- Name : Anjishnu Mukherjee
- Registration Number : B05-511017020
- Class Roll Number : CS Gy-70
- Exam Roll Number : 510517086
- Email : 511017020.anjishnu@students.iiests.ac.in

Any regular expression can be expressed using an equivalent Finite Automata, and any Finite Automata can be represented using simple computer programs.

Lex uses this idea to translate regexes into programs that simulate a FA which uses the current state and next input character to determine the next state by referring to a computer-generated state table.

Since the grammar for regexes is Regular Grammar, there are some limitations to what lex can recognise. For example, nested structures like parentheses can't be recognised, as those would need a stack-based machine for parsing instead of just a simple Finite Automata.

Lex regexes are composed using the following set of rules.

Metacharacter	*Matches*
.	any character except newline
\n	newline
*****	zero or more copies of preceding expression
+	one or more copies of preceding expression
?	zero or one copy of preceeding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[ ]	character class

A metacharacter is a basic pattern, using which we can define more complex patterns for the regexes to match different types of strings in the input.

The scanner generated by lex, when executed will analyze its input looking for strings matching any of the defined patterns. Once a match is found, the text corresponding to the match (called the token) is made available in the global char* variable yytext and the length of the token is

stored in yyleng. Then the action corresponding to the token is executed. After that, the scanner continues on to analysing the rest of the input file.

- If it finds more than 1 match, the longest match is the one considered to be matching.
- If it finds 2 or more matches of the same length, the rule listed first in the lex file is chosen. (Thus, the order in which rules are written in the file matters)
- If it finds no match, then by default, the next character of input is considered matched and copied to output. (This is why, the simplest possible lex input file contains only %%, which basically generates a scanner that copies the entire input to the output.)

A lex input file has 3 sections :

Definitions -> Format : <name> <definition>, optional
%%
Rules -> Format : <pattern> <action>
%%
User code -> Optional. Consists of the main() function, which calls the yylex() function.

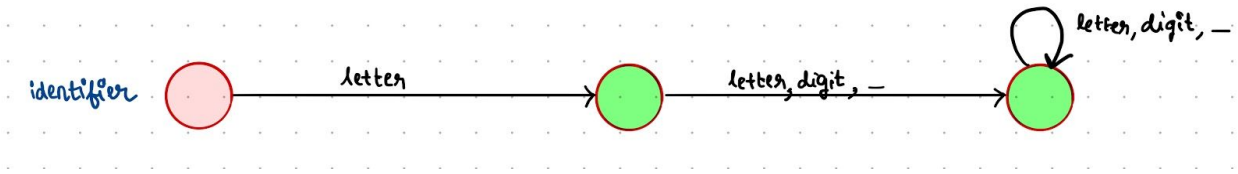
For our analyser, the **<action>** that we are using is to **print (token, token ID, token length)** for all valid tokens and to print an error message along with the line number for all invalid tokens.

We will be skipping through preprocessor statements, single-line comments, multi-line comments and empty lines / white spaces. For all other tokens, we will check if it is 1 of the 32 valid keyword tokens, or a character or string, or integer or real number, or single symbols and then return a corresponding token ID which we define separately in a header file (that is included in the definitions section of the lex file). If it is not one of the above, we check if it matches the pattern of a defined invalid identifier (starting with real number and followed by an identifier), and if so, we skip that token and print an error message with the line number. If the scanned token doesn't match any of the above, then it is an unknown token, which is neither valid nor matches the pattern for invalid which we defined. For such a token, we would also print an error message and the line number. To keep track of the line number throughout this process, we use a global variable that is incremented by 1 every time we encounter the symbol \n in the pattern. Do note that this won't be an issue if \n is present within a print statement, because we would be treating the contents of the string inside the print statement as an escape sequence. For our analyser, the following set of definitions are sufficient.

```
identifier [a-zA-Z][a-zA-Z0-9_]*
character \'(\\.|.)\'
string  \"(\\.|[^"\\])*\"
integer [0-9]*
real [0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
invalid ([0-9][0-9]*|[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?)[a-dA-Df-zF-Z_][a-zA-Z0-9_]*.*[\n]
single_symbol [\.\,\!\#\$\%\^\&\*\(\)\-\_\+\=\{\}\[\]\\\|\~\:\;\"\'\/\?\>\<]
whitespace [ \t]*
```

The reasoning behind each of the statements in these definitions will become apparent from the next sections.

