# Assignment no. A3
# Roll no.4202

**TITLE:**

Lexical analyzer for HLL using LEX

**PROBLEM STATEMENT**

Lexical analyzer for sample language using LEX.

**OBJECTIVE**

- Understand the importance and usage of LEX automated tool
- Appreciate the role of lexical analysis phase in compilation
- Understand the theory behind design of lexical analyzers and lexical analyzer generator

**S/W AND HARDWARE REQUIREMENT**

- Linux OS (Fedora 20)
- 64-bit Fedora or equivalent OS with 64-bit Intel-i5/ i7
- LEX
- YACC

**Theory:**

Lex is a tool for generating programs that perform pattern-matching on text. It is a tool for generating scanners i.e. Programs which recognize lexical patterns in text. The description of the scanner is in the form of pairs of regular expressions and C code calls rules. Lex generates as output a C source file called lex.yy.c

**The general format of lex source (lex specification) shall be:**

Definitions
%%
Rules
%%
User Subroutines

- The **definition** section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how lex operates.

- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

**How the input is matched? :**

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the `flex` input file is chosen.

Once the match is determined, the text corresponding to the match (called the *token*) is made available in the global character pointer `yytext`, and its length in the global integer `yyleng`. The *action* corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the **default rule** is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal `flex` input is: which generates a scanner that simply copies its input (one character at a time) to its output.

**Actions in lex :**

The action to be taken when a RE is matched can be a C program fragment or the special actions described below; the program fragment can contain one or more C statements, and can also include special actions.

Four special actions shall be available:

**|** The action '|' means that the action for the next rule is the action for this rule.

**ECHO:** Write the contents of the string yytext on the output.

**REJECT:** Usually only a single expression is matched by a given string in the input. REJECT means "continue to the next expression that matches the current input", and shall cause whatever rule was the second choice after the current rule to be executed for the same input. Thus, multiple rules can be matched and executed for one input string or overlapping input strings.

**BEGIN The action:**

BEGIN newstate;

**Regular Expressions in lex**

- "..." - Any string enclosed in double-quotes shall represent the characters within the double-quotes as themselves, except that backslash escapes.

- <state>r, <state1,state2,...>r - The regular expression r shall be matched only when the program is in one of the start conditions indicated by state, state1, and so on.

- r/x - The regular expression r shall be matched only if it is followed by an occurrence of regular expression x

- * - matches zero or more occurrences of the preceding expressions

- [] - a character class which matches any character within the brackets. If the first character is a circumflex '^' it changes the meaning to match any character except the ones within brackets.

- ^ - matches the beginning of a line as the first characters of a regular expression. Also used for negation within square brackets.
- {} - Indicates how many times the previous pattern is allowed to match, when containing one or two numbers.

- $ - matches the end of a line as the last character of a regular expressions

- \ - used to escape metacharacters and a part of usual c escape sequences e.g '\n' is a newline character while '*' is a literal asterisk.

- + - matches one more occurrences of the preceding regular expression.

- | - matches either the preeceeding regular expression or the following regular expression.

- ( ) - groups a series of regular expressions together, into a new regular expression.

**Examples :**

DIGIT [0-9]+
IDENTIFIER [a-zA-Z][a-zA-Z0-9]*

The functions or macros that are accessible to user code:

int yylex(void)

Performs lexical analysis on the input; this is the primary function generated by the lex utility. The function shall return zero when the end of input is reached; otherwise, it shall return non-zero values (tokens) determined by the actions that are selected.

int yymore(void)

When called, indicates that when the next input string is recognized, it is to be appended to the current value of yytext rather than replacing it; the value in yyleng shall be adjusted accordingly.

int yyless(int n)

Retains n initial characters in yytext, NUL-terminated, and treats the remaining characters as if they had not been read;the value in yyleng shall be adjusted accordingly.

int yywrap(void)

Called by yylex() at end-of-file; the default yywrap() shall always return 1. If the application requires yylex() to continue processing with another source of input, then the application can include a function yywrap(), which associates another file with the external variable FILE * yyin and shall return a value of zero.

int main(int argc, char *argv[]): Calls yylex() to perform lexical analysis, then exits. The user code can contain main() to perform application-specific operations, calling yylex() as applicable.

**Mathematical Model :**

Let S be the solution for the system.
S= { St , E , I , O , F, DD , NDD}

Where,
St is an initial state such that St={Fl | Fl is lex file}. At initial state, system consists of one lex file with three sections.

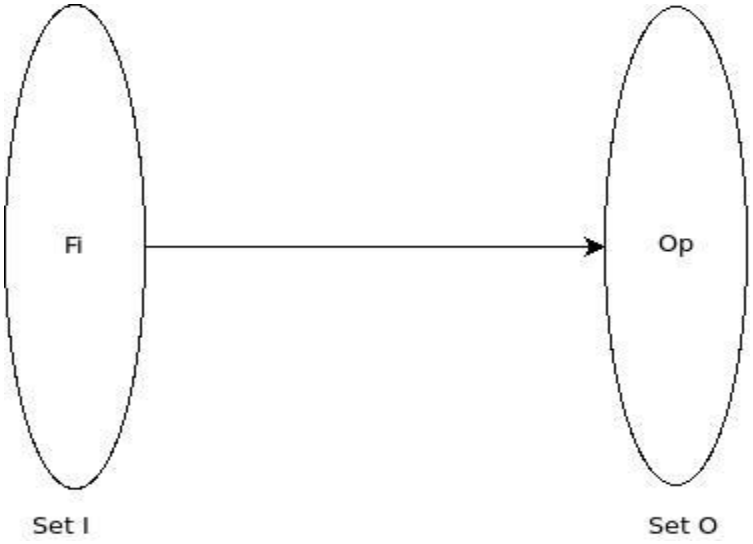
E is a End state.
E={Sc, Fc | Sc = success case which gives the lexemes-token table, symbol table entries and Fc = failure case which gives the lexical errors}.

