

Experiment No 4

Aim: Write a program to implement Lexical analyzer

Learning Objective: Converting a sequence of characters into a sequence of tokens .

Theory:

THE ROLE OF LEXICAL ANALYZER

The lexical analyzer is the first phase of a compiler. Its main task is to read the input

characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

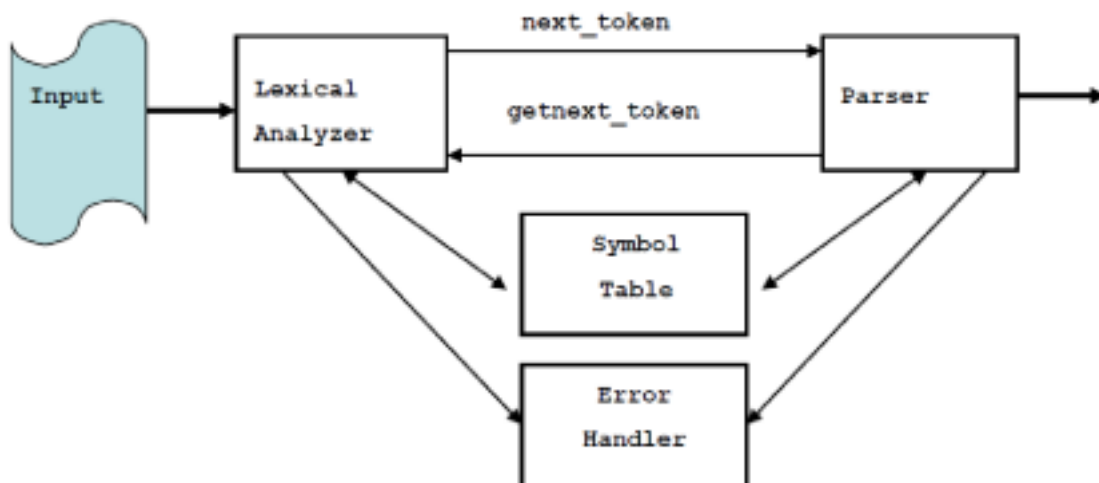


Figure 4.1 Interaction of Lexical Analyzer with Parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white spaces in the form of blank, tab, and new line characters. Another is correlating error messages from the compiler with the source program. Sometimes lexical analyzers are divided into a cascade of two phases first called “scanning” and the second “lexical analysis”. The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

Implementation Details

1. Read the high level language as source program
2. Convert source program in to categories of tokens such as Identifiers, Keywords, Constants, Literals and Operators.

Test cases:

1. Input undefined token

Implementation:

```
import re

s = input("Enter filename : ")
f = open(s, 'r')
text = f.read()

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']

Specialsymbols = [' ', ' ', '.', ',', '\n', ';', '(', ')', '<', '>',
                  '{', '}', '[', ']', '!', '@', '#', '$', '&', '^', '*']

in_keywords = []
in_spl_symbols = []
in_operators = []
in_constants = []
in_invalidconstants = []
in_Specialsymbols = []
in_identifiers = []

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:
```

