 1: interop78@r(); break;**

**case 2: indiast90@r(); break;**

**default: printf("Enter Right Choice...\n"); break;**

**}**

**detailsmn56@r();**

**brochurexy56@r();**

**receiptvw12@r();**

**return 0;**

**}**

**void headingkl34@r() {**

**system("cls");**

**printf("\t\t\t\t\*\*\*\*\* Tourism Management System \*\*\*\*\*\n");**

**}**

**void indiast90@r() {**

**int choicebc12@r;**

**system("cls");**

**headingkl34@r();**

**strcpy(typepq56@r, "India Tour Package");**

**printf("\t\t\t\t1. Shimla Tour Packages 6 Days 7 Nights (18880/-)\n");**

**printf("\t\t\t\t2. Kashmir Tour Packages 5 Days 4 Nights (35500/-)\n");**

**printf("\t\t\t\t3. Kolkata Tour Packages 11 Days 10 Nights (10000/-)\n");**

**printf("\t\t\t\t4. Goa Tour Packages 4 Days 3 Nights (12000/-)\n");**

**printf("\t\t\t\t5. Kerala Tour Packages 7 Days 6 Nights (20000/-)\n");**

**printf("\t\t\t\t6. Rajasthan Tour Packages 8 Days 7 Nights (25000/-)\n");**

**printf("\t\t\t\t7. Mumbai Tour Packages 3 Days 2 Nights (8000/-)\n");**

**printf("\t\t\t\t8. Delhi Tour Packages 5 Days 4 Nights (15000/-)\n");**

**printf("\t\t\t\t9. Taj Mahal Tour Packages 2 Days 1 Night (5000/-)\n");**

**printf("\t\t\t\t10. Himalayan Tour Packages 10 Days 9 Nights (30000/-)\n");**

**printf("\t\t\t\tEnter Choice: ");**

**scanf("%d", &choicebc12@r);**

**switch (choicebc12@r) {**

**case 1: strcpy(placevw78@r, "Shimla Tour"); amountst12@r = 18880; break;**

**case 2: strcpy(placevw78@r, "Kashmir Tour"); amountst12@r = 35500; break;**

**case 3: strcpy(placevw78@r, "Kolkata Tour"); amountst12@r = 10000; break;**

**case 4: strcpy(placevw78@r, "Goa Tour"); amountst12@r = 12000; break;**

**case 5: strcpy(placevw78@r, "Kerala Tour"); amountst12@r = 20000; break;**

**case 6: strcpy(placevw78@r, "Rajasthan Tour"); amountst12@r = 25000; break;**

**case 7: strcpy(placevw78@r, "Mumbai Tour"); amountst12@r = 8000; break;**

**case 8: strcpy(placevw78@r, "Delhi Tour"); amountst12@r = 15000; break;**

**case 9: strcpy(placevw78@r, "Taj Mahal Tour"); amountst12@r = 5000; break;**

**case 10: strcpy(placevw78@r, "Himalayan Tour"); amountst12@r = 30000; break;**

**default: printf("Enter Correct Choice...