I= set of inputs
I={Fi | Fi is an input file which consists of High Level Language code}.

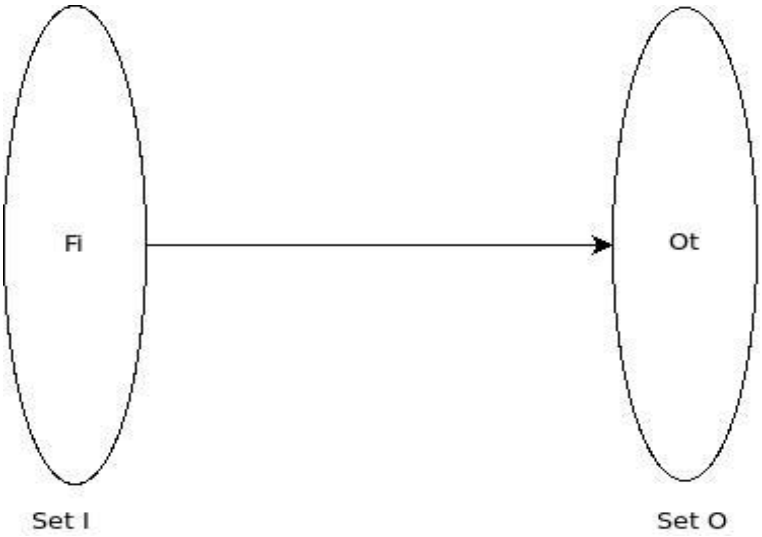
O= set of outputs
O={Op, Ot, Os, | Op is program after preprocessing i.e. after removing comments, white spaces, Ot is lexeme-token table, Os is symbol table}.

F=A set of Functions.

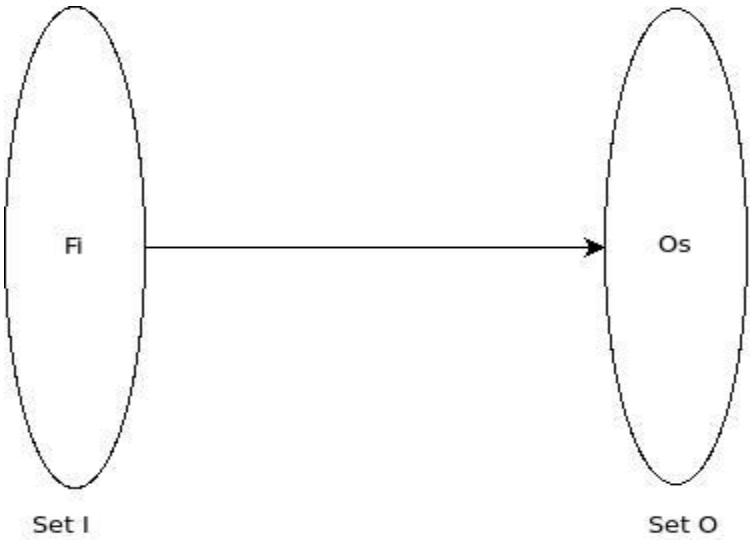
F={f1,f2,f3 | f1,f2,f3:I → O, function f1 removes comments and white spaces from input program, function f2 generates the lexeme-token table from input program, function f3 generates the symbol table from input program}




Function f1: I → O




Function f2: I → O



Function f3: I → O

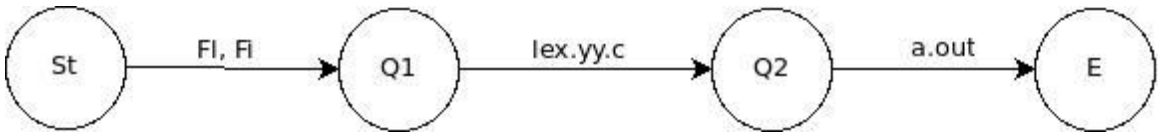
DD=Deterministic Data

DD={x | x can be a token, data type, attribute}

NDD= Non-Deterministic Data

NDD={y | y can be a syntax or semantics of the input language}

**State Transition Diagram:**




**Algorithm:**

1. Accept input filename (file is in HLL) as a command line argument.
2. Separate the tokens as identifiers, constants, keywords etc. and fill the generic symbol table.
3. Check for lexical errors and give error messages if needed.

**Test Input:**

```
#include<stdio.h>
void main()
{
 int a,b;
char bilateral;
a=3;
 c=a+d;
}
```

**Test Output:**

| **Lexeme** | **Token** |
|---|---|
| #include<stdio.h> | Directive |
| void | Reserved |
| main() | Reserved |
| { | Punctuation |
| int | Keyword |
| a | Identifier |
| b | Identifier |
| ; | Delimiter |
| char | Reserved |
| bilateral | **Lexical error** (as Identifier is more than 8 characters) |
| ; | Delimiter |
| a | Identifier |
| = | Operator |

| | |
|---|---|
| 3 | Constant |
| ; | Delimiter |
| c | Identifier |
| = | Operator |
| a | Identifier |
| + | Operator |
| d | Identifier |
| ; | Delimiter |
| } | Punctuation |

**Conclusion:**

Thus, We have successfully implemented Lexical analyzer for HLL using LEX