Explanation for **identifier** :



This Finite Automata represents the rule written above, where the identifier has to start with a letter, but can have any number of letters or numbers or underscores after that.

Explanation for **character** :

A character consists of a single quote **‘** followed by either an escaped anything **\\.** or any non-escaped thing **.** and finally a terminating single quote **’**.

Putting it all together, we get **\‘(\\. | .)\’** . The delimiting single quotes are escaped because they are Flex meta-characters.

Explanation for **string** :

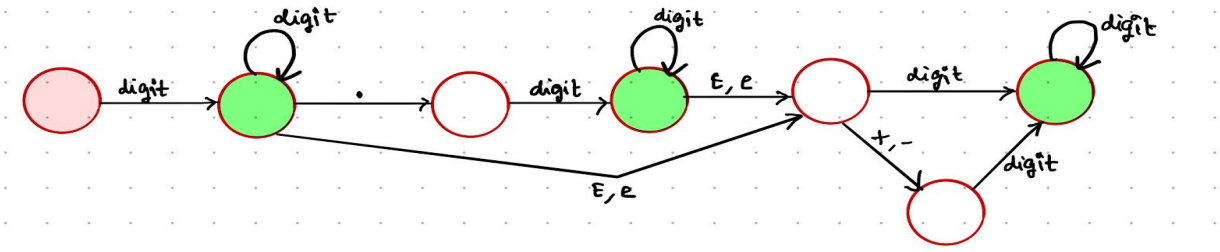
A string consists of a quote mark **"** followed by zero or more of either an escaped anything **\\.** or a non-quote character, non-backslash character **[^"\\]** and finally a terminating quote **"**.

Put it all together, and you've got **\"(\\.|[^"\\])*\"** . The delimiting quotes are escaped because they are Flex meta-characters.

Explanation for **integer** :

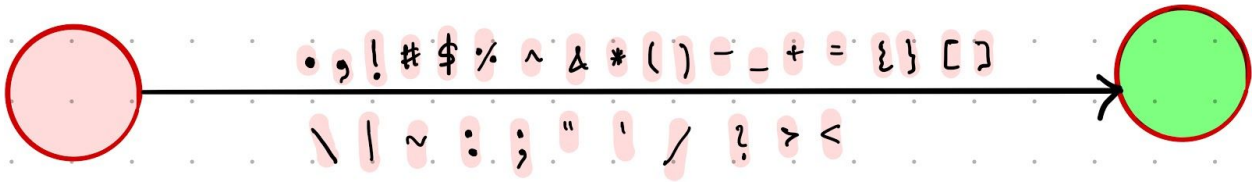
An integer can have any number of digits between 0 and 9 with no restrictions.

Explanation for **real** :



This finite automata considers all possible real numbers written using scientific notation. **[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?** is the regex corresponding to this.

Explanation for **single symbols** :



This finite automata will accept any of these single symbols.
[\.\,\!\#\$\%\^\&*\(\)\-_\+\=\{\}\[\]\\\|\~\:\;\"\'\/\?\>\<] is the regex corresponding to this.

Explanation for **whitespace** :

A whitespace is matched with **0 or more number of tabbed spaces [\t]*** .

Explanation for the rule to match **known invalid patterns** :

Starting with a digit and followed by an integer or a real number and then having an identifier. **([0-9][0-9]*|[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?)[a-dA-Df-zF-Z_][a-zA-Z0-9_]*.*[\n]** is the regex corresponding to this. For such examples, we skip that token and print an error message, before continuing parsing the rest of the line.

The **32 keywords** supported by C are :

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

We write the rule for keywords before the rule for identifiers, so that identifiers won't contain any keywords due to the top-down preference order of Lex. For each of the keywords, we define in a separate header file all the token IDs. Apart from that, we also define token IDs for all multi-character lexemes which C supports like "+=", "-=", "*=", "/=", "&=", "|=", "^=", "<<", ">>", "<<=", ">>=", "<=", ">=", "==", "::".

This document thus briefly describes some key ideas about lex and how we can use them for building a lexical analyser using the rules described above.

References :

- http://web.mit.edu/gnu/doc/html/flex_2.html
- A Compact Guide To Lex & Yacc by Thomas Niemann