\n"); break;**

**}**

**}**

**void interop78@r() {**

**int choicecd34@r;**

**system("cls");**

**headingkl34@r();**

**strcpy(typepq56@r, "International Tour Package");**

**printf("\t\t\t\t1. England Tour Packages 6 Days 7 Nights (28880/-)\n");**

**printf("\t\t\t\t2. Thailand Tour Packages 5 Days 4 Nights (15500/-)\n");**

**printf("\t\t\t\t3. New York Tour Packages 11 Days 10 Nights (567800/-)\n");**

**printf("\t\t\t\t4. Paris Tour Packages 7 Days 6 Nights (45000/-)\n");**

**printf("\t\t\t\t5. Australia Tour Packages 10 Days 9 Nights (60000/-)\n");**

**printf("\t\t\t\t6. Japan Tour Packages 8 Days 7 Nights (50000/-)\n");**

**printf("\t\t\t\t7. Dubai Tour Packages 5 Days 4 Nights (30000/-)\n");**

**printf("\t\t\t\t8. Singapore Tour Packages 4 Days 3 Nights (25000/-)\n");**

**printf("\t\t\t\t9. Maldives Tour Packages 6 Days 5 Nights (40000/-)\n");**

**printf("\t\t\t\t10. Europe Tour Packages 15 Days 14 Nights (100000/-)\n");**

**printf("\t\t\t\tEnter Choice: ");**

**scanf("%d", &choicecd34@r);**

**switch (choicecd34@r) {**

**case 1: strcpy(placevw78@r, "England Tour"); amountst12@r = 28880; break;**

**case 2: strcpy(placevw78@r, "Thailand Tour"); amountst12@r = 15500; break;**

**case 3: strcpy(placevw78@r, "New York Tour"); amountst12@r = 567800; break;**

**case 4: strcpy(placevw78@r, "Paris Tour"); amountst12@r = 45000; break;**

**case 5: strcpy(placevw78@r, "Australia Tour"); amountst12@r = 60000; break;**

**case 6: strcpy(placevw78@r, "Japan Tour"); amountst12@r = 50000; break;**

**case 7: strcpy(placevw78@r, "Dubai Tour"); amountst12@r = 30000; break;**

**case 8: strcpy(placevw78@r, "Singapore Tour"); amountst12@r = 25000; break;**

**case 9: strcpy(placevw78@r, "Maldives Tour"); amountst12@r = 40000; break;**

**case 10: strcpy(placevw78@r, "Europe Tour"); amountst12@r = 100000; break;**

**default: printf("Enter Correct Choice...\n"); break;**

**}**

**}**

**void detailsmn56@r() {**

**int loopde78@r, agefg90@r;**

**char namehi12@r[20], genjk34@r[6];**

**system("cls");**

**headingkl34@r();**

**printf("\t\t\t\tEnter Number Of Passengers: ");**

**scanf("%d", &countmn34@r);**

**printf("\t\t\t\tEnter Date (DD/MM/YY): ");**

**fflush(stdin);**

**gets(datexy90@r);**

**for (loopde78@r = 1; loopde78@r <= countmn34@r; loopde78@r++) {**

**system("cls");**

**headingkl34@r();**

**printf("\t\t\t\tEnter Details of Passenger %d\n", loopde78@r);**

**printf("\t\t\t\tEnter Name: ");**

**fflush(stdin);**

**gets(namehi12@r);**

**printf("\t\t\t\tEnter Gender (Male/Female): ");**

**gets(genjk34@r);**

**printf("\t\t\t\tEnter Age: ");**

**scanf("%d", &agefg90@r);**

**addnodekl34@r(namehi12@r, genjk34@r, agefg90@r);**

**}**

**}**

**void addnodekl34@r(char lname[20], char lgen[6], int lage) {**

**nodegh90@r \*newnodeab12@r = (nodegh90@r \*)malloc(sizeof(nodegh90@r));**

**strcpy(newnodeab12@r->nameab12@r, lname);**

**strcpy(newnodeab12@r->gendef34@r, lgen);**

**newnodeab12@r->agebc56@r = lage;**

**newnodeab12@r->linkxy78@r = NULL;**

**if (startkl12@r == NULL) {**

**startkl12@r = newnodeab12@r;**

**} else {**

**nodegh90@r \*tempbc34@r = startkl12@r;**

**while (tempbc34@r->linkxy78@r != NULL) {**

**tempbc34@r = tempbc34@r->linkxy78@r;**

**}**

**tempbc34@r->linkxy78@r = newnodeab12@r;**

**}**

**}**

**void receiptvw12@r() {**

**nodegh90@r**

**int loopfg78@r**

**system("cls");**

**headingkl34@r();**

**printf("\t\t\t\t\*\*\*\*\*\*\*\* Receipt \*\*\*\*\*\*\*\*\n");**

**printf("\t\t\t\tTour Type: %s\n", typepq56@r);**

**printf("\t\t\t\tPlace: %s\n", placevw78@r);**

**printf("\t\t\t\tAmount Per Person: %d\n", amountst12@r);**

**printf("\t\t\t\tTotal Amount: %d\n", amountst12@r \* countmn34@r);**

**printf("\t\t\t\tDate: %s\n", datexy90@r);**

**printf("\t\t\t\tPassengers:\n");**

**while (tempcd56@r != NULL) {**

**printf("\t\t\t\tPassenger %d:\n", loopfg78@r);**

**printf("\t\t\t\tName : %s\n", tempcd56@r->nameab12@r);**

**printf("\t\t\t\tGender : %s\n", tempcd56@r->gendef34@r);**

**printf("\t\t\t\tAge : %d\n", tempcd56@r->agebc56@r);**

**tempcd56@r = tempcd56@r->linkxy78@r;**

**loopfg78@r++;**

**}**

**}**

**void brochurexy56@r() {**

**system("cls");**

**headingkl34@r();**

**printf("\t\t\t\tBrochure for %s\n", placevw78@r);**

**printf("\t\t\t\tDay 1: Arrival and check-in\n");**

**printf("\t\t\t\tDay 2: Local sightseeing\n");**

**printf("\t\t\t\tDay 3: Adventure activities\n");**

**printf("\t\t\t\tDay 4: Cultural experiences\n");**

**printf("\t\t\t\tDay 5: Relaxation and spa\n");**

**printf("\t\t\t\tDay 6: Shopping and souvenirs\n");**

**printf("\t\t\t\tDay 7: Departure\n");**

**printf("\t\t\t\tEnjoy the beautiful landscapes.\n");**

**printf("\t\t\t\tExperience local cuisine.\n");**

**printf("\t\t\t\tVisit historical sites.\n");**

**printf("\t\t\t\tParticipate in festivals.\n");**

**printf("\t\t\t\tExplore nature trails.\n");**

**printf("\t\t\t\tMeet friendly locals.\n");**

**printf("\t\t\t\tCapture memorable photos.\n");**

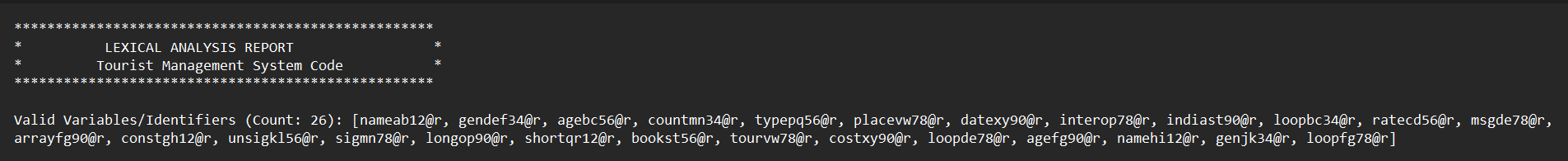
**printf("\t\t\t\tCreate lasting memories.\n");**

**}**

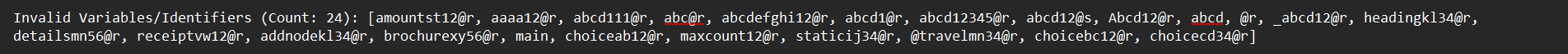
**11. Sample Output (output.txt and Terminal) and Explanation**

Output.txt Sections:

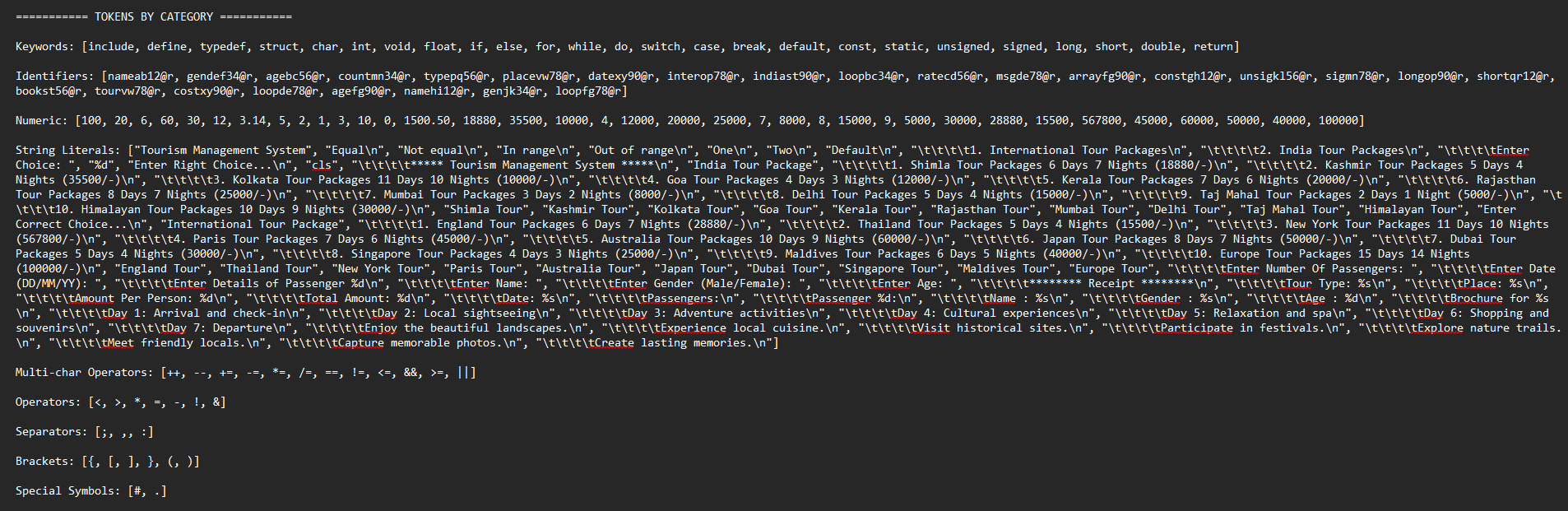
* **Valid Identifiers:** List with count



