# Experiment 03: Lexical Analyzer

**Learning Objective**: Students should be able to design a handwritten lexical analyzer.

**Tools:** Jdk1.8,Turbo C/C++, Python, Notepad++

## Theory:

**Design of lexical analyzer**

. Allow white spaces, numbers, and arithmetic operators in an expression
. Return tokens and attributes to the syntax analyzer
. A global variable tokenval is set to the value of the number
. Design requires that
- A finite set of tokens be defined
- Describe strings belonging to each token

**Regular Expressions**

• We use regular expressions to describe tokens of a programming language.
• A regular expression is built up of simpler regular expressions (using defining rules)
• Each regular expression denotes a language.
• A language denoted by a regular expression is called as a **regular set**.

**Regular Expressions (Rules)**

Regular expressions over alphabet S

Regular Expression — Language it denotes
ε — { ε }
aЄ∑ — S {a}
(r1) | (r2) — L(r1) È L(r2)
(r1) (r2) — L(r1) L(r2)
(r)* — (L(r))*
(r) — L(r)

• (r)+ = (r)(r)*
• (r)? = (r) | ε
• We may remove parentheses by using precedence rules.
* — highest
concatenation — next
| — lowest

**How to recognize tokens**

Construct an analyzer that will return <token, attribute> pairs
We now consider the following grammar and try to construct an analyzer that will return **<token, attribute> pairs.**

**relop** — **< | = | = | <> | = | >**
**id** — **letter (letter | digit)***
**num** — **digit+ ('.' digit+)? (E ('+' | '-')? digit+)?**
**delim** — **blank | tab | newline**
**ws** — **delim+**

Using set of rules as given in the example above we would be able to recognize the tokens. Given a regular expression R and input string x, we have two methods for determining whether x is in L(R). One approach is to use algorithm to construct an NFA N from R, and the other approach is using a DFA.

**Finite Automata**

• A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.

– We call the recognizer of the tokens as a *finite automaton*.

• A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
• This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
• Both deterministic and non-deterministic finite automaton recognizes regular sets.
• Which one?

– deterministic – faster recognizer, but it may take more space
– non-deterministic – slower, but it may take less space
– Deterministic automatons are widely used lexical analyzers.

• First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

**Algorithm1:** Regular Expression ⟶ NFA ⟶ DFA (two steps: first to NFA, then to DFA)
**Algorithm2:** Regular Expression ⟶ DFA (directly convert a regular expression into a DFA)
**Converting Regular Expressions to NFAs**

• Create transition diagram or transition table i.e. NFA for every expression
• Create a zero state as start state and with an e-transition connect all the NFAs and prepare a combined NFA.

<u>**Algorithm**</u>:for lexical analysis

1) Specify the grammar with the help of regular expression
2) Create transition table for combined NFA
3) read input character
4) Search the NFA for the input sequence.
5) On finding accepting state
    i. if token is id or num search the symbol table
        1. if symbol found return symbol id
        2. else enter the symbol in symbol table and return its id.
    ii. Else return token
6) Repeat steps 3 to 5 for all input characters.

## <u>Input:</u>

```
#include<stdio.h>
void main(){
int a,b;
printf("Hello");
getch();
}
```

## <u>Output:</u>

```
Preprocessor Directives: #include
```

```
Header File: stdio.h
Keyword : void, main, int, getch
Symbol: <> , ; ( ) ; }
Message: Hello
```

**Application:** To design a lexical analyzer.

**Design:**

```
key_words = ["void", "main", "printf", "getch", "int"]
pre_processor = ["#include"]
header = ["stdio.h"]
symbols = ['<', '>', ',', ';', '(', ')', '{', '}']

def lexical_analyzer(code):
    tokens = []
    lines = code.split(' ')
    for line in lines:
        words = line.split()
        for word in words:
            # Check for preprocessor directives
            if word in pre_processor:
                tokens.append(('PREPROCESSOR', word))
            # Check for headers
            elif word in header:
                tokens.append(('HEADER', word))
            # Check for symbols
            elif any(char in word for char in symbols):
                for char in word:
                    if char in symbols:
                        tokens.append(('SYMBOL', char))
            # Check for keywords
            elif word in key_words:
                tokens.append(('KEYWORD', word))
            # If none of the above, assume it's an identifier
            else:
                # Remove non-alphanumeric characters from the word
                cleaned_word = ''.join(char for char in word if char.isalnum())
                if cleaned_word:
                    tokens.append(('IDENTIFIER', cleaned_word))
    return tokens

code = ''' #include < stdio.h >
void main ( )
{
    int a, b;
    printf ("Hello");
    getch ( );
} '''

tokens = lexical_analyzer(code)
for token in tokens:
    print(token)
```

**OUTPUT:**

```
('PREPROCESSOR', '#include')
('SYMBOL', '<')
('HEADER', 'stdio.h')
('SYMBOL', '>')
```

```
('KEYWORD', 'void')
('KEYWORD', 'main')
('SYMBOL', '(')
('SYMBOL', ')')
('SYMBOL', '{')
('KEYWORD', 'int')
('SYMBOL', ',')
('SYMBOL', ';')
('KEYWORD', 'printf')
('SYMBOL', '(')
('SYMBOL', ')')
('SYMBOL', ';')
('KEYWORD', 'getch')
('SYMBOL', '(')
('SYMBOL', ')')
('SYMBOL', ';')
('SYMBOL', '}')
```

**Result and Discussion:**

The lexical analyzer effectively categorizes C-like code elements, accurately recognizing keywords, symbols, and identifiers. This foundational step facilitates subsequent stages in compiler construction. The system's adaptability allows for expansion, making it a versatile tool for processing and understanding diverse programming languages.

**Learning Outcomes:** The student should have the ability to

LO1: Appreciate the role of lexical analyzer in compiler design

LO2: Define role of lexical analyzer.

**Course Outcomes**: Upon completion of the course students will be able to Illustrate the working of the compiler and handwritten /automatic lexical analyzer.

**Conclusion:**

**For Faculty Use**

| Correction Parameters | Formative Assessment [40%] | Timely completion of Practical [ 40%] | Attendance / Learning Attitude [20%] | |
|---|---|---|---|---|
| Marks Obtained | | | | |