BHARATIYA VIDYA BHAVAN'S

SARDAR PATEL INSTITUTE OF TECHNOLOGY

(Empowered Autonomous Institute Affiliated to University of Mumbai)

[Knowledge is Nectar]

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300108
Name	Hatim Sawai
Class and Batch	TE Computer Engineering Class B – Batch C
Date	26/1/2024
Lab #	1
Aim	Design a Lexical analyzer for different programming languages and implement using lex tool
Objective	To design and implement a lexical analyzer for different programming languages using lex tool
Theory	**Token in Different Contexts:** A token is essentially a unit of meaning in a specific context. It can be a physical object like a game piece, a voucher, or a symbol representing something larger, like a gift representing appreciation. In digital contexts, a token typically refers to a string of characters that represents a specific concept or unit of data. Examples: In programming languages, tokens can be keywords, identifiers, operators, punctuation marks, etc. Each token carries a specific meaning within the program's syntax. In natural language processing, tokens can be words, punctuation marks, or even sub-word units like morphemes. **Tokens in C:** C recognizes several types of tokens: Keywords: Reserved words with specific meanings within the language, like \`for\`, \`int\`, or \`while\`. Identifiers: User-defined names for variables, functions, and types. Literals: Constant values like integers, floating-point numbers, strings, and characters. Operators: Symbols used for mathematical, logical, and relational operations like \`+\`, \`*\`, or \`==\`. Punctuation: Delimiters that separate different syntactic elements, like brackets, commas, and semicolons. Whitespace: Spaces, tabs, and newlines that are generally ignored by the compiler but can affect readability. C tokenization is part of the lexical analysis phase of compilation, where the source code is broken down into these meaningful units.

Lexical Analysis:
Lexical analysis is the first stage of compiler processing that transforms the source code into a stream of tokens. It involves:

Scanning: Identifying individual characters and grouping them into tokens.
Tokenization: Classifying each token based on its syntax and semantics.
Attribute attachment: Associating additional information with each token, like its line number or data type.

Lexical analysis ensures the compiler understands the basic building blocks of the program before moving on to subsequent stages.

Lexical Analyzer:
A lexical analyzer, also known as a lexical scanner, is the software tool responsible for performing lexical analysis. It typically uses a set of rules to recognize tokens based on their patterns and context. Popular lexical analyzers include Lex and Flex.

Lex Tools:
Flex is a widely used tool for building lexical analyzers in C. It allows developers to define rules for tokenization using regular expressions and C code. Flex generates a scanner program that efficiently reads the source code and produces a stream of tokens for the compiler.

Running a Lex Program:
1. Write the Lex code: Define tokenization rules using regular expressions and actions in a `.l` file.
2. Generate the scanner: Run the `flex` command on the `.l` file to produce a C file containing the scanner code.
3. Link the scanner: Combine the scanner code with your main program code and compile them together.
4. Run the program: Execute the compiled program, and the scanner will tokenize the input while processing it.

These are just introductory explanations. Each topic deserves further exploration depending on your specific interests and level of understanding. Feel free to ask if you have any further questions or want to delve deeper into any specific aspect.

