Experiment 04 : Lexical Analyzer

Learning Objective: Student should be able to design handwritten lexical analyser.

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.

Input:

#include<stdio.h>
void main()
{
inta,b;
printf(“Hello”);
getch();
}

Output:

Preprocessor Directives: #include
Header File: stdio.h
Keyword : void main intgetch
Symbol: <> , ; ( ) ; }
Message: Hello

Application: To design lexical analyzer.

Design:

```
KEYWORDS = ["alignas",

"alignof",

"and",

"and_eq",

"asm",

"atomic_cancel",

"atomic_commit",

"atomic_noexcept",

"auto",

"bitand",

"bitor",

"bool",
```

```
"break",

"case",

"catch",

"char",

"char8_t",

"char16_t",

"char32_t",

"class",

"compl",

"concept",

"const",

"consteval",

"constexpr",

"constinit",

"const_cast",

"continue",

"co_await",

"co_return",

"co_yield",

"decltype",

"default",

"delete",

"do",

"double",

"dynamic_cast",
```

```
"else",

"enum",

"explicit",

"export",

"extern",

"false",

"float",

"for",

"friend",

"goto",

"if",

"inline",

"int",

"long",

"mutable",

"namespace",

"new",

"noexcept",

"not",

"not_eq",

"nullptr",

"operator",

"or",

"or_eq",

"private",
```

```
"protected",

"public",

"reflexpr",

"register",

"reinterpret_cast",

"requires",

"return",

"short",

"signed",

"sizeof",

"static",

"static_assert",

"static_cast",

"struct",

"switch",

"synchronized",

"template",

"this",

"thread_local",

"throw",

"true",

"try",

"typedef",

"typeid",

"typename",
```

```
"union",

"unsigned",

"using",

"virtual",

"void",

"volatile",

"wchar_t",

"while",

"xor",

"xor_eq"]


PREPROCESSORS = ['#define', '#undef', '#include', '#if', '#ifdef',
'#ifndef', '#else', '#elif', '#elifdef', '#elifndef', '#endif', '#line',
'#error', '#pragma']


SYMBOLS = ['=', '+=', '/=', '*=', '-=', '==', '!=', '<', '<=', '>', '>=',
'&', '&', '&&', '|', '||', '!', '>>', '>>>', '<<', '<<<' ,  ';', '{', '}',
'[', ']', '(', ')']


# header files

# identifires

# keywords

# symbol

# preprocessor

# comments


code = []
```

```python
with open('a.cpp') as f:
    code = f.readlines()

inlineComment = False
blockComment = False

output = []
for line in code:
    for word in line.split():
        if word == '//':
            break
        if word in KEYWORDS:
            output.append([word, "KEYWORD"])
        elif word in SYMBOLS:
            output.append([word, "SYMBOL"])
        elif word in PREPROCESSORS:
            output.append([word, "PREPROCESSOR"])
        elif word.isdigit():
            output.append([word, "CONSTANT"])
        elif word[0] == '<' and word[-1] == '>':
            output.append([word, "FILE DIRECTIVE"])
        elif word[0] == '"':
            output.append([word])
        else:
```

```
            output.append([word, "IDENTIFIER"])


for i in output:
    for j in i:
        print(j, end=" ")
    print()
```

a.cpp

```
#include <iostream>
using namespace std ;


int main ( ) {


  string s = "Hello world" ;


  cout << s << "\n" ;


  return 0 ;
}
```

Output:

```
E:\TE\SEM 6\SPCC\expt5>python a.py
#include PREPROCESSOR
<iostream> FILE DIRECTIVE
using KEYWORD
namespace KEYWORD
std IDENTIFIER
; SYMBOL
int KEYWORD
main IDENTIFIER
( SYMBOL
) SYMBOL
{ SYMBOL
string IDENTIFIER
s IDENTIFIER
= SYMBOL
"Hello
world" IDENTIFIER
; SYMBOL
cout IDENTIFIER
<< SYMBOL
s IDENTIFIER
<< SYMBOL
"\n"
; SYMBOL
return KEYWORD
0 CONSTANT
; SYMBOL
} SYMBOL
```

Result and Discussion:

- Leant about the lexical analyzer
- using space as delimiter
- it makes easy to find syntactical errors

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 design handwritten laexical analyzer using HL programming language.

Conclusion:

Learnt about lexical analyzer and implementation of cpp lexical analyzer using python

For Faculty Use

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