Vivekanand Education Society's Institute of Technology

Department of Computer Engineering

Subject: SPCC

Class :- CMPN Semester:- 6 Div :- D12A

Roll No: 32	Name: Radhika.Katiyara		
Exp No: 1	Title: Implementing lexical analyzer using C/ Java/Python for recognizing keywords, operators,identifiers and special operators		
DOP:	30-1-22	DOS:	2-2-22
GRADE:		LAB OUTCOMES: LO1	SIGNATURE:

Radhika. J. K.
D12A 32

System Programming & Compiler Construction

Experiment No. 1

Aim:
Implementation of Lexical Analyzer in C / Java / Python

Theory:

1. What is Lexical Analyzer?

→ Lexical analysis is the first phase of the compiler. The main task of lexical analysis is to identify the set of a valid word of the language that occurs in an input stream. A program or function which performs lexical analysis is called a lexical analyzer, lexer or scanner. It often exists as a single entity which is called by the parser or another function.

2. Functions of Lexical Analyzer:

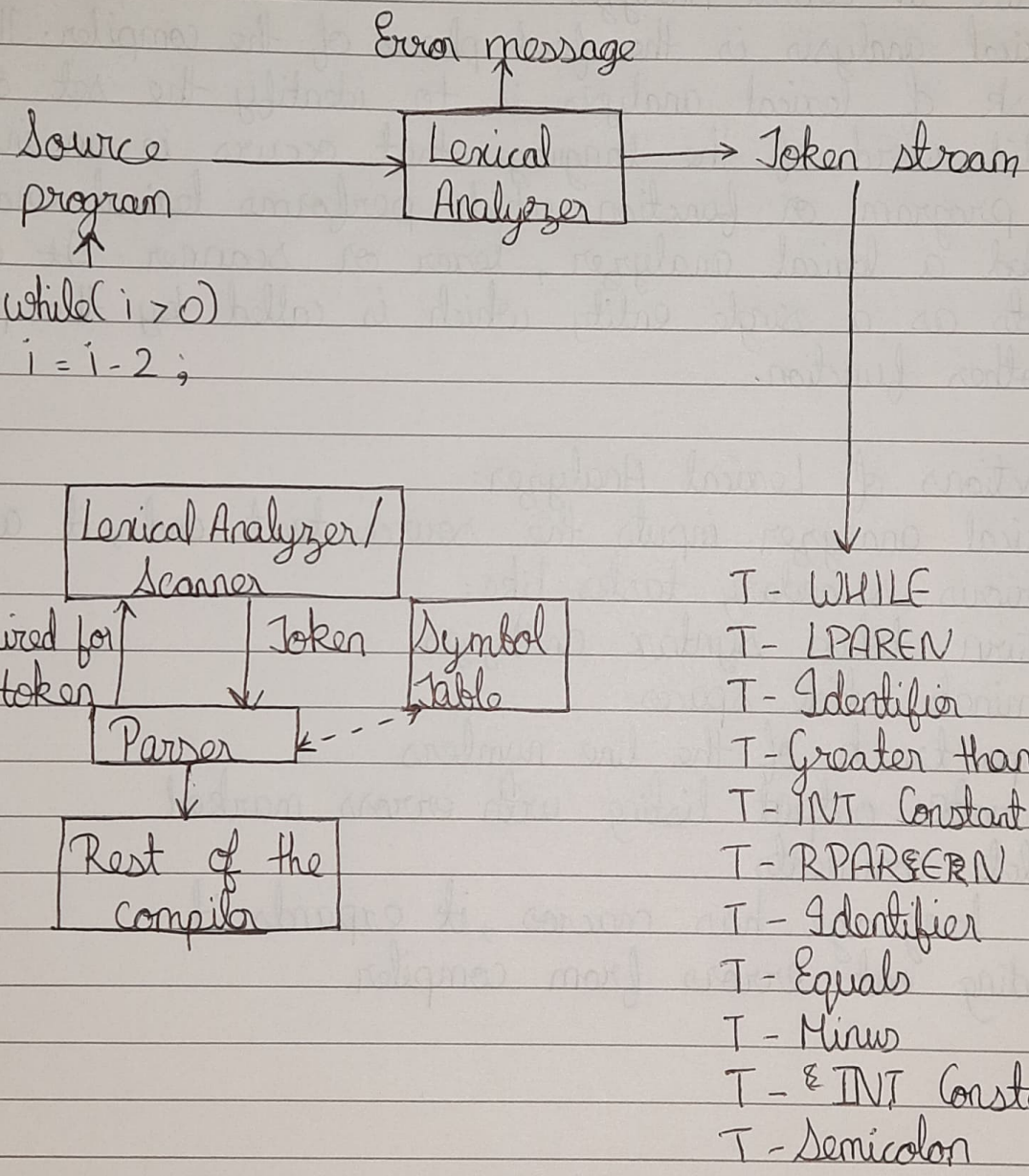
→ Lexical analyzer inputs the source text and, it accomplishes various secondary tasks like:

- Return token syntax analyzer.
- Eliminate white spaces
- Keeps track of the line numbers
- Generates output listing with errors marked
- Delete comments
- If language has macros, it expands it
- Finding out errors from compiler

Radhika. J. K.
D12A 32

3. Significance of Lexical Analyzer with block diagram

→ The stream of characters making up source program is read from left-to-right by the lexer and is grouped into tokens, which may be handled more easily by the parser. It acts as an interface between the source program to be compiled and later stages of the compiler



Radhika. J. K.
D12A 32.