Implementation / Code

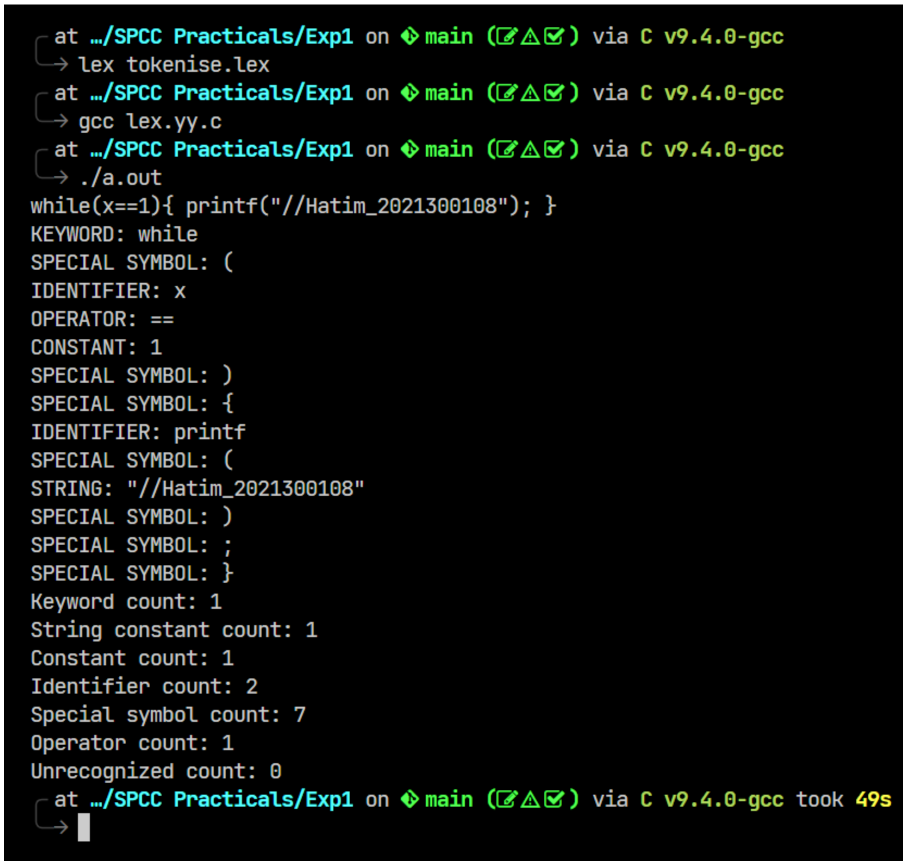
tokenise.lex:
```
%{
#include <stdio.h>
#include <string.h>

int keywordCount = 0;
int stringCount = 0;
int constantCount = 0;
int identifierCount = 0;
int specialSymbolCount = 0;
int operatorCount = 0;
int unrecognizedCount = 0;
%}

%%
"auto"|"break"|"default"|"const"|"void"|"union"|"extern"|"if"|"else"|"while"|"do"|"break"|"continue"|"int"|"double"|"float"|"return"|"char"|"case"|"sizeof"|"long"|"short"|"typedef"|"switch"|"unsigned"|"void"|"static"|"struct"|"goto" { printf("KEYWORD: %s\n", yytext); keywordCount++; }
\"[^\n\"]*\"    { printf("STRING: %s\n", yytext); stringCount++; }
[0-9]+          { printf("CONSTANT: %s\n", yytext); constantCount++; }
[a-zA-Z_][a-zA-Z0-9_]* { printf("IDENTIFIER: %s\n", yytext); identifierCount++; }
[ \t\n]         /* Ignore whitespace */
[{}()[\],;.]    { printf("SPECIAL SYMBOL: %s\n", yytext); specialSymbolCount++; }
[+\-*/%&|!<>^=]=?  { printf("OPERATOR: %s\n", yytext); operatorCount++; }
[@.]            { printf("UNRECOGNIZED: %s\n", yytext); unrecognizedCount++; }
%%

int yywrap(void) {}

int main() {
    yylex();
    printf("Keyword count: %d\n", keywordCount);
    printf("String constant count: %d\n", stringCount);
    printf("Constant count: %d\n", constantCount);
    printf("Identifier count: %d\n", identifierCount);
    printf("Special symbol count: %d\n", specialSymbolCount);
    printf("Operator count: %d\n", operatorCount);
    printf("Unrecognized count: %d\n", unrecognizedCount);
    return 0;
}
```

Output	
Conclusion	In this experiment, we learned what is a Lexical Analyzer and what is Lex, how to install it and write a program in Lex to make a lexical analyzer for C programming language, and how to run the lex code on Ubuntu.
References	[1]Google Bard - Understanding Tokens, Lexical Analysis, and Lex: Google. (26/1/2024). Google Bard https://g.co/bard/share/221fcc81077e

```
╭ at …/SPCC Practicals/Exp1 on main (📝⚠☑) via C v9.4.0-gcc
╰→ lex tokenise.lex
╭ at …/SPCC Practicals/Exp1 on main (📝⚠☑) via C v9.4.0-gcc
╰→ gcc lex.yy.c
╭ at …/SPCC Practicals/Exp1 on main (📝⚠☑) via C v9.4.0-gcc
╰→ ./a.out
while(x==1){ printf("//Hatim_2021300108"); }
KEYWORD: while
SPECIAL SYMBOL: (
IDENTIFIER: x
OPERATOR: ==
CONSTANT: 1
SPECIAL SYMBOL: )
SPECIAL SYMBOL: {
IDENTIFIER: printf
SPECIAL SYMBOL: (
STRING: "//Hatim_2021300108"
SPECIAL SYMBOL: )
SPECIAL SYMBOL: ;
SPECIAL SYMBOL: }
Keyword count: 1
String constant count: 1
Constant count: 1
Identifier count: 2
Special symbol count: 7
Operator count: 1
Unrecognized count: 0
╭ at …/SPCC Practicals/Exp1 on main (📝⚠☑) via C v9.4.0-gcc took 49s
╰→ █
```