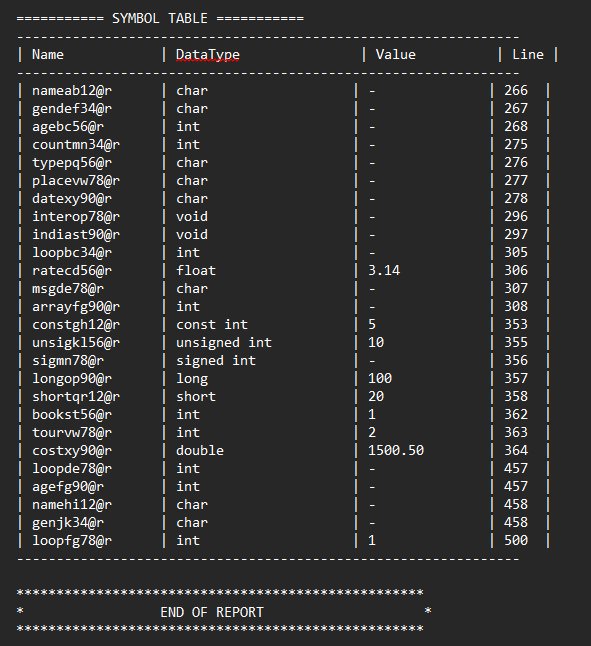
* **Invalid:** List with count



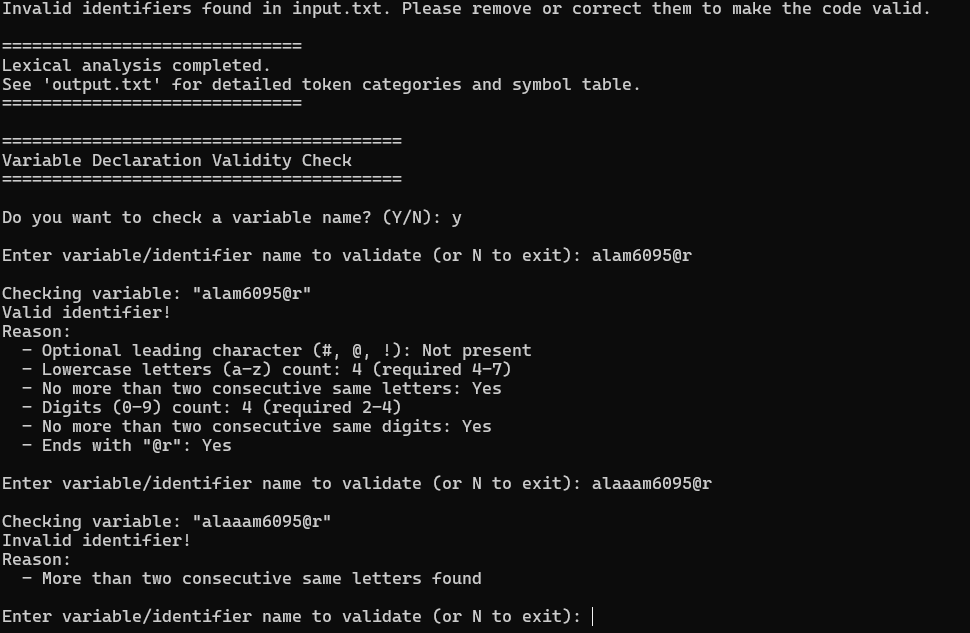
* **Tokens by Category:**



* **Symbol Table:** Table with name, type, value, line.



* **Terminal:** "Invalid identifiers found..." if any and option to check identifier validity.



**12. Conclusion**

This project successfully implements a lexical analyzer, demonstrating compiler phases, regex, and FA. It handles file I/O efficiently and validates variables rigorously. Future enhancements could include full compiler integration.

**13. References**

1. GeeksforGeeks. "Phases of a Compiler.
2. Tutorialspoint. "Lexical Analysis."
3. Wikipedia. "Regular Expression."
4. GeeksforGeeks. "Finite Automata.
5. GeeksforGeeks. "File Handling in C.”