4. Role of Finite Automata in Lexical Analyzer.

→ Finite Automata is a machine with finite number of states in which machines can perform. There is a distinctive start state from which the machine starts. It is recognized for a language that takes as input a string x and answer "yes" if x is a sentence of the language and "no" otherwise. We compile a regular expression into recognizer by constructing a generalized transition diagram called finite automata.

An FSM consists of:

a. Finite set of states
b. Set of transition from state to another
c. A special start ~~&~~ state
d. A set of final or accepting states.

5. What is Lexeme, Token and Pattern?

→ Lexeme: A Lexeme is a basic ~~lexed~~ lexical unit of a language consisting of one word or several words. That corresponds to a set of words that are different forms of the same word. For example: if, 10.0, t, etc.

Tokens: Tokens are a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Example:

Type Tokens - (a-z), (0-9), id

Alphabetic tokens - "Keywords"

Radhika J.K.
D12A 32

Pattern: A pattern is a term, template or model, or a set of rules which can be used to make or to generate things or part of a thing especially if the things that are generated have enough in common for the underlying pattern to be inferred.

Example:

For the lexeme "pi, count D2",
the pattern will be: letter followed by letter & digits

Conclusion:

Hence, we have understood the role of lexical analyzer in phases of compilation and implemented it successfully.

System Programming & Compiler Construction

Experiment no.1

Program code:

```
import re
f = open("expt1.c", 'r')
text = f.read()
symbols = ['!', '@', '#', '$', '%', '&', '^', '*']
operators = ['+', '-', '*', '/', '=', '+=', '-=', '==', '<', '>', '<=', '>=']
keywords = ['auto','break', 'case', 'char', 'const', 'continue', 'default', 'do',
            'double', 'else', 'enum', 'extern', 'float', 'for', 'goto', 'if',
        'int', 'long', 'register', 'return', 'short', 'signed', 'sizeof', 'static',
        'struct', 'switch', 'typedef', 'union', 'unsigned', 'void', 'volatile', 'while']
delimiters = [' ', '    ', '.', ',', '\n', ';', '(', ')', '<', '>', '{', '}', '[', ']']
in_keywords = []
in_spl_symbols = []
in_operators = []
in_delimiters = []
in_identifiers = []
in_constants = []
tokens = []
isStr = False
isWord = False
isCmt = 0
token = ''
for i in text:
    if i == '/':
        isCmt = isCmt+1
```

```
    elif isCmt == 2:
        if i == '\n':
            token = ''
            isCmt = 0
    elif i == "'" or i == '"':
        if isStr:
            tokens.append(token)
            token = ''
        isStr = not isStr
    elif isStr:
        token = token+i
    elif i in symbols:
        tokens.append(i)
    elif i.isalnum() and not isWord:
        isWord = True
        token = i
    elif (i in delimiters) or (i in operators):
        if token:
            tokens.append(token)
            token = ''
        if not (i==' ' or i=='\n' or i=='    '):
            tokens.append(i)
    elif isWord:
        token = token+i
for token in tokens:
    if token in symbols:
        in_spl_symbols.append(token)
```

```
    elif token in operators:
        in_operators.append(token)
    elif token in keywords:
        in_keywords.append(token)
    elif re.search("^[_a-zA-Z][_a-zA-Z0-9]*$",token):
        in_identifiers.append(token)
    elif token in delimiters:
        in_delimiters.append(token)
    else:
        in_constants.append(token)
print("\tKeywords : " , in_keywords);
print("\n\tSpecial Symbols : " , in_spl_symbols);
print("\n\tOperators : " , in_operators);
print("\n\tIdentifiers : " , in_identifiers);
print("\n\tDelimiters : " , in_delimiters);
f.close()
```

InputProg - Notepad

File Edit Format View Help

```
#include <stdio.h>
int main() {

    int number1, number2, sum;

    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    sum = number1 + number2;

    printf("%d + %d = %d", number1, number2, sum);
    return 0;
}
```

Output:

```
C:\Windows\System32\cmd.exe

C:\SPCC_Lab>python P1.py
        Keywords :  ['int', 'int', 'return']

        Special Symbols :  ['#', '&', '&']

        Operators :  ['<', '>', '=', '+']

        Identifiers :  ['include', 'stdio', 'h', 'main', 'number1', 'number2', 'sum', 'printf', '
scanf', 'number1', 'number2', 'sum', 'number1', 'number2', 'printf', 'number1', 'number2', 'sum']


        Delimiters :  ['.', '(', ')', '{', ',', ',', ';', '(', ')', ';', '(', ',', ',', ')', ':',
 ':', '(', ',', ',', ',', ')', ':', ':', '}']

C:\SPCC_Lab>
```