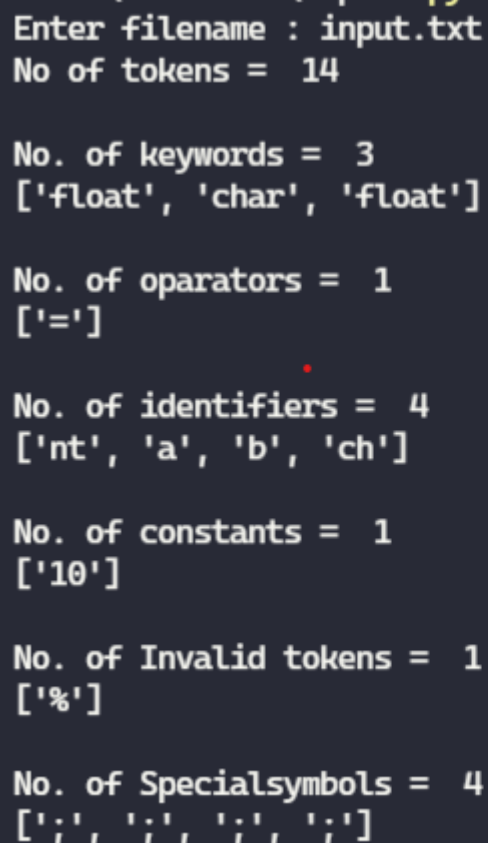
```

        token = token+i
    elif i.isalnum() and not isWord:
        isWord = True
        token = i
    elif (i in Specialsymbols) 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
    elif i == '%':
        print("Invalid Operators\n")
for token in tokens:
    if 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]*$",token):
        in_identifiers.append(token)
    elif token in Specialsymbols:
        in_Specialsymbols.append(token)
    elif re.search("[0-9]*$",token):
        in_constants.append(token)
    else:
        in_invalidconstants.append(token)
print("No. of tokens = ", len(tokens))
print("\nNo. of keywords = ",len(in_keywords))
print(in_keywords);
print("\nNo. of operators = ",len(in_operators))
print(in_operators);
print("\nNo. of identifiers = ",len(in_identifiers))
print(in_identifiers);
print("\nNo. of constants = ",len(in_constants))
print(in_constants);
print("\nNo. of Invalid tokens = ",len(in_invalidconstants))
print(in_invalidconstants);
print("\nNo. of Specialsymbols = ",len(in_Specialsymbols))
print(in_Specialsymbols);
f.close()

```

Input File:

```
int a = 10 ;  
  
float b ;  
  
char ch ;  
  
float % ;
```

Output:

```
Enter filename : input.txt  
No of tokens = 14  
  
No. of keywords = 3  
['float', 'char', 'float']  
  
No. of operators = 1  
['=']  
  
No. of identifiers = 4  
['int', 'a', 'b', 'ch']  
  
No. of constants = 1  
['10']  
  
No. of Invalid tokens = 1  
['%']  
  
No. of Specialsymbols = 4  
[';', ':', ':', ':', ':']
```

Conclusion:

1. Lexical Analysis is the very first phase in the compiler designing. A Lexer takes the modified source code which is written in the form of sentences . In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analyzer breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code
2. Learned how to represent identifier ,keyword ,constants, special symbol and operators.

Post Lab Questions:

Exp SPCC (PostLab) 8875

Q1. Take any one statement of programming language and parse into different phases of compiler.

solⁿ → input statement: while $(a > b)$, $c = b + 2.0 * a - 50$,

Source code (HLL) $C = b + 2.0 \times a - 50$

↓ Lexical analyzer

Lexeme	Taken	Symbol table
C	id ₁	<id ₁ , entry of C in ST>
=	RELOP	<assign>
b	id ₂	<id ₂ , entry of b in ST>
+	ADDOP	<ADD>
2.0	Num	<CONSTANT>
*	MULOP	<MUL>
a	id ₃	<id ₃ , entry of a in ST>
-	MINUSOP	<MINUS>
50	NUM	<CONSTANT>

↓ Syntax analysis

```

graph TD
    Root["="] --- C["C"]
    Root --- Node1["*"]
    Node1 --- Node2["+"]
    Node1 --- Node3["-"]
    Node2 --- b["b"]
    Node2 --- 2.0["2.0"]
    Node3 --- a["a"]
    Node3 --- 50["50"]
    
```

↓ Semantic analysis

No action are required during the phase as all variable and constants are in same case or type.

⇓ Intermediate code generation

$$t_1 = b + 2.0$$

$$t_2 = a - 50$$

$$t_3 = t_1 * t_2$$

$$c = t_3$$

⇓ code optimization

$$t_1 = b + 2.0$$

$$t_2 = a - 50$$

$$c = t_1 * t_2$$

⇓ code generation

$$R_1 \leftarrow b$$

$$R_3 \leftarrow b + 2.0$$

$$R_4 \leftarrow a - 50$$

$$R_2 \leftarrow 2.0$$

$$R_1 \leftarrow a$$

$$R_1 \leftarrow R_3 * R_4$$

$$R_2 \leftarrow 50$$

$$C \leftarrow R_1$$



Q2 What is symbol table?

- ① Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, object, classes, interfaces, etc. symbol table is used by both the analysis and the synthesis parts of a compiler.

A Symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

< symbol name, type, attribute >

For example, if a symbol table has to store information about the following variable declaration:

static int interest;

then it should store the entry such as:

< interest, int, static >

The attribute clause contains the entries related to the name.