Name: Digvijay Jondhale

Roll No: PC-32

PRN: 1032201770

SSCD LCA 05

Aim: Generate lexical analyzer for C language using LEX tool.

Theory:

- Token lexeme and pattern:

> Token: In the context of the programming languages and lexical analysis, a token is a fundamental unit of a language's syntax. Tokens represent specific elements or symbols in the source code, such as keywords (eg., "if", "while"), identifiers (e.g, variable names), literals (eg., numbers or strings), and operators (eg., "+", "="). Tokens are used by the parser to understand the structure of the code and perform syntactic analysis.

> lexeme: A lexeme is the actual sequence of characters in the source code that corresponds to a specific token. It is the concrete representation of a token in the source code. For example, in the statement "int x = 42";, the lexemes for the tokens are "int," "x", "=", and "42.".

> Pattern: A pattern is a description or regular expression that defines the structure or format of lexemes. It specifies the rules that lexemes must follow to be recognized as as a particular token. For example, a pattern for recognizing integer literals in a programming language might be "\d+".

- Use of Regular Expression (RE) in specifying lexical structure of a language:

> Define Patterns: Programmers use regular expressions to define patterns for various tokens in the language. For example, a pattern for identifying identifiers may be "[a-zA-Z_][a-zA-Z0-9_]*," which matches valid variable names.

> Tokenization: The lexer or lexical analyzer processes the source code character by character and tries to match the input against the defined regular expressions. When a pattern matches a portion of the input, it generates a token with the associated lexeme.

> Generating Tokens: Tokens are created based on the patterns that matched. Each token contains the lexeme (the matched text) and the token type (eg, identifier, keyword, number). These tokens are then passed on to the parser for further syntactic analysis.

> Handling Ambiguity: Regular expressions can help handle ambiguity by allowing you to specify rules for resolving situations where the input could match multiple patterns. This is often done by giving priority to the first matching pattern encountered or using other disambiguation rules.

- Format of lex Specification and Execution Steps of a lex File (4.1): lexical analyzers for programming languages are often generated using tools like lex (or its open-source counterpart, Flex). These tools use a specification file with a specific format, typically with the ".l" extension. Here's an overview of the

- Lex Specification File Format (*.l):
  - Definations: You can define regular expressions and macros at the begining of the file. These definations are enclosed in "%{" and "%}" delimiters.
  - Rules: The main part of the file consists of rules that specify regular expressions and associated actions. Rules are written in the format:

    $\boxed{\text{regex action;}}$

    where "regex" is a regular expression pattern, and "action" is the code or action to be executed when the pattern is matched.
  - User Code: You can include C code within the specification file, usually enclosed in "%{" and "%}" delimiters, for custom actions or additional code.

- Execution Steps:

1) Write the Lex Specification: Create a ".l" file that defines the lexical structure of the language using regular expressions and associated actions.
2) Generate Lexer Code: Use a tool like Lex or Flex to generate C code for the lexical analyzer based on the ".l" file. This code includes functions for tokenizing the input source code.
3) Compile and Link: Compile the generated C code and link it with your parser or the rest of your compiler or interpreter.
4) Execute: Run the resulting executable, which will tokenize the input source code according to the defined lexical rules and generate a sequence of tokens that can be used for further parsing and analysis.

Code :

```
%{
#include<stdio.h>
#include<string.h>
typedef struct node {
char ID[10], Type[10];
struct node *next;
}node;
node *head = NULL, *curr=NULL, *prev=NULL;
int declare_flag = 0;
%}

delim [ \t\n]
ws {delim}+
letter [a-zA-Z]
digit [0-9]
package {letter}+\.{letter}+(\.{letter}+[*]?)*
Id {letter}({letter}|{digit}|(\_))*
Number {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
anychar ({letter}|{digit}|[ !@#$%^&*()_.,?:])*
String \"{anychar}\"

%%

{ws}
"System.out.println"|"System.out.print" {printf("\n%s : Print
Statement",yytext);}
{package} {printf("\n%s : Package",yytext);}
class+" "+{letter}+ {printf("\n%s : Class",yytext);}
import|if|else {printf("\n%s : Keyword",yytext);}
{String} {printf("\n%s : String",yytext);}
```

```
"+"|"-"|"*"|"/"|"=" {printf("\n%s : Operator",yytext);}
{Number} {printf("\n%s : Number",yytext);}
"("|")"|"{"|"}"|";"|"}" {printf("\n%s : Punctuation",yytext);}
int|float|char|double {
printf("\n%s : Data Type",yytext);
prev = curr;
curr = curr->next;
curr = (node*)malloc(sizeof(node));
prev->next=curr;
strcpy(curr->Type, yytext);
declare_flag=1;}
{Id} {
printf("\n%s : Identifier",yytext);
if(declare_flag==1)
{
strcpy(curr->ID, yytext);
declare_flag=0;
}}
"*/"
"//"+{anychar}*"\n"|"/*"+{anychar}*"*/" {printf("\n%s :
Comment",yytext);}
%%
int main()
{
head = (node*)malloc(sizeof(node));
curr = head;
yyin=fopen("Java_Sample.txt","r");
yylex();
printf("\n\nSymbol Table:\nID\tType\n");
node *temp = head->next;
while(temp!= NULL)
{
```

```
printf("%s\t%s\n",temp->ID,temp->Type);
temp=temp->next;
}
}
int yywrap()
{
return 1;
}
```

Input (Java Code Txt File) :

```
import java.io.*
class main
{
//This is a comment
int a=1;
int b=2;
int c;
c=a+b;
System.out.println("Hello World: " +c);
}
```

Output :

```
(base) digvijay@Digvijays-MacBook-Air LCA 05 % lex Code05.l
(base) digvijay@Digvijays-MacBook-Air LCA 05 % cc lex.yy.c
(base) digvijay@Digvijays-MacBook-Air LCA 05 % ./a.out

import : Keyword
java.io : Package.
* : Operator
class main : Class
{ : Punctuation
//This is a comment
 : Comment
int : Data Type
a : Identifier
= : Operator
1 : Number
; : Punctuation
int : Data Type
b : Identifier
= : Operator
2 : Number
; : Punctuation
int : Data Type
c : Identifier
; : Punctuation
c : Identifier
= : Operator
a : Identifier
+ : Operator
b : Identifier
; : Punctuation
System.out.println : Print Statement
( : Punctuation
"Hello World: " : String
+ : Operator
c : Identifier
) : Punctuation
; : Punctuation
} : Punctuation

Symbol Table:
ID       Type
a        int
b        int
c        int
(base) digvijay@Digvijays-MacBook-Air LCA